CHAPTER FOUR

INTERRUPTS

With a computer as sophisticated as the BBC Micro, which supports a large number of software-driven peripherals, there are a number of 'housekeeping' tasks the computer must perform regularly to keep these peripherals ready for the user. The processor is not, unfortunately, able to do two jobs at once so it must regularly stop what it's doing to check up on the peripherals. However, there is no point in the processor doing this unless a peripheral actually needs servicing.

To get around this problem the computer uses a system called INTERRUPTS. What happens is that there is a wire, connected to the processor, that is normally at logical level one (5 Volts). This wire is also connected to ALL the peripherals that may need servicing. When, say, the cassette system needs attention it pulls this wire down to logic level zero (0 volts). This interrupts the processor in what it's doing. The processor then finishes the machine code command it was processing at the time the interrupt occurred and then pushes the contents of the program counter (high then low) on the stack, and then the status register. It then looks at two bytes at the end of memory (&FFFE and &FFFF). These two bytes (low then high) make up the address of the operating system routine which handles interrupts.

There are two types of interrupts on the 6502 processor. These are IRQ (Interrupt ReQuest) and NMI (Non Maskable Interrupt). These are triggered by two separate wires on the processor. The most used form is the IRQ. When this occurs the processor looks at bit 2 of the status register − the interrupt disable flag. If this is SET then it IGNORES

the interrupt, otherwise it jumps to the address pointed to by &FFFE (low) and &FFFF (high). In contrast, the NMI is not masked by the interrupt disable flag and so cannot be ignored. It jumps to the routine pointed to by &FFFA (low) and &FFFB (high).

Because the NMI is unstoppable it is only used for very important periperhals such as the disc system and the Econet interface which need fast service to function properly. All the other peripherals are on the IRQ. Invetiably, servicing their interrupts takes time. If you are prepared to ignore all the hardware that is interrupt-driven you can speed up a program quite noticeably by disabling interrupts. To do this you must set the interrupt disable flag in the status register with the SEI command. It is important to clear the flag again, when you have finished, by using the CLI command.

The operating system now has a chance to service the peripheral that generated the interrupt. However, before it can do this it must save the registers on the stack. This way the routine can reload them before returning execution to the main program. If this is not done then the main program will suddenly find its registers have changed and will probably crash. To save the registers the operating system uses the following commands:

```
PHA
TXA
PHA
TYA
PHA
```

When the operating system has finished servicing the interrupt it must return to the main program. First it must reload the registers:

```
PLA
TAY
PLA
TAX
PLA
```

Then it must use the command RTI. This reloads the status register and program counter from the stack and allows the processor to carry on from where it left off before the interrupt occured.

Next we need to look at the way the operating system handles an interrupt. It only knows that an interrupt has been generated somewhere in the computer. It doesn't know which piece of hardware has generated it. To find out it must look at each piece of hardware in turn until it finds which is the culprit (it is possible, though unlikely, that two or more devices may have generated interrupts simultaneously).

Luckily, each piece of hardware that can generate an interrupt has a register stored in the memory which contains a flag bit which indicates whether it has generated an interrupt or not. There are more details on each piece of hardware in THE ADVANCED USER GUIDE.

The devices which can cause interrupts on a BBC Micro are:

NMI
1 Mhz bus
Econet interface
Disc interface

IRQ
TUBE interface
1 Mhz bus
Cassette / RS423
System VIA

The system VIA is the most interesting to us, as this generates all the interrupts that keep the computer working normally.

Because a device must have an interrupt flag in it for the operating system to check, devices that don't have such a flag cannot directly generate interrupts. Instead their interrupts are connected to some inputs on the system VIA. This has four inputs that can generate interrupts; and it has registers in it with flags for each input. The four devices in the

'Beeb' which can generate interrupts in this way are the light pen input on the analogue connector, the analogue to digital converter, the video controller and the keyboard.

We are most interested in the last two. The video controller generates an interrupt every time a vertical sync pulse is sent to the video monitor. This can be used for generating flicker-free graphics (see chapters 7 and 10). Here we will discuss the other interrupts and will also discuss events.

The system VIA

The keyboard generates an interrupt each time a key is pressed. When this happens the operating system looks to see which key has been pressed and updates the keyboard buffer. If you are writing a game and you don't need to use the GET command then you can disable this interrupt to speed up the game as this will not prevent you from using INKEY with a negative number.

There are also two timers in each VIA (user and system) which can be used either to generate an interrupt on a regular basis or to provide a single interrupt after a set amount of time.

The system VIA is memory mapped as sixteen registers at addresses &FE40 to &FE4F. These are regarded as registers zero to fifteen. (The user VIA is mapped similarly but at addresses &FE60 to &FE6F.) The VIAs are quite complicated to use and a lot of their functions are not particularly useful. There is a full description of them in THE ADVANCED USER GUIDE.

Let's first look at the four interrupts for the system's VIA. Its register 12 controls how these are used. For all the interrupts to work correctly this register must contain either 4 or 5. Normally it contains 4. This causes an interrupt at the end of the vertical sync pulse. By setting it to 5 the interrupt is caused at the beginning of the sync pulse. This is about two pixels earlier vertically. This may seem pretty pointless, but if you are using *FX19 for flicker-free graphics and the graphics flicker just at the top two pixels then this should cure it.

Normally you would not have to alter register 12.

Actual control of interrupts is done using registers 13 and 14. Register 14 is used to enable and disable the various interrupts that the VIA can produce and can only be written to. When writing to register 14, if bit 7 is set, then one in any other of the bit positions will enable the corresponding interrupt; if bit 7 is clear, then a one will disable the corresponding interrupt. This means that any interrupt can be enabled or disabled without affecting the other interrupts. Bits 0 to 6 of the system VIA's register 14 represent the following interrupts:

BIT	INTERRUPT
0	Keyboard
1	Vertical sync
2	Shift register
3	Light pen
4	A to D converter
5	Timer 2
6	Timer 1

Register 13 contains the interrupt flags themselves for each part of the VIA. Each of bits 0 to 6 represents an interrupt as in register 14. If a bit is set this means that the relevant part of the VIA has caused an interrupt. Also, bit 7 is set if ANY one of the other bits is set. This provides a quick way for the processor to check if the VIA is responsible for the interrupt − it looks at bit 7 first.

These bits will not clear themselves so the first job the interrupt routine must do, once it has identified which interrupts have occurred, is to clear any bits that are set, ready for the next interrupt. To do this it must write to register 13 with the corresponding bits of the flag to be cleared, set. Note that writing a one to bit 7 of this register has no effect − in other words, to clear bit 7 you have to clear ALL the other bits.

There are also a number of jobs, such as updating the TIME clock, that the computer must do regularly. This is done using TIMER 1 in the system VIA. This is set to produce an interrupt every hundredth of a second. It is probably the most useful interrupt

to us as it can enable us to do a bit of extra processing every hundredth of a second. This means that we can run two programs simultaneously so long as one of them does not need much processing time and breaks down into convenient short sections which can be executed every hundredth of a second.

Interrupt-driven music Now that we have seen a bit about how interrupts work we can look at an example. Because of the nature of interrupts they can, to a limited extent, to make the computer seemingly do two jobs at once. In this example we are going to make the computer play a tune, using interrupts. This will leave the computer free to do almost everything else (apart from using the sound port) in the meantime.

To make the program simple we will take a tune that can be played on channels one to three without envelopes. To make the program as 'transparent' to the user as possible we will not use any zero page addresses but will use variables stored directly after the program. The program itself can be placed in page ten of memory. The data for the tune can be placed in page nine. We will need five variables: TIME which will count the interrupts to produce a regular beat; COUNT which will count the beats for an individual note; POINT which will point into the note table; and TEMPX and TEMPY for temporary storage of the registers. The first three we can set to their initial values for the start of the tune. We will also need an eight-byte OSWORD command block for the SOUND command. While we are setting this up we can set some of the eight bytes that will not change throughout the program, so saving a few bytes of program.

```
.time    EQUB 1
.count   EQUB 1
.point   EQUB 0
.tempx   EQUB 0
.tempy   EQUB 0
.cblock  EQUD 0
         EQUD &FF0000
```

(90-150)

The main IRQ intercept routine will start at IRQ so first we need an initialisation routine to set up the IRQ vector.

We must first set the interrupt disable flag to prevent an interrupt occurring while we are changing the vector.

Then we must make a copy of the contents of the vector. This is so that when we have finished our interrupt work we can pass the interrupt on to the usual operating system routine.

Then we must reset the IRQ vector to point to our own routine.

Finally, we need to clear the interrupt disable flag and return.

```
.init   SEI             \ Set irq vector
        LDA &204        \ to point to our
        STA &230        \ routine.
        LDA &205
        STA &231
        LDA #irq MOD256 \ Set IRQ vector
        STA &204        \ to irq.
        LDA #irq DIV256
        STA &205
        CLI
        RTS
```

(160-260)

Now we can write the main program. The first thing this must do is to set the interrupt disable flag. This should stop any untimely interruptions. Next we must save the registers on the stack. The accumulator has already been stored at &FC for us by the operating system, so we need only save the X and Y registers.

```
.irq    SEI
        TXA
        PHA
        TYA
        PHA
```

(270-310)

Next we need to check that TIMER 1 is responsible for the interrupt. We examine bit 6 of register 13 of the system VIA. If this is set then TIMER 1 is responsible. We must not reset this flag as the operating system also wants a chance to service this interrupt. If TIMER 1 is not responsible then we can let the operating system cope with the interrupt. First we must reload the X and Y registers from the stack and then jump back to the normal IRQ routine in the operating system.

```
        LDA #&40
        BIT &FE4D
        BNE irq1
.exit   PLA
        TAY
        PLA
        TAX
        JMP (&230)
```

(320-390)

We now have a routine that is called every hundredth of a second. However, we only want to change the notes of our tune every eight hundredths of a second, otherwise the tune would be much too fast. To do this we decrement the variable TIME every hundredth of a second and, every time it reaches zero, reset it to eight and call the music routine.

```
.irq1   DEC time
        BNE exit
        LDA #8
        STA time
```

(400-430)

Now we have the problem that some notes are longer than others. If we have the length of the previous note in eight-hundredths of a second

stored in COUNT then we can decrement it each time until it reaches zero, at which point we can play the next note.

```
DEC count
BNE exit
```

(450-460)

Now we are almost ready to play a note. However, before we do this we must look at the way the notes are stored in the table. For this program they are stored four bytes per note. The first byte is the length of the note in eight-hundredths of a second and the other three bytes are the pitches of the three channels.

So the first thing we must do is to store the length of the note in COUNT. The pointer into the table (which is less than 256 bytes long) is stored in POINT. This pointer counts in bytes so we must load it into the X register to use ABSOLUTE addressing.

```
LDX point
LDA &900,X
STA count
```

Now we must play the three notes of the chord. We can do this with a loop using the Y register to count with.

```
LDY #3
```

(460)

Firstly we must increment X to point to the second byte of the entry. Then we must set up the OSWORD command block. The layout of the block for a SOUND command is that the first two bytes are the channel number, the next two bytes are the amplitude, the next two bytes are the pitch and the last two bytes are the duration.

Now we must set the channel number. We are also going to use the flush control so that each note

wipes out the previous one. This is to make each note start during the interrupt. We will also make the duration 255 so that each note carries on until the next note is played. We have the channel number in Y − we only have to add &10 to set the flush control. The high byte of the channel number is already zero:

```
.channel INX              \ Set up each
         TYA              \ channel.
         ORA #&10
         STA cblock
```

(500-530)

Next we have the volume. For this tune we need some short rests to stop notes running into each other (a sort of staccato effect). For these notes the pitch number is zero. We must first set the volume to zero and then look at the pitch. If the pitch is zero then we have finished setting up the command block, otherwise we must set the volume to -15 (&FFF1 in two's compliment) and set the pitch accordingly (again, the high byte of the pitch is already zero).

```
         LDA #0
         STA cblock+2
         STA cblock+3
         LDA &900,X
         BEQ rest
         LDA #&F1
         STA cblock+2
         LDA #&FF
         STA cblock+3
         LDA &900,X
         STA cblock+4
```

(540-640)

Notice that the duration of the note is already set to 255.

Next we must save the X and Y registers and then

set them to point to the command block. We must set the accumulator to seven for a SOUND command and call OSWORD.

```
.rest    STX tempx
         STY tempy
         LDX #cblock MOD256
         LDY #cblock DIV256
         LDA #7
         JSR &FFF1
```

(650-700)

Now we must reload the registers and if there is still a channel to be done we must go back and do it.

```
         LDX tempx
         LDY tempy
         DEY
         BNE channel
```

The X register now points to the fourth byte of the note in the table. By incrementing X it will point to the first byte of the next note and we can save it in POINT. If it has reached 168 then the whole tune has been played and we need to go back to the beginning by resetting POINT to zero. At this point we have finished with the interrupt routine and can pass control back to the operating system.

```
         INX
         STX point
         CPX #168
         BNE exit
         LDX #0
         STX point
         JMP exit
```

(750-810)

We have finished the machine code now, so we only have to set up the actual tune. The easiest way to do this is to put it into DATA statements after the assembly code (lines 850-1050). For convenience we

can set it up in lines of eight hexadecimal bytes run into a string. For example:

```
1000 DATA02685040025C5040
```

To place this data in the memory we need a few lines of BASIC. There are 21 lines of DATA for our tune so we need to read each one in as a string.

```
  10 FORA%=0TO20
  20 READA$
```

Next we need to extract the eight bytes from A$. These will be placed at &900 onwards. We can extract each byte using MID$. Then we precede the string we have obtained with & and use EVAL to find its value.

```
  30 FORB%=0TO7
  40 B%?(&900+A%*8)=EVAL("&"+MID$(A$,B%*2+1,2))
  50 NEXT,
```

Notice that line 840 calls INIT.

The complete programs with data looks like this:

```
  10 FORA%=0TO20
  20 READA$
  30 FORB%=0TO7
40 B%?(&900+A%*8)=EVAL("&"+MID$(A$,B%*2+1,2))
  50 NEXT,
  60 FORpass%=0TO2STEP2
  70 P%=&A00
  80 [OPTpass%
  90 .time    EQUB 1
 100 .count   EQUB 1
 110 .point   EQUB 0
 120 .tempx   EQUB 0
 130 .tempy   EQUB 0
 140 .cblock  EQUD 0
 150          EQUD &FF0000
 160 .init    SEI           \ Set irq vector
 170          LDA &204      \ to point to our
 180          STA &230      \ routine.
```

```
190          LDA &205
200          STA &231
210          LDA #irq MOD256
220          STA &204
230          LDA #irq DIV256
240          STA &205
250          CLI
260          RTS
270 .irq     SEI            \ Main routine.
280          TXA
290          PHA
300          TYA  310          PHA
320          LDA #&40
330          BIT &FE4D
340          BNE irq1
350 .exit    PLA
360          TAY
370          PLA
380          TAX
390          JMP (&230)
400 .irq1    DEC time       \ Centisecond
410          BNE exit       \ clock trapped.
420          LDA #8
430          STA time
440          DEC count
450          BNE exit
460          LDX point
470          LDA &900,X
480          STA count
490          LDY #3
500 .channel INX              \ Set up each
510          TYA              \ channel.
520          ORA #&10
530          STA cblock
540          LDA #0
550          STA cblock+2
560          STA cblock+3
570          LDA &900,X
580          BEQ rest
590          LDA #&F1
600          STA cblock+2
610          LDA #&FF
620          STA cblock+3
630          LDA &900,X
```

```
 640          STA cblock+4
 650 .rest    STX tempx
 660          STY tempy
 670          LDX #cblock MOD256
 680          LDY #cblock DIV256
 690          LDA #7
 700          JSR &FFF1
 710          LDX tempx
 720          LDY tempy
 730          DEY
 740          BNE channel
 750          INX
 760          STX point
 770          CPX #168
 780          BNE exit
 790          LDX #0
 800          STX point
 810          JMP exit
 820 ]
 830 NEXT
 840 CALLinit
 850 DATA0B64544D01000000
 860 DATA0364544D01000000
 870 DATA0368544801000000
 880 DATA0668544802000000
 890 DATA0364584801000000
 900 DATA06685C4802645C48
 910 DATA035C4B3B01000000
 920 DATA0354483801000000
 930 DATA0250402C0240402C
 940 DATA0248402C0250402C
 950 DATA0254402C025C402C
 960 DATA0264402C0268402C
 970 DATA0770645401000000
 980 DATA0768504001000000
 990 DATA0264544002545440
1000 DATA02685040025C5040
1010 DATA0370544001000000
1020 DATA03685C4801000000
1030 DATA0764544001000000
1040 DATA075C403801000000
1050 DATA0F54342401000000
```

Events There is, however, a simpler way to use interrupts:

an interrupt system called EVENTS. In this system ten common events that can occur are offered to the user for processing. These events are slightly slower to respond than interrupts. The events available are:

Event no.	Event
0	Output buffer empty.
1	Input buffer full.
2	Character entering input buffer.
3	ADC conversion complete.
4	Start of vertical sync.
5	Interval timer reaching zero
6	ESCAPE has occured
7	RS423 error
8	Econet event
9	User event

There is a complete description of events in THE ADVANCED USER GUIDE. If one of these events occurs, the operating system jumps to the subroutine pointed to by the vector at &220 and &221 (with a JSR command) with the event number in the accumulator. The X and Y registers may contain extra data according to the event.

Once an event routine has been set with the vector, the relevant event must be enabled. If more than one event is to be used, the event routine must test the accumulator first to check which event has occurred. If only one event has been enabled then this is not necessary. To enable an event use *FX14 with the number of the event.

Event routines must save all three registers, as with interrupts, and must not enable interrupts.

Probably the two most useful of these are events four and six. The vertical sync event (four) can either be used for flicker-free graphics (see chapters 7 and 10) or can be used to provide a regular interrupt every fiftieth of a second. For example, in our interrupt-driven sound program we could re-write the program in the following ways. The initialisation routine will need to be re-written.

```
160 .init    LDA #event MOD256
```

```
170          STA &220
180          LDA #event DIV256
190          STA &221
200          LDA #14
210          LDX #4
220          LDY #0
230          JMP &FFF4
```

We will need to delete lines 240 to 260. And the following lines must be changed in the main routine.

```
270 .event   PHA
280          DEC time
290          BEQ event1
```

Delete line 340.

```
385          PLA
390          RTS
```

Delete lines 400 and 410

```
420          LDA #4
```

Notice that, as the vertical sync only occurs half as often as the centisecond interrupt, the value stored in TIME is halved.

The other event that is quite useful is the ESCAPE event. When this is enabled, the normal action of ESCAPE is disabled − in fact, the ESCAPE key does absolutely nothing except generate an event. In a long machine code program such as a game this can be very useful, as it stops the ESCAPE key from having undesirable effects, but still enables it to be used to stop the game, etcetera.

In fact, because the relative simplicity of events makes them less versatile, they are not as useful as interrupts. For instance you can't use the interval timer in the system VIA using events. This timer is useful for accurate timing, as we shall see in chapters 7 and 10.

We will see the next few chapters just how useful interrupts can be.

BRK

Before we leave the subject of interrupts, there is one other machine code command we must discuss − BRK or (BReaK). Probably the closest equivalent to this in BASIC is the command STOP. BRK is always used with implied addressing.

When the processor comes across a BRK command it sets the Break flag in the status register and then carries on as if an IRQ has occurred. So, when the operating system handles an IRQ, it first has to check whether it is an IRQ or BRK that has occurred. There is no machine code command that gives direct access to the Break flag, so at first sight this would seem to be a problem.

However, as with an IRQ, when the processor comes across a BRK command, it pushes the program counter then the status register, on the stack. Thus the operating system only has to pull the top byte from the stack. This will be the contents of the processor status register at the moment when the BRK or IRQ occurred. By testing bit 4 of this, the operating system can tell which of the two has occurred.

On the BBC Micro BRK commands are used for trapping errors. When a language ROM has found an error (for instance a syntax error), it has a BRK command followed by the error number, the error message, and a zero. We can use this in our machine code programs if they are called from BASIC.

We can run the following program and BASIC will assume that the error is an ordinary BASIC error and will behave accordingly.

```
 10 ONERRORGOTO120
 20 DIMmc%25
 30 P%=mc%
 40 [OPT2
 50.error   BRK
 60         EQUB 255
 70         EQUS "An error has occured"
 80         EQUB 0
 90 ]
100 CALLerror
```

```
110 END
120 REPORT:PRINT" at line ";ERL
130 PRINT"Error number ";ERR
```

What happens is that when the operating system handles a BRK command it jumps to the Break vector at &202 and &203 (low, high). The A, X and Y registers are unchanged from when the BRK occurred. Also, the operating system pushes the status register from when the BRK occurred and the address of the next-but-one byte from the BRK command on the stack. This means that if the Break vector is set to point to an RTI command the processor returns to the next-byte-but-one from the BRK command, and then carries on from where it left off.

BASIC claims this vector and sets it to its own error handling routine. This routine pulls the top three bytes off the stack and discards them. Thus the BRK command has much the same effect as a jump to the error routine. It is this error routine that expects the error number and string directly after the BRK command.

Other languages may claim the BRK command for themselves simply by changing the contents of the Break vector.

Another example of the use of this vector is in intercepting errors in BASIC. The following program intercepts all BASIC errors. It must first save the two registers it uses to that the routine is 'transparent'. It then looks at the byte after the BRK command. It can do this because the operating system makes a copy of this address in the zero page locations &FD and &FE. If this error is an 'escape' (error number 17) then it ignores it, otherwise it prints a message before returning to the normal BASIC error routine.

```
10 FORpass%=0TO2STEP2
20 P%=&A00
30 [OPTpass%
40.brk     PHA
50         TYA
60         PHA
70         LDY #0
80         LDA (&FD),Y
```

```
 90         CMP #17
100         BEQ exit
110.loop    LDA fool,Y
120         JSR &FFE3
130         INY
140         CPY #13
150         BNE loop
160.exit    PLA
170         TAY
180         PLA
190         JMP (&230)
200.fool    EQUS CHR$13+"Silly Billy!"
210 ]
220 NEXT
230 ?&230=?&202
240 ?&231=?&203
250 ?&202=0
260 ?&203=10
```