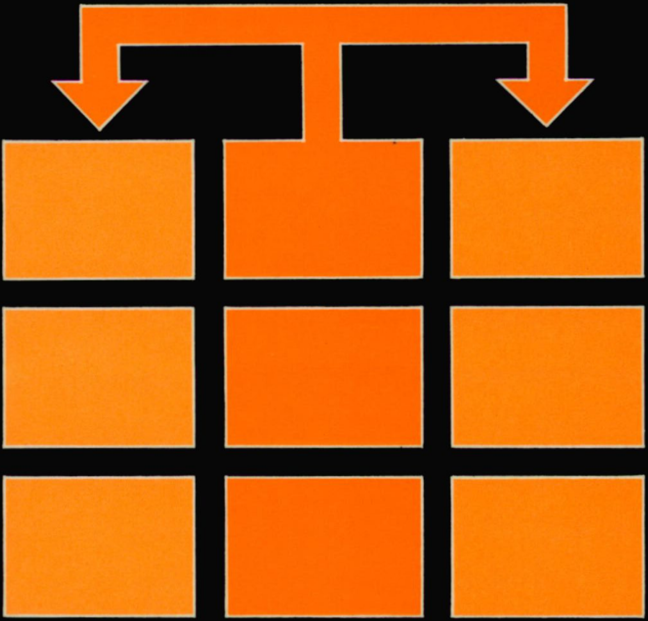
evised for
COMPACT 128

BRUCE SMITH

Advanced Sideways RAM User Guide

for the MASTER and BBC Computers



VICTORY PUBLISHING

BRUCE SMITH

Advanced Sideways RAM User Guide

for the MASTER and BBC Computers

Editor
Tony Quinn

VICTORY PUBLISHING

PO Box 19
London
N11 1DS

ISBN 0 948938 00 5

Printed by A Wheaton & Co Ltd Exeter

God bless all who sail in her

Contents

Preface to Second Edition

The contents of this book are fully applicable to the Master Compact The Compact has one new feature plus some changes which are discussed below There has been much feedback on the first edition from readers and friends for which I am most grateful Most of this has been incorporated, and those points which could not be dealt with in the text are discussed on page 229

Bruce Smith November 1986

Master Compact

The most significant improvement is the addition of an optional I (insert) parameter as part of the *SRLOAD and *SRWRITE commands If I is specified (with or without a Q parameter - see page 28), the ROM image will be automatically inserted into the ROM memory map and be available for immediate use Typical examples of use of the I parameter are

```
    *SRLOAD Rom 8000 7 I
    *SRWRITE 5000+500 8000 6 QI
```

If the I option is left off, CTRL-BREAK must be used to 'insert' the ROM

The Compact comes with 64k of ROM supplied, whereas the Master has 128k Hence *ROMS (page 20) shows that only sockets &F, &E and &D contain ROM titles

```
    &F UTILS 01
    &E Basic 40
    &D ADFS  10
```

The word 'RAM' will appear in sockets 4,5,6 and 7

The ROM slots occupy the lower right hand side of the printed circuit board inside the Compact keyboard These compare with the Master 128 as follows

Compact	Max size	Slot number	Master 128 position
IC38	32k	0,1	Cartridge (sk3)
IC23	16k	2	Cartridge (sk4)
IC17	16k	3	Cartridge (sk4)
IC29	16k	8	IC27

Appendix D link notes are not relevant to the Compact

Introduction
Using This Book

Sideways RAM is a philosophy central to the design of the Master series of microcomputers Four sideways memory areas are provided for you to load in ROM software images from disc This facility not only allows you to keep copies of ROM software on disc and avoid having to handle chips and cartridges, but also, and even more exciting, to develop your own software that can be accessed by * commands And that is what this book is really all about - providing you with the theory, backed up with tried and tested programs

Of course sideways RAM was not invented for the Master, its predecessor the BBC B+128 also has the sideways RAM available to use - and the other Acorn machines - BBC B, BBC B+ and Electron can also be fitted with plug-in sideways RAM boards which instantly open up this new world for you If you're looking for a RAM board then consult magazine reviews (typically that by Chris Drage in the May 1986 edition of Acorn User)

The practical approach of this book is emphasised by the fact that there are over 25 listings, the majority of which all form sideways RAM images which will give you extra * commands when they are loaded into a sideways RAM bank As a reference guide, even the most devote sideways RAM follower will find it invaluable and the full index will allow you to locate information quickly and simply

Although programs are written with the Master in mind, Appendix B contains full conversion details for the BBC B, BBC B+, BBC B+128 and Electron microcomputers

The ROM Formatter program presented in chapter 13 is worth the price of the book alone! It will enable you to format your favourite BASIC or machine code programs in

such a way that they to can be used as a RAM image and loaded in or run without the need to access disc or tape
 For those of you who suffer from tired fingers after the first three lines then a disc of the programs can be obtained - details at the rear of this book

Listings

All the listings have been tried and tested before being dumped to a printer The listings are written in BASIC 4 and will run on BASIC 2 with no correction - users of BASIC 1 will need to doctor the listings slightly but full details are provided In an effort to provide clarity a daisywheel printer has been used to produce the listings Note the difference between the following characters

number one	1
small letter 'el'	l
number zero	Ø
large letter 'oh'	O
hash	≠

All the listings are dumped with LISTO1, WIDTH4Ø

The future

Sideways RAM has proved itself to be among the most popular aspects of BBC computing - to continue this we at Victory Publishing with your co-operation hope to provide a regular newsletter on just this topic So let's hear your views, your ideas and see your programs

Acknowledgments

Many thanks to the following Linda Dhondy, Alex van Someren, Steve Mansfield, Derek Coombes, Kitty Milne (keep on Computing), and everyone on Acorn User

June 1986

Chapter One What is Sideways RAM?

Question What is sideways RAM?
Answer   I'm not going to tell you! (Yet)

Now don't get me wrong, I'm not trying to be difficult What I am going to do is first show you Master owners what sideways RAM is -- and just how useful it can be Then in the rest of this book I'll explain how you can use it to further your own needs, so that by the last page you'll feel confident in being able to approach the task of writing your own software in sideways format without too much trouble All you have to know for the moment is that the sideways RAM is ready and waiting in your Master and I'm going to prove it with a little demonstration

Type in listing 1 1 found at the end of the chapter It's just 5Ø lines long so shouldn't tax even the worst typists Leave out the first five lines if you want Once it has all been typed in, save the program to a disc or tape before running it - just in case there's an accident (If so you can load it back in) Use the filename 'DEMO' to save the program under, ie

```
    SAVE  DEMO'
```

Right, now for the moment of truth RUN the program If you get any errors, correct them and save the program again The most likely place for errors is in the DATA lines For example

```
    Out of data at line 14Ø
```

means you have missed out an item or items of DATA in lines 29Ø to 51Ø If you get the message

```
    Data error - please check
```

then you have made a typing error within the data which has been picked up by a checking routine in lines 19Ø-26Ø So look through it all again carefully If the message persists then get someone else to check it for you When everything is okay the usual cursor prompt will appear Remember, if you make any corrections to the program, re-save it each time

All OK? Then congratulations, you have written a sideways RAM program Simple wasn't it? Writing sideways RAM programs isn't normally as boring as entering in rows of numbers - I've just done it that way to make it easier for the time being The next thing to do is to initialise the sideways RAM program - in other words tell the micro it's there This is done by performing a CTRL-BREAK (To do this, hold down the CTRL key, press the Break key once and then release the CTRL key) The Master will display its normal start-up message as when you first turn on Now type

```
    *ROMS
```

A list similar to this should appear

```
    ROM F TERMINAL Ø1
    ROM E VIEW Ø4
    ROM D Acorn ADFS 5Ø
    ROM C BASIC Ø4
    ROM B Edit Ø1
    ROM A ViewSheet Ø2
    ROM 9 DFS Ø2
    ROM 8 ?
    ROM 7 ?
    ROM 6 ?
    ROM 5 ?
    ROM 4 Beep 1 Ø Ø1
    ROM 3 ?
    ROM 2 ?
    ROM 1 ?
    ROM Ø ?
```

This lists the software held in the Master The seven at the top of the list are standard and come with the micro when you buy it We are interested in number 4

```
    ROM 4 Beep 1 Ø Ø1
```

What you have done is to put a program called 'Beep' into an area of sideways RAM when you ran the DEMO program above 'Beep' is designed to act as if it was an actual ROM chip

The command *HELP lists the ROMs present in the machine Try it now

```
    *HELP
```

The result will look something like this

```
    OS 3 2Ø
      MOS

    TERMINAL 1 2Ø

    VIEW B3 Ø

    EDIT 4

    ViewSheet B1 Ø

    SRAM 1 ØØ

    Beep 1 Ø
```

So Beep 1 Ø has been added to the *HELP list Now type

```
    NEW
```

followed by the following short program

```
    1Ø FOR N%=1 TO 2Ø
    2Ø *BEEP
    3Ø NEXT N%
```

RUN this, and you'll get a continuous tone from the speaker of your micro When the din stops try typing in *BEEP This time you'll get a single beep

What the original program has done is to add a new command to the Master's vocabulary - *BEEP If you don't believe me turn the Master off for a few seconds, then switch it back on and go through the *ROMS, *HELP procedure and see where it gets you!

Sideways ROMs or RAMs

The terms sideways RAM and sideways ROM often go hand in hand - but what is the difference between them?

Well, in short, sideways RAM is volatile (ie it can change) while sideways ROM is non-volatile (it can't)

RAM stands for Random Access Memory - you can read its contents and you can also change them It is a volatile medium in that its contents are only preserved while it has power supplied to it As soon as the power source is removed the contents are lost They can be restored by switching on the power and loading them back in This was exemplified with the *BEEP example The DEMO program wrote the code for this into the correct area of RAM The *BEEP command was available to us all the time the power was switched on As soon as the power was removed, ie by switching the micro off, the code for *BEEP was lost It can be placed back into the Master by loading and running the program again

ROM stands for Read Only Memory As its name suggests this memory can only be read from - it cannot be written to ROMs are an example of a non-volatile memory, their contents are not affected by power being present or not The Master is fitted with a single ROM chip when you buy it - the 'MegaROM' as it is called in Acorn's user guides This contains all the machine code that is required to run your micro - BASIC, the MOS, View, Viewsheet, DFS, ADFS, Edit and Terminal

Sideways RAM and sideways ROM both have their pros and cons The obvious advantage of sideways ROMs is that they are always present within the machine - ready for instant use as soon as you switch the Master on The disadvantage is that if you wish to add a new or extra sideways ROM you need to take the lid of the micro and physically fit it into one of the ROM sockets, or use a cartridge Sideways RAM does not suffer from this disadvantage because you can just load it in from disc or tape Its other big advantage is that it allows you to write and develop customised software - a rewarding and possibly profitable hobby!

When a program is written to work in sideways RAM, it is converted into a 'ROM image' This can be loaded into a sideways RAM bank or saved to disc for future loading Once the ROM image is in sideways RAM we can for all intents and purposes use it as if it were a ROM, although strictly speaking it's not This interchange of terms is quite common in books and magazines, not to mention the Master itself, so don't be put off - it's simply quicker and easier to say!

Why 'Sideways'?

The nagging question you may have at this moment is what is the relevance of the term 'sideways'? To answer this we need to understand something about the 65C12 microprocessor at the heart of the Master The amount of memory that any microprocessor can actually address (ie write to or read from) at any one time depends on the number of 'addressing lines' it has The address lines, collectively called the address bus, are the wires that radiate from the 65C12 chip These lines can have one of two states - either on or off The two states can be indicated by the numeric values 1 and 0 By switching on combinations of address lines we can build up patterns of 1's and 0's You may already realise that this forms what is termed a binary number In computing however, rather than talk in strings of ones and zeroes we convert to a special number system called hexadecimal, based on 16 rather than 10 You may already be familiar with this term - if not then take a look at Appendix A and the Glossary before going any further

The 65C12 chip has sixteen address lines, so the maximum address of a byte of memory is when all the lines are 'on' In binary terms this is represented as 11111111 11111111, which is &FFFF or 65535 decimal Therefore there are 65536 addressable locations within the Master (65536 because the first is at location 0) If we convert this figure into kilobytes by dividing through 1024 we arrive at 64k Figure 1 1 shows how this is traditionally arranged The first 32k is given over as RAM, the top 32k contains first the 16k BASIC language and above this the 16k MOS (machine operating system) But think about what your Master contains It has 128k of RAM and 128k of ROM in its standard configuration - that's 256k in all, yet we have just determined that the maximum addressable memory of the

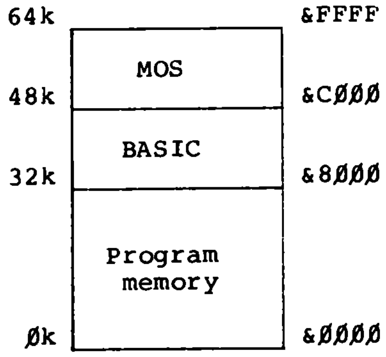


Figure 1 1 The traditional memory map

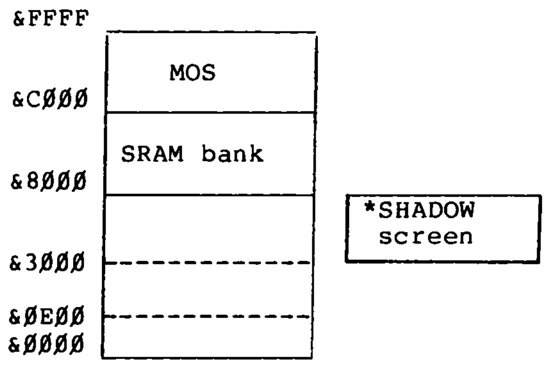


Figure 1 2 Software can be read into SRAM bank Memory for shadow screen sits alongside main block

BASIC	Term	View	ADFS	Edit	View Sheet	DFS

Figure 1 3 Sideways software can be 'paged' into RAM bank

Master is a mere quarter of this! The answer lies in having the extra memory present but 'overlaying' it into the main memory map as and when it is requested For example you can gain extra programming memory in that first 32k bank of RAM by selecting shadow memory with the command *SHADOW When this command is executed the Master intercepts any subsequent commands to do with the screen and redirects it away from program memory to the shadow RAM area In other words, the contents are put to one side (figures 1 2 and 1 3) That last word provides the clue to how the term sideways arose

The ROMs supplied in the MegaROM, with the exception of the MOS, all occupy the same section or bank of memory That is the 16k from &8000 to &BFFF So in figures 1 2 and 1 3 they appear to stretch sideways across the memory map When you use a particular command, the MOS locates which ROM that command belongs to and allows that ROM to take control of the 16k of memory normally occupied by the BASIC language This technique is often called 'paging' and gives rise to the term 'paged ROM'

Sideways RAM is simply a bank of RAM that is treated in exactly the same way - in fact the MOS cannot tell the difference between sideways RAM and sideways ROM -

only we can! A total of 64k of the Master's RAM is capable of being paged into this sideways slot As each slot is 16k, that gives us a total of four sideways RAM banks, ie 4*16k=64k

You may be wondering why only a 16k section of RAM was provided as the slot for sideways RAM, and why the whole of the 32k of memory above &8000 is not used The answer is straightforward The Master cannot operate without the MOS present, because this handles all the donkey work such as writing to the screen, reading the keyboard and moving to and from other ROMs All that sideways RAM, or ROM programs ever need to do to perform such a task is to call the appropriate routine within the MOS In fact in many instances it is done automatically by the MOS without you ever knowing - reading and displaying key presses for example

Getting your priorities right!

As we saw in the list of ROMs earlier, each sideways ROM or RAM slot is given a number In fact the Master has the capability to address 16 ROM or RAM sockets and these are numbered from 0 to 15 To keep in line with normal computing tradition these sockets are normally referred to by their hexadecimal equivalents, ie &0 to &F Of course I use the term socket for descriptive purposes There are not 16 sockets physically inside the case of the Master, but they are in theory

The MegaROM containing the standard software occupies seven 'sockets', &9 to &F The four banks of sideways RAM are overlaid into positions 4, 5, 6 and 7 Make a mental note of these now as we will need to refer to these positions constantly when writing sideways RAM programs

One final point is that the sideways RAM memory is taken over when you use BAS128, the special version of BASIC supplied on the Welcome disc to give you access to 64k of RAM for programs Obviously, you cannot use BAS128 and sideways RAM at the same time

Summary

So, sideways RAM is a special area of memory into which we can place programs These programs are written to special, but easy to follow rules which will be explained in the next chapter Once in sideways RAM, a program will behave as if it were in a ROM chip

As we have seen we can extend the Master's vocabulary so it can carry out special tasks for us at the whim of typing in a chosen command *BEEP was not exactly earth-shattering stuff, but no matter how simple or complex a command you decide to write, it will still need to be implemented in the same way Incidentally, the two-digit number given alongside the ROM name when you type *HELP is the version number There'll be more on this in the next chapter

The tutorial approach of this chapter using simple programs as examples will be followed throughout this book The end result of some of the examples may be mundane, but it's getting there and seeing how things are done that counts Therefore the implementation will be covered in depth while the example will be kept as simple as possible to avoid confusion The programs are not written for efficiency or to optimise speed and space When you have mastered the basics then tacking on your own routines will be a minor problem

Listing 1 1 Sideways RAM demonstration Save as DEMO

```
   10 REM A simple demo
   20 REM DEMO
   30 REM (C) Bruce Smith June 1986
   40 REM Advanced SRAM  Guide
   50
   60 PROCread
   70 PROCchecksum
   80 *SRWRITE 4000 +72 8000 4
   90 END
  100
  110 DEF PROCread
  120 base=&4000
  130 FOR loop=0 TO 113
  140 READ data
  150 base?loop=data
  160 NEXT loop
  170 ENDPROC
  180
  190 DEF PROCchecksum
  200 N%=0
  210 FOR loop=0 TO 113
  220 N%=N%+base?loop
  230 NEXT
  240 IF N%=13477 THEN ENDPROC
  250 VDU 7
  260 PRINT  Data error - please check
  270 STOP
  280
  290 DATA 0,0,0,76,23
  300 DATA 128,130,18,1,66
  310 DATA 101,101,112,32,49
  320 DATA 46,48,0,0,40
  330 DATA 67,41,0,72,201
  340 DATA 9,240,6,201,4
  350 DATA 240,33,104,96,152
  360 DATA 72,138,72,32,47
  370 DATA 128,104,170,104,168
  380 DATA 104,96,32,231,255
  390 DATA 162,255,232,189,9
  400 DATA 128,32,227,255,208
  410 DATA 247,32,231,255,96
  420 DATA 152,72,138,72,162
  430 DATA 255,136,232,200,177
  440 DATA 242,41,223,221,109
  450 DATA 128,240,245,189,109
  460 DATA 128,16,15,201,255
  470 DATA 208,11,169,7,32
  480 DATA 227,255,104,104,104
```

Listing 1.1 continued

```
  490 DATA 169,0,96,104,170
  500 DATA 104,168,104,96,66
  510 DATA 69,69,80,255
```

Chapter Two
The Sideways Header

I mentioned in the first chapter that programs placed in sideways RAM or ROM must conform to a specific format for them to work correctly In this chapter we'll look at the arrangement of this format, in particular the couple of dozen bytes generally referred to as the 'ROM header' We'll be using a few terms that may well be new to you so I'll explain each one as we go along The first six bytes of the header, from &8ØØØ to &8ØØ5, contain two entry points into the sideways ROM These are two sets of machine code instructions arranged like this

```
    &8ØØØ   JMP xxxx
    &8ØØ3   JMP yyyy
```

where xxxx and yyyy are hex addresses

The first three-byte address is the language entry point and the second the service entry point With one exception all ROMs have service entry points, but not all ROMs have a language entry point

The service entry point causes a jump to a piece of machine code designed to handle requests for information given by the MOS For example, when you type in *HELP, the MOS asks each ROM in turn to print out its individual *HELP message Similarly, if the MOS finds a command it does not know beginning with an asterisk (eg *BEEP), it asks each ROM in turn if it recognises the command The one exception to this rule is the BASIC ROM, which the MOS recognises by its lack of a service entry point The only ROMs that have language entry points are, surprise, surprise, language

ROMs View, Viewsheet, and Edit are examples of language ROMs, as well as more traditional names such as Forth, Pascal, etc This language entry point provides the means for the sideways ROM to take control of the Master as we shall see later on If a ROM is not a language then it should leave these first three bytes of the ROM image alone, setting them to zero
Table 2 1 provides a brief list of the bytes at the front of the ROM header which have a specific function

Byte	Function
8000	Language entry point
8003	Service entry point
8006	ROM type (language or service)
8007	Copyright offset pointer
8008	Binary version number
8009	ASCII title string (terminator byte &00)
8xxx	ASCII version string (terminator byte &00)
8yyy	ASCII Copyright string (terminator byte &00)
8zzz	Tube relocation address

Table 2 1 Format of the ROM header

The first 10 bytes of the ROM header (8000-8009) are fixed and may always be found at a specific address The bytes after the ASCII string title, though important, may be of variable length These bytes consist mainly of ASCII character strings that define the ROM title and copyright labels Each of these ASCII character strings ends with a zero (terminator byte)
While the language and service entry points into a sideways ROM are obviously important to the functioning of the ROM, the information in the bytes following is of equal importance The copyright string, and in particular the C itself within the brackets, (C), is most important as without it the ROM will not be recognised as one! The byte at &8007 is the 'copyright string offset pointer' which contains the number of bytes (called the 'offset') from the start of the ROM to the &00 byte immediately prior to the copyright string (see figure 2 1)
When switched off, or after a CTRL-BREAK, the MOS extracts the value of the offset and uses it as an index to test for the presence of a copyright string If there is one then the MOS is sure that a sideways

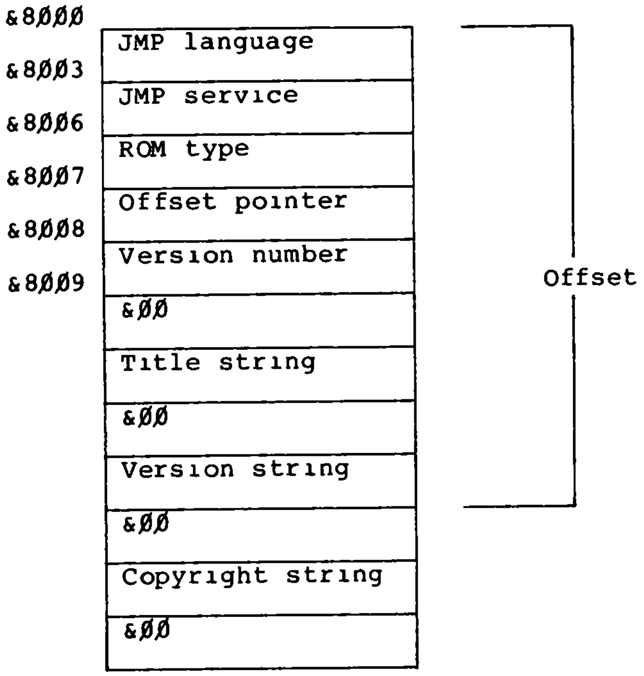


Figure 2 1 Calculating the offset

ROM is present in that particular bank The next stage in the initialisation process is to build a 'ROM type table' by extracting the type byte from each ROM and storing it in table form in memory for reference (The ROM type table in the Master's MOS, version 3 0, is located from the 16 bytes starting at &2A1) The address of the table can be read for other operating systems using OSBYTE 170 Entering and running listing 1 2 will do this for you

```
10 REM Read ROM table address
20 A%=170
30 X%=0
40 Y%=255
50 addr%=(USR(&FFF4)) AND &FFFF00
60 addr%=addr% DIV 256
70 PRINT ~addr%
```

Listing 2 1 Reads ROM table Save as TABLE

Table 2 2 gives the address where the type number for each ROM is stored Reading the address will tell you what type of ROM software is in a particular sideways ROM/RAM bank

ROM bank	Address
Ø	&2A1
1	&2A2
2	&2A3
3	&2A4
4	&2A5
5	&2A6
6	&2A7
7	&2A8
8	&2A9
9	&2AA
1Ø	&2AB
11	&2AC
12	&2AD
13	&2AE
14	&2AF
15	&2BØ

Table 2 2 Address associated with each ROM bank

If any ROM banks are found to be 'empty' then a zero is placed in the relevant table byte Thus if ROM 6 were empty a zero byte would be placed at location &2A7 If a sideways ROM position contains a ROM, its 'type byte' is read and placed in the byte position in the type table

ROM Types

The byte at &8ØØ6 contains the ROM type (table 2 1) which gives the MOS information about the ROM Each bit in the byte conveys the following information
Bit 7 Set if the ROM has a service entry, therefore it must always be set as all ROMs must have a service entry point The ONLY exception to this rule is BASIC
Bit 6 Set if the ROM has a language entry point
Bit 5 Set if the ROM has a second processor relocation address, for example if the ROM contains a 'Hi' version of a language For this to happen the code in the ROM, bar the service entry coding, must have been assembled for second processor addressing in mind The service call coding is not copied across the Tube interface to the second processor, and only languages may be copied across the Tube
Bit 4 This bit is used by ROMs operating on the Electron only (If set it controls the use of soft key

expansion allowing the Electron to implement function key operations using the CTRL and SHIFT sequences, as these are not normally available)

```
  Bit 3   If set, ROM contains Z8Ø code
  Bit 2   If set, indicates other processor code
  Bit 1   Must always be set (only exception is BASIC)
  Bit Ø   Set to zero, but may be used as bit 2
```

The following short program will produce a list of the ROM types in your machine

```
    1Ø REM List ROM types
    2Ø FOR N%=&2A1 TO &2BØ
    3Ø PRINT ~N%   =   ~?N%
    4Ø NEXT N%
```

On a standard Master 128 the list produced will look like this (I have added the ROM names)

```
    2A1 = Ø
    2A2 = Ø
    2A3 = Ø
    2A4 = Ø
    2A5 = Ø
    2A6 = Ø
    2A7 = Ø
    2A8 = Ø
    2A9 = Ø
    2AA = 82    -   DFS
    2AB = C2    -   Viewsheet
    2AC = C2    -   Edit
    2AD = 6Ø    -   Basic
    2AE = 82    -   Acorn ADFS
    2AF = C2    -   View
    2BØ = C2    -   Terminal
```

The above list shows that the most common ROM type numbers are &C2 and &82 Let's look at these in detail

```
    &C2  = 11ØØ ØØ1Ø
           ||      |______ bit 1  always set
           ||_____________ bit 6  set for language entry
           |______________ bit 7  set for service entry
```

We can see the ROM has language and service entry points ROMs with this type number include View and Viewsheet The other main ROM type is

```
    &82 = 1ØØØ ØØ1Ø
```

showing that this ROM has only a service entry ROMs with this type number include the DFS and ADFS The BASIC ROM has the type number

```
&6Ø = ØIIØ ØØØØ
```

This indicates it has a language entry point and a Tube relocation address As already mentioned BASIC has no service entry point In addition bit I is also clear, which must normally be set

Copyright Offset

This byte at &8ØØ7 gives the number of bytes (the offset) from the start of the ROM to the &ØØ terminator byte immediately before the copyright string As described earlier the copyright string is used to identify a sideways ROM The following lines of code could be used to test for the presence of a ROM (the variable 'vector' is a zero page byte address vector containing &8ØØØ)

```
LDY ≠7              \ offset at +7
LDA (vector),Y      \ get offset
TAY                 \ move into Y register
INY                 \ add one
INY                 \ add one
LDA (vector),Y      \ get byte
CMP ≠ASC('C")       \ is it 'C' from (C)?
BNE norom           \ if no, there's no ROM!
```

Of course it is possible to pick up ASCII C as garbage from an empty bank (so says Murphy's law), so it is worth testing the bytes either side of the C to ensure that they are equal to the ASCII values for "(and ')"

IMPORTANT A capital C must be used for the copyright indicator - a lower case c will not be recognised as such!

Version Number

The version number is not used by the MOS at all It is simply a byte provided for you to keep track of software development The eight bit value should relate to the version number of the software herein Thus if the software was version 5 the byte here could contain &Ø5 This is the number displayed after the ROM name when *ROMS is used

Title String

This is an ASCII string starting from &8009 and terminated by a zero byte If the ROM is a language then the MOS prints this string on the screen when the ROM is initialised This string is also normally the one printed out when *HELP is performed

Version String

This ASCII string is optional It allows the user to print the version number of the ROM during the processing of *HELP This string must be terminated by a zero byte, &00 If the ROM is a language then on entry to it the error pointer vector at &FD and &FE will be made to point to the version number if it is present If the version string is not present the error pointer will go to the copyright string

Tube Relocation Address

If bit 5 of the ROM type byte is set then the MOS expects to find a Tube relocation address at this point This is the address to which the ROM contents, which will be a language, will be copied The code must therefore be written with the second processor relocation address borne in mind The service coding should not though, and should assemble as normal This is because the service code is not copied across the Tube, as discussed in Chapter 14

Standard Header

Writing all the above information into a BASIC or assembler program is not difficult Program 2 1 shows just how simple it is Type it in and save it under the name 'HEADER' RUN the program, and reply as you wish to the two questions Here's how I fared

```
    Enter ROM title  Tester
    Enter copyright string  Bruce Smith
    Assembling header
    8000
    8000
    >
```

The program first asks you to enter the ROM title string, I used 'Tester', followed by the copyright string, my name Note that I did not enter the (C) as this has already been incorporated into the program

The message 'Assembling header' is printed The two numbers 8ØØØ show assembly of the code is underway When the prompt returns the ROM header is in position Type *ROMS to check it has happened My results were

```
ROM F TERMINAL Ø1
ROM E VIEW Ø4
ROM D Acorn ADFS 5Ø
ROM C BASIC Ø4
ROM B Edit Ø1
ROM A ViewSheet Ø2
ROM 9 DFS Ø2
ROM 8 ?
ROM 7 Tester Ø1
ROM 6 ?
ROM 5 ?
ROM 4 ?
ROM 3 ?
ROM 2 ?
ROM 1 ?
ROM Ø ?
```

Sure enough, ROM 7 contains the Tester title Let's look at the assembly program in detail now to see what happened and how

Lines 16Ø to 18Ø We cannot assemble our machine code header directly into sideways RAM So we need to assemble it elsewhere in memory, but make the assembler think it's going into memory from &8ØØØ This can be done using OPTs in the range 4 to 6 (or 7) and using the O% pointer in conjunction with P% (If you are not too familiar with this aspect of the assembler, then refer to Appendix B) The pointer O% points to &5ØØØ which is where the machine code for the header is actually assembled

Lines 19Ø to 2ØØ Assemble the language entry point As there is no language entry, three zero bytes are inserted using EQUB and EQUW

Line 21Ø Assembles the service entry point The code is simply a direct JMP to the start of the service polling code This is marked by the label 'service' (line 32Ø)

Line 22Ø Assembles the ROM type code The byte assembled is the standard one for a service only ROM, ie &82 Refer to text above for details

Line 23Ø Assembles the copyright offset pointer Uses the MOD function to calculate the offset, which is the remainder of the division by 256 (line 28Ø)

Line 24Ø Assembles the binary version number of the ROM coding, 1 in this case

Line 25Ø Label marking start of the ROM title string
Lines 26Ø to 27Ø Assemble the ASCII title string terminated by a zero byte
Line 28Ø Label marking 'offset'
Lines 29Ø to 31Ø Assemble the copyright string, prefixed and terminated by a zero byte
Line 32Ø Label marking the start of the service entry code
Line 33Ø The service entry coding here consists simply of a return Any service calls to this ROM will be returned without effect

Sideways Writing

Once the code has been assembled, it needs to be transferred across into a sideways RAM bank This is done in line 9Ø by the command *SRWRITE This command requires four items of information to work on, thus

```
*SRWRITE(<start addr>+<length>)<relocated addr> <ROM id>
```

Let's examine each parameter in turn

<start address>	this is the start address of the assembled code in memory
+<length>	length, in bytes, of the assembled code
<start address>	start address to which code is to be relocated in sideways RAM
<ROM id>	number of the sideways RAM bank into which code is to be copied, ie, 4, 5, 6 or 7

The two start addresses are to be in hexadecimal Line 9Ø in program 2 1 is as follows

```
*SRWRITE 5ØØØ +2ØØ 8ØØØ 7
```

This shows that the start address of the assembled code is &5ØØØ, its length is &2ØØ bytes, and it is to be relocated at &8ØØØ in sideways RAM bank number 7 The length value can be substituted by the address +1 of the end of the code if so desired For example, if the code started at &5ØØØ and ended at &6123, we could use

```
*SRWRITE 5ØØØ 6124 8ØØØ 7
```

The absence of the + sign before the second value informs the MOS that the number following is an address and not an offset Note that the second number is 1 greater that the actual end address of the code and must follow on from the first value

There are several more sideways RAM utility commands supplied with the Master and these will be examined in the next chapter

Listing 2 1 Reads ROM table address Save as TABLE

```
   10 REM Read ROM table address
   20 A%=170
   30 X%=0
   40 Y%=255
   50 addr%=(USR(&FFF4)) AND &FFFF00
   60 addr%=addr% DIV 256
   70 PRINT ~addr%
```

Listing 2 2 Produces standard ROM header Save as HEADER

```
   10 REM Form ROM header
   20 REM (C) Bruce Smith June 1986
   30 REM The Advanced SRAM Guide
   40 REM
   50 MODE 7
   60 PROCgetstrings
   70 PRINT''"Assembling header"''
   80 PROCassemble
   90 *SRWRITE  5000 +200 8000 7
  100 END
  110
  120 DEF PROCassemble
  130 osnewl=&FFE7
  140 osasci=&FFE3
  150 FOR pass=4 TO 6 STEP 2
  160 P%=&8000   O%=&5000
  170 [
  180 OPT pass
  190 EQUB 0
  200 EQUW 0
  210 JMP service
  220 EQUB &82
  230 EQUB offset MOD 256
  240 EQUB 1
  250  title
  260 EQUS title$
  270 EQUB 0
  280  offset
  290 EQUB 0
  300 EQUS "(C) "+copy$
  310 EQUB 0
  320  service
  330 RTS
  340 ]
  350 NEXT
  360 ENDPROC
  370
```

Listing 2 2 continued

```
38Ø DEF PROCgetstrings
39Ø PRINT 'Enter ROM title  ',
4ØØ INPUT   "title$
41Ø PRINT "Enter copyright string    ,
42Ø INPUT " "copy$
43Ø ENDPROC
```

Chapter Three
Service ROMs

As we have seen, all sideways ROMs, except BASIC, must have a service entry point The machine code here will depend on the sophistication of the software, but the ROM must be capable of identifying all the calls it needs to function correctly Service calls that are of no importance to it can be ignored by returning to the MOS with an RTS instruction The coding to look after service calls must include an 'intepreter' capable of recognising individual commands, and acting on them

In all there are 31 possible service calls, though most will not require processing by service-only ROMs

When a service call is made, the highest priority ROM bank is polled first (ROM &F) and the call is then passed down through the ROMs until one recognises the call and acts on it When a service entry is required, the three processor registers are used to pass the service call details as shown in table 3 1

Register		Service call information
Accumulator	-	Service type
X register	-	Number of current ROM
Y register	-	Any extra service parameter

Table 3 1 Service call register initialisation

If the service call is not recognised by the current ROM the service coding must restore all register values and return using RTS In most instances it will be the MOS issuing the service call, but other ROMs may issue a service call using an OSBYTE call as follows

```
    LDA ≠&8F        \ issue ROM service call
    LDX ≠type       \ X contains service type requested
    LDY ≠param      \ Y contains any parameter
    JSR OSBYTE      \ execute
```

On return from the OSBYTE call, the Y register will contain any resultant value so this should be checked as required

Table 3 2 lists all the service call types It is

Call	Type
Ø (&ØØ)	Call already provided
1 (&Ø1)	Claim absolute workspace in normal RAM
2 (&Ø2)	Claim private workspace in normal RAM
3 (&Ø3)	ROM auto boot
4 (&Ø4)	Command not recognised
5 (&Ø5)	Interrupt not recognised
6 (&Ø6)	BRK
7 (&Ø7)	OSBYTE not recognised
8 (&Ø8)	OSWORD not recognised
9 (&Ø9)	*HELP
1Ø (&ØA)	Claim static workspace in normal RAM
11 (&ØB)	Release NMI
12 (&ØC)	Claim NMI
13 (&ØD)	Initialise ROM filing system
14 (&ØE)	Return byte from ROM filing system
15 (&ØF)	Vectors claimed
16 (&1Ø)	EXEC/SPOOL files about to close
17 (&11)	Character set about to explode/implode
21 (&15)	Polling interrupt
24 (&18)	Interactive *HELP
33 (&21)	Indicate static RAM in hidden RAM
34 (&22)	Claim private workspace in hidden RAM
35 (&23)	Tell top of static workspace in hidden RAM
36 (&24)	Private workspace requirements
37 (&25)	Inform MOS of filing system name and information
38 (&26)	Close all files
39 (&27)	Reset has occurred
4Ø (&28)	Unknown *CONFIGURE option
41 (&29)	Unknown *STATUS option
42 (&2A)	ROM based language starting up
254 (&FE)	Secondary Tube initialisation
255 (&FF)	Main Tube initialisation

Table 3 2 Service call types

not necessary to memorise or understand these at present as each will be explained later on as needed In many instances full working examples will also be provided The table is purely for reference
The service calls are not all dished out one by one by the MOS In fact on a hard reset only 11 calls are issued by the MOS The others are issued as and when they are needed
  Listing 3 1 will allow you to see what service calls are issued by the MOS and when The service entry coding simply contains a short routine that will print out the current service call number before returning control back to the MOS As all service calls are issued to every ROM, every service call number issued will be printed When you have typed in and saved the program enter the following three lines directly at the keyboard to check the accuracy of what you have typed

```
    N%=Ø
    FOR X%=&5ØØØ TO &5Ø4B N%=N%+?X% NEXT
    PRINT N%
```

The value printed should be 6525 if not then recheck your listing carefully Once the checksum value is correct save the program under the filename "TRACE" Once saved, simply RUN the program Then, as the program does not contain a transfer routine, type this in at the keyboard

```
    *SRWRITE 5ØØØ+1ØØ 8ØØØ 7
```

Typing *ROMS should show that the code is installed safely in position
  To initialise our Trace ROM we need to perform a 'hard reset' by doing a CTRL-BREAK Once you have done this, the Master should re-initialise itself and the normal start-up messages will be preceded by some hexadecimal numbers as follows

```
    ØF 24 21 22 Ø1 Ø2 23 25 FE
    Acorn MOS

    27 Acorn 177Ø DFS

    ØF BASIC

    >
```

These hex numbers are the service calls issued by the MOS during the hard reset, and there are 11 of them

Compare them to table 3 2 above to see what is happening

Now try pressing just the BREAK key, and the screen will clear as follows

```
25 FE
Acorn MOS

27 ØF BASIC

>
```

This time just four calls are issued by the MOS, clearly showing the major differences between a 'hard' and 'soft' break, or reset

Assuming you're using a disc system, type

```
*CAT
```

Before the disc catalogue appears you'll see that service calls ØC and ØB are issued

Typing *HELP shows that requests Ø9 and 18 are issued

Play around for yourself to see what calls are issued when To remove the Trace ROM coding you'll need to switch the Master off

SRAM Utilities Explained

When you type *HELP the last message you see is

```
SRAM 1 ØØ
```

These are the sideways RAM utilities, and typing

```
*HELP SRAM
```

provides the following list

```
SRDATA <id >
SRLOAD <filename> <sram address> (<id >) (Q)
SRREAD <dest  start> <dest  end> <sram start>
 (<id >)
SRROM <id >
SRSAVE <filename> <sram start> <sram end> (<id >)
 (Q)
SRWRITE <source start> <source end> <sram start>
 (<id >)
End addresses may be replaced by +<length>
```

SRWRITE has been explained, and SRDATA and SRROM are not directly applicable, so let's look at the others

*SRLOAD

This command is similar in action to *SRWRITE with which we are familiar However, instead of writing a block of memory into sideways RAM, it writes a file from disc, tape or whatever filing system is in use Let's try an example First load program 3 1, TRACE, and RUN this to assemble the machine code Next we must *SAVE this block of code in the normal manner

```
     *SAVE R TRACE  5ØØØ +1ØØ
```

The prefix R reminds us it is a ROM image If your filing system will not readily accept this name then choose something suitable (eg RTRACE on networks)
  To use *SRLOAD we need three items of information the filename, the memory address it is to be loaded to and the RAM bank in which it is to be placed

```
     *SRLOAD  R TRACE  8ØØØ  7
```

We can specify an extra item of information, a Q

```
     *SRLOAD  R TRACE  8ØØØ  7  Q
```

When the Q is seen by the routine it will perform a Quick, that is fast, transfer It will load the file into memory at PAGE and copy it straight across into the RAM bank The advantage of this method is its speed The disadvantage is that it will overwrite any program or data in memory If you wish to keep memory contents intact then use the former method and omit the Q In such cases the routine saves the memory area from corruption However it is considerably slower Try both methods to see for yourself Don't forget to CTRL-BREAK to initialise the ROM image once loaded

*SRREAD

Performs the reverse operation of *SRWRITE It reads the contents of the specified ROM into memory starting at a defined location For example, to read a ROM in socket 7 into memory starting at &2ØØØ use

```
     *SRREAD 2ØØØ 6ØØØ 8ØØØ  7
```

This command will obviously cause the contents of memory to be overwritten The first two addresses are the start and end addresses in memory to which ROM data is to be transferred The third address (8ØØØ) is the start address in RAM of the data to be transferred, while finally the ROM identity (id) is specified Therefore in this case, the data in ROM 7 starting at &8ØØØ will be transferred to &2ØØØ until &6ØØØ is reached

*SRSAVE

This command is like *SRREAD except that the ROM contents are saved to the current filing system under the specified filename For example, to save a ROM in RAM bank 7 we could use

```
    *SRSAVE R ROM 8ØØØ CØØØ 7
```

The first two address specified are the start and end addresses of the sideways RAM to be transferred The final value is the ROM identity number Like *SRLOAD a Q may be tagged onto the end of the command line for a quick, but memory destroying, save

Note that it is not possible to save a ROM with an identity number greater than 8 (ie 9 to 15) This is to prevent illegal copying of the ROMs in the Megabit ROM An illegal address error will result if you try

ROM Copyright

It is apparent now that it would be advantageous to have copies of ROMs on disc Using the above commands it is a simple process to do this A whole library of ROM images can be held on a single disc and loaded in when needed - thus avoiding the need to continually get inside the case to change chips or swap ROM cartridges However, images of some commercially available ROMs will not function in sideways RAM because of the protection mechanisms employed by software houses to prevent the abuse of such copying If a ROM image hangs up or will simply not function then it is almost certainly protected

Listing 3 1 Traces service calls as they are issued by the MOS Save as TRACE

```
   10 REM Service call Trace
   20 REM (C) Bruce Smith June 1986
   30 REM The Advanced SRAM Guide
   40
   50 PROCassemble
   60 END
   70
   80 DEF PROCassemble
   90 oswrch=&FFEE
  100 FOR pass=4 TO 7 STEP 3
  110 P%=&8000   O%=&5000
  120 [
  130 OPT pass
  140 EQUB 0
  150 EQUW 0
  160 JMP service
  170 EQUB &82
  180 EQUB offset MOD 256
  190 EQUB 1
  200  title
  210 EQUS "Service Trace ROM"
  220 EQUB 0
  230  offset
  240 EQUB 0
  250 EQUS "(C) Bruce Smith"
  260 EQUB 0
  270  service
  280 PHA
  290 PHA
  300 LSR A
  310 LSR A
  320 LSR A
  330 LSR A
  340 JSR convert
  350 PLA
  360 AND #15
  370 JSR convert
  380 LDA #32
  390 JSR oswrch
  400 PLA
  410 RTS
  420  convert
  430 SED
  440 CMP #10
  450 ADC #&30
  460 CLD
  470 JMP oswrch
```

Listing 3 1 continued

```
  480 ]
  490 NEXT
  500 ENDPROC
```

Chapter Four
The Help Service

I don't intend to deal with each service call in numeric order Some are much more difficult to process than others, so I'll start with the easiest, *HELP

*HELP

When the MOS encounters a *HELP command it looks through its ROM type table and polls each of the ROMs it sees with service call 9 All ROMs should respond to this call by printing out at least their title string The equivalent in BASIC of the service coding is

```
    IF A%=9 THEN PRINT title$ ELSE RETURN
```

In machine code this becomes

```
    CMP #9          \ is it *HELP?
    BEQ help        \ yes, process it
    RTS             \ no, so return
     help           \
    JSR osnewl      \ print a new line
    LDX #255        \ use X as index
     helploop
    INX             \ increment index
    LDA title,X     \ get byte
    BEQ done        \ if zero, then finished
    JSR osasci      \ print byte
    BRA helploop    \ and repeat once more
     done
    JSR osnewl      \ print a new line
    RTS             \ and return
```

The coding uses OSNEWL to print a blank line before and after the title string for layout purposes The main ROM title string is the one being printed out The end of this is located by testing for the terminating zero byte Note that this should not be printed itself and an exit should be performed once it has been identified Program 4 1 puts this simple *HELP implementation into action Save the program under the filename 'HELP1' You can see the machine code described above in lines 27Ø to 43Ø inclusive The title string and terminating byte are in lines 2ØØ-22Ø
One very good reason for including a *HELP routine in your software is to aid debugging By issuing the command it is immediately evident whether your sideways software is present If you have a working *HELP service routine and it will not respond then there's more likely to be a bug in your software - or perhaps you've forgotten to load it into sideways RAM correctly (I've made that error on many an occasion!)
If you type *HELP you will notice that all of the resident software in the Master print out a version number as well as its title string This can be important when you wish to see whether you are using the latest version of your firmware and is not difficult to do It simply involves inserting the ASCII version string in the correct position (after the ASCII title string) and terminating it with a zero byte Once the service coding has printed out the title string, it goes around once again to print the version number This could be done by placing another print loop at the end of the 'helploop' routine, but it's inefficient and by rewriting the 'help' routine slightly a single 'helploop' can be used

```
 help
JSR osnewl     \ print new line
LDX #255       \ start index
JSR helploop   \ print title string
JSR helploop   \ print version string
JSR osnewl     \ print new line
RTS            \ and return
```

Three new lines are needed to insert a version string, directly after the title string terminating byte

```
 version
EQUS " 1 ØØ'
EQUB Ø
```

You can use program 4 1 as a base to work on If you insert and change the relevant lines, then RENUMBER the program, you should arrive at listing 4 2 Save this as 'HELP2' Once this has been RUN and initialised, typing *HELP should give

```
    Help Test ROM 1 ØØ
```

near the bottom of your *HELP list
  Obviously it will help when debugging if, each time a change is made to the ROM coding, you simply increment the version number by Ø ØI, resave it and make a note in a safe place detailing the change

Extended *HELP (Service Call 9)

Type *HELP again The resulting list on a Master 128 will look like this

```
    *HELP

    OS 3 2Ø
      MOS

    TERMINAL 1 2Ø

    VIEW B3 Ø

    Advanced DFS 1 5Ø
      ADFS

    EDIT 4

    ViewSheet B1 Ø

    DFS 2 2Ø
      DFS

    SRAM 1 ØØ
```

Some of the help strings have another line of information set below them, indented slightly For example

```
    OS 3 2Ø
      MOS
```

This shows that an extended help facility is provided By typing *HELP followed by a space and the word on the next line we can gain more information, normally a list of the commands provided by that particular ROM Typing *HELP MOS gives

```
OS 3 2Ø
CAT         ADFS        APPEND      BASIC
BUILD       CLOSE       CONFIGURE   CODE
CREATE      DUMP        DELETE      EXEC
EX          FX          GOIO        GO
HELP        INFO        IGNORE      INSERT
KEY         LOAD        LIST        LINE
LIBFS       MOTOR       MOVE        OPT
PRINT       RUN         REMOVE      ROM
ROMS        SAVE        SHADOW      SHOW
SHUT        SPOOL       SPOOLON     STATUS
TAPE        TV          TIME        TYPE
UNPLUG      X
```

Adding this facility to our own *HELP is a two-stage affair First we need to print out the extra item(s) of information below the title string when service call 9 is processed Second, we need to be able to distinguish between a straight *HELP and an extended *HELP The first part is to just extend the printing routine For example, suppose we wish to print out

```
Help Test ROM  1 ØØ
  Commands
```

The coding for this would simply become

```
JSR  help        \ print title string and version
LDX ≠255         \ start X
 details
INX              \ increment index
LDA command,X    \ get byte
BEQ donecom      \ finish if it is zero
JSR osasci       \ print it
BRA details      \ do next byte
JSR osnewl       \ print new line
RTS              \ return
 command
EQUS  Commands"
EQUB Ø
```

The 'help' routine is as listed in the previous program Checking for extended help is a two-stage affair First we need to see if there is a command after the *HELP or not If there is, we test to see if it's ours, ie 'COMMAND'

Testing for the presence of an extra command involves looking for a return character (ASCII 13 or &ØD) If there is one, the command is a straight *HELP if not, there is an extended *HELP command present

The next question is how do we go about locating the presence of a return? The answer lies in memory locations &F2 and &F3 These two bytes, which I have termed 'comline', contain the address of the first non-space character after the help when combined with the Y register using indirect indexed addressing In other words the code

```
    LDA (comline),Y
```

will load either a return character or the first letter of the extended help into the accumulator If the character is not a return, we need to test each character from here against a copy of our own This is done by incrementing Y and comparing it with a copy of the command using the X register, as in the following example

```
    LDA (comline),Y     \ get byte after *HELP
    CMP ≠13             \ is it return?
    BEQ out             \ yes to return
    LDX ≠&FF            \ initialise X index
    DEY                 \ decrement Y index
     again
    INX                 \ increment X index
    INY                 \ increment Y index
    LDA (comline),Y     \ get byte
    AND ≠&DF            \ force to upper case
    CMP com,X           \ compare against table
    BEQ again           \ if same do again
    LDA com,X           \ get unlike byte from table
    CMP ≠&FE            \ is it end marker flag?
    BEQ mine            \ yes go to print routine
    RTS
     com
    EQUS "COMMANDS"
    EQUB &FE
```

The first two instructions test for a return character - if this is encountered then a branch to an RTS is made The X register is set to &FF and the Y register decremented to make it point to the character before the start of the extended help The label 'again' marks the main loop Both index registers are incremented to make them point at the first character in the extended help and 'com' table The first byte is extracted from after the *HELP Using &DF this byte is then forced into an upper case, or capital, character (see note at end of chapter) This is important as the protocols allow us to enter

```
*HELP COMMANDS
*HELP commands
*HELP CoMmAnDs
```

or any such combination Forcing the byte to upper case allows us simply to test it against a table of upper case characters starting at 'com' If the bytes are the same the loop is repeated until an unlike character is encountered I have marked the end of the 'com' string with a particular byte &FE This can then be tested for If it is indeed &FE then we have identified the string as COMMANDS and the relevant extra details can be printed out If the unlike byte is not an &FE then this is not our extended help and an RTS can be made Or can it? The trouble now is that we have destroyed the contents of all three index registers, so when the MOS passes this call onto the next ROM some very confusing things could happen So, to avoid this, on entering the service coding push all three registers onto the hardware stack, ie

```
PHA - push accumulator
PHX - push X
PHY - push Y
```

and restore them prior to returning, ie

```
PLY - pull Y
PLX - pull X
PLA - pull accumulator
```

Listing 4 3 puts all this into operation Once RUN, type *HELP, then type *HELP COMMANDS This will respond with

```
Help Test ROM 1 ØØ
   COMPRESS
   EXPAND
```

Where COMPRESS and EXPAND might be two commands implemented by our ROM If you look through the listing you can see the important new sections of code

```
Lines 34Ø to 36Ø - save registers
Lines 37Ø to 38Ø - test for return
Line  39Ø        - branch to check if extended *HELP
```

Lines 4ØØ to 5ØØ - print *HELP message
Lines 52Ø to 64Ø - check for COMMANDS
Lines 65Ø to 7ØØ - restore registers and return
Lines 72Ø to 78Ø - print *HELP messages
Lines 8ØØ to 87Ø - printing routine
Lines 89Ø to 96Ø - print extended help information
Lines 97Ø to 1Ø2Ø - restore and return
Lines 1Ø3Ø to 114Ø - extended help details

Interactive Help (Service Call 24)

Once the MOS has polled all ROMs with service call 9 it then polls them with service call 24 This allows your ROM to take up and provide any more help information required This will generally be an 'interactive help' and the exact nature of the help may well depend on answers to questions that are prompted by the ROM How and what you do with this service call, if anything at all, is up to you One example is seen on Econet network machines fitted with the Advanced Network Filing System (ANFS) When this service call is issued the ANFS will look for a file called 'HELP on the fileserver You could implement this from tape or disc or even load in routines from the ROM itself

Program 4 4 shows how the service call can be trapped to see if the user wished further information on imaginary commands within the Help Test ROM In fact it replaces the extended help detailed above

Save the listing under the filename 'HELP4' After running and initialising the ROM, type *HELP After the standard help messages the following line will be printed on the screen

```
    Do you wish more help? (Y/N)
```

Pressing the Y key will display

```
    The following commands are available
    with the Master ROM
      COMPRESS - compacts a graphics screen
      EXPAND   - unpacks a graphics screen
```

Pressing any key other than Y will cause the routine to exit

Masking

In a few instances above we used the byte &DF with the AND command to force an ASCII character to its upper case component Let's examine how this works Consider the ASCII and binary representation of the letters B and b

```
    ASC("B") = &42  = 0100 0010
    ASC("b") = &62  = 0110 0010
```

The only difference between these two values at bit level is that bit 5 is either set or clear Therefore by toggling bit 5 we can swap the case of an ASCII alpha character To force lower case to capital we need to ensure that every bit in the byte is set to 1 with the exception of bit 5 In binary the mask is

```
    1101 1111 = &DF
```

If the accumulator holds &62 (ASC"b') and this is ANDed with &DF we get

```
    ASC"b" = 0110 0010
    &DF    = 1101 1111
    AND    = 0100 0010  =  &42 or ASC "B"
```

Listing 4 1 Traps service call 9 to output a simple *HELP message Save as HELP1

```
   10 REM Simple *HELP
   20 REM (C) Bruce Smith June 1986
   30 REM Advanced SRAM Guide
   40
   50 PROCassemble
   60 *SRWRITE 5000 +100 8000 6
   70 END
   80 DEF PROCassemble
   90 osnewl=&FFE7
  100 FOR pass=4 TO 7 STEP 3
  110 P%=&8000   O%=&5000
  120 [
  130 OPT pass
  140 EQUB 0
  150 EQUW 0
  160 JMP service
  170 EQUB &82
  180 EQUB offset MOD 256
  190 EQUB 1
  200  title
  210 EQUS  Help Test ROM"
  220 EQUB 0
  230  offset
  240 EQUB 0
  250 EQUS "(C) Bruce Smith"
  260 EQUB 0
  270  service
  280 CMP #9
  290 BEQ help
  300 RTS
  310 \
  320  help
  330 JSR osnewl
  340 LDX #&FF
  350  helploop
  360 INX
  370 LDA title,X
  380 BEQ done
  390 JSR &FFE3
  400 BRA helploop
  410  done
  420 JSR osnewl
  430 RTS
  440 ]
  450 NEXT
  460 ENDPROC
```

Listing 4 2 Gives version number as well as ROM string title Save as HELP2 Developed from listing 4 1 (HELP1)

```
   10 REM *HELP with Version No
   20 REM (C) Bruce Smith June 1986
   30 REM Advanced SRAM Guide
   40
   50 PROCassemble
   60 *SRWRITE 5000 +100 8000 6
   70 END
   80 DEF PROCassemble
   90 osnewl=&FFE7
  100 FOR pass=4 TO 7 STEP 3
  110 P%=&8000   O%=&5000
  120 [
  130 OPT pass
  140 EQUB 0
  150 EQUW 0
  160 JMP service
  170 EQUB &82
  180 EQUB offset MOD 256
  190 EQUB 1
  200  title
  210 EQUS "Help Test ROM"
  220 EQUB 0
  230  version
  240 EQUS " 1 00"
  250 EQUB 0
  260  offset
  270 EQUB 0
  280 EQUS "(C) Bruce Smith"
  290 EQUB 0
  300  service
  310 CMP ≠9
  320 BEQ help
  330 RTS
  340 \
  350  help
  360 JSR osnewl
  370 LDX ≠&FF
  380 JSR helploop
  390 JSR helploop
  400 JSR osnewl
  410 RTS
  420 \
  430  helploop
  440 INX
  450 LDA title,X
```

Listing 4 2 continued

```
  460 BEQ done
  470 JSR &FFE3
  480 BRA helploop
  490  done
  500 RTS
  510 ]
  520 NEXT
  530 ENDPROC
```

Listing 4 3 Adds description of ROM commands to *HELP message Save as HELP3 This program forms the basis of many others in this book

```
   10 REM Extended *HELP
   20 REM (C) Bruce Smith June 1986
   30 REM Advanced SRAM Guide
   40
   50 PROCassemble
   60 *SRWRITE 5000 +200 8000 6
   70 END
   80 DEF PROCassemble
   90 osnewl=&FFE7
  100 comline=&F2
  110 FOR pass=4 TO 7 STEP 3
  120 P%=&8000   O%=&5000
  130 [
  140 OPT pass
  150 EQUB 0
  160 EQUW 0
  170 JMP service
  180 EQUB &82
  190 EQUB offset MOD 256
  200 EQUB 1
  210  title
  220 EQUS "Help Test ROM"
  230 EQUB 0
  240  version
  250 EQUS " 1 00"
  260 EQUB 0
  270  offset
  280 EQUB 0
  290 EQUS "(C) Bruce Smith
  300 EQUB 0
  310  service
  320 CMP #9
  330 BNE nothelp
  340 PHA
```

Listing 4 3 continued

```
350 PHX
360 PHY
370 LDA (comline),Y
380 CMP #13
390 BNE check
400 JSR help
410 LDX #255
420  details
430 INX
440 LDA command,X
450 BEQ donecommand
460 JSR &FFE3
470 BRA details
480  donecommand
490 JSR osnewl
500 BRA restore
510 \
520  check
530 LDX #255
540 DEY
550  again
560 INX
570 INY
580 LDA (comline),Y
590 AND #&DF
600 CMP com,X
610 BEQ again
620 LDA com,X
630 CMP #&FE
640 BEQ mine
650  restore
660 PLY
670 PLX
680 PLA
690  nothelp
700 RTS
710 \
720  help
730 JSR osnewl
740 LDX #&FF
750 JSR helploop
760 JSR helploop
770 JSR osnewl
780 RTS
790 \
800  helploop
810 INX
820 LDA title,X
```

Listing 4 3 continued

```
  830 BEQ done
  840 JSR &FFE3
  850 BRA helploop
  860  done
  870 RTS
  880  mine
  890 JSR help
  900 LDX #255
  910  more
  920 INX
  930 LDA lists,X
  940 BMI alldone
  950 JSR &FFE3
  960 BRA more
  970  alldone
  980 PLY
  990 PLX
 1000 PLA
 1010 LDA #0
 1020 RTS
 1030  com
 1040 EQUS"COMMANDS
 1050 EQUB &FE
 1060  command
 1070 EQUS   Commands'
 1080 EQUB 0
 1090  lists
 1100 EQUS "  COMPRESS"
 1110 EQUB 13
 1120 EQUS "  EXPAND"
 1130 EQUB 13
 1140 EQUB &FF
 1150 ]
 1160 NEXT
 1170 ENDPROC
```

Listing 4 4 Extra help information can be called by the user Save as HELP4

```
   10 REM Interactive *HELP
   20 REM (C) Bruce Smith June 1986
   30 REM Advanced SRAM Guide
   40
   50 PROCassemble
   60 *SRWRITE 5000 +200 8000 6
   70 END
   80
```

Listing 4 4 continued

```
  90 DEF PROCassemble
 100 osnewl=&FFE7
 110 oswrch=&FFEE
 120 osasci=&FFE3
 130 osrdch=&FFE0
 140 FOR pass=4 TO 7 STEP 3
 150 P%=&8000   O%=&5000
 160 [
 170 OPT pass
 180 EQUB 0
 190 EQUW 0
 200 JMP service
 210 EQUB &82
 220 EQUB offset MOD 256
 230 EQUB 1
 240  title
 250 EQUS "Help Test ROM"
 260 EQUB 0
 270  version
 280 EQUS ' 1 00'
 290 EQUB 0
 300  offset
 310 EQUB 0
 320 EQUS "(C) Bruce Smith"
 330 EQUB 0
 340  service
 350 CMP #9
 360 BEQ help
 370 CMP #24
 380 BEQ interact
 390 RTS
 400 \
 410  help
 420 JSR osnewl
 430 LDX #&FF
 440 JSR helploop
 450 JSR helploop
 460 JSR osnewl
 470 RTS
 480 \
 490  helploop
 500 INX
 510 LDA title,X
 520 BEQ done
 530 JSR &FFE3
 540 BRA helploop
 550  done
 560 RTS
```

Listing 4 4 continued

```
570 \
580  interact
590 LDX #255
600  first
610 INX
620 LDA message,X
630 BEQ done1
640 JSR oswrch
650 BRA first
660  done1
670 JSR osrdch
680 AND #&DF
690 CMP #ASC("Y")
700 BEQ carryon
710 RTS
720  carryon
730 INX
740 LDA message,X
750 JSR osasci
760 BPL carryon
770 RTS
780 \
790  message
800 EQUS "Do you wish more "
810 EQUS "help? (Y/N)"
820 EQUB 0
830  EQUB 13
840 EQUS "The following commands "
850 EQUS "are available"
860 EQUB 13
870 EQUS "within the Master ROM "
880 EQUB 13
890 EQUS " COMPRESS - compacts a "
900 EQUS "graphics screen
910 EQUB 13
920 EQUS " EXPAND   - unpacks a "
930 EQUS "graphics screen"
940 EQUB 13
950 EQUB 255
960 ]
970 NEXT
980 ENDPROC
```

Chapter Five
Interpreters

Command Action (service call 4)

When the MOS encounters an unrecognised command it issues service call 4 to each ROM in turn As with *HELP, the vector at &F2 (comvec) is used with the Y register, and contains the start address of the unrecognised command It does not point to the asterisk but to the first character after the asterisk If the ROM cannot recognise the command as one of its own then it must restore all registers to their original values and perform a simple RTS The command is then offered to the disc or net filing systems If the command is recognised, the service coding should jump to the correct routine within Once complete the registers need not be restored, but the accumulator MUST be loaded with Ø

```
    LDA ≠Ø
```

before an RTS is performed This tells the MOS the command has been recognised and acted upon, so it will not be offered to any of the other ROMs or filing systems

Writing the Interpreter

What is an interpreter? In fact, it is no different from its linguistic counterpart - a device for identifying and translating, in this case commands which are strings of characters in a certain order

The ROM image that we will construct in this chapter will contain three commands, although only two will be of immediate practical use The commands will be

    *ITALICS - does nothing initially
    *MODERN - selects a 'modern' style character font
    *STANDARD - reselects the normal character font

*MODERN then will redefine the shape of the letters in the Master's character font so they look 'modern' when printed on the screen (except in mode 7) Figure 5 1 shows the appearance of the font I will use program 4 3 from Chapter 4 (saved as 'HELP3'), as the basis of the interpreter

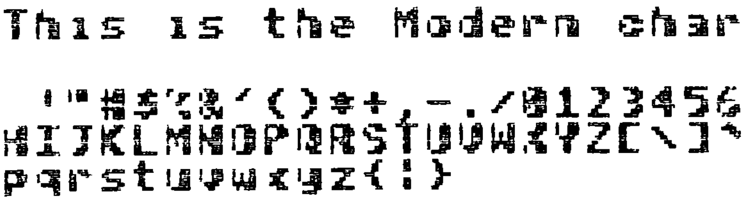

This is the Modern char
!"#$%&'()*+,-./Ø123456
HIJKLMNOPQRSTUVWXYZ[\]^
pqrstuvwxyz{|}

Figure 5 1 Character style selected by *MODERN

We trap service call 4 by comparing and branching

```
CMP ≠4
BEQ unrecognised
```

Identifying the command is performed in a similar way to trapping an extended *HELP command name It involves comparing the unrecognised command against a table of commands - the command table In BASIC this would be

```
INPUT 'Command" com$
addr%=Ø
REPEAT
addr%=addr%+1
READ table$
UNTIL table$=com$ OR table$="END"
ON addr% GOTO 5ØØ, 6ØØ, 7ØØ, 8ØØ
DATA "ITALICS"
DATA "MODERN"
DATA "STANDARD"
DATA 'END"
```

The command is found and compared against the command table, until it is recognised or the end of the table is reached If the command is identified then its address must also be found - here a variable is used to hold it which is incremented each time through the REPEAT UNTIL loop

Translating this into code is not too hard but you will need to study the following assembler carefully

```
 unrecognised
LDX ≠255          \ start command table index
DEY               \ decrement comvec index
PHY               \ keep a copy on the stack
 identify
INX               \ increment X index
INY               \ increment Y index
LDA  (comvec),Y   \ get byte from unrecognised
                  \ command
AND ≠&DF          \ force to upper case
CMP comtable,X    \ compare against command table
BEQ identify      \ if it compares try next byte
```

This code is very similar to the extended help facility in chapter 4 Now consider the command table itself

```
 comtable
EQUS "MODERN"
EQUB modern DIV 256
EQUB modern MOD 256
EQUS  ITALICS"
EQUB italics  DIV 256
EQUB italics  MOD 256
EQUS "STANDARD
EQUB standard DIV 256
EQUB standard MOD 256
EQUB &FF
```

This consists of the ASCII string of each command name, minus the asterisk, followed by the address, high byte first, of each command The command table is terminated by &FF Figure 5 2 shows this diagramatically

MODERN
modern high byte
modern low byte
ITALICS
italics high byte
italics low byte
STANDARD
standard high byte
standard low byte
&FF

Figure 5 2 Construction of command table

As the execution address of any command is going to be within a ROM, it will have a high byte of &8Ø or higher This is useful as it means that it will set the negative flag when loaded into the accumulator Carrying on with the code from 'identify' above gives

```
    LDA comtable,X      \ get unlike byte from command
                        \ table
    BMI address         \ if negative must be address
```

Of course it may not be an address it may simply be that comvec and comtable are not alike Therefore we need a means of moving onto the start of the next command in the command table The way to do this is by finding the first address byte and incrementing the X register by one

```
     moveon
    INX                 \ increment X
    LDA comtable,X      \ get next byte from table
    BPL moveon          \ if not negative do again
    INX                 \ increase X by one
    PLY                 \ restore original value of Y
    PHY                 \ save once again for next time
    JMP identify        \ and repeat identifying loop
```

The routine 'address' needs to test to see if the negative byte is in fact &FF, the end of command table marker If it is not then the accumulator will contain the high byte of the command's execution address This along with the low byte can be placed into a zero page vector and an indirect jump to the interpreted command's address performed

```
     address
    CMP #&FF            \ is it top of command table?
    BNE notFF           \ branch if not
    PLY                 \ else balance stack
    BRA alldone         \ and branch to return routine
     notFF
    STA &39             \ else save high byte
    INX                 \ increment X
    LDA comtable,X      \ and get low byte
    STA &38             \ and save
    JMP (&38)           \ and go to it!
```

I have made a habit of first testing interpreters before actually adding the code that makes each individual command To do this I give them all the same execution address and then get the commands to do somethıng obvıous such as make a beep on the speaker, or prınt a letter on the screen

```
     modern
     ıtalıcs
     standard
    LDA ≠7
    JMP osascı
```

Once the command has been executed, the stack, whıch was previously pushed wıth the regıster contents, must be pulled and the accumulator loaded wıth Ø

```
     found
    PLY
    PLY
    PLX
    PLA
    LDA ≠Ø
    RTS
```

The extra PLY ıs to balance the extra PHY made at the start of 'unrecognısed' and subsequently ın 'moveon'
  Lıstıng 5 I puts all thıs ınto play Enter thıs and save ıt under the fılename 'INTERP' (We'll be usıng thıs agaın later) RUN the program and ınıtıalıse the ROM Now try typıng any of the commands and you should get a beep on the speaker

Debuggıng Interpreters

As the programs hereın all contaın checksum calculators ıt should be easy for you to get programs runnıng correctly before they are transferred ınto sıdeways RAM The trouble starts when you wrıte your own - no checksums Debuggıng ROM ımages can be ınfurıatıng but ıs ultımately rewardıng' So I'll provıde some useful poınters
  Wıthout doubt the most common cause of programs crashıng ıs bad stack management Thıs shows up ın two maın ways Fırst, when you execute the command you get a message somethıng lıke

```
    at lıne 23Ø
```

on the screen Of course the lıne number wıll probably be dıfferent and vary The second manıfestatıon ıs that you execute the command and nothıng happens other than a couple of returns are echoed to the screen In both cases check your lıstıng and ensure every push ıs balanced by a pull I often keep using pushes instead of pulls!

Always test your commands fully They may only crash after a bit of continuous use I do this by putting them inside a REPEAT UNTIL loop To test the above interpreter I used

```
    1Ø REPEAT
    2Ø *MODERN
    3Ø *ITALICS
    4Ø *STANDARD
    5Ø UNTIL Ø
```

Once you set this running you'll get a continuous beep And leave it running for a few minutes - have a coffee - if it's still making a noise when you get back - you're okay!

If the command executes but then accesses the disc or filing system then you have forgotten to do a LDA #Ø before the RTS

If all else fails you'll need the help of a machine code monitor program, such as BBC Soft's Monitor ROM This will enable you to step through the program and see (hopefully) just where it's going wrong

Writing Commands

There are three stages in writing commands for use in sideways format

Stage 1 Write it in BASIC where possible! This has the advantage of being quick and allows you to calculate tabs for screen printing etc, with the minimum of fuss

Stage 2 Convert it to machine code, but get it running in normal memory first Make sure you don't use addresses within the code itself to store things (you won't be able to do so once the program is in sideways RAM!) and make it as self-contained and compact as possible

Stage 3 add it to your sideways RAM assembly listing

Now let's apply these rules to our ROM image The first step is to design the modern character font with a suitable program or pencil and paper Once this is done, all the characters can be placed into a BASIC program in the form of VDU23 statements When run this should, if correct, redefine the character font to take a modern appearance

The *STANDARD command can now be tested To return to the standard font, the character font needs to be imploded using *FX2Ø,Ø To convert all this to machine code, we need to create a loop to read in data items and send them to the VDU output stream using OSWRCH The total length of data can be optimised by including the 23 and ASCII character code inside the printing loop

```
 modern
LDA ≠data DIV 256    \ get high byte of data
STA &71              \ save it
LDA ≠data DIV 256    \ get low byte of data
STA &7Ø              \ save it
LDA ≠33              \ start character is ASCII 33
STA &72              \ save it
 outerloop
LDY ≠Ø
LDX ≠Ø
LDA ≠23
JSR oswrch           \ do VDU23
LDA &72              \ get character code
JSR oswrch           \ send it to VDU stream
 innerloop
LDA (&7Ø),Y          \ get data byte
JSR oswrch           \ send it to VDU stream
INC &7Ø              \ increment low byte address
BEQ nohi             \ branch if not zero
INC &71              \ else increment high byte
 nohi
INX                  \ increment counter
CPX ≠8               \ 8 bytes sent yet?
BNE innerloop        \ branch if not
LDA &72              \ get character
INC A                \ increment by one
STA &72              \ save result
CMP ≠127             \ all done yet?
BNE outerloop        \ repeat until all done
```

The coding for *STANDARD is easy, and takes just a few lines of assembler

```
 standard
LDA ≠2Ø
LDX ≠Ø
LDY ≠Ø
JSR osbyte           \ do *FX2Ø,Ø
```

Listing 5 2 combines both of these segments Note that line numbering starts at 2ØØØ This is because we can use listing 5 1 ('INTERP') as the basis, and cut down on typing So enter this and *SPOOL it to a file Use the filename SCOMMS - S for Spool, COMMS for Commands In case you've forgotten, type in

```
    *SPOOL SCOMMS
    LIST
    *SPOOL
```

The program will not run as it stands so don't try! Now reload 'INTERP' and make these additions and changes

```
    Add line          55 PROCread
    Change line       7Ø *SRWRITE 5ØØØ +5ØØ 8ØØØ 6
    Add line         115 oswrch=&FFEE
    Delete lines    179Ø, 18ØØ and 182Ø
    Add line        1915 RTS
    Delete lines    192Ø to 2Ø4Ø inclusive
```

Now *EXEC in the spooled listing

```
    *EXEC SCOMMS
```

Save the program under the filename 'MODERN' RUN and initialise as normal Then enter mode 6 and type

```
    *MODERN
    OLD
    LIST
```

to see the effect *STANDARD makes everthing normal again *ITALICS will still give a beep, though you can of course extend it to give you italic text (The disc which contains all the listings in this book also has the VDU codes for this in case you don't feel like designing your own)

Gaining Workspace

Your ROMs will at times require workspace -- areas of memory in which they can place information You need to choose this with care as it could be the space used by other ROMs The *MODERN code used three locations in zero page, &7Ø, &71 and &72 These are of course in the user area and are free for use, but it is not good practice to go around changing locations that are meant to be free and available to normal programs One way to overcome this is to save the contents of memory somewhere before using it and then restore it to its original value before returning I tend to to save memory from &7Ø to &8F on the very bottom of the hardware stack from &1ØØ The stack pointer should never get this low -- I'd love to see a program that does it! The two routines to push and pull are

```
 pushzero
LDX #255        \ initialise index
 loop
INX             \ increment index
LDA &70,X       \ get byte
STA &100,X      \ and save it
CPX #1F         \ all done?
BNE loop        \ no, continue
RTS             \ and return

 pullzero
LDX #255        \ initialise index
 loop
INX             \ increment index
LDA &100,X      \ get byte
STA &70,X       \ and save it
CPX #1F         \ all done?
BNE loop        \ no, continue
RTS             \ and return
```

I tend to use both as subroutine calls, ie using JSR As a rule, call pushzero after you push the registers and pullzero before you pull them

Listing 5 1 Test interpreter (save as INTERP)

```
   10 REM Test Interpreter
   20 REM (C) Bruce Smith June 1986
   30 REM Advanced SRAM Guide
   40
   50 PROCassemble
   60 PROCchecksum
   70 *SRWRITE 5000 +200 8000 6
   80 END
   90
  100 DEF PROCassemble
  110 osnewl=&FFE7
  120 osasci=&FFE3
  130 comline=&F2
  140 FOR pass=4 TO 7 STEP 3
  150 P%=&8000   O%=&5000
  160 [
  170 OPT pass
  180 EQUB 0
  190 EQUW 0
  200 JMP service
  210 EQUB &82
  220 EQUB offset MOD 256
  230 EQUB 1
  240  title
  250 EQUS "Test Interpreter ROM"
  260 EQUB 0
  270  version
  280 EQUS " 1 00"
  290 EQUB 0
  300  offset
  310 EQUB 0
  320 EQUS "(C) Bruce Smith"
  330 EQUB 0
  340  service
  350 PHA
  360 PHX
  370 PHY
  380 CMP ≠9
  390 BNE nothelp
  400 LDA (comline),Y
  410 CMP ≠13
  420 BNE check
  430 JSR help
  440 LDX ≠255
  450  details
  460 INX
  470 LDA command,X
  480 BEQ donecommand
```

Listing 5 1 continued

```
490 JSR &FFE3
5ØØ BRA details
51Ø  donecommand
52Ø JSR osnewl
53Ø BRA restore
54Ø \
55Ø  check
56Ø LDX ≠255
57Ø DEY
58Ø  again
59Ø INX
6ØØ INY
61Ø LDA (comline),Y
62Ø AND ≠&DF
63Ø CMP com,X
64Ø BEQ again
65Ø LDA com,X
66Ø CMP ≠&FE
67Ø BEQ mine
68Ø  restore
69Ø PLY
7ØØ PLX
71Ø PLA
72Ø RTS
73Ø \
74Ø  nothelp
75Ø CMP ≠4
76Ø BEQ unrecognised
77Ø BRA alldone
78Ø \
79Ø  help
8ØØ JSR osnewl
81Ø LDX ≠&FF
82Ø JSR helploop
83Ø JSR helploop
84Ø JSR osnewl
85Ø RTS
86Ø \
87Ø  helploop
88Ø INX
89Ø LDA title,X
9ØØ BEQ done
91Ø JSR &FFE3
92Ø BRA helploop
93Ø  done
94Ø RTS
95Ø \
96Ø  mine
```

Listing 5 1 continued

```
  970 JSR help
  980 LDX ≠255
  990  more
 1000 INX
 1010 LDA lists,X
 1020 BMI alldone
 1030 JSR &FFE3
 1040 BRA more
 1050 \
 1060  alldone
 1070 PLY
 1080 PLX
 1090 PLA
 1100 RTS
 1110 \
 1120  com
 1130 EQUS"COMMANDS"
 1140 EQUB &FE
 1150  command
 1160 EQUS'  Commands"
 1170 EQUB 0
 1180  lists
 1190 EQUS "  MODERN"
 1200 EQUB 13
 1210 EQUS "  ITALICS"
 1220 EQUB 13
 1230 EQUS "  STANDARD"
 1240 EQUB 13
 1250 EQUB &FF
 1260 \
 1270  unrecognised
 1280 LDX ≠255
 1290 DEY
 1300 PHY
 1310  identify
 1320 INX
 1330 INY
 1340 LDA (comline),Y
 1350 AND ≠&DF
 1360 CMP comtable,X
 1370 BEQ identify
 1380 LDA comtable,X
 1390 BMI address
 1400 \
 1410  moveon
 1420 INX
 1430 LDA comtable,X
 1440 BPL moveon
```

Listing 5 1 continued

```
 1450 BNE notend
 1460 PLY
 1470 BRA alldone
 1480 \
 1490  notend
 1500 INX
 1510 PLY
 1520 PHY
 1530 JMP identify
 1540 \
 1550  address
 1560 CMP #&FF
 1570 BNE notFF
 1580 PLY
 1590 BRA alldone
 1600  notFF
 1610 \
 1620 STA &39
 1630 INX
 1640 LDA comtable,X
 1650 STA &38
 1660 JMP (&38)
 1670 \
 1680  comtable
 1690 EQUS "MODERN"
 1700 EQUB modern DIV 256
 1710 EQUB modern MOD 256
 1720 EQUS "ITALICS"
 1730 EQUB italics DIV 256
 1740 EQUB italics MOD 256
 1750 EQUS "STANDARD"
 1760 EQUB standard DIV 256
 1770 EQUB standard MOD 256
 1780 EQUB &FF
 1790 \
 1800  modern
 1810  italics
 1820  standard
 1830 LDA #7
 1840 JSR osasci
 1850 \
 1860  found
 1870 PLY
 1880 PLY
 1890 PLX
 1900 PLA
 1910 LDA #0
 1920 RTS
```

Listing 5 1 continued

```
 1930 ]
 1940 NEXT
 1950 ENDPROC
 1960
 1970 DEF PROCchecksum
 1980 X%=0
 1990 FOR N%=&5000 TO &5136
 2000 X%=X%+?N%
 2010 NEXT
 2020 IF X%=35915 THEN ENDPROC
 2030 VDU 7
 2040 PRINT"Assembler error - re check'"
 2050 STOP
```

Listing 5 2 Font command code (these lines should be saved as SCOMMS and SPOOLed onto listing 5 1, see text for details)

```
   55 PROCread
 2000  standard
 2010 LDA ≠20
 2020 LDX ≠0
 2030 LDY ≠0
 2040 JSR &FFF4
 2050 JMP found
 2060 \
 2070  modern
 2080 LDA ≠data DIV 256
 2090 STA &71
 2100 LDA ≠data MOD 256
 2110 STA &70
 2120 LDA ≠33
 2130 STA &72
 2140  outerloop
 2150 LDY ≠0
 2160 LDX ≠0
 2170 LDA ≠23
 2180 JSR oswrch
 2190 LDA &72
 2200 JSR oswrch
 2210  innerloop
 2220 LDA (&70),Y
 2230 JSR oswrch
 2240 INC &70
 2250 BNE nohi
 2260 INC &71
 2270  nohi
```

Listing 5 2 continued

```
 228Ø INX
 229Ø CPX ≠8
 23ØØ BNE innerloop
 231Ø LDA &72
 232Ø INC A
 233Ø STA &72
 234Ø CMP ≠127
 235Ø BNE outerloop
 236Ø \
 237Ø JMP found
 238Ø \
 239Ø  data
 24ØØ ]
 241Ø NEXT
 242Ø ENDPROC
 243Ø
 244Ø DEF PROCread
 245Ø RESTORE
 246Ø DATA 12,12,12,12,12,Ø,12,Ø
 247Ø DATA 46,46,46,Ø,Ø,Ø,Ø,Ø
 248Ø DATA 23,23,63,23,63,23,23,Ø
 249Ø DATA 6,31,44,31,3,63,12,Ø
 25ØØ DATA 48,39,6,12,24,39,3,Ø
 251Ø DATA 28,46,46,28,45,39,27,Ø
 252Ø DATA 6,12,24,Ø,Ø,Ø,Ø,Ø
 253Ø DATA 6,12,24,24,24,12,6,Ø
 254Ø DATA 24,12,6,6,6,12,24,Ø
 255Ø DATA Ø,12,63,3Ø,63,12,Ø,Ø
 256Ø DATA Ø,12,12,63,12,12,Ø,Ø
 257Ø DATA Ø,Ø,Ø,Ø,Ø,12,12,24
 258Ø DATA Ø,Ø,Ø,63,Ø,Ø,Ø,Ø
 259Ø DATA Ø,Ø,Ø,Ø,Ø,12,12,Ø
 26ØØ DATA Ø,3,6,12,24,48,Ø,Ø
 261Ø DATA 3Ø,39,47,63,55,39,3Ø,Ø
 262Ø DATA 12,28,12,12,12,12,63,Ø
 263Ø DATA 3Ø,39,3,6,12,24,63,Ø
 264Ø DATA 3Ø,39,3,14,3,39,3Ø,Ø
 265Ø DATA 6,14,3Ø,46,63,6,6,Ø
 266Ø DATA 63,48,62,3,3,39,3Ø,Ø
 267Ø DATA 14,24,48,62,39,39,3Ø,Ø
 268Ø DATA 63,3,6,12,24,24,24,Ø
 269Ø DATA 3Ø,39,39,3Ø,39,39,3Ø,Ø
 27ØØ DATA 3Ø,39,39,31,3,6,28,Ø
 271Ø DATA Ø,Ø,12,12,Ø,12,12,Ø
 272Ø DATA Ø,Ø,12,12,Ø,12,12,24
 273Ø DATA 6,12,24,48,24,12,6,Ø
 274Ø DATA Ø,Ø,63,Ø,63,Ø,Ø,Ø
 275Ø DATA 24,12,6,3,6,12,24,Ø
```

Listing 5 2 continued

```
 276Ø DATA 3Ø,39,6,12,12,Ø,12,Ø
 277Ø DATA 3Ø,39,47,43,47,48,3Ø,Ø
 278Ø DATA 3Ø,39,39,63,39,39,39,Ø
 279Ø DATA 62,39,39,62,39,39,62,Ø
 28ØØ DATA 3Ø,39,48,48,48,39,3Ø,Ø
 281Ø DATA 6Ø,46,39,39,39,46,6Ø,Ø
 282Ø DATA 63,48,48,62,48,48,63,Ø
 283Ø DATA 63,48,48,62,48,48,48,Ø
 284Ø DATA 3Ø,39,48,47,39,39,3Ø,Ø
 285Ø DATA 39,39,39,63,39,39,39,Ø
 286Ø DATA 63,12,12,12,12,12,63,Ø
 287Ø DATA 31,6,6,6,6,46,28,Ø
 288Ø DATA 39,46,6Ø,56,6Ø,46,39,Ø
 289Ø DATA 48,48,48,48,48,48,63,Ø
 29ØØ DATA 35,55,63,43,43,35,35,Ø
 291Ø DATA 39,39,55,63,47,39,39,Ø
 292Ø DATA 3Ø,39,39,39,39,39,3Ø,Ø
 293Ø DATA 62,39,39,62,48,48,48,Ø
 294Ø DATA 3Ø,39,39,39,43,46,23,Ø
 295Ø DATA 62,39,39,62,46,39,39,Ø
 296Ø DATA 3Ø,39,48,3Ø,3,39,3Ø,Ø
 297Ø DATA 63,12,12,12,12,12,12,Ø
 298Ø DATA 39,39,39,39,39,39,3Ø,Ø
 299Ø DATA 39,39,39,39,39,3Ø,12,Ø
 3ØØØ DATA 35,35,43,43,63,55,35,Ø
 3Ø1Ø DATA 39,39,3Ø,12,3Ø,39,39,Ø
 3Ø2Ø DATA 39,39,39,3Ø,12,12,12,Ø
 3Ø3Ø DATA 63,3,6,12,24,48,63,Ø
 3Ø4Ø DATA 62,48,48,48,48,48,62,Ø
 3Ø5Ø DATA Ø,48,24,12,6,3,Ø,Ø
 3Ø6Ø DATA 31,3,3,3,3,3,31,Ø
 3Ø7Ø DATA 12,3Ø,39,3,Ø,Ø,Ø,Ø
 3Ø8Ø DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,127
 3Ø9Ø DATA 14,23,24,62,24,24,63,Ø
 31ØØ DATA Ø,Ø,3Ø,3,31,39,31,Ø
 311Ø DATA 48,48,62,39,39,39,62,Ø
 312Ø DATA Ø,Ø,3Ø,39,48,39,3Ø,Ø
 313Ø DATA 3,3,31,39,39,39,31,Ø
 314Ø DATA Ø,Ø,3Ø,39,63,48,3Ø,Ø
 315Ø DATA 14,24,24,62,24,24,24,Ø
 316Ø DATA Ø,Ø,31,39,39,31,3,3Ø
 317Ø DATA 48,48,62,39,39,39,39,Ø
 318Ø DATA 12,Ø,28,12,12,12,3Ø,Ø
 319Ø DATA 12,Ø,28,12,12,12,12,56
 32ØØ DATA 48,48,39,46,6Ø,46,39,Ø
 321Ø DATA 28,12,12,12,12,12,3Ø,Ø
 322Ø DATA Ø,Ø,23,63,43,43,35,Ø
 323Ø DATA Ø,Ø,62,39,39,39,39,Ø
```

Listing 5 2 continued

```
 3240 DATA 0,0,30,39,39,39,30,0
 3250 DATA 0,0,62,39,39,62,48,48
 3260 DATA 0,0,31,39,39,31,3,3
 3270 DATA 0,0,46,55,48,48,48,0
 3280 DATA 0,0,31,48,30,3,62,0
 3290 DATA 24,24,62,24,24,24,14,0
 3300 DATA 0,0,39,39,39,39,31,0
 3310 DATA 0,0,39,39,39,30,12,0
 3320 DATA 0,0,35,43,43,63,23,0
 3330 DATA 0,0,39,30,12,30,39,0
 3340 DATA 0,0,39,39,39,31,3,30
 3350 DATA 0,0,63,6,12,24,63,0
 3360 DATA 6,12,12,56,12,12,6,0
 3370 DATA 12,12,12,0,12,12,12,0
 3380 DATA 24,12,12,7,12,12,24,0
 3390 DATA 17,43,7,0,0,0,0,0
 3400
 3410 C%=0
 3420 FOR R%=0 TO 751
 3430 READ D%
 3440 R%?O%=D%
 3450 C%=C%+D%
 3460 NEXT
 3470 IF C%=18388 THEN ENDPROC
 3480 VDU 7
 3490 PRINT "Error in data
 3500 STOP
 3510
 3520 DEF PROCchecksum
 3530 N%=0
 3540 FOR R%=&5000 TO &5178
 3550 N%=N%+?R%
 3560 NEXT
 3570 IF N%=44796 THEN ENDPROC
 3580 VDU 7
 3590 PRINT "Assembler error
 3600 STOP
```

Chapter Six OSBYTE and OSWORD

Service calls 7 and 8 are provided to allow you to implement your own OSBYTE and OSWORD calls This is a great way to add clever little routines to your Master, particularly those that you use regularly

For instance, I have over the past year or so been adding to a ROM image The ROM is a conversion ROM It provides an increasing number of handy machine code conversion routines, such as hex to decimal, floating point and mathematical routines Rather than keep including them in each machine code program I write, I just access an OSBYTE or OSWORD call The one major disadvantage here is that without the conversion ROM present the machine code won't work, and in many cases I have to add the routines proper at a later stage - but it does allow me to get on and get the task in hand working correctly first without having to write long assembler programs to do it

I have presented a couple of the routines here to show how easy OSBYTE and OSWORD calls are to implement

Before starting you will need to find an unused OSBYTE and OSWORD number to use If you are hoping to sell your firmware commercially then you can ask Acorn Computers to assign you a number officially Once this has been done you are the bona fida user of that number and no other commercial software should clash Of course it is unlikely that Acorn will give you a number unless it is for commercial purposes, so just choose one that is not used by the Master or any of the software running on it In the examples below I use &64 and &65

When the MOS encounters a new OSBYTE or OSWORD call it issues service call 7 or 8 respectively Before doing this however it will place the contents of the accumulator, X and Y registers in zero page locations, thus

&EF	accumulator
&FØ	X register
&F1	Y register

It is important to know this as both OSBYTE and OSWORD calls use these registers to pass information
  The first thing your polling routine should do is to extract the contents of &EF and see if the call is for you If it is then the index register contents can be extracted from &FØ and &F1 In the case of an OSWORD call these two locations can be used as a vector to an information parameter block On completion any information that is to be returned should be placed into the relevant registers and copied to the respective zero page locations The accumulator should be set to zero to tell the MOS that the service call has been successful

OSBYTE

An OSBYTE call is made by placing the call number in the accumulator and any further information required by the call in the X and Y registers On return from the call the index registers will contain any results or information
  Listing 6 1 implements OSBYTE call &64, or *FX1ØØ if you prefer It will convert the number passed to it in the X register to its ASCII counterpart with the character codes returned in the X and Y registers, high and low bytes respectively For example, if X contained 255 then on return X and Y would contain &46, the ASCII code for 'F', ie 255=&FF The OSBYTE extraction code can be found from line 116Ø

```
 osbyte
PHA             \ push all registers
PHX
PHY
LDA &EF         \ get call number
CMP ≠&64        \ is it us?
BEQ yes64       \ yes, branch
JMP restore     \ else return
 yes64
LDA &FØ         \ get X register
PHA             \ save it on stack
```

```
    LSR A           \ move high nibble to low nibble
    LSR A
    LSR A
    LSR A
    JSR convert     \ perform conversion
    STA &F0         \ save high character in 'X'
    PLA             \ retrieve byte
    JSR convert     \ perform conversion on low nibble
    STA &F1         \ save low character in 'Y'
    PLY             \ balance stack
    PLX
    PLA
    LDY &F1         \ make sure registers match
    LDX &F0
    LDA #0          \ signal to MOS
    RTS             \ and return
```

Listing 6 1 is shown below in full If you have listing 4 3 from Chapter 4 ('HELP3') available then you can use this as the base and make the following additions and alterations

```
    Change Lines    10,220,1040,1070,1100,1120
    Add lines       55 PROCchecksum
                    311 CMP #7
                    312 BNE notseven
                    313 JMP osbyte
                    314  notseven
    Delete lines    1150 onwards
    Add lines       1150 to 1650 inclusive
```

Listing 6 2 tests the new OSBYTE call provided by listing 6 1 and provides a tutorial by showing how easy it is to use

OSWORD

OSWORD calls are performed by placing the call number into the accumulator and then seeding an address into the index registers before calling &FFF1 The address is in fact that of a parameter block anywhere within RAM that contains further information for the call to manipulate Information is passed back to the calling program via the parameter block As with OSBYTE, locations &EF, &F0 and &F1 contain the register contents

If you box clever then a single OSWORD number can meet all your needs as the listing below illustrates You can use the first byte in the parameter block to contain a number which your OSWORD routine looks at before deciding what action to take In this way a single OSWORD number, &65 in this case, can be used to call up to 256 different functions, simply by placing a number from Ø to 255 in the first parameter block byte If this is not enough, use the second byte as well that'll give you over 6Ø,ØØØ possible functions! I don't quite go that far, but two functions are possible here Placing a 1 or 129 in the first byte of the parameter block will allow you to use the following two routines

1	Convert two bytes into a four-byte hex ASCII string
129	As above but print it as well

Before you write your OSWORD call, work out how the parameter block will be constructed OSWORD &65 requires this set-up

Address	Function
XY+Ø	action byte
XY+1	low byte to be converted
XY+2	high byte to be converted
XY+3	high order ASCII character
XY+4	ASCII character
XY+6	ASCII character
XY+7	low order ASCII character

For example, if XY+Ø contained 1 and XY+1=&8Ø and XY+2=&FF, then on return from the call the parameter block would look like this

Address	Contents
XY+Ø	1
XY+1	&8Ø
XY+2	&FF
XY+3	ASC"F"
XY+4	ASC'F'
XY+6	ASC"8"
XY+7	ASC"Ø"

As with OSBYTE, the OSWORD coding is not difficult to follow

```
 osword
PHA                 \ save registers
PHX
PHY
LDA ≠&65            \ is it ours?
BEQ yes65           \ branch if so
```

```
   JMP restore        \ else return
    yes65
   LDY ≠Ø             \ clear index
   LDA(&FØ),Y         \ get action byte from parameter
                      \ block
   AND ≠&7F           \ mask off high (printing) bit
   CMP ≠1             \ is it a 1?
   BEQ hexasci        \ continue if so
   JMP restore        \ else return
    hexasci
   INY                \ increment index
   LDA (&FØ),Y        \ get low byte from parameter
                      \ block
   PHA                \ save it on stack
   INY                \ increment index
   LDA (&FØ),Y        \ get high byte from parameter
                      \ block
   JSR hexconvert     \ convert and save in parameter
                      \ block
   PLA                \ get low byte
   JSR hexconvert     \ convert and save in parameter
                      \ block
   LDY ≠Ø             \ clear index again
   LDA (&FØ),Y        \ get action byte again
   BMI display        \ if negative then print ASCII
                      \ string
   JMP alldone        \ else return
```

You can now see why I chose 129 for printing It's not the number itself I'm interested in, but the fact that it has the top bit set

```
   129 = &81 =  1ØØØ ØØØ1
```

This means that any of your routines can be printed by the same piece of code if they conform to the same parameter block layout The printing routine is

```
    display
   LDY ≠3             \ first byte to print is at XY+3
    print
   LDA (&FØ),Y        \ get ASCII character
   JSR osasci         \ print it
   INY                \ bump index
   CPY ≠7             \ four bytes done yet?
   BNE print          \ branch if not
   JMP alldone        \ else finish
```

Both of these routines can be seen in listing 6 3 As before if you have listing 4 3 ('HELP3') handy you can use this as a base for the OSWORD listing by making the following changes

```
    Change lines   1Ø, 22Ø, 1Ø4Ø, 1Ø7Ø, 11ØØ, 112Ø
    Add lines        55 PROCchecksum
                    311 CMP #8
                    312 BNE noteight
                    313 JMP osword
                    314  noteight
                   1133 EQUS " XY+Ø=129   Print 1"
                   1134 EQUB 13
                   115Ø to 192Ø inclusive
```

Listing 6 4 puts OSWORD &65 into action

Listing 6 1 Implements new OSBYTE call &64 Save as OSBYTE Use listing 4 3 (HELP3) as the basis for this

```
   10 REM Implement OSBYTE
   20 REM (C) Bruce Smith June 1986
   30 REM Advanced SRAM Guide
   40
   50 PROCassemble
   55 PROCchecksum
   60 *SRWRITE 5000 +200 8000 6
   70 END
   80 DEF PROCassemble
   90 osnewl=&FFE7
  100 comline=&F2
  110 FOR pass=4 TO 7 STEP 3
  120 P%=&8000   O%=&5000
  130 [
  140 OPT pass
  150 EQUB 0
  160 EQUW 0
  170 JMP service
  180 EQUB &82
  190 EQUB offset MOD 256
  200 EQUB 1
  210  title
  220 EQUS 'Osbyte Extension ROM"
  230 EQUB 0
  240  version
  250 EQUS " 1 00"
  260 EQUB 0
  270  offset
  280 EQUB 0
  290 EQUS "(C) Bruce Smith"
  300 EQUB 0
  310  service
  311 CMP #7
  312 BNE notseven
  313 JMP osbyte
  314  notseven
  320 CMP #9
  330 BNE nothelp
  340 PHA
  350 PHX
  360 PHY
  370 LDA (comline),Y
  380 CMP #13
  390 BNE check
  400 JSR help
  410 LDX #255
  420  details
```

Listing 6 1 continued

```
430 INX
440 LDA command,X
450 BEQ donecommand
460 JSR &FFE3
470 BRA details
480  donecommand
490 JSR osnewl
500 BRA restore
510 \
520  check
530 LDX ≠255
540 DEY
550  again
560 INX
570 INY
580 LDA (comline),Y
590 AND ≠&DF
600 CMP com,X
610 BEQ again
620 LDA com,X
630 CMP ≠&FE
640 BEQ mine
650  restore
660 PLY
670 PLX
680 PLA
690  nothelp
700 RTS
710 \
720  help
730 JSR osnewl
740 LDX ≠&FF
750 JSR helploop
760 JSR helploop
770 JSR osnewl
780 RTS
790 \
800  helploop
810 INX
820 LDA title,X
830 BEQ done
840 JSR &FFE3
850 BRA helploop
860  done
870 RTS
880  mine
890 JSR help
900 LDX ≠255
```

Listing 6 1 continued

```
  91Ø  more
  92Ø INX
  93Ø LDA lists,X
  94Ø BMI alldone
  95Ø JSR &FFE3
  96Ø BRA more
  97Ø  alldone
  98Ø PLY
  99Ø PLX
 1ØØØ PLA
 1Ø1Ø LDA ≠Ø
 1Ø2Ø RTS
 1Ø3Ø  com
 1Ø4Ø EQUS"OSBYTE"
 1Ø5Ø EQUB &FE
 1Ø6Ø  command
 1Ø7Ø EQUS"  Osbyte
 1Ø8Ø EQUB Ø
 1Ø9Ø  lists
 11ØØ EQUS "  A=&64, X=byte for conversi
on"
 111Ø EQUB 13
 112Ø EQUS "  On completion   X=hi, Y=lo
"
 113Ø EQUB 13
 114Ø EQUB &FF
 115Ø \
 116Ø  osbyte
 117Ø PHA
 118Ø PHX
 119Ø PHY
 12ØØ LDA &EF
 121Ø CMP ≠&64
 122Ø BEQ yes64
 123Ø JMP restore
 124Ø \
 125Ø  yes64
 126Ø LDA &FØ
 127Ø PHA
 128Ø LSR A
 129Ø LSR A
 13ØØ LSR A
 131Ø LSR A
 132Ø JSR convert
 133Ø STA &FØ
 134Ø PLA
 135Ø JSR convert
 136Ø STA &F1
```

Listing 6 1 continued

```
 1370 PLY
 1380 PLX
 1390 PLA
 1400 LDY &F1
 1410 LDX &F0
 1420 LDA ≠0
 1430 RTS
 1440 \
 1450  convert
 1460 AND ≠15
 1470 CMP ≠10
 1480 BCC over
 1490 ADC ≠6
 1500  over
 1510 ADC ≠48
 1520 RTS
 1530 ]
 1540 NEXT
 1550 ENDPROC
 1560
 1570 DEF PROCchecksum
 1580 N%=0
 1590 FOR X%=&5000 TO &5127
 1600 N%=N%+?X%
 1610 NEXT
 1620 IF N%=33496 THEN ENDPROC
 1630 VDU 7
 1640 PRINT"Assembler error"
 1650 STOP
```

Listing 6 2 Tutorial and test for new OSBYTE Save as OSBTEST

```
   10 REM OSBYTE Tutorial
   20 REM (C) Bruce Smith June 1986
   30 REM Advanced SRAM Guide
   40
   50 MODE 7
   60 REPEAT
   70 REPEAT
   80 INPUT "Enter number in range 1-255
  'X%
   90 UNTIL X%>0 AND X%<256
  100 A%=&64   Y%=0
  110 R%=USR(&FFF4)
  120 X%=(R% AND &FF00) DIV &FF
  130 Y%=(R% AND &FFFF00) DIV &FFFF
```

Listing 6 2 continued

```
  14Ø PRINT In hex that is  " CHR$(X%) C
HR$(Y%)
  15Ø PRINT
  16Ø UNTILØ
```

Listing 6 3 Creates OSWORD &65 to convert and print binary numbers as ASCII hex string Save as OSWORD Can be built up from listing 4 3 (HELP3)

```
   1Ø REM Implement OSWORD
   2Ø REM (C) Bruce Smith June 1986
   3Ø REM Advanced SRAM Guide
   4Ø
   5Ø PROCassemble
   55 PROCchecksum
   6Ø *SRWRITE 5ØØØ +2ØØ 8ØØØ 6
   7Ø END
   8Ø DEF PROCassemble
   9Ø osnewl=&FFE7   osasci=&FFE3
  1ØØ comline=&F2
  11Ø FOR pass=4 TO 7 STEP 3
  12Ø P%=&8ØØØ   O%=&5ØØØ
  13Ø [
  14Ø OPT pass
  15Ø EQUB Ø
  16Ø EQUW Ø
  17Ø JMP service
  18Ø EQUB &82
  19Ø EQUB offset MOD 256
  2ØØ EQUB 1
  21Ø  title
  22Ø EQUS  Osword Extension ROM"
  23Ø EQUB Ø
  24Ø  version
  25Ø EQUS ' 1 ØØ"
  26Ø EQUB Ø
  27Ø  offset
  28Ø EQUB Ø
  29Ø EQUS "(C) Bruce Smith'
  3ØØ EQUB Ø
  31Ø  service
  311 CMP ≠8
  312 BNE noteight
  313 JMP osword
  314  noteight
  32Ø CMP ≠9
  33Ø BNE nothelp
```

Listing 6 3 continued

```
  340 PHA
  350 PHX
  360 PHY
  370 LDA (comline),Y
  380 CMP #13
  390 BNE check
  400 JSR help
  410 LDX #255
  420  details
  430 INX
  440 LDA command,X
  450 BEQ donecommand
  460 JSR &FFE3
  470 BRA details
  480  donecommand
  490 JSR osnewl
  500 BRA restore
  510 \
  520  check
  530 LDX #255
  540 DEY
  550  again
  560 INX
  570 INY
  580 LDA (comline),Y
  590 AND #&DF
  600 CMP com,X
  610 BEQ again
  620 LDA com,X
  630 CMP #&FE
  640 BEQ mine
  650  restore
  660 PLY
  670 PLX
  680 PLA
  690  nothelp
  700 RTS
  710 \
  720  help
  730 JSR osnewl
  740 LDX #&FF
  750 JSR helploop
  760 JSR helploop
  770 JSR osnewl
  780 RTS
  790 \
  800  helploop
  810 INX
```

Listing 6 3 continued

```
  820 LDA title,X
  830 BEQ done
  840 JSR &FFE3
  850 BRA helploop
  860  done
  870 RTS
  880  mine
  890 JSR help
  900 LDX #255
  910  more
  920 INX
  930 LDA lists,X
  940 BMI alldone
  950 JSR &FFE3
  960 BRA more
  970  alldone
  980 PLY
  990 PLX
 1000 PLA
 1010 LDA #0
 1020 RTS
 1030  com
 1040 EQUS"OSWORD"
 1050 EQUB &FE
 1060  command
 1070 EQUS'  Osword'
 1080 EQUB 0
 1090  lists
 1100 EQUS "  A=&65, XY+0=action byte"
 1110 EQUB 13
 1120 EQUS "  XY+0=1      Hex to ASCII'
 1130 EQUB 13
 1133 EQUS "  XY+0=129    Print 1
 1134 EQUB 13
 1140 EQUB &FF
 1150 \
 1160  osword
 1170 PHA
 1180 PHX
 1190 PHY
 1200 LDA &EF
 1210 CMP #&65
 1220 BEQ yes65
 1230 JMP restore
 1240  yes65
 1250 \
 1260 LDY#0
 1270 LDA (&F0),Y
```

Lısting 6 3 contınued

```
 1280 AND ≠&7F
 1290 CMP ≠1
 1300 BEQ hexascı
 1310 JMP restore
 1320 \
 1330  hexascı
 1340 INY
 1350 LDA (&FØ),Y
 1360 PHA
 1370 INY
 1380 LDA (&FØ),Y
 1390 JSR hexconvert
 1400 PLA
 1410 JSR hexconvert
 1420 LDY≠Ø
 1430 LDA (&FØ),Y
 1440 BMI dısplay
 1450 JMP alldone
 1460 \
 1470  dısplay
 1480 LDY≠3
 1490  prınt
 1500 LDA (&FØ),Y
 1510 JSR osascı
 1520 INY
 1530 CPY ≠7
 1540 BNE prınt
 1550 JMP alldone
 1560 \
 1570  hexconvert
 1580 INY
 1590 PHA
 1600 LSR A
 1610 LSR A
 1620 LSR A
 1630 LSR A
 1640 JSR fırst
 1650 STA (&FØ),Y
 1660 INY
 1670 PLA
 1680 JSR fırst
 1690 STA (&FØ),Y
 1700 RTS
 1710 \
 1720  fırst
 1730 AND ≠15
 1740 CMP ≠10
 1750 BCC over
```

Listing 6 3 continued

```
 1760 ADC ≠6
 1770  over
 1780 ADC ≠48
 1790 RTS
 1800 ]
 1810 NEXT
 1820 ENDPROC
 1830
 1840 DEF PROCchecksum
 1850 N%=0
 1860 FOR X%=&5000 TO &515E
 1870 N%=N%+?X%
 1880 NEXT
 1890 IF N%=38567 THEN ENDPROC
 1900 VDU 7
 1910 PRINT"Assembler error'
 1920 STOP
```

Listing 6 4 Tutorial and test for new OSWORD call
Save as OSWTEST

```
   10 REM OSWORD Tutorial
   20 REM (C) Bruce Smith June 1986
   30 REM Advanced SRAM Guide
   40
   50 MODE 7
   60 REPEAT
   70 REPEAT
   80 INPUT "Enter number in range 1-655
35  "R%
   90 UNTIL R%>0 AND R%<65536
  100 A%=&65   Y%=0   X%=&70
  110 ?&70=129
  120 ?&71=R% MOD 256
  130 ?&72=R% DIV 256
  140 PRINT"In hex that is  ",
  150 CALL &FFF1
  160 PRINT
  170 UNTIL0
```

Chapter Seven
Extended Vectors

When designing ROM images you might need to add facilities that are automatically accessed by the MOS as and when required For example, when an error occurs during program operation the Master will print out an error message, it would be nice however to add a patch that would call the routine in your sideways ROM to print out the erroneous line and perhaps even highlight the error itself

Similarly, filing system ROMs, such as disc filing systems, must be accessible from the calling program or command However, calls cannot be made directly by a simple JSR command as the coding is held within a sideways ROM What needs to take place is for the current ROM to be switched out and for the new one to be switched in then the JMP or JSR can be carried out This sounds, and indeed would be, a convoluted and difficult job as it would mean keeping a machine code patch in RAM to handle all the switching

A mechanism exists in the MOS called 'extended vector entry' This allows the main operating system vectors to point into the MOS and tell it to hand control over to another ROM It will also handle transfer of control from the current ROM to a routine in another ROM and then switch back to the original ROM

Twenty seven such vectors are implemented on the Master and each is allocated a number such that the physical address of the vector is located at

```
    &2ØØ+2*<vector number>
```

Table 7 I lists all the information you'll need so you

don't have to worry about having to calculate anything yourself

To redirect a vector we need to point the vector itself into a part of the MOS called the 'extended vector processing area' The vector must have the following address placed into it

```
&FF00+3*<vector number>
```

Table 7 1 lists the actual entry points each vector must be made to point to so it will be processed correctly

Vector	Location	Entry point	Offset
USERV	&200	&FF00	0-2
BRKV	&202	&FF03	3-5
IRQ1V	&204	&FF06	6-8
IRQ2V	&206	&FF09	9-11
CLIV	&208	&FF0C	12-14
BYTEV	&20A	&FF0F	15-17
WORDV	&20C	&FF12	18-20
WRCHV	&20E	&FF15	21-23
RDCHV	&210	&FF18	24-26
FILEV	&212	&FF1B	27-29
ARGSV	&214	&FF1E	30-32
BGETV	&216	&FF21	33-35
BPUTV	&218	&FF24	36-38
GBPBV	&21A	&FF27	39-41
FINDV	&21C	&FF2A	42-44
FSCV	&21E	&FF2D	45-47
EVENTV	&220	&FF30	48-50
UPTV	&222	&FF33	51-53
NETV	&224	&FF36	54-56
VDUV	&226	&FF39	57-59
KEYV	&228	&FF3C	60-62
INSV	&22A	&FF3F	63-65
REMV	&22C	&FF42	66-68
CNPV	&22E	&FF45	69-71
IND1V	&230	&FF48	72-74
IND2V	&232	&FF4B	75-77
IND3V	&234	&FF4E	78-80

Table 7 1 Extended vectors

As table 7 1 shows, each extended vector entry point is offset by three bytes from the next, to allow the instruction JSR &FF51 to be assembled (NB MOS3 0 may vary on other MOS's) This address marks the start of the extended vector processing coding

The MOS needs to know which ROM to page in and which address to call in it Space is provided in RAM for this information in an area called the 'extended vector space', the start address of which is ascertained by issuing an OSBYTE call as follows

```
    LDA ≠&A8
    LDX ≠&ØØ
    LDY ≠&FF
    JSR osbyte
```

This call will return with the start address of the extended vector space in the index registers In MOS3 Ø this address is &D9F

Each vector is allocated three bytes in the extended vector space, and the bytes corresponding to each are found by calculating

```
    Vector space+3*<vector number>
```

These bytes must have the following information poked into them

1 Low byte of address in ROM
2 High byte of address in ROM
3 ROM number - copied from &F4

<u>Working example</u>

All of what went on above may have seemed long-winded, but as the following example of an extended vector proves it really is straightforward - all you need to do is to refer to table 7 1

The USERV is located at &2ØØ and &2Ø1 This is normally associated with the two commands *LINE and *CODE These are implemented solely to allow you to add customised routines to the Master Try executing either of these now type in *LINE or *CODE and you will get the 'Bad command' error message This is because they have not been assigned a task at present and currently point directly to the error message

We can change this In this worked example we'll get both of these commands to produce a bleep when called Again, the action is not spectacular but it is kept simple so that you can concentrate on the extended vector coding

The first task is to save the current vector contents as we will either need to restore them or jump onto the vector once we have finished with it Here they are stored at &8E and &8F Looking at table 7 1 we see that the USRV is located at &2ØØ and &2Ø1, so

```
    LDA &2ØØ         \ get low byte
    STA &8E          \ save it
    LDA &2Ø1         \ get high byte
    STA &8F          \ save it
```

Remember to preserve the low byte first so that you can use &8E as a vector to jump through

The next step is to point the vector to the extended vector processing code Again looking at table 7 1 we can see that this address is at &FFØØ This is placed in the vector thus

```
    LDA ≠Ø           \ low byte of &FFØØ
    STA &2ØØ
    LDA ≠&FF         \ high byte of &FFØØ
    STA &2Ø1
```

The final action required is to place the address of the new vector routine and the identity number of the ROM it is contained in within the correct three bytes in the extended vector space The extended vector space starts at &D9F in MOS3 Ø Looking at table 7 1 we see that the offset into this area is at Ø,1 and 2 Assuming the new USERV routine is at 'new' we get

```
    LDA ≠new MOD 256     \ calculate low byte address
    STA &D9F             \ save it
    LDA ≠new DIV 256     \ calculate high byte address
    STA &D9F+1           \ save it
    LDA &F4              \ get ROM identity
    STA &D9F+2           \ save it
```

Location &F4 always contains the identity number of the currently selected ROM And that's all there is to it!

If you need to reset the vector to its original address simply transfer the contents of &8E and &8F back to &2ØØ and &2Ø1 - there's no need to reset the extended vector space

```
    LDA &8E
    STA &2ØØ             \ reset low byte
    LDA &8F
    STA &2Ø1             \ reset high byte
```

Obviously it is important not to alter the contents of &8E and &8F in any way

Listing 7 1 puts all this into action Once run and initialised the commands *ON and *OFF turn the extended vector on and off respectively Typing *ON and entering *LINE or *CODE will cause a beep to be made Typing *OFF will mean you get the error message 'Bad command' when you use either command

If you have listing 5 1 ('INTERP') to hand you can use this as the basis for this new listing ('VECTOR') To do this make the following alterations

Change lines	1Ø,25Ø,112Ø,115Ø,118Ø,12ØØ,126Ø 169Ø to 177Ø inclusive, 18ØØ, 181Ø,182Ø,184Ø
Delete lines	194Ø to 2Ø5Ø inclusive
Add new lines	194Ø onwards

Note that when you are changing the contents of a vector in this way you should always disable interrupts first with SEI and enable them with CLI after the change This is to prevent the vector being used while it is being changed which could be disastrous if only one byte had been changed at the time! It is also a good idea to save the status register on the stack at this time as well for later restoration Hence use

```
    PHP
    SEI
```

prior to change, and

```
    CLI
    PLP
```

after revectoring

Listing 7 1 How to use extended vector to point into a sideways ROM Save as VECTOR Based on listing 5 1 (INTERP)

```
  10 REM Extended Vector ROM
  20 REM (C) Bruce Smith June 1986
  30 REM Advanced SRAM Guide
  40
  50 PROCassemble
  60 PROCchecksum
  70 *SRWRITE 5000 +500 8000 6
  80 END
  90
 100 DEF PROCassemble
 110 osnewl=&FFE7
 120 osasci=&FFE3
 130 comline=&F2
 140 FOR pass=4 TO 7 STEP 3
 150 P%=&8000   O%=&5000
 160 [
 170 OPT pass
 180 EQUB 0
 190 EQUW 0
 200 JMP service
 210 EQUB &82
 220 EQUB offset MOD 256
 230 EQUB 1
 240  title
 250 EQUS "Extended Vector ROM"
 260 EQUB 0
 270  version
 280 EQUS " 1 00"
 290 EQUB 0
 300  offset
 310 EQUB 0
 320 EQUS "(C) Bruce Smith"
 330 EQUB 0
 340  service
 350 PHA
 360 PHX
 370 PHY
 380 CMP ≠9
 390 BNE nothelp
 400 LDA (comline),Y
 410 CMP ≠13
 420 BNE check
 430 JSR help
 440 LDX ≠255
 450  details
 460 INX
```

Listing 7 1 continued

```
47Ø LDA command,X
48Ø BEQ donecommand
49Ø JSR &FFE3
5ØØ BRA details
51Ø  donecommand
52Ø JSR osnewl
53Ø BRA restore
54Ø \
55Ø  check
56Ø LDX ≠255
57Ø DEY
58Ø  again
59Ø INX
6ØØ INY
61Ø LDA (comline),Y
62Ø AND ≠&DF
63Ø CMP com,X
64Ø BEQ again
65Ø LDA com,X
66Ø CMP ≠&FE
67Ø BEQ mine
68Ø  restore
69Ø PLY
7ØØ PLX
71Ø PLA
72Ø RTS
73Ø \
74Ø  nothelp
75Ø CMP ≠4
76Ø BEQ unrecognised
77Ø BRA alldone
78Ø \
79Ø  help
8ØØ JSR osnewl
81Ø LDX ≠&FF
82Ø JSR helploop
83Ø JSR helploop
84Ø JSR osnewl
85Ø RTS
86Ø \
87Ø  helploop
88Ø INX
89Ø LDA title,X
9ØØ BEQ done
91Ø JSR &FFE3
92Ø BRA helploop
93Ø  done
94Ø RTS
```

Listing 7 1 continued

```
 950 \
 960  mine
 970 JSR help
 980 LDX ≠255
 990  more
1000 INX
1010 LDA lists,X
1020 BMI alldone
1030 JSR &FFE3
1040 BRA more
1050 \
1060  alldone
1070 PLY
1080 PLX
1090 PLA
1100 RTS
1110 \
1120  com
1130 EQUS"VECTORS"
1140 EQUB &FE
1150  command
1160 EQUS"  Vectors"
1170 EQUB 0
1180  lists
1190 EQUS "  Redirects USERV at &200"
1200 EQUB 13
1210 EQUS    ON"
1220 EQUB 13
1230 EQUS "  OFF"
1240 EQUB 13
1250 EQUB &FF
1260 \
1270  unrecognised
1280 LDX ≠255
1290 DEY
1300 PHY
1310  identify
1320 INX
1330 INY
1340 LDA (comline),Y
1350 AND ≠&DF
1360 CMP comtable,X
1370 BEQ identify
1380 LDA comtable,X
1390 BMI address
1400 \
1410  moveon
1420 INX
```

Listing 7.1 continued

```
 1430 LDA comtable,X
 1440 BPL moveon
 1450 BNE notend
 1460 PLY
 1470 BRA alldone
 1480 \
 1490  notend
 1500 INX
 1510 PLY
 1520 PHY
 1530 JMP identify
 1540 \
 1550  address
 1560 CMP ≠&FF
 1570 BNE notFF
 1580 PLY
 1590 BRA alldone
 1600  notFF
 1610 \
 1620 STA &39
 1630 INX
 1640 LDA comtable,X
 1650 STA &38
 1660 JMP (&38)
 1670 \
 1680  comtable
 1690 EQUS "ON"
 1700 EQUB on DIV 256
 1710 EQUB on MOD 256
 1720 EQUS "OFF"
 1730 EQUB off DIV 256
 1740 EQUB off MOD 256
 1750
 1760
 1770
 1780 EQUB &FF
 1790 \
 1800
 1810
 1820
 1830
 1840
 1850 \
 1860  found
 1870 PLY
 1880 PLY
 1890 PLX
 1900 PLA
```

Listing 7 1 continued

```
1910 LDA #0
1920 RTS
1930 \
1940  on
1950 LDA &200
1960 STA &8E
1970 LDA &201
1980 STA &8F
1990 LDA #0
2000 STA &200
2010 LDA #&FF
2020 STA &201
2030 LDA #new MOD 256
2040 STA &D9F
2050 LDA #new DIV 256
2060 STA &D9F+1
2070 LDA &F4
2080 STA &D9F+2
2090 CLI
2100 JMP found
2110 \
2120  new
2130 LDA #7
2140 JSR &FFEE
2150 RTS
2160 \
2170  off
2180 LDA &8E
2190 STA &200
2200 LDA &8F
2210 STA &201
2220 JMP found
2230 ]
2240 NEXT
2250 ENDPROC
2260
2270 DEF PROCchecksum
2280 N%=0
2290 FOR X%=&5000 TO &515D
2300 N%=N%+?X%
2310 NEXT
2320 IF N%=40201 THEN ENDPROC
2330 VDU 7
2340 PRINT"Assembler error'"
2350 STOP
```

Chapter Eight
Pot Pourri

So far it's been quite easy to demonstrate the service calls, but some are not that simple For example, service call 12 allows a ROM to claim the non-maskable interrupts (NMIs) which effectively control the operation of the Master Filing systems such as net and disc are typical cases where you would want to claim NMIs - but space doesn't allow me to include a filing system program! In this chapter we will have a general look at routines of this type, and briefly examine writing a filing system Calls not covered here will be dealt with in greater depth in chapters to come

Service Call 5

This call is issued by the MOS when an interrupt request (IRQ) that it does not recognise occurs If your paged ROM makes use of the IRQ line then it should be directed to a suitable interrupt request polling routine and check any device(s) to find the source of the IRQ Any ROM recognising the interrupt should process it, set the accumulator to zero to indicate that the interrupt has been serviced, and then return with an RTS instruction and NOT an RTI as is generally the norm when an IRQ has occurred The following code shows how a suitable polling routine might be instigated

```
    CMP #5         \ is it unrecognised IRQ?
    BNE next       \ branch to next test if not
    JSR polling    \ execute IRQ polling
    BCC notfound   \ carry clear if IRQ not identified
    PLA           \ pull stack to balance previous push
    LDA ≠Ø        \ call serviced
    RTS           \ return
     notfound
    PLA           \ put service code in accumulator
    RTS           \ and return
```

As usual, this code assumes that on entry the service call type was preserved on the stack If an IRQ is not identified by any of the ROMs then the MOS directs a final call through the user vector IRQ2

Service Call 6

This service call is used to inform paged ROMs that a BRK has occurred, before the MOS hands control over to the current language ROM via the BRK vector (BRKV), to process the BRK In many instances the BRK is used to signal an error, and print an error message If the service call does not intend to process the BRK, for example to produce some extra fancy error messages and pointers, the contents of all registers should be preserved The vectored address at &FD and &FE points to the error number in memory while the byte at &FØ contains the value of the hardware stack pointer after the BRK was executed As the BRK may have occurred in a ROM other than the one currently processing the BRK, it is important to be able to ascertain in which ROM it did occur To do this OSBYTE &BA should be called, and the ROM number is returned in the X register As always if your ROM traps this call it should load the accumulator with Ø before returning to the MOS, otherwise the service call number should be preserved so that the MOS can poll other ROMs

Service Call 11

The major filing and networking systems on the BBC micros make extensive use of non-maskable interrupts (NMIs) This call should be used when the ROM system currently using it (ie a DFS ROM) no longer needs to do so and is prepared to release it When this service call is issued the Y register holds the number of the ROM that was using the NMIs before the claim for them was made Each ROM that recognises this call should check the contents of the Y register with its own ROM number, available in the X register and location &F4 at the time of the service If it is the same then the accumulator should hold zero on return from the service call, otherwise all registers should be preserved The following coding shows the general procedure for handling this call if the ROM is making use of the NMIs

```
    CMP ≠12          \ is it NMI release?
    BNE next         \ branch to next if not
    TYA              \ move last ROM number into
                     \ accumulator
    CMP &F4          \ is it this ROM?
    BNE notme        \ branch if not
    PLA              \ balance previous push
    LDA ≠0           \ recognised code
    RTS              \ and return
     notme
    PLA              \ restore service call number
    RTS
```

To return the use of the NMI to its previous user OSBYTE 143 should be executed as follows

```
    LDA ≠143         \ OSBYTE code
    LDX ≠11          \ service call code
    LDY ≠255
    JSR OSBYTE       \ issue request
```

As mentioned before, service call 11 will normally only be trapped if the user is implementing a filing system Care must be taken when using NMIs as all BBC micros are interrupt-driven and funny things can happen to general housekeeping chores if they are not treated with respect Note also that the zero page locations associated with the filing system should not be used during this service call's execution

<u>Service Call 12</u>

This call is issued to claim use of the NMIs It is called with OSBYTE 143 as follows

```
    LDA ≠143         \ OSBYTE code
    LDX ≠12          \ service call 12
    LDY ≠255
    JSR OSBYTE       \ perform call
```

The ROM that currently has claim of the NMIs should place its ROM number into the Y register and store any important data in its own private storage area (covered shortly) It should also inhibit any further use of NMIs until it has reclaimed the NMI at a later stage The claiming ROM should also store the ROM number of the current NMI for use when releasing the NMI claim If the ROM was not using the NMIs, then the Y register must remain unaltered Once claimed the zero page area associated with NMI handling from &AØ to &A7 is free for use and the ROM's resident NMI service routine should be copied to &DØØ

Service Call I5

Whenever a new filing system is initialised it must perform a number of operations One of these is to repoint all vectors into the coding of the new filing system After writing relevant addresses into the various filing system vectors, service call I5 should be issued by the new filing system, to inform all other paged software that a change in filing system has taken place

Service Call I6

This call is performed to inform paged ROMs that may be using SPOOL or EXEC files that a filing system change is being performed On receipt of this call any ROM using such files should perform any required housekeeping to tidy things up If a ROM wants a SPOOL or EXEC file to remain open, a zero should be placed in the accumulator before releasing the service call

Service Call I7

This call is issued when the character font, ie the user definable character set, is about to explode or implode On the Master of course the character font is permanently exploded, so checking for a pending explosion is not important - unless you also wish your code to work on a BBC micro Checking for an implosion may be of use however - perhaps in conjunction with the Font ROM listing given earlier This service call is therefore issued when the MOS encounters OSBYTE &I5 (*FX2Ø)

Service Call I8

This call is provided to allow filing systems to be initialised without having to issue any operating system commands This is important as a program may need to have files open in two or more filing systems The filing system should check the contents of the Y register to see if they agree with its operating system filing system code as defined by OSARGS If the ROM identifies as the called filing system it should initialise itself and restore all the files that were open at the time it was previously shut down

Service Call 21

This call allows you to process software-generated interrupts It is issued 100 times every second after an OSBYTE &16 (*FX22) command has been used On receipt of an OSBYTE &17 (*FX23) the MOS stops issuing the service call The call is effectively an interrupt polling routine Its main use would be for control of peripherals, and on receipt your ROM should check its hardware accordingly Of course it can be used for other things as the next example shows!

Listing 8 1 will begin to increment a two-byte counter after *FX22 has been issued It will stop incrementing the counter, held at &70 and &71, after *FX23 The code requires nothing other than to catch the service call and direct it to the polling routine You can use listing 4 3 from Chapter 4 (saved as 'HELP3') as the base for the program and make the following changes

Change lines	10, 220, 1040, 1070,1100,1120
Add lines	55, 311, 312, 313, 314, 1150 to 1370 inclusive

To see the program in action, type *FX22 and enter this one-liner

```
REPEAT P !&70 AND &FFFF UNTIL 0
```

Pressing ESCAPE and typing *FX23 will stop the interrupt polling

Obviously the ROM processing this call will need to know if it should return with A=0 to terminate this call, however other ROMs may also be using this call to operate their own routines and as such the accumulator should return with &15 to ensure the MOS passes the service call onto other ROMs To this end the Y register will always hold a number which should be decremented by the number of claims made by your ROM For example if you previously claimed this call twice to poll devices, then the Y register should be decremented twice on completion of the two polling routines If Y is 0 then the accumulator should return holding 0 else it should be re-set with the service call number

Service Call 37

This call should be trapped by filing system ROMs only It is issued as a request for information about the filing systems installed in the micro If your filing system ROM receives this call it should supply 11 bytes of information as follows

8 bytes	Filing system name padded out with spaces
1 byte	Lowest handle number used
1 byte	Highest handle number used
1 byte	The filing system number

This information should be stored at the address pointed to by the vector at &F2 and &F3 Y should be set to zero and will end up incremented by 11

```
 service37
LDY #0
 nextbyte
LDA info,Y              \ get information byte
STA (&F2),Y             \ write byte
INY                     \ increment index
CPY #11                 \ 11 bytes done?
BNE nextbyte            \ if not continue
```

As an example, the ADFS would return the following on receipt of this service call

Decimal	ASCII
96	a
100	d
102	f
115	s
32	<space>
32	<space>
32	<space>
32	<space>
45	Lowest handle used
57	Highest handle used
8	Filing system number

Service Call 38

This is issued to all ROMs when *SHUT is used

Service Call 39

Call 39 is issued on a hard reset, ie when the micro is switched on or after CTRL-BREAK If you run the trace program (listing 3 1 in Chapter 3), you will notice that it is issued immediately prior to the initialisation of the Acorn DFS, and obviously directly after the MOS has initialised Intercepting this call will allow you to initialise your own ROM as needs be Program 8 2 does just that and prints the date onto the screen It does this by intercepting call 39 (&27), reads the CMOS clock with OSWORD &E, and then prints just the date onto the screen It uses memory from &7Ø for the OSWORD parameter block and you could print the time as well simply by extending the loop from line 129Ø You can use listing 4 3 as the base for the program and adapt as follows

Change lines	1Ø, 22Ø, 1Ø4Ø, 1Ø7Ø,11ØØ,112Ø
Add lines	55, 311, 312, 313, 314, 115Ø to 146Ø inclusive

Service Call 254

If the Tube interface is present this service call is issued after OSHWM has been defined, to see if it is active If it is then service call 255 will be issued

Service Call 255

This call is issued if a co-processor or second processor is active It is issued prior to final setting of the OSHWM in the co- or second processor, thus allowing languages and start-up messages to be copied

About Filing Systems

A filing system ROM such as the Disc Filing System (DFS), Advanced DFS (ADFS) or Network Filing System (NFS) can be selected in a number of ways The most obvious of these is via a MOS command such as *DISC The MOS will issue service call 4 here and the DFS ROM will recognise the command and muscle its way in as the active filing system as described below

A filing system may be selected in two other ways however, namely through service call 18 with the Y register containing the filing system number, or via service call 3 where a special recognised key is pressed in conjunction with the BREAK key, eg D-BREAK for Disc, N-BREAK for Net and so on

Because a filing system will never be required to run directly in either a second or co-processor it should always be written in the native code of the 65Ø2-series microprocessor in the BBC micro or Master

All filing systems must be capable of responding to the following service calls

1,2,3,4,9,10,15,16,18,33,34,36,37,38 and 39

When a filing system recognises that it has been selected it should initialise itself in the following stages

First, call OSFSC with A=6 thus enabling the outgoing filing system to tidy up shop and shut itself down This MOS call does not have a recognised vector entry point so it should be called in a slightly convoluted way to allow control to be restored to the point after the JMP, this is done by performing a JSR to a place which contains the instruction, JMP(&21E), ie

```
JSR dofsc
xx
xx
 dofsc JMP (&21E)
```

Second, set up the extended vectors required by the filing system

Next, issue service call 15 to inform other ROMs that the filing system vectors have been altered

Finally, restore any files that remain open since the filing system was last in use

In addition to its own static workspace claimed, the filing system has some other exclusive memory locations free for use These are

&A0 to &A7	The NMI workspace - available when filing system has claimed NMIs
&A8 to &AF	* workspace - for when * commands are issued
&B0 to &BF	Temporary workspace Contents may change between commands
&C0 to &CF	Private workspace Contents here do not change between commands if filing system does not change
&D00 to &D5F	NMI service code space If ROM uses NMIs the service code for them should be loaded here (claim with service call 12 first)

Listing 8 1 Traps service call 21 which polls ROMs 100 times a second Save as POLLING Adapt from listing 4 3 (HELP3)

```
   10 REM Polling ROM
   20 REM (C) Bruce Smith June 1986
   30 REM Advanced SRAM Guide
   40
   50 PROCassemble
   55 PROCchecksum
   60 *SRWRITE 5000 +200 8000 7
   70 END
   80 DEF PROCassemble
   90 osnewl=&FFE7
  100 comline=&F2
  110 FOR pass=4 TO 7 STEP 3
  120 P%=&8000   O%=&5000
  130 [
  140 OPT pass
  150 EQUB 0
  160 EQUW 0
  170 JMP service
  180 EQUB &82
  190 EQUB offset MOD 256
  200 EQUB 1
  210  title
  220 EQUS "Polling Interrupt ROM"
  230 EQUB 0
  240  version
  250 EQUS " 1 00"
  260 EQUB 0
  270  offset
  280 EQUB 0
  290 EQUS "(C) Bruce Smith"
  300 EQUB 0
  310  service
  311 CMP ≠&15
  312 BNE tryhelp
  313 JMP time
  314  tryhelp
  320 CMP ≠9
  330 BNE nothelp
  340 PHA
  350 PHX
  360 PHY
  370 LDA (comline),Y
  380 CMP ≠13
  390 BNE check
  400 JSR help
  410 LDX ≠255
```

Listing 8 1 continued

```
420  details
430 INX
440 LDA command,X
450 BEQ donecommand
460 JSR &FFE3
470 BRA details
480  donecommand
490 JSR osnewl
500 BRA restore
510 \
520  check
530 LDX ≠255
540 DEY
550  again
560 INX
570 INY
580 LDA (comline),Y
590 AND ≠&DF
600 CMP com,X
610 BEQ again
620 LDA com,X
630 CMP ≠&FE
640 BEQ mine
650  restore
660 PLY
670 PLX
680 PLA
690  nothelp
700 RTS
710 \
720  help
730 JSR osnewl
740 LDX ≠&FF
750 JSR helploop
760 JSR helploop
770 JSR osnewl
780 RTS
790 \
800  helploop
810 INX
820 LDA title,X
830 BEQ done
840 JSR &FFE3
850 BRA helploop
860  done
870 RTS
880  mine
890 JSR help
```

Listing 8 1 continued

```
  900 LDX ≠255
  910  more
  920 INX
  930 LDA lists,X
  940 BMI alldone
  950 JSR &FFE3
  960 BRA more
  970  alldone
  980 PLY
  990 PLX
 1000 PLA
 1010 LDA ≠0
 1020 RTS
 1030  com
 1040 EQUS'POLLING"
 1050 EQUB &FE
 1060  command
 1070 EQUS"  Polling"
 1080 EQUB 0
 1090  lists
 1100 EQUS "  Start with *FX22   Cancel
with *FX23"
 1110 EQUB 13
 1120 EQUS    Increments a two byte numb
er at &70"
 1130 EQUB 13
 1140 EQUB &FF
 1150 \
 1160  time
 1170 PHA
 1180 PHX
 1190 PHY
 1200 INC &70
 1210 BNE nohigh
 1220 INC &71
 1230  nohigh
 1240 JMP restore
 1250 ]
 1260 NEXT
 1270 ENDPROC
 1280
 1290 DEF PROCchecksum
 1300 N%=0
 1310 FOR X%=&5000 TO &5114
 1320 N%=N%+?X%
 1330 NEXT
 1340 IF N%=30501 THEN ENDPROC
 1350 VDU 7
```

Listing 8 1 continued

```
 1360 PRINT"Assembler error!"
 1370 STOP
```

Listing 8 2 Traps service call 39 to print date on the screen after a hard reset Save as TIME Developed from listing 4 3 (HELP3)

```
   10 REM Date on Reset
   20 REM (C) Bruce Smith June 1986
   30 REM Advanced SRAM Guide
   40
   50 PROCassemble
   55 PROCchecksum
   60 *SRWRITE 5000 +200 8000 6
   70 END
   80 DEF PROCassemble
   90 osnewl=&FFE7
  100 comline=&F2
  110 FOR pass=4 TO 7 STEP 3
  120 P%=&8000   O%=&5000
  130 [
  140 OPT pass
  150 EQUB 0
  160 EQUW 0
  170 JMP service
  180 EQUB &82
  190 EQUB offset MOD 256
  200 EQUB 1
  210  title
  220 EQUS "Date ROM"
  230 EQUB 0
  240  version
  250 EQUS " 1 00
  260 EQUB 0
  270  offset
  280 EQUB 0
  290 EQUS "(C) Bruce Smith"
  300 EQUB 0
  310  service
  311 CMP ≠&27
  312 BNE tryhelp
  313 JMP time
  314  tryhelp
  320 CMP ≠9
  330 BNE nothelp
  340 PHA
  350 PHX
```

Listing 8 2 continued

```
360 PHY
370 LDA (comline),Y
380 CMP ≠13
390 BNE check
400 JSR help
410 LDX ≠255
420  details
430 INX
440 LDA command,X
450 BEQ donecommand
460 JSR &FFE3
470 BRA details
480  donecommand
490 JSR osnewl
500 BRA restore
510 \
520  check
530 LDX ≠255
540 DEY
550  again
560 INX
570 INY
580 LDA (comline),Y
590 AND ≠&DF
600 CMP com,X
610 BEQ again
620 LDA com,X
630 CMP ≠&FE
640 BEQ mine
650  restore
660 PLY
670 PLX
680 PLA
690  nothelp
700 RTS
710 \
720  help
730 JSR osnewl
740 LDX ≠&FF
750 JSR helploop
760 JSR helploop
770 JSR osnewl
780 RTS
790 \
800  helploop
810 INX
820 LDA title,X
830 BEQ done
```

Listing 8 2 continued

```
  840 JSR &FFE3
  850 BRA helploop
  860  done
  870 RTS
  880  mine
  890 JSR help
  900 LDX #255
  910  more
  920 INX
  930 LDA lists,X
  940 BMI alldone
  950 JSR &FFE3
  960 BRA more
  970  alldone
  980 PLY
  990 PLX
 1000 PLA
 1010 LDA #0
 1020 RTS
 1030  com
 1040 EQUS"DATE"
 1050 EQUB &FE
 1060  command
 1070 EQUS"  Date"
 1080 EQUB 0
 1090  lists
 1100 EQUS "  Date is displayed on Reset
"
 1110 EQUB 13
 1120 EQUS "  Time string stored at &70"
 1130 EQUB 13
 1140 EQUB &FF
 1150 \
 1160  time
 1170 PHA
 1180 PHX
 1190 PHY
 1200 LDA #14
 1210 LDX #&70
 1220 LDY #0
 1230 JSR &FFF1
 1240 LDY #0
 1250  date
 1260 LDA &70,Y
 1270 JSR &FFE3
 1280 INY
 1290 CPY #15
 1300 BNE date
```

Listing 8 2 continued

```
1310 JSR &FFE7
1320 JSR &FFE7
1330 JMP restore
1340 ]
1350 NEXT
1360 ENDPROC
1370
1380 DEF PROCchecksum
1390 N%=0
1400 FOR X%=&5000 TO &5103
1410 N%=N%+?X%
1420 NEXT
1430 IF N%=30419 THEN ENDPROC
1440 VDU 7
1450 PRINT"Assembler error!"
1460 STOP
```

Chapter Nine
Configure and Status

The commands *CONFIGURE and *STATUS allow the Master power-up and reset configuration to be defined by writing to certain of the 5Ø bytes of battery-backed CMOS RAM The allocation of these bytes is as follows

Byte numbers	Allocation
Ø to 19	Configuration system
2Ø to 29	Acorn future use
3Ø to 38	Third party ROM use
39 to 49	User memory

Bytes 3Ø to 38 are allocated for specific use of sideways RAM and ROM software

Service calls 4Ø and 41 are provided by the MOS to allow extension of the range of *CONFIGURE and *STATUS options by trapping each call as appropriate In the examples that follow, the date printing on reset routine from the last chapter is extended so it may be switched on and off via the *CONFIGURE command Use of *STATUS will enable the user to read the currently-selected option at any time

Choosing the byte

First let us examine how we go about deciding which of the reserved ROM bytes we can use Well, that's easy -- the choice is already made for us There are only eight bytes available and these correspond directly with the spare number of sideways ROM/RAM slots available For example, ROM slot Ø corresponds with byte number 3Ø, slot 1 with byte number 31 slot 2 with byte number 32 and so on To locate the byte associated with our ROM slot we simply need to find which ROM slot the program is running in and add it to 3Ø, thus

```
    CLC                 \ clear carry read to add
    LDA &F4             \ get ROM slot number
    ADC ≠3Ø             \ add it to 3Ø
```

The byte number is now held in the accumulator Location &F4 always contains a copy of the selected ROM bank

Once we know which byte is ours we need to be able to read and write to it OSBYTE 161 and 162 allow us to do this respectively The accumulator should hold the OSBYTE number and the X register the byte in CMOS RAM to be read from or written to In the case of a read operation the Y register returns the data, in the instance of a write it should contain the data to be written

For example, to read the contents of our byte we proceed as follows

```
    TAX                 \ move byte number into X
    LDA ≠161            \ OSBYTE number
    JSR &FFF4           \ and read byte
```

The Y register now holds the contents of the allocated battery-backed byte

The allocation of a single byte may seem mean, but remember that a single bit can be used to signal an ON/OFF condition Therefore by conservative use your own firmware could provide up to eight new options

It's a Date

The new status/configuration extension chosen here is

```
    DATE ON/OFF
```

Typing

```
    *CONFIGURE DATE ON
```

will cause the date to be printed on the screen whenever a reset is made (ie on CTRL-BREAK) Likewise, typing

```
    *CONFIGURE DATE OFF
```

will disable this action and no date will be printed at

a reset *STATUS DATE will print either ON or OFF depending on the current configuration
The two straight commands *CONFIGURE and *STATUS must also be catered for The former will print

```
    DATE ON/OFF
```

and the latter will print, DATE followed by the current configuration status
Just by defining these objectives we have really already clarified what our new firmware must be capable of and each item can be coded (and tested if your are writing your own specific items) in turn
The first action would be to extend the 'time' routine given in the last chapter so that it first reads the appropriate CMOS battery-backed byte as already described By examining the contents of the Y register the routine can determine whether to print the date If a 0 is returned the date is not printed otherwise it is printed

```
    CPY #0          \ is DATE OFF?
    BNE carryon     \ no, so read and print
    JMP restore     \ yes, so return
```

Extending status

The MOS will issue service call &29 (41) on two occasions First, if a straight *STATUS is encountered This requires that a complete list of all options are printed After printing the configuration status as defined in bytes 0 to 19, the MOS issues the call to allow other ROMs to respond Second, the call is issued if the status command is followed by an unknown option This enables the current ROM to check to see if it is familiar with the option As with other * commands the vector at &F2 is used to point to the first unknown, non-space, character after the *STATUS command The first instance is easily checked, simply test directly for a return character, ie 13 (&0D) and branch to the printing routine thus

```
    LDA (&F2),Y     \ get first non-space character
    CMP #13         \ is it a return?
    BEQ dotime      \ if yes then branch to print
```

If the character is not a return then we need to check this against our own possible status options - just one in this case - in a similar manner to checking for exended help options as described in chapter 3

However, here's the routine to do just that

```
 trytime
DEY
LDX ≠255        \ initialise counters
 loopt
INY
INX             \ increment counters
CPX ≠4          \ only 4 letters in DATE
BEQ ours        \ if here then it must be ours
LDA (&F2),Y     \ get next byte
AND ≠&DF        \ force to upper case
CMP string,X    \ is it the same?
BEQ loopt       \ if yes, try next byte
SEC             \ set carry to signal failure
RTS             \ and return to calling routine
 ours
CLC             \ set carry to signal success
RTS             \ and return
 string
EQUS"DATE"
```

This routine is written in the form of a subroutine, called with JSR trytime, as it will be needed by the configure routine to be discussed later The carry flag is used to indicate whether the command is identified A successful match is indicated by clearing the carry flag, while a failure sets it On return, the carry flag can be tested and the necessary action taken

```
BCC end         \ it's us, so branch to end routine
JMP restore     \ not known, so return to MOS
```

Obviously we now need to find out which option to print, ie 'ON' or 'OFF' This is done by reading the ROM status byte within the CMOS RAM as already described and printing the correct ASCII string

```
 end
CLC
LDA &F4         \ get ROM number
ADC ≠3Ø         \ calculate byte number
TAX             \ move into X register
LDA ≠161        \ OSBYTE number
JSR &FFF4       \ get byte
CPY ≠Ø          \ is it 'off'?
BEQ off         \ yes branch
LDY ≠5          \ no, it's 'on' so get new index
 off
LDA onoff,Y     \ get byte
```

```
    BEQ finished    \ if zero then finished
    JSR &FFE3       \ else print it
    BRA off         \ branch until finished
     onoff
    EQUS"OFF"       \ index Y=Ø
    EQUB 13
    EQUB Ø
    EQUS"ON"        \ index Y=5
    EQUB 13
    EQUB Ø
```

Extending Configure

Service call 4Ø is issued by the MOS whenever it encounters an unknown configure option, such as, *CONFIGURE DATE It is also put out to each ROM when a simple *CONFIGURE is encountered, and in such instances the ROM should print the possible options In both cases the code is not too different from that used above for *STATUS A simple *CONFIGURE is indicated by looking at the next byte and testing for a return character The string to be printed is simply 'DATE" followed by the two possible options separated by a slash character The character strings at 'string' and 'onoff' can be used with the ASCII character for '/' being printed at the appropriate point to give

```
    DATE     OFF/ON
```

The spaces between DATE and ON/OFF are deliberate to keep to the format taken by the MOS options
  To configure DATE we will specify two slightly different command strings

```
    *CONFIGURE DATE ON
    *CONFIGURE DATE OFF
```

Our ROM will receive the call with (&F2),Y pointing to D Our code must therefore test for DATE (using the routine described above - 'trytime') and then, after moving past any spaces, test for ON or OFF

```
     spaces          \ start after DATE
    LDA (&F2),Y      \ get next byte
    CMP #32          \ is it a space?
    BNE none         \ carry on if not
    INY              \ else increment index
    BRA spaces       \ and try again
     none
    AND #&DF         \ force to upper case
    CMP ≠ASC("O")     \ is it O?
    BEQ tryN          \ branch if so
    JMP restore       \ else not us so restore
    INY               \ increment index
    LDA (&F2),Y       \ get next byte
    AND ≠&DF          \ force to upper case
    CMP ≠ASC("N")     \ is it N (for ON)
    BNE tryF          \ no try F
    LDY ≠1            \ get byte for ON
    BRA write         \ and branch to write it there
     tryF
    CMP ≠ASC("F")     \ is it F (for OFF)
    BEQ yesF          \ branch if so
    JMP restore       \ else not us so restore
     yesF
    LDY ≠Ø            \ get OFF flag
```

All that is needed now is to write the relevant byte into the correct byte for future reference The Y register already contains the byte to be written so

```
     write
    CLC               \ clear carry
    LDA &F4           \ get ROM number
    ADC ≠3Ø           \ calculate byte number
    TAX               \ move into X register
    LDA ≠162          \ write code number
    JSR &FFF4         \ and call OSBYTE
```

In both specific *STATUS and *CONFIGURE instances, once the call has been identified and serviced the accumulator should return to the MOS containing zero

The Program

Listing 9 1 puts the above code into practice It uses listing 8 2 ('TIME') as its base, and the changes and additions needed to adapt this are listed below Save the program under the filename 'DATE'

Change lines	1Ø, 6Ø, 22Ø, 32Ø, 1Ø4Ø, 1Ø7Ø, 11ØØ, 112Ø
Add lines	311, 312, 313, 314, 315, 316, 317, 318 319, 1131, 1132, 115Ø to 313Ø inclusive

The Right Byte

In the examples above the numbers Ø and 1 have been loaded directly into the Y register before writing to the battery-backed RAM As already mentioned however we are only using a single bit in the assigned ROM byte and we may wish to use more In such cases it is most important that the status of the other bits in the byte are preserved otherwise we will change options when not wishing to To counteract this make good use of the AND and OR operators to either mask or force bits in the byte Look at the following byte, represented at bit level

```
    1111 1ØØ1
```

Suppose we wish to set bit 2 (third from the right) to a 1 We need to logically OR the byte with

```
    ØØØØ Ø1ØØ
```

to give

```
    1111 11Ø1
```

In assembler this would be

```
    LDA byte        \ get byte, ie 1111 1ØØ1
    ORA ≠4          \ OR with ØØØØ Ø1ØØ
    STA byte        \ save result ie 1111 11Ø1
```

To clear or mask a byte the AND operator can be used Assuming we now wish to clear the same bit we need to AND the byte with

```
    1111 1Ø11
```

Bit 3 is clear and will therefore be masked clear no matter what its original contents Set bits will be preserved as 1's are placed in every other position The assembler is simply

```
    LDA byte        \ get byte, ie 1111 11Ø1
    AND ≠&F7        \ OR with 1111 1Ø11
    STA byte        \ save result ie 1111 1ØØ1
```

<u>Compact Note</u>

The techniques in this chapter are applicable to the Compact and the listings which follow do work However, they rely on the real-time clock which is present in the Master but not the Compact Hence only the default TIME$ will be displayed

Listing 9 1 Adds date display to configure options
Save as DATE Based on listing 8 2 (TIME)

```
  10 REM CONFIG  and *STATUS
  20 REM (C) Bruce Smith June 1986
  30 REM Advanced SRAM Guide
  40
  50 PROCassemble
  55 PROCchecksum
  60 *SRWRITE 5000 +300 8000 6
  70 END
  80 DEF PROCassemble
  90 osnewl=&FFE7
 100 comline=&F2
 110 FOR pass=4 TO 7 STEP 3
 120 P%=&8000   O%=&5000
 130 [
 140 OPT pass
 150 EQUB 0
 160 EQUW 0
 170 JMP service
 180 EQUB &82
 190 EQUB offset MOD 256
 200 EQUB 1
 210  title
 220 EQUS "Configure and Status ROM"
 230 EQUB 0
 240  version
 250 EQUS " 1 00"
 260 EQUB 0
 270  offset
 280 EQUB 0
 290 EQUS "(C) Bruce Smith"
 300 EQUB 0
 310  service
 311 CMP #&27
 312 BNE tryhelp
 313 JMP time
 314  tryhelp
 315 CMP#41 BNE nextry
 316 JMP status
 317  nextry
 318 CMP#40 BNE andnext
 319 JMP configure
 320  andnext CMP #9
 330 BNE nothelp
 340 PHA
 350 PHX
 360 PHY
 370 LDA (comline),Y
```

Listing 9 1 continued

```
380 CMP #13
390 BNE check
400 JSR help
410 LDX #255
420 .details
430 INX
440 LDA command,X
450 BEQ donecommand
460 JSR &FFE3
470 BRA details
480 .donecommand
490 JSR osnewl
500 BRA restore
510 \
520 .check
530 LDX #255
540 DEY
550 .again
560 INX
570 INY
580 LDA (comline),Y
590 AND #&DF
600 CMP com,X
610 BEQ again
620 LDA com,X
630 CMP #&FE
640 BEQ mine
650 .restore
660 PLY
670 PLX
680 PLA
690 .nothelp
700 RTS
710 \
720 .help
730 JSR osnewl
740 LDX #&FF
750 JSR helploop
760 JSR helploop
770 JSR osnewl
780 RTS
790 \
800 .helploop
810 INX
820 LDA title,X
830 BEQ done
840 JSR &FFE3
850 BRA helploop
```

Listing 9 1 continued

```
  860  done
  870 RTS
  880  mine
  890 JSR help
  900 LDX ≠255
  910  more
  920 INX
  930 LDA lists,X
  940 BMI alldone
  950 JSR &FFE3
  960 BRA more
  970  alldone
  980 PLY
  990 PLX
 1000 PLA
 1010 LDA ≠0
 1020 RTS
 1030  com
 1040 EQUS"DATE
 1050 EQUB &FE
 1060  command
 1070 EQUS"  Date"
 1080 EQUB 0
 1090  lists
 1100 EQUS '  *CONFIG  DATE ON"
 1110 EQUB 13
 1120 EQUS    *CONFIG  DATE OFF"
 1130 EQUB 13
 1131 EQUS "  *STATUS DATE"
 1132 EQUB 13
 1140 EQUB &FF
 1150 \
 1160  time
 1170 PHA
 1180 PHX
 1190 PHY
 1200 CLC
 1210 LDA &F4
 1220 ADC ≠30
 1230 TAX
 1240 LDA ≠161
 1250 JSR &FFF4
 1260 CPY ≠0
 1270 BNE carryon
 1280 JMP restore
 1290 \
 1300  carryon
 1310 LDA ≠14
```

Listing 9 1 continued

```
1320 LDX ≠&7Ø
133Ø LDY ≠Ø
134Ø JSR &FFF1
135Ø LDY ≠Ø
136Ø \
137Ø  date
138Ø LDA &7Ø,Y
139Ø JSR &FFE3
14ØØ INY
141Ø CPY ≠15
142Ø BNE date
143Ø JSR &FFE7
144Ø JSR &FFE7
145Ø JMP restore
146Ø \
147Ø  status
148Ø PHA
149Ø PHX
15ØØ PHY
151Ø LDA (&F2),Y
152Ø CMP ≠13
153Ø BEQ dotime
154Ø JSR trytime
155Ø BCC end
156Ø JMP restore
157Ø \
158Ø  itstime
159Ø  dotime
16ØØ LDX ≠255
161Ø  timeloop
162Ø INX
163Ø LDA string,X
164Ø BEQ end
165Ø JSR &FFE3
166Ø BRA timeloop
167Ø  end
168Ø CLC
169Ø LDA &F4
17ØØ ADC ≠3Ø
171Ø TAX
172Ø LDA ≠161
173Ø JSR &FFF4
174Ø CPY ≠Ø
175Ø BEQ off
176Ø LDY ≠5
177Ø  off
178Ø LDA onoff,Y
179Ø BEQ finished
```

Listing 9 1 continued

```
1800 JSR &FFE3
1810 INY
1820 BRA off
1830 \
1840  finished
1850 PLY
1860 PLX
1870 PLA
1880 LDA ≠0
1890 RTS
1900 \
1910  string
1920 EQUS'DATE"
1930 EQUD &20202020
1940 EQUW &0020
1950 \
1960  onoff
1970 EQUS "OFF'
1980 EQUB 13
1990 EQUB 0
2000 EQUS "ON"
2010 EQUB 13
2020 EQUB 0
2030 \
2040  configure
2050 PHA
2060 PHX
2070 PHY
2080 LDA (&F2),Y
2090 CMP ≠13
2100 BNE notCR
2110 LDX ≠255
2120  conloop
2130 INX
2140 LDA string,X
2150 BEQ condone
2160 JSR &FFEE
2170 BRA conloop
2180 \
2190  condone
2200 INX
2210 LDA string,X
2220 CMP≠13
2230 BEQ nextcon
2240 JSR &FFE3
2250 BRA condone
2260 \
2270  nextcon
```

Listing 9 1 continued

```
 228Ø INX
 229Ø LDA ≠ASC"/
 23ØØ JSR &FFE3
 231Ø  doon
 232Ø INX
 233Ø LDA string,X
 234Ø BEQ thatsall
 235Ø JSR &FFE3
 236Ø BRA doon
 237Ø \
 238Ø  thatsall
 239Ø JMP restore
 24ØØ  notCR
 241Ø JSR trytime
 242Ø BCC spaces
 243Ø JMP restore
 244Ø \
 245Ø  spaces
 246Ø LDA (&F2),Y
 247Ø CMP ≠32
 248Ø BNE none
 249Ø INY
 25ØØ BRA spaces
 251Ø \
 252Ø  none
 253Ø AND ≠&DF
 254Ø CMP ≠ASC"O"
 255Ø BEQ tryN
 256Ø JMP restore
 257Ø \
 258Ø  tryN
 259Ø INY
 26ØØ LDA(&F2),Y
 261Ø AND ≠&DF
 262Ø CMP ≠ASC"N"
 263Ø BNE tryF
 264Ø LDY ≠1
 265Ø BRA write
 266Ø \
 267Ø  tryF
 268Ø CMP ≠ASC'F"
 269Ø BEQ yesF
 27ØØ JMP restore
 271Ø  yesF
 272Ø \
 273Ø LDY ≠Ø
 274Ø  write
 275Ø CLC
```

Listing 9 1 continued

```
 2760 LDA &F4
 2770 ADC ≠30
 2780 TAX
 2790 LDA ≠162
 2800 JSR &FFF4
 2810 JMP finished
 2820 \
 2830  trytime
 2840 DEY
 2850 LDX ≠255
 2860  loopt
 2870 INY
 2880 INX
 2890 CPX≠4
 2900 BEQ ours
 2910 LDA (&F2),Y
 2920 AND ≠&DF
 2930 CMP string,X
 2940 BEQ loopt
 2950 SEC
 2960 RTS
 2970 \
 2980  ours
 2990 CLC
 3000 RTS
 3010 ]
 3020 NEXT
 3030 ENDPROC
 3040
 3050 DEF PROCchecksum
 3060 N%=0
 3070 FOR X%=&5000 TO &520E
 3080 N%=N%+?X%
 3090 NEXT
 3100 IF N%=64511 THEN ENDPROC
 3110 VDU 7
 3120 PRINT"Assembler error!"
 3130 STOP
```

Chapter Ten Booting ROMs

ROMs may be turned on at a 'hard reset' -- by depressing a particular key while also pressing the CTRL and BREAK keys together For example, holding the D key down while pressing CTRL and BREAK together (written CTRL-D-BREAK) and the disc filing system will be booted In a similar manner, hold the A key down when pressing CTRL-BREAK (CTRL-A-BREAK) to select the Advanced Disc Filing System If you have an Econet board fitted then you can boot the Advanced Network Filing System by pressing down the N key with CTRL-BREAK (CTRL-N-BREAK) This auto selection does not happen by magic - obviously the ROM concerned must look to see if its chosen key is being depressed at the same time, and if so take the necessary action To facilitate this a service call is provided - number 3 - and ROMs which can take advantage of it should test for it and trap in the normal fashion

Service call 3 is not issued by the MOS at every hard reset When this occurs the MOS looks at the keyboard to see if any other key(s) is being pressed Only if one is will it issue a service call 3

Once the service call is caught the first step is to 'look' at the keyboard to see what 'other' key was being pressed This is performed by OSBYTE &7A which will return the INTERNAL key number of any key detected in the X register Note this is the internal key number as used by the Master itself Table 1Ø 1 lists the internal number for each key

Obviously you will need to choose a key that is not being used to auto-boot another ROM so beware of choosing letters such as D, A, N and F (which is also used by the ADFS)

In the example program detailed here two boot options are

Key	ASCII	INKEY	Key	ASCII	INKEY
SPACE	32	&62	'	44	&66
-	45	&17		46	&67
/	47	&68	Ø	48	&27
1	49	&3Ø	2	5Ø	&31
3	51	&11	4	52	&12
5	53	&13	6	54	&34
7	55	&24	8	56	&15
9	57	&25		58	&48
,	59	&57		64	&47
A	65	&41	B	66	&64
C	67	&52	D	68	&32
E	69	&22	F	7Ø	&43
G	71	&53	H	72	&54
I	73	&26	J	74	&45
K	75	&46	L	76	&56
M	77	&65	N	78	&55
O	79	&36	P	8Ø	&37
Q	81	&1Ø	R	82	&33
S	83	&51	T	84	&23
U	85	&35	V	86	&63
W	87	&21	X	88	&42
Y	89	&44	Z	9Ø	&61
[	91	&38	\	92	&78
]	93	&58	^	94	&18
_	95	&28	ESCAPE	27	&7Ø
TAB	9	&6Ø	CAPSLK		&4Ø
CTRL		&1	SHIFTLK		&5Ø
SHIFT		&Ø	DELETE	127	&59
COPY		&69	RETURN	13	&49
UP CRSR		&39	DN CRSR		&29
LT CRSR		&19	RT CRSR		&79
fØ		&2Ø	f1		&71
f2		&72	f3		&73
f4		&14	f5		&74
f6		&75	f7		&16
f8		&76	f9		&77

Table 1Ø 1 Internal key numbers

provided The first allows you to catalogue a disc by pressing CTRL-C-BREAK, and the second will instigate the ROM Filing System (examined in Chapter 14) with CTRL-R-BREAK

Therefore we need to test the X register for the internal key codes for the letters C and R, that is &52 and &33

respectively The coding to do this is given in the following lines

```
     boot
    PHA                 \ save registers
    PHX
    PHY
    LDA ≠&7A
    JSR &FFF4           \ read keyboard
    CPX ≠&52            \ was it a 'C'?
    BEQ cat             \ yes so do *CAT
    CPX ≠&33            \ was it an 'R'?
    BEQ rom             \ yes so do *ROM
    JMP restore         \ else restore and return
```

If a 'C' is detected then before we can *CAT the disc, the disc filing system must be selected To do this we place the command *DISC (abbreviated to *DI) into the input buffer using OSBYTE &8A X should contain Ø and the Y register the ASCII value of the character to be inserted Writing a return character (ASCII 13) will complete the operation Before doing this the keyboard buffer should be flushed with OSBYTE 15 to remove any surplus keypresses

```
    LDA ≠15
    JSR &FFF4           \ flush buffers
    LDA ≠&8A            \ character insert code
    LDX ≠Ø
    LDY ≠ASC("*")       \ insert *
    JSR &FFF4
    LDY ≠ASC("D")       \ insert D
    JSR &FFF4
    LDY ≠ASC("I")       \ insert I
    JSR &FFF4
    LDY ≠ASC(" ")       \ insert
    JSR &FFF4
    LDY ≠13             \ do *DI
    JSR &FFF4
```

The next action is to catalogue the disc using * as an abbreviation for *CAT

```
    LDA ≠&8A            \ character insert code
    LDX ≠Ø
    LDY ≠ASC("*")       \ insert *
    JSR &FFF4
    LDY ≠ASC(" ")       \ insert
    JSR &FFF4
    LDY ≠13             \ do *CAT
    JSR &FFF4
```

All that remains is for the stack to be pulled and the accumulator loaded with zero to acknowledge a successful boot

The *ROM coding is the same except that we insert *ROM into the keyboard buffer, remembering to flush it first of all though

```
    LDA ≠15
    JSR &FFF4          \ flush buffers
    LDA ≠&8A           \ character insert code
    LDX ≠Ø
    LDY ≠ASC("*")      \ insert *
    JSR &FFF4
    LDY ≠ASC("R")      \ insert R
    JSR &FFF4
    LDY ≠ASC("O")      \ insert O
    JSR &FFF4
    LDY ≠ASC("M")      \ insert M
    JSR &FFF4
    LDY ≠13            \ do *ROM
    JSR &FFF4
```

Before restoring the registers the routine prints a short filing system message on to the screen to signify that the ROM filing system is active

<u>Entering the Program</u>

You can use program 4 3 (saved as 'HELP3') as the basis for listing 1Ø 1 and make the changes and additions detailed below Once complete save the program as 'BOOT'

```
    Change lines    1Ø, 22Ø, 1Ø4Ø, 1Ø7Ø, 11ØØ, 112Ø
    Add lines       55, 311, 312, 313, 314, 115Ø to 199Ø
                    inclusive
```

Listing 10 1 Sets up two boot options CTRL-C-BREAK will catalogue a disc, CTRL-R-BREAK will set up the ROM Filing System Save as BOOT Can be adapted from listing 4 3 (HELP3)

```
   10 REM Autoboot ROM
   20 REM (C) Bruce Smith June 1986
   30 REM Advanced SRAM Guide
   40
   50 PROCassemble
   55 PROCchecksum
   60 *SRWRITE 5000 +200 8000 6
   70 END
   80 DEF PROCassemble
   90 osnewl=&FFE7
  100 comline=&F2
  110 FOR pass=4 TO 7 STEP 3
  120 P%=&8000   O%=&5000
  130 [
  140 OPT pass
  150 EQUB 0
  160 EQUW 0
  170 JMP service
  180 EQUB &82
  190 EQUB offset MOD 256
  200 EQUB 1
  210  title
  220 EQUS "CTRL Boot ROM'
  230 EQUB 0
  240  version
  250 EQUS " 1 00
  260 EQUB 0
  270  offset
  280 EQUB 0
  290 EQUS "(C) Bruce Smith"
  300 EQUB 0
  310  service
  311 CMP ≠3
  312 BNE tryhelp
  313 JMP boot
  314  tryhelp
  320 CMP ≠9
  330 BNE nothelp
  340 PHA
  350 PHX
  360 PHY
  370 LDA (comline),Y
  380 CMP ≠13
  390 BNE check
  400 JSR help
```

Listing 10 1 continued

```
  410 LDX #255
  420  details
  430 INX
  440 LDA command,X
  450 BEQ donecommand
  460 JSR &FFE3
  470 BRA details
  480  donecommand
  490 JSR osnewl
  500 BRA restore
  510 \
  520  check
  530 LDX #255
  540 DEY
  550  again
  560 INX
  570 INY
  580 LDA (comline),Y
  590 AND #&DF
  600 CMP com,X
  610 BEQ again
  620 LDA com,X
  630 CMP #&FE
  640 BEQ mine
  650  restore
  660 PLY
  670 PLX
  680 PLA
  690  nothelp
  700 RTS
  710 \
  720  help
  730 JSR osnewl
  740 LDX #&FF
  750 JSR helploop
  760 JSR helploop
  770 JSR osnewl
  780 RTS
  790 \
  800  helploop
  810 INX
  820 LDA title,X
  830 BEQ done
  840 JSR &FFE3
  850 BRA helploop
  860  done
  870 RTS
  880  mine
```

Listing 10.1 continued

```
  890 JSR help
  900 LDX ≠255
  910  more
  920 INX
  930 LDA lists,X
  940 BMI alldone
  950 JSR &FFE3
  960 BRA more
  970  alldone
  980 PLY
  990 PLX
 1000 PLA
 1010 LDA ≠0
 1020 RTS
 1030  com
 1040 EQUS"BOOT"
 1050 EQUB &FE
 1060  command
 1070 EQUS"  Boot"
 1080 EQUB 0
 1090  lists
 1100 EQUS "  CTRL-C-BREAK   Catalogue D
isc"
 1110 EQUB 13
 1120 EQUS "  CTRL-R-BREAK   ROM Filing
System"
 1130 EQUB 13
 1140 EQUB &FF
 1150 \
 1160  boot
 1170 PHA
 1180 PHX
 1190 PHY
 1200 LDA ≠&7A
 1210 JSR &FFF4
 1220 CPX ≠&52
 1230 BEQ cat
 1240 CPX ≠&33
 1250 BEQ rom
 1260 JMP restore
 1270 \
 1280  cat
 1290 LDA ≠15
 1300 JSR &FFF4
 1310 LDA ≠&8A
 1320 LDX ≠0
 1330 LDY ≠ASC("*")
 1340 JSR &FFF4
```

Listing 1Ø 1 continued

```
1350 LDY ≠ASC("D")
1360 JSR &FFF4
1370 LDY ≠ASC("I")
1380 JSR &FFF4
1390 LDY ≠ASC("  ')
14ØØ JSR &FFF4
141Ø LDY ≠13
142Ø JSR &FFF4
143Ø LDA ≠&8A
144Ø LDX ≠Ø
145Ø LDY ≠ASC("*")
146Ø JSR &FFF4
147Ø LDY ≠ASC("   )
148Ø JSR &FFF4
149Ø LDY ≠13
15ØØ JSR &FFF4
151Ø JMP out
152Ø \
153Ø  rom
154Ø LDA ≠15
155Ø LDA ≠&8A
156Ø LDX ≠Ø
157Ø LDY ≠ASC("*")
158Ø JSR &FFF4
159Ø LDY ≠ASC("R")
16ØØ JSR &FFF4
161Ø LDY ≠ASC('O")
162Ø JSR &FFF4
163Ø LDY ≠ASC("M")
164Ø JSR &FFF4
165Ø LDY ≠13
166Ø JSR &FFF4
167Ø \
168Ø LDX ≠255
169Ø  rfs
17ØØ INX
171Ø LDA romfs,X
172Ø BEQ out
173Ø JSR &FFE3
174Ø BRA rfs
175Ø \
176Ø  out
177Ø PLY
178Ø PLX
179Ø PLA
18ØØ LDA ≠Ø
181Ø RTS
182Ø \
```

Listing 10.1 continued

```
1830  romfs
1840 EQUS "ROM Filing System"
1850 EQUD &0D0D
1860 EQUB 0
1870 ]
1880 NEXT
1890 ENDPROC
1900
1910 DEF PROCchecksum
1920 N%=0
1930 FOR X%=&5000 TO &5183
1940 N%=N%+?X%
1950 NEXT
1960 IF N% = 45718 THEN ENDPROC
1970 VDU 7
1980 PRINT"Assembler error!"
1990 STOP
```

Chapter Eleven Workspace

Finding workspace in which ROM software can perform calculations and keep tabs on various values and addresses can be problematic when writing service ROMs Language ROMs are no problem as they are allowed a free run of the memory map and need avoid only the space allocated to the MOS and VDU drivers But service ROMs are another matter They must interact with the current language ROM and not corrupt any of its or the MOS's data One way was mentioned earlier - to use the zero page user's area from &7Ø to &8F inclusive, preserving its contents first by copying it onto the bottom of the

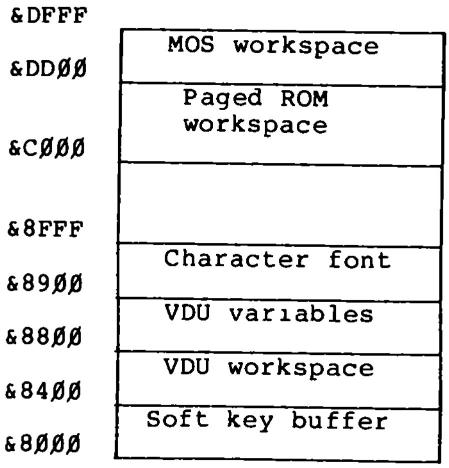


Figure 11 1 Hidden memory map

stack and restoring it by copying it back before returning While this method works it does not provide any permanent means of storing data across several commands or actions However, there is a way by which service ROMs can grab their own memory in steps of 256 bytes - a memory page at a time

Figure 11 1 shows how the 12k of 'hidden' memory is arranged This contains the function key definition buffer, MOS drivers, exploded character fonts and, most importantly, ROM workspace This ROM workspace is a 7 25k block that stretches from &C000 to &DCFF inclusive - 29 pages of memory in all, which is free for use by ROMs

However this is not all for our use remember that the Master comes fitted with a host of firmware and two of these, the DFS and the ADFS bite heavily into this - but more on this in a moment

Static and Dynamic

There are two types of ROM workspace The first is the 'static' type, so called because it has fixed start and end boundaries The start is &C000 and the end can be any value up to a maximum of &DBFF Static workspace is open to all ROMs - generally to use as they wish, although before doing so they must inform other ROMs of their needs via service call 10 The second type of ROM workspace is 'dynamic', which has a moveable boundary and depends on a ROM or ROM's requirements Any ROM can claim its own 'private' workspace that only it has access to Thus important data and information can be stored away without fear of corruption by other ROMs or the MOS

Obviously there is only a finite amount of private workspace within the alloted 7 25k hidden RAM If this is exceeded then the ROM workspace is moved into the user RAM starting at PAGE, which you mightn't notice, but the user will when using a program where memory is tight! As a general rule it should be considered as bad form to exceed and break out of the hidden RAM workspace

Now let's go through the the service calls associated with this workspace

Service Call 36 (&24)

This is the second call issued by the MOS after a hard reset and it asks ROMs to indicate how much private workspace they will require However, it does not actually allocate any workspace

On entry the Y register will contain the page number of the current upper limit of the private workspace All that ROMs need do is to increment the Y register by the number of pages of memory required If one page is required they increment it once, if two are required then increment it twice and so forth On completion the accumulator should be cleared, thus

```
 call36
INY                 \ only one page (256 bytes) needed
LDA #0              \ acknowledge
RTS                 \ and return
```

Note how simple it is With all the calls discussed here it is vitally important that the Y register is treated with respect It must not be decremented, or a crash is sure to result!

Service Call 33 (&21)

This is the next service call issued Its action is similar to the call above but is more concerned with static workspace, that is, workspace that may be used by all ROMs, though only one at a time Static workspace starts at &C000 and has an upper limit of &DBFF, which should not be exceeded Any ROM which requires static workspace should check the contents of the Y register on receiving the call If there is not enough space then Y should be incremented to the desired value It should not go beyond &DB at any time For example, if a ROM requires static workspace from &C000 to &D600 then if the contents of the Y register are less than &D6, Y should be loaded with &D6 If the contents are higher then they should not be altered in any way

```
 call33
CPY #&D6        \ is it?
BCC loadit      \ branch if less than
JMP return      \ enough, so return
 loadit
LDY #&D6        \ set Y to our requirements
JMP return
```

Service Call 34 (&22)

This call allows ROMs to locate their own private workspace in hidden RAM The first two calls detailed above allow the MOS to calculate where this begins, and then use this call to inform each ROM just where its own private workspace starts On issuing the service call the Y register contains the value of the first free page If the ROM is claiming private workspace it must save the current contents of the Y register in a special ROM workspace table that runs from &DFØ Table 11 1 details the byte associated with each ROM

ROM number	Table byte
Ø	&DFØ
1	&DF1
2	&DF2
3	&DF3
4	&DF4
5	&DF5
6	&DF6
7	&DF7
8	&DF8
9	&DF9
1Ø	&DFA
11	&DFB
12	&DFC
13	&DFD
14	&DFE
15	&DFF

Table 11 1 ROM workspace

ROM 6 would therefore place the Y register contents at &DF6 However, firmware can be placed in any one of 15 slots, so the correct way to locate the correct ROM table position is to use the X register as an index

```
    LDX &F4          \ get ROM position
    TYA              \ move Y across
    STA &DFØ,X       \ and save
```

The contents of the Y register can then be incremented by the desired amount to make space for the current ROM's private area before the 'new' base value is passed back to the MOS It is important that you only claim the number of pages specified during service call 36 (&24)

Whenever the private workspace is needed its start address can be obtained from the table and used with indirect addressing as required Further details on this along with a working program example can be found below

Service Call 1

This service call provides compatibility with standard BBC micros and should be used by BBC B, B+ and B+128 users Its purpose is akin to that of service call 36 in that it is trying to determine the total amount of shared workspace required by ROMs The memory used for this is not in private RAM but is claimed directly above PAGE, ie from &E00, and as such will reduce the amount of programming memory available to the user

When this service call is issued the Y register contains the page number of the present upper limit of this absolute workspace This value should be checked by a ROM requiring workspace If the value is less than that required then the value of the Y register should be incremented until there is sufficient memory

As an example, consider that a ROM you are writing requires two pages of RAM for workspace The coding to check and implement this might look like this

```
    CMP #1          \ was it absolute claim?
    BNE next        \ branch if not
    CPY #&10        \ is it >= &E00+&200?
    BCC no          \ branch if needs incrementing
    RTS             \ all okay so return
     no
    CPY #&0E        \ is it +1 or +2?
    BNE one         \ branch if only one page
    INY             \ increment page value
     one
    INY             \ increment page value
    RTS             \ and return
```

It is vital that the value in the Y register is not decremented, as this could lead to corrupted programs

Service Call 2

This call is issued after service call 1 has been completed It allows ROMs to claim their very own private workspace area above the static workspace area This area of memory is exclusive to the ROM claiming it and may not be used by any other ROM Trapping this call and storing the Y register in the ROM table is performed as described above when servicing call &22

Using Private Workspace

Because of the way the sideways RAM/ROM memory is addressed, it is not possible to read and write directly to the area of memory designated as private workspace for a particular ROM The reasons for this are somewhat technical, but it is not necessary to understand them to use the private workspace (The reasons are discussed below for readers who are more technically-minded) For the general reader it is sufficient to know that before any information is written to or read from private RAM the following code must be performed

```
     writeon
    LDA ≠8
    TSB &FE34
```

And on completion the following code

```
     writeoff
    LDA ≠&F7
    TRB &FE34
```

IMPORTANT Once you have performed 'writeon' you cannot use any of the regular MOS calls until 'writeoff' is performed

Listing 11 1 demonstrates the use of private RAM workspace to implement two new commands, these are *PUSH and *PULL *PUSH saves the contents of zero page locations &7Ø to &8F to private workspace, while *PULL will restore them This routine will allow you to use these locations for workspace without fear of losing the contents of this area

The two routines 'writeon' and 'writeoff' are used to select and de-select the private workspace as described above Of course these two routines need only be used when private RAM is being used in the hidden RAM If service calls 1 and 2 have been trapped then the private workspace will be located in normal RAM above &EØØ and this can be read from and written to in the normal manner

The actual *PUSH and *PULL routines can be located in lines 2Ø2Ø to 219Ø inclusive

Once entered, save the program as 'PRIVATE'

Using Static Workspace

Static workspace is straightforward to use, but before you do you must inform the other ROMs in your computer by issuing service call 1Ø (&ØA) This is done with OSBYTE 143, with the X register holding the service call number

```
    LDA ≠143          \ OSBYTE code
    LDX ≠10           \ service call number
    LDY ≠255
    JSR &FFF4
```

Once this has been done the ROM is free to use the static workspace It is a good idea to keep a flag within the ROM's private workspace so that the ROM can determine whether it has use of the static workspace at any time

Obviously a ROM that is capable of claiming static workspace must also be capable of releasing it So therefore the service call software must be capable of trapping service call 10 On receiving the call the ROM should save any vital information in private workspace, and generally close up shop Once this has been done the accumulator should be loaded with zero before returning

As with private workspace, static workspace can only be used after 'writeon' has been performed No MOS calls can be used until 'writeoff' has be completed Similarly if the static RAM is located in normal memory above &E00 then read-write can be performed directly

For the Technically Inclined

Location &FE34 is the Access Control Latch, ACCCON for short The state of individual bits within this latch determine what areas of memory are in use at any time It effectively dictates the activity of two regions of memory

1) &3000 to &7FFF
2) &C000 to &DFFF

It is the second area we are concerned with here - this is the static and private ROM workspace in the hidden RAM

As our firmware is itself in paged memory it cannot directly access the private and static workspace in the hidden RAM which is mapped in a similar area What the routine 'writeon' does is to overlay this area on the MOS, so that it appears 'above' the firmware using it It therefore 'covers' the MOS, and in particular the VDU drivers which therefore cannot be seen It is for this reason that firmware should not attempt to use them, until the hidden RAM is removed from this area by the 'writeoff' routine What these two routines do is to toggle bit 3 within ACCCON - this is the bit that determines whether the hidden RAM is overlaid or not

To overlay the hidden RAM this bit must be set, but do not disturb the other bits in this latch which have specific functions themselves

```
    LDA #8              \ set bit 3, ie 0000 1000
    TSB &FE34           \ test and reset bit in latch
```

Removing the hidden RAM, thus giving access to the MOS calls again, simply involves clearing bit 3 in the ACCCON latch Again the status of the other bits in the latch must be preserved so the AND operator should be used

```
    LDA #&F7            \ clear bit 3, ie 1111 0111
    TRB &FE34           \ test and reset bit in latch
```

Listing 11 1 Implements two new commands, *PUSH and *PULL to demonstrate use of private RAM workspace Save as PRIVATE

```
   10 REM Private Workspace ROM
   20 REM (C) Bruce Smith June 1986
   30 REM Advanced SRAM Guide
   40
   50 PROCassemble
   60 PROCchecksum
   70 *SRWRITE 5000 +200 8000 6
   80 END
   90
  100 DEF PROCassemble
  110 osnewl=&FFE7
  120 osasci=&FFE3
  130 comline=&F2
  140 FOR pass=4 TO 7 STEP 3
  150 P%=&8000   O%=&5000
  160 [
  170 OPT pass
  180 EQUB 0
  190 EQUW 0
  200 JMP service
  210 EQUB &82
  220 EQUB offset MOD 256
  230 EQUB 1
  240  title
  250 EQUS "Private Workspace ROM"
  260 EQUB 0
  270  version
  280 EQUS ' 1 00"
  290 EQUB 0
  300  offset
  310 EQUB 0
  320 EQUS "(C) Bruce Smith"
  330 EQUB 0
  340  service
  350 CMP #34
  360 BNE try36
  370 JMP its34
  380  try36
  390 CMP #36
  400 BNE tryhelp
  410 JMP its36
  420  tryhelp
  430 PHA
  440 PHX
  450 PHY
  460 CMP #9
```

Listing 11 1 continued

```
470 BNE nothelp
480 LDA (comline),Y
490 CMP ≠13
500 BNE check
510 JSR help
520 LDX ≠255
530  details
540 INX
550 LDA command,X
560 BEQ donecommand
570 JSR &FFE3
580 BRA details
590  donecommand
600 JSR osnewl
610 BRA restore
620 \
630  check
640 LDX ≠255
650 DEY
660  again
670 INX
680 INY
690 LDA (comline),Y
700 AND ≠&DF
710 CMP com,X
720 BEQ again
730 LDA com,X
740 CMP ≠&FE
750 BEQ mine
760  restore
770 PLY
780 PLX
790 PLA
800 RTS
810 \
820  nothelp
830 CMP ≠4
840 BEQ unrecognised
850 BRA alldone
860 \
870  help
880 JSR osnewl
890 LDX ≠&FF
900 JSR helploop
910 JSR helploop
920 JSR osnewl
930 RTS
940 \
```

Listing 11 1 continued

```
  95Ø  helploop
  96Ø INX
  97Ø LDA title,X
  98Ø BEQ done
  99Ø JSR &FFE3
 1ØØØ BRA helploop
 1Ø1Ø  done
 1Ø2Ø RTS
 1Ø3Ø \
 1Ø4Ø  mine
 1Ø5Ø JSR help
 1Ø6Ø LDX ≠255
 1Ø7Ø  more
 1Ø8Ø INX
 1Ø9Ø LDA lists,X
 11ØØ BMI alldone
 111Ø JSR &FFE3
 112Ø BRA more
 113Ø \
 114Ø  alldone
 115Ø PLY
 116Ø PLX
 117Ø PLA
 118Ø RTS
 119Ø \
 12ØØ  com
 121Ø EQUS COMMANDS"
 122Ø EQUB &FE
 123Ø  command
 124Ø EQUS"  Commands"
 125Ø EQUB Ø
 126Ø  lists
 127Ø EQUS    PUSH"
 128Ø EQUB 13
 129Ø EQUS "  PULL"
 13ØØ EQUB 13
 131Ø EQUB &FF
 132Ø \
 133Ø  unrecognised
 134Ø LDX ≠255
 135Ø DEY
 136Ø PHY
 137Ø  identify
 138Ø INX
 139Ø INY
 14ØØ LDA (comline),Y
 141Ø AND ≠&DF
 142Ø CMP comtable,X
```

Listing 11 1 continued

```
1430 BEQ identify
1440 LDA comtable,X
1450 BMI address
1460 \
1470  moveon
1480 INX
1490 LDA comtable,X
1500 BPL moveon
1510 BNE notend
1520 PLY
1530 BRA alldone
1540 \
1550  notend
1560 INX
1570 PLY
1580 PHY
1590 JMP identify
1600 \
1610  address
1620 CMP #&FF
1630 BNE notFF
1640 PLY
1650 BRA alldone
1660  notFF
1670 \
1680 STA &39
1690 INX
1700 LDA comtable,X
1710 STA &38
1720 JMP (&38)
1730 \
1740  comtable
1750 EQUS "PUSH"
1760 EQUB push DIV 256
1770 EQUB push MOD 256
1780 EQUS 'PULL"
1790 EQUB pull DIV 256
1800 EQUB pull MOD 256
1810 EQUB &FF
1820 \
1830  found
1840 PLY
1850 PLY
1860 PLX
1870 PLA
1880  okay
1890 LDA #0
1900 RTS
```

Listing 11 1 continued

```
1910 \
1920  its36
1930 INY
1940 BRA okay
1950 \
1960  its34
1970 TYA  LDX &F4
1980 STA &DF0,X
1990 INY
2000 BRA okay
2010 \
2020  push
2030 JSR writeon
2040 LDX &F4
2050 LDA &DF0,X
2060 STA &39
2070 LDY #0
2080 STY &38
2090 DEY
2100  pushloop
2110 INY
2120 LDA &70,Y
2130 STA (&38),Y
2140 CPY #&1F
2150 BNE pushloop
2160 JSR writeoff
2170 JMP found
2180 \
2190  pull
2200 JSR writeon
2210 LDX &F4
2220 LDA &DF0,X
2230 STA &39
2240 LDY #0
2250 STY &38
2260 DEY
2270  pullloop
2280 INY
2290 LDA (&38),Y
2300 STA &70,Y
2310 CPY #&1F
2320 BNE pullloop
2330 JSR writeoff
2340 BRA found
2350 \
2360  writeon
2370 LDA #8
2380 ORA &FE34
```

Listing 11.1 continued

```
 2390 STA &FE34
 2400 RTS
 2410 \
 2420  writeoff
 2430 LDA #&F7
 2440 AND &FE34
 2450 STA &FE34
 2460 RTS
 2470 ]
 2480 NEXT
 2490 ENDPROC
 2500
 2510 DEF PROCchecksum
 2520 N%=0
 2530 FOR X%=&5000 TO &517C
 2540 N%=N%+?X%
 2550 NEXT
 2560 IF N%=46643 THEN ENDPROC
 2570 VDU 7
 2580 PRINT"Assembler error!"
 2590 STOP
```

Chapter Twelve
ROM Calls

The switch that controls which of the sideways ROMs is paged in at any time is software controlled In fact a particular ROM is selected simply by writing the binary representation of the ROM socket number, into the low four bits of the paged ROM select register at &FE3Ø The MOS also keeps a copy of this at location &F4 and this must also be written to when you wish to select a particular ROM in this way The coding required to select a particular sideways ROM is very simple For example, to select the ROM in ROM socket 15 the coding would be

```
    LDX ≠15          \ load X register with ROM number
    STX &F4          \ write RAM copy
    STX &FE3Ø        \ write to ROM select register
```

Note the order in which the ROM slot number is written It must be written to the RAM copy at &F4 first Performing the operation in reverse could cause the Master to crash if an interrupt was to occur during the two write operations Note also that &FE3Ø should never be read - always use &F4 when you wish to ascertain the ROM number Poking these addresses directly from BASIC or any other language will almost certainly result in the Master hanging up

In addition to location &F4 there are several other bytes within zero page RAM that are associated with the ROM system These are detailed in Table 12 1

We have already used the vectored address at &F2 several times To recap, the MOS uses it as a text pointer for processing commands Normally it holds the address of the first character after the asterisk in the command, and the Y register holds the 'post indirect index' to the command

Address	Function
&F2 - &F3	Text pointer vector
&F4	Value of currently selected ROM (copy of ROM select register)
&F6 - &F7	Vectored address of current position in paged ROM

Table 12 1 Paged ROM associated RAM addresses

Finally, the vectored address at &F6 holds the exact address of a position in a paged ROM (see below) Manipulating any of these addresses including the paged ROM select register must be done from machine code, otherwise the Master will hang up

Operating System Read ROM Call (OSRDRM)

At &FFB9 is the 'operating system read byte from paged ROM' call - OSRDRM for short This call allows single bytes within paged ROMs to be read from machine code or from other paged ROMs On entry the Y register should contain the number of the ROM to be read, while the vector at &F6 holds the address of the byte to be read On return from OSRDRM the accumulator contains the byte itself Listing 12 1 illustrates how this call can be used to read the BASIC title string The program begins by poking the address vector at &F6 with the start address of the title string, &8ØØ8 (lines 12Ø to 15Ø) The print 'loop' is entered at line 16Ø As we are entering the machine code from BASIC itself then the ROM socket number of BASIC can be extracted directly from &F4 (line 17Ø) The low byte of

OSBYTE	Function
&16 (22)	Increment ROM polling semaphore
&17 (23)	Decrement ROM polling semaphore
&8D (141)	Perform *ROM
&8E (142)	Enter language ROM
&8F (143)	Issue service request
&A8 (168)	Read address of ROM pointer table
&AA (17Ø)	Read address of ROM information table
&B3 (179)	Read/Write ROM polling semaphore
&BA (186)	Read number of ROM active at last BRK/error
&BB (187)	Read number of socket holding BASIC ROM
&FC (252)	Read/write current language ROM number

Table 12 2 OSBYTE calls associated with sideways ROMs the address to be read is incremented (line 180), and the byte is read (line 190) A zero byte will indicate the end of the title string so this is tested for by line 200, otherwise the byte in the accumulator is printed and the loop re-executed (lines 210 to 220) If you have several sideways ROMs present then their title strings can be printed simply by altering line 170

ROM Byte

There are several OSBYTE calls associated with the sideways ROM system, these are outlined here Table 12 2 lists the associated calls

OSBYTE &16 (*FX 22)

This causes the MOS to begin issuing service call number &15 (21) 100 times every second See Chapter 8 for details

OSBYTE &17 (*FX 23)

Stops MOS issuing service call number &15 (21) Chapter 8 has more details

OSBYTE &8D (*FX 141)

Allows the *ROM filing system to be selected There are no set up entry parameters and the accumulator contents are preserved See Chapter 13 for full details of the ROM filing system

OSBYTE &8E (*FX 142)

This call will boot up a selected language ROM On entry the X register contains the socket number of the language to be entered To enter View from BASIC or any other language, use

```
    *FX 142,14
```

View being in socket number 14 (&E)

OSBYTE &8F (*FX143)

This call will cause the MOS to issue a paged ROM service request Thus any ROM can get the MOS to issue a particular service call at any time it wishes The entry parameters for this call are that the X register contains the service code and the Y register the service argument, if any On exit the Y register may return a result if appropriate

OSBYTE &A8 (*FX 168)

This call returns the address of a ROM pointer table containing vectored addresses for entry into ROMs This subject is dealt with in chapter 7 On exit from the call the index registers return the address of the pointer table, low byte in X, high byte in Y For the 3 20 MOS the address returned is &D9F

OSBYTE &AA (*FX 170)
This call returns the address of a ROM information table that contains details of types of sideways ROMs present in the Master This information table is detailed in Chapter 11 The address is returned in the index registers - low byte in X, high byte in Y For the 3 20 MOS this address is &2A1

OSBYTE &B3 (*FX 179)
Using this call it is possible to read or write the state of the ROM polling semaphore

```
    A=179 X=n Y=0   will read the semaphore into X and set
                     the state to n
    A=179 X=0 Y=255 reads semaphore into X
```

Note that use of this call to set the state directly will interfere with *FX22 and *FX23

OSBYTE &BA (*FX 186)
This call returns the number of the ROM that was active when the last BRK error occurred The value is returned in the X register

OSBYTE &BB (*FX 187)
This call reads the number of the ROM socket which contains the BASIC ROM The number is returned in the X register Chapter 14 contains details of its use to re-boot BASIC to exit from another language ROM

OSBYTE &FC (*FX 252)
This call returns the number of the ROM socket containing the current language ROM in the X register It is written to whenever a new language ROM is booted with OSBYTE &8E

Listing 12 1 Reads BASIC title string to demonstrate OSRDRM Save as READ

```
   10 REM Read title string from ROM
   20 REM Advanced SRAM Guide
   30 REM (C) Bruce Smith June 1986
   40
   50 osrdrm=&FFB9
   60 osasci=&FFE3
   70 FOR pass=0 TO 3 STEP 3
   80 P%=&A00
   90 [
  100 OPT pass
  110  readstring
  120 LDA ≠&80
  130 STA &F7
  140 LDA ≠8
  150 STA &F6
  160  loop
  170 LDY &F4
  180 INC &F6
  190 JSR osrdrm
  200 BEQ out
  210 JSR osasci
  220 BNE loop
  230  out
  240 RTS
  250 ]
  260 NEXT
  270 CALL readstring
```

Chapter Thirteen ROM Filing System

The ROM Filing System (RFS), may contain either BASIC or machine code programs and may be loaded, chained, or run as normal Table 13 1 lists the commands and operating system calls with the RFS, which is selected by the *ROM command

As with any other filing system, eg tape, disc, ADFS, files must be saved to a particular format In the case of RFS they must be formatted as an image that can be loaded directly into sideways RAM or blown into an EPROM The number of files stored per ROM image is limited only by the amount of space within the sideways RAM block Thus RFS-formatted images may be up to 16k in length

Just like any other software that is to be placed in sideways RAM, the RFS image must contain a standard header along with a service entry point and any relevant coding In addition to any standard *HELP messages, etc, that you may wish to include, service

```
LOAD
*CAT
*EXEC
*LOAD
*RUN
OSARGS (filing system identification only)
OSBGET
OSFILE (save is not possible)
OSFIND (output opening is not possible)
```

Table 13 1 ROM Filing System commands and calls

calls 13 (&ØD) and 14 (&ØE) must be caught and processed as these inform the MOS of the RFS details

Service Call 13

This is the RFS initialisation call It is issued by the MOS when a filing system command is used while the RFS is active It allows a ROM to inform the MOS that it contains a ROM image, along with the start address

The service call number is held in the accumulator on entry and the Y register contains a number which is 15 minus the number of the next ROM to be scanned If this value is less than the number of the current ROM being investigated then the ROM should ignore the service call as it has already been processed earlier If not, the current ROM's number (at &F4) should be put in the accumulator and placed in zero page location &F5 This is an important step as it indicates to the MOS that an RFS-formatted ROM is present

The final act of the ROM header coding should be to place the start address of the ROM file data into the vector at &F6 and &F7 To complete the call and inform the MOS that the current ROM is now active the call should return with the accumulator holding zero

```
  entry
 CMP ≠13                  \ service call 13?
 BNE tryagain             \ if not branch over
 PHA                      \ push accumulator
 TYA                      \ Y holds ROM number
 EOR ≠15                  \ calculate (15-ROM number)
 CMP &F4                  \ less than current ROM
                          \ number?
 BCC return               \ yes if carry clear so
                          \ return
 LDA ≠filename MOD 256    \ low byte file start
                          \ address
 STA &F6                  \ save in vector low byte
 LDA ≠filename DIV 256    \ high byte file start
                          \ address
 STA &F7                  \ save in vector high byte
 LDA &F4                  \ get current ROM number
 EOR ≠15                  \ restore ROM number on
                          \ entry
 STA &F5                  \ save the flag
 JMP restore              \ jump to exit routine
  return
 PLA                      \ restore service type
 RTS                      \ back to MOS
  tryagain
```

The label 'filename' is used to mark the start of the programs in RFS format In the formatting program presented later on, I have in fact given this an absolute address, namely &8100 and have therefore loaded these two bytes immediately into the accumulator on both occasions

Service Call 14

This call is a simple RFS 'get byte' routine To respond to this the current ROM must check location &F5 If this byte is equal to 15-?&F4, then the current ROM is indicated The MOS uses this call to read bytes from the ROM when performing filing system actions, such as *CAT, LOAD etc

To extract the correct byte the Y register must be cleared as used for post indirect address to peek the byte held at the vectored address in &F6 The read byte should be returned in the Y register with the accumulator clear to indicate that the call has been serviced correctly The ROM byte request is handled as follows

```
 tryagain         \ entry
CMP #&0E          \ is it ROM byte get?
BNE back          \ if not then branch to back
PHA               \ save service call
LDA &F5           \ get 'current' ROM value
EOR #15           \ calculate (15-ROM number)
CMP &F4           \ is it same as this ROM number?
BNE return        \ no, it's another ROM so return
LDY #0            \ clear indexing register
LDA (&F6),Y       \ read byte into accumulator
TAY               \ move into Y register
INC &F6           \ increment low byte vector
BNE restore       \ branch if not zero
INC &F7           \ else increment high byte of
                  \ vector
 restore
PLA               \ pull service type off stack
LDA #0            \ clear accumulator to indicate
                  \ that service has been performed
RTS               \ and back to MOS
```

Again, nothing too difficult in the coding The service call handling routine is the minimum required It can be expanded to include a *HELP service call, as in the formatting program (listing 13 1) where the above code can be seen in lines 1740 to 1910 inclusive Standard service ROM utility programs and languages can be mixed with RFS formatted programs, as long as there is enough space and the correct service call coding is present

The ROM Image

The construction of the actual RFS program image is similar to the cassette filing system, using a block structure Each block consists of a header followed by the file data The header construction is important and is laid out as follows

1 Synchronisation byte, &2A (ASCII"*")
2 Filename, up to ten characters long
3 A filename terminating zero byte, &ØØ
4 File load address
5 File execution address
6 Two-byte block number
7 Two-byte block length
8 File flag
9 Address of first byte after end of file
1Ø Two-byte header cyclic redundancy check (CRC)

The synchronisation byte must always be &2A (ASCII "*") so that the filename always looks as though it was in *RUN format, ie, *FILENAME A filename cannot be a null string so must contain a minimum of just one character though it must not exceed ten characters in length The filename is terminated with a zero byte The load and execution addresses occupy four bytes, the low byte being stored first The high two bytes provide space for a second or co-processor relocation address The block number and length details consist of two bytes stored low byte first

The file flag provides details about the file stored at bit level Three bits are used thus

Bit 7	If set, indicates this is the last block of the current file
Bit 6	If set, indicates this block contains no data
Bit Ø	Protection bit If set, the file can only be *RUN

The function of bit 6 may seem odd at first sight An empty block can be created at ROM image formation time if the file is opened for output and then closed before any data can be written to it using BPUT

The header cyclic redundancy check (CRC) is contained in two bytes, stored high byte first The CRC is an error check against data corruption Each CRC is unique to the item it refers to as it is calculated from all the data that it relates to' A suitable algorithm for calculating the CRC of a piece of data would be

```
High Byte = data EOR high byte
For loop=1 TO 8
Carry=0
IF (msb of high byte=1) THEN high and low bytes EOR
&810  Carry=1
high and low bytes=(high and low bytes*2+carry) AND
&FFFF
NEXT loop
```

After the header comes the file block which is, for a full block, 256 bytes The last block of a file may be shorter if the file length is not exactly divisible by 256 The length of the block is specified in the two header bytes, block length The file data is terminated by the two CRC bytes as calculated for the header CRC

To save ROM space the header of file blocks, other than the first and last file blocks, may be abbreviated by a single character, the hash, '≠' which is ASCII &43 If a hash header is used the MOS assumes the header details are the same as in the first file block

Finally, the end of the ROM image, that is the byte after the last file of the last program, is marked by an end-of-ROM marker, typically '+', ASCII code &2B This marker may be omitted only if the ROM image spans over to another ROM which must be positioned as the next ROM number in order of priority

ROM Filing System Vectors

As is common on the Master micro, indirect entry to the RFS processing is performed via the standard page two vectors Table 13 2 lists the vectors changed by initialisation of the RFS and the address contained within them for the 3 20 MOS

Vector	Address	Indirection address
FILEV	&212	&FF1B
ARGSV	&214	&FF1E
BGETV	&216	&FF21
BPUTV	&218	&FF24
GPPBV	&21A	&FF27
FINDV	&21C	&FF2A
FSCV	&21E	&FB69

Table 13 2 ROM Filing System Vectors

Table 13 2 indicates that provision for a OSBPUT has been included in the RFS vectored entry, but this is rather meaningless as a ROM may only be read from

ROM Image Formatter

Listing 13 1 is a tried and tested ROM image formatting program Because of its use of random access filing, it will not work effectively on a tape-based system, and as such has been written with disc or net in mind The program will read in specified files from the storage medium in use and format them into a ROM image for use with the RFS As written the program assumes that a 16k image is required though it may be any length up to this value

If the total file image length will exceed 16k then you are informed and the last file entered is not accepted On completion, the ROM image may be saved to the current filing system, written directly into sideways RAM or left in memory

Using the Formatter

A brief description of the program can be found at the end of this chapter Once you have entered the program save it under the filename 'ROMFS' You can then go ahead and use it as described below

The RFS formatter is simple to use First ensure that you have all the programs that you wish to format to hand, preferably on the same disc (or in the current directory if running ADFS or ANFS)

On running the screen will clear and you will be asked to enter the title of the ROM you wish to format This is the string that will be printed out in response to a *HELP After this you will be prompted for any copyright string You need not enter anything if you so wish, the obligatory '(C)' is entered by the program Now you will be requested to enter the name of the first file Do this and press return The file will then be read in and formatted The formatting process may take several moments for a longish program Once the file has been read in and formatted the amount of memory remaining in the ROM image will be displayed You will then be asked for the name of the next file If the file size exceeds that of the space remaining or the specified file cannot be found then an error will be displayed and you will be asked for the next filename once again

Once formatting is complete, simply press the return key when the next filename prompt is issued to complete the construction of the ROM image You will then be asked if you wish to

1) Quit
2) Save the ROM image
3) Write the image to sideways RAM

Simply press the appropriate key to select In the case of 1 and 2 you will be asked to enter either a filename or the sideways RAM bank number respectively
  Once in sideways RAM press CTRL-BREAK to initialise the ROM Typing *ROM and then *CAT should show that all is in order Files can then be loaded in as normal

<u>Checking the Image</u>

Of course it is possible that your ROM image will not work correctly first time round The two possible errors you could get are 'Bad Rom' and 'Data' The Bad ROM error message means that your header coding is not correct so recheck through lines 1360 to 2090 'Data' infers that the problem lies elsewhere in the program and will need to be checked thoroughly The following pages details the formation and checking of a 'standard' ROM image If you are having problems, work through what follows and try to locate where your problem lies
  The first step is to enter and save a short test program

```
10REM demo listing
20REM for use with
30REM RFS format
40REM program
```

It is important that the program is entered exactly as shown, with no extra spaces The program should occupy just 67 bytes, so ensure that the memory marker TOP is &E43 Check this by typing

```
PRINT ~TOP
```

If this is not the case ensure that you have not entered any extra spaces at the end of a line Once you are satisfied all is well save the program twice using the filenames 'DEMO1' and 'DEMO2'
  Access to a hexadecimal and ASCII dump routine is vital If you have a suitable utility available in a sideways ROM then all is well In case you don't, program 13 2 is just such a routine Enter and test

this then save it to disc as 'DUMPER' The next stage is to enter the formatter, and clear the buffer using

```
    FOR N%=&3000 TO &4000 STEP 4 'N%=0 NEXT
```

This process is not normally required but it will enable us to see where the ROM image ends clearly The next step is to run the formatter and use DEMO and TEST (in uppercase) as the title and copyright strings Now enter DEMO1 as the name of the first file to be formatted Once this has been read in the number of bytes remaining should be shown to be 16033 Enter DEMO2 as the second file to be formatted After formatting the bytes remaining should be 15938 Now press return and select option 1 from the menu, ie Quit

The next stage in the process is to load the DUMPER program Running this will produce a dump of the ROM image as shown in figure 13 1 This should be examined closely byte by byte and the following description should help

The bytes from &3000 to &306D contain the ROM service call header as described earlier The title and copyright strings can clearly be seen in the ASCII dump section on the right hand side of the listing

```
    3000  00 00 00 4C 17 80 82 0D     L
    3008  00 44 45 4D 4F 00 28 43    DEMO (C
    3010  29 20 54 45 53 54 00 C9   ) TEST
    3018  09 F0 3B C9 0D D0 1B 48          H
    3020  98 49 0F C5 F4 90 11 A9    I
    3028  00 85 F6 A9 81 85 F7 A5
    3030  F4 49 0F 85 F5 4C 52 80    I   LR
    3038  68 60 C9 0E D0 FB 48 A5   h      H
    3040  F5 49 0F C5 F4 D0 F1 A0    I
    3048  00 B1 F6 A8 E6 F6 D0 02
    3050  E6 F7 68 A9 00 60 DA 20     h
    3058  E7 FF A2 FF E8 BD 09 80
    3060  F0 05 20 E3 FF 80 F5 20
    3068  E7 FF FA 4C 52 80 00 00     LR
```

Figure 13 1 Hexadecimal dump of ROM image

The bytes from 3070 to 30FF should all contain zero as these are not used

The line starting 3100 contains the synchronisation byte, &2A, followed by the ASCII filename and then the terminator byte, &00

```
    3100  2A 44 45 4D 4F 31 00 00   *DEMO1
```

The next four bytes, the last one, &ØØ, in the above dumped line and the first three in the line beginning 31Ø8 hold the program load address This is stored low byte first It should show as being FFFFØEØØ The next four bytes hold the exection address, again low byte first This should be FFFF8Ø2B The next two bytes (one in this line and one first in the next line) are the block number Both are zero as this is block zero

```
    31Ø8  ØE FF FF 2B 8Ø FF FF ØØ        +
    311Ø  ØØ 43 ØØ 8Ø 5F 81 ØØ ØØ     C  _
```

The line beginning 311Ø contains the file length in the second and third bytes, 43 ØØ in this case, low byte first The next byte, 8Ø, is the block byte, followed by four bytes holding the address of the byte after the end of the current file, which should be as shown

The first two bytes in the line beginning at 3118 is the header CRC The test program is then stored in file form from 311A to 315C, with the last byte &FF being the program TOP The next two bytes at 315D and 315E contain the file data CRC

```
        3118  B8 AB ØD ØØ ØA 12 F4 2Ø
        312Ø  64 65 6D 6F 2Ø 6C 69 73   demo lis
        3128  74 69 6E 67 ØD ØØ 14 12   ting
        313Ø  F4 2Ø 66 6F 72 2Ø 75 73     for us
        3138  65 2Ø 77 69 74 68 ØD ØØ   e with
        314Ø  1E 1Ø F4 2Ø 52 46 53 2Ø       RFS
        3148  66 6F 72 6D 61 74 ØD ØØ   format
        315Ø  28 ØD F4 2Ø 7Ø 72 6F 67   (   prog
        3158  72 61 6D ØD FF 6C 5D 2A   ram  l]*
```

The last byte in the line above is the synchronisation byte for the second file DEMO2 This then follows the same format and is listed below

```
          316Ø  44 45 4D 4F 32 ØØ ØØ ØE   DEMO2
          3168  FF FF 2B 8Ø FF FF ØØ ØØ     +
          317Ø  43 ØØ 8Ø BE 81 ØØ ØØ 38   C      8
          3178  F5 ØD ØØ ØA 12 F4 2Ø 64          d
          318Ø  65 6D 6F 2Ø 6C 69 73 74   emo list
          3188  69 6E 67 ØD ØØ 14 12 F4   ing
          319Ø  2Ø 66 6F 72 2Ø 75 73 65    for use
          3198  2Ø 77 69 74 68 ØD ØØ 1E    with
          31AØ  1Ø F4 2Ø 52 46 53 2Ø 66       RFS f
          31A8  6F 72 6D 61 74 ØD ØØ 28   ormat  (
          31BØ  ØD F4 2Ø 7Ø 72 6F 67 72      progr
          31B8  61 6D ØD FF 6C 5D 2B ØØ   am  l]+
```

The final byte in the ROM image is the end of ROM marker, &2B, located at &31BE All bytes beyond this should be set to zero
If your ROM image is as shown then the program is operating correctly If it will not function as a ROM image then check that you are installing it into sideways RAM correctly Because of space, the hash headers are not checked, so if the formatter works for small programs, but not longer ones, then the error will almost certainly be in PROChash

The Procedures

The formatter includes eleven procedures which form the basis of the program The function of each is as follows
PROCformat This procedure first tests to see if there is more than one block in a file If this is the case then PROChandle is called On return from PROChandle only the last block remains to be formatted so this is undertaken by the call to PROCfilehead The last action of this procedure is to close the open reading channel
PROCfilehead Constructs a detailed block header for the first and last blocks of a file, including the calculation of the header CRCs
PROCgetdata As its name implies this procedure reads each byte of data from a file and pokes it into the correct position in the ROM image It also provides the data CRC value
PROChash This procedure is called for all file blocks except the first and last It creates the abbreviated hash header for the intermediary files and also initialises each PROCgetdata call to fetch 256 bytes, in addition to keeping track of the block count
PROCassemble This simply assembles the machine code that calculates the CRC for both headers and data bytes
PROCromhead Assembles the ROM head details required by the MOS and also the service call polling as required These are assembled directly in the front of the ROM image
PROChandle This creates the first block image of a file and then controls the formatting of the intermediate blocks but not the very last block of a file
PROCnottape Reads the catalogue information of the specified file from a disc using OSFILE
PROCsave Simple saves the ROM image to the current filing medium

Listing 13 1 RFS Formatter Save as RFS

```
   10 REM ROM Filing System Formatter
   20 REM (C) Bruce Smith June 1986
   30 REM Advanced SRAM Guide
   40
   50 ON ERROR GOTO 2790
   60
   70 MODE7
   80 @%=0
   90 DIM block% 20,name% 20,mcode% 250
  100 size=&4000   flag%=1
  110 remain%=16128
  120 buffer%=&3000   marker%=&3100
  130 PROCassemble
  140 PROCheading
  150 PROCdetails
  160 PROCromhead(buffer%)
  170
  180 REPEAT
  190 PRINTTAB(0,5),SPC(38)
  200 PRINTTAB(0,5),
  210 PRINT Enter file name  ' CHR$(129)
,
  220 INPUT" '$name%
  230 IF $name%<> " THEN IF FNinfo THEN
 PROCformat
  240 IF flag% remain%=size-(nextfile%-&
 8000)
  250 flag%=1
  260 PRINT "Space remaining  ",
  270 PRINTremain%," bytes      "
  280 UNTIL $name%="'
  290 ?marker%=ASC("+")
  300 finish%=marker%+1
  310 PROCsave
  320 VDU 26
  330 CLS
  340 END
  350
  360 DEF PROCformat
  370 LOCAL block%
  380 block%=0
  390 IF extent%>256 THEN PROChandle
  400 PROCfilehead(marker%,name%,load%,e
 xecution%,block%,&80,extent%)
  410 CLOSE≠channel%
  420 ENDPROC
  430
  440 DEF FNinfo
```

Listing 13 1 continued

```
  450 LOCAL L$
  460 A%=0 Y%=0
  470 channel%=OPENUP($name%)
  480 PROCnottape
  490 nextfile%=&8000+marker%-buffer%+(L
EN($name%)+23)-(LEN($name%)+23)*(extent%
>256)-3*(extent%>512)*(((extent%-1) DIV
256)-1)+extent%
  500 space%=(nextfile%<&8000+size-1)AND
 (channel%<>0)
  510 IF NOT space% THEN PROCerror
  520 =space%
  530
  540 DEF PROCfilehead(position%,file%,l
address%,execaddr%,bcount%,flag%,length%
)
  550 LOCAL pos%
  560 ?position%=ASC("*")
  570 position%=position%+1
  580 $position%=$file%
  590 pos%=LEN($file%)+position%
  600 ?pos%=0
  610 pos%'1=laddress%
  620 pos%'5=execaddr%
  630 pos%'9=bcount%
  640 pos%'11=length%
  650 pos%?13=flag%
  660 pos%'14=nextfile%
  670 ?&84=pos%-position%+18
  680 '&80=position%
  690 CALL  docrc
  700 pos%'18='&82
  710 marker%=pos%+20
  720 PROCgetdata(length%)
  730 block%=block%+1
  740 ENDPROC
  750
  760 DEF PROCgetdata(length%)
  770 LOCAL pos%
  780 FOR pos%=0 TO length%-1
  790 marker%?pos%=BGET≠channel%
  800 NEXTpos%
  810 '&80=marker%
  820 ?&84=length%
  830 CALL docrc
  840 marker%'length%='&82
  850 extent%=extent%-length%
  860 marker%=marker%+length%+2
```

Listing 13.1 continued

```
 870 ENDPROC
 880
 890 DEF PROChash
 900 ?marker%=ASC("≠")
 910 marker%=marker%+1
 920 PROCgetdata(&100)
 930 block%=block%+1
 940 ENDPROC
 950
 960 DEF PROCassemble
 970 address=&80
 980 crc1=&82
 990 crc2=&83
1000 FOR pass=0 TO 2 STEP 2
1010 P%=mcode%
1020 [OPT pass
1030  docrc
1040 LDA ≠0
1050 STA crc1
1060 STA crc2
1070 TAY
1080  next
1090 LDA crc1
1100 EOR (address),Y
1110 STA crc1
1120 LDX ≠8
1130  again
1140 LDA crc1
1150 ROL A
1160 BCC over
1170 LDA crc1
1180 EOR ≠8
1190 STA crc1
1200 LDA crc2
1210 EOR ≠&10
1220 STA crc2
1230  over
1240 ROL crc2
1250 ROL crc1
1260 DEX
1270 BNE again
1280 INY
1290 CPY crc2+1
1300 BNE next
1310 RTS
1320 ]
1330 NEXT pass
1340 ENDPROC
```

Listing 13 1 continued

```
 135Ø
 136Ø DEF PROCromhead (header%)
 137Ø FOR pass=4 TO 6 STEP 2
 138Ø P%=&8ØØØ   O%=header%
 139Ø [OPT pass
 14ØØ EQUW Ø
 141Ø EQUB Ø
 142Ø JMP entry
 143Ø EQUB &82
 144Ø EQUB offset  MOD 256
 145Ø EQUB Ø
 146Ø  title
 147Ø EQUS title$
 148Ø  offset
 149Ø EQUB Ø
 15ØØ EQUS "(C) "+copy$
 151Ø EQUB Ø
 152Ø  entry
 153Ø CMP ≠9
 154Ø BEQ help
 155Ø CMP ≠13
 156Ø BNE tryagain
 157Ø PHA
 158Ø TYA
 159Ø EOR ≠15
 16ØØ CMP &F4
 161Ø BCC return
 162Ø LDA ≠Ø
 163Ø STA &F6
 164Ø LDA ≠&81
 165Ø STA &F7
 166Ø LDA &F4
 167Ø EOR ≠15
 168Ø STA &F5
 169Ø JMP restore
 17ØØ  return
 171Ø PLA
 172Ø  back
 173Ø RTS
 174Ø  tryagain
 175Ø CMP ≠&ØE
 176Ø BNE back
 177Ø PHA
 178Ø LDA &F5
 179Ø EOR ≠15
 18ØØ CMP &F4
 181Ø BNE return
 182Ø LDY ≠Ø
```

Listing 13 1 continued

```
 183Ø LDA (&F6),Y
 184Ø TAY
 185Ø INC &F6
 186Ø BNE restore
 187Ø INC &F7
 188Ø  restore
 189Ø PLA
 19ØØ LDA ≠Ø
 191Ø RTS
 192Ø \
 193Ø  help
 194Ø PHX
 195Ø JSR &FFE7
 196Ø LDX ≠255
 197Ø  helploop
 198Ø INX
 199Ø LDA title,X
 2ØØØ BEQ alldone
 2Ø1Ø JSR &FFE3
 2Ø2Ø BRA helploop
 2Ø3Ø  alldone
 2Ø4Ø JSR &FFE7
 2Ø5Ø PLX
 2Ø6Ø JMP restore
 2Ø7Ø ]
 2Ø8Ø NEXT
 2Ø9Ø ENDPROC
 21ØØ
 211Ø DEF PROChandle
 212Ø PROCfilehead(marker%,name%,load%,e
xecution%,block%,Ø,&1ØØ)
 213Ø IF extent%>256 THEN REPEAT PROChas
h UNTIL extent%<=256
 214Ø ENDPROC
 215Ø
 216Ø DEF PROCnottape
 217Ø 'block%=name%
 218Ø A%=5 X%=block% MOD 256
 219Ø Y%=block% DIV 256
 22ØØ CALL &FFDD
 221Ø load%=block%'2
 222Ø execution%=block%'6
 223Ø extent%=block%'1Ø
 224Ø flen%=extent%
 225Ø ENDPROC
 226Ø
 227Ø DEF PROCsave
 228Ø CLS
```

Listing 13.1 continued

```
 229Ø PRINT'Please select "
 23ØØ PRINT'  1) Quit"
 231Ø PRINT" 2) Save Formatted File"
 232Ø PRINT" 3) Write Formatted File
 233Ø key%=GET
 234Ø IF key%=ASC("1 ) THEN ENDPROC
 235Ø IF key%=ASC( 3") THEN GOTO 242Ø
 236Ø PRINT''
 237Ø INPUT 'Enter filename   title$
 238Ø save$="SAVE "+title$+' 3ØØØ +4ØØØ
 239Ø OSCLI (save$)
 24ØØ ENDPROC
 241Ø
 242Ø PRINT''
 243Ø INPUT "Enter RAM bank (4,5,6,7)
 rb$
 244Ø OSCLI ("SRWRITE 3ØØØ +4ØØØ 8ØØØ  +
rb$)
 245Ø PRINT'"Press CTRL-BREAK to initali
se"
 246Ø END
 247Ø
 248Ø DEF PROCheading
 249Ø FOR N%=1 TO 2
 25ØØ PRINTCHR$(13Ø) CHR$(141) SPC(9),
 251Ø PRINT"RFS Formatter"
 252Ø NEXT N%
 253Ø PRINT'CHR$(129),SPC(6),
 254Ø PRINT "(C) Bruce Smith 1986'
 255Ø PRINT TAB(Ø,24),CHR$(131) SPC(7)
 256Ø PRINT "Press RETURN to end',
 257Ø PRINT TAB(Ø,5),
 258Ø VDU 28,Ø,23,39,5
 259Ø ENDPROC
 26ØØ
 261Ø DEF PROCdetails
 262Ø INPUT "Enter ROM title   title$
 263Ø INPUT "Enter Copyright  "copy$
 264Ø ENDPROC
 265Ø
 266Ø DEF PROCerror
 267Ø PRINTTAB(Ø,8), ERROR"
 268Ø VDU 7
 269Ø PRINT File not found / File to big
"
 27ØØ CLOSE≠channel%   flag%=Ø
 271Ø PRINT''Press any key to continue",
 272Ø REPEAT UNTIL GET
```

Listing 13 1 continued

```
 2730 PRINTTAB(0,8),SPC(30)
 2740 PRINTSPC(30)'SPC(30)'SPC(30)
 2750 PRINTTAB(0,6)
 2760 ENDPROC
 2770
 2780 ***** ERROR HANDLER *******
 2790 CLOSE #0
 2800 VDU 26,7
 2810 CLS
 2820 REPORT
 2830 PRINT" ERROR at line ",ERL
 2840 END
```

Listing 13 2 Hex and ASCII dump utility Save as DUMPER

```
   10 REM Hex & ASCII Dump
   20 REM (C) Bruce Smith June 1986
   30 REM Advanced SRAM Guide
   40
   50 MODE 7
   60 %=0
   70 FOR P%=&3000 TO &31FF STEP8
   80 PRINT~P% "  ',
   90 FOR N%=0 TO 7
  100 IF P%?N%<16 PRINT'0"
  110 PRINT ~P%?N% "  ,
  120 NEXT
  130 PRINT'  ",
  140 FOR N%=0 TO 7
  150 A%=P%?N%
  160 IF A%<32 OR A%>127  PRINT" ', ELSE
 PRINTCHR$(A%),
  170 NEXT
  180 PRINT
  190 NEXT
```

Chapter Fourteen Language ROMs

The first three bytes of a paged ROM are referred to as its language entry point, the first byte will normally contain the JMP opcode, &4C, followed by the two-byte address of the beginning of the language coding If it is a service ROM, these three bytes should be set to zero

The normal way in which a language is entered is to type in a command that the ROM will recognise For this purpose all language ROMs must contain a service entry point to an interpreter that will attempt to recognise the command, for example, the command *FORTH might select a FORTH language ROM The service entry interpreter must be capable of recognising this command and then select itself as the new language ROM To do this the ROM must issue OSBYTE &8E, with the X register containing the ROM number It is important to remember that this OSBYTE call returns through the ROM's language entry point, so there is no real need to preserve registers as they are destroyed anyway' The coding to perform the language entry is

```
    LDA ≠&8E
    JSR &FFF4
```

The X register should already hold the ROM identity though this can always be extracted from &F4 if it is lost for some reason

To start up the selected language the MOS notes the number of the ROM so that it can reselect the language ROM when a 'soft break' is performed, and then displays the ROM title string to indicate the particular language is in use The error message vector is pointed towards the copyright message or version string if it is present, whereupon the

language point is entered with the accumulator containing the value 1 to indicate a normal start up
Once a language has been initialised it has 1024 bytes of workspace free for private use running in a single block from &400 to &800 in addition to the zero page locations normally associated with a language ROM between &00 to &8F The language program space exists between the Operating System High Water Mark (OSHWM) and the bottom of the currently selected screen mode
Language ROMs may also be entered by two other methods First by issuing an *FX142 call This call must be postfixed by a number which relates to the ROM socket number containing the language to be switched in Thus to select the language in ROM socket number 12, use

```
    *FX 142,12
```

A language may also be auto-booted by pressing the BREAK key in combination with another specific key To do this the service interpreter must trap the auto-boot service call, 3, issued by the MOS on BREAK, and test for its particular auto-boot key (This technique is explained in Chapter 10)

Absolute Musts

There are three things a language must do, otherwise it will cause the Master to 'hang up' First, interrupt requests must be enabled for the MOS to continue to work correctly, a simple CLI will perform this Second, the BRK vector, BRKV at &202, must also be set ready to handle errors as they occur All language ROMs must include error handling facilities, as even the simplest task such as an OSWRCH call can generate an error The technique of error handling is examined in chapter 15 Finally the stack pointer will be undefined so this should be re-initialised These three tasks require a minimum of code

```
    CLI                        \ enable IRQs
    LDX ≠&FF                   \ reset stack pointer
    TXS
    LDA ≠brkhandle MOD 256 \ get low byte error handing
                               \ entry
    STA &202                   \ store low-byte BRKV
    LDA ≠brkhandle DIV 256 \ get high byte of same
    STA &203                   \ and poke into BRKV high byte
```

On entering a language, the accumulator will contain a language entry code Normally these can be ignored though two will be of interest if the language ROM is to be compatible with the Electron The four entry codes are as follows

Accumulator=0 There is no language present and the Tube ROM is being called This call must not be intercepted other than by the Tube ROM itself

Accumulator=1 Normal entry to language

Accumulator=2 Request next byte of soft key expansion The key number is set using a call with the accumulator containing 3, and the byte result is in the Y register This entry call is applicable on the Electron only

Accumulator=3 Requesting length of soft key expansion The key number is held in the Y register and the length should be substituted for it Again, an Electron-only call

Language entry calls 2 and 3 are Electron-specific and should not be looked at by Master or BBC-only firmware

Languages and the Tube

Because of the popularity of BBC model B second processors and Master coprocessors it is essential that languages will run across the Tube This simply means that they are capable of relocating in the second processor and running correctly If you write your languages 'correctly' this is automatic But what is correctly? Well, it simply means that all the input/ouput processes should be performed using the MOS commands and memory should not be peeked and poked Thus the screen should be written to using OSWRCH and not by poking the ASCII character of a code there directly For example, the letter A should be printed at the current cursor position using

```
    LDA #ASC("A )
    JSR oswrch
```

and not by using poking such as

```
    LDA #ASC("A")
    STA screen+offset
```

To take advantage of the increased memory capacity offered by the second processor, a 'Hi' version of the language you are writing may be required This option is available simply by assembling your language coding so that it will run at a higher re-location address, &B800 for example as with Hi-BASIC The service entry point and its associated coding should remain assembled at the normal addresses as this is not copied across in the second processor by the Tube ROM and is required to function within the Master Such a Hi version of your language would not run in the normal Master however due to the change in absolute addresses Changing the addressing is done simply by resetting the value assigned to P% at the language entry, as defined by the address given at the language entry point For example

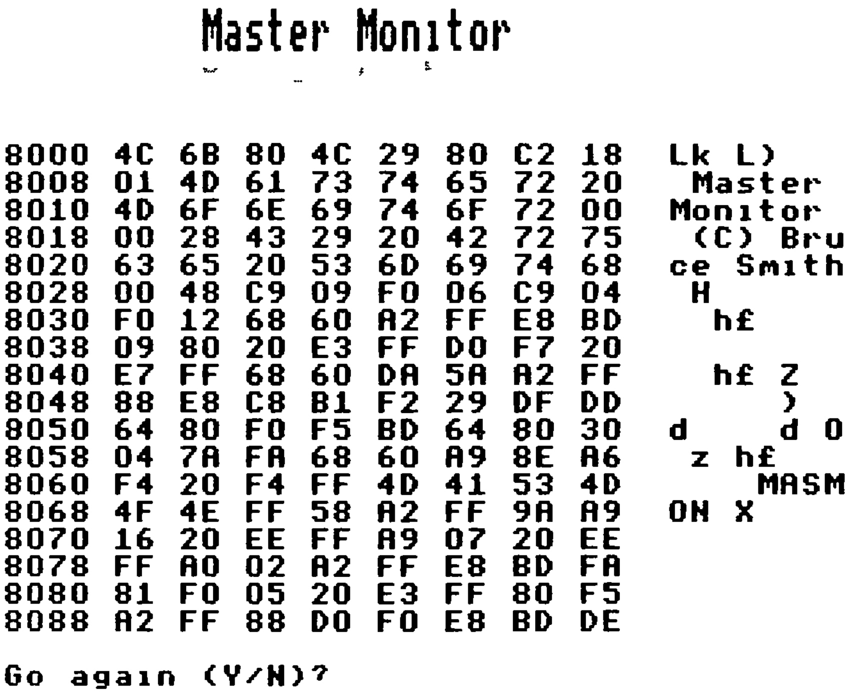
```
    FOR pass=4 TO 7 STEP 3
    O%=&5ØØØ   REM assemble at &5ØØØ
    P%=&8ØØØ   REM service code at &8ØØØ
    [OPT pass
    JMP language
    JMP service
    \ rest of service code is here
    \
    ]   REM exit at end of service code
    P%=&B8ØØ   REM repoint P%
    [OPT pass
     language
    \ language code here
    \ assembled for &B8ØØ
    ]
    NEXT pass
```

Listing 14 1 provides a very simple but working example of a language ROM The listing forms the machine code for a language that will give a hex and ASCII dump The language is called by *MASMON

When MASMON is entered, the screen clears and the title and copyright strings are displayed A text window is set up which ensures that these items remain on-screen throughout the languages operation You are then prompted to enter a start and end address in hexadecimal format - note that the '&' is already provided so you need only enter the hex digits themselves Once this has occurred, the area of memory between these two addresses is dumped to the screen The format for each line is current address, followed by the eight bytes from this address displayed in hex, and then in ASCII form (figure 14 1) If the byte is not displayable ASCII, a full stop is shown instead The listing may be halted by the CTRL-SHIFT key combination in the usual manner

When the listing has completed, you will be asked if you wish to display a further area of memory Pressing Y will reset the language and the process will repeat, otherwise BASIC will be re-entered

Having said that redirecting the BRK vector into your own language ROM is an absolute must, listing 14 1 does not do that! The reason for my madness will be looked at in the chapter 15 where errors will be discussed As it stands, the language is not capable of creating an error - although pressing the ESCAPE key will lock the language up The



```
                    Master Monitor

8000 4C 6B 80 4C 29 80 C2 18  Lk L)
8008 01 4D 61 73 74 65 72 20   Master
8010 4D 6F 6E 69 74 6F 72 00  Monitor
8018 00 28 43 29 20 42 72 75   (C) Bru
8020 63 65 20 53 6D 69 74 68  ce Smith
8028 00 48 C9 09 F0 06 C9 04   H
8030 F0 12 68 60 A2 FF E8 BD    h£
8038 09 80 20 E3 FF D0 F7 20
8040 E7 FF 68 60 DA 5A A2 FF    h£ Z
8048 88 E8 C8 B1 F2 29 DF DD        )
8050 64 80 F0 F5 BD 64 80 30  d    d 0
8058 04 7A FA 68 60 A9 8E A6   z h£
8060 F4 20 F4 FF 4D 41 53 4D      MASM
8068 4F 4E FF 58 A2 FF 9A A9  ON X
8070 16 20 EE FF A9 07 20 EE
8078 FF A0 02 A2 FF E8 BD FA
8080 81 F0 05 20 E3 FF 80 F5
8088 A2 FF 88 D0 F0 E8 BD DE

Go again (Y/N)?
```

Figure 14 1 Screen dump of MASMON display

reasons why this happens are discussed in the next chapter on errors

Now for a listing description PROCvars sets up the variables required by the language to operate, namely operating system calls and zero page storage for vectored addresses

The ROM header is assembled in lines 27Ø to 9ØØ It is much the same as for service ROMs, but there are some differences A language entry jump address must be assembled into the first three ROM header bytes (line 27Ø) the ROM type value must also be amended to include the now set language bit, bit 6, therefore the byte to be assembled is 11ØØ ØØ1Ø, or &C2 hex Line 29Ø takes care of this A service entry point, and therefore interpreter, must also be included to handle any *HELP service requests and unrecognised command requests This is assembled by lines 4ØØ to 79Ø The unrecognised command we are trying to trap is MASMON This is assembled in lines 88Ø to 99Ø and looked for by the interpreter assembled at lines 6ØØ to 73Ø Once recognised the language is entered through the language entry point by executing OSBYTE &8E (lines 81Ø to 84Ø)

The language entry point is entered via the JMP instruction located at &8ØØØ, which is in effect a jump to line 92Ø First things first, the MOS must be reset by re-enabling interrupts with CLI, followed closely by resetting of the stack (lines 93Ø to 95Ø) To see how important these processes are, try omiting these lines and running the re-assembled code!

The screen set-up routine and hex/ASCII dump output is controlled in a standard manner by lines 96Ø to 2I6Ø, using machine code subroutines based at lines 229Ø to 3IØØ

BASIC is re-entered by locating its ROM socket number via OSBYTE &BB OSBYTE &8E is used to select it in the standard way (lines 22IØ to 225Ø)

Service Call 42 (&2A)

The MOS issues service call 42 (&2A) before a ROM-based language starts up This gives other languages including the current one, plus service ROMs, the chance to do any necessary house-keeping

Listing 14.1 Master machine code hex and ASCII dump
Save as MASMON

```
   10 REM Implement a language ROM
   20 REM (C) Bruce Smith June 1986
   30 REM Advanced SRAM Guide
   40
   50 PROCvars
   60 PROCassemble
   70 PROCchecksum
   80 *SRWRITE 5000 +300 8000 7
   90 END
  100 DEF PROCvars
  110 mshigh=&50 mslow=&51
  120 lshigh=&52 lslow=&53
  130 temp=&54
  140 hibyte=&63 lobyte=&62
  150 hibegin=&61 lobegin=&60
  160 osrdch=&FFE0 osbyte=&FFF4
  170 oswrch=&FFEE osnewl=&FFE7
  180 osasci=&FFE3
  190 comline=&F2
  200 ENDPROC
  210
  220 DEF PROCassemble
  230 FOR Pass=4 TO 7 STEP 3
  240 P%=&8000   O%=&5000
  250 [
  260 OPT Pass
  270 JMP language
  280 JMP service
  290 EQUB &C2
  300 EQUB offset MOD 256
  310 EQUB 1
  320  title
  330 EQUS "Master Monitor"
  340 EQUB 0
  350  offset
  360 EQUB 0
  370 EQUS "(C) Bruce Smith
  380 EQUB 0
  390
  400  service
  410 PHA
  420 CMP ≠9
  430 BEQ help
  440 CMP ≠4
  450 BEQ unrecognised
  460 PLA
  470 RTS
```

Listing 14 1 continued

```
 480
 490  help
 500 LDX ≠&FF
 510  helploop
 520 INX
 530 LDA title,X
 540 JSR osasci
 550 BNE helploop
 560 JSR osnewl
 570 PLA
 580 RTS
 590
 600  unrecognised
 610 PHX
 620 PHY
 630 LDX ≠&FF
 640 DEY
 650  ctloop
 660 INX
 670 INY
 680 LDA (comline),Y
 690 AND ≠&DF
 700 CMP table,X
 710 BEQ ctloop
 720 LDA table,X
 730 BMI found
 740 \
 750  nothisrom
 760 PLY
 770 PLX
 780 PLA
 790 RTS
 800 \
 810  found
 820 LDA ≠&8E
 830 LDX &F4
 840 JSR &FFF4
 850 \ No return'
 860 \
 870 \ set up Command Table
 880  table
 890 EQUS  MASMON"
 900 EQUB &FF
 910
 920  language
 930 CLI
 940 LDX ≠&FF
 950 TXS
```

Listing 14 1 continued

```
  960 LDA ≠22
  970 JSR oswrch
  980 LDA ≠7
  990 JSR oswrch
 1000 LDY ≠2
 1010 LDX ≠&FF
 1020  langloop
 1030 INX
 1040 LDA heading,X
 1050 BEQ out
 1060 JSR osasci
 1070 BRA langloop
 1080  out
 1090 \
 1100 LDX ≠&FF
 1110 DEY
 1120 BNE langloop
 1130 \
 1140  copyloop
 1150 INX
 1160 LDA copyright,X
 1170 BEQ out2
 1180 JSR osasci
 1190 BRA  copyloop
 1200 \
 1210  out2
 1220 LDA ≠28
 1230 JSR oswrch
 1240 LDA ≠0
 1250 JSR oswrch
 1260 LDA ≠24
 1270 JSR oswrch
 1280 LDA ≠39
 1290 JSR oswrch
 1300 LDA ≠5
 1310 JSR oswrch
 1320 LDX ≠&FF
 1330 \
 1340  stloop
 1350 INX
 1360 LDA start,X
 1370 JSR osasci
 1380 BNE stloop
 1390 JSR inputaddr
 1400 LDA hibyte
 1410 STA hibegin
 1420 LDA lobyte
 1430 STA lobegin
```

Listing 14 1 continued

```
 1440 \
 1450 LDX ≠&FF
 1460  endloop
 1470 INX
 1480 LDA end,X
 1490 JSR osasci
 1500 BNE endloop
 1510 JSR inputaddr
 1520 LDA ≠13
 1530 JSR osasci
 1540 \
 1550  nextline
 1560 JSR address
 1570 LDY ≠0
 1580 LDX ≠7
 1590  hexloop
 1600 LDA (lobegin),Y
 1610 JSR hexout
 1620 JSR space
 1630 INY
 1640 DEX
 1650 BPL hexloop
 1660 LDA ≠134
 1670 JSR oswrch
 1680 \
 1690 LDY ≠0
 1700 LDX ≠7
 1710  ascloop
 1720 LDA (lobegin),Y
 1730 CMP ≠32
 1740 BCC spot
 1750 CMP ≠128
 1760 BCC jumpover
 1770 \
 1780  spot
 1790 LDA ≠ASC(" ")
 1800  jumpover
 1810 JSR oswrch
 1820 INY
 1830 DEX
 1840 BPL ascloop
 1850 \
 1860 LDA ≠&0D
 1870 JSR osasci
 1880 CLC
 1890 LDA lobegin
 1900 ADC ≠8
 1910 STA lobegin
```

Listing 14 1 continued

```
1920 BCC nocarry
1930 INC hibegin
1940  nocarry
1950 LDA lobegin
1960 CMP lobyte
1970 BCC nextline
1980 LDA hibegin
1990 CMP hibyte
2000 BCC nextline
2010 \
2020 JSR osnewl
2030 LDX ≠&FF
2040  goonloop
2050 INX
2060 LDA continue,X
2070 JSR oswrch
2080 BNE goonloop
2090  testkey
2100 JSR osrdch
2110 CMP ≠ASC("Y")
2120 BNE skipover
2130 JMP language
2140  skipover
2150 CMP ≠ASC("N")
2160 BNE testkey
2170 \
2180 LDA ≠26
2190 JSR oswrch
2200 LDA ≠12
2210 JSR oswrch
2220 LDA ≠&BB
2230 JSR osbyte
2240 LDA ≠&8E
2250 JMP osbyte
2260
2270 \ machine code subroutines
2280
2290  inputaddr
2300 JSR characters
2310 LDA mshigh
2320 JSR check
2330 ASL A
2340 ASL A
2350 ASL A
2360 ASL A
2370 STA temp
2380 LDA mslow
2390 JSR check
```

Listing 14.1 continued

```
 2400 ORA temp
 2410 STA hibyte
 2420 LDA lshigh
 2430 JSR check
 2440 ASL A
 2450 ASL A
 2460 ASL A
 2470 ASL A
 2480 STA temp
 2490 LDA lslow
 2500 JSR check
 2510 ORA temp
 2520 STA lobyte
 2530 RTS
 2540 \
 2550  characters
 [illegible]0 JSR osrdch
 [illegible]0 JSR osasci
 [illegible]80 STA mshigh
 2590 JSR osrdch
 2600 STA mslow
 2610 JSR osasci
 2620 JSR osrdch
 2630 JSR osasci
 2640 STA lshigh
 2650 JSR osrdch
 2660 JSR osasci
 2670 STA lslow
 2680 RTS
 2690 \
 2700  check
 2710 CMP ≠58
 2720 BCS atof
 2730 AND ≠15
 2740 RTS
 2750  atof
 2760 SBC ≠55
 2770 RTS
 2780 \
 2790  space
 2800 LDA ≠32
 2810 JMP oswrch
 2820 \
 2830  address
 2840 LDA ≠129
 2850 JSR oswrch
 2860 LDX ≠lobegin
 2870 LDA 1,X
```

Listing 14 1 continued

```
 288Ø JSR hexout
 289Ø LDA Ø,X
 29ØØ JSR hexout
 291Ø LDA ≠13Ø
 292Ø JSR oswrch
 293Ø RTS
 294Ø \
 295Ø  hexout
 296Ø PHA
 297Ø LSR A
 298Ø LSR A
 299Ø LSR A
 3ØØØ LSR A
 3Ø1Ø JSR digit
 3Ø2Ø PLA
 3Ø3Ø  digit
 3Ø4Ø AND ≠15
 3Ø5Ø CMP ≠1Ø
 3Ø6Ø BCC no
 3Ø7Ø ADC ≠6
 3Ø8Ø  no
 3Ø9Ø ADC ≠48
 31ØØ JMP oswrch
 311Ø
 312Ø \ ASCII string storage area
 313Ø
 314Ø  copyright
 315Ø EQUD &2Ø2Ø2Ø86
 316Ø EQUW &2Ø2Ø
 317Ø EQUS "(C) Bruce Smith 1986"
 318Ø EQUB 13
 319Ø EQUB Ø
 32ØØ  heading
 321Ø EQUB 141
 322Ø EQUB 131
 323Ø EQUD &2Ø2Ø2Ø2Ø
 324Ø EQUD &2Ø2Ø2Ø2Ø
 325Ø EQUS  Master Monitor
 326Ø EQUB 13
 327Ø EQUB Ø
 328Ø EQUB 141
 329Ø  start
 33ØØ EQUB 13Ø
 331Ø EQUS "Start  &"
 332Ø EQUB 129
 333Ø EQUB Ø
 334Ø  end
 335Ø EQUB 13Ø
```

Listing 14.1 continued

```
 336Ø EQUS "    End    &"
 337Ø EQUB 129
 338Ø EQUB Ø
 339Ø  continue
 34ØØ EQUB 13Ø
 341Ø EQUS "Go again (Y/N)? "
 342Ø EQUB Ø
 343Ø ]
 344Ø NEXT
 345Ø ENDPROC
 346Ø
 347Ø DEF PROCchecksum
 348Ø N%=Ø
  49Ø FOR X%=&5ØØØ TO &524Ø
   ØØ N%=N%+?X%
    Ø NEXT
    Ø IF N%=7Ø332 THEN ENDPROC
    Ø VDU 7
   4Ø PRINT"Assembler error!"
 355Ø STOP
```

Chapter 15
Errors

When writing any sideways ROM format program that ne
input from the user, other than just entering
command name, the ROM code must be capable of
identifying what is acceptable and what is not In the
latter case it must signal the fact to the user in the
way of an error message

For example, consider the two-line program

```
   1Ø MODE 2
   2Ø MOVE
```

When BASIC interprets this program it expects to find a number after the MODE command It looks to find one that is acceptable so performs a mode 2 command It moves onto the next line and identifies the MOVE command which it expects to be followed by two numbers, variables or expressions for evaluation In this case it finds none, just a carriage return Obviously this is not acceptable, so it signals the error message

```
    No such variable at line 2Ø
```

The BRK command is used on the Master to print error messages When the MOS sees a BRK it tries to print the string following on the screen until it encounters another BRK Listing 15 1 shows how the technique works Enter and run the program

Lines 5Ø and 6Ø simply signal an error with a customary beep Line 7Ø assembles the first BRK Line 8Ø assembles the error number that you are assigning to the error line 9Ø assembles the error message and finally line 1ØØ the final BRK instruction In fact we're not so much interested in BRK as its opcode, &ØØ, so EQUB Ø is equally as effective as BRK for that purpose Figure 15 1 shows how the error message is stored in memory

Address	Contents	Description
&3ØØØ	&A9	LDA≠
&3ØØ1	&Ø7	7
&3ØØ2	&2Ø	JSR
&3ØØ3	&EE	&FFEE
&3ØØ4	&FF	
&3ØØ5	&ØØ	BRK
&3ØØ6	&2Ø	Error code
&3ØØ7	&54	ASC"T
&3ØØ8	&68	ASC"h
&3ØØ9	&69	ASC'i"
&3ØØA	&73	ASC's'
&3ØØB	&2Ø	ASC" '
&3ØØC	&69	ASC"i
&3ØØD	&73	ASC's"
&3ØØE	&2Ø	ASC"
&3ØØF	&61	ASC"a
&3Ø1Ø	&6E	ASC"n
&3Ø11	&2Ø	ASC' '
&3Ø12	&65	ASC e"
&3Ø13	&72	ASC"r"
&3Ø14	&72	ASC"r'
&3Ø15	&6F	ASC'o
&3Ø16	&72	ASC'r
&3Ø17	&ØØ	BRK

Figure 15 1 Error message storage

When the Master executes a BRK instruction the following events take place The address of the BRK instruction plus 2 is pushed onto the hardware stack, high byte first The status register is pushed onto the stack Interrupts are disabled and the BRK flag is set, ie bit 4 of the status register Execution continues from the address found at &FFFE and &FFFF (In Master 3 2Ø MOS this is &E59E)

Once here, the following action takes place First the accumulator is saved in location &FC The stack is then pulled into the accumulator - this will be the status register It is then pushed back to leave a copy in the accumulator This is then ANDed with &1Ø to isolate bit four If the result is not zero then a BRK has occurred - otherwise it was an IRQ and an appropriate jump to IRQ1V is made The previously pushed address is removed from the stack, has one subtracted from it and stored in locations &FD and &FE This address now points to the error number, stored directly before the error message Location &F4 is read to get the currently active ROM and this is copied into &24A Service call 6 is then issued to each of the ROMs present On return the currently active language ROM is re-enabled, interrupts are re-enabled and a jump to BRKV is performed

Service ROM Errors

Errors within service ROMs are easy to process, however we must bear in mind that the currently active language ROM at this time (BASIC say) would be responsible for handling this error and as such would not expect t find it within another paged ROM So what the servic ROM must do is to copy the error details down into area of RAM that the language ROM can access The ar of memory reserved for this is in fact the error message buffer located at the very bottom of the hardware stack, &100 upwards This is easy to do

```
    LDY #0              \ BRK opcode
    STY &100            \ put it at &100
     errorloop
    LDA message,Y       \ get character
    STA &101,Y          \ save it on stack
    BEQ ifdone          \ exit if 0
    INY                 \ increment index
    BRA errorloop       \ do next byte
     ifdone
    JMP &100            \ execute BRK
     message
    EQUB 20             \ error number
    EQUS "Error"        \ error message
    EQUB 0              \ terminating BRK
```

Listing 15 2 sets up a service ROM with a single command, *CONVERT This will convert the hexadecimal value following it into binary and store the result in zero page locations &70 and &71 Two error conditions can occur here First, the number may not be a legitimate hex value - this is signalled with the 'Bad hex' error message Second, only numbers in the range 0 to &FFFF are allowed and so numbers bigger than this must be signalled and rejected with a 'Too big' error Enter the program and save as 'ERROR1' Try the program yourself, the hex number should not be prefixed with &

```
    *CONVERT            -  gives no error
    *CONVERT DS         -  gives 'Bad hex' error
    *CONVERT FFFFFF     -  gives 'too big' error
    *CONVERT EF         -  is legitimate
```

If you use this command from within a BASIC program you will notice that BASIC will add 'at line xx' onto the end of the error message This shows that BASIC is extending your error message to make it more explicit and is an example of the sort of responsibility language ROMs can take

Language ROMs

As already mentioned, it is the responsibility of the current language ROM to handle any errors that occur ithin it This is normally done by pointing BRKV at 2Ø2 and &2Ø3 to the appropriate handling routine

What a language ROM does when it receives an error is p to you As a rule however, it should print out the error message after a BRK so the user at least has some idea as to what is wrong and second to re-initialise the stack pointer

As we have seen, the vector at &FD is set by the MOS to point to the data immediately after the BRK that caused the error, so printing the error message is straightforward

```
     error
    LDY ≠Ø          \ initialise index
    LDA (&FD),Y     \ get error number
    STA errno       \ and save where appropriate

     loop
    INY             \ increment Y
    LDA (&FD),Y     \ get character
    BEQ ifdone      \ branch if done
    JSR &FFEE       \ print it
    BRA loop        \ do next byte
     ifdone
```

The stack pointer should be initialised as follows

```
    LDX ≠255
    TSX
```

Listing 15 3 produces a language ROM that expects an error! Basically anything other than a RETURN or an "*" is an error, with a suitable message printed out If you type in an asterisk, the language, suitably called

ErrorWise, will expect you to enter a star command, such as *HELP and will pass it to the command line interpreter Save the program as 'ERROR2' and enter the language with *ERRORWISE

ESCAPE

When you write any sort of ROM software you must look to see if the ESCAPE key is pressed This is even more important if you are looking at the keyboard for data If you don't then your ROM will lock up, crash if you prefer All escapes must be acknowledged with an OSBYTE 126

```
    LDA #126
    JSR &FFF4
```

There are two ways in which the ESCAPE key can b tested The best way is to use OSRDCH at &FFE0 to re the keyboard If ESCAPE is pressed then the carry fla will be set on return so that

```
    BCS escape
```

as in line 1020 of listing 15 3 is acceptable Less acceptable is to look at location &FF If bit 7 is set then ESCAPE has been pressed

```
    BIT &FF
    BMI escape
```

Error Numbers

If writing a language ROM, you can choose and use your own error numbers Service ROMs should be more discrete however and use numbers not used by the MegaROM, ie Basic, DFS and ADFS These can be found in the Advanced Reference Guide published by Acorn

Listing 15 1 Shows how error codes and messages are stored in ROMs

```
 10 REM BRK error demo
 20
 30 P%=&3000
 40 [
 50 LDA ≠7
 60 JSR &FFEE
 70 BRK
 80 EQUB 32
 90 EQUS "This is an error"
100 BRK
110 ]
120 CALL &3000
```

sting 15 2 Printing error messages from within a rvice ROM

```
 10 REM Error Test ROM
 20 REM (C) Bruce Smith June 1986
 30 REM Advanced SRAM Guide
 40
 50 lo=&70 hi=&71
 60 PROCassemble
 70 PROCchecksum
 80 *SRWRITE 5000 +200 8000 7
 90 END
100
110 DEF PROCassemble
120 osnewl=&FFE7
130 FOR pass=4 TO 7 STEP 3
140 P%=&8000   O%=&5000
150 [
160 OPT pass
170 EQUB 0
180 EQUW 0
190 JMP service
200 EQUB &82
210 EQUB offset MOD 256
220 EQUB 1
230  title
240 EQUS "Error Test ROM"
250 EQUB 0
260  version
270 EQUS " 1 00'
280 EQUB 0
290  offset
300 EQUB 0
```

Listing 15 2 continued

```
310 EQUS "(C) Bruce Smith"
320 EQUB 0
330  service
340 CMP #9
350 BEQ help
360 CMP #4
370 BEQ unrecognised
380 RTS
390 \
400  help
410 JSR osnewl
420 LDX #&FF
430 JSR helploop
440 JSR helploop
450 JSR osnewl
460 RTS
470 \
480  helploop
490 INX
500 LDA title,X
510 BEQ finish
520 JSR &FFE3
530 BRA helploop
540  finish
550 RTS
560
570  exit
580 PLY
590 PLX
600 PLA
610 RTS
620 \
630  complete
640 PLY
650 PLX
660 PLA
670 LDA #0
680 RTS
690  \
700  convert
710 EQUS 'CONVERT"
720 \
730  unrecognised
740 PHA
750 PHX
760 PHY
770 LDX #0
780  loop
```

Listing 15 2 continued

```
  79Ø LDA (&F2),Y
  8ØØ AND ≠&DF
  81Ø CMP convert,X
  82Ø BNE exit
  83Ø INY
  84Ø INX
  85Ø CPX ≠7
  86Ø BNE loop
  87Ø \
  88Ø STZ lo
  89Ø STZ hi
  9ØØ \
  91Ø JSR spaces
  92Ø \
  93Ø  nextchar
  94Ø LDA (&F2),Y
  95Ø CMP ≠13
  96Ø BEQ complete
  97Ø CMP ≠ASC" "
  98Ø BEQ end
  99Ø CMP ≠ASC'Ø'
 1ØØØ BCC bad
 1Ø1Ø CMP ≠&3A
 1Ø2Ø BCC digit
 1Ø3Ø CMP ≠ASC"A"
 1Ø4Ø BCC bad
 1Ø5Ø CMP ≠ASC"G"
 1Ø6Ø BCS bad
 1Ø7Ø SBC ≠&36
 1Ø8Ø \
 1Ø9Ø  digit
 11ØØ ASL A
 111Ø ASL A
 112Ø ASL A
 113Ø ASL A
 114Ø LDX ≠4
 115Ø \
 116Ø  aslrol
 117Ø ASL A
 118Ø ROL lo
 119Ø ROL hi
 12ØØ BCS large
 121Ø DEX
 122Ø BNE aslrol
 123Ø INY
 124Ø BNE nextchar
 125Ø \
 126Ø  end
```

Listing15 2 continued

```
1270 JSR spaces
1280 CMP ≠13
1290 BEQ complete
1300 LDX ≠0
1310 BEQ skip1
1320  bad
1330 LDX ≠badnum-size
1340 BNE skip1
1350 \
1360  large
1370 LDX ≠0
1380  skip1
1390 LDY ≠0
1400 STY &100
1410 \
1420  transfer
1430 LDA size,X
1440 STA &101,Y
1450 BEQ done
1460 INX
1470 INY
1480 BNE transfer
1490 \
1500  done
1510 JMP &100
1520 \
1530  size
1540 EQUB 20
1550 EQUS "ErrorROM  Too big"
1560 EQUB 0
1570  badnum
1580 EQUB 28
1590 EQUS "ErrorROM  Bad Hex
1600 EQUB 0
1610
1620  loop2
1630 INY
1640  spaces
1650 LDA (&F2),Y
1660 CMP≠ASC" "
1670 BEQ loop2
1680 RTS
1690
1700 ]
1710 NEXT pass
1720 ENDPROC
1730
1740 DEF PROCchecksum
```

Listing 15 2 continued

```
1750 N%=0
1760 FOR X%=&5000 TO &5100
1770 N%=N%+?X%
1780 NEXT
1790 IF N%=28080 THEN ENDPROC
1800 VDU 7
1810 PRINT"Assembler error'
1820 STOP
```

Listing 15 3 Demonstration of printing and handling errors from within a ROM

```
  10 REM ErrorWise Language ROM
  20 REM (C) Bruce Smith June 1986
  30 REM Advanced SRAM Guide
  40
  50 osrdch=&FFE0   buffer=&3000
  60 brklo=&202   brkhi=&203
  70 PROCassemble
  80 PROCchecksum
  90 *SRWRITE 5000 +200 8000 7
 100 END
 110
 120 DEF PROCassemble
 130 osnewl=&FFE7
 140 FOR pass=4 TO 7 STEP 3
 150 P%=&8000   O%=&5000
 160 [
 170 OPT pass
 180 JMP language
 190 JMP service
 200 EQUB &C2
 210 EQUB offset MOD 256
 220 EQUB 1
 230  title
 240 EQUS "ErrorWise"
 250 EQUB 0
 260  version
 270 EQUS " 1 00"
 280 EQUB 0
 290  offset
 300 EQUB 0
 310 EQUS "(C) Bruce Smith"
 320 EQUB 0
 330  service
 340 CMP ≠9
 350 BEQ help
```

Listing 15 3 continued

```
 360 CMP #4
 370 BEQ unrecognised
 380 RTS
 390 \
 400  help
 410 JSR osnewl
 420 LDX #&FF
 430 JSR helploop
 440 JSR helploop
 450 JSR osnewl
 460 RTS
 470 \
 480  helploop
 490 INX
 500 LDA title,X
 510 BEQ finish
 520 JSR &FFE3
 530 BRA helploop
 540  finish
 550 RTS
 560
 570  exit
 580 PLY
 590 PLX
 600 PLA
 610 RTS
 620 \
 630  command
 640 EQUS "ERRORWISE"
 650 \
 660  unrecognised
 670 PHA
 680 PHX
 690 PHY
 700 LDX #0
 710  loop
 720 LDA (&F2),Y
 730 AND #&DF
 740 CMP command,X
 750 BNE exit
 760 INY
 770 INX
 780 CPX #9
 790 BNE loop
 800 \
 810
 820 LDX &F4
 830 LDA #142
```

Listing 15 3 continued

```
 840 JMP &FFF4
 850
 860  language
 870 LDA ≠error DIV 256
 880 STA brkhi
 890 LDA ≠error MOD 256
 900 STA brklo
 910
 920  stackset
 930 LDX ≠255
 940 TXS
 950 CLI
 960 JSR &FFE7
 970
 980  mainloop
 990 LDA ≠ASC"="
1000 JSR &FFEE
1010 JSR osrdch
1020 BCS escape
1030 CMP ≠ASC"*"
1040 BEQ star
1050 CMP ≠13
1060 BNE doerror
1070 JSR &FFE7
1080 BRA mainloop
1090
1100  star
1110 JSR &FFE7
1120 LDA ≠ASC"*"
1130 JSR &FFEE
1140 LDY ≠blk DIV 256
1150 LDX ≠blk MOD 256
1160 LDA ≠0
1170 JSR &FFF1
1180 BCS escape
1190
1200 LDY ≠buffer DIV 256
1210 LDX ≠buffer MOD 256
1220 JSR &FFF7
1230 JMP mainloop
1240
1250  escape
1260 LDA ≠126
1270 JSR &FFF4
1280 BRK
1290 EQUB 1
1300 EQUS "ErrorROM  Escape"
1310 EQUB 0
```

Listing 15 3 continued

```
 1320
 1330  doerror
 1340 BRK
 1350 EQUB 2
 1360 EQUS "ErrorROM  Illegal Command"
 1370 EQUB 0
 1380
 1390
 1400  error
 1410 LDY ≠1
 1420 JSR &FFE7
 1430  loop
 1440 LDA (&FD),Y
 1450 BEQ end
 1460 JSR &FFEE
 1470 INY
 1480 BNE loop
 1490  end
 1500 JSR &FFE7
 1510 JMP stackset
 1520
 1530  blk
 1540 EQUB buffer MOD 256
 1550 EQUB buffer DIV 256
 1560 EQUB 20
 1570 EQUB 32
 1580 EQUB 127
 1590
 1600 ]
 1610 NEXT pass
 1620 ENDPROC
 1630
 1640 DEF PROCchecksum
 1650 N%=0
 1660 FOR X%=&5000 TO &5108
 1670 N%=N%+?X%
 1680 NEXT
 1690 IF N%=31281 THEN ENDPROC
 1700 PRINT"Assembler error'"
 1710 VDU 7
 1720 STOP
```

Glossary

absolute address	an exact address, ie &2ØØØ is an absolute address
absolute workspace	workspace given over to ROMs which may be used freely by all ROMs
accumulator	the main register of the 65Ø2/65I2 microprocessor
ADFS	Advanced Disc Filing System
ANFS	Advanced Network Filing System
assembler	mnemonic language in which assembly language programs may be written Part of the BASIC ROM which converts assembler mnemonics into machine code
auto-boot	pressing the SHIFT and BREAK keys together will allow a previously-written 'BOOT file to be run directly
ASCII	American Standard Code for Information Interchange - the character coding scheme whereby each number, letter or symbol key has its own special code that may be printed to display the character
bank	a sideways RAM bank - one of several areas of memory similar in size and memory address
battery-backed	memory that has a charge applied by a small battery when the machine is turned off thereby preserving its contents

binary	a numbering system to a base of 2 using only the 1 and Ø digits
bit	a single digit in a binary number
boot	to initialise/start-up a computer or program
branch	to move the operation of a program to another point, normally calculated as an offset from the current position
BRK	the assembler mnemonic for the BRK (break) operation
BRKV	vector through which control is passed when the computer executes a BRK instruction
buffer	area of memory used to store incoming/outgoing information
bump	to increment by one
byte	the smallest area of memory - capable of holding a number in the range Ø to 255 inclusive
carry (flag)	flag in the status register used to indicate overflow or underflow during addition or subtraction
channel	memory path along which information is passed
checksum	utility which counts the number of commands in a program This can be compared to a given value
CMOS	Complementary Metal Oxide Semiconductor - a family of chips with low power consumption
configure	to define a system to personal needs
co-processor	board which fits inside the main cas and takes over the computer's main tasks
crash	to cease operating as expected, normally caused by a program malfunction
CRC	cyclic redundancy check - a common error detecting code
debugging	process of weeding out errors in a program
DFS	Disc Filing System controls access of micro to disc drive
directory	a specially-defined area of a disc into which files can be saved
DNFS	Disc Network Filing System - a combined DFS and NFS chip

dump	paper copy (usually) or screen display of memory or file contents
dynamic workspace	memory claimed by ROM for its own use The amount claimed will vary from ROM to ROM and the memory boundary will move, ie, is dynamic
entry point	point in a program where control is transferred to, ie from where it begins its operation
EPROM	Erasable/Programable Read Only Memory - a chip which may be programmed with an EPROM programmer The contents are permanent unless erased by an ultra violet light source
EQUB/EQUW	commands used by the assembler in BASIC 2 and later versions, to assemble specific items of data
error number	code number which defines the last error that occurred Obtained with PRINT ERR
execution address	the point at which control is transferred to carry out the task of the program or command
explode	to load the character set from ROM into main user memory thus allowing it to be changed
extended vector entry	a means whereby a vector may be redirected into ROMs other than the current language thus enabling them to perform tasks along with the current language ROM
filing system	ROM chip that controls the flow of data to and from storage medium such as disc, net or cassette
firmware	programs supplied in chip form
flag	a byte, bit or variable that is used to signal that a condition has or has not be met with
font	design of the letters in a character set, eg standard, italic, bold etc
garbage	undefined or random memory contents
handle	number assigned to the current file by the filing system

hash header	an abbreviated form of header used in the ROM Filing System, so called because it uses the ≠ symbol
hex	hexadecimal - a number system based on 16
high-level language	language not written in the native language of the computer (ie not machine code)
hang-up	micro becomes unresponsive unless the BREAK or CTRL-BREAK keys are pressed to reset the system
header	section of code containing a CRC and file information at the start of a file stored on a filing medium
Hi language	languages that have been customised to run in a second or co-processor, eg Hi-BASIC
hidden RAM	12k area of memory that is used by the MOS and ROMs It is not available for normal use and hence is 'hidden'
high byte	the upper, higher value byte in a two-byte number
housekeeping	to tidy up and do chores required to keep operation smooth running
id	identifier each ROM slot has an id number associated with it
image	copy of a ROM on disc or cassette, etc
implode	opposite of explode, ie, to use the ROM-based character set
increment	add one to the contents of a number or register
initialise	reset/set before continuing
internal key number	each ASCII character has a specific number for the micro's own use
interpreter	section of machine code that is capable of recognising/ identifying a sequence of ASCII characters
interrupt	signal generated by a chip or external device that stops the microprocessor's operation
IRQ	Interrupt Request - an interrupt that may be ignored by the processor under certain conditions
JMP/JSR	assembler mnemonics JMP is JuMP to a given address, JSR is Jump Save Return, a form of GOSUB

anguage ROM	ROM containing a language
ow byte	lower value byte of a two-byte number
achine code	native language of all microprocessors
ain Tube initialisation	main routine used to set up and define the operation of the Tube
asking	technique whereby the bits within a byte may be manipulated using a logical operator, eg, AND
egaROM	main sideways ROM within the Master containing all of the ROMs supplied Referred to as the Megabit ROM because of its 128k size (1 million bits)
onitor	watch, observe and alter the contents of memory Software used to watch the operation of a machine code program
OS	Machine Operating System - the firmware that controls the micro
et	network filing systems, eg Econet, whereby machines are linked together via a series of cables and share one main disc drive
FS	Network Filing System
ibble	group of 4 bits, half a byte
MI	Non-Maskable Interrupt - this may not be ignored by the microprocessor
ffset pointer	index which, when added to an absolute address, will point to a selected address
pcode	term given to the operation code for a byte of machine code
PT	variable used to assign the output required during the assembly of a machine code program
SHWM	Operating System High Water Mark - usually the same value as PAGE, but the first byte free for use above the workspace stored at the beginning of memory
AGE	address at which a program will be stored
aged ROM	another term for sideways ROM
aging	process of selecting and

	deselecting sideways ROMs
parameter block	area of memory into which information is stored to pass it to the MOS when an OS call is used
peek	examine memory contents
poke	alter memory contents
polling interrupt	process of ascertaining which device caused an interrupt by asking each one in turn
power-up	switching on the micro
private RAM	memory used privately by a ROM - no other ROM may use the private workspace
pull	remove an item of data from a stack
push	place an item of data on a stack
RAM	Random Access Memory - volatile memory used to store programs and information for use by the computer The contents are erased when power is removed
RAM bank	see bank
register	special location within the microprocessor
reset	initialise the system
second processor	see co-processor
service call	message issued by the MOS The resulting action is defined by a code number in the accumulator
service entry point	point in a ROM through which a service call is processed
service ROM	ROM that is only capable of actioning service calls, ie it does not have a language - only * commands are recognised
sideways RAM	technique whereby ROM images can be switched in and out of the same area of memory
soft reset	reset that is only partial Happens when BREAK key is pressed
stack	area of memory onto which data can be pushed and pulled in the form of a linear list which appears to move up and down on each push and/or pull
static workspace	reserved memory that does not change in size and is available for use by all ROMs (ie the Master

	version of BBC micro's absolute workspace in hidden RAM)
table	list of commands, addresses or data for use by a program
toggle	single command that works like a switch Use of the command will set or reset depending on the current status, ie if set it will reset and vice versa
TOP	BASIC's variable that stores the address of the next free byte after the program
transfer routine	small program or subroutine that will move data from one point to another
Tube	registered trademark of Acorn used to describe the connection mechanism between computer and second or co-processor
type table	table stored in memory in which the ROM type is held by the MOS
USERV	USER Vector address
vector	two-byte location which contains an address
volatile (RAM)	contents are erased when power is removed
workspace	area of memory in which ROMs may perform calculations
X register	store in the 65Ø2/65I2 microprocessor used by assembler
Y register	store in the 65Ø2/65I2 microprocessor used by assembler
zero page	area of memory whose address runs from &ØØ to &FF
65Ø2 series	family of microprocessors which includes the 65Ø2 and 65I2 chips used in the BBC and Master series

Appendix A
Hex and Binary

This book makes no attempt to teach assembly language to readers, and those who are unclear on this topic should look to the books in the reference list However, as a grasp of assembler and binary techniques is necessary to make the most of this book, a brief summary is given here

Computers work by manipulating numbers, though in most instances this is transparent to the user, especially in high level languages such as BASIC However, the numberin system used by computers is not a decimal one It is base on the binary number system

In binary there are only two numbers, namely Ø and 1 At first sight this seems to be a severe limitation But consider a number system that you are familiar with - the decimal system Here there are just ten digits, Ø to 9 inclusive However we can form larger numbers by formin groups of numbers, for example, 12, 345 and 5678 are large numbers, using the base numbers, Ø to 9, as buildin blocks In the same way we can use the base digits Ø and 1 to form larger binary numbers, for example, Ø1, 1Ø1Ø, ØØ1ØØ1 and so forth

The next step is to learn how these numbers of binar digits, or bits, are read Again consider the decimal system and the number 1234 As we are working to a base o 1Ø, the value of each digit increases ten fold as we move from right to left Therefore reading from right to left we have

4	units, or 4*1=4
3	tens, or 3*(1Ø*1)=3Ø
2	hundreds, or 2*(1Ø*1Ø*1)=2ØØ
1	thousand, or 1*(1Ø*1Ø*1Ø*1)=1ØØØ

adding this together gives

    1000+200+30+4=1234

It is no different in binary except that we are now working to a base of 2, therefore numbers increase by a factor of two as we move from right to left Consider the binary number 1101 Reading from right to left we have

1	unit, or 1*1=1
0	twos, or 0*(2*1)=0
1	fours, or 1*(2*2*1)=4
1	eights, or 1*(2*2*2*1)=8

adding these together gives

    8+4+0+1=13

therefore 1101 is 13 in decimal

Each piece of computer memory, into which an item of data can be stored is called a byte There are eight binary digits in a single byte, or in the jargon, eight bits in a byte Therefore the largest binary value that can be stored in a single byte is

    11111111

If you multiply this out as above you find that this is 255 decimal

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

Table A1 Number conversion

ASR-N

As you can imagine when dealing with computers using a numbering system that is simply lines of ones and zeroes is somewhat long-winded and very prone to error So a numbering system called hexadecimal was introduced The hexadecimal system is calculated to a base of 16 This is not as difficult as it may at first seem

First, we cannot just use the numbers Ø to 9 to represent all possible 16 digits So to represent the decimal equivalents of 1Ø to 15 we use the letters A to F inclusive Table A1 sumarises the decimal, binary and hexadecimal equivalents

Table A1 will enable you to convert any binary number to hexadecimal and vice versa, believe it or not!

If you look at the binary column you will see that I have always used four digits This is because the largest single hex (short for hexadecimal) digit is represented in four bits, ie F=1111 I mentioned above that a byte is eight bits wide, therefore the value of a byte can be represented by just two hex digits, simply by converting the byte into two halfs and using the above table

Consider the binary number, 11ØØØ1Ø1 Break this into two halves of four bits, called nibbles, and we have

```
        11ØØ and Ø1Ø1
```

Use the above table and we see that

```
        11ØØ = C  and   Ø1Ø1 = 5
```

therefore 11ØØØ1Ø1 in binary is C5 in hex To distinguish that it is a hex number we place an & in front of it, &C5

To convert a hex number into binary we work in the opposite direction Thus the number &DA simplifies to

```
    D = 11Ø1
    A = 1Ø1Ø
    therefore &DA = 11Ø11Ø1Ø
```

Converting between hex and decimal and vice versa is less straightforward It can be done using the Master For example, typing

```
    PRINT &DA
```

would cause the Master to print the decimal equivalent of &DA Typing

```
    PRINT ~123
```

would print the hexadecimal value of the decimal 123

Obviously we have just dealt with single-byte numbers But multibyte numbers are converted in exactly the same fashion Just break the number down into single bytes and then proceed as normal As an example the number &CAFE becomes

```
    C  = IIØØ
    A  = IØIØ
    F  = IIII
    E  = IIIØ
```

Therefore &CAFE = IIØØ IØIØ IIII IIIØ Note that the number is split into its basic nibbles This makes reading and manipulating it that much easier on the eyes'

Appendix B Conversions and Compatibility: BBC and Electron

The techniques contained in this book are primarily written for the Master series of computers However, bearing in mind that the BBC series of computers is an evolutionary one and that compatibility is an important feature of that evolution, many programs will run with minimal changes Certainly the techniques discussed are applicable The notes that follow will point out the main areas of incompatibility with solutions where possible Table B1 lists the programs that will work providing suitable adaptations, as detailed below, are performed

Assembler

The Master is based on the 65C12 microprocessor chip This has a slightly better machine code instruction set than the 65Ø2 and 6512 microprocessors that form the heart of the BBC B and BBC B+ micros The assembler listings presented here are written to take advantage of the increased instruction set

Instructions used within listings that are not supported by the 65Ø2 and 6512 microprocessors in the BBC B, BBC B+ and BBC B+128 micros are

```
PHX - Push X onto stack
PHY - Push Y onto stack
PLX - Pull X from stack
PLX - Pull Y from stack
STZ - Store zero at location
```

Program	BBC B+128k	BBC B+	BBC B
1 1	Yes	Yes	Yes
2 1	Yes	Yes	Yes
2 2	Yes	Yes	Yes
3 1	Yes	Yes	Yes
4 1	Yes	Yes	Yes
4 2	Yes	Yes	Yes
4 3	Yes	Yes	Yes
4 4	No	No	No
5 1	Yes	Yes	Yes
5 2	Yes *	Yes*	Yes *
6 1	Yes	Yes	Yes
6 2	Yes	Yes	Yes
6 3	Yes	Yes	Yes
6 4	Yes	Yes	Yes
7 1	Yes	Yes	Yes
8 1	No	No	No
8 2	No	No	No
9 1	No	No	No
10 1	Yes	Yes	Yes
11 1	Yes	Yes	Yes
13 1	Yes	Yes	Yes
14 1	Yes	Yes	Yes
15 1	Yes	Yes	Yes
15 2	Yes	Yes	Yes

* Character font must be exploded with *FX20,6

Table B1 Program compatibility

However, these are simple to simulate, the equivalents of each are

Master	BBC B,B+,B+128
PHX	TXA PHA
PHY	TYA PHA
PLX	PHA TAX
PLY	PHA TAY
STZ location	PHA LDA ≠0 STA location PHA

Assembler options 4 to 7 rely on BASIC 2 in non-Master machines The only easy way to simulate these in BASIC 1 is to add an offset to all absolute addresses

```
 10 d%=&3000 P%=&5000
100 STA base+d%
```

would ensure address in the &8000 range (details later)

Important Note Altering assembler listings will alter the checksum value calculated, so giving an 'Assembler error'' message This should be ignored or the checksum calculation routine left out of the program

Service Calls

The following service calls are unique to the Master series

&15 - Polling interrupt
&18 - Interactive *HELP
&21 - Indicate static workspace in hidden RAM
&22 - Claim private workspace
&23 - Top of static workspace
&24 - Indicate workspace requirements
&25 - Inform MOS of filing system details
&26 - Close all files
&27 - Reset has occurred
&28 - Unknown *CONFIGURE
&29 - Unknown *STATUS
&2A - ROM language starting up

Calls &24 and &22 have direct equivalents in calls &1 and &2 in the BBC B series computers and these should be used instead, claiming normal RAM as static and private RAM
It is likely that Acorn may release an upgraded version of the MOS to include these calls at a future date In such a case the above calls may well be of use

BBC B+ 128k

This micro has the same memory arrangements as the Master, except that the hidden RAM is not used as per the Master Refer to your User Guide for possible applications for this space As the BBC B+ 128k contains four 16k banks of sideways RAM, the sideways RAM utilities descibed, ie *SRWRITE, *SRLOAD, *SRREAD and *SRSAVE are all implemented However the ROM identities are different, sideways RAM banks have identities W, X, Y and Z Thus references to ROM identities 4,5,6 and 7 in text and listings should be changed to W, X, Y or Z

BBC B and B+

These micros need to have sideways RAM fitted Consult magazine reviews and advertisements for details on the types available The sideways RAM utilities detailed above are not present, however your RAM board should explain how to transfer ROM images from disc into the RAM In addition to assembler changes you will need to alter the line containing the sideways RAM utility, typically *SRWRITE This need just be changed to a simple *SAVE to save the code generated by the assembly listing Thus

```
    *SAVE name <start addr> <end addr>
```

where 'name' is the assigned filename The file can then be loaded into sideways RAM with the appropriate utility Some sideways RAM boards allow you to assemble directly into the sideways RAM itself In such cases *SAVE is not necessary P% can be set directly to &8ØØØ, O% can be omitted and the OPT parameters can be adapted to Ø and 3 thus

```
    FOR pass=Ø TO 3 STEP 3
```

For assembling in this manner you will need to use a special routine to read the ROM image back into memory prior to saving it to tape or disc Again your ROM board instruction manual should contain details If not, simply place the OSRDRM routine at &FFB9 in a suitable loop Details of this call can be found in Chapter 12

Electron

There are some SRAM boards available for the Electron, notably those marketed by Advanced Computer Products (ACP) and Solidisk (incorporated with their Electron DFS - EFS) Both of these require the Plus1 to be fitted

To convert the programs to run then follow the conversion notes above for the BBC B micro BASIC 2 is fitted on the Electron therefore both OPT and EQU functions are present

OPT 4 to 7

BASIC version 4 as supplied with the Master series, and BASIC version 2 as supplied with later versions of the BBC B, B+ and B+ 128k micros contain extra assembly options that cater directly for offset assembly

It is not normally possible to assemble ROM images into sideways RAM Straight assembly into other areas of memory is of little use as all absolute addresses will not be correct Consider the following short segment of assembler

```
    FOR pass=Ø TO 3 STEP 3
    P%=&5ØØØ
    [OPT pass
     start
    JMP language
    JMP service
```

```
     service
    ]   NEXT pass
```

If the label service was offset from start by &52 byte then the code assembled by JMP service will be the equivalent of

```
    JMP &5Ø52
```

This could not be used correctly within sideways RAM The address that should be assembled, needs to be

```
    JMP &8Ø52
```

One way around this is to add an offset to all absolute addresses, thus the above code becomes

```
    FOR pass=Ø TO 3 STEP 3
    P%=&5ØØØ   D%=&8ØØØ-P%
    [OPT pass
     start
    JMP language+D%
    JMP service+D%

     service
    ]   NEXT pass
```

Now when JMP service is evaluated it will have &3ØØØ added to it to give the correct address

This is not an elegant and friendly solution The BASIC assembler now caters for this OPTs 4, 5, 6, and 7 are the direct equivalents of OPTs Ø, 1, 2, and 3 except that offset assembly takes place Now code is listed on the screen as though it is assembling to P% but in actual fact it is being assembled at the location pointed to by O%

```
    FOR pass=4 TO 7 STEP 3
    P%=&8ØØØ   O%=&5ØØØ
    [OPT pass
     start
    JMP language
    JMP service

     service
    ]   NEXT pass
```

In the above program machine code is generated correctly for &8ØØØ, but is in fact stored at &5ØØØ

Appendix C
Listings Details

Details of the 25 programs contained in this book are listed below, along with any special commands or procedures they include which could be used in your own programs The suggested save names in the chapters are also given in brackets

Listing 1 1 Simple sideways RAM demonstration (DEMO)
This listing shows just how easy it can be to produce a sideways RAM image It does not use assembler, instead it uses machine code as DATA statements

Commands added	*BEEP produces a beep on the speaker
	*HELP processes a simple *HELP message
Procedures	PROCread reads DATA
	PROCchecksum checks program entered
Other features	First use of *SRWRITE command

Listing 2 1 Read ROM table address (TABLE)
Demonstrates use of OSBYTE 170 to read start address of ROM table in RAM

Listing 2 2 Form ROM header (HEADER)
Illustrates how a standard ROM header is produced

Commands added	*HELP processes a simple *HELP message
Procedures	PROCgetstring inputs title and copyright strings
	PROCassemble assembles header using above information
Other features	Illustrates use of service entry

Listing 3 1 Trace ROM (TRACE)
Forms sideways ROM image to show what service calls are being issued by the MOS as and when they are

Other features	Binary to ASCII hexadecimal string conversion routine to print service call number as a two-digit hex number

Listing 4 1 Simple *HELP ROM (HELP1)
Shows how service call 9 is trapped to output a standard *HELP message defined by the title string of the ROM

Commands added	*HELP processes a simple *HELP message
Other features	Provides standard print routine for *HELP response Outputs ROM title string

Listing 4 2 Print version number on *HELP (HELP2)
Enhanced version of listing 4 1 It prints the ASCII version number in addition to the standard *HELP message

Commands added	*HELP processes a simple *HELP
Other features	Recodes *HELP printing algorithm more efficiently

Listing 4 3 Extended *HELP (HELP3)
Shows how to make *HELP response more informative In addition to printing title string message, it prints a string called 'Command' When *HELP COMMAND is entered the ROM will respond with a description of commands that would be contained within the ROM image

Commands added	*HELP prints title message version number *HELP COMMANDS prints details of commands that may be held within that ROM
Other features	Illustrates use of a simple one-command interpreter Shows how characters may be forced to upper case Uses MOS vector at &F2 Shows how marker bytes may be used Demonstrates how to preserve and restore processor registers This listing is the basis for many in the book

Listing 4 4 Interactive *HELP (HELP4)
Shows how service call 24 may be trapped to provide interactive *HELP messages, perhaps to print more information should it be required by the user

Commands added	*HELP prints title string and version number
Other features	Traps service call 24 Asks if you wish more details about the ROM If reply is Y then more information is printed by the print routine, else ROM returns control

Listing 5 1 Test interpreter (INTERP)
Shows how three new commands can be added This demonstrates the standard way of interpreting commmands entered at the keyboard

Commands added	*MODERN bleeps speaker initially *STANDARD bleeps speaker initially *ITALICS bleeps speaker *HELP prints title string, version *HELP COMMANDS prints extended help
Other features	First use of service call 4 trapping Shows construction of command and address table Illustrates use of marker bytes and use of status register flags to indicate where you are in the command table Provides interpreter routine Uses search and compare routine to compare command entered with commands in the command table Provides 'move on' routine to search for next command in table Shows how command execution address may be extracted from command table and jumped to

Listing 5 2 Command coding (MODERN)
This listing is added to listing 5 1 and provides you with a 'modern' style character font that can be used in all modes except mode 7

Commands added	*MODERN selects modern characters *STANDARD reselects standard font *ITALICS produces bleep on speaker *HELP prints title string, version number
Other features	Shows how look-up and data tables can be used Routines provided to save and restore zero page workspace onto stack

Listing 6 1 OSBYTE ROM (OSBYTE)
Implements a new OSBYTE call number &64 to convert the binary value in X to a two-digit ASCII hex value returned in X and Y

Commands added	*HELP prints title and version number *HELP OSBYTE prints OSBYTE call details
Other features	Traps service call 7 Uses binary to ASCII hex conversion routine

Listing 6 2 Test new OSBYTE call (OSBTEST)
This routine shows how easy it is to use the new OSBYTE call provided by listing 6 1

Listing 6 3 OSWORD ROM' (OSWORD)
Implements new OSWORD call, number &65, to convert, and, if required, print two binary numbers into a ASCII hex string

Commands added	*HELP prints title string, version number *HELP OSWORD prints OSWORD call details
Other features	Two-byte binary to ASCII hex conversion routine Illustrates how to place and extract details from a parameter block How to use sign bytes in parameter block

Listing 6 4 Test new OSWORD call (OSWTEST)
Shows how to use new OSWORD call provided by listing 6 3

Listing 7 1 Extended vector ROM (VECTOR)
Demonstrates how to set up an extended vector to point into a sideways ROM It resets USERV

Commands added	*HELP prints title string and version number *HELP VECTORS prints extended vector details *ON turns extended vector on *OFF turns extended vector off
Other features	Shows use of ROM extended vector table Illustrates resetting of MOS vectors to point into a sideways ROM, and how to reset them again

Listing 8 1 Polling interrupt ROM (POLLING)
Shows how to trap service call 21 after *FX22 issued

Commands added	*HELP prints title string and version number *HELP POLLING prints polling details
Other features	Shows how interrupts can be caught 100 times per second to increment a counter Shows how to increment two-byte number

Listing 8 2 Print date on reset (TIME)
Traps reset service call, number 39, and uses it to print the date onto the screen Thus each time a hard reset is performed the date will be displayed as well as the standard start-up messages

Commands added	*HELP prints title string, version number *HELP DATE prints date details
Other features	Reads real time clock using OSWORD &E Illustrates trapping of service call 39

Listing 9 1 Configure and status ROM (DATE)
This program adds a new *CONFIGURE and *STATUS option to the ones already existing Namely whether or not to display the date on a reset as detailed above

Commands added	*HELP prints title string and version *HELP DATE prints configure/status details *CONFIGURE DATE ON/OFF configures date option so it is either on or off *STATUS DATE displays current date status
Other features	Shows use of service calls 40 and 41 and how to use battery-backed bytes allocated to sideways ROM Use of OSWORD &E to read real-time clock

Listing 10 1 Auto-boot ROM (BOOT)
How ROMs may be booted to perform specific tasks by pressing another key in addition to SHIFT-BREAK

Commands added	*HELP prints title string and version *HELP BOOT prints Boot options available
Other features	Shows use of service call 3 Provides boot facilities for choosing ROM filing

system and to catalogue disc How to use OSBYTE &8A to insert commands into input buffer

Listing 11 1 Private workspace ROM (PRIVATE)
Claiming and using private ROM workspace in hidden RAM

Commands added	*HELP prints title string and version *HELP COMMANDS prints command details *PUSH saves locations &70 to &8F in private ROM workspace *PULL transfers *PUSHed bytes from private ROM workspace back into locations &70 to &8F
Other features	Shows use of service calls 34 and 36 How to claim 256 bytes of private ROM workspace within hidden RAM Shows how to use private ROM workspace and ROM workspace table Routines 'writeon' and 'writeoff' supplied to enable hidden RAM workspace to be used

Listing 12 1 Read title string from ROM (READ)
Demonstrates OSRDRM

Listing 13 1 ROM Filing System (RFS) formatter (ROMFS)
Converts any BASIC programs into a 16k ROM image Programs can be loaded from sideways RAM directly into memory using the ROM filing system (RFS)

Commands added	*HELP prints title string and version
Other features	Shows use of service calls 13 and 14 How to calculate header and program checksum values Formatting of BASIC programs into RFS format
Procedures	PROCformat controls main formatting PROChandle formats multi-block code PROCfilehead forms block header and calculates header check (CRC) PROCgetdata reads program from disc and places it in ROM image PROChash creates hash header PROCassemble assembles machine code to calculate CRC PROCromhead assembles ROM header with service calls PROCnottape reads file catalogue PROCsave saves ROM image to disc, etc

Listing 13 2 Hex and ASCII dump utility (DUMPER)

Listing 14 1 MASMON language ROM (MASMON)
How to write a simple language ROM The example is MASMON the Master Monitor, a machine code hex and ASCII dump program

Commands added	*HELP prints title string and version *MASMON enters the larguage ROM
Other features	Shows use of language entry point Use of OSBYTE &8E Hex and ASCII dump routine How to convert ASCII hex string into a two-byte binary number

Listing 15 1 BRK errors (BRK)
Small assembly language program showing how error codes and error messages are stored within ROMs

Listing 15 2 Error test ROM (ERROR1)
How to print error messages from within a service ROM

Commands added	*HELP prints title string and version *CONVERT converts following hex number into a two-byte binary value
Other features	Supplies two new errors ErrorROM Too Big, greater than &FFFF ErrorROM Bad Hex, number not hex Sets up error table and shows use of error numbers Uses stack as error buffer and provides routine to copy message from ROM onto stack

Listing 15 3 Errorwise language ROM (ERROR2)
How to print and handle errors from within a language

Commands added	*HELP prints title string and version *ERRORWISE enters language ROM
Other features	Shows how to claim BRKV How to restore ROM after an error How to implement OSCLI within a ROM Further example of a language ROM

Program Disc

Note that all these programs are available on a disc from Victory Publishing Please turn to order form after the Index

Appendix D
Links

When you receive your Master, the sideways RAM is set up and ready to use However, if you wish to use any of the internal ROM sockets this can only be done at the loss of some SRAM The option you decide to take up is defined by the position of two links inside the Master's case on the main circuit board The links are LK18 and LK19

To change the links you will need to remove the top of the Master case - this is done by undoing the four fixing screws marked 'fix' on the underside With the lid removed and the keyboard facing you the four ROM slots can be clearly seen to the right side immediately

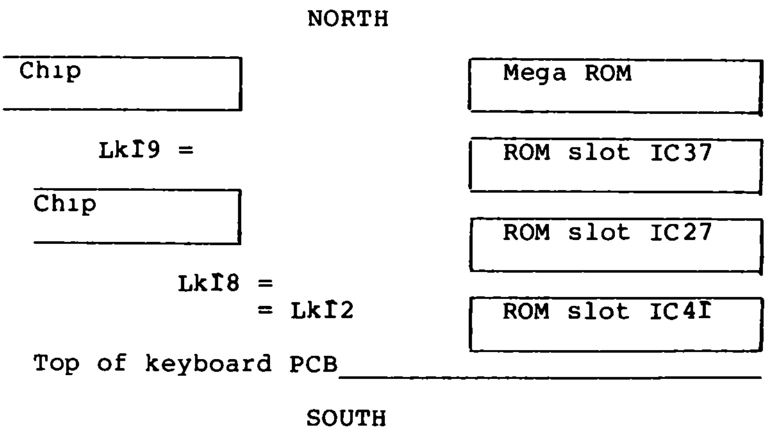


Figure D1 Position of links on circuit board

above the keyboard The MegaROM can be clearly seen in the topmost of these sockets Link LK19 can be found to the left and slightly below the MegaROM while link LK18 is to the left of the bottom most 'empty' ROM socket, and above link LK12 The links are marked on the main circuit board in white Take a look at figure D1

<u>Link 18</u>

When fitted in the WEST position, this link cause 16k of RAM to appear in each of the SRAM memory slots numbers 6 and 7 When fitted in the EAST position, a ROM up to 32k in size occupying slots 4 and 5 may be plugged into the socket labelled IC41

<u>Link 19</u>

When fitted in the WEST position, this link will allow 16k of SRAM to appear in slots 4 and 5 When fitted in the EAST position a ROM up to 32k may be plugged in socket IC37 The ROM will occupy slots 6 and 7

<u>Link Geography</u>

Link settings are referred to by points of the compass With the keyboard facing you, south is nearest, north is to the rear, west is to the left and east is to the right Most links consist of three pins and a shorting link of two pins is placed across the central pin to one on either side If a link is made WEST then the shorting link is placed on the west or leftmost of the three pins Similarly if a link is made east, the shorting pins are placed across the rightmost of the three links Figure D2 shows this

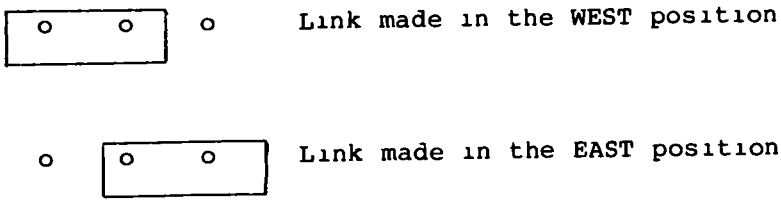

Link made in the WEST position
Link made in the EAST position

Figure D2 Link settings

Note also that all the chips on the circuit board are placed with the half-moon at one end facing north

Appendix E
Postscript

Since writing and preparing this text the following items have come to light prior to going to press

Interactive Help
This service call is primarily intended for use with networks On receiving the call, the ANFS will look at the fileserver for a file called 'HELP and run this With the ANFS installed the service call may not get passed to sideways ROMs of a lower priority

Language ROMs
If you try to boot a language ROM that does not conform strictly to the protocols defined in this book then the MOS will respond with the error message

'This is not a Language'

and refuse to boot the ROM The most common culprit is the use of a lower-case 'c' rather that the required upper-case 'C' to form the copyright string

OSWORD Calls
Two OSWORD calls are provided to allow emulation of *SRREAD, *SRWRITE, SRLOAD and SRSAVE from machine code programs - details are as follows

```
    OSWORD 66 (&42)  Block transfer to/from SRAM
                     (performs *SRREAD and *SRWRITE)
    Parameter block
               XY+Ø  bit7 - Ø to read SRAM
                     bit7 - I to write to SRAM
                     bit6 - Ø for absolute addressing
                     bit6 - 1 for psuedo addressing
                     bits Ø-5 - all at Ø
               XY+1  <LSB of start address>
               XY+2
               XY+3
               XY+4  <MSB of start address>
               XY+5  <LSB of block length>
               XY+6  <MSB of block length>
               XY+7  <ROM id>
               XY+8  <LSB of sideways address>
               XY+9  <MSB of sideways address>
```

On exit, the parameter block remains unchanged Note that ROM ids W,X,Y and Z for the BBC B+128 are denoted by the values &1Ø, &11, &12, &13 respectively

```
    NB  LSB = least significant byte
        MSB = most significant byte

    OSWORD 67(&43)  Block save to/from SRAM
                    (performs *SRLOAD and *SRSAVE)
    Parameter bock
               XY+Ø  bit7 - Ø to save from SRAM
                     bit7 - 1 to load into SRAM
                     bit6 - Ø for absolute addressing
                     bit6 - 1 for pseudo addressing
               XY+1  <LSB of file name address>
               XY+2  <MSB of file name address>
               XY+3  <ROM id>
               XY+4  <LSB of start address>
               XY+5  <MSB of start address>
               XY+6  <LSB of file length> - save only
               XY+7  <MSB of file length>
               XY+8  <LSB of buffer start address>
               XY+9  <MSB of buffer start address>
              XY+1Ø  <LSB of buffer length>
              XY+11  <MSB of buffer length>
```

On exit, the parameter block remains unaltered unless the buffer addresses cause it to be overwritten during file transfer The buffer relates to the area of memory used to save file blocks during the transfer to or from the filing system If the bytes at XY+1Ø and XY+11 are set to zero then the default buffer is used, using any start address specified in XY+8 and XY+9 - this is the equivalent operation of *SRLOAD or *SRSAVE without specifying a Q parameter, ie a slow transfer is performed If the value in XY+1Ø and XY+11 is a value between 1 and 32768 then the specified number of bytes are used for the buffer starting at the buffer start address given in XY+8 and XY+9 If the value in XY+1Ø and XY+11 is greater than 32768 then a buffer that runs from OSHWM to just below the screen is used for the transfer This is the equivalent of specifying a Q parameter

As with OSWORD 66 ROM ids W, X, Y and Z are represented by &1Ø, &11, &12 and &13 respectively

OSBYTE Calls

Two OSBYTE calls are implemented for use with SRAM, these are calls 68 and 69

OSBYTE 68(&44) Test RAM presence

This call simply allows you to test if each of the four SRAM banks are present, ie if they can be used SRAM cannot be used if PCB links are altered (Appendix D)

Entry parameters none

Exit parameters the X register returns a value in the least significant four bits to indicate which banks are present If the bit is set the bank is present, if clear it is absent The corresponding bits are

bit	bank
Ø	4
1	5
2	6
3	7

OSBYTE 69(&45) Test use of SRAM bank

This call allows use of each of the four SRAM banks, ie if they are being used in pseudo or absolute mode

Entry parameters None

Exit parameters the X register returns a value in the least significant four bits to indicate the operation mode If the bit is set pseudo addressing is being used, if clear, absolute addressing The corresponding bits are

bit	bank
Ø	4
1	5
2	6
3	7

*INSERT and *UNPLUG

To prevent clashes of ROM commands it is possible to 'remove' ROMs under software control - this is done with *UNPLUG The command should be followed by the ROM id, ie *UNPLUG 7 *INSERT will 'plug' the ROM back in - *INSERT 7 A CTRL-BREAK will complete the process

Index

The following books and articles are recommended reading

Title	The Advanced Disc User Guide
Author	Colin Pharo
Publisher	Cambridge Microcomputer Centre
Price	£14 95
Comments	Good detailed description of the Acorn DFS Contents apply to 8271 disc controller chip

'Chatting with a chip' by David Atherton Details differences between 8271 and 1770 disc controllers Published in Acorn User July 1986 Pages 143, 144,145 Back issues available from Redwood Publishing, 141-143 Drury Lane, London, WC2B 5TF Describes how the 1770 works and in particular its 8271 emulation

Title	Mastering Practical Interpreters and Compilers
Author	Bruce Smith
Publisher	BBC Publications
Price	£14 95 (published April 1987)
Comments	A book describing the writing of languages and compilers Practical examples are given throughout and include a graphics language (Grafrite) and a compiler that will compile to stand-alone machine code

Title	BASIC ROM User Guide
Author	Mark Plumbley
Publisher	Adder Publishing
Price	£9 95
Comments	A good description of how BASIC works Contents are limited to BASIC 1 and 2, but are applicable to later versions of BASIC though the routine addresses will have changed

Title	Advanced User Guide
Authors	Bray, Dickens and Holmes
Publisher	Cambridge Microcomputer Centre
Price	£14 95
Comments	Limited to BBC B but still a useful guide

Title	Mastering Assembly Language
Author	Richard Vialls
Publisher	BBC Publications
Price	£8 95

Addendum

Chapter 4 All ROMs should respond to the command *HELP and provide full extended help details and lists

Chapter 7 Location &D9F is used (page 83) to gain the start address for extended vectors The 'legal' way to do this is to use OSBYTE &A8 However, since Acorn uses &D9F in ROMs I feel it is safe to use the 'illegal' method'

Chapter 9 See note on page 111

Chapter 10 It is not necessary to press the CTRL key when auto-booting ROMs, with a key-BREAK combination Pressing key-BREAK is sufficient Page 121 and 122 *DISC could be selected more elegantly using OSBYTE &8F, Y=&12, X=4 Similarly *CAT via the OSFSC vector with A=5 (page 97) *ROM can be done with OSBYTE &8D

Chapter 11 The use of memory from &100 upwards (ie the error message buffer) to act as a temporary store for the contents of memory &70 to &8F by listing 11 1 has been described as 'untidy', and that it would be better to push the contents onto the top of the stack directly This is not necessarily so - pushing directly onto the stack creates problems in that the command *PUSH could not be used from within a subroutine as the top of stack contents will have changed Error messages when issued by service ROMs will overwrite the pushed data, but as the program will exit this is of no importance Of course the effect of a *PUSH would be to render REPORT usless However, I would remind readers of the philosophy of this book (page 10)

Listing 11 1 uses memory locations &38 and &39 as a vector This is fine when BASIC is the main language resident or if you are writing your own language ROM However, it should be avoided in service ROMs The user locations &70 to &8F inclusive are an alternative Use of &F2 and &F3 is acceptable or better still the 'official' workspace locations &A8 to &AF

Appendix B The BRA instruction can be replaced with JMP to run on a BBC B or Electron

General

Some confusion has occurred over the clearing of the accumulator after a service call is trapped In general the accumulator should only be cleared with zero (ie LDA £0) if the service call is not to be passed onto another ROM (ie if a command is identified on service call 4) On the other hand, it should not be cleared on service calls 1,2,9,15,16,34,35,36,37,38,39 and 42

Discs

Long listings mean tired eyes and fingers So avoid the strain and the pain and treat yourself to a copy of the programs listing disc The Master 128 and Compact discs contain several extra listings showing the new OSBYTE and OSWORD calls in action All discs (including the BBC version) include a ROM image combining many of listings into a single ROM image The following versions are available

Master	5 25in	DFS	£7 95	(inclusive)
BBC B/B+	5 25in	DFS	£7 95	
Electron	5 25in	DFS	£7 95	
Electron	3 5in	ADFS	£9 95	
Compact	3 5in	ADFS	£9 95	

For overseas orders, please add £1 to these prices

The 5 25in discs can be loaded on 40 and 80-track drives

Please state which version you require and send your cheque or postal order, payable to Victory Publishing, to Victory Publishing, PO Box 19, London N11 1DS

Note that all mail orders are processed using a database which is kept solely for the use of Victory Publishing If you do not wish to be included in this, please inform us and we will delete your name

---

Please send me the listing disc for the Advanced Sideways RAM Guide I require the version for

a micro on -in disc in format

I enclose a cheque for £ made payable to Victory Publishing

Name
Address

SIDEWAYS RAM is a technique central to the philosophy of Acorn's new MASTER series of computers. It allows you to load in software from disc or cassette which is designed to behave as if it were part of the machine, and is always on tap to the user. This book allows you to exploit SIDEWAYS RAM to the full. Written by BRUCE SMITH, the foremost author on this subject, the book is backed up with tried and tested software for you to use straight away or adapt to your needs (and for readers with less nimble fingers it's available on disc).

The secrets of Acorn's ROM Filing System (RFS) are laid open to you and explained in full with numerous examples – there's even a ROM Formatter, developed by the author for use on commercial software, ready for you to type in and use.

The many features of this book include:

- 25 tried and tested programs and routines demonstrating the use of SIDEWAYS RAM.
- Compatibility tables and conversion notes for the BBC and Electron micros.
- Checking routines built into all major programs.
- ROM Formatter to convert BASIC programs on disc to be treated as ROM software.
- Writing languages and interpreters.
- Details of all the SIDEWAYS RAM utilities built into the MASTER.
- Service calls explained and illustrated.
- Routines, programs and text extensively indexed.

BRUCE SMITH is the Technical Editor of the best-selling magazine *Acorn User* and has already written 11 books on the BBC Microcomputer, including *The BBC Micro ROM Book*, and *Mastering Practical Interpreters and Compilers.*

£9.95

ISBN 0 948938 00 5

VICTORY PUBLISHING