

## LEARNINGNERD

HOME > CATEGORIES: DAILY LEARNING NOTES

# Daily Learning Notes for January 28, 2016

I was so happy I got my first assembly program to work before I went to sleep yesterday! And today I finished the fourth NAND2Tetris project!

### Overview

Input/output handling program specs

How to turn pixels on and off

How to handle keyboard input

Experimenting with pixels and assembly

Questions

Learning summary (#TIL)

Next steps

Time breakdown

## Input/output handling program specs

The second program I need to write for this week's project will display or clear pixels on the screen based on keyboard input. Here's an outline of my objective:

- The program should run an infinite loop that listens for keyboard input.
- When any key is pressed, the screen should make a pixel black and repeat this until every pixel on the screen is black.

- When no key is pressed, the screen clears the screen and turns the pixels white, repeating this until the whole screen is white.
- There is test script but no compare file; I should confirm the program works by watching the simulated screen in the CPU emulator.
- I can choose to change the pixels on the screen in any order I want, as long as eventually the screen turns all black if keys are pressed for long enough or all white if no keys are pressed for long enough.

OK, that's pretty straightforward. At least I understand my objective! That's always a good start. But I don't know how to turn pixels on or off and I don't know how to check for keyboard input! They just glossed over that at the end of [chapter 4 \(PDF\)](#).

## How to turn pixels on and off

Digging back through the book again, I see that the Hack computer uses a *memory map* to associate devices with segments of the memory. To draw pixels on the screen, all I have to do is write a binary value (1 for black, 0 for white) to the right RAM addresses.

### Important screen specs:

- The screen has 256 rows, with 512 pixels per row. So its display resolution is 512 x 256 – 512 columns, 256 rows.
- The screen is linked to an 8K memory map, starting at RAM address 16384 (0x4000).
- “Each row in the physical screen, starting at the screen's top left corner, is represented in the RAM by 32 consecutive 16-bit words.”
- The symbol `SCREEN` is an I/O pointer to RAM address 16384 (0x4000), representing the top left pixel on the screen. So I can access that first pixel with `@SCREEN` and make it black by writing 1 to it with the command `M=1`. Cool!

- The formula for accessing a pixel at row  $r$  and column  $c$  is

`RAM[16384 + (r * 32) + c/16]`. (I don't understand why yet, but I'll look at it closely later.)

I tried running the example code to draw one black pixel on the screen, and sure enough, it worked! It's so tiny, I almost didn't see it in the CPU emulator, but it certainly worked! So that part was easy enough. I think the most challenging part of this project will be implementing that formula. Argh, math! I always get stuck on the math!

## How to handle keyboard input

I searched through the book for any mention of the keyboard, and I gathered a few very helpful details. In a way, working with the keyboard seems like it will be even more simple than working with the screen.

### Important keyboard specs:

- The Hack computer uses a single-word memory map for the keyboard – just 16 bits!
- The symbol `KBD` is an I/O pointer to RAM address 24576 (0x6000).
- When a key is pressed, its 16-bit ASCII code is stored in `RAM[24576]`.
- If no key is pressed, that location contains the value 0.

I realized that I can test the keyboard in the CPU emulator without writing any code! All I had to do was scroll down to the very last RAM location and press a key on my keyboard. A number showed up! The letter A has the ASCII code 65 and Z is 90. Cool stuff! I wonder what the story is behind ASCII codes: how long ago was it invented, how did it become the standard, who came up with it, what were its competitors?

But anyway, back to business. I have my conditional statement now: if `RAM[24576]` is zero, clear the screen. Otherwise, make the pixels black one at a time in a pattern that

would eventually cover the whole screen. But I don't know how to do that part!

## Experimenting with pixels and assembly

Just for fun (and to check my understanding), I wrote a little program that just turns the same pixel on or off depending on whether there's keyboard input:

```
// Set the loop's start location with a label
(LOOP)
// set the A register to point to the keyboard's memory map
@KBD
// load the keyboard value into D register
D=M
// Point to next jump location to clear the screen
@CLEAR
// If no key is pressed, jump to CLEAR
D;JEQ
// ELSE: set the A register to point to the first pixel
@SCREEN
// load 1 to turn on the top left pixel
M=1
// Skip over the CLEAR section
@END
// Unconditional jump
0;JMP
// Label to jump to if clearing the screen
(CLEAR)
@SCREEN
M=0
// Label for skipping the CLEAR section
(END)
// Jump back to the start: infinite loop!
@LOOP
// Unconditional jump
0;JMP
```

Woohoo, it works! That one pixels shows up if I press a button on my keyboard while the program is running, and it disappears when I let go. This is a good start!

But here's what I'm stuck on now: how do I increment the value of an address and then save that value to access the new address later?

Oh, my brain hurts again! I need to review what I actually understand in order to pinpoint exactly where it is that I'm getting confused:

- I know I can use `@100` followed by `0;JMP` to run whatever instruction is stored the instruction memory at address 100.
- I know I can also do `@100` followed by commands like `D=M+1` to perform operations on the *value* stored in the data memory at address 100.
- I know that if I did `@100` followed by `D=A+1` then the D register would hold the value 101, whereas `@100` followed by `D=M+1` would give me one plus the value *stored* in data memory location 100, not the number 100 itself. That's an important difference.

But how do I take a *variable* address, add one to it, and then access that new location? I need more caffeine or something, because I feel like I'm missing something *very* simple but I just can't see it!

Time for a break. I need to go to a working lunch meeting soon, anyway. That should help reset my brain, I hope.

**Three hours later:** Lunch, a good brainstorming session, a new freelance project, and a 30-minute walk – what a perfect way to break up my day! All I need is a little caffeine and I'll be good to go.

OK, so I watched through some of the video lecture, and then I realized what I was missing: I can save results to the A register, too! So I can do `@SCREEN` to get the base address for the screen, then `A=A+1` to iterate it, and then `M=1` to set the next pixel to black! At least I think that should work. Let's test that out...

Woohoo! It works! Not exactly as I hoped, though. It creates a row of pixels evenly spaced a certain width apart, not a continuous line. I left the program running for a long time and ended up with 32 vertical lines on the screen.

That number means something... 32 is 2 to the power of 5. My screen is 512 pixels wide. And 512 divided by 32 is 16. And 16 is the magic number! That's how many bits

each of my registers can hold. I've been setting every register to hold the value for 1, so that would explain why my pixels are 16 bits apart!

**Aha moment:** Every bit represents a pixel, so every register represents 16 pixels! So if I want to make the entire screen black, I need to store `1111111111111111` in each register instead of `0000000000000001`.

Let's test that out...

Yay, success! My screen is slowly turning completely black. It runs awfully slowly, but I love the fact that I can watch every register store the new value one at a time as the CPU emulator runs each line of assembly code.

I'm not sure if I'm supposed to turn the screen black *one pixel* at a time, or if it's OK if I turn the screen black 16 pixels at a time. I'm happy with it as it is, though. If I wanted to do it one pixel at a time, I would just have to change my program to add one to each register's existing value instead of setting every bit to 1.

OK, I'm going to finish implementing the final program. I won't write about it in detail so as not to give away the answer. Everything else I need to know to complete it is already written up in this blog post or it's spelled out in the book.

**A couple hours later:** Woohoo! It's working! I had to make a few adjustments to account for the limits of the screen's memory map and get my loops working properly, but one by one I fixed each bug and in the end, the process was actually very straightforward!

So that's it for week 4! Woohoo! Time to call it a night and eat some dinner.

## Questions

- What is a pointer?
  - **Answer:** A pointer is a variable that stores a memory address! Woohoo, this finally makes sense to me now!
- How long ago was 512 x 256 a common display resolution for computer screens?

- What's the story behind the invention and adoption of ASCII code?

## Learning summary (#TIL)

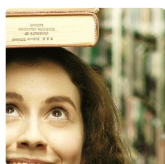
- I learned how to use input and output devices with the Hack computer.
- I wrote another working program in assembly language, completing the fourth NAND2Tetris project!

## Next steps

- Decide if I want to jump right into week 5 or take a break for a while. (After all, I want to start a NAND2Tetris study group and I'm toying with the idea of creating a series of videos about what I've learned so far.)

## Time breakdown

- Study time: 5 hours 10 min
  - Second assembly program: 3 hours 12 min
  - Watching video lectures: 1 hour 58 min



### Written by Liz Krane

Liz is a full-time nerd, sharing everything she learns as she tries to learn everything!

 Share on Twitter

 Share on Facebook

 Share on Google+

Updated January 28, 2016

**0 Comments**   **LearningNerd****1** **Login** ▾ **Recommend**    **Share****Sort**

Start the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 

Be the first to comment.

## ALSO ON LEARNINGNERD

**Learn Teach Code Vlog #1 (Unedited):  
Starting my own programming bootcamp**

1 comment • 7 months ago

**paper services** — Thanks for sharing this video blog Liz. Although it is very evident that you are anxious in the video. Keep it up.

**English Grammar: Basic Sentence  
Elements**

1 comment • 7 months ago

**case study help online** — Thanks posting these important terms of the basic sentence elements for English grammar. These are all important

**Weekly Word: Lucubrate • LearningNerd**




1 comment • 7 months ago

**Natalia Scherba** — Hi, Liz) Would you say "I lucubrate for my English"? I have doubts about the preposition. Thanks)

**English Parts of Speech: Adjectives,  
Determiners, and Adverbs**

1 comment • 7 months ago

**stevej101** — Nice tips!

 **Subscribe**    **Add Disqus to your site** **Add Disqus** **Add**    **Privacy**

Advertisement



