Chapter Three
The Machine Operating System

The BBC Micro has so many unique features that it is difficult to pick out any one for special praise. Also it is easy to overlook its broader design philosophy because individual features capture the attention. The MOS (Machine Operating System) is a machine code program roughly 16K bytes in size. A program of this size rivals the BASIC in its complexity. However, unlike the BASIC ROM, it is difficult to sum up what the MOS actually does. It is responsible for so many different things that it would be easy to dismiss the MOS as simply a collection of all the 'odds and ends' that wouldn't fit into the BASIC ROM. However, this would be an underestimation of the careful thought that obviously went into writing the MOS. There are two approaches to building a machine. You can design the hardware and then implement a version of BASIC by interfacing it with the hardware directly. This can be thought of as the 'solve the problems as they arise' approach. For example, you would write things like printer drivers only when they were required by a BASIC statement that listed a program to a printer. Writing a version of BASIC with this sort of approach tends to take short cuts to providing access to hardware features which results in a shorter BASIC interpreter. However, it also tends to transmit any difficulties and shortcomings in the hardware back to the programmer. The second approach to building a machine starts in the same way with the design of the hardware, but before implementing BASIC an extra layer of software is *installed* to iron out any problems and generally improve the hardware's appearance. In the BBC Micro this extra layer of software is provided in the MOS. In some senses it is more accurate to think of the BBC Micro, its hardware and the MOS providing an environment that is suitable for running BASIC or any other high level language. Another way of looking at this is that the MOS creates a 'soft

machine' that is easier to use and more sophisticated than the underlying 'hardmachine' So although the MOS has to do a very wide range of things, it has a single purpose.

The rest of this chapter looks at some of the interesting things that you can do with the MOS. Even though the MOS has a single purpose it is impossible to find a logical order in which to discuss it because of the wide range of things that it does to achieve this purpose. There has already been a number of versions of the MOS issued since the BBC Micro was first produced. The version described in this book is Version I. This is the first version of the MOS to include all of the intended features. To find out which version you have simply type *FX 0. If you have an earlier version and find that you lack facilities that you wish to use, then contact your dealer for a new ROM.

In general, it is true to say that the MOS provides software to handle the following I/O devices:

- The graphics display and VDU drivers.
- Printer and serial I/O.
- Cassette filing system.
- Keyboard.
- A to D convertor.
- Sound generator.
- The tube.

It also makes extensive use of interrupts to improve the overall performance of the machine.

There are three ways that the MOS is used by the programmer. First, many BASIC commands are implemented directly by the MOS. Secondly, there is a range of MOS commands such as *FX and *KEY which cause the MOS to carry out certain tasks. Finally, there are machine code routines within the MOS that the assembly language programmer can use. It is difficult to avoid considering the use of the MOS from assembly language programs here even though assembly language is not discussed until Chapter Seven. The MOS is useful to a BASIC programmer but it is fascinating from the point of view of assembly language! Any sections that make reference to assembly language should be read without worrying too much about understanding the material completely. You will only find this chapter completely comprehensible after you have made the acquaintance of assembly language in Chapters Seven and Eight. If you have no desire to learn assembly language then don't worry - there is still much to be gained by using the MOS from BASIC.

Indirection and MOS subroutines

One important feature of the MOS is that all its important subroutines are available for use by the assembly language programmer. In addition, the assembly language programmer can actually replace any of the important subroutines by user-defined routines. The way that this works is particularly simple. All external MOS subroutines are used by a CALL to the region &FF00 to &FFFF. For example, the ' print a character on the screen' subroutine, OSWRCH, is positioned at &FFEE. However, at this high memory location there is very little of the code for each of the subroutines. In fact, all that happens is a jump to the true location of the subroutine inside the main part of the MOS ROM. The address of the true location of the subroutine is obtained from RAM in the region &200-&2FF which is known as the indirection area. For example, the true address of the OSWRCH subroutine is contained in &20F. You can find a table of MOS subroutines, their fixed addresses and their indirection routines on page 452 of the User Guide. The advantage of this roundabout method of getting to the MOS subroutines is two-fold. First, the true locations of the MOS subroutines can be changed in later versions without invalidating user programs. Secondly, by changing the address stored in the indirection area of RAM the user can intercept MOS calls and supply alternative versions.

The MOS subroutines that are available to the user fall into three categories - tape I/O routines, screen and keyboard I/O routines and three miscellaneous routines. The tape 1/O routines OSFIND, OSGBPB, OSBPUT, 0SARGS and OS FILE are used by BASIC to manipulate cassette files and may be used by the assembly language programmer for the same purpose. They are all adequately described in the User Guide and it is unlikely that a programmer would ever want to replace them with special versions.

The keyboard and screen I/O routines are OSRDCH, OSASCI, OSNEWL and OSWRCH. These form the basic way of handling text from BASIC and assembly language programs. Once again, they are well described in the User Guide and no further comment is necessary.

The three miscellaneous subroutines are quite another matter, however! Between them they carry out so many different functions that it is worth highlighting some of the possible ways that they could be

used. The three subroutines are OSBYTE, OSWORD and OSCLI. OSBYTE and OS WORD are general purpose subroutines that can be used to configure the BBC Micro or control I/O devices. The OSCLI subroutine is a command line interpreter that allows the BASIC programmer direct access to the OSBYTE subroutine. Any command line that starts with an asterisk, such as *MOTOR I, is not processed by the BASIC interpreter; instead it is passed to the OSCLI subroutine for processing. The OSCLI decodes the command and then calls the OSBYTE subroutine to carry out the correct action. Most of operating system commands are of the form *FX ' parameters but some are used so often that they are given names all of their own, for example, *MOTOR, "TAPE etc. Thus, OS BYTE calls that do not need to return any results are available to the BASIC programmer as operating, system commands. The OSBYTE calls that return results can only be used from BASIC via the USR function. The OSBYTE subroutine deals with everything that can be specified using only three bytes (these are held in the A,X and Y registers). Anything that needs more than three bytes is handled by the OSWORD subroutine. As there is no simple way of passing more than three bytes to an assembly language subroutine, OSWORD calls can really only be used by assembly language programs.

Although the User Guide describes all the OSBYTE and OSWORD calls in some detail it doesn' t always make clear what they might be used for. 1n order to avoid repeating the details given in the User Guide, a complete list of calls will not be given here. Instead, the sort of thing that the less obvious calls might be used for will be briefly described.

The first *FX call that is worthy of further discussion is *FX 4. Following *FX 4,1 the five cursor keys return ASCII codes just like the other keys on the keyboard. The normal condition is for the cursor keys not to return ASCII codes but move the cursor round the screen. This condition can be restored by *FX 4,0. There are two reasons why you might want the cursor keys to return ASCII codes. First, you may simply want to disable the cursor editing facility in an applications program to stop inexperienced users from getting out of their depth. Secondly, if you want to use the four *arrow* keys to control the movement of a graphics character in a game, then the only way that this can be done is for the cursor keys to return ASCII codes.

The *FX 11 call sets the time that a key has to be held down before it starts to auto repeat. The required time delay in centi-seconds

is the only parameter in the call, and if you specify a time of zero then the auto repeat is disabled. *FX 12 sets the rate at which keys auto repeat. Once again there is a single parameter that sets the time between repeats in centi-seconds. These two calls are often used together to change the response of the keyboard. For example, it is a good idea to turn the auto repeat facility off in applications programs. However, in games programs where a quick response is required, the keyboard can be set to auto repeat after only 1 centi-second and produce characters at the same rate. By using *FX 11 and *FX 12, the BBC Micro' s keyboard' s response can be adjusted to suit any situation.

One *FX call that seems particularly puzzling is *FX 138 which inserts a character into the keyboard buffer. The format of the call is *FX 138,0 ' ASCIIcode' which inserts the character whose code is ' ASCIIcode' into the keyboard buffer. Any characters placed in the keyboard buffer in this way are treated in exactly the same way as if they had been typed in. The BBC Micro doesn' tcare where the characters in the keyboard buffer come from, only what they are. This means that a running program can place a string of characters in the keyboard buffer and when the program ends the string will be obeyed as if it had been typed in. For example, if the string ' LISTfollowed by a carriage return is placed in the keyboard buffer by a running program, then the program effectively lists itself as soon as it stops! To see this in action try:

```
10 *FX 138,0,76
20 *FX 138,0,73
30 *FX 138,0,83
40 *FX 138,0,84
50 *FX 138,0,13
```

This ability for a running program to ' typeon the keyboard' is something to be kept in mind when all else fails. It can be used to good advantage to provide default answers to questions asked by an applications program. For example, if you remove line 50 from the above program, the keyboard buffer is filled with LIST but without the carriage return. If you also add 60 INPUT A$ to the end of the program you will see that the word LIST appears after the usual' ?'prompt printed by the INPUT statement. To accept it, all you have to do is press RETURN; to reject it you backspace and type whatever you want.

The final "FX code that deserves special mention is *FX 229. The call *FX 229,1 disables the action of the ESCAPE key and makes it

return the ASCII code 27. In other words, following *FX 229,1 the ESCAPE key no longer interrupts the running of a BASIC program. To restore its normal action use *FX 229,0. Using this call and the definition of *KEY 10 as OLD|M RUN|M makes a BASIC program completely unstoppable. The call disables the ESCAPE key and the definition of key 10 (the BREAK key) effectively disables the BREAK key. The only way to stop the program is to switch the machine off.

Adding commands

The OSCLI subroutine provides a simple method of adding new commands to the MOS or to BASIC. Any line that starts with an asterisk, be it a direct command or in a BASIC program, is handled by the OSCLI subroutine at &FFF7. As mentioned earlier, all the MOS subroutines indirect through RAM locations and OSCLI is no exception. The address of the actual OSCLI subroutine is stored in &0208 and L0209. To add new commands, we could intercept the OSCLI call by changing the address stored in &0208 to point to a specially written assembly language routine. Although assembly language isn't discussed until Chapter Seven, it is worth including an example here.

Consider the following short program:

```
 10 DIM CODE% 10
 20 ?&208=CODE% AND &00FF
 30 ?&209=(CODE% AND &FF00)/&FF
 40 P%=CODE%
 50 [ LDA #65
 60   JSR &FFEE
 70   RTS
 80 ]
 90 FOR I=1 TO 10
100 *
110 NEXT
```

Lines 50 to 80 form an assembly language program that simply prints the letter A on the screen. Lines 20 and 30 change the address of OSCLI to the address of the 'print letter A' routine. Lines 90 to I 10 form a perfectly simple FOR loop apart from the fact that line 100 is nothing but an asterisk! If you run the program you will find that the letter A is printed on the screen ten times, thus proving that the asterisk now means 'print the letter A'.

In any real application, the address of the OSCLI subroutine

would be saved within the new assembly language routine. The new routine would check to see that what followed the asterisk was a command that was its concern. For example, we might decide to call the ' printA' routine by the command *PRINTA and the first job that the routine would do would be to check that the word ' PRINTA' followed the asterisk. If this was not the case then the OSCLI proper would be called using the address that was originally stored in the RAM locations. In this way new commands can be added to the existing set of commands rather than replacing them.

The video display

The hardware and software that makes up the video display is discussed fully in the next chapter. However, it is worth pointing out that the MOS is entirely responsible for the software that drives the video hardware. In particular, the MOS contains the VDU drivers and the character generator table. The method of communication with the video section of the MOS is not via a long list of subroutine calls. Instead, the OSWRCH subroutine detects and acts upon an extended set of ASCII control codes. These control codes are sent to the OSWRCH subroutine in exactly the same way as a printable character, but their effects can be very extensive. In BASIC the command VDU appears to be the fundamental graphics command. In fact, all it does is to transmit the necessary control codes to the VDU drivers via the OSWRCH subroutine. So the following are equivalent:

```
VDU 8
PRINT CHR$(08);

[ LDA #8
  JSR &FFEE
]
```

and each sends a backspace command to the VDU drivers. In practice, many of the control codes are followed by a number of parameters.

Interrupts

The BBC Micro makes extensive use of interrupts to improve its overall performance. An interrupt is simply a wav of switching the

' attentionof the 6502 processor inside the machine from one task to another and back again. For example, if a BASIC program is running, then the 6502 is giving its full attention to this task. If, however, a key is pressed on the keyboard this causes an interrupt which makes the 6502 stop what it is doing and start running the keyboard service routine in the MOS. This finds out which key was pressed and stores the correct ASCII value in the keyboard buffer. Once the keyboard is dealt with, the 6502 returns to the original task of running your BASIC program, starting from the point where it was interrupted. This idea is not a difficult one - after all, humans respond to interrupts. If you are reading a book and the telephone rings then you process this interrupt by marking your place in the book, answering the telephone and then returning to your reading at the point where you were interrupted. However, even though the idea of an interrupt is simple in theory, in practice things are often difficult to handle. The trouble is that an interrupt may happen at any time and may be caused by any number of devices. For example, as well as the keyboard interrupt the 6502 has to service an interrupt from a timer in VIA - A every one hundredth of a second. On receiving this timer interrupt, all the 6502 does is to increment the value stored in the variable TIME but how does it know where the interrupt came from? Was it from the keyboard or was it from the timer? In fact, there are many sources of interrupts that we haven' tyet considered. The key to finding out which device has caused an interrupt is contained in the hardware causing the interrupt. Each I/O device that can cause an interrupt has an a bit known as an interrupt flag somewhere in its status register. This flag is normally set to zero but if the device has caused an interrupt then it is set to one. The method of finding which device has caused an interrupt is simply to examine each of the I/O devices' interrupt flags to find which are set to one.

As already mentioned, the BBC Micro makes extensive use of interrupts. However, if you do not intend to become involved with writing assembly language programs that make use of interrupts then you can ignore this fact. The only unexpected consequence it has is that you cannot use delay loops for exact timing simply because you cannot always guarantee that the 6502 is executing your program - it might be off servicing an interrupt for some part of the time! Apart from this, interrupts simply alter the general way that the BBC Micro behaves. For example, the fact that the keyboard is serviced by an interrupt as described in the last paragraph means that anything that

you type on the keyboard goes into the keyboard buffer even if the computer appears to be busy doing something else - i.e. it provides type-ahead. The overall effect of interrupts on the BBC Micro is to give it the appearance of being able to do more than one thing at a time!

If you are interested in making use of interrupts in assembly language programs, then you will certainly need to know a little more than outlined above. The 6502 recognises three distinct types of interrupt - NMI or Non-Maskable Interrupts, IRQ or Interrupt ReQuest and BRK or Break. The first type, NMI, is strictly reserved for use by the disc operating system and need not concern us further. All the other I/O devices that can cause interrupts use the IRQ interrupt. The BRK interrupt is a little different in that it is a software interrupt. A software interrupt is exactly the same as a normal interrupt except for the fact that it originates internally rather than being caused by an external device. In fact, 6502 assembly language includes the mnemonic BRK which causes a BRK interrupt to occur.

When the machine detects an IRQ interrupt it immediately passes control to a routine whose address is stored in &0204 (IRQV1). In other words, it indirects through this location in the same way as the MOS subroutines indirect through their own particular locations. The standard MOS routine to handle interrupts looks at the interrupt flags of all the devices that it knows about to discover the source of the interrupt. If it finds it then the appropriate action is carried out. For example, if it finds that the timer is responsible for the interrupt it will increment TIME and then return control to the program that was interrupted. However, it is possible that some I/O device that the MOS doesn't know anything about has caused the interrupt. In this case the standard interrupt service routine will not locate the cause of the interrupt. When this happens, control is passed to the routine whose address is stored in &0206 (IRQV2). Of course, this routine has to be supplied by the user to handle the interrupt in the appropriate way. When the user interrupt handler has finished it should return control to the MOS interrupt handler (by RTS) which will finish the interrupt procedure and return control to the program that was interrupted. Thus, adding routines is fairly straightforward. If you cannot afford to wait while the MOS checks all its possible sources of interrupts then you could intercept the IRQ interrupt at IRQV1 instead of IRQV2. In this case, of course, you should check, and possibly deal with, your source of interrupts and then pass control to the MOS interrupt service routine

(whose address was originally in IRQV1).

The BRK interrupt is used by BASIC to report errors. How this is done is well-described in the User Guide and so will not be repeated here. As BRK indirects through &0202, it, too can be intercepted and handled by a user-supplied routine as required.

Interrupts that the MOS can handle might still be of interest to the user. For example, it may be that a user program needs to know when any key has been pressed although it is quite happy for the MOS to handle the interrupt. To deal with this requirement the M OS recognises a number of events. An event is either an interrupt or something that is detected during an interrupt that the MOS can handle perfectly well on its own. However, the MOS will inform the user of the event' soccurrence on request. The normal state of affairs is for all events to be disabled. If an enabled event occurs, however, then control is passed to the routine whose address is stored in @0220. The events are enabled using "FX 14, ' code' where ' code' is one of the following:

Code	*Event*	
0	A buffer is empty	X=buffer identity
1	A buffer is full	X=buffer identity Y=character that couldn' tbe stored
2	A key has been pressed	
3	ADC conversion complete	
4	Start of TV field pulse	
5	Interval timer crossing zero	
6	Escape condition detected	

To disable an event, the same codes should be used in *FX 13,code. Notice that any normal interrupt handling happens before an enabled event indirects through A0220.

As a demonstration of how events work, type in and run the following program:

```
10 DIM CODE% 20
20 ?&220=CODE% AND &00FF
30 ?&221=(CODE% AND &FF00)/&FF
40 P%=CODE%
50 [ LDA #65
60   JSR &FFEE
70   RTS
80 ]
```

You might recognise the ' printan A' assembly language subroutine, that has been used in earlier examples, in lines 50 to 80. However, in this case its address is placed in &0220 and k0221. This means that following this program, any enabled event will cause a letter A to be printed on the screen every time it happens. To see this in action simply use *FX 14,code to enable the event of your choice. For example, following *FX 14,2 you will see a letter A printed following every character you type on the keyboard! Following "FX 14,4 you will see the letter A appear on the screen almost continuously - a field pulse event occurs every fiftieth of a second! To recover from most of the above examples it is easier to press BREAK and then type OLD and run the program again rather than try to disable the event with *FX 13,code - the A' sappearing on the screen make typing difficult! Although this is not a very useful example, it does show how events can be used. A typical real example would be to synchronise the running of a program to the start of a TV frame display. Dealing with characters typed on the keyboard as soon as they are typed is another practical example.

Conclusion

The MOS is a very complex piece of software. Many of its functions are concerned with important I/O devices such as the graphics display, the sound generator and the A to D convertor, and these are discussed in the next three chapters. In this chapter, some of the less obvious features of the MOS have been described so that the BASIC programmer and assembly language programmer can both begin to make good use of the range of facilities available. It should also have made apparent just how clever the BBC Micro' s MOS is.