7 Interrupts

7.1 An introduction to interrupts

An interrupt is a hardware signal to the microprocessor. It informs the 6502 that a hardware device, somewhere in the Electron or on an expansion module, requires immediate attention. When the microprocessor receives an interrupt, it suspends whatever it was doing, and executes an interrupt servicing routine. Upon completion of the servicing routine, the 6502 returns to whatever it was doing before the interrupt occured.

A simple analogy of an interrupt is a man working hard at his desk writing a letter (a foreground task). Suddenly the telephone rings (an interruption). The man has to stop writing and answer the telephone (the interrupt service routine). After completion of the call, he has to put the telephone down, and pick up his writing exactly where he left off (return from interrupt).

In an Electron, the main objective is to perform foreground tasks such as running BASIC programs. This is equivalent to writing the letter in the above example. The computer may however be concerned with performing lots of other functions in the background (equivalent to the man answering the telephone). An Electron which is running the house heating system for example would not wish to keep on checking that the temperature in every room is correct – this would take up too much of its processing time. However, if the temperature gets too high or too low in any of the rooms it must do something about it very quickly. This is where interrupts come in. The thermostat could generate an interrupt, causing the computer to jump quickly to the interrupt service routine, switch a heater on or off, and return to the main program.

There are two basic types of interrupts available on the 6502. These are maskable interrupts (IRQs) and non-maskable interrupts (NMIs). To distinguish between the two types, there are two separate pins on a 6502. One of these is used to generate IRQs (maskable) and the other is used to generate NMIs (nonmaskable).

7.1.1 Non-Maskable Interrupts

In order to generate a non-maskable interrupt, a piece of hardware must pull the NMI line low. This forces the 6502 to stop whatever it was doing, and to start executing the NMI service routine at &0D00. NMLs are extremely powerful, because they cannot be turned off under software control. If the ULA is currently accessing RAM to produce the video display in modes 0 to 3, it is also forced to give the memory back to the 6502. NMIs can therefore create snow on the screen – the urgency of this signal is such that even the screen cannot take priority over the interrupting device.

Only very high priority devices, such as the Floppy Disc or Econet interfaces, are allowed to generate NMIs. This ensures that the 6502 is only interrupted in very urgent situations. These high priority devices are then guaranteed to get immediate attention from the 6502. To return to the main program from an NMI, an RTI instruction is executed. It is always necessary to ensure that all of the 6502 registers are restored to their original state before returning to the main program. If they are modified , the main program will suddenly find garbage in its registers in the middle of some important processing. It is highly probable that a total system crash would result from this.

7.1.2 Maskable Interrupts

Maskable interrupts are similar to non-maskable interrupts in most respects. A hardware device can generate a maskable interrupt to which the 6502 must normally respond. The difference is that the 6502 can choose to ignore all maskable interrupts, if it so desires, using software control. To disable interrupts (only the maskable ones though), an SEL (set interrupt disable flag) instruction is executed. Interrupts can be reenabled at a later time using the CLI (clear interrupt disable flag) instruction.

When an interrupt is generated, the processor knows that an interrupt must have come from either the ULA, or an expansion module device. Initially though, it can't tell where the interrupt has come from. If there was only one device that could have caused the interrupt, then there would be no problem. However,

since there is more than one device causing interrupts in the Electron, each device must be interrogated. Each device is asked whether it caused the interrupt, This is normally quite easy, because all of the standard Electron devices are controlled by the ULA register at address &FE00. Any other devices connected to the expansion bus would have to be interrogated seperately.

When the interrupt processing routine has discovered the source of a maskable interrupt, it must decide upon the type of action is required. This usually involves transferring some data to or from the cassette interface, incrementing the clock, or flashing the colours on the screen. The interrupt condition must then be cleared by writing to &FE05. This is because most devices (except the cassette receive and transmit registers) continue to signal an interrupt until they have been serviced. The completion of servicing often has to be signalled by the processor writing to a special register in the device, or, in the case of interrupts from the ULA, to address &FE05.

Interrupts must **never** affect the interrupted program. All of the processor registers and flags must therefore be exactly the same after return from an interrupt routine as they were before the interrupt occured. Thus an interrupt routine must either not alter any registers (which is difficult) or restore all register contents to their original values before returning.

Interrupt routines are entered with interrupts disabled. An additional interrupt will therefore not be recognised whilst the first interrupt routine is still processing. If the interrupt service routine is going to take an appreciable time, this could create problems. Other more urgent interrupts may occur, and have to wait until the previous one has finished processing. The solution is normally to ensure that interrupt routines are not too long. However, if care is taken, interrupts can be re-enabled inside a long interrupt routine, In this case, fixed memory locations must **not** be used to store variables within the routine, because these locations will be overwritten if another interrupt routine uses them (or indeed if the same interrupt occurs again!). All variables should therefore be stored on the stack so they can be restored at the end of any routine.

7.2 Interrupts on the Electron

Interrupts are required on the Electron to process all of the *background* operating system tasks. These tasks include incrementing the clock, processing envelopes, or transferring keys pressed to the input buffer. All of these tasks must continue whilst the user is typing in, or running his program. Using interrupts gives the impression that there is more than one processor; one for the user, one for updating the clock, one for processing envelopes, etc.

As was mentioned in the introduction, normal (maskable) interrupts can be disabled. Interrupts should only be disabled for critical operations. For example, when changing the two bytes of a vector. If an interrupt occurs in the middle of the change, it might be indirected to an erroneous address.

When interrupts are disabled, the clock stops, and all other interrupt activities cease. Interrupts are disabled by the SEL assembler instruction, and re-enabled with CLI. Most devices that generate interrupts will continue to signal an interrupt until it is serviced. The cassette read register is one exception. If it isn't serviced within 2ms, data from the cassette will almost certainly be lost forever.

7.3 Using Non-Maskable Interrupts

Generally, NMIs are reserved for specialised pieces of hardware which require very fast response from the 6502. NMIs are not used on a standard system. They are used on DISC and ECONET systems. An NMI causes a jump to location &0D00 to be made.

7.4 Using Maskable Interrupts

Most of the interrupts on the Electron are maskable. This means that a machine code program can choose to ignore the interrupts by disabling them. Since all of the operating system features such as scanning the keyboard, updating the clock, and running the cassette system are run on an interrupt basis, interrupts should never be disabled for more than about 2ms.

There are two levels of priority for maskable interrupts, defined by two indirection vectors in page &02. The priority of an interrupt indicates its relative importance with respect to other interrupts. If two devices signal an interrupt simultaneously, the higher priority interrupt is serviced first.

7.5 Intercepting interrupts

Maskable interrupts can be intercepted on the Electron, and redirected to a user specified address. This interception process consists of changing the value of a vector.

There are two interrupt interception vectors called IRO1V and IRO2V, The first of them is indirected via the vector stored at &204,5 and the second via &206,7. If either of the vectors stored in these locations is changed to point at a user supplied routine, that user routine will be called when there is next an interrupt.

Interrupt Request Vector 1 (IRQV1)

Indirects through &204,5

This is the highest priority vector through which all maskable interrupts are indirected, This is nominally reserved for the system interrupt processing routine, which copes with all of the interrupts from the ULA. Any interrupt which cannot be dealt with by the operating system routine (those which are generated by a user expansion module) are passed on through the second interrupt vector, IRQ2V. Occasionally, IRQ1V can be intercepted before the operating system gets hold of it. This will only be necessary for high priority user interrupts.

Interrupt Request Vector 2 (IRQ2V)

Indirects through &206,7

This vector is normally used to deal with any interrupts which cannot be dealt with by the operating system. On an unexpanded Elecctron, the vector simply points to a couple of lines of code to restore the A register from &FC, then return from the interrupt service.

Several points should be born in mind when producing interrupt service routines.

a) When the vector value is changed to point at the new user supplied routine, the previous contents of the vector should be saved somewhere. This will allow the user routine to go on to the correct address after it has finished, Note that this method of linking into IRQ1V or IRQ2V allows several independent routines to link in seperately. Each stores the previous contents of the vector (which point to the next routine).

b) Disable interrupts using the SEI instruction before changing the contents of the interrupt vectors, This is merely a precaution to guard against the possibility of interrupts occuring between writing the low and high bytes of the vector If an interrupt were to occur in the middle of this operation, the indirection vector would be erroneous, and would probably cause the machine to crash.

c) The conditions which will be in force when the user routine is entered are that; the original 6502 status byte and return address are already stacked on the 6502 stack (ready for an RTI instruction to resume normal operation). The X and Y registers are still in their original states, but haven't been saved anywhere. The original A register contents are in location &FC.

d) Operating system calls should not normally be made from within an interrupt service routine, This is because they may not be re-entrant (eg. if any zero page locations are used). Most OSBYTEs and some OSWORDs are ‘IRO-proof’. Avoid *FX0, OSBYTE &81 (positive INKEY), fast Tube BPUT, OSWORD 0, and all VDU OSWORDs except palette write/read. Such use of OS calls will often cause the foreground task to be disturbed and crash.

e) The user's interrupt routine should be re-entrant. This means that if there is a possibility of interrupts being re-enabled during the routine (eg. because it is very long), the code can be run again without affecting the first foreground interrupt. This can only be done by pushing the X and Y registers plus the contents of &FC onto the stack, and restoring them after

the call. It is also important to ensure that no fixed memory locations are used for storing variables, since these will be overwritten by an interrupting routine.

The following example illustrates most of these points. When run, it will cause the Electron to make a continuous decreasing pitch tone.

Several points in the program are worthy of note. The first is that IRQ1V is used instead of IRQ2V. On an unexpanded Electron, all interrupts are serviced by IRQ1V, so the OS doesn't bother to pass them on to IRQ2V, When the tone is running, switch the listing to page mode (by pressing CTRL N). Then list the program. The sound is totally messed up because the OS is writing to the ULA as well. This illustrates one of the reasons why the official operating system calls should normally be used – to avoid clashes like that.

```
 10 REM Interrupt utilisation example
 20 REM Must operate in mode 6
 30 MODE 6
 40 REM AlLocate space for program
 50 DIM M% 100
 60 FOR opt%= 0 TO 3 STEP 3
 70 P%=M%
 80 [
 90 OPT opt%
100 .init   SEI                 \ Disable interrupts
110         LDA &204            \ Save old IRQ1V vector
120         STA oldv
130         LDA &205
140         STA oldv+1
150         LDA #int MOD 256    \ Low byte of address
160         STA &204            \ IRQ1V Low
170         LDA #int DIV 256    \ High byte of address
180         STA &205
190         CLI                 \ Turn interrupts on again
200         RTS                 \ Exit initialisation routine
205
210 \ This is the interrupt service routine
220 .int    TXA                 \Save X register
230         PHA
240         TYA                 \Save Y register
250         PHA
260         INC &70             \ Counter in zero page
270         LDA &70
```

```
280        STA &FE06           \ Load into ULA counter
290        LDA #&32            \ Set sound mode
300        STA &FE07           \ Write to ULA control register
310        PLA                 \ Restore the registers
320        TAY
330        PLA
340        TAX
350        JMP (oldv)          \ Go on to next service routine
355
360 .oldv  EQUW0               \ Reserve space for old vector
370 ]
380 NEXT opt%
390 REM Grab the interrupt vector
400 CALL init
410 REM Bleeping should now start
420 END
```