Speeding up & Optimising demo routines

This article is mainly aimed for novice to intermediate coders of the Commodore 64, as professionals will already know this riff-raff.

For many coders, when they begin to write their very nice looking demo effects, they sometimes fall into the black hole of slow speed. This is due to how much rastertime their routines use. Code is very inefficient to timing if you tend to write everything in real-time. It just does not work. Therefore this information will hopefully get you to understand how to write efficient code and retrieve your speed of 50/60 frames per second (corresponding to PAL/NTSC machines respectively).

Problems with the Kernal

Kernal routines are perhaps the worst routines ever to use when it comes to consideration of speed. They are slow and use way too much raster time. At the time, these routines were developed to make use for programmers to write business software. Speed wise, this was not a major problem for the employees of Commodore Business Machines, but it is quite a problem for us demo sceners today. The best advice is to AVOID use of them at all times and instead write your own version of the same routines.

Here is a typical example, this is of course how you would call the kernal routine to clear the screen, but in a REPEATED LOOP via a simple raster compare:

```
        SEI
loop    LDA #$3B
        CMP $D012
        BNE *-3

        DEC $D020        ; This is to check how much rastertime is used via
scanlines.

        ; And now the slow routine...
        LDA #$01
        STA $0286        ; Set the cursor colour to white
                         ; ($01 = VIC colour white, $0286 = Current cursor
colour).
        JSR $E544        ; Kernal routine to clear the screen.

        INC $D020

        JMP loop
```

When you execute this routine, it uses more rastertime than the CPU can handle per frame, thus slowing down the computer. Code this in yourself and you will see what I mean.

Below is a screenshot of what the result looks like on a c64. Look closely at where the light green scanline-colour (controlled by DEC $D020) intersects with the scanline position that was compared by $d012 - #$3B. Because it intersects and passes scanline #$3b, the raster comparing has to go

through all scanlines again, causing the frame rate to decrease to 25(PAL) or 30(NTSC) - thus slowing the execution down by half of what it should be.

So how can we come around this problem?

We will now look at the code which does EXACTLY the same process, but hand coded by better knowledge:

```
        SEI
loop1   LDA #$3B
        CMP $D012
        BNE *-3
        DEC $D020        ; This is to check how much rastertime is used.

        ;and now OUR version of the routine...
        LDX #$00
loop2   LDA #$01         ; Here is where we store our character colour
        STA $D800,X      ; ...and now store it into colour memory ($d800-
$dbe8)
        STA $D900,X      ; ""
        STA $DA00,X      ; ""
        STA $DB00,X      ; ""

        LDA #$20         ; Here we store our character to be placed on each
part of the screen
        STA $0400,X      ; ...and now store it into screen memory (typically
$0400-$07e8)
        STA $0500,X      ; ""
        STA $0600,X      ; ""
        STA $0700,x      ; ""
        INX
        BNE loop2

        INC $D020
        JMP loop1
```

This method obviously uses more bytes in memory, but technically it is much faster than the kernal method. Code this in and notice the difference.

Here is a c64 screen shot of the better result. You can now see that light-green scanline colour finishes at between raster line #$F8-FF (done by INC $D020) - showing evidence that this method is faster.

If you want to keep your code fairly short, then by all means use the kernal routines, but use them OUTSIDE real-time procedures. The best place to use them is in the start-up section of your code.

If you use IRQ timing, you may use the kernal outside if you wish, but if you use kernal routines that modify the graphics in any way, you are more likely to see the ugly side effects on your screen, depending how much raster time your IRQ routines use.

You can also completely switch off the kernal by setting zero page $01. Bit position 1 (or #$02) of $01 sets the C64 memory chip to allow users to call built in routines of the Kernal ROM, stored from $E000 to $FFFF. By default, this is enabled when you switch on/reset your computer. By simply switching it off like this...

```
        LDA $01
        AND #$FD
        STA $01
```

...the KERNAL ROM is then disabled, and you are also free to write from $E000 to $FFFF RAM. It is a good thing to have the kernal switched off when wanting to execute code fast. The only downfall though is that when you begin to write IRQ routines, they need to be set up differently.

Here is a code example of a typical IRQ routine when the KERNAL is switched ON:

```
        SEI
        LDA #$01
        STA $D01A
        LDA #<irq
        LDX #>irq
        LDY #$32
        STA $0314
        STX $0315
        STY $D012
        LDA #$1B
        STA $D011
        LDA #$7F
        STA $DC0D
        LDA $DC0D
        CLI
        JMP *

irq
        (your code here)
        INC $D019
        JMP $EA7E
```

Of course there is theoretically nothing wrong with this method. This code however does actually itself use a particular kernal routine. JMP $EA7E is a kernal routine to call the next interrupt. The routine itself does not really use a lot of raster time, however the KERNAL ROM needs to be switched on for this to work.

You can however, still write an IRQ routine with the kernal switched OFF. Shown below is sample code of how to do this...

```
        SEI
        LDA #$35
        STA $01          ;Switch off the KERNAL ROM via value #$35
        LDA #$01
        STA $D01A
        LDA #<irq
        LDX #>irq
        LDY #$32
        STA $FFFE
        STX $FFFF
        STY $D012
        LDA #$1B
        STA $D011
        LDA #$7F
        STA $DC0D
        LDA $DC0D
        CLI
        JMP *

irq     STA $02
        LDA $DC0D
        STX $03
        STY $04
        (your code here)

        LDA #$01
        STA $D019
        LDY $04
        LDX $03
        LDA $02
        RTI
```

As you can see, there are quite a few changes compared to the previous example. First of all, the KERNAL is switched off (LDA #$35, STA $01). Because it is switched off, $0314 and $0315 (the IRQ memory pointers) are now useless, because they are used by the kernal routine, $EA7E. Therefore, $FFFE and $FFFF are used instead , as they are also IRQ registers but can be only be used when the KERNAL is switched off.

Now take a look at the “irq” code section. Notice there are three storage opcodes at the beginning and three load opcodes at the end? These need to be included because every time an interrupt is finished, the values stored in the accumulator (A), X and Y registers need to be exactly the same as

when the interrupt was first triggered. When you look at the ROM code of kernal routine $EA7E/EA31, they already do this for you. This concludes why you need to code everything yourself when the kernal is switched off. This of course may seem to be lot of work, but implementing this way will give you more available cycles for your hardcore demo routines.

Using efficient Opcodes

Over the years I have always took the pleasure of watching some really great demos and then take a peek at the source code with an MC monitor. In most demos that perform really cool effects in real-time, the actual assembly code seems to look too repetitive. This makes you think that the section of code uses up most of your memory, which in most cases is true. However the speed efficiency of the code is a lot better.

Here is a particular example…a section of code to perform a 3×3 scroller:

```
       ldx #$00
loop   lda $0401,x
       sta $0400,x
       lda $0429,x
       sta $0428,x
       lda $0451,x
       sta $0450,x
       inx
       cpx #$27
       bne loop

       (your scroll text code here)
       rts
```

Theoretically there is nothing wrong with this piece of code, as it simply performs a left scroller. Plus it uses less bytes, which memory-wise is pretty good. If however you need to perform this routine along side other real-time effects that use up most of the raster lines, then there could be a situation where there is not enough cycles left for the scroller routine to take place, causing the number of cycles used to overflow the limited amount for each frame, thus slowing down the computer.

The best way to avoid this problem is to use opcodes that take up less cycles than the ones currently used. There are many C64 programming manuals and internet links available, which display a table of the MOS Technology 6502/6510 opcodes, including the number of cycles each opcode uses. My advice is to read and thoroughly understand which opcodes use less cycles, as this will be a very handy technique to write faster code.

Now, let's take a closer look at the example code above. Logically, the code is executed 39 (or $27) times in a loop. For this loop to work, the opcodes INX, CPX, and BNE are used.

According to the opcodes table in many programming manuals, INX uses 2 cycles, CPX uses 2 cycles, and BNE uses 2 cycles (or sometimes 3 cycles if the branch goes over/under the page boundary.) These three opcodes are executed 39 times in the loop, giving the result of:

39 * (2 + 2 + 2) = 234 cycles.

This only goes for those three opcodes in the loop. We still need to calculate the number of cycles for the rest of the routine. Therefore:

lda $0000,x is equal up to 5 cycles (or 4 if it does not go over page boundary, but we will stick to 5 for the worst case)

sta $0000,x is also equal up to 5 cycles. (as above)

Because this is a 3×3 scroller, these two opcodes are theoretically called 3 times, giving us a total of (5 + 5) * 3 = 30 cycles. But because of the loop:

((5 + 5) * 3) * 39 = 1170 cycles.

Now add this with the previous calculation we did: 1170 + 234 = 1404 cycles in TOTAL (plus however many cycles used for the rest of the code)

Judging from all this, it is quite inefficient for speed. But not to worry, there is a way around this! First of all, do we **really** need to put the routine in a loop? What if we just repeatedly write the same code 39 times? This obviously sounds inefficient for memory use, but this article isn't about memory use, it is about speeding things up, so this is the only way possible.

So, instead we can take out that useless loop feature and then write the code like this:

```
       lda $0401,x
       sta $0400,x
       lda $0429,x
       sta $0428,x
       lda $0451,x
       sta $0450,x
       ^^
       ^^
       (repeat this code 39 times!)

       (your scroll text code here)
       rts
```

The syntax is correct, however logically it won't work. It will however use lesser cycles. Basically we just subtract 234 from 1404 cycles now that the loop code is gone, leaving us with 1170 cycles. This isn't too bad at all (at least it is theoretically faster) but we can still optimise the code furthur, by using better opcodes….

Remember that LDA $0000,x and STA $0000,x are both equal to up to 5 cycles. We CAN use however:

LDA $0000 (equal to 4 cycles) STA $0000 (equal to 4 cycles)

So now we can write the code like this:

```
       lda $0401
       sta $0400
       lda $0429
       sta $0428
```

```
        lda $0451
        sta $0450
        ^^
        lda $0402
        sta $0401
        lda $042A
        sta $0429
        lda $0452
        sta $0451
        ^^
        (...and so on after 39 times)

        (your scroll text code here)

        rts
```

This version is correct in both syntax and logical, and uses the following number of cycles:

(4 + 4 * 3) * 39 = **936** cycles!

This is a result of 468 cycle difference between the original version and the optimised version, which in theory is way faster on your machine.

Avoid repetitive use of Sub-routines

There is always one popular advantage and disadvantage about using the JSR and RTS opcodes. The advantage is of course, allows you to shorten your code and that it can be called as many times as you want. The disadvantage is that these two opcodes use 6 cycles each. So every time you call a working sub-routine, you are wasting 12 cycles. What if you need to call a subroutine around twenty times for your effect/whatever to look nice every frame? You won't realise how many cycles you are wasting.

The best advice is to try and avoid them in speed-critical code. Sometimes however it is quite difficult to avoid and that you must make way for use of them in some places - for example, playing music.

Remember - think before you do.

Shrinking down optimized code for better crunching performance

Now that we have our nicely optimized and faster piece of code, there is still one slight problem. Crunching this code tends to output a large file to decrunch, due to the repetitive code (unless you use a very good cruncher such as Exomizer or PUCrunch.)

There are however some very nice tricks in shortening down your code for crunching. One time I was looking at some code from the demo “Royal Arte” by HCL/Booze Design. In the bonus part (Starion into remake), I noticed a routine that basically creates duplicates of the same code at so many times.

Looking at this I thought, that is a NEAT trick! So therefore we will look at a good example of how you can do this yourself:

```
          *=$1000

          lda $d012      ;Compare raster
          cmp $d012      ;
          bne *-3        ;
          lda #$02       ;Do some random effects...
          sta $d020      ;
          sta $d021      ;
          lda #$03       ;
          sta $d021
          lda #$04       ;
          sta $d021
          lda #$05
          sta $d021
          lda #$06
          sta $d021
          sta $d020

          [ NOTE: a copy of the code above is repeated 64 ($80) times from
the start location ]
```

Looking at the example, this section of code above is stored 64 times in memory. As far as I know, this uses up $1380 (or 4992) bytes. This doesn't sound too bad for run-time, but crunching the code may leave your object file quite large.

What you **could** write is a routine which copies and pastes the same code every time...

```
          ;This code is used in set-up before you actually start the IRQ
          *=$0f00

          ;Store the start memory location in zeropage pointers
          lda #<CodeSource    ;Low byte memory
          sta $02
          lda #>CodeSource    ;High byte memory
          sta $03
          ldx #64             ;Number of times to copy and clone the main
code (64)

loop
          ldy #$00            ;Always set Y at zero
          lda CodeSource,Y    ;Read hex number for CodeSource memory +
offset Y
          sta ($02),y

          iny
```

```
          ;compare
          cpy #$27            ;Number of bytes the raw code uses.
          bne loop

          tya                 ;Add offset
          clc
          adc $02             ;Sets carry, if: (value in $02 >= (256 % 256))
          sta $02
          lda $03
          adc #$00            ;Becomes #$01 is there is a carry from before.
          sta $03
          dex
          bne loop

          [....jump to irq from this point forward...]

;This is the raw code to be copied and pasted each time.
          *=$1000

          lda $d012
          cmp $d012
          beq *+2
          lda #$02
          sta $d020
          sta $d021
          lda #$03
          sta $d021
          lda #$04
          sta $d021
          lda #$05
          sta $d021
          lda #$06
          sta $d021
          sta $d020
```

Let's go through this useful routine step by step.

First of all, you need to know how many bytes the main raw code uses. Simply do this by entering the raw code into $1000 (using an MC-monitor) and then count how many bytes it uses via the the memory position. As tested, the end address is $1027. Therefore subtract that with the start position ($1000) and that will give us $27 bytes - the **length** of the source code. You store this value in where it says **compare** in the example code - cpy #$<>

Next, we need to store the number of times cloning in the X register - ldx #$<>. When the routine executes, the X register is decremented each time.

Zeropage addresses $02 and $03 are used to carry the memory pointer of where the code is written.

- It starts off with the start position of the code $1000 - or $03 = $10, $02 = $00.

- Each time in the loop, these addresses are added by the size of the raw code - $27. In other words:

[Value in $02] = [Value in $02] + #$27

- If there is a carry [Value in $02 goes over #$ff and back to #$00] then $03 is incremented. This is a nice trick in increasing the memory pointer. There are other ways, but this would be the most efficient.

Now that all of this is coded in, the length of the whole code will be a sum of:

Length of the clone routine [$0f00-$0f26] + Raw code [$1000-$1027]

(Simplified:)

$26 + $27 = $4d bytes!

This is a difference of: $1380 - **$4d** equal to $1333 bytes less!

Now that the whole code has shrunk down, this will make the output crunched file and smaller size. Remember that this can only be recommended to execute at set-up. Do not try to execute this in run-time, otherwise you will get slow performance again.

Thanks for reading! ... ... Article by Conrad (1-7/06/2007)