NES ASM Tutorial

Tools

Tutorial Tools

These tutorials use a specific set of tools that you should get:

NESASM 3 - This is the assembler you will use to convert your source code into the .NES file.

Tile Layer Pro - This app is a graphics tile editor for the NES format. It can also import bitmaps you have created other places.

FCEUXD SP - This is a good accurate emulator, with many extra debugger features added. If your game runs in this emulator, it is very likely to run on the real NES. You should NOT use old inaccurate emulators like Nesticle.

PowerPak - This is the easiest way to see your game running on the actual hardware. No mods are needed to your NES or your game.

PowerPak Lite - This is the fastest cycle time when you are frequently testing your game on real hardware. Requires a CopyNES system.

NESDevWiki - Great technical reference, most errors have been corrected....

NESDev - Collection of links and messageboards for technical info. Ask questions there once you have learned the basics.

Alternate Tools

You may want to investigate these tools as you get deeper into NES programming:

Assemblers - There are many other popular assemblers including CA65, P65, and WLA-DX. They tend to handle larger projects better, but are harder to set up correctly. Each assembler has small differences in syntax so your code will generally need modification to work on them.

Graphics Editors - The main alternatives are Tile Molester and YYCHR. Many people also create their own graphics editors.

Emulators - Other accurate emulators include Nintendulator, Nestopia, and FCEU. Inaccurate emulators like Nesticle are the reason so many older homebrew games and hacks will not run on the actual NES.

Flash Carts - If you have an eprom programmer you can rewire NES carts with your game. This can be significantly cheaper than the PowerPak or PowerPak Lite but is lots more manual work.

NES Architecture

ROM - Read Only Memory, holds data that cannot be changed

RAM - Random Access Memory, holds data that can be read and written. When power is removed, the chip is erased.

PRG - Program memory, the code for the game

CHR - Character memory, the data for graphics

CPU - Central Processing Unit, the main processor chip

PPU - Picture Processing Unit, the graphics chip

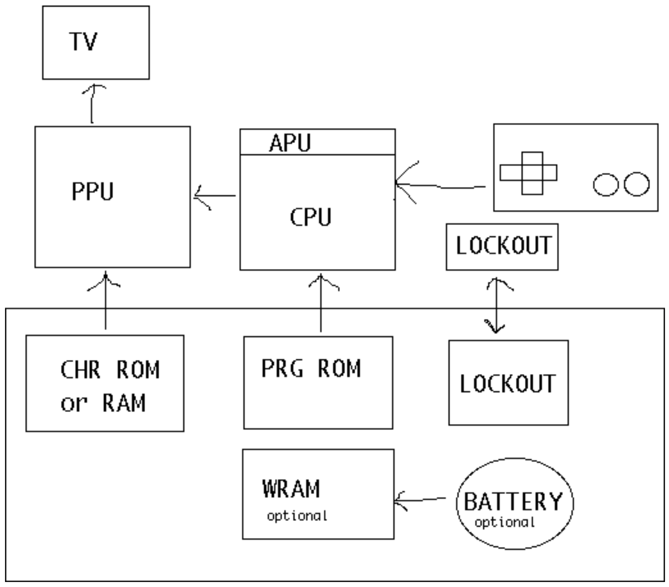
APU - Audio Processing Unit, the sound chip inside the CPU

System Overview

The NES include a CPU with built in APU and controller handling, and a PPU that displays graphics. Your code runs on the CPU and sends out commands to the APU and PPU. The NOAC clones put all of these parts onto one chip.

There is only 2KB of RAM connected to the CPU for storing variables, and 2KB of RAM connected to the PPU for holding two screens of background graphics. Some carts add extra CPU RAM, and a few add extra PPU RAM.

Each cart includes at least two chips. One holds the program code (PRG) and the other holds the character graphics (CHR). The graphics chip can be RAM instead of ROM, which means the code would copy graphics from the PRG chip to the CHR RAM.

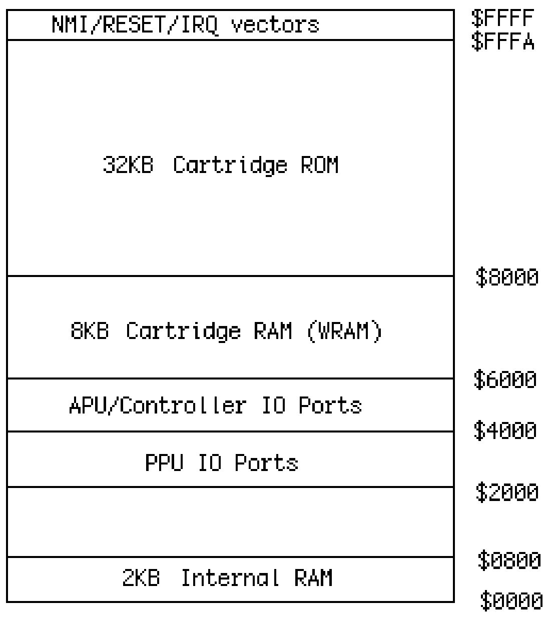


CPU Overview

The NES CPU is a modified 6502, an 8 bit data processor similar to the Apple 2, Atari 2600, C64, and many other systems. By the time the Famicom was created it was underpowered for a computer but great for a game system.

The CPU has a 16 bit address bus which can access up to 64KB of memory. Included in there is the 2KB of RAM, ports to access PPU/APU/controllers, and 32KB for PRG ROM.

32KB quickly became too small for games, which is why memory mappers were used. Those mappers can swap in different banks of PRG code or CHR graphics. No mappers will be used for these docs.



PPU Overview

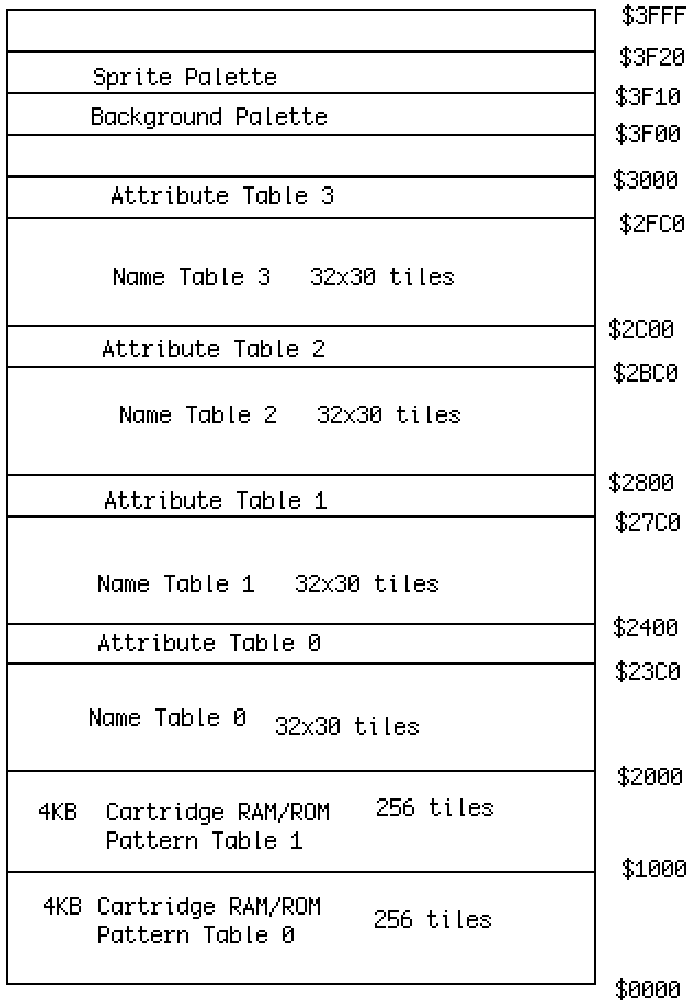
The NES PPU is a custom chip that does all the graphics display. It includes RAM for sprites and the color palette. There is RAM on board that holds the background, and all actual graphics are fetched from the cart CHR memory.

Your program does not run on the PPU, it always does the same process. You only set some options like colors and scrolling.

Both the NTSC and PAL systems have a resolution of 256x240 pixels, but the top and bottom 8 rows are typically cut off by the NTSC TV resulting in 256x224.

NTSC runs at 60Hz and PAL runs at 50Hz. Running an NTSC game on a PAL system will be slower because of this timing difference.

PAL has a longer VBlank time (when the TV scanline is going back to the top of the screen) which allows more time for graphics updates. This is why some PAL games and demos do not run on NTSC systems.


$3FFF
Sprite Palette
$3F20
Background Palette
$3F10
$3F00
$3000
Attribute Table 3
$2FC0
Name Table 3   32x30 tiles
$2C00
Attribute Table 2
$2BC0
Name Table 2   32x30 tiles
$2800
Attribute Table 1
$27C0
Name Table 1   32x30 tiles
$2400
Attribute Table 0
$23C0
Name Table 0   32x30 tiles
$2000
4KB Cartridge RAM/ROM Pattern Table 1   256 tiles
$1000
4KB Cartridge RAM/ROM Pattern Table 0   256 tiles
$0000

Graphics System Overview

Tiles

All graphics are made up of 8x8 pixel tiles. Large characters like Mario are made from multiple 8x8 tiles. All the backgrounds are also made from these tiles.

Sprites

The PPU has enough memory for 64 sprites, or things that move around on screen like Mario. Only 8 sprites per scanline are allowed, any more than that will be ignored. This is where the flickering comes from in some games when there are too many objects on screen.

Background

This is the landscape graphics, which scrolls all at once. The sprites can either be displayed in front or behind the background. The screen is big enough for 32x30 background tiles.

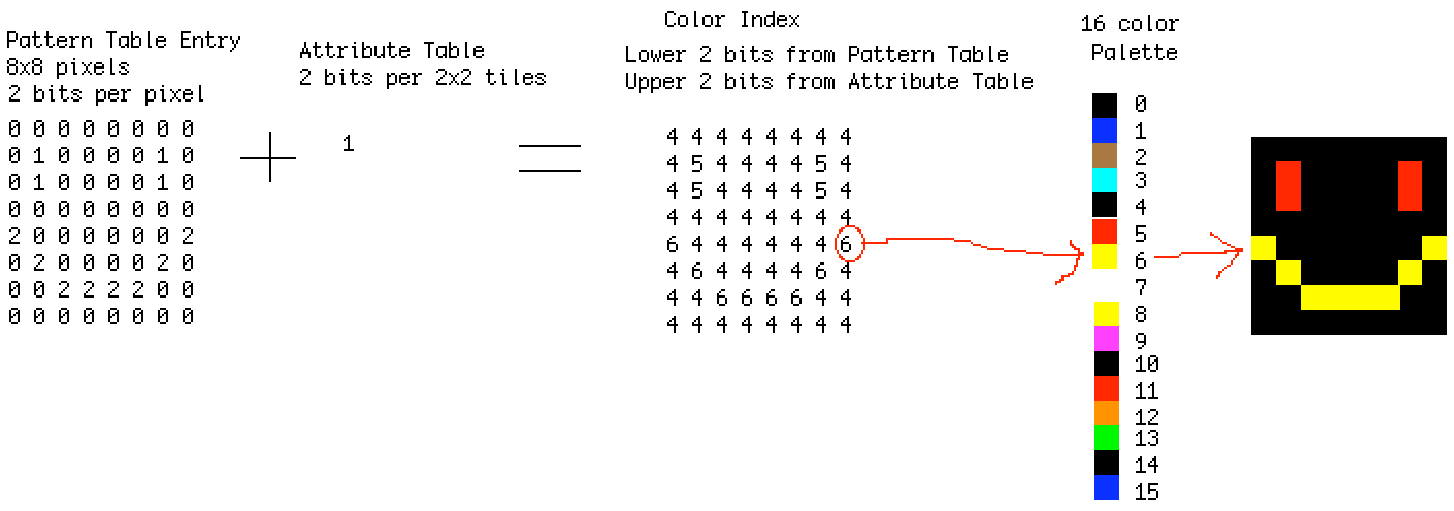
Pattern Tables

These are where the actual tile data is stored. It is either ROM or RAM on the cart. Each pattern table holds 256 tiles. One table is used for backgrounds, and the other for sprites.

Attribute Tables
These tables set the color information in 2x2 tile sections. This means that a 16x16 pixel area can only have 4 different colors selected from the palette.

Palettes
These two areas hold the color information, one for the background and one for sprites. Each palette has 16 colors.

To display a tile on screen, the pixel color index is taken from the Pattern Table and the Attribute Table. That index is then looked up in the Palette to get the actual color.


Pattern Table Entry
8x8 pixels
2 bits per pixel

```
0 0 0 0 0 0 0 0
0 1 0 0 0 0 1 0
0 1 0 0 0 0 1 0
0 0 0 0 0 0 0 0
2 0 0 0 0 0 0 2
0 2 0 0 0 0 2 0
0 0 2 2 2 2 0 0
0 0 0 0 0 0 0 0
```

$+$

Attribute Table
2 bits per 2x2 tiles

1

$=$

Color Index

Lower 2 bits from Pattern Table
Upper 2 bits from Attribute Table

```
4 4 4 4 4 4 4 4
4 5 4 4 4 4 5 4
4 5 4 4 4 4 5 4
4 4 4 4 4 4 4 4
6 4 4 4 4 4 4 6
4 6 4 4 4 4 6 4
4 4 6 6 6 6 4 4
4 4 4 4 4 4 4 4
```

16 color
Palette

⬛	0
🟦	1
🟫	2
🟦	3
⬛	4
🟥	5
🟨	6
⬜	7
🟨	8
🟪	9
⬛	10
🟥	11
🟧	12
🟩	13
⬛	14
🟦	15

6502 Assembly

Bit - The smallest unit in computers. It is either a 1 (on) or a 0 (off), like a light switch.

Byte - 8 bits together form one byte, a number from 0 to 255. Two bytes put together is 16 bits, forming a number from 0 to 65535. Bits in the byte are numbered starting from the right at 0.

Instruction - one command a processor executes. Instructions are run sequentially.

Overview

This section is only a quick overview of the 6502, not a detailed explanation. Much better explanations can be found at sites like http://www.obelisk.demon.co.uk/6502/

Binary/Hex

Before starting programming you should understand binary and hex encoding. The Windows Calculator app, set to Scientific mode, can quickly translate between these encodings.

Binary (bin) - Base 2, every digit is a 0 or 1. In NESASM this is used with a % like %00101110. Because this is an 8 bit data system there will be 8 digits. The bits are numbered starting from 0 on the right side.

```
  0 = %00000000
  1 = %00000001   D7-1 = 0, D0 = 1
  2 = %00000010
  3 = %00000011
128 = %10000000
255 = %11111111
```

Hexadecimal (hex) - Base 16, every digit is 0-F. In NESASM this is used with a $ like $8B. Each hex digit is 4 bits, so the 8 bit data is 2 hex digits.

```
  0 = $00
  1 = $01
  2 = $02
  9 = $09
 10 = $0A
 11 = $0B
 12 = $0C
 13 = $0D
 14 = $0E
 15 = $0F
 16 = $10
 17 = $11
128 = $80
255 = $FF
```

Addresses are 16 bits, or 4 hex digits.

```
$0000 = start of memory space
$8000 = start of PRG
$FFFF = last byte in memory space
```

6502 Registers

The 6502 has three 8 bit registers and a status register that you will be using. All your data processing uses these registers. There are additional registers that are not covered in this tutorial.

Accumulator

The Accumulator (A) is the main 8 bit register for loading, storing, comparing, and doing math on data. Some of the most frequent operations are:

```
LDA #$FF  ;load the hex value $FF (decimal 256) into A
STA $0000 ;store the accumulator into memory location $0000, internal RAM
```

Index Register X

The Index Register X (X) is another 8 bit register, usually used for counting or memory access. In loops you will use this register to keep track of how many times the loop has gone, while using A to process data. Some frequent operations are:

```
LDX $0000 ;load the value at memory location $0000 into X
INX       ;increment X   X = X + 1
```

Index Register Y

The Index Register Y (Y) works the same as X. Some operations are:

```
STY $00BA ;store Y into memory location $00BA
TYA       ;transfer Y into Accumulator
```

Status Register

The Status Register holds flags with information about the last instruction. For example when doing a subtract you can check if the result was a zero.

Code Layout

In assembly language there are 5 main parts. Some parts must be in a specific horizontal position for the assembler to use them correctly.

Directives

Directives are commands you send to the assembler to do things like locating code in memory. They start with a . and are indented. Some people use tabs, or 4 spaces, and I use 2 spaces. This sample directive tells the assembler to put the code starting at memory location $8000:

```
  .org $8000
```

Label

The label is aligned to the far left and has a : at the end. The label is just something you use to organize your code. The assembler translates the label into an address. Sample label:

```
  .org $8000
MyFunction:
```

Opcode
The opcode is the instruction that the processor will run, and is indented like the directives. In this sample, JMP is the opcode that tells the processor to jump to the MyFunction label:

```
  .org $8000
MyFunction:
  JMP MyFunction
```

Operands
The operands are additional information for the opcode. Opcodes have between one and three operands. In this example the #$FF is the operand:

```
  .org $8000
MyFunction:
  LDA #$FF
  JMP MyFunction
```

Comments
Comments are to help you understand in English what the code is doing. When you write code and come back later, the comments will save you. You do not need a comment on every line, but should have enough to explain what is happening. Comments start with a ; and are completely ignored by the assembler. They can be put anywhere horizontally, but are usually spaced beyond the long lines.

```
  .org $8000
MyFunction:        ; loads FF into accumulator
  LDA #$FF
  JMP MyFunction
```

This code would just continually run the loop, loading the hex value $FF into the accumulator each time.

NES Code Structure

Getting Started
This section has a lot of information because it will get everything set up to run your first NES program. Much of the code can be copy/pasted then ignored for now. The main goal is to just get NESASM to output something useful.

iNES Header
The 16 byte iNES header gives the emulator all the information about the game including mapper, graphics mirroring, and PRG/CHR sizes. You can include all this inside your asm file at the very beginning.

```
  .inesprg 1   ; 1x 16KB bank of PRG code
  .ineschr 1   ; 1x 8KB bank of CHR data
  .inesmap 0   ; mapper 0 = NROM, no bank swapping
  .inesmir 1   ; background mirroring
```

Banking
NESASM arranges everything in 8KB code and 8KB graphics banks. To fill the 16KB PRG space 2 banks are needed. Like most things in computing, the numbering starts at 0. For each bank you have to tell the assembler where in memory it will start.

```
  .bank 0
  .org $C000
;some code here

  .bank 1
  .org $E000
; more code here

  .bank 2
  .org $0000
; graphics here
```

Vectors
There are three times when the NES processor will interrupt your code and jump to a new location. These vectors, held in PRG ROM tell the processor where to go when that happens. Only the first two will be used in this tutorial.

NMI Vector - this happens once per video frame, when enabled. The PPU tells the processor it is starting the VBlank time and is available for graphics updates.

RESET Vector - this happens every time the NES starts up, or the reset button is pressed.

IRQ Vector - this is triggered from some mapper chips or audio interrupts and will not be covered.

These three must always appear in your assembly file in this order:

```
  .bank 1
  .org $FFFA     ;first of the three vectors starts here
  .dw NMI        ;when an NMI happens (once per frame if enabled) the
                   ;processor will jump to the label NMI:
  .dw RESET      ;when the processor first turns on or is reset, it will jump
                   ;to the label RESET:
  .dw 0          ;external interrupt IRQ is not used in this tutorial
```

Adding Binary Files

Additional data files are frequently used for graphics data or level data. The incbin directive can be used to include that data in your .NES file.

```
  .bank 2
  .org $0000
  .incbin "mario.chr"   ;includes 8KB graphics file from SMB1
```

Reset Code

There are some things you must do when the NES starts up. Some modes are set, RAM is cleared out, and you have to wait for the PPU to start up. All of this is needed, but you do not need to understand it for now:

```
  .bank 0
  .org $C000
RESET:
  SEI        ; disable IRQs
  CLD        ; disable decimal mode
  LDX #$40
  STX $4017  ; disable APU frame IRQ
  LDX #$FF
  TXS        ; Set up stack
  INX        ; now X = 0
  STX $2000  ; disable NMI
  STX $2001  ; disable rendering
  STX $4010  ; disable DMC IRQs


vblankwait1:       ; First wait for vblank to make sure PPU is ready
  BIT $2002
  BPL vblankwait1

clrmem:
  LDA #$00
  STA $0000, x
  STA $0100, x
  STA $0200, x
  STA $0400, x
  STA $0500, x
  STA $0600, x
  STA $0700, x
  LDA #$FE
  STA $0300, x
  INX
  BNE clrmem

vblankwait2:       ; Second wait for vblank, PPU is ready after this
  BIT $2002
  BPL vblankwait2
```

Completing The Program

Your first program will be very exciting, displaying an entire screen of one color! To do this the first PPU settings need to be written. This is done to memory address $2001. The 76543210 is the bit number, from 7 to 0. Those 8 bits form the byte you will write to $2001.

```
PPUMASK ($2001)

76543210
||||||||
|||||||+- Grayscale (0: normal color; 1: AND all palette entries
|||||||   with 0x30, effectively producing a monochrome display;
|||||||   note that colour emphasis STILL works when this is on!)
||||||+-- Disable background clipping in leftmost 8 pixels of screen
|||||+--- Disable sprite clipping in leftmost 8 pixels of screen
||||+---- Enable background rendering
|||+----- Enable sprite rendering
||+------ Intensify reds (and darken other colors)
|+------- Intensify greens (and darken other colors)
+-------- Intensify blues (and darken other colors)
```

So if you want to enable the sprites, you set bit 3 to 1. For this app bits 7, 6, 5 will be used to set the screen color:

```
  LDA %10000000   ;intensify blues
  STA $2001
Forever:
  JMP Forever     ;infinite loop
```

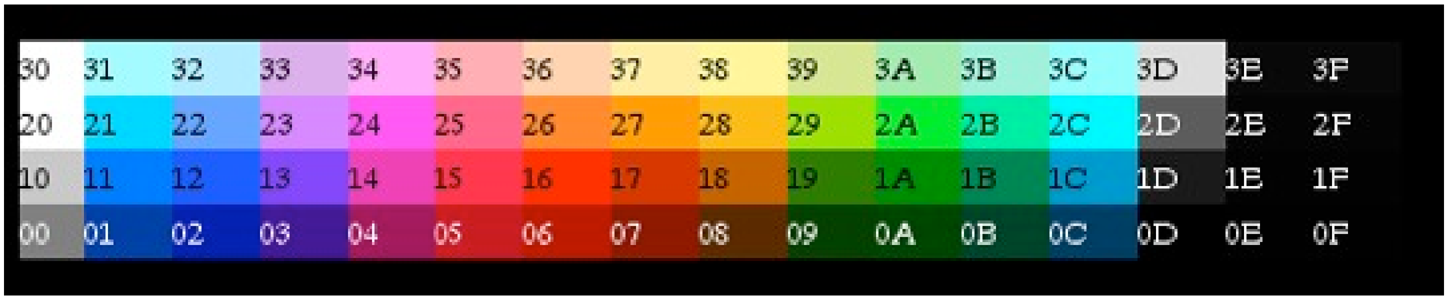
Putting It All Together

Download and unzip the background.zip sample files. All the code above is in the background.asm file. Make sure that file, mario.chr, and background.bat is in the same folder as NESASM, then double click on background.bat. That will run NESASM and should produce background.nes. Run that NES file in FCEUXD SP to see your background color! Edit background.asm to change the intensity bits 7-5 to make the background red or green.

You can start the Debug... from the Tools menu in FCEUXD SP to watch your code run. Hit the Step Into button, choose Reset from the NES menu, then keep hitting Step Into to run one instruction at a time. On the left is the memory address, next is the hex opcode that the 6502 is actually running. This will be between one and three bytes. After that is the code you wrote, with the comments taken out and labels translated to addresses. The top line is the instruction that is going to run next. Try figuring out what the instruction will do before hitting Step Into, then see if you were right.

Palettes

Before putting any graphics on screen, you first need to set the color palette. There are two separate palettes, each 16 bytes. One is used for the background, and the other for sprites. The byte in the palette corresponds to one of the 52 base colors the NES can display. $0D is a bad color and should not be used.



30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

The palettes start at PPU address $3F00 and $3F10. To set this address, PPU port $2006 is used. This port must be written twice, once for the high byte then for the low byte:

```
  LDA $2002    ; read PPU status to reset the high/low latch
  LDA #$3F
  STA $2006    ; write the high byte of $3F10 address
  LDA #$10
  STA $2006    ; write the low byte of $3F10 address
```

Now the PPU data port at $2007 is ready to accept data. The first write will go to the address you set ($3F00), then the PPU will increment the address ($3F01). You can keep writing data and it will keep incrementing. This sets the first 4 colors in the palette:

```
  LDA #$32   ;light blueish
  STA $2007
  LDA #$14   ;pinkish
  STA $2007
  LDA #$2A   ;greenish
  STA $2007
  LDA #$16   ;redish
  STA $2007
```

You would continue to do writes to fill out the rest of the palette. Fortunately there is a smaller way to write all that code. First you can use the .db directive to store data bytes.

```
PaletteData:
  .db $0F,$31,$32,$33,$34,$35,$36,$37,$38,$39,$3A,$3B,$3C,$3D,$3E,$0F
  .db $0F,$1C,$15,$14,$31,$02,$38,$3C,$0F,$1C,$15,$14,$31,$02,$38,$3C
```

Then a loop is used to copy those bytes to the palette in the PPU. The X register is used as an index into the palette, and used to count how many times the loop has repeated. You want to copy 32 bytes, so the loop starts at 0 and counts up to 32.

```
  LDX #$00                 ; start out at 0
LoadPalettesLoop:
  LDA PaletteData, x       ; load data from address (PaletteData + the value in x)
                           ; 1st time through loop it will load PaletteData+0
                           ; 2nd time through loop it will load PaletteData+1
                           ; 3rd time through loop it will load PaletteData+2
                           ; etc
  STA $2007                ; write to PPU
  INX                      ; X = X + 1
  CPX #$20                 ; Compare X to hex $20, decimal 32
  BNE LoadPalettesLoop     ; Branch to LoadPalettesLoop if compare was Not Equal to zero
                           ; if compare was equal to 32, keep going down
```

Sprites

Anything that moves separately from the background will be made of sprites. A sprite is just an 8x8 pixel tile that the PPU renders anywhere. The PPU has enough internal memory for 64 sprites.

Sprite DMA

The fastest and easiest way to transfer your sprites to the sprite memory is using DMA (direct memory access). This just means a block of RAM is copied from CPU memory to the PPU. The on board RAM space from $0200-02FF is usually used for this purpose. To start the transfer, two bytes need to be written to the PPU ports:

```
  LDA #$00
  STA $2003  ; set the low byte (00) of the RAM address
  LDA #$02
  STA $4014  ; set the high byte (02) of the RAM address, start the transfer
```

Like all graphics updates, this needs to be done at the beginning of the VBlank period, so it will go in the NMI section.

Sprite Data

Each sprite needs 4 bytes of data for its position and tile information in this order:

Y Position - vertical position of the sprite on screen. $00 is the top of the screen. Anything above $EF is off the bottom of the screen.

Tile Number - this is the tile number (0 to 256) for the graphic to be taken from a Pattern Table.

Attributes - this byte holds color and displaying information:

```
  76543210
  |||   ||
  |||   ++- Palette (4 to 7) of sprite
  |||
  ||+------ Priority (0: in front of background; 1: behind background)
  |+------- Flip sprite horizontally
  +-------- Flip sprite vertically
```

X Position - horizontal position on the screen. $00 is the left side, anything above $F9 is off screen.

If you want to edit sprite 0, you change bytes $0300-0303. Sprite 1 is $0304-0307, sprite 2 is $0308-030B, etc

Turning NMI/Sprites On

The PPU port $2001 is used again to enable sprites. Setting bit 4 to 1 will make them appear.

NMI also needs to be turned on, so the Sprite DMA will run and the sprites will be copied. This is done with the PPU port $2000. The Pattern Table 1 is also selected to choose sprites from.

```
PPUCTRL ($2000)

76543210
| ||||||
| ||||++- Base nametable address
| ||||    (0 = $2000; 1 = $2400; 2 = $2800; 3 = $2C00)
| |||+--- VRAM address increment per CPU read/write of PPUDATA
| |||     (0: increment by 1, going across; 1: increment by 32, going down)
| ||+---- Sprite pattern table address for 8x8 sprites (0: $0000; 1: $1000)
| |+----- Background pattern table address (0: $0000; 1: $1000)
| +------ Sprite size (0: 8x8; 1: 8x16)
|
+-------- Generate an NMI at the start of the
          vertical blanking interval (0: off; 1: on)
```

And the code:

```
LDA #$80
STA $0200        ;put sprite 0 in center of screen
STA $0203        ;put sprite 0 in center of screen
LDA #$00
STA $0201        ;tile number = 0
STA $0202        ;color = 0, no flipping

LDA #%10000000   ; enable NMI, sprites from Pattern Table 0
STA $2000

LDA #%00010000   ; no intensify, enable sprites
STA $2001
```

Putting It All Together

Download and unzip the sprites.zip sample files. All the code above is in the sprites.asm file. Make sure that file, mario.chr, and sprites.bat is in the same folder as NESASM, then double click on sprites.bat. That will run NESASM and should produce sprites.nes. Run that NES file in FCEUXD SP to see your sprite! Tile number 0 is the back of Mario's head and hat, can you see it? Edit sprites.asm to change the sprite position, or to change the color palette.

You can choose the PPU viewer in FCEUXD SP to see both Pattern Tables, and both Palettes. You can also see that the color in Palette entry $3F00 gets copied to multiple places and used for the background color. Generally this color will be black or white in games.

Multiple Sprites

Another Block Copy

Instead of writing 4 LDA/STA lines of code for each sprite, you can use the .db directive like the palette data and use a loop to copy it all at once. First the data is set:

```
sprites:
     ;vert tile attr horiz
  .db $80, $32, $00, $80   ;sprite 0
  .db $80, $33, $00, $88   ;sprite 1
  .db $88, $34, $00, $80   ;sprite 2
  .db $88, $35, $00, $88   ;sprite 3
```

There are 4 bytes per sprite, each on one line. The bytes are in the correct order and easily changed. This is only the starting data, when the program is running the copy in RAM can be changed to move the sprite around.

Next you need the loop to copy the data into RAM. This loop also works the same way as the palette loading, with the X register as the loop counter.

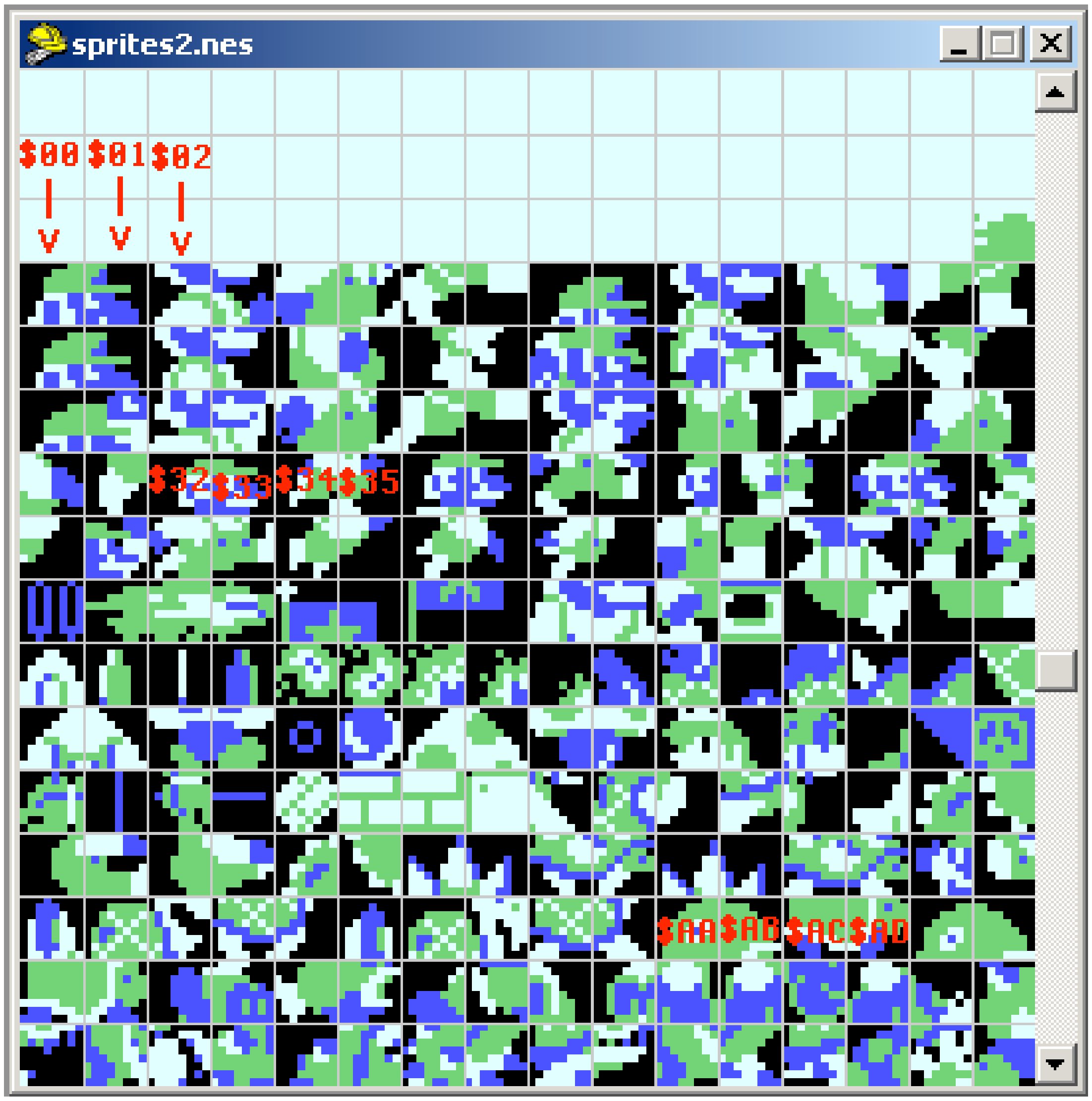
```
LoadSprites:
  LDX #$00              ; start at 0
LoadSpritesLoop:
  LDA sprites, x        ; load data from address (sprites +  x)
  STA $0200, x          ; store into RAM address ($0200 + x)
  INX                   ; X = X + 1
  CPX #$20              ; Compare X to hex $20, decimal 32
  BNE LoadSpritesLoop   ; Branch to LoadSpritesLoop if compare was Not Equal to zero
                        ; if compare was equal to 32, keep going down
```

If you wanted to add more sprites, you would add lines into the sprite .db section then increase the CPX compare value. That will run the loop more times, copying more bytes.

Putting It All Together

Download and unzip the sprites2.zip sample files. All the code above is in the sprites2.asm file. Make sure that file, mario.chr, and sprites2.bat is in the same folder as NESASM, then double click on sprites2.bat. That will run NESASM and should produce sprites2.nes. Run that NES file in FCEUXD SP to see small Mario! Try editing the color palette to get Marios colors correct. You can also change the horiz/vert values in the sprites data to move him around.

Start Tile Layer Pro, then open your sprites2.nes file. The static you see in the top left is your program code. Scroll down until you see the graphics tiles. First will be the 256 (16x16) sprite tiles, then the 256 background tiles. Use the scroll bar arrow to go to the bottom of the file, then back up to the sprites. Count the sprite tiles starting from 0 to see where the Mario $32-35 is. Keep counting and edit the sprite .db data to put a turtle or a beetle on screen.


sprites2.nes
$00
$01
$02
$32
$33
$34
$35
$AA
$AB
$AC
$AD

Reading Controllers

This section will show how to read from both controllers, and use that input to move sprites around.

Controller Ports

The controllers are accessed through memory addresses $4016 and $4017. First you have to write the value $01 then the value $00 to port $4016. This tells the controllers to latch the current button positions. Then you read from $4016 for first player or $4017 for second player. The buttons are sent one at a time, in bit 0. If bit 0 is 0, the button is not pressed. If bit 0 is 1, the button is pressed.

Button status for each controller is returned in the following order: A, B, Select, Start, Up, Down, Left, Right.

```
  LDA #$01
  STA $4016
  LDA #$00
  STA $4016     ; tell both the controllers to latch buttons

  LDA $4016     ; player 1 - A
  LDA $4016     ; player 1 - B
  LDA $4016     ; player 1 - Select
  LDA $4016     ; player 1 - Start
  LDA $4016     ; player 1 - Up
  LDA $4016     ; player 1 - Down
  LDA $4016     ; player 1 - Left
  LDA $4016     ; player 1 - Right

  LDA $4017     ; player 2 - A
  LDA $4017     ; player 2 - B
  LDA $4017     ; player 2 - Select
  LDA $4017     ; player 2 - Start
  LDA $4017     ; player 2 - Up
  LDA $4017     ; player 2 - Down
  LDA $4017     ; player 2 - Left
  LDA $4017     ; player 2 - Right
```

AND Instruction

Button information is only sent in bit 0, so we want to erase all the other bits. This can be done with the AND instruction. Each of the 8 bits is ANDed with the bits from another value. If the bit from both the first AND second value is 1, then the result is 1. Otherwise the result is 0.

```
0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1
```

For a full random 8 bit value:

```
      01011011
AND   10101101
--------------
      00001001
```

We only want bit 0, so that bit is set and the others are cleared:

```
      01011011    controller data
AND   00000001    AND value
--------------
      00000001    only bit 0 is used
```

So to erase all the other bits when reading controllers, the AND should come after each read from $4016 or $4017:

```
  LDA $4016       ; player 1 - A
  AND #%00000001

  LDA $4016       ; player 1 - B
  AND #%00000001

  LDA $4016       ; player 1 - Select
  AND #%00000001
```

BEQ instruction

The BNE instruction was used earlier in loops to Branch when Not Equal to a compared value. Here BEQ will be used without the compare instruction to Branch when EQual to zero. When a button is not pressed, the value will be zero, so the branch is taken. That skips over all the instructions that do something when the button is pressed:

```
ReadA:
  LDA $4016       ; player 1 - A
  AND #%00000001  ; only look at bit 0
  BEQ ReadADone   ; branch to ReadADone if button is NOT pressed (0)

                  ; add instructions here to do something when button IS pressed (1)

ReadADone:        ; handling this button is done
```

CLC/ADC instructions

For this demo we will use the player 1 controller to move the Mario sprite around. To do that we need to be able to add to values. The ADC instruction stands for Add with Carry. Before adding, you have to make sure the carry is cleared, using CLC. This sample will load the sprite position into A, clear the carry, add one to the value, then store back into the sprite position:

```
  LDA $0203   ; load sprite X (horizontal) position
  CLC         ; make sure the carry flag is clear
  ADC #$01    ; A = A + 1
  STA $0203   ; save sprite X (horizontal) position
```

SEC/SBC instructions

To move the sprite the other direction, a subtract is needed. SBC is Subtract with Carry. This time the carry has to be set before doing the subtract:

```
  LDA $0203   ; load sprite position
  SEC         ; make sure carry flag is set
  SBC #$01    ; A = A - 1
  STA $0203   ; save sprite position
```

Putting It All Together

Download and unzip the controller.zip sample files. All the code above is in the controller.asm file. Make sure that file, mario.chr, and controller.bat is in the same folder as NESASM, then double click on controller.bat. That will run NESASM and should produce controller.nes. Run that NES file in FCEUXD SP to see small Mario. Press the A and B buttons on the player 1 controller to move one sprite of Mario. The movement will be one pixel per frame, or 60 pixels per second on NTSC machines. If Mario isn't moving, make sure your controls are set up correctly in the Config menu under Input... If you hold both buttons together, the value will be added then subtracted so no movement will happen.

Try editing the ADC and SBC values to make him move faster. The screen is only 256 pixels across, so too fast and he will just jump around randomly! Also try editing the code to move each of the 4 sprites together.

Variables

To store the button information for later use, a variable is created. First the space for the variable is reserved in RAM:

```
    .rsset  $0000   ; start the reserve counter at memory address $0000
Buttons1  .rs 1     ; reserve one byte of space
```

Now in your code you can do STA Buttons1, which will store the accumulator into that memory space. You can also do LDA Buttons1 to load that memory value into the accumulator.