

LESSON 6

We've spent a lot of time setting up some fundamental architecture without seeing much immediate feedback. One of the things NESmaker's default modules is meant to do is to allow a user to hyperspeed beyond these common engine setup steps and launch right into the creative part of development. But under the hood, these are all of the same foundational elements that are present, and just as we can get in and edit the scripts while we're creating things from scratch, those template modules that offer a massive head start on all this stuff are editable in the exact same way.

From now on, though, we should have relatively instant feedback for most of the things we'll be doing, and our lessons will begin to yield something that looks a bit more like a game.

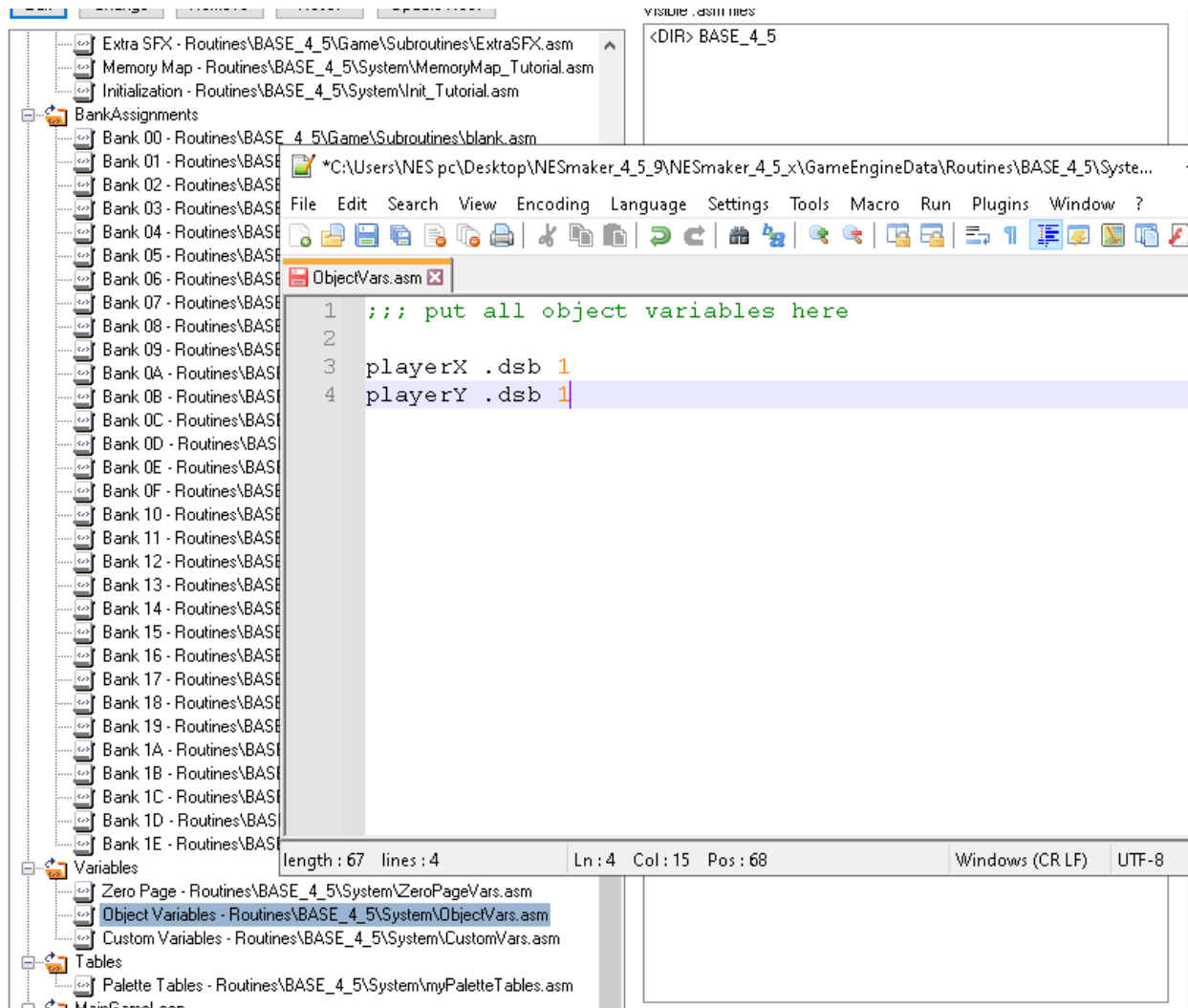
In the previous lesson, we finally drew our player to the screen. In this lesson, we're going to set up a very rudimentary system for managing objects, and as a result, rewrite our drawing routine to allow for this object management. Additionally, we'll do some very simple work with reading the controller. By the end of this lesson, you will be able to move the player around the screen using the controller, and we'll set up a frame management system so that the movement is consistent.

Step 1: Creating object property variables.

Right now, we're drawing our object to a static position. We don't want to draw him in a static position. We want to draw him to a variable position, and as we press the dpad, we want that variable position to change. For this, we will create some variable handlers.

From Script Settings, open our Object Variables script. This was a script that we included into memory management, and it points to a chunk of RAM that we have declared. Here, we're going to place a few variables for our player to keep track of his x and y value. If you're familiar with programming, you would understand this effectively as declaring a pair of one-byte variables. In ASM terms, we are giving particular memory addresses (again, think mailboxes) easy to read labels with one byte of storage space each. Functionally, the concept is the same. We can write to and read from the address represented by the label the same way we would write to and read from a variable.

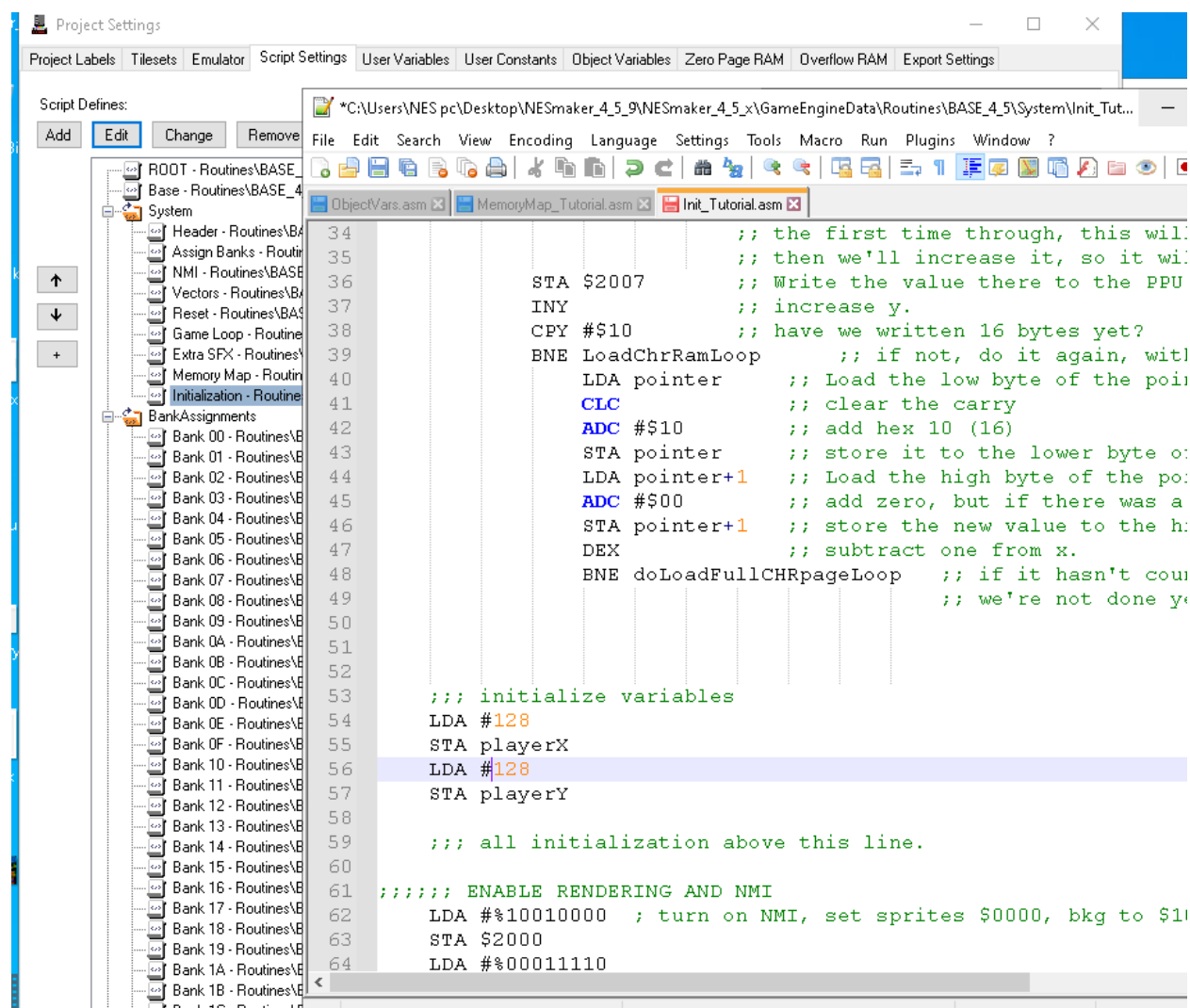
But for clarity to understand a bit better what's actually happening, our ObjectRam space is where this file is included. We set our ObjectRam space to be \$0400. These represent the first two bytes at that address. So effectively, right now playerX lives at address \$0400 and playerY lives at address \$0401. Reading from or writing to these memory addresses would have the exact same outcome as reading and writing to playerX or playerY as labels.



Step 2: Setting the initial state of the variables

Next, navigate to your initialization script and scroll down to just after the palettes are finished loading, but before we've re-enabled rendering and NMI. Here, you'll see I added a comment to denote a section for initialization of variables. Here's where you can put all the beginning states of variables that we might need to set at the start of your game.

Right now, I am setting each to decimal value 128.



```
34                                     ;; the first time through, this will
35                                     ;; then we'll increase it, so it will
36     STA $2007                         ;; Write the value there to the PPU
37     INY                               ;; increase y.
38     CPY #$10                          ;; have we written 16 bytes yet?
39     BNE LoadChrRamLoop                ;; if not, do it again, with
40     LDA pointer                       ;; Load the low byte of the pointer
41     CLC                               ;; clear the carry
42     ADC #$10                          ;; add hex 10 (16)
43     STA pointer                       ;; store it to the lower byte of the pointer
44     LDA pointer+1                     ;; Load the high byte of the pointer
45     ADC #$00                          ;; add zero, but if there was a carry
46     STA pointer+1                     ;; store the new value to the higher byte
47     DEX                               ;; subtract one from x.
48     BNE doLoadFullCHRpageLoop        ;; if it hasn't counted, do it again
49                                     ;; we're not done yet
50
51
52
53     ;;; initialize variables
54     LDA #128
55     STA playerX
56     LDA #128
57     STA playerY
58
59     ;;; all initialization above this line.
60
61     ;;; ENABLE RENDERING AND NMI
62     LDA #%10010000                    ; turn on NMI, set sprites $0000, bkg to $10
63     STA $2000
64     LDA #%00011110
```

Step 3: Drawing our player at a variable position.

Open our Sprite Drawing script from the script settings. We want to replace the y values with playerY and the x values with playerX, but we also have to remember that each of the four sprites has some sort of pixel offset. The second sprite is drawn 8 px to the right of our object's origin. The third sprite is drawn 8 px down from our object's origin. Our fourth sprite is drawn 8 pixels to the right and 8 pixels down from our object's origin. So in addition to just using the variables to define positioning, we'll also have to add the appropriate math.

I have indented the math so it's a bit easier to see, but this is just an aesthetic to make it clearer to look at. Indenting the code does not change the functionality.

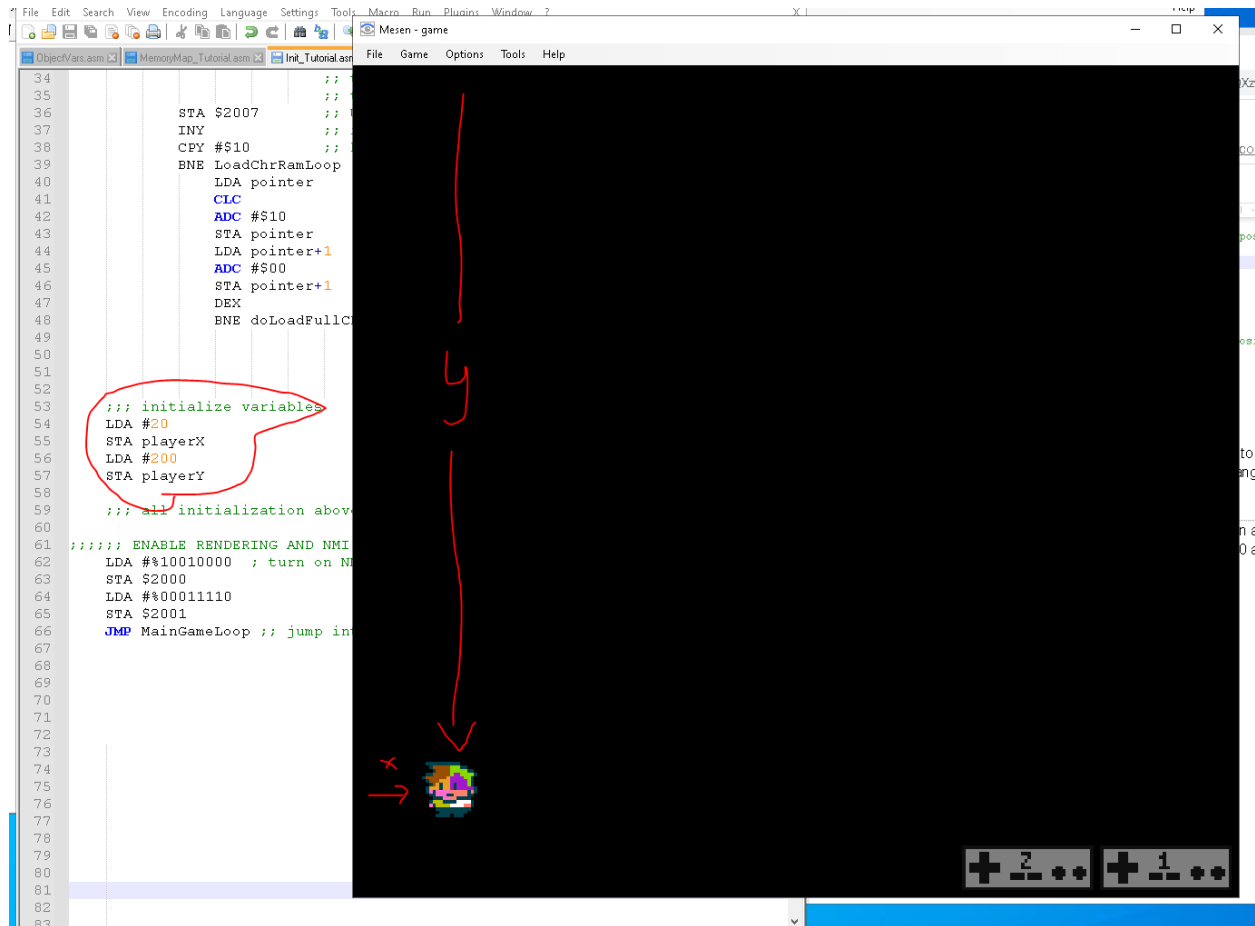
```

1  ;;; This is where we'll draw our sprites.
2
3  ;; SPRITE 1
4  LDA playerY      ; y - Variable Position
5  STA $0200
6  LDA #$00        ; tile
7  STA $0201
8  LDA #%00000000  ; att
9  STA $0202
10 LDA playerX ; x - Variable Position
11 STA $0203
12
13 ;; SPRITE 2
14 LDA playerY ; y - Variable Position
15 STA $0204
16 LDA #$01    ; tile
17 STA $0205
18 LDA #%00000001 ; att
19 STA $0206
20 LDA playerX ; x - Variable Position
21 CLC
22 ADC #$08    ; plus 8
23 STA $0207
24
25 ;; SPRITE 3
26 LDA playerY ; y - variable position
27 CLC
28 ADC #$08    ; plus 8
29 STA $0208
30 LDA #$10    ; tile
31 STA $0209
32 LDA #%00000010 ; att
33 STA $020A
34 LDA playerX ; x - variable position
35 STA $020B
36
37 ;; SPRITE 4
38 LDA playerY ; y - variable position
39 CLC
40 ADC #$08 ; plus 8
41 STA $020C
42 LDA #$11    ; tile
43 STA $020D
44 LDA #%00000011 ; att
45 STA $020E
46 LDA playerX ; x - variable position
47 CLC
48 ADC #$08    ; plus 8
49 STA $020F

```

Test your game and everything should look identical to our last test, but we know that now, simply by changing playerX and playerY, we can change the position of the entire drawn object.

To test this theory, open your initialization script again and play with the numbers. For instance, I'll change both numbers, so that playerX=decimal 20 and playerY= decimal 200 at the start of the game.



```
34      ;;
35      ;;
36      STA $2007      ;;
37      INY           ;;
38      CPY #$10      ;;
39      BNE LoadChrRamLoop
40      LDA pointer
41      CLC
42      ADC #$10
43      STA pointer
44      LDA pointer+1
45      ADC #$00
46      STA pointer+1
47      DEX
48      BNE doLoadFullC
49
50
51
52
53      ;; initialize variables
54      LDA #20
55      STA playerX
56      LDA #200
57      STA playerY
58
59      ;; all initialization above
60
61      ;;;; ENABLE RENDERING AND NMI
62      LDA #%10010000 ; turn on NMI
63      STA $2000
64      LDA #%00011110
65      STA $2001
66      JMP MainGameLoop ;; jump into main game loop
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
```

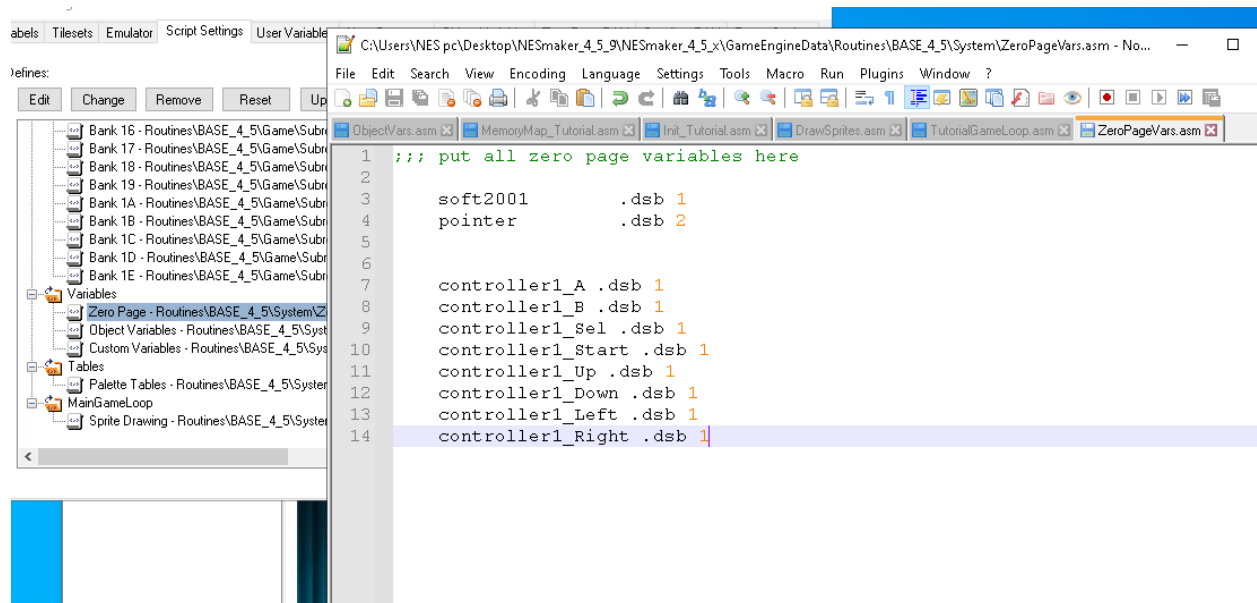
You should see that your player shows up in a different position. The higher the x value, the further to the right. The higher the y value, the further down.

Now we need to work on a way to change these positions using controller input. For ease and clarity, and because we're creating a very light and agile engine, we're again going to get controller reads the long way, wasting a lot of variable space and creating a routing that is very unoptimized. As you get more comfortable with coding in ASM, there are much more efficient ways to get controller data, but the method I'll be showing in the next step should be very easy to read and understand even if you have very little understanding of code.

Step 4: Setting up variables for the controller.

In the unoptimized version of reading a controller, we're going to treat each button state as its own variable. The reason this is wasteful is that each button on the controller really only needs one bit to convey its state - it's either pressed or it's not. This means that a better optimized controller read would likely be able to keep all info about all 8 controller buttons in a single byte (each bit representing the state of the button, 0 for not pressed, 1 for pressed). That means that using a whole byte for each button wastes 7 bytes of precious RAM variable data, however it will make it much easier to read and work with.

In script settings, open the file for your zero page variables. We are going to create a RAM variable for each button - controller1_A, controller1_B, etc.



The screenshot shows an IDE window titled 'C:\Users\NES pc\Desktop\NESmaker_4_5\NESmaker_4_5_x\GameEngineData\Routines\BASE_4_5\System\ZeroPageVars.asm - No...'. The code in the editor is as follows:

```
1 ;; put all zero page variables here
2
3     soft2001      .dsb 1
4     pointer      .dsb 2
5
6
7     controller1_A .dsb 1
8     controller1_B .dsb 1
9     controller1_sel .dsb 1
10    controller1_start .dsb 1
11    controller1_up .dsb 1
12    controller1_down .dsb 1
13    controller1_left .dsb 1
14    controller1_right .dsb 1
```

The left sidebar shows a project tree with 'Variables' expanded, containing 'Zero Page - Routines\BASE_4_5\System\ZeroPageVars.asm'.

Our RESET routine zeros out all RAM, so at the beginning of the game, these will all get set to zero. Because of that, we do not have to set their initial state to zero.

As of this moment, these variables don't mean anything. Effectively, they're just labels for addresses \$0003–000A. In our main game loop, we will populate them with the hardware button states every frame.

Step 5: Reading the hardware button states and assigning to the controller variables

In Script Settings, open the Main Loop. Before we draw routine, we'll add our controller reading routine. To get the data of the state of the hardware, you write twice to the memory address \$4016. Writing a 1 starts reading the controller data, writing a 0 stops reading the controller data. Once you have done these successive writes, you can read \$4016 eight consecutive times to get the status of each button. The first read will be the status of the A button. The second read will be the status of the B button. Etc.

What we want to do is read from \$4016, then mask off the non-important bits so that we're only reading bit 0 (the bit all the way to the right). For this, we'll use an AND statement.

This requires a little bit of explanation of logic. When you AND two bytes together, you are looking at same bits from each to get a result. A 1 AND a 1 will return a 1. A 1 AND a 0 will return a 0. A 0 and a 1 will return a 0. A 0 and a 0 will return a zero.

So think about these numbers:

```
LDA #%11111111  
AND #%00000000
```

The result here, currently loaded in the accumulator, would be `#%00000000`, because a 1 and a 0 logically works out to zero.

Here's another example:

```
LDA #%11111111  
AND #%01010101
```

The result here would be `#%01010101`, because the only bits that would result in a one through this operation would be any comparison where both same place bits have a 1.

A third example:

```
LDA #%01010101
AND #%10101010
```

The result here would be `00000000` for the same reason.

So in our example of reading gamepad data, we would load information from `$4016`, which is the controller read. We would “and out” all of the bits except for the last one (so no matter what they are, they’d result in a zero), and only focus on bit 0 (the one all the way to the right). If in `$4016`, this hardware read is affirmative, it would be a 1. And-ing in a 1 to a 1 would cause the result to be a 1. If, on the other hand, the read from `$4016` is zero, and-ing in a 1 would cause the result to be a zero, because and-ing a 1 and a 0 logically results in 0.

Pretend the read of `$4016` read as true, and had a bunch of garbage data in the other bits that we don’t care about, here’s what the read would look like, with its logical result.

```
LDA #%11010111 ; whatever is in $4016, the last bit
being the important one.
AND #%00000001
(the result is)
      #%00000001 ; because the 1 is only carried through
if both bits were 1.
```

Meanwhile, if it was false, and the read from `$4016` resulted in a zero, it would read like this:

```
LDA #%11010110
AND #%00000001
(the result is)
      #%00000000 ; because 1 would not be carried
through since one of the bits is a zero.
```

Using that logic, we'll read each button state, mask out all of the bits except the one we're interested in, and push the result to the controller variables for each button that we made.

```
LDA #$01
STA $4016      ; start reading the data from the controller
LDA #$00
STA $4016      ; finish reading the data from the controller

;; now, one at a time, populate our controller variables.
;; Once the controller data is polled as above, a read from
;; $4016 will read each button in turn. The order is:
;; A, B, Sel, Start, Up, Down, Left, Right.

;; We will read $4016, and with each read, populate the
;; corresponding variable.

LDA $4016
AND #%00000001
STA controller1_A
LDA $4016
AND #%00000001
STA controller1_B
LDA $4016
AND #%00000001
STA controller1_Sel
LDA $4016
AND #%00000001
STA controller1_Start
LDA $4016
AND #%00000001
STA controller1_Up
LDA $4016
AND #%00000001
STA controller1_Down
LDA $4016
AND #%00000001
STA controller1_Left
LDA $4016
AND #%00000001
STA controller1_Right
```

Now, we can simply read our variables any time we want to know the status of a button. If `controller1_A` is 0, that means it is not pressed. If it is 1, that means that it is pressed.

Step 5: Checking controller input variables and make something happen.

Just below our controller reading code, we're going to check to see if the controller variable that we've associated with the right button is 1 or zero. This code will **Branch if E**Qual to zero passed the movement code. Otherwise, it'll flow right into it.

When you test your game, this will work, but something will be very wrong. In fact, it may even appear that the object is moving left instead of right, and chances are it will be flickering in a weird way. We'll tackle that in a moment, but for now, just try this out and make sure that the player object does move in some way.

(Also, make sure in your emulator, your inputs are set up to whatever you're using. If nothing happens, check to make sure that your gamepad buttons are linked to the right keys or joystick buttons).

```
LDA $4016
AND #%00000001
STA controller1_Start
LDA $4016
AND #%00000001
STA controller1_Up
LDA $4016
AND #%00000001
STA controller1_Down
LDA $4016
AND #%00000001
STA controller1_Left
LDA $4016
AND #%00000001
STA controller1_Right

;;;;; move based on controller input.
LDA controller1_Right      ;read the controller right variable
BEQ +controllerRightNotPressed ;if it's zero, jump passed the action to move
                              ;if it is not zero, end up here.
    INC playerX             ;Increase the value in playerX
                              ;which means "move right"
+controllerRightNotPressed

.include SCR_DRAW_SPRITES

JMP MainGameLoop
```

Step 6: Regular Frame timing.

The reason that the crazy amount of jittering happened rather than a nice fluid motion is because the increase to the player's x value was happening many times per frame. Think about our game loop so far. All it does is read the controller, then move, then do it over again. It's going to have the time to work through those instructions many, many, many times per frame, which will result in the x value increasing many many many times per frame. What we want to do is regulate our main game loop so that all of its logic only occurs once per frame.

We do have a sort of built in method in keeping track of frame timing. We know predictably that the NMI hits every time rendering is finished. This happens predictably 60 times per second. So this is how we want it to work. We want our main game loop to do all of the things that it needs to do. When it's finished, it should loop back up to the top. However, it should not continue to do all of its things again until we've reached the next frame (the next NMI has fired). Once it has, we're in the next frame and we should do all the main loop stuff again.

We need a new RAM variable. We'll call it `frameTimer`. Open up your zero page and make a one byte variable called `frameTimer`.

```
ObjectVars.asm | MemoryMap_Tutorial.asm | Init_Tutorial.asm | DrawSprites.asm
1  ;;; put all zero page variables here
2
3  soft2001      .dsb 1
4  pointer      .dsb 2
5
6
7  controller1_A .dsb 1
8  controller1_B .dsb 1
9  controller1_Sel .dsb 1
10 controller1_Start .dsb 1
11 controller1_Up .dsb 1
12 controller1_Down .dsb 1
13 controller1_Left .dsb 1
14 controller1_Right .dsb 1
15
16 frameTimer .dsb 1
```

We want the fame timer to change every time we hit a new frame. We'll have it increase every time we hit the NMI. Open your NMI and scroll down until after the palettes are loaded but before we've restored the PPU. If you've set yours up exactly like mine, you'll see a comment for a space where we will handle frame time. There, simply add INC frameTimer.

```

LDX #$00
doLoadSpritePaletteLoop:
    LDA objectPal_0,x          ;; load the value from
                                ;; objectPal_0, but
                                ;; with the offset of whatever
                                ;; value is in the X register
    STA $2007                  ;; Write it to vRam
    INX                        ;; increase the x register
    CPX #$10                   ;; did X hit 16 yet?
    BNE doLoadSpritePaletteLoop ;; if it did not, do the loop
                                ;; again. If so, flow forward.

;; handle frame timing

INC frameTimer

;; 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

```

Lastly, in our Main Loop, if we happen to hit the top of the loop again, we want it to hang out until frameTimer has changed.

This small couple of lines can be quite cryptic if you don't think it through. Let's work it out logically.

Step 1: Load the value of frameTimer into the accumulator.

Step 2: Jump into a loop.

Step 3: Compare the value in the accumulator to frameTimer

Step 4: if frameTimer is the same as what's in the accumulator, loop until it's not.

```
ObjectVars.asm x MemoryMap_Tutorial.asm x Init_Tutorial.asm x DrawSprites.asm x TutorialGameLoop.asm x Z
1 MainGameLoop:
2
3     ;;; handle frame timing
4     LDA frameTimer
5     doTimingLoop
6         CMP frameTimer
7         BEQ doTimingLoop
8     ;;; end handle frame timing
9
10
11     LDA #$01
12     STA $4016     ; start reading the data from the controller
13     LDA #$00
14     STA $4016     ; finish reading the data from the controller
15
16     ;; now, one at a time, populate our controller variables.
17     ;; Once the controller data is polled as above, a read from
18     ;; $4016 will read each button in turn. The order is:
```

So how does this work? Well, let's imagine it like the face of a clock. I'm going to look at the minutes. It's 12:00, so the minute says 00. I'm going to do one pushup every minute. The numbers zero zero are what I have to focus on, so I put that in the back of my brain. I do a push up. Then I look at the clock. Is it zero zero? Yup...so I look at the clock. Is it zero zero? Yup...so I look at the clock. Is it zero zero? Yup...so I look at the clock. Is it zero zero? Wait! It changed to zero one! Ok, another pushup. Now I need to compare it to zero one. Is it zero one? Yup...so I look at the clock. Is it zero one? Yup...so I look at the clock. Is it zero one? Wait! It changed to zero two! Ok, another pushup....

And so on and so forth. To put that in terms of the code, we're looking at frameTimer. At the beginning of the game, it will be zero. We load zero into the accumulator, then we start our loop. Is frame timer still the value in the accumulator (zero)? Yes, ok...keep checking. Still zero? Yes, ok...keep checking. Still zero? Yes, ok...keep checking. Then at some point, the NMI fires because the frame is over. The NMI takes us out of our code and does

all of its stuff, at the end of which, increases frameTimer to one, then throws us right back where it nabbed us from, which is in that loop. The accumulator has been restored to zero, but frameTimer is no longer equal to what is in that accumulator - it has advanced to one! Time to do a proverbial pushup (run our MainGameLoop code). When the MainGameLoop code is done, it jumps back up to the top and the first thing it does is load frameTimer, which is now one. One gets pushed into the accumulator. Ok, is the frameTimer equal to what's in the accumulator? Yes...keep checking.

And repeat.

This is how we can regulate our frame so that our MainGameLoop runs exactly once every frame, which is a great way to manage the game's timing.

With this little bit of code written, try out the game, and you should see that movement is as expected.

Step 7: Adding controls for four directions.

Simple enough - we will check each of our button variables in turn and move our player's x and y position based on those button states.

Writing INC before a variable will increase the variable by one. Writing DEC before a variable will decrease the variable by one. In later steps, we will add a variable for the speed of the player.


```

3      AND #00000001
4      STA controller1_Left
5      LDA $4016
6      AND #00000001
7      STA controller1_Right
8
9      ;;;; move based on controller input.
0      ;;; RIGHT
1      LDA controller1_Right           ;read the controller right variable
2      BEQ +controllerRightNotPressed ;if it's zero, jump passed the action to move
3      |                               ;if it is not zero, end up here.
4      |      INC playerX               ;Increase the value in playerX
5      |                               ;which means "move right"
6      |      +controllerRightNotPressed
7      ;;; LEFT
8      LDA controller1_Left
9      BEQ +controllerLeftNotPressed
0      |      DEC playerX
1      |      +controllerLeftNotPressed
2
3      ;;; DOWN
4      LDA controller1_Down
5      BEQ +controllerDownNotPressed
6      |      INC playerY
7      |      +controllerDownNotPressed
8
9      ;;; UP
0      LDA controller1_Up
1      BEQ +controllerUpNotPressed
2      |      DEC playerY
3      |      +controllerUpNotPressed
4
5      ;;;; End controll based input
6
7      .include SCR_DRAW_SPRITES
8

```

END OF LESSON 6.

By the end of this lesson, you should have a character moving around the screen. You should have a basic understanding of working with variables. You should also be familiar with how to poll the controller, and how to use that data to affect change to your game. You should understand how to use AND to mask bits, and you should know a process for regulating your frame timing.