Round 7 • Char by char text scroller

OCA / Atlantis

// Introduction

In this episode we have a look at a very basic 'char by char' text scroller implementation. This is going to be the final boss for Part I.

// Char by char text scroller

Not sure if you read the previous articles, and whether you gave a try at writing this challenge already.

Maybe that's already something very easy for you, maybe not.

As a beginner, you may be confused, and don't really know where to start. In this article I will do my best to explain how I'm approaching things for when I don't have a clear picture about how to do stuff.

// Getting started.

I'm always starting my new ideas from there, an empty shell...

```
BasicUpstart2(main)
main:
  inc $d021
  jmp *
```

At least this helps me to get the ball rolling.

Then I'm trying to defer what I don't know how to do and to focus on describing what to do first, before trying to actually do the stuff.

// Main parts for a text scroller

First need a text to scroll...

Ok, that one is easy... Let's put this directly into our empty shell.

```
BasicUpstart2(main)
main:
  inc $d021
  jmp *

scroll_text:
  .text "test 123"
  .byte 0 // there is no len(scroll_text) or
          // scroll_text.size(), so adding a byte with
          // value 0 or $ff is a common way to indicate
          // the end of the text.
```

If you run this prg, you won't see the text yet... it's just there in memory, waiting to be used.

There in memory? but where exactly? Let's tell KickAssembler to indicate where this text is in memory, like this

```
BasicUpstart2(main)
main:
  inc $d021
  jmp *

.pc = * "SCROLL TEXT DATA"
scroll_text:
  .text "test 123"
  .byte 0 // there is no len(scroll_text) or
          // scroll_text.size(), so adding a byte with
          // value 0 or $ff is a common way to indicate
          // the end of the text.
```

> **Side note:** .pc = *MEMORY_LOCATION*
>
> tells kickassembler that the following section should be placed at address *MEMORY_LOCATION*.
>
> * is a way to tell kick assembler to use the current address. but we could also force a specific address this way
>
> `.pc = $4000`
>
> Additionally, you can use a description like "DATA SECTION" to bring clarity.
>
> `.pc = $4000 "DATA SECTION"`

When compiling your prg with argument -showmem, and assuming your source file is named scroll-text.asm, you should get the exact memory location (or even memory range) for "SCROLL TEXT DATA".

```
 java -jar KickAss.jar -showmem scroll-text.asm

 ...

 Memory Map
----------
Default-segment:
  $0801-$080c Basic
  $080e-$0813 Basic End
  $0814-$081c SCROLL TEXT DATA
```

But we are getting a bit off topic — let's refocus on our next steps for the scroller.

The anatomy of a text-scroller

There are a few things that we need to do all the time when doing a scroller.

- We usually start with a whole blank line
- The whole line is shifted to the left,
- Then there is a new letter appearing on the right.
- This continues like this until the end of text is reached (remember that .byte 0 we put at the end of the text?)...
- then you have to decide what to do... restart from the first letter, continue filling with space until the whole text is gone on the left side, or whatever.

As a warmup, let's get started with something easy, putting the first letter of our text on the right side.

Putting the first letter of the text on the right side

The right side means column 39, but we should decide on which row we are going to place our scroller. To keep things flexible, let's define the line number as a constant.

```
.const SCREEN = $0400
.const LINE = 12
.const LINE_ADR = SCREEN + LINE * 40
BasicUpstart2(main)
main:

  lda scroll_text

  // + 39 for last column
  // of that line on the right
  sta LINE_ADR + 39

  jmp *

.pc = * "SCROLL TEXT DATA"
scroll_text:
  .text "test 123"
  .byte 0  // there is no
  // len(scroll_text)
  // or scroll_text.size(),
  // so adding a byte with
  // value 0 or $ff is a
  // common way to indicate
  // the end of the text.
```

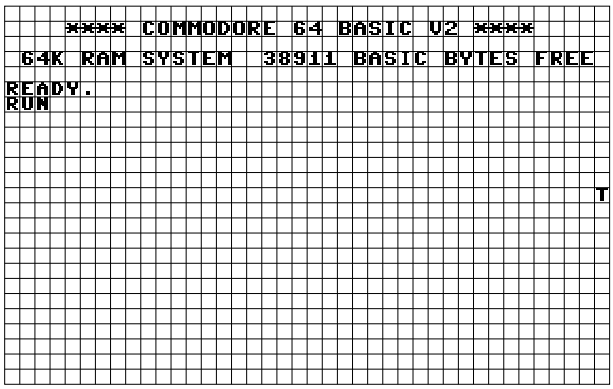


Figure 1: expected output for LINE=12, column=39

Side note: lda scroll_text? but scroll_text is the whole text right?, not a single letter?

Not really... scroll_text is a label, so just an address at which the letters for our whole text are stored as a sequence of bytes from address $0817 to $081f as we saw earlier. 'lda scroll_text' is the same as 'lda $0817' and $0817 contains the first letter $14, the screen code for letter 't'. It is rendered as 'T' as the default font for the c64 is uppercase.

(Earlier, with the shorter "empty shell" program, this same text sat at $0814. Here it's at $0817 because this warmup's `main` is a few bytes longer — `lda/sta/jmp` instead of just `inc/jmp` — which pushes everything after it a bit further down in memory.)

The Vice emulator contains a monitor from which we can inspect the memory, using 'm start end'

```
(C:$08a7) m $0817 $081f
>C:0817  14 05 13 14  20 31 32 33  00                        test 123@
(C:$0820)
```

Ok great, we can display the first letter on the right. But that letter is going to be the second letter next time... and then the 3rd... 'lda scroll_text' only works for the first iteration. That's for sure something we will have to revisit later.

But first, let's try to do things in order, and start with our 1st bullet point "The whole line is shifted to the left".

Shifting the whole line to the left

Before we write any code, let's visualize what "shifting a line to the left" actually means in C64 memory.

To shift a row of 40 characters left by 1 position:

- Column 0 gets the value of Column 1
- Column 1 gets the value of Column 2
- Column 2 gets the value of Column 3
- ...
- Column 38 gets the value of Column 39 (where the 'T' is for now)

In other words, for every character position X from 0 to 38, we copy the value from position $X + 1$.

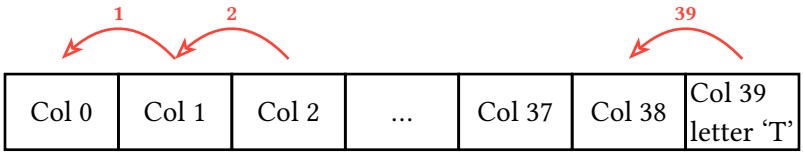


Figure 2: Copying character at column X+1 into column X

Side note: Pay attention to the red numbers above the arrows—these indicate the exact sequence in which we must copy the characters.

A very common beginner mistake is reversing this order: starting at column 39, copying its character into column 38, then copying column 38 into column 37, and so on. If you do that, you'll immediately overwrite column 38 with the 'T' from column 39, then copy that same 'T' into column 37, effectively filling the entire row with 'T's in a single frame!

By moving left-to-right (copying column 1 to 0, then 2 to 1), we safely shift the screen content without overwriting data before we've had a chance to move it.

We can easily achieve this loop using the x index register:

```
shift_line:
    ldx #0
shift_loop:
    lda LINE_ADR + 1, x   // Read char from Col X+1
    sta LINE_ADR, x       // Store char into Col X
    inx                   // Move to next column
    cpx #39               // Have we done 39 columns (0 to 38)?
    bne shift_loop        // If not, keep looping
    rts
```

Or we could use Kick Assembler's .for directive and do something called *unrolled* code (or *speed* code):

```
shift_line_unrolled:
  .for(var i=0; i < 39; i++ ) {
    lda LINE_ADR +i +1
    sta LINE_ADR +i
  }
  rts
```

The .for loop is performed at compile time, not at runtime. That means the compiler is going to produce instructions like this directly in your final program:

```
shift_line_unrolled:
    lda LINE_ADR + 1 // as i starts with 0
    sta LINE_ADR + 0
    lda LINE_ADR + 2 // i = 1
    sta LINE_ADR + 1
    lda LINE_ADR + 3 // i = 2
    sta LINE_ADR + 2
    ...
    lda LINE_ADR + 39 // i = 38
    sta LINE_ADR + 38
    rts
```

Unrolled code is really fast because there is no need for loop bookkeeping like inx, cpx, or bne. The trade-off is the extra memory it takes up in your program file.

Let's now see what's the next step: "Then there is a new letter appearing on the right."

Putting a letter on the right side

We already did this during our warmup:

```
...
  lda scroll_text

  // + 39 for last column
  // of that line on the right
  sta LINE_ADR + 39
  ...
```

“sta LINE_ADR + 39” is fine, the next char should always be there. This is “lda scroll_text” that we should adapt.

It’s tempting to use indexed addressing here too. In pseudo-code:

```
  ldx <index of the next letter>
  lda scroll_text, x
```

This works great! We can store our position in memory using a simple variable byte (e.g., text_index: .byte 0) and load it into the X register every frame.

But that means our text would have a maximum of 255 letters (+ .byte 0 as terminating char). How are you going to fit your full greetings list with only 255 chars ;)

The proper way is to use indirect addressing, using a pointer, like we already did in the previous article. As a reminder, that would be something like this.

```
.pc = $02 "Zero Page" virtual
  text_ptr: .word 0

BasicUpstart2(main)
main:
  lda #<scroll_text
  sta text_ptr
  lda #>scroll_text
  sta text_ptr+1

  ldy #0
  lda (text_ptr), y

  ...

  scroll_text:
    .text "test 123"
    .byte 0
```

> **Side note:** Wait… that’s the same limitation… y is also a value from 0 to 255!
>
> Yes, but we are not going to use ‘y’ here, we will increment text_ptr directly instead and keep y to 0.

Putting together everything we've seen so far

Ok there are still some bullet points that are not addressed yet, but let's already put what we saw so far together. We should be able to do this:

- We usually start with a whole blank line
- The whole line is shifted to the left,
- Then there is a new letter appearing on the right.
- This continues like this until the end of text is reached

Let's maybe use subroutines to keep the flow easy to read. The core of our scrolling routine is actually going to be just this.

```
do_scroll:
    jsr shift_line_left
    jsr put_letter_on_right_side
    jsr next_letter
    jsr reset_scroll_if_needed
    rts
```

Then let's add a delay subroutine to make it a bit slower and here is the whole code:

```
.const SCREEN = $0400
.const LINE = 12
.const LINE_ADR = SCREEN + LINE * 40
.pc = $02 "Zero Page" virtual
text_ptr: .word 0


BasicUpstart2(main)
main:
    jsr init_scroll
    jsr clear_scroll_line
loop:
    jsr delay
    jsr do_scroll
    jmp loop

init_scroll:
    lda #<scroll_text
    sta text_ptr
    lda #>scroll_text
    sta text_ptr+1
    rts

clear_scroll_line:
    ldx #39
    lda #' '
    !:
    sta LINE_ADR,x
    dex
    bpl !-
    rts

delay:
    ldy #8
    !:
    jsr wait
    dey
    bne !-
    rts

wait:
    bit $d011
    bpl *-3
    bit $d011
    bmi *-3
    rts
```

```
do_scroll:
    jsr shift_line_left
    jsr put_letter_on_right_side
    jsr next_letter
    jsr reset_scroll_if_needed
    rts

shift_line_left:
    ldy #0
    !:
    lda LINE_ADR + 1,y
    sta LINE_ADR,y
    iny
    cpy #39
    bne !-
    rts

put_letter_on_right_side:
    ldy #0
    lda (text_ptr),y
    sta LINE_ADR + 39
    rts

next_letter:
    inc text_ptr
    bne !+
    inc text_ptr+1
    !:
    rts

reset_scroll_if_needed:
    ldy #0
    lda (text_ptr),y
    bne !+
    jsr init_scroll
    !:
    rts

.pc = * "SCROLL TEXT DATA"
scroll_text:
  .text "test 123 - "
  .byte 0
```

Listing 1: 01_text-scroll-char-by-char.asm

// Building blocks we skipped in Part I

Part I gave you a working toolbox, but it isn't the *complete* 6502. A few general-purpose techniques didn't make the cut — not because they don't matter, but because you can get a long way into Part II without them. Here's what we left out, so you know the gaps are deliberate and where to look if you want to fill them yourself.

- **Bit manipulation with `and`, `ora` and `eor`.** We shifted and rotated bits in Round 2 and tested one with `bit` in Round 3, but never used *masks* to set, clear or toggle individual bits while leaving the others alone. We'll pick this up in Part II, because the hardware registers often need exactly that: setting one sprite's bit in $D010 without touching the others, or flipping a single flag in $D011 or $D016 without disturbing the rest.
- **The stack, and saving registers with `pha` / `pla`.** `jsr` and `rts` have quietly used the stack since Round 1, but we never manipulated it ourselves. You'll meet this again the first time we touch interrupts, since an interrupt has to preserve `A`, `X` and `Y` — we'll cover exactly what's needed, when it's needed.
- **Signed numbers and the `V` flag.** Round 6 stayed strictly *unsigned* (`bcc` / `bcs`), which is all screen positions and counters ever need. Signed arithmetic, two's complement and the `bvc` / `bvs` branches are a whole story of their own that we simply didn't require.
- **Two addressing modes: `lda (ptr,x)` and `jmp (addr)`.** We used indirect-indexed `(ptr),y` heavily, but not its rarer sibling `(ptr,x)`, nor the indirect jump `jmp (addr)` that turns a table into a jump table. Useful tools, but not ones we leaned on here.

// Recap of the mnemonics we used so far

Mnemonic	Description
`lda`	Load Accumulator — Loads a value into register A.
`ldx`	Load X Register — Loads a value into register X.
`ldy`	Load Y Register — Loads a value into register Y.
`sta`	Store Accumulator — Writes the value from register A into memory.
`jmp`	Jump — Sets the Program Counter (PC) to jump directly to a target address.
`jsr`	Jump to Subroutine — Pushes the return address onto the stack and jumps to a subroutine.
`rts`	Return from Subroutine — Pulls the return address from the stack and resumes execution after the `jsr`.
`clc`	Clear Carry Flag — Clears the Carry flag ($C = 0$), required before 8-bit or 16-bit addition (`adc`).
`adc`	Add with Carry — Adds a value and the Carry flag to register A ($A = A + M + C$).
`bcc`	Branch if Carry Clear — Jumps to a target address if the Carry flag is clear ($C = 0$), e.g. when an addition produced no carry out of the byte (the result stayed ≤ 255).
`bne`	Branch if Not Equal — Jumps to a relative target address if the Zero flag is clear ($Z = 0$).
`dex`	Decrement X Register — Decreases register X by 1 ($X = X - 1$).
`dey`	Decrement Y Register — Decreases register Y by 1 ($Y = Y - 1$).
`inc`	Increment Memory — Increases a value in memory by 1 ($M = M + 1$).
`asl`	Arithmetic Shift Left. Shifts all bits in the accumulator or memory left by 1. Bit 7 is shifted into the Carry flag (C), and Bit 0 is filled with 0.
`lsr`	Logical Shift Right. Shifts all bits in the accumulator or memory right by 1. Bit 0 is shifted into the Carry flag (C), and Bit 7 is filled with 0.
`rol`	Rotate Left. Rotates all bits in the accumulator or memory left by 1 through the Carry flag. Bit 7 moves into Carry, and the old Carry value moves into Bit 0.
`ror`	Rotate Right. Rotates all bits in the accumulator or memory right by 1 through the Carry flag. Bit 0 moves into Carry, and the old Carry value moves into Bit 7.
`tya`	Transfer Y to Accumulator — Copies the value from register Y into register A ($A = Y$). Sets the Zero flag ($Z = 1$) if the value is 0 and the Negative flag ($N = 1$) if bit 7 is set.
`inx`	Increment X Register — Increases register X by 1 ($X = X + 1$).
`iny`	Increment Y Register — Increases register Y by 1 ($Y = Y + 1$).
`bit`	Bit Test. Performs a bitwise AND between the memory contents and the Accumulator without altering either register. Sets the Zero flag (Z) based on the AND result, and copies memory bit 7 into the Negative flag (N) and memory bit 6 into the Overflow flag (V).
`sec`	Set Carry Flag — Sets the Carry flag ($C = 1$), required before 8-bit or 16-bit subtraction (`sbc`).
`sbc`	Subtract with Carry — Subtracts a value and the borrow from register A ($A = A - M - (1 - C)$). Set the Carry with `sec` first so the initial borrow is 0.
`eor`	Exclusive OR — Performs a bitwise XOR between register A and memory ($A = A \oplus M$), storing the result in A. Sets the Zero flag ($Z = 1$) if the result is 0 and the Negative flag ($N = 1$) if bit 7 is set.
`cpx`	Compare X Register — Compares the value in register X with memory by performing an internal subtraction ($X - M$). Sets the Carry flag ($C = 1$) if $X \geq M$, the Zero flag ($Z = 1$) if $X = M$, and the Negative flag ($N = 1$) if bit 7 of the difference is set.
`cpy`	Compare Y Register — Compares the value in register Y with memory by performing an internal subtraction ($Y - M$). Sets the Carry flag ($C = 1$) if $Y \geq M$, the Zero flag ($Z = 1$) if $Y = M$, and the Negative flag ($N = 1$) if bit 7 of the difference is set.
`bcs`	Branch if Carry Set — Jumps to a relative target address if the Carry flag is set ($C = 1$), e.g. after a comparison (`cmp`) when the register is greater than or equal to memory.

// Comments?

Something is wrong or missing? Feel free to send me a PM (OCA/Atlantis) on CSDb.

// What's coming next

- **Congratulations!** You have completed Part I – Building Blocks.
- Part II will focus on the specific hardware registers of the C64 ($D000–$DFFF).
- We will kick off by covering the overall memory layout of the Commodore 64, along with a few common beginner traps to avoid.
- Later in Part II, we will upgrade our rough char-by-char scroller into a smooth, pixel-by-pixel text scroller using the $D016 control register.

Want to give it an early try? Just update the current code with this

```
.const SCREEN = $0400
.const LINE = 12
.const LINE_ADR = SCREEN + LINE * 40
.pc = $02 "Zero Page" virtual
text_ptr: .word 0

BasicUpstart2(main)
main:
    jsr init_scroll
    jsr clear_scroll_line
    ldx #0
loop:
    jsr wait
    jsr do_d016_scroll
    jmp loop

do_d016_scroll:
    dex
    bpl !+
    ldx #7
    jsr do_scroll // !!! do_scroll and its subroutines must not change x
    !:
    stx $d016
    rts

init_scroll:
  ... same code from here ...
```

Listing 2: 02_broken-smooth-text-scroll.asm

That's far from being optimal… The scroll line is now scrolling smoothly, 1 px at a time, but that's the only thing looking good. The whole screen shakes now… Would you be able to fix that? We are going to tackle this issue (and quite a few other hidden issues) later in Part II when talking in more detail about the $D016 register.