LESSON 2

Step 1: Set up the NMI

The NMI, or Non-Maskable Interrupt, can be a bit tricky to explain. Before we talk about it in terms of the NES, let's just talk about what Non-Maskable Interrupt means in terms of computer programming. An interrupt is exactly what it sounds like - something that interrupts the normal flow of a program. A non-maskable interrupt is an interruption to the code that is prioritized over the program and unable to be disabled. Forgetting the NES for a moment, pretend you're surfing the internet. In conjunction with your operating system, your web browser observes certain code in order to function. Now, for some reason, your computer starts to lag. You press CNTRL-ALT-DELETE.to bring up the task manager. This invokes an operating system back door that *interrupts* the normal functioning of the web browser. Nothing in the web browser's programming allows it to ignore this interruption. This sort of paints a picture of what a non-maskabe interrupt is.

On to how this concept relates to the NES - In simplest terms, when running your game, the part of the NES that draws to the screen (the PPU, or picture processing unit) is either preparing to draw the next frame or actively drawing the next frame. Our game logic is going on in parallel to this - you're game is reading inputs, doing mathematical operations, branching and comparing and setting up for the next frame. The PPU is rendering scanline by scanline, the whole way, evaluating what pixels it should draw. Then, the PPU finishes rendering the last pixel of the last scanline, and enters into what is called vBlank, short for Vertical Blank (or Vertical Blanking Interval).

For a mental image, the vBlank is the incredibly short amount of time between frames. This is when the light beam is traveling from the bottom of your screen back to the top of your screen in order to begin drawing the top scanline again. When it gets to the top of the screen, vBlank time is over and the next screen begins to render. This happens 60 times per second, giving the NES game 60 individually redrawn frames per second.

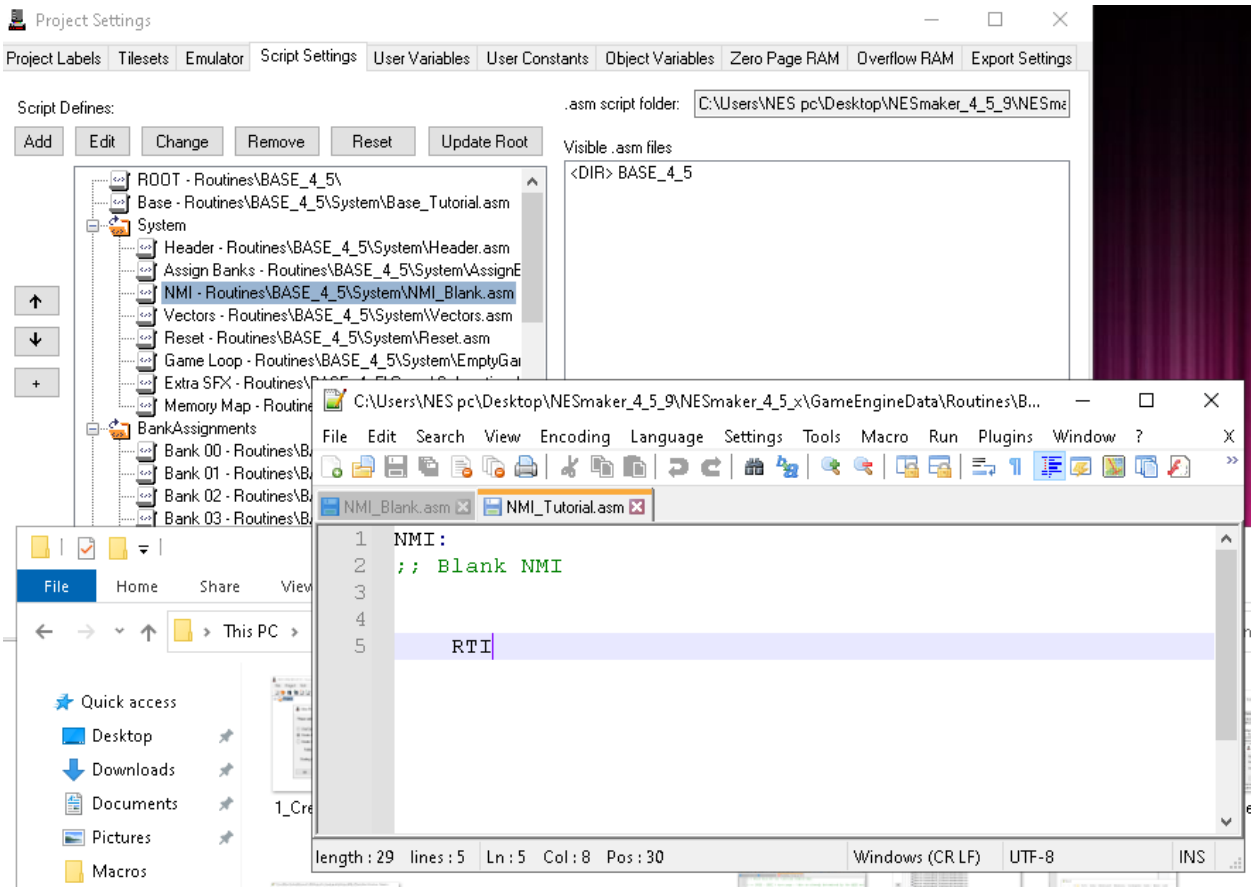
During this vBlank time, you need to send the PPU all of the updated instructions on what to draw where. Once it begins rendering again, you can't tell it to stop what it's doing to make a change, because it's just barreling through drawing line by line. You have to wait for that cool down period where it's not drawing lines to give it the new marching orders.

For the NES, the NMI (or Non-Maskable Interrupt) triggers at the beginning of each vBlank. Here is where you have to write all of the new drawing instructions to the PPU. If there are tiles that need to be updated, this is where the instruction needs to be given. If a sprite graphic has moved to a new position, this is where that new information needs to be conveyed. And because this is the portion of the game that deals with regulating your game's timing, it's usually advisable to update music and sound effects here too, so that you can be sure they stay consistent frame to frame.

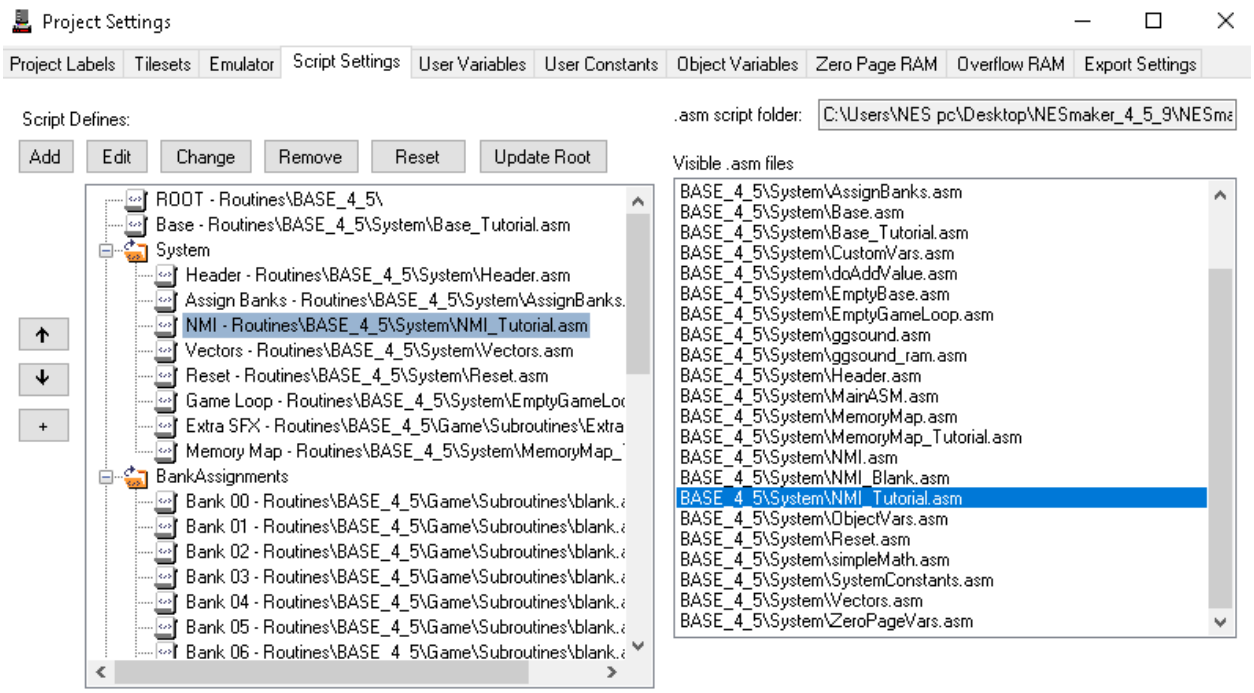
But as stated, the unfortunate news is that the time here is finite. There is only so much that we can accomplish during one vBlank period, and if that light bar gets back to the top of the screen before we've finished the vBlank, it can have unexpected, game crashing results.

We'll back up our temporary variables and registers in case we need to use them and corrupt their values during NMI functions. We'll create a simple NMI. We'll push all of the new frame's sprite data to the PPU, we'll set the scroll, we'll handle frame timing, and eventually we may want to come back here to regulate music timing. Lastly, we'll restore our temporary variables and registers to their state when we hit the vBlank.

Open Script Settings, click on the NMI Script Define, and click edit. In your script editor, save a copy of the NMI file as NMI_Tutorial.

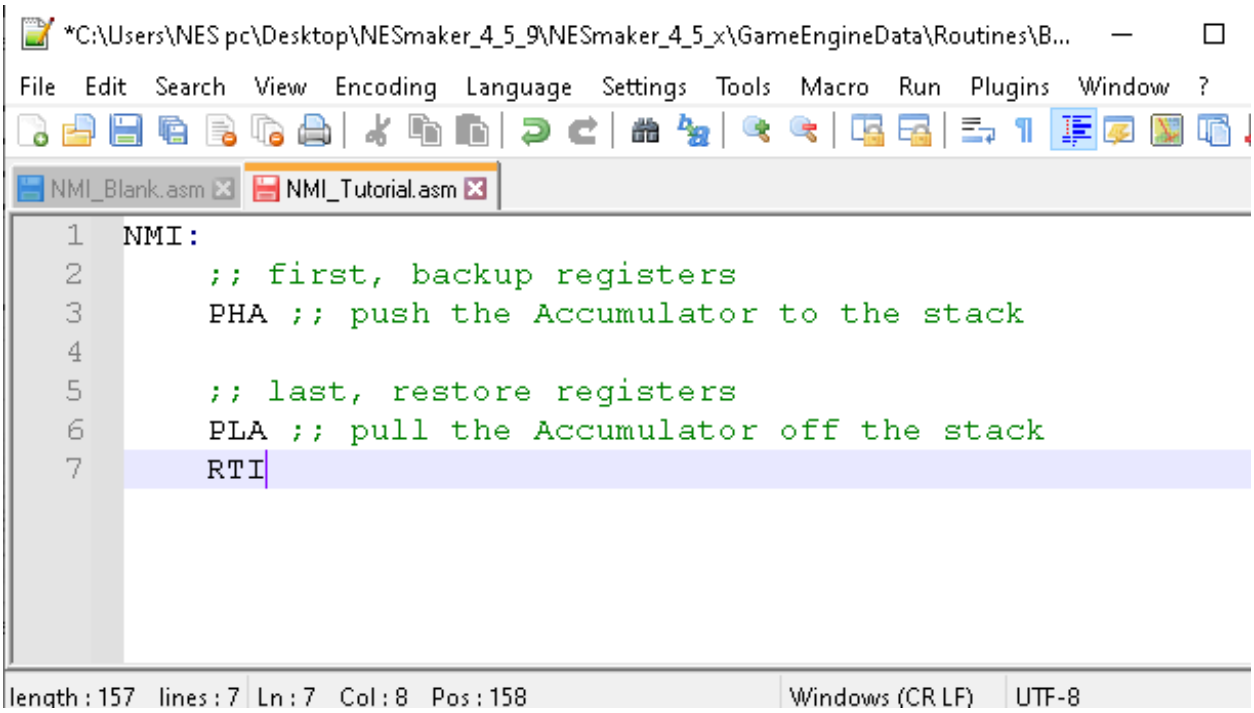


Step 2: Assign this new file as our NMI.
In the script settings, navigate the script finder to the System folder and attach our new NMI_Tutorial.asm to the NMI Script definition.



Step 3: Construct a simple NMI file between the NMI label and the RTI command, which is the 6502 ASM command that means Return From Interrupt.

The first thing that we're going to do is back up all of our registers, and then restore all of our registers. Whenever I have a piece of code that requires an open and close section, I like to do them at the same time and then fill in the middle part, often with a tab. This makes it very easy to spot check and make sure that everything that needs closure has, in fact, been closed.



The first thing that this NMI is doing is PHA, and the last is PLA. The command PHA pushes the accumulator (PH = Push, A = the accumulator) to the stack, and PLA pulls the value off the stack and puts it back into the accumulator (PL = Pull, A = accumulator). If you're unfamiliar with this sort of programming, that sentence is a lot of words and acronyms that don't mean much. So let's put it in easier terms. We'll start with the stack.

When programming for the NES, you have something called the stack. This is basically a scratch pad space in RAM that you can toss values to hold temporarily. Consider the stack a buddy who will hold your drink for you while you shoot a free throw. You had hands. They were busy drinking. But you needed your hands to shoot the free throw. So you say, "Hey, Stack good buddy, could you hold my beverage while I take this free throw?" He obliges, you use your hands to shoot the freethrow, and then he gives you back your frosty libation so that you can resume using your hands to enjoy the drink.

In the code above, as soon as the vBlank happens and the NMI consequently triggers, your game says, "Hey, Stack good buddy, hold on to this accumulator for me." Then, it will use the accumulator to do a bunch of other things. At the end of doing those things, your game says, "Thanks, Stack, can I get that accumulator back?", and when the game returns from

the vBlank and continues to the next frame, the accumulator is in the exact same state it was in as if the NMI never tripped. This means that if the code was in the middle of running logic that was making use of the accumulator when the NMI hit (which it almost certainly was), we put things back exactly the way they were upon returning to running the logic.

Hopefully that makes the function of the stack easy enough to understand, but what is the accumulator? The accumulator is a register that can hold one byte of data at a time, and it is used in pretty much every function and operation in your game. You load values to the accumulator, you use comparisons on the accumulator, you perform math on the accumulator, you send the value that is in the accumulator to memory addresses. Basically, the accumulator is the flux capacitor of 6502 ASM - it is “what makes ASM programming possible”.

To make it a bit clearer, let’s talk about some of the things you could do with the accumulator. We’ve already talked a bit about memory addresses. We know that a zero page memory address with a label can effectively act like a variable. Let’s say for a moment that we gave memory address $0007 a label of myPositionX, and planned to use it as a variable to keep track of the x position of our player. The mailbox is set up, but how do we send that mailbox a message? The accumulator is how. We would tell our program to load a value to the accumulator, and then store what is in the accumulator to that memory address. That would look like this (do not write this anywhere, it is just to explain the accumulator register):

```
LDA #$05 ;; loads the number 5 to the accumulator
STA myPositionX ;; stores the value in accumulator to myPositionX
```

This two lines of code would say Load (LD) into the Accumulator (A) the number 5, then Store (ST) the value in the Accumulator (A) to myPositionX. The practical effect of this is that the variable myPositionX would be set to 5.

Or if we wanted to check the value at myPositionX, we would need to do some sort of comparison. If we wanted to check to see if myPositionX is 8, we would write:

```
LDA myPositionX ; load the value in myPositionX to the accumulator
CMP #$08; is what is in the accumulator equal to 8?
```

Basically, just about everything that we're going to do in coding for the NES will involve the accumulator. And why that's important to understand is that our NMI will need to make use of the accumulator too. Let's pretend the last thing we did in our logic before the NMI suddenly hit was the LDA myPositionX command. Before we enter the accumulator, we loaded myPositionX. Let's say it is 8 at that moment. That means that next in the logic, we should get a positive "Yes! It is 8! Do the thing you want to do if it is 8!"

But then the NMI hit. And inside the NMI, we had to use the accumulator in our prepping of the screen graphics for the next frame to be drawn. At the end of doing so, some arbitrary number like 27 is loaded into the accumulator. Now we return to the logic, picking up right where it left off. Except now it will not return a positive, because the accumulator is no longer 8, it's 27, its value corrupted by the NMI.

This is why it's important to preserve this value so it's the same coming out as it was going in. We store the accumulator to the stack. We do all the NMI things, corrupting the accumulator as much as we need to, then at the end before jumping back to the logic, pull the stack value and put it back into the accumulator - as far as the logic is concerned, it was there the whole time and never changed.

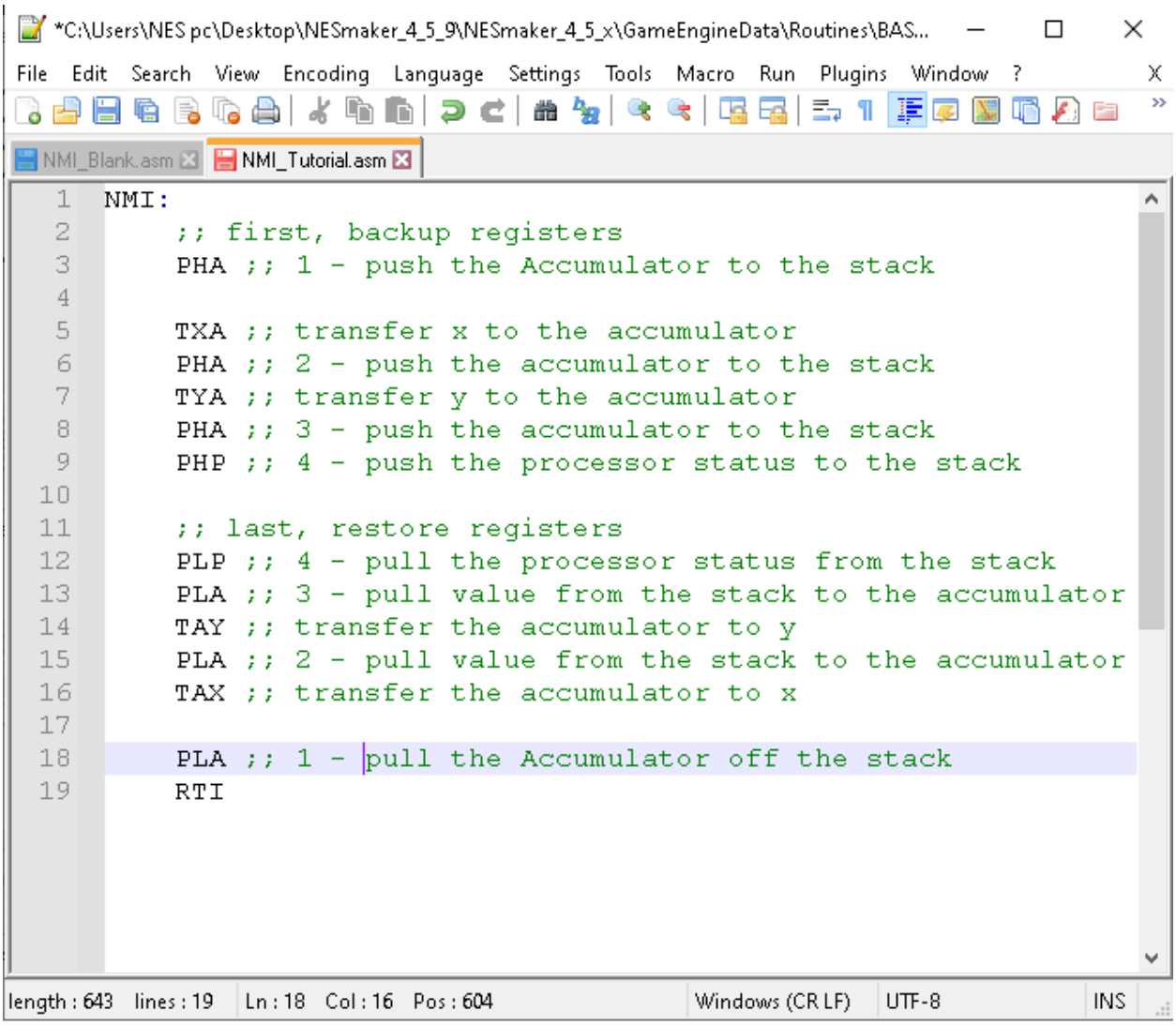
Step 4: Backing up the other registers, too.
There are three main registers that we will use constantly and very well might corrupt in our eventual NMI code, and there is one that it is a good idea to back up and restore as well.

I left a space between our existing code and what I added, and made sure to comment everything appropriately. Besides the accumulator, we are going to store and restore the processor status (P), the x register, (x) and the y register (y). The x and y registers are similar in many ways to the accumulator, but have specific uses that we'll talk about through this tutorials

You'll notice, I backed up all the registers in one spot at the beginning of the NMI, then restored them all at the end of the NMI. You'll also notice that they are restored in the reverse order that they were added to the stack. Again, this is pretty easy to understand if you already understand the

stack. When you place a value into the stack, it is placed on top of everything that came before it on the stack. Imagine dealing five playing cards into a pile. If you dealt them in the order Jack, Deuce, Queen, Ace, and Ten, you'd be looking at the 10 on top of the pile. If you then pulled them off that pile one at a time, you'd pull them off in the order Ten, Ace, Queen, Deuce, then Jack. The stack works in the same way. The first value added to the stack is at the bottom, and every subsequent value is placed on top. When we pull values from the stack, it starts at the top.

So in this example, we're pushing values to the stack in the order A, X, Y, P, and then pulling them from the top, restoring from the stack in the reverse order of P, Y, X, A.



Step 5: Add some NMI Functionality.
We're going to continue to come back and revise this as we start to build out our code, but for now we can put some placeholders for our graphic updates in the middle. Graphic updates include updates to vRam (video ram), which is specific RAM dedicated to the PPU (picture processing unit). In our game's logic, we are setting up potential changes for the game's next frame, during the NMI we write those changes to this vRam in the PPU, and on the next frame, the PPU renders those changes. For instance, the logic of your game has the player object move to the left 4 pixels. This has happened mathematically, and the memory addresses that determine his position have changed to reflect the change. But the change won't be visible yet. Not until vBlank, when the NMI is tripped, the new data from those memory addresses are pushed into vRam, and the screen is redrawn in the next frame. Of course, this happens 60 times per second so it appears instantaneous, but it is still good to understand how it is all working.

Before adding code, let's add a skeleton for the types of things we'll update in our NMI. For the purposes of this instructional, we're not worried about optimizing and are more concerned with the concepts. A legitimate game might have better ways to mine this data, but we're going to try to convey it as clearly as possible for the time being.

```
NMI:
    ;; first, backup registers
    PHA ;; 1 - push the Accumulator to the stack

    TXA ;; transfer x to the accumulator
    PHA ;; 2 - push the accumulator to the stack
    TYA ;; transfer y to the accumulator
    PHA ;; 3 - push the accumulator to the stack
    PHP ;; 4 - push the processor status to the stack
;;=========== END OF REGISTER BACKUP

    ;; Handle sprite updates

    ;; Handle background tile updates

    ;; handle palette updates

    ;; handle frame timing

    ;; RESTORE PPU CONTROL


;;=========== BEGINNING OF REGISTER RESTORE
    ;; last, restore registers
    PLP ;; 4 - pull the processor status from the stack
    PLA ;; 3 - pull value from the stack to the accumulator
    TAY ;; transfer the accumulator to y
    PLA ;; 2 - pull value from the stack to the accumulator
    TAX ;; transfer the accumulator to x

    PLA ;; 1 - pull the Accumulator off the stack
    RTI
```

Here, I added some section dividers so it's easy to see what I am adding. Anything commented out is not necessary, but sure makes things easier to read and to find when you need to!

There is more that we'll eventually add into the body of our NMI, and there are some ways that we can better approach the NMI too, but this gives a clear idea of the types of things that will go on here. First,

Let's definitely handle the sprite updates first, which we can do directly here in the NMI with just a few lines of code.

Step 6: Push our sprite data to the vRam so it can be drawn during the next frame render.

```
1  NMI:
2      ;; first, backup registers
3      PHA ;; 1 - push the Accumulator to the stack
4
5      TXA ;; transfer x to the accumulator
6      PHA ;; 2 - push the accumulator to the stack
7      TYA ;; transfer y to the accumulator
8      PHA ;; 3 - push the accumulator to the stack
9      PHP ;; 4 - push the processor status to the stack
10 ;;============ END OF REGISTER BACKUP
11
12     ;; Handle sprite updates
13         LDA #$00    ; Load zero to the accumulator
14         STA $2003   ; Store it to the low byte of the OAM Address
15         LDA #$02    ; Load two to the accumulator
16         STA $4014   ; Store it to the OAM DMA high address
17
18     ;; Handle background tile updates
19
20     ;; handle palette updates
21
22     ;; handle frame timing
23
24     ;; RESTORE PPU CONTROL
25
26
```

There is a bit to unpack here. In thesefour lines of code, what we are doing is twofold. The first thing to notice is that we're talking about a 16 bit variable; one high byte and one low byte. This is pretty easy to imagine. If a single byte that is made of 8 bits can be expressed with two digits, from #$00 - #$FF, then you can imagine how a 16 bit value can be expressed in four digits; #0000-#$FFFF. All of our memory addresses are expressed with 16 bit values. Remember when creating our memory map, we used values such as $0600 and $0200, and $0000? Those are sixteen bit values. The first byte is the high byte, and the second byte is the low byte. So for the random address $038F, the high byte is #$03 and the low byte is #$8F.

Here, $2003 and $4014 are special memory locations that are specific to the NES architecture. They both have to do with the OAM (Object Attribute Memory). That is the part of the internal PPU memory that has to do with

drawing sprites. There is room for 64 sprites, each needing 4 bytes to represent; x value, y value, tile number and attribute information (flips, palette choice, etc). So the reason that the NES has a hard 64 sprite limit is that there are only 256 bytes allocated to this in its PPU ram, and 64x4 = 256.

Now, understanding what we do about 16 bit variables, we can see we're doing something with pushing #$00 to a low byte of something, and #02 to the high byte of something. So we're effectively pushing $0200 to something, and we know this involves sprites.

If you look back at our memory map, we determined that our label SpriteRam would be placed at $0200. Basically, our logic is going to use $0200 to deal with the y value of our first sprite, $0201 to deal with the tile choice of our first sprite, $0202 to deal with the attribute information of our first sprite, and $0203 to deal with the x value of our first sprite. We will be writing that during logic time to the SpriteRam address. Here during the NMI, we will be copying the values of those addresses to vRam to prepare for rendering accordingly in the next frame.

Writing #$00 to the PPU address $2003 tells the program to prepare a copy routine at the first available PPU slot (#$00) for sprite data. Writing #$02 to $4014 lets the program know from where it will copy data. It is the high byte of our SpriteRam, which we set to be $0200. The result of all this is that we can write sprite values to $0200-$02ff during regular logic time, and the PPU will update the render based on that in the next frame by copying the contents of those mailboxes to the picture processing unit's vRam.

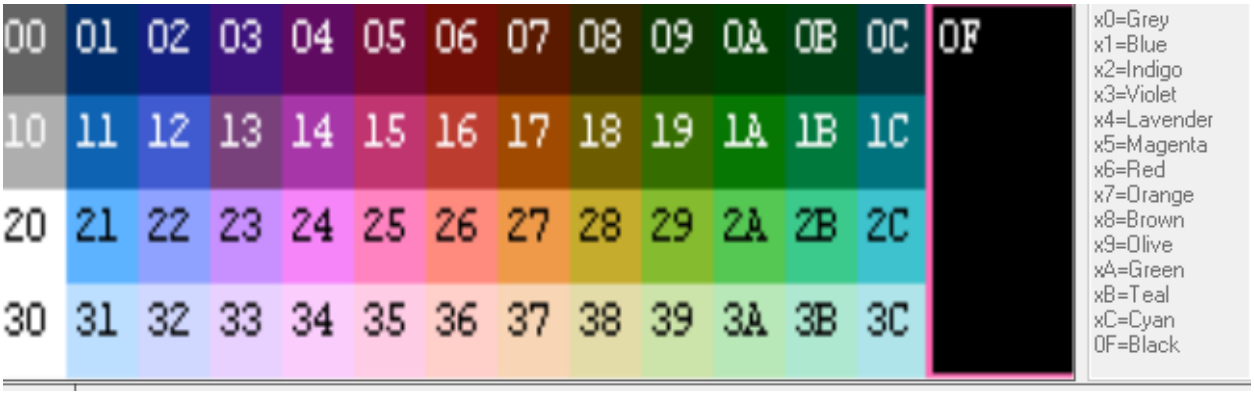
For the moment, we'll skip over handling background tile updates.

Step 7: Create some constants for colors.

Temporarily, we're going to directly write to the palette addresses in the PPU, though this does give us a great opportunity to create some constants with proper names so that we don't have to try to remember which color is what.

This is a list of colors that are accessible to NESmaker through the GUI tools. On each color is a hex value, and off to the right there are common

names that just help identify these things for the tool. We'll use this as a guide to set up our constants. Remember, constants are not actually being factored into your code or taking up ROM space - they are just identifiers that get substituted with their value at run time, so we can really put our constant definitions anywhere.



I could go through and give these all common names with constants, and I might eventually do that. For now, I'm just going to set up constants for a handful of them.

```
NMI:
    ;; first, backup registers
    PHA ;; 1 - push the Accumulator to the stack

    TXA ;; transfer x to the accumulator
    PHA ;; 2 - push the accumulator to the stack
    TYA ;; transfer y to the accumulator
    PHA ;; 3 - push the accumulator to the stack
    PHP ;; 4 - push the processor status to the stack
;;============ END OF REGISTER BACKUP

    ;; Handle sprite updates
        LDA #$00    ; Load zero to the accumulator
        STA $2003   ; Store it to the low byte of the OAM Address
        LDA #$02    ; Load two to the accumulator
        STA $4014   ; Store it to the OAM DMA high address

    ;; Handle background tile updates

    ;; handle palette updates
    C_GRAY = #$00
    C_WHITE = #$30
    C_BLACK = #$0F
    C_BLUE = #$01
    C_PURPLE = #$03
    C_RED = #$05
    C_BROWN = #$08
    C_GREEN = #$0A
    C_GOLD = #$28
    C_PEACH = #$26
    C_ORANGE = #$17
    C_OLIVE = #$28
    C_PINK = #$24
    C_LIGHT_BLUE = #$31
    C_LIGHT_GREEN = #$39
    C_LIGHT_GRAY = #$10
```

Step 8: Directly set some palettes via vRam addresses.
To write to the PPU, the rendering must be turned off. Fortunately during vBlank, rendering is always turned, so we can do it directly. When we flesh the game out more, we'll likely want to handle this with RAM variables similar to how we're handling sprites, but for now, we'll just slam in a bunch of values to the palette addresses in vRam, which begin at $3F00.

To write to the PPU, we write our high and low bytes to $2006, and then the value we'd like to place in that proverbial mailbox to $2007. So, for instance, if I wrote:

```
LDA #$3F
STA $2006
LDA #$00
STA $2006
LDA #$00
STA $2007
```

This would write a value of 00 to the memory address $3F00, which is the address for the first palette color.

```
10  ;;=========== END OF REGISTER BACKUP
11
12      ;; Handle sprite updates
13          LDA #$00    ; Load zero to the accumulator
14          STA $2003   ; Store it to the low byte of the OAM Address
15          LDA #$02    ; Load two to the accumulator
16          STA $4014   ; Store it to the OAM DMA high address
17
18      ;; Handle background tile updates
19
20      ;; handle palette updates
21      ;;; CREATE SOME COLOR CONSTANTS. These can go anywhere, they
22      ;;; do not have to go here in the NMI
23              C_GRAY = #$00
24              C_WHITE = #$30
25              C_BLACK = #$0F
26              C_BLUE = #$01
27              C_PURPLE = #$03
28              C_RED = #$05
29              C_BROWN = #$08
30              C_GREEN = #$0A
31              C_GOLD = #$28
32              C_PEACH = #$26
33              C_ORANGE = #$17
34              C_OLIVE = #$28
35              C_PINK = #$24
36              C_LIGHT_BLUE = #$31
37              C_LIGHT_GREEN = #$39
38              C_LIGHT_GRAY = #$10
39
40      ;;; push a color value to vRam
41      LDA #$3f ;; the high byte of the destination
42      STA $2006 ;; gets written to $2006 first
43      LDA #$00 ;; the low byte of the destination
44      STA $2006 ;; gets written to $2006 second
45      LDA #C_PURPLE ;; the color value
46      STA $2007 ;; gets written to $2007
47
```

For the moment, we'll hold off on the other colors, but right now, we will be writing purple to the very first palette slot. Once we get through a few more steps, we'll check our work.

Step 9: Restore PPU Control.

We have two more addresses we want to write to at the end of our NMI; the PPU CONTROL and the PPU MASK, located at addresses $2000 and $2001 respectively. For a quick glance, this is what each bit for these bytes do.

```
$2000, PPU CONTROL

7  bit  0
---- ----
VPHB SINN
|||| ||||
|||| ||++- Base nametable address
|||| ||    (0 = $2000; 1 = $2400; 2 = $2800; 3 = $2C00)
|||| |+--- VRAM address increment per CPU read/write of PPUDATA
|||| |     (0: add 1, going across; 1: add 32, going down)
|||| +---- Sprite pattern table address for 8x8 sprites
||||       (0: $0000; 1: $1000; ignored in 8x16 mode)
|||+------ Background pattern table address (0: $0000; 1: $1000)
||+------- Sprite size (0: 8x8 pixels; 1: 8x16 pixels
|+-------- PPU master/slave select
+--------- Generate an NMI at the start of the
           vertical blanking interval (0: off; 1: on)

$2001, PPU MASK
7  bit  0
---- ----
BGRs bMmG
|||| ||||
|||| |||+- Greyscale (0: normal color, 1: greyscale)
|||| ||+-- 1: Show background in leftmost 8 pixels of screen, 0: Hide
|||| |+--- 1: Show sprites in leftmost 8 pixels of screen, 0: Hide
|||| +---- 1: Show background
|||+------ 1: Show sprites
||+------- Emphasize red (green on PAL/Dendy)
|+-------- Emphasize green (red on PAL/Dendy)
+--------- Emphasize blue
```

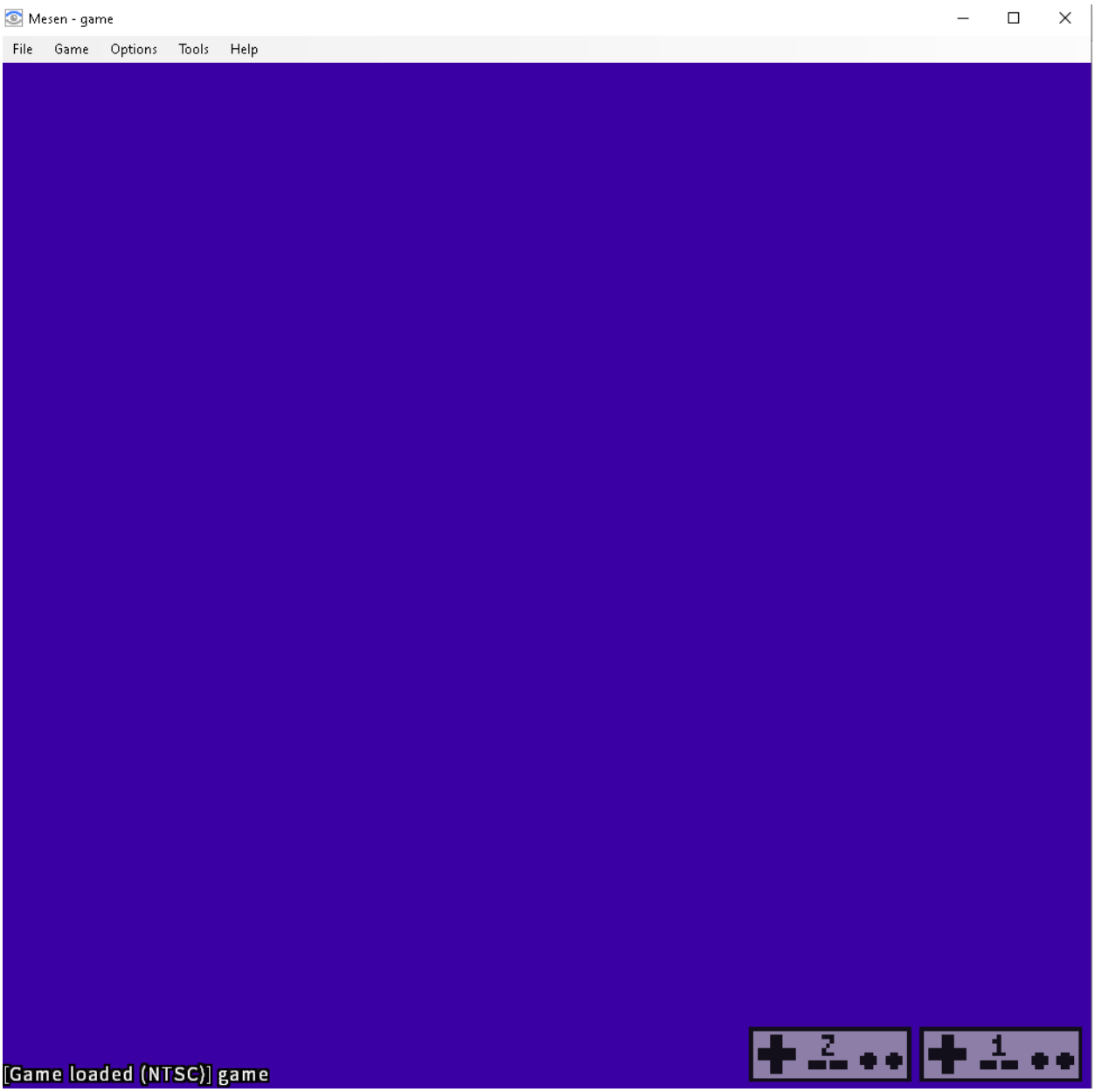
There are only a few things we need to worry about in each right now, but later we'll get into more details. This is all going into the Restore PPU Control section that we set up in the NMI file.

```
;; This will make sure that we continue hitting NMI during vblank
;; and will set our sprite graphics to the first pattern table and
```

```
;; background tiles to the second pattern table.

      LDA #%10010000
      STA $2000

;; This will enable sprites and background graphics.
;; It will not hide sprites at the edges of the screen.
      LDA #%00011110
      STA $2001

;; This will make sure that our scroll is set to zero
;; in both x and y direction
      LDA #$00
      STA $2005
      STA $2005
```

```
    LDA #$3f ;; the high byte of the destination
    STA $2006 ;; gets written to $2006 first
    LDA #$00 ;; the low byte of the destination
    STA $2006 ;; gets written to $2006 second
    LDA #C_PURPLE ;; the color value
    STA $2007 ;; gets written to $2007


    ;; handle frame timing

    ;; RESTORE PPU CONTROL
    ;; This will make sure that we continue hitting NMI during vblank
    ;; and will set our sprite graphics to the first pattern table and
    ;; background tiles to the second pattern table.

    LDA #%10010000
    STA $2000

    ;; This will enable sprites and background graphics.
    ;; It will not hide sprites at the edges of the screen.
        LDA #%00011110
        STA $2001

    ;; This will make sure that our scroll is set to zero
    ;; in both x and y direction
        LDA #$00
        STA $2005
        STA $2005


;;=========== BEGINNING OF REGISTER RESTORE
    ;; last, restore registers
```

Make sure to save your file. Now, you can go back to NESmaker and test your game with the export and test button. The screen should turn purple. Eureka! But wow - all this to get the screen to change color!



The reason the screen turned purple is because we set purple to the very first palette slot, basically “slot zero”. Since we haven’t painted with any color to the screen yet, everything is a null value, thus zero, thus purple.

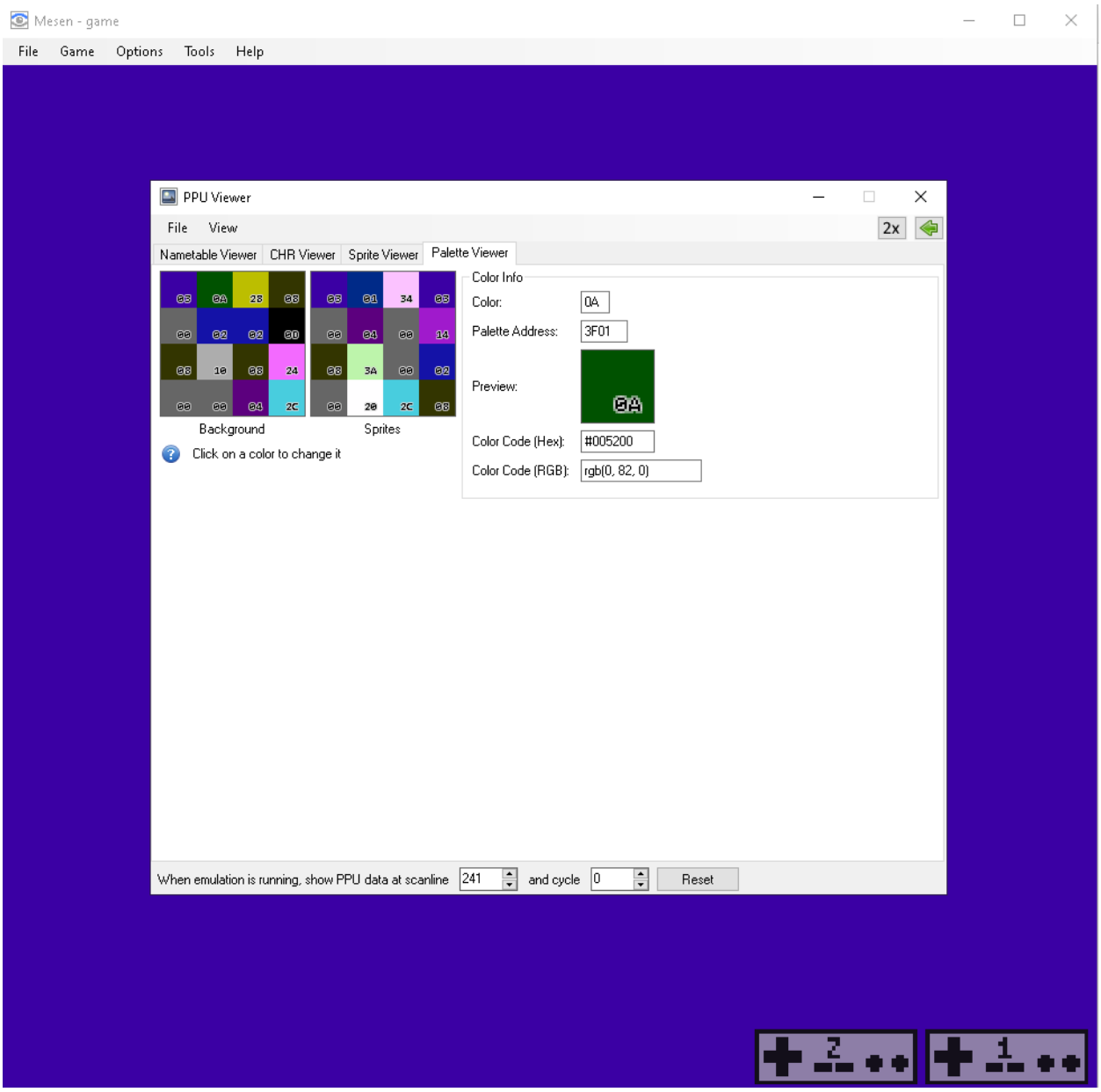
Let’s add four colors to our background palette the long way based on this understanding. The following is a terrible way to do this, but it will work, and you should be able to follow along and understand what we’re doing based on what we’ve done so far. We will refine this later.

Step 10: Add colors to our palette.
Where we defined that purple color in the last step, copy and paste those six lines of code. Now, change the second write to $2006. We should see #$00 for the first color, #$01 for the second color, #$02 for the third color, and #$03 for the fourth color. This is telling the game to write to slot 0, 1, 2 and 3 in the vRam memory reserved for palette info.

Then change the colors so we have four distinct colors being drawn. I used purple, green, gold, and brown.

```
            C_GOLD = #$28
            C_PEACH = #$26
            C_ORANGE = #$17
            C_OLIVE = #$28
            C_PINK = #$24
            C_LIGHT_BLUE = #$31
            C_LIGHT_GREEN = #$39
            C_LIGHT_GRAY = #$10

    ;;; push a color value to vRam
    LDA #$3f ;; the high byte of the destination
    STA $2006 ;; gets written to $2006 first
    LDA #$00 ;; the low byte of the destination
    STA $2006 ;; gets written to $2006 second
    LDA #C_PURPLE ;; the color value
    STA $2007 ;; gets written to $2007

    LDA #$3f ;; the high byte of the destination
    STA $2006 ;; gets written to $2006 first
    LDA #$01 ;; the low byte of the destination
    STA $2006 ;; gets written to $2006 second
    LDA #C_GREEN ;; the color value
    STA $2007 ;; gets written to $2007


    LDA #$3f ;; the high byte of the destination
    STA $2006 ;; gets written to $2006 first
    LDA #$02 ;; the low byte of the destination
    STA $2006 ;; gets written to $2006 second
    LDA #C_GOLD ;; the color value
    STA $2007 ;; gets written to $2007


    LDA #$3f ;; the high byte of the destination
    STA $2006 ;; gets written to $2006 first
    LDA #$03 ;; the low byte of the destination
    STA $2006 ;; gets written to $2006 second
    LDA #C_BROWN ;; the color value
    STA $2007 ;; gets written to $2007
```

Make sure to save the file, then run your game. It looks the same so far, but try pressing Control P to bring up the MESESN emulator's PPU viewer, and then clock on the palette tab. If you did this correctly, what you'll see is that the first four colors are as you defined them. Feel free to play around with changing the values of the colors and see the different results here in the PPU viewer.



END OF LESSON 2

Make sure to save your project from the file menu in NESmaker. This has been a lot of effort to get something happening in the emulator. You can see why we began developing tools to handle some of this up front heavy lifting! But you'll definitely be a better developer, whether you use NESmaker as a tool or not, for having a better understanding of what exactly the GUI of NESmaker is doing under the hood.

By now, you should understand bit about the NMI and what vBlank is. You should understand 16 bit memory addressing. You should have reinforced how to make constants, and learned how to write values directly to the vRam from inside the NMI.