Amélie IRQ handler

Preliminary specification - 2006/11/12

Introduction

The IRQ system is one of the primary parts of Amélie. It is, by necessity, the main point of responsiveness of the system.

Amélie's BIOS works using a vectored approach. Upon certain conditions happening, some actions will be performed, and then a vector will be called. The vector locations are held in RAM, at the end of Page Zero.

By default, the vectors point to one of:

* Ignore code
  This is simply code that returns to the caller. Obviously, multiple vectors may point to this code.

* Cancel code
  This code reads or writes to the hardware for the effect of clearing the hardware. The event itself is discarded.

It is intended that the application code will replace the vector pointers to its own code.

RESETs and NMIs

The **RESET** handler points to reset-specific code which *always* causes a full reset. The use of the BRK instruction is not supported within Amélie.

Amélie makes no use of **NMI**s, so the hardware NMI vector calls a software vector, which by default calls the RESET vector. The implication is to force a reset upon an NMI as it is supposed to be held inactive (via a link); yet allowing for the possibility of NMIs in the future…

As for the **IRQ** handling, here goes with the pseudo-code:

IRQs

The IRQ handler has to intercept interrupts from two sources - the versatile interrupt adapter (VIA), and the asynchronous communications interface adaptor (ACIA).

The ACIA deals with serial communications to and from a host computer. The BIOS offers interrupt-driven serial handling; the application code need only push data to a buffer and await the BIOS having sent the data. Thus, the BIOS will usually handle the serial system by itself.

The VIA provides two functions. The first function is a system timer that is triggered 50 times a second. This provides an accurate time reference for the system, as it runs independently off of the 2MHz system clock. The second function provided is two 8 bit ports which can be either inputs or outputs. Because these are part of the "open specification" nature of Amélie, once the interrupt handler knows that the interrupt was not the 50Hz ticker, it will call a vector. By way of example, *RickBot* will use port A for sensor inputs and an IIC bus, and port B for motor control.

.IRQ handler
The processor is directed here upon an IRQ occurring. PSR, A, X, and Y are stacked, in that order. Generally, each IRQ call will take *one* path through the code and will unstack and `RTI` as soon as possible. Multiple simultaneous interrupts will require multiple IRQ calls.

* Processor status & registers are stacked.

* Call the *IRQENTRY* vector.
  This is for if you wish to completely replace the internal IRQ system, or to pick up on a specific interrupt early.

* Check **VIA** status flag.
  If a VIA interrupt, go to *VIAIRQ* code. This code returns itself.

* Check **ACIA** status flag.
  If an ACIA interrupt, go to *ACIAIRQ* code. This code returns itself.

* Call *IRQUNKNOWN* vector.

* Processor status and registers restored, then return from interrupt.

.VIAIRQ
This is called as soon as we have narrowed the IRQ to being from the VIA. We first check to see if the 50Hz ticker caused the event. If it did, then we handle the system tick functions. Otherwise, we call a vector to which VIA code should be attached.

* Examine flags - was it the 50Hz ticker?
  If so, go to *TICKER* handler. This returns itself.

* If not, call *VIAIRQ* vector, which should return itself.

.ACIAIRQ
This is called upon narrowing the IRQ down to having originated in the ACIA. We call a vector which, normally, points to the BIOS serial handler.

* Call *ACIAIRQ* vector.

.TICKER
This is called, by a 2ms timeout on the VIA (40,000 clock ticks) 50 times per second. It provides the internal timing facilities for the watchdog and for the automatic LED blinking.

* Decrement 50Hz ticker byte.
  If 50Hz ticker byte *is zero*:

  * Reset ticker byte to 49.

  * Decrement watchdog byte.
    We do *not* check watchdog here…

  * Increment system second count.
    If `second = 60`, call *TIMEPATCHUP* code.

* Call *LEDFLASH* code.
  If watchdog byte *is zero*, call the *WATCHDOG_CRITICAL* handler.

* Restore processor registers/status and then **RTI**.

.TIMEPATCHUP

This is simply a subroutine to handle "new second", and any cascade patchups that may be required.
Only *one* flag is maintained, so the vector called (out of "NEWSECOND", "NEWMINUTE", and "NEWHOUR") is that of the highest precedence. If you receive a "NEWMINUTE" call, you can assume a "NEWSECOND" also occurred.
You will *always* receive "TIMEWRAPPED" upon time wrapping, which will be followed by "NEWHOUR".

* Flag "*new second*"

* If `seconds > 59`
  `minute =+ 1`
  `seconds = 0`
  Flag "*new minute*"

* If `minute > 59`
  `hour =+ 1`
  `minute = 0`
  Flag "*new hour*"

* If `hour > 23`
  `hour = 0`
  Call *TIMEWRAPPED* vector. [we don't yet handle "days"]

* If flag is "*newsecond*", call the *NEWSECOND* vector.
  If flag is "*newminute*", call the *NEWMINUTE* vector.
  If flag is "*newhour*", call the *NEWHOUR* vector.

* Restore processor state and registers and `RTI`.

.LEDFLASH
This code deals with the LED flash functions. In Page Zero are fourteen bytes. The first four contain the LED state (0 = off, 1 = on). The next four contain the flash counter, which is decremented every 50Hz tick, and the final four contain the flash reload counter, the value that is reloaded into the flash counter upon it reaching zero. Following is a byte that represents is *zero* if no LEDs are in flash state, 1 otherwise; and finally a workspace byte.
If may appear wasteful to have four *bytes* for the LED state when four *bits* would suffice. We do it this way in order that we only need an LDA to get the status of each individual LED. The only time we mess with bit shifting is for constructing the final bit pattern to send to the latch.
In order to optimise the code, you should use the OS routines to set the flash rates - because otherwise it will be assumed that the LED status is static and the flash will never occur.
Here is how it works in practice:

* If LEDs are static, return.

* Set LED workspace byte to zero.

* If LED1 flash counter is non-zero:
  Decrement flash counter.
  If flash counter is now zero:
  Reload flash value into counter
  Invert state of LED
  Increment LED workspace byte.

* Ditto for LED2…
  …and LED3…
  …and LED4.

* If LED workspace byte is not zero (LED state changed), then:
  Construct bit pattern of LED state to send to the latch, and then write to hardware.

The LED code supports 6 modes of operation:

0 LED is off

1 LED is on

2 Fast blink LED (12 ticks, or ~ 1/4 second blink)

3 Medium blink LED (25 ticks, or 1/2 second blink)

4 Slow blink LED (37 ticks, or ~ 3/4 second blink)

5 Long blink LED (50 ticks, or 1 second blink)

The blink duration given specifies the time for which the LED is either off or on, not the combined time; thus mode = 2 means the LED is off for 12 ticks, on for 12 ticks, etc…
Just for reference, a static LED has a countdown value of zero, so the flash code skips over that LED.

If you wish for custom durations, the way to handle this would be to set a default flash mode, and *then* poke your desired duration into the flash reload byte.

.WATCHDOG CRITICAL

The watchdog starts at 255 and counts down at 50Hz. Periodically, the application code should reset the watchdog value to 255, for if this value ever reaches zero…
The *WATCHDOG_CRITICAL* vector is called. This vector normally points to code which does the following:

* *VIAIRQ* is set to point to "cancel" code.
  All VIA lines are set to be outputs.
  Zero is written to all VIA lines.
  All VIA lines are set to be inputs.

* The serial buffers are cleared and the serial handler *ACIAIRQ* is redirected to the watchdog handler.

* *All* LEDs are set to *fast* blink.

* *UNKNOWNIRQ* is set to "ignore" code.
  IRQMAIN is also set to "ignore" code.

Then a recursive loop is entered, which looks for a newline (ASCII 10) on the serial port, and also resets the watchdog.
There is no way out of this, except for a system reset.

If you connect a serial cable and press `Enter`, a number of hex values will be returned. The first value is the value of the stack pointer, this will be followed by all of the values in the stack. This, hopefully, will enable you to unwind the code to try to work out what went wrong.
The output will look like:

```
watchdog> F6 04 02 FF FE 3D 01 FB DE 00
```

Rick, 2006/11/12
heyrick *-at-* merseymail *-dot-* com
http://www.heyrick.co.uk/amelie/