CHAPTER SIX

THE KEYBOARD

At first sight there is not a lot that can be said about the keyboard except that it is the most important means of input the computer has.

The first thing that must be said is that programs of a 'professional' standard must be fool-proof. Probably the most common problem with commercial software is that particular keys on the keyboard have not been disabled properly. As a general rule, at any point in a program ALL keys except the ones that can be used legally should be disabled. This applies in BASIC and in machine code.

With this in mind, we should take a look at all the keys and the methods of disabling them.

When you need to input data from the keyboard in a program you should check to ensure that it is made up of legal key presses. In BASIC the easiest way to do this is to compare each character of the string with a string containing all the legal characters. This can be done with the command INSTR.

For example, suppose you want the user to input his first name. You want letters, upper or lower case, and nothing else.

The BASIC to check for this would be:

```
1000 CLS
1010 INPUT"Your first name";A$
1020 E%=0
1030 FORA%=1TOLENA$
1040 IFINSTR("ABCDEFGHIJKLMNOPQRSTUVWXYZabcde
     fghijklmnopqrstuvwxyz",MID$(A$,A%,1))=0
     THENE%=1
1050 NEXT
1060 IFE%=1THEN1000
1070 PRINTA$
```

A BASIC input routine This method of solving the problem still allows the user to type the wrong letters in the first place and a name of up to 250 characters, which would be a little silly! Better to write an input function which uses the GET command and checks each character as it is typed. If a character (such as a %, say) is illegally typed it ignores it and waits for another key to be pressed. Also, a maximum number of characters can be imposed. If this is done it is a good idea to set out a row of full stops, over which the user types, to show him how much he is allowed to type.

```
10 CLS:VDU23;1,0;0;0;0;
20 PRINT"Your first name?":A$=FNinput
   (20,"ABCDEFGHIJKLMNOPQRSTUVWXYZabc
   defghijklmnopqrstuvwxyz")
30 PRINTA$
40 END


1000 DEFFNinput(N%,I$):LOCALA$,G$
1010 PRINTSTRING$(N%,".");STRING$(N%,CHR$8);
1020 A$="":VDU23,1,1;0;0;0;
1030 G$=GET$:IFG$=CHR$13VDU13,10,23,1,0;0;0;0;
     :=A$
1040 IFG$<>CHR$127THEN1070
1050 IFLENA$=0VDU7:GOTO1030

1060 VDU8,46,8:A$=LEFT$(A$,LENA$-1):GOTO1030
1070 IFINSTR(I$,G$)=0VDU7:GOTO1030
1080 IFLENA$=N%THENVDU7:GOTO1030
1090 A$=A$+G$:PRINTG$;:GOTO1030
```

Note also that this program only turns the cursor on (line 1020) when an input is expected. This is good practice as it helps to give the user a clue as to when he is expected to type something.

A machine code input routine

A like method can be used from machine code. Here we need a place to put the string. As BASIC is not being used we can use the space taken up by the BASIC string input buffer (appropriately enough).

This is the whole of page seven of the memory.
Let's specify that the routine is entered with the maximum length of the string in X. We also need a string containing all the legal characters. This we can place at the end of the machine code program with the EQUS command. We can then say that the routine must be entered with the length of the string minus one in Y. This way we can allow different amounts of this string to be legal by changing Y. For instance, if you wanted to use the routine twice − the first time allowing the letters A to Z and the second time allowing only the letters A to F − then you could set out the legal string as 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'. The first time you called the routine, Y would be 25 to allow all the string to be legal, the second time, Y would be 5 so as to only allow the first six letters of the string to be legal.

The routine will exit with the string stored from &700 onwards, and followed by a carriage return. Thus the example would be:

```
10 PROCass
20 CLS:VDU23,1,0;0;0;0;
30 PRINT"Your first name?"
40 X%=20:Y%=51:CALLinput
50 PRINT$&700:END
```

Now we must write the routine.

The first job is to store the contents of the X register at &70 and the Y register at &72 temporarily.

```
1000 DEFPROCass
1010 DIMmc%250
1020 FORpass%=0TO2STEP2
1030 P%=mc%
1040 [OPTpass%
1050 .input   STY &72
1060          STX &70
```

The next job is to print the row of dots over which the user will type. The number is already conveniently in the X register for us.

```
1070 .loop1   LDA #48
1080          JSR &FFE3
1090          DEX
1100          BNE loop1
```

Next we have to move back the same number of spaces to allow the user to start typing over the top of the dots. This time the X register will have to be reloaded from &70.

```
1110          LDX &70
1120 .loop2   LDA #8
1130          JSR &FFE3
1140          DEX
1150          BNE loop2
```

While we are at it we need a variable to count how many characters the user has typed in. This we can store in &71; it will initially be zero. As we have just finished a loop in X, the X register will contain zero, so we can store this at &71.

Then we must turn on the cursor with VDU23,1,1;0;0;0;

```
1160          STX  &71
1170          LDA #23
1180          JSR &FFE3
1190          LDA #1
1200          JSR &FFE3
1210          JSR &FFE3
1220          LDX #7
1230 .loop3   LDA #0
1240          JSR &FFE3
1250          DEX
1260          BNE loop3
```

Now we are ready to get a key from the keyboard using OSRDCH. If this routine exits with the carry flag set then the ESCAPE key has been pressed and we need to acknowledge this and exit the routine with a null string stored at &700.

As we do this we will need to put a carriage

return at the end of the string and turn the cursor off. This part of the program we can use as an exit once we have obtained a valid string, so we must give it the label EXIT.

```
1270 .key     JSR &FFE0
1280          BCC noerror
1290          LDA #&7E
1300          JSR &FFF4
1310          LDA #0
1320          STA &71
1330 .exit    LDX &71
1340          LDA #13
1350          STA &700,X
1360          LDA #13
1370          JSR &FFE3
1380          LDA #23
1390          JSR &FFE3
1400          LDA #1
1410          JSR &FFE3
1420          LDX #8
1430 .loop4   LDA #0
1440          JSR &FFE3
1450          DEX
1460          BNE loop4
1470          RTS
```

Next we must check to see if the key pressed is the RETURN key. If so, then we can branch to EXIT.

```
1480 .noerror CMP #13
1490          BEQ exit
```

Next we check for DELETE. If this has been pressed then we check whether there is any string to be deleted. If &71 is zero then there is no string so we output a ‘beep’ before jumping back to KEY.

```
1500          CMP #127
1510          BNE notdel
1520          LDA &71
1530          BNE del
1540 .error   LDA #7
1550          JSR &FFE3
```

```
1560          JMP key
```

If there is something to delete then we need to go back a space, print a dot and back-space again to leave the cursor over the dot. We also need to decrement the length of the string stored at &71.

```
1570 .del     LDA #8
1580          JSR &FFE3
1590          LDA #48
1600          JSR &FFE3
1610          LDA #8
1620          JSR &FFE3
1630          DEC &71
1640          JMP key
```

If DELETE is not pressed then we need to check for a legal key. The string of legal characters starts at LEGSTR and the length of this string minus one is stored at &72. We need to compare all the characters in this string with the accumulator. We can do this with a loop in X, carefully preserving A throughout the loop. If the contents of the accumulator match with one character of the string then the key is legal; otherwise, if we get to the end of the string without finding a match then we must cause a 'beep' and go back for another key.

```
1650 .notdel  LDX &72
1660 .loop5   CMP legstr,X
1670          BEQ legal
1680          DEX
1690          BPL loop5
1700          JMP error
```

If the key is legal then we must check that there is still a space left to place this key. If the length of the string has already reached the maximum allowed length then we must branch to ERROR. If not then we can store the character at the relevant place in page 7, increment the length of the string, print the character on the screen and go back for the next key.

```
1710 .legal   LDX &71
1720          CPX &70
1730          BEQ error
1740          STA &700,X
1750          INC &71
1760          JSR &FFE3
1770          JMP key
1780 .legstr  EQUS "ABCDEFGHIJKLMNOPQRS
                    TUVWXYZabcdefghijkl
                    mnopqrstuvwxyz"
1790 ]:NEXT
1800 ENDPROC
```

We have now dealt with all the standard ASCII keys. The next thing to look at is the cursor keys. If a GET, INPUT, or INKEY (with a positive parameter) command is used during a program then the cursor keys become enabled. Pressing them will cause the copy cursor to move around the screen. This also leaves a block cursor on the screen. In most cases this is not desirable. There is any easy way to overcome this problem but not many people seem to even realise that there is a problem. For example, many games leave the cursor keys enabled − pressing them during the game will eave 'flying blobs' on the screen.

The way around the problem is to set the cursor keys to generate ASCII codes with the command *FX4,1. This way they can be used as ordinary keys. This command also causes the COPY keys to generate an ASCII code.

The next keys we need to look at are the SHIFT LOCK and CAPS LOCK keys. These keys cannot be disabled easily but the state of the keyboard (and the LEDs) can be changed from the software. This is done using a *FX202 call. This command needs one number after it. If bit 4 of this number is zero then the CAPS LOCK is engaged. If bit 5 is zero then the SHIFT LOCK is engaged. If bit 7 is set then the shift key's action is reversed.

For example, if *FX202,160 were used this would set the keyboard so that it would normally produce capitals and numbers, etcetera, but with the shift key pressed it would produce lower case and the

exclamation mark, and so on.

This command also changes the state of the keyboard LEDs. An example of where it could be used is in a word processor to turn the CAPS LOCK and SHIFT LOCK off at the beginning of the program.

The next key we must look at is the ESCAPE key. In machine code this key has little effect until OSRDCH is called. In this case the escape condition must be acknowledged using OSBYTE &7E call. If this is done then the ESCAPE key is effectively disabled. However, if a more complete form of disablement is needed, say for a BASIC program, then *FX229,1 should be used. This simply causes the ESCAPE key to generate ASCII code 27. Another useful trick is that any key on the keyboard can be made the ESCAPE key. This is done by using the command *FX220 followed by the ASCII code for the key. For instance, if you wanted <CTRL @> to be the ESCAPE key then you would use the command *FX220,0.

One final method is to use *FX200. This has two functions. If bit 0 of the byte following it is 1 then ESCAPE completely disabled (it won't even generate an ASCII code) and if bit 1 is set then the entire contents of the memory will be cleared the next time the BREAK key is pressed! Even <CTRL BREAK> cannot get around this command. This is very useful for protecting programs as it means that once a program has been run it is impossible to get out of it again without losing the program (see chapter 5).

The BREAK key

The break key is probably the most difficult key to disable. In fact, it is impossible to disable it. However, it can be intercepted. Many programs define the BREAK key like a soft key to produce a string that runs the program. This means that if you accidentally press BREAK in the middle of typing in a letter on your word processor, you won't lose your text. However, this doesn't solve the problem of <CTRL-BREAK>. It would be nice if we could stop the computer every time the BREAK key is pressed and check whether a <CTRL-BREAK> has occurred.

If so, we could then convince the computer that it is imagining things and that the BREAK was really a normal one! Well, here's how to do it.

When a BREAK occurs, the operating system looks at location &287. If &287 contains zero then it carries on as usual, and, if the machine has just been turned on then it must be a zero! However, if this byte contains a machine code JUMP instruction (&4C) instead, then it will jump to the address pointed to by the contents of location &288 and &289. In fact, it does this twice after a BREAK occurs. The first time the carry flag is clear, and this occurs before the message 'BBC Computer' appears; the second time is after this message appears but this time with the carry flag set. Locations &287 to &289 can be set using *FX247 to *FX249.

Those of you with fevered imaginations will already have seen some of the possibilities this opens up. For example, it is possible to change the 'BBC Computer' message, which is normally printed, to something completely different. For those of you who like the idea, here is the program:

```
 10 FORA%=0TO3STEP3
 20 P%=&900
 30 [OPTA%
 40.break    BCC exit
 50          LDX #0
 60.loop     LDA string,X
 70          JSR &FFE3
 80          INX
 90          CPX #24
100          BNE loop
110.exit     RTS
120.string   EQUS CHR$12+"Beware of the
             hacker!"+CHR$13+CHR$13
130]:NEXT
140 *FX247,76
150 *FX248,0
160 *FX249,9
```

Notice that this is put in the cassette output buffer (&900). This is totally safe for disc users but cassette

users will find that, if they save a program, the next time they press BREAK the computer will crash.

However, the object of all this was to disable <CTRL BREAK>. When a break occurs one of the first things the computer does is to check whether it is a soft reset, a hard reset or a power-on reset. When it has done this it sets up a variable at &28D. This has the value zero for a soft reset, one for a power-on reset and two for a hard reset. By changing the contents of &28D to zero in our break intercept routine we will fool the computer into believing that a soft reset has occurred. Unfortunately, before we get to do this the computer has already reset the clock and cleared the function keys. However, if we set &28D to zero in the FIRST intercept we can redefine the break key to do whatever we want in the second intercept. Here is an example program:

```
 10 FORA%=0TO3STEP3
 20 P%=&900
 30 [OPTA%
 40.break    BCS second
 50          LDA #0
 60          STA &28D
 70          RTS
 80.second   LDX #string MOD256
 90          LDY #string DIV256
100          JSR &FFF7
110          RTS
120.string   EQUS"*KEY10OLD|MRUN|M"+CHR$13
130 ]:NEXT
140 *FX247,76
150 *FX248,0
160 *FX249,9
```

This clears &28D during the first intercept and uses OSCLI to perform a *KEY10 command during the second intercept.

Once this is run neither BREAK nor <CTRL BREAK> can stop the computer from ‘olding’ and running the current program!