Round 1 • Indirect Addressing

OCA / Atlantis

// Introduction

Having fun programming the C64 in assembly — a series of articles for beginners who already have some basic knowledge of assembly and would like to go a bit further.

The whole series is articulated in three main parts:

- **Part I — Building Blocks.** General-purpose techniques that aren't really specific to the C64: addressing modes, lookup tables, self-modifying code, and the like. We stay away from interrupts and use only a handful of the $d000–$dfff registers.
- **Part II — I/O ($D000 - $DFFF Registers).** Where we get properly C64-specific and start building visual effects: switching banks, bitmap and text modes, interrupts, sprites, plasma, raster bars, scrolling, starfield, SID and friends.
- **Part III — Trickery.** As you probably know, there are quite a few tricks we can do with the C64… we will explore a few of them.

Each part consists of a few "Rounds", covering a well-scoped topic. Each Round ends with a challenge; the next Round opens with the answer.

Ideally you should already be comfortable using some basic opcodes like `lda` or `sta`, and be familiar with hexadecimal. Something like this should already make sense to you.

```
lda #1
sta $d020  // turn the border color to white
```

// A quick word about compiling and running your code

In this series of articles we are going to use Kick Assembler as our main tool.

To use Kick Assembler, you need Java installed on your computer, and you need to download the KickAss "jar" file.

This is the command to compile a .prg from its assembly source file.

```
java -jar KickAss.jar your-file.asm
```

That should produce a .prg file that you can drag and drop directly into the Vice C64 emulator, for instance.

// Putting a char on the screen

Let's get started and put an '@' char on row 2, col 3 (we start counting rows and cols from 0). Let's do that together.

This is very similar to changing the border color: let's use "lda" and "sta" to alter the screen memory.

```
lda #'@'  // lda #0 would do the same
sta <we need to know which address to put here>
```

So let's have a look at Figure 1, the default screen memory layout.

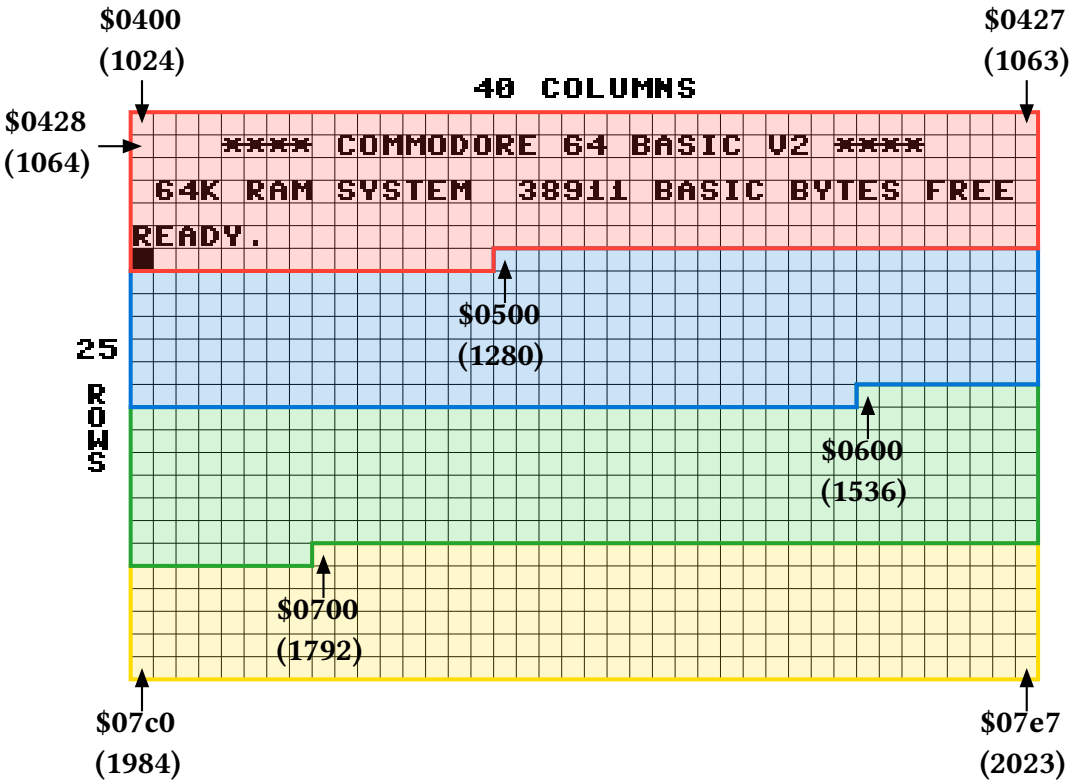
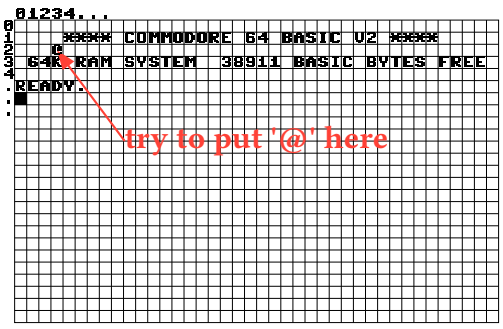


Figure 1: The default c64 screen

We can see that the default screen starts at $0400 (or 1024 in decimal) and goes up to $07e7, which is the bottom-right corner.

The screen consists of 40 x 25 chars (1000 chars).

Given the '@' char should be at row = 2, col = 3, the address to use can be computed like this.



$$
\begin{aligned}
\text{address} &= \text{BASE_SCREEN_ADR} + \text{ROW} \times \text{WIDTH} + \text{COL} \\
&= 1024 + 2 \times 40 + 3 \\
&= 1107 \\
&= \$0453 \quad \text{(in hexadecimal)}
\end{aligned}
$$

The whole code should then look like this.

```
BasicUpstart2(main)
main:
    lda #'@'
    sta $0453
forever:
    jmp forever
```

Listing 1: 01_at_char.asm

> **What is this BasicUpstart2(main) for?**
>
> This line is a call to a Kick Assembler macro that generates the SYS call you see from the C64 BASIC screen when typing LIST, in our case "10 SYS2062"
>
> This is the execution address for your program. 2062 ($080e) is a default address for the routine "main". this is something you can change using the .pc directive, like this:
>
> ```
> BasicUpstart2(main)
> .pc = $2000  // or  * = $2000
> main:
> ...
> ```
>
> In that case, the SYS BASIC line that is going to be generated by Kick Assembler is 10 SYS8192 (as 8192 = $2000)

I hope this is still very basic stuff for you, because we are going to complicate things a bit now with the challenge coming next...

// Challenge: Putting a char on the screen at any given position

Could you write the draw_char routine below? The goal is to build a routine that puts a char at a given position, expressed using the row and col bytes.

```
BasicUpstart2(main)
main:
    lda #'a'  // or lda #1
    sta char
    lda #10
    sta row
    lda #12
    sta col
    jsr draw_char

forever:
    jmp forever

draw_char:
   // The challenge is to write
   // this routine.
   rts

row:  .byte 0
col:  .byte 0
char: .byte 0
```

Listing 2: 02_challenge.asm

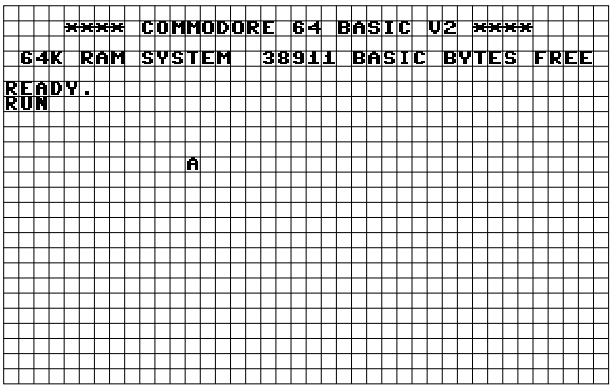


Figure 2: expected output for row=10 and col=12

The main issue is that we cannot write it this way

```
draw_char:
   lda char
   sta $0400 + row * 40 + col  // this is not going to compile
   rts
```

You may already be familiar with something like this, where x is an offset from an address

```
sta address,x
```

So we could try something like this

```
sta $0400,x
```

and then compute x to be “row * 40 + col”. Unfortunately, that’s not going to work… x is a byte, so its maximum value is 255, and we would need up to 1000 to place a char anywhere on the screen, as shown in Figure 3.

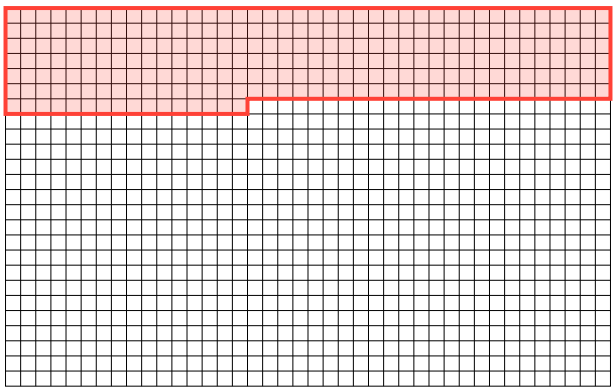


Figure 3: portion of the screen that could be addressed with sta $0400,x

In order to write a generic draw_char routine, one way is to use a pointer, with something like this

```
sta (pointer),y
```

This is called indirect addressing. Are you familiar with it already? If so, you are all set: you can give writing draw_char a try.

If not, you should take some time to get familiar with it, as this is a very important concept.

// Indirect addressing

When you do something like “sta $0400”, you write the value in A (where A stands for Accumulator) directly into memory at address $0400.

With indirect addressing, you first set a pointer to $0400, and then you write something at the address your pointer points to…

You cannot declare your pointer just anywhere; it must live in a specific address range called Zero-Page (ZP).

> **Side note:** Zero-Page (ZP) is the first page of memory ($0000 to $00FF, or 0 to 255 in decimal). Because the high byte is always $00, the CPU can refer to these addresses with a single address byte instead of two — so zero-page instructions are one byte shorter and a cycle faster than their full-address equivalents.

There are several ways of declaring a ZP pointer in Kick Assembler.

```
// we need two consecutive bytes to hold a 16-bit address
.label pointer_lo = $02
.label pointer_hi = $03 // should always be pointer_lo+1

BasicUpstart2(main)
main:

  // set the pointer to the desired address
  lda #<$0500 // or just lda #00 as the address ends with $xx00
  sta pointer_lo
  lda #>$0500 // or just lda #05 as the address starts with $05xx
  sta pointer_hi

  // use the pointer
  ldy #0  // no offset
  lda #01 // 'A'
  sta (pointer_lo),y   // write 'A' at address $0500

  jmp * // a short way of writing a forever loop
```

Listing 3: 03_pointer_label.asm

Here is another syntax that I slightly prefer myself.

```
.pc = $02 virtual // don't put 0 or 1 here
pointer: .word 0  // a word is 2 bytes long

BasicUpstart2(main)
main:

  // set the pointer to the desired address
  lda #<$0500
  sta pointer
  lda #>$0500
  sta pointer+1

  // use the pointer
  ldy #0  // no offset
  lda #01 // 'A'
  sta (pointer),y   // write 'A' at address $0500

  jmp * // a short way of writing a forever loop
```

Listing 4: 04_pointer_pc.asm

Why skip $00 and $01?

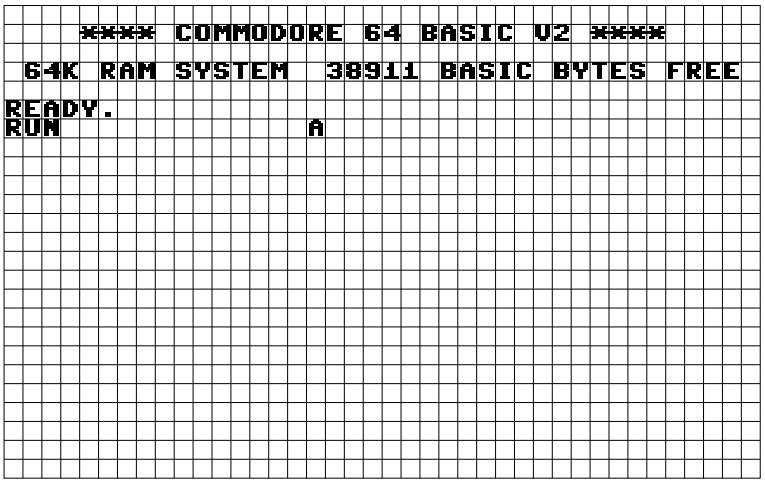
On the 6510 CPU, $0000 and $0001 aren't normal RAM.

- $00 (DDR): Data Direction Register — sets which port bits are inputs or outputs.
- $01 (processor port): its low 3 bits bank BASIC ROM, KERNAL ROM, and I/O in and out of view; the upper bits drive the Datasette.

> Because these two bytes don't behave like RAM, storing a value in them (a pointer, say) won't work as expected — and changing $01's banking bits can swap ROM or I/O out from under you, crashing the machine if you're not careful. Normal zero-page RAM begins at $02.

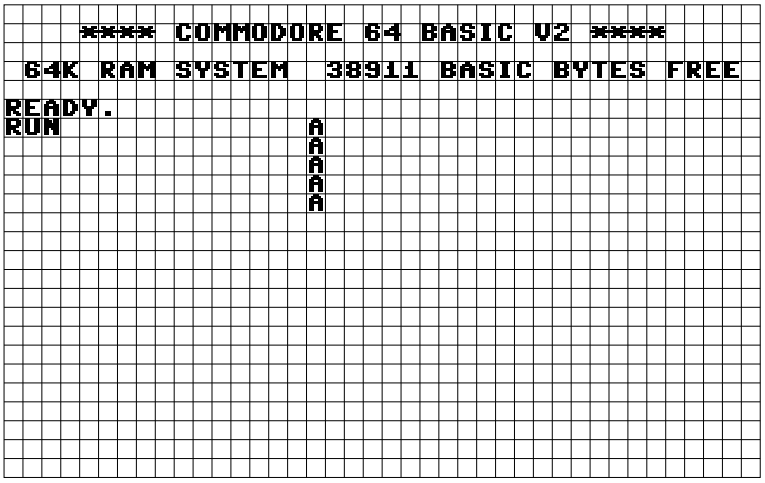
the "virtual" keyword on the first line is also very important: it tells Kick Assembler not to include this block in the .prg. Also note that the second line can be a bit misleading, as we might think the pointer is initialized to 0, which is not the case.

Both syntaxes are completely equivalent and produce identical output.



At this point you may feel a bit lost — all that for what, exactly?

The key thing is that you can now manipulate your pointer to address any screen char position. Here is an example of manipulating the pointer to draw a column of five 'A' chars.



```
.const LINE_WIDTH = 40
.pc = $02 virtual // don't put 0 or 1 here, that would fail!
pointer: .word 0  // a word is 2 bytes long

BasicUpstart2(main)
main:
  // set the pointer to the desired address
  lda #<$0500
  sta pointer
  lda #>$0500
  sta pointer+1

  ldy #0  // no offset
  ldx #05 // a loop to display 5 chars
  loop:
  lda #01 // 'A'
  sta (pointer),y

  lda pointer
  clc
  adc #LINE_WIDTH // adding 40 will point to next row, same col
  sta pointer
  bcc !+
  inc pointer+1  // our pointer is 16-bit
  !:
  dex
  bne loop

  jmp *
```

Listing 5: 05_pointer_adc.asm

That's probably already a lot if this is the first time you are playing with indirect addressing. I would advise stopping here for a while and playing a bit with this sample before going back to the draw_char challenge. For instance

- Do you understand 16-bit addition logic, what !+ and !: are, and why we use bcc here?
- What happens if you add an offset using y? (changing ldy #0 to another value)
- Could you draw a 5x5 square of 'A' instead of a vertical line of 'A'?

Try swapping x and y.

```
  ldx #0  // x is now used for the offset
  ldy #5  // and y is now used for looping
  loop:
  ...
  sta (pointer),x  // does that still work?
  ...
  dey
  bne loop
```

- Does that compile? (hint: yes, it should still compile fine)
- Do you understand why it is not working anymore?

// A visual representation of indirect addressing

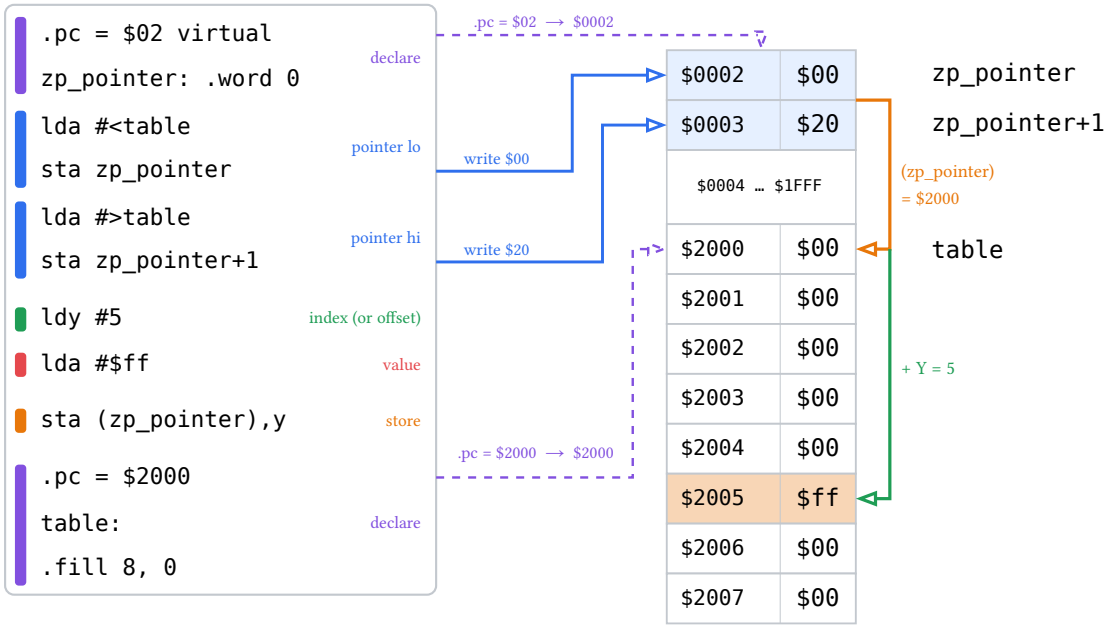


Figure 4: Indirect addressing

// 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 ($M = M + 1$).

> **Side note:** Mnemonic vs Opcode: *mnemonic* is a name like 'lda' or 'sta', while *opcode* is the value of the byte stored in memory like $a9 for 'lda'.
>
> Also don't worry if this doesn't make complete sense just yet — we will revisit branching (bne, bcc, bpl, etc.) and talk more about flags in a later article.

// Stuck?

Stuck somewhere in the challenge? Feel free to send me a PM (OCA/Atlantis) on CSDb. I'll be happy to help when I can.

// What's coming next

In the next article, we are going to explore a few implementations of draw_char.

- Doing the multiplication as a loop
- Using a proper multiplication (there is a trick for when we have to multiply by 40)
- Using a lookup table (LUT)