CHAPTER FIVE

A FEW WAYS TO PROTECT YOUR PROGRAMS

Piracy is a problem that is worrying many software houses these days. It is impossible to produce a commercial program that cannot be copied − however clever the protection is, someone will find a way around it. However, it is possible to make it very difficult for anyone to copy a program. Very few people have the skill and patience to copy a program which has been properly protected. Here you will find a few of the many techniques that can be used to make copying difficult.

It is possible to write a program in such a way that, once it is run, it cannot be stopped. To do this we need to disable the ESCAPE key and the BREAK key. The ESCAPE key is easy to disable but the BREAK key cannot be disabled completely. It is, however, possible to make the computer clear the memory when the BREAK key is pressed. The Operating System conveniently provides a *FX call to deal with both the ESCAPE and BREAK keys. This is *FX200. This is used to change a flag byte within which only bits 0 and 1 do anything. If bit 0 is set then the ESCAPE key is completely disabled, and if bit 1 is set the BREAK key causes the memory to be cleared. Thus, by placing the command *FX200,3 at the beginning of a program, it is impossible to get out of the program without the memory being cleared. (We will see in chapter 6 how to intercept the BREAK key.)

Locked tape files

This is all very well but a program can still be loaded and then saved. We need a way of stopping people doing this. Acorn have kindly placed a protection system in the BBC Model B Operating System for tape users. This system produces a file on tape which is 'locked'. If you try to *LOAD such a file you will get the message 'file locked'. In fact, the only command the operating system will allow you to use on a locked file is *RUN. This means that only machine code programs can be locked.

A locked tape file is produced by setting a particular bit in the header of each block. This tells the operating system that the file is locked. The operating system does not, unfortunately, provide a command for saving locked files, so we need a program that will do this. At first it would appear that we need to write a complete save routine to do this. There is, however, a beautifully simple way.

While the operating system is saving a file it keeps a copy of the header for each block in locations &3B2-&3D0. The bit we are interested in is bit 0 of location &3CA. Because the tape hardware is interrupt driven, not only does an interrupt occur every time a byte is saved to tape, but also, the centisecond clock is left running. It is very simple to redirect the main interrupt vector so that as each byte is saved bit 0 of &3CA is set. The following program does just this. Notice that the machine code is stored in ZERO page where it can't get in the way. Once this program is run all files saved will be locked until BREAK is pressed.

```
 10 P%=&50
 20 [OPT3
 30 .irq    PHA
 40         ORA #1
 50         STA &3CA
 60         PLA
 70         JMP (&230)
 80 .init   SEI
 90         LDA &204
110         STA &230
120         LDA &205
130         STA &231
```

```
140         LDA #irq MOD256
150         STA &204
160         LDA #irq DIV256
170         STA &205
180         CLI
190         RTS
200 ]
210 CALLinit
```

We now would appear to have an uncopyable program which can only be *RUN and, once run, cannot be broken into. Of course, it would be naive to think even this system is infallible. It is possible to crack a locked file by substituting line 50 with AND #254 .

Unlistable programs

It is sometimes useful to be able to write a BASIC program which cannot be listed. This is done using ASCII code 21 which disables the VDU drivers. When this has been sent to the VDU drivers the screen ignores everything that is sent to it until a VDU6 command is used to re-enable the VDU drivers again. By placing CHR$21 and CHR$6 codes in a listing, part or all of a program can be made unlistable.

Unfortunately, these code cannot just be typed into a program listing. If, however, two character codes are not used anywhere else in a program are used − codes such as the ‘curly’ brackets − then it is quite easy to write a short program that will convert these into CHR$21 and CHR$6 codes. So that the program will run properly these characters should be placed in REM statements. Other characters can be used, such as delete (CHR$127); or cursor controls, such as line feed. The obvious way to use this is to disable the whole listing, but it is often much more effective to use it to remove particular lines without which the program appears to do something completely different. This system can be used to great effect as the first part of a game which prints instructions and then chains or *RUNs the next section. Try typing in this example program:

```
 10 REM @*******************************[
 20 REM ]*                              *[
 30 REM ]*   This program is copyright  *[
 40 REM ]*                              *[
 50 REM ]*   It is illegal to copy it.  *[
 60 REM ]*                              *[
 70 REM ]*******************************[
 80 REM REST OF PROGRAM
 90 PRINT"THIS PROGRAM WILL STILL RUN"
100 GOTO90
110 REM]
```

We want to change @ (line 10) to a clear screen (code 12); {chr$21 ; and } to CHR$6 (lines 20 to 70 and 110). To do this you must first type the commands:

```
PAGE=&2000
NEW
```

Then you must type in and run the following program. Disc users will need to change the setting of A% at line 10 to &1900

```
10 A%=&E00
20 IFA%?1=&FF END
30 A%=A%+3
40 A%+A%+1:IF?A%=13 THEN20 ELSE IF?A%<>&F4
THEN40
50 REPEATA%=A%+1:IF?A%=ASC"@" ?A%=12
60 IF?A%=ASC"}" ?A%=6
70 IF?A%=ASC"{" ?A%=21
80 UNTIL?A%=13:GOTO20
```

Notice that in line 50 the REPEAT command does not need a : after it. This is never needed after a REPEAT statement.

Now set PAGE back to its usual value and try listing the program. Note that the program still runs normally.

This program first finds the start of each line of the program then looks for the token (&F4) for a REM statement. It then searches the rest of the line for the characters we want to replace. This way the

program won't accidentally alter line numbers or lines of BASIC.

Alternatively, you can use a machine code monitor (such as the one in chapter 4) to look directly at and modify the relevant characters in the BASIC coding.

A very effective (and far more subtle) way of using this technique is to double-bluff the pirate. For instance, say that in the BASIC loader for a macine code game the last two commands are:

```
*LOAD game 3000
CALL &30B2
```

The pirate will look at this and will be able to examine the machine code. If we can arrange for him not to know the load and execution addresses then his task is much harder. Even better, if we can convince him that the load and execution addresses are, say, &2800 and &293A respectively.

The way to do this is to put a dummy set of commands at the end of the program and then conceal the real ones. The original program ending would look like this:

```
320 REM * load machine code *{
322 *LOAD game 3000
324 CALL &30B2
326 REM }
330 *LOAD game 2800
340 CALL &293A
```

Try typing this in and using the alteration program as before, then listing the program.

Note that the line numbers that the pirate will see are in steps of ten, leaving no clue to the missing lines.

One word of warning before you use this technique. There is a major, very annoying bug in the operating system! When the VDU drivers have been disabled with VDU21 any carriage return sent to the VDU drivers are sent to the printer: even if the printer hasn't been enabled − even if your printer has been turned off − even if you don't have a

printer! This irritating quirk means that every time you list your program the printer spits paper at you! Also, if your printer is not turned on, these carriage returns (and line-feeds if you have used *FX6,10 accumulate in the printer buffer. If the number of these reaches 63 the whole computer seizes up until either ESCAPE or BREAK is pressed. So don't fill your programs full of CHR$21s.

If you are writing a program with the VDU21 command in it you can get around this by using the command *FX3,64 before using VDU21. This disables the printer driver completely except for characters sent using the VDU1,x command.

Disc tricks

For those of you with disc drives the list of fiendish tricks you can play on the pirate is endless. These tricks all rely on knowing how to access the blocks on a disc directly, using an OSWORD call. The DFS adds a series of extra calls to the standard list of OSWORD calls. We are going to use one of these − OSWORD &7F. This routine saves or loads a section, or all, of a track.

On entry to this routine the X and Y registers must point to a parameter block (Y high). This parameter block consists of 10 bytes and should be laid out as follows:

XY + 0	Drive number
XY + 1 to XY + 4	Load / Save address.
XY + 5	Number of parameters (3)
XY + 6	Command (&53 for load, &4B for save).
XY + 7	Track number.
XY + 8	Sector number.
XY + 9	High nibble: sector length in 128-byte groups. Low nybble: number of sectors to be loaded/saved.
XY + 10	Error number (0 if no error).

For example, if we wanted to read the contents of the block at track 10 sector 5 we would need to reserve 256 bytes in the memory to store the block.

Then we would need to set up a parameter block. The first byte of this parameter block would be zero and the next four would point to the reserved block of memory (the top two bytes would be set to zero). The number of parameters refers to the number of parameters after the command byte that are sent to the routine (i.e. not including the error byte) and so would be three. The command would be &53. The track number would be 10 and the sector number 5. The last (ninth) byte is more complicated. The size of a block on a standard DFS disc is 256 bytes, so the high nibble of the parameter byte needs to be 2. We only want to load one block, so the low nibble is 1, i.e. the ninth byte is &21.

For protection against piracy there are a number of things we can do with this. The first is to save files with names that include control codes. As with the unlistable programs, we can apparently totally eliminate files from the catalogue and yet still load them. This means that unless the pirate tries every name he can think of (quite a long job!) only a person who knows the filename can load the file. This is not much use if you want to sell the program, but if you write a well-protected loader which then chains the main program then only the loader's filename need appear on the catalogue.

If you have the old DFS then saving control codes in filenames is easy. For example, to save a file that appears on the catalogue as TEST but actually has a different filename, try:

```
SAVE "TESS|HT"
```

The problem with this is that the catalogue and compact routines use the length of the filename to set out the catalogue on the screen. So, if you are not careful, filenames will tend to be shifted left if they are printed on the same line, but further right than, a doctored filename. This is where the subtle approach is not necessarily the best. Very good effects can be obtained by titling the disc with a clear screen, a suitable message, and a CHR$21! You can then add a CHR$6 to the end of the last file on the catalogue. If your disc boots then you don't

even have to let the user catalogue the disc at all!

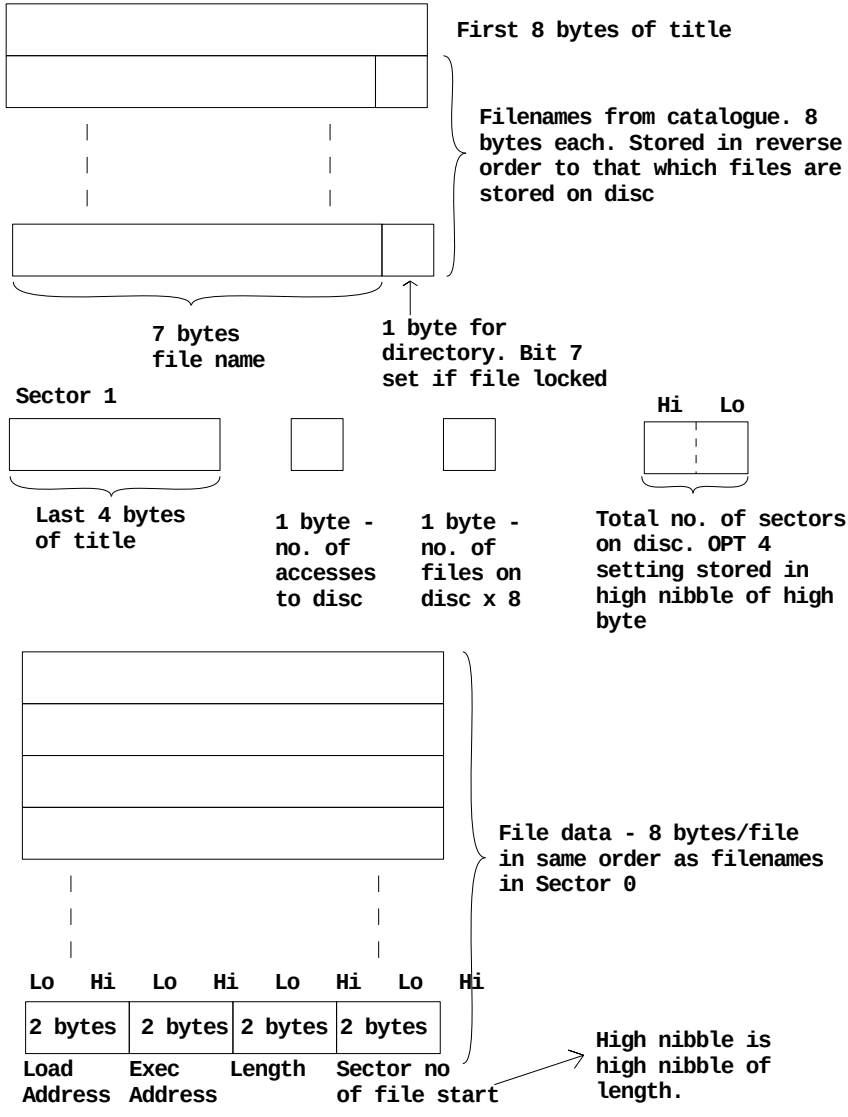
If you don't have the old DFS you will need to load the first block on the disc (track zero, sector zero) into the memory, alter it accordingly, and save it back again − a somewhat more cumbersome method.

However, all this is child's play next to what can be achieved. It is not very useful to stop the pirate loading the program and looking at it if he can just 'backup' to a new disc. Here's how to stop him:

The second block of the disc contains the load and execution addresses, the lengths, and the locations on the disc, of all the files. It also contains eight bytes at the very start of the block. The first four of these are the last four bytes of the title (the first eight bytes of the title are the first eight bytes of block zero). Then comes one byte which stores the number of times the disc has been written to since formatting (this number appears in brackets after the title in the catalogue). Then comes a byte which gives the number of files in the catalogue, times eight. And finally, two bytes that give the total number of blocks on the disc. These are stored high byte first (yes, this is unusual). Also, the high nibble of the high byte is used to store the OPT number.

You may wonder of what use all this is. The answer is that we can set the total number of blocks on the disc to zero! This may seem foolish but it doesn't affect the way the DFS loads from disc. It only has an effect when you start saving on the disc or when you back-up the disc. When backing-up, the computer copies on to the new disc the number of blocks specified in block one of the source disc. This means that, by setting this to zero, the back-up command will copy precisely no blocks. If you want to be even more cruel to the poor, defenceless pirate, you could arrange for just enough blocks to be copied to copy the loader from disc but not the main program. The loader could then check a block at the end of the disc to see if it contains a specific string − which, of course, you will have put there on the original disc; and, if it is not there, then it would print a suitable message, such as 'This is an illegal copy' and crash!

Track O
Sector 0



Note that if the length has been set to zero you can't save any more programs on disc. It also fools certain utilities' commands for looking at the disc directly − pirates occasionally use this for breaking into discs.

A useful side effect is that *COPYing all the files on disc will not copy the string at the end and so the copy won't work. This leads me to another suggestion for disc protection: have a well-protected loader which loads the main program off the disc directly using OSWORD &7F. This way you need not even put the main program on the catalogue. You might even like to store the program on disc in some personal encryption code.

Another interesting point is that, if you add 128 to all the bytes that make up a filename in block zero, that filename will promptly vanish from the catalogue completely! It will, however, still load and run perfectly normally − IF you know the filename!

Yet another thing that you can do if you have a utility such as DISC DOCTOR which allows you to format sections of a disc, is take an unformatted disc and format it leaving some tracks unformatted. If the pirate tries to make a back-up of this he will get a load error! To cover these tracks set up a dummy file in the catalogue that occupies the space.

There are plenty of other ways of confounding the pirate. You could try coding (that is, encrypting) your program and placing a decoding routine at the start. This won't stop a determined pirate as he can deduce the code from the decoding program, but it will slow him down. In fact, you will find it very difficult to stop a determined pirate. You can only protect against the person who casually copies programs. By combining a large number of protection methods you can make the pirate's job difficult enough so that he will think twice before attempting anything. What you must decide is whether all this is worth it.

A companion book in the MASTER GUIDE series, MASTERING THE DISC DRIVE (BBC Publications, 1985) goes into the disc system in great detail.