Simon’s Mostly Reliable Guide to Interrupts

Table of Contents

Introduction

This document tries to explain the “why”, “what” and “how” details of interrupt handling in Commodore 8-bit computers. It specifically describes the behaviour of the VIC-20 and C64, other systems may have differences to what is described here.

In contrast to many other computers of the era Commodore designed a flexible hardware and software system that allowed new peripherals to be created and applications to interface with them.

Terminology

I will use the same terms commonly referred to in Commodore documentation and other reference material. These will be shown in *italics* when they are first mentioned.

Classes of Interrupts

There are three types of event that cause the processor to stop its current sequence of instructions and do something else (an *interrupt*). Listed in priority order they are:

1. Reset – While not usually considered an interrupt a hardware reset is processed in the same manner as other interrupts so will be included here
2. Non-maskable Interrupt (*NMI*) – This class of interrupt cannot be ignored (or *masked*) by software
3. Normal interrupt (*IRQ*) – Software may choose to defer handling this class of interrupt while it performs other time-critical tasks

Typical Interrupt Handling

The following sequence is usually used to process an interrupt:

1. A hardware condition occurs, for example, a timer expires
2. The appropriate control line to the processor is taken from high (1) to low (0)
3. The processor jumps to an address (or *vector*) specific to the control line
4. The software discovers the source of the interrupt by examining registers in the devices attached to the control line
5. The software performs the appropriate routine to process the interrupt, one task is to clear the interrupt source
6. The control line becomes high again

Processing of one class of interrupt may start while a lower priority one is already occurring.

Reset Handling

A timer circuit is used to cause the $\overline{\text{RESET}}$ line of the processor to go low momentarily when the computer is powered on. The processor reads the vector at $FFFC and jumps to that address. A reset can also be triggered by a device attached to the expansion port or user port.

The *cold start* behaviour on Commodore systems depends on if an *autostart* cartridge is inserted. The KERNAL reset routine checks for an *autostart signature* in memory and, if that is found, then jumps to another vector (usually also within the cartridge).

On the VIC-20 the autostart signature will be present at $A004 and is the sequence

> $41,$30,$C3,$C2,$CD – “A0”,‘C’+$80,‘B’+$80,‘M’+$80

On the C64 the autostart signature will be present at $8004 and is the sequence

> $C3,$C2,$CD,$38,$30 – ‘C’+$80,‘B’+$80,‘M’+$80,“80”

The cold start vector used if the signature is found is $A000 on the VIC-20 and $8000 on the C64.

If the signature is not found the KERNAL performs the following:

1. Initializes and tests RAM
2. Initializes KERNAL vectors
3. Initializes registers of I/O devices
4. Enables interrupts
5. Calls the BASIC cold start routine

NMI Handling

In the VIC-20 the $\overline{\text{NMI}}$ line is connected to a Versatile Interface Adapter (*VIA*), if any of the conditions under which that device can trigger an interrupt occurs then an NMI is generated. The **[RESTORE]** key is attached to the CA1 line. In the C64 the $\overline{\text{NMI}}$ line is connected to a Complex Interface Adapter (*CIA*) and also a circuit attached to the **[RESTORE]** key. An NMI can also be triggered by a device attached to the expansion port.

When the $\overline{\text{NMI}}$ line transitions from high to low the processor completes the current instruction then performs the following:

1. Pushes the *Program Counter* on to the stack
2. Pushes the *Status Register* on to the stack
3. Sets the *Interrupt Disable* bit in the Status Register
4. Reads the vector at $FFFA and jumps to that address

The code in the KERNAL immediately jumps to a vector held in RAM at $0318. The default KERNAL routine at that address first pushes the registers on to the stack in the order

.A, .X, .Y

then handles the following interrupts:

- The **[RESTORE]** key being pressed
- An event related to the RS-232 serial port

When the interrupt source has been handled the routine performs the following:

1. Pulls the registers from the stack
2. Pulls the Status Register from the stack
3. Pulls the Program Counter from the stack and resumes from the point the NMI occurred

The $\overline{\text{NMI}}$ line is edge-triggered: all interrupt sources must be cleared so the line returns to high before another interrupt can occur.

Warm Start Handling

On the VIC-20 and C64 the **[RESTORE]** key is used in combination with the **[STOP]** key to return the system to a working state if it becomes unresponsive (a *warm start*).

The KERNAL routine checks for an autostart signature, if one is found it then jumps to the vector at $A002 on the VIC-20 and $8002 on the C64. Otherwise the routine:

1. Clears the interrupt source in the VIA (VIC-20 only)
2. Updates the real-time clock
3. Checks if the **[STOP]** key is pressed

If the **[STOP]** key is also down the routine performs the following:

- Initializes KERNAL vectors
- Initializes registers of I/O devices
- Calls the BASIC warm start routine (which re-enables interrupts)

RS-232 Processing

The RS-232 serial port in the VIC-20 and C64 is implemented entirely in software (bit banging). Timers are used to measure the interval the transmit and receive lines move between 0 and 1. On the

VIC-20 the T1 timer of the VIA is used for bit transmission and the T2 timer for reception. On the C64 timer A of the CIA is used for bit transmission and timer B for reception.

The receive line is also attached to an I/O pin that generates an interrupt when a start bit appears. On the VIC-20 this is the CB1 line on the VIA, on the C64 the $\overline{\text{FLAG}}$ line of the CIA.

The KERNAL RS-232 routines support data in both directions simultaneously (*full duplex*). The NMI routine performs the following:

1. Checks if the transmit timer has expired. If so the next outgoing bit is sent
2. Checks if the receive timer has expired. If so the next incoming bit is received
3. Checks if the receive start bit has occurred. If so the receive timer is started

IRQ Handling

In the VIC-20 the $\overline{\text{IRQ}}$ line is connected to a VIA, if any of the conditions under which that device can trigger an interrupt occurs then an IRQ may be generated. In the C64 the $\overline{\text{IRQ}}$ line is connected to a CIA and also the Video Interface Chip (VIC-II).

The KERNAL sets a repeating timer that triggers an interrupt every 1/60th of a second. On the VIC-20 the T1 timer of the VIA is used, on the C64 timer A of the CIA.

An IRQ is the lowest priority interrupt, it only occurs of both the following conditions are true:

- The $\overline{\text{IRQ}}$ line is low
- The Interrupt Disable bit in the Status Register is clear

The processor completes the current instruction then performs the following:

1. Pushes the Program Counter on to the stack
2. Pushes the Status Register on to the stack
3. Sets the Interrupt Disable bit in the Status Register
4. Reads the vector at $FFFE and jumps to that address

The code in the KERNAL at that address first pushes the registers on to the stack in the order

.A, .X, .Y

then jumps to a vector held in RAM at $0314.

The default KERNAL routine performs the following:

1. Updates the real-time clock
2. Blinks the cursor if the countdown counter reached zero
3. Checks the cassette switch and starts or stops the motor accordingly
4. Scans the keyboard and inserts a character into the keyboard buffer
5. Clears the timer interrupt
6. Pulls the registers from the stack

7. Pulls the Status Register from the stack
8. Pulls the Program Counter from the stack and resumes from the point the IRQ occurred

The $\overline{\text{IRQ}}$ line is level-triggered: if an interrupt source is still asserted when the processor returns from the interrupt routine then another IRQ will immediately occur (unless the Interrupt Disable bit is set).

Use of Interrupts by the Cassette System

When transferring blocks to and from the cassette the KERNAL replaces the normal IRQ routine with one of three routines:

READT – read a stream of cycle pairs and convert them into bytes of data

WRTZ – write a tape leader or trailer

WRTN – write a block of bytes as a stream of cycle pairs

These routines use timers to detect the frequency of incoming cycle pairs and to control the duration of outgoing cycle pairs.

The cassette KERNAL routines wait for the normal IRQ vector to be restored before returning.

Software Interrupts

In addition to interrupts generated by a hardware event the 6502 also has the **BRK** instruction. Executing this causes the processor to perform the following:

1. Increments the Program Counter by two bytes
2. Pushes the Program Counter on to the stack
3. Pushes the Status Register with the Break bit set on to the stack
4. Sets the Interrupt Disable bit in the Status Register
5. Reads the vector at $FFFE and jumps to that address

This is the same vector as is used for an IRQ, the KERNAL routine examines the value of the Break bit in the Status Register that was pushed on to the stack. When this bit is set the code jumps to a vector held in RAM at $0316.

When the user-supplied routine completes it must:

1. Pull the registers from the stack
2. execute an **RTI** instruction

The processor will resume execution two bytes after the **BRK** instruction, in other words it acts like it had a one byte operand (that is otherwise unused).

> Warning – The NMOS 6502/6510 used in the VIC-20 and C64 has a design defect that means if an NMI occurs at the point the processor executes a **BRK** instruction the interrupt will be processed and the **BRK** instruction skipped.