Number Systems

Decimal
The decimal system is base 10. Every digit can be 0-9. Each digit place is a power of 10. Each digit place to the left is 10 times more than the previous digit place. If you take the number 10 and put a 0 to the right, it becomes 100 which is 10 times more. Remove the 0 from the right, it becomes 1 which is 10 times less.

100's place	10's place	1's place	
0	0	1	= 001
0	1	0	= 010
1	0	0	= 100

To get the value of a number, you multiply each digit by it's place value and add them all together.

100's place	10's place	1's place	
3	8	0	= 3*100 + 8*10 + 0*1 = 380
0	4	1	= 0*100 + 4*10 + 1*1 = 41

Binary
Everything in computers is done in base 2, binary. This is because the lowest level of computing is a switch; on/off, 1/0.

Base 2 binary works the same way, except each digit can be 0-1 and the place values are powers of 2 instead of 10. Insert a 0 to the right of a number and it becomes 2 times bigger. Remove a 0 and it becomes 10 times smaller.

8's place	4's place	2's place	1's place	
0	1	0	0	= 0*8 + 1*4 + 0*2 + 0*1 = 4
1	1	1	1	= 1*8 + 1*4 + 1*2 + 1*1 = 15

The NES is an 8 bit system, which means the binary number it works with are 8 binary digits long. 8 bits is one byte. Some examples are:

Binary		Decimal
00000000	=	0
00001111	=	15
00010000	=	16
10101010	=	170
11111111	=	255

Eventually you become fast at reading binary numbers, or at least recognizing patterns. You can see that one byte can only range from 0-255. For numbers bigger than that you must use 2 or more bytes. There are also no negative numbers. More on that later.

Hexadecimal
Hexadecimal or Hex is base 16, so each digit is 0-15 and each digit place is a power of 16. The problem is anything 10 and above needs 2 digits. To fix this letters are used instead of numbers starting with A:

Decimal		Hex
0	=	0
1	=	1
9	=	9
10	=	A
11	=	B
12	=	C
13	=	D
14	=	E
15	=	F

As with decimal and hex the digit places are each a power of 16:

```
 16's place   1's place
     6            A      = 6*16 + A(10)*1 = 106
     1            0      = 1*16 +      0*1 = 16
```

Hex is largely used because it is much faster to write than binary. An 8 digit binary number turns into a 2 digit hex number:

```
Binary    01101010
split       | |
in half    /   \
         0110   1010
into      |     |
 hex      6     A
          |     |
 put       \   /
 back       6A

 01101010 = 6A
```

And more examples:

```
Binary    Hex   Decimal
00000000 = 00  = 0
00001111 = 0F  = 15
00010000 = 10  = 16
10101010 = AA  = 170
11111111 = FF  = 255
```

For easy converting open up the built in Windows calculator and switch it to scientific mode. Choose the base (Hex, Dec, or Bin), type the number, then switch to another base.

When the numbers are written an extra character is added so you can tell which base is being used. Binary is typically prefixed with a %, like %00001111. Hex is prefixed with a $ like $2A. Some other conventions are postfixing binary with a b like 00001111b and postfixing hex with an h like 2Ah.

The NES has a 16 bit address bus (more on that later), so it can access 2^16 bytes of memory. 16 binary digits turns into 4 hex digits, so typical NES addresses look like $8000, $FFFF, and $4017.

Core Programming Concepts

All programming languages have three basic concepts. They are instructions, variables, and control flow. If any of those three are missing it is no longer a true programming language. For example HTML has no control flow so it is not a programming language.

Instructions
An instruction is the smallest command that the processor runs. Instructions are run one at a time, one after another. In the NES processor there are only 56 instructions. Typically around 10 of those will be used constantly, and at least 10 will be completely ignored. Some examples of these would be addition, loading a number, or comparing a variable to zero.

Variables
A variable is a place that stores data that can be modified. An example of this would be the vertical position of Mario on the screen. It can be changed any time during the game. Variables in source code all have names you set, so it would be something like MarioHorizPosition.

Control Flow

Normally your instructions run in sequential order. Sometimes you will want to run a different section of code depending on a variable. This would be a control flow statement which changes the normal flow of your program. An example would be if Mario is falling, jump to the code that checks if he hit the ground yet.

NEXT WEEK: basic NES architecture

This week: general overview of the NES architecture with the major components covered. All general purpose computers are arranged the same way with a place to store code (ROM), a place to store variables (RAM), and a processor to run code (CPU). The NES also adds another processor to generate the graphics (PPU) and a section of the CPU to generate audio (APU). Everything here is very general and will have more details than you want in the next few weeks.

NES System Architecture

KB - Memory size is listed in KiloBytes or KB. 1KB = 1024 bytes. Everything is powers of 2, so 2^10 = 1024 is used instead of 1000. If the capitalization is different, the meaning can change. Kb is Kilobits. Divide Kb by 8 to get KB, because 1 byte = 8 bits.

ROM - Read Only Memory, holds data that cannot be changed. This is where the game code or graphics is stored on the cart.

RAM - Random Access Memory, holds data that can be read and written. When power is removed, the chip is erased. A battery can be used to keep power and data valid.

PRG - Program memory, the code for the game

CHR - Character memory, the data for graphics

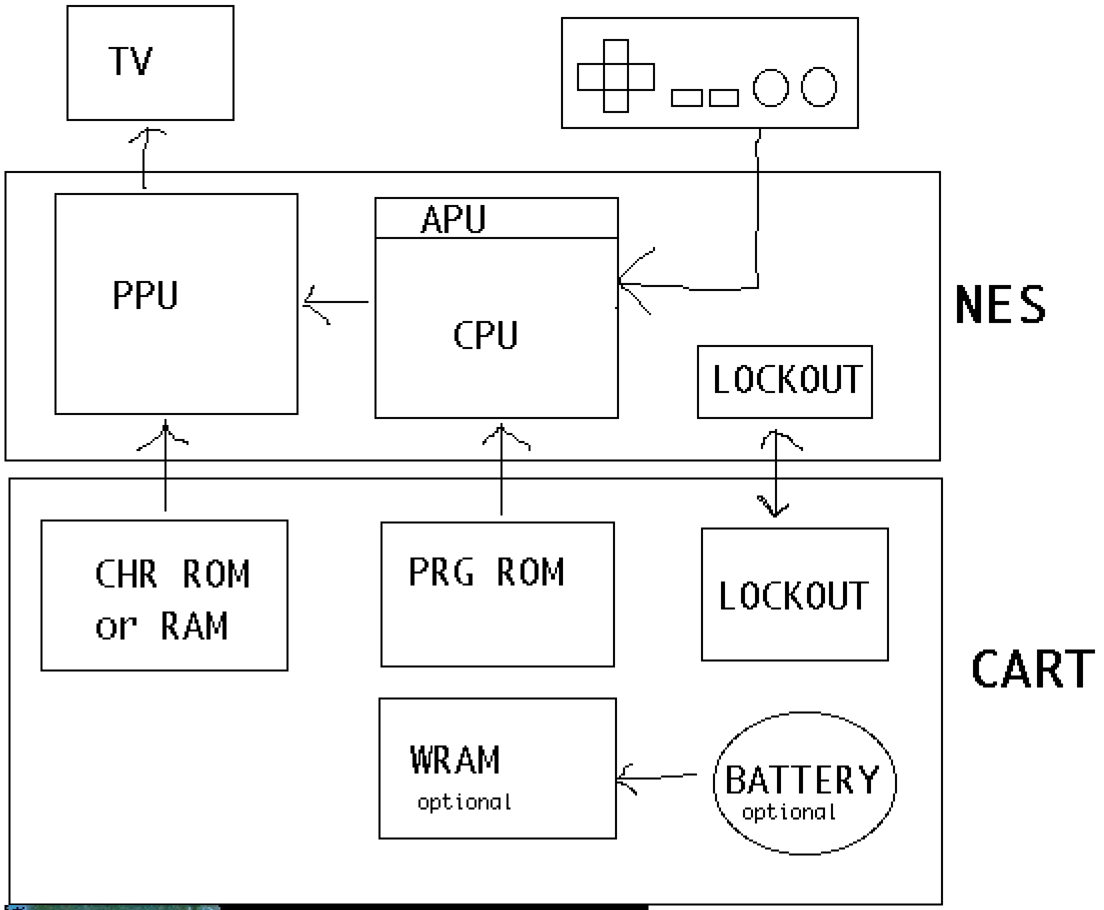
CPU - Central Processing Unit, the main processor chip

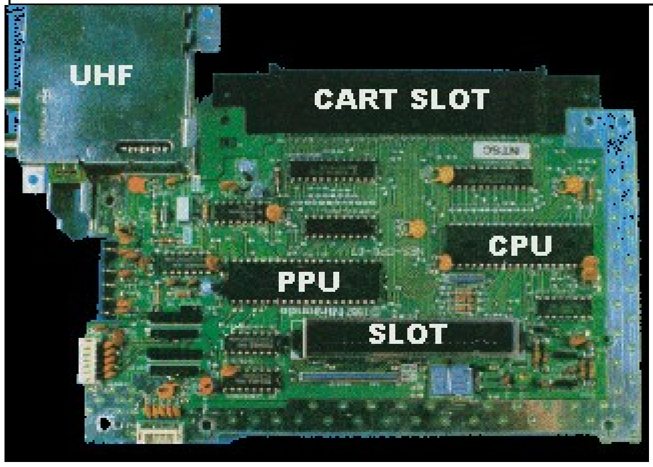
PPU - Picture Processing Unit, the graphics chip

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

System Overview

The NES includes a custom 6502 based CPU with the APU and controller handling inside one chip, and a PPU that displays graphics in another chip. Your code instructions run on the CPU and sends out commands to the APU and PPU. The NOAC (NES On A Chip) clones like the Yobo and NEX 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 TV screens of background graphics. Some carts add extra CPU RAM, called Work RAM or WRAM. If a cart needs to store saved games, this WRAM will have a battery attached to make sure it isn't erased. A few carts add extra PPU RAM to hold four screens of background graphics at once. This is not common. The rest of this tutorial will not use WRAM or four screen RAM.

Each cart includes at least three chips. One holds the program code (PRG), another holds the character graphics (CHR), and the last is the lockout. The graphics chip can be RAM instead of ROM, which means the game code would copy graphics from the PRG ROM chip to the CHR RAM. PRG is always a ROM chip.

Lockout Chip
Inside the NES and the cart are also two lockout chips. The lockout chip controls resetting the console. First the NES lockout sends out a stream ID, 0-15. The cart lockout records this number. Then both lockout chips run a complex equation using that number and send the results to each other. Both chips know what the other is supposed to send so

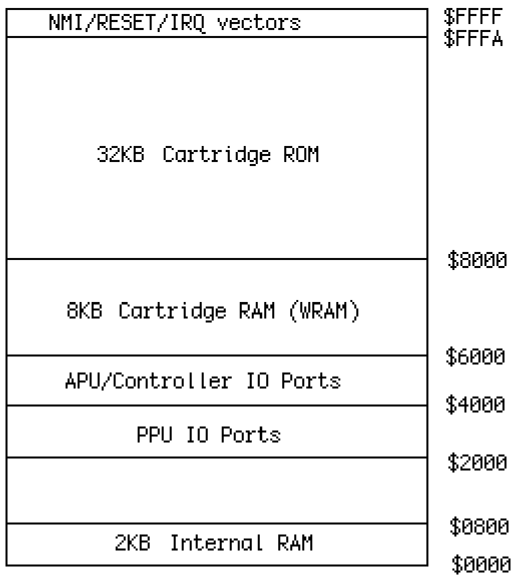
they both know when something is wrong. If that happens the system enters the continuous reseting loop. This is the screen flashing you see with a dirty cart.

When you cut pin 4 of the NES lockout chip, you are making it think it is inside the cart. It sits there waiting for the ID from the NES which never happens, so the system is never reset. If you were to completely remove the NES lockout chip the system would not work because it controls the reset button.

Most lockout defeaters used by the unlicensed game companies used large voltage spikes sent from the cart to the NES lockout. When timed right those would crash the NES lockout, preventing it from resetting the system. Nintendo slowly added protection against those on the NES board. Next time you open your NES, check the board for the revision number. Right in the middle it will say NES-CPU- then a number. That number is the revision. If you have 05 it is an early one. 07 and 09 added some lockout protection. 11 was the last version with the most lockout protection. Almost all unlicensed carts that use lockout defeaters will not work on a NES-CPU-11 system.

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.

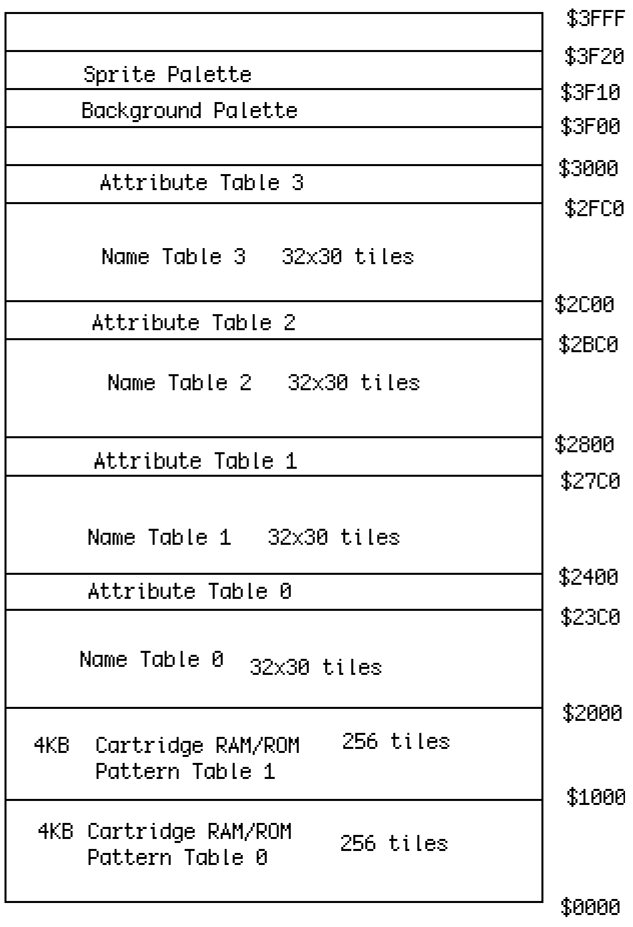
The CPU has a 16 bit address bus which can access up to 64KB of memory. 2^16 = 65536, or 64KB. Included in that memory space is the 2KB of CPU RAM, ports to access PPU/APU/controllers, WRAM (if on the cart), and 32KB for PRG ROM. The 16 bit addresses are written in hex, so they become 4 digits starting with a $ symbol. For example the internal RAM is at $0000-0800. $0800 = 2048 or 2KB. 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. Mappers like the MMC3 allowed up to 512KB of PRG, and 256KB of CHR. There is no limit to the memory size if you create a new mapper chip, but 128KB PRG and 64KB CHR was the most common size.



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

Your program does not run on the PPU, the PPU always goes through the same display order. You only set some options like colors and scrolling. The PPU processes one TV scanline at a time. First the sprites are fetched from the cart CHR memory. If there are more than 8 sprites on the scanline the rest are ignored. This is why some games like Super Dodge Ball will blink when there is lots happening on screen. After the sprites the background is fetched from CHR memory. When all the scanlines are done there is a period when no graphics are sent out. This is called VBlank and is the only time graphics updates can be done. PAL has a longer VBlank time (when the TV cathode ray gun is going back to the top of the screen) which allows more time for graphics updates. Some PAL games and demos do not run on NTSC systems because of this difference in VBlank time. 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. TV variations will cut off an additional 0-8 rows, so you should allow for a border before drawing important information.

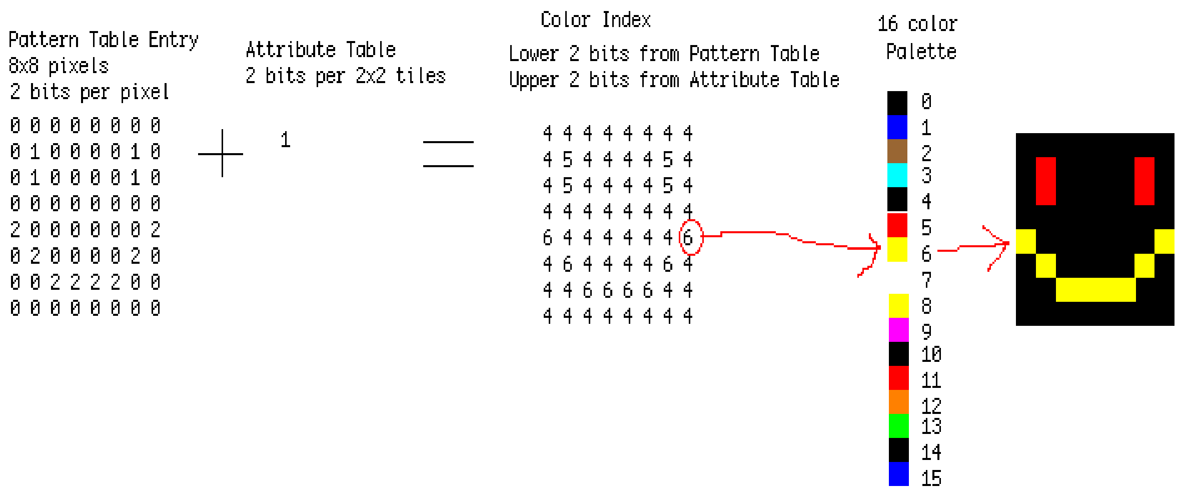
NTSC runs at 60Hz and PAL runs at 50Hz. Running an NTSC game on a PAL system will be slower because of this timing difference. Sounds will also be slower.



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. The tile system means less memory is needed (was expensive at the time) but also means that things like bitmap pictures and 3d graphics aren't really possible. To see all the tiles in a game, download Tile Molester and open up your .NES file. Scroll down until you see graphics that don't look like static. You can see that small tiles are arranged by the game to make large images.
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, and there is enough internal RAM to hold 2 screens. When games scroll the background graphics are updated off screen before they are scrolled on screen.
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. All graphics currently on screen must be in these tables.
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.



```
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
```

+ 1 =

```
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
```

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

To see all the graphics, download the FCEUXD SP emulator. Open up your .NES game and choose PPU Viewer from the Tools menu. This will show you all the active background tiles, all the active sprite tiles, and the color palettes. Then choose Name Table Viewer from the Tools menu. This will show you the backgrounds as they will appear on screen. If you choose a game that scrolls like SMB you can see the off screen background sections being updated.

NEXT WEEK: CPU details, start of 6502 assembly programming

This Week: starts getting into more details about the 6502 and intro to assembly language. The lessons for asm usage and NES specifics will be done in sections together. There are many other 6502 websites and good books which may help you learn better.

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.

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, which is inside the game ROM area:

```
  .org $8000
```

Labels
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 and make it easier to read. The assembler translates the label into an address. Sample label:

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

When the assembler runs, it will do a find/replace to set MyFunction to $8000. The if you have any code that uses MyFunction like:

```
  STA MyFunction
```

It will find/replace to:

```
  STA $8000
```

Opcodes
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.

6502 Processor Overview
The 6502 is an 8 bit processor with a 16 bit address bus. It can access 64KB of memory without bank switching. In the NES this memory space is split up into RAM, PPU/Audio/Controller access, and game ROM.

```
$0000-0800 – Internal RAM, 2KB chip in the NES
$2000-2007 – PPU access ports
$4000-4017 – Audio and controller access ports
$6000-7FFF - Optional WRAM inside the game cart
$8000-FFFF - Game cart ROM
```

Any of the game cart sections can be bank switched to get access to more memory, but memory mappers will not be included in this tutorial.

6502 Assembly Overview
The assembly language for 6502 starts with a 3 character code for the instruction "opcode". There are 56 instructions, 10 of which you will use frequently. Many instructions will have a value after the opcode, which you can write in decimal or hex. If that value starts with a # then it means use the actual number. If the value doesn't have then # then it means use the value at that address. So LDA #$05 means load the value 5, LDA $0005 means load the value that is stored at address $0005.

6502 Registers
A register is a place inside the processor that holds a value. 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 almost the same as X. Some instructions (not covered here) only work with X and not Y. 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.

6502 Instruction Set

These are just the most common and basic instructions. Most have a few different options which will be used later. There are also a few more complicated instructions to be covered later.

Common Load/Store opcodes

```
LDA #$0A   ; LoaD the value 0A into the accumulator A
           ;  the number part of the opcode can be a value or an address
           ;  if the value is zero, the zero flag will be set.
LDX $0000  ; LoaD the value at address $0000 into the index register X
           ;  if the value is zero, the zero flag will be set.
LDY #$FF   ; LoaD the value $FF into the index register Y
           ;  if the value is zero, the zero flag will be set.
STA $2000  ; STore the value from accumulator A into the address $2000
           ;  the number part must be an address
STX $4016  ; STore value in X into $4016
           ;  the number part must be an address
STY $0101  ; STore Y into $0101
           ;  the number part must be an address
TAX        ; Transfer the value from A into X
           ;  if the value is zero, the zero flag will be set
TAY        ; Transfer A into Y
           ;  if the value is zero, the zero flag will be set
TXA        ; Transfer X into A
           ;  if the value is zero, the zero flag will be set
TYA        ; Transfer Y into A
           ;  if the value is zero, the zero flag will be set
```

Common Math opcodes

```
ADC #$01   ; ADd with Carry
           ;  A = A + $01 + carry
           ;  if the result is zero, the zero flag will be set
SBC #$80   ; SuBtract with Carry
           ;  A = A - $80 - (1 - carry)
           ;  if the result is zero, the zero flag will be set
CLC        ; CLear Carry flag in status register
           ;  usually this should be done before ADC
SEC        ; SEt Carry flag in status register
           ;  usually this should be done before SBC
INC $0100  ; INCrement value at address $0100
           ;  if the result is zero, the zero flag will be set
DEC $0001  ; DECrement $0001
           ;  if the result is zero, the zero flag will be set
INY        ; INcrement Y register
           ;  if the result is zero, the zero flag will be set
INX        ; INcrement X register
           ;  if the result is zero, the zero flag will be set
DEY        ; DEcrement Y
           ;  if the result is zero, the zero flag will be set
DEX        ; DEcrement X
           ;  if the result is zero, the zero flag will be set
ASL A      ; Arithmetic Shift Left
           ;  shift all bits one position to the left
           ;  this is a multiply by 2
           ;  if the result is zero, the zero flag will be set
LSR $6000  ; Logical Shift Right
           ;  shift all bits one position to the right
           ;  this is a divide by 2
           ;  if the result is zero, the zero flag will be set
```

Common Comparison opcodes

```
CMP #$01   ; CoMPare A to the value $01
                ;  this actually does a subtract, but does not keep the result
                ;  instead you check the status register to check for equal,
                ;  less than, or greater than
CPX $0050  ; ComPare X to the value at address $0050
CPY #$FF   ; ComPare Y to the value $FF
```

Common Control Flow opcodes

```
JMP $8000  ; JuMP to $8000, continue running code there
BEQ $FF00  ; Branch if EQual, contnue running code there
                ;  first you would do a CMP, which clears or sets the zero flag
                ;  then the BEQ will check the zero flag
                ;  if zero is set (values were equal) the code jumps to $FF00 and runs there
                ;  if zero is clear (values not equal) there is no jump, runs next instruction
BNE $FF00  ; Branch if Not Equal - opposite above, jump is made when zero flag is clear
```

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 (ignore for now)
```

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
```

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. This data will not be used yet, but is needed to make the .NES file size match the iNES header.

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

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 the right order. The .dw directive is used to define a Data Word (1 word = 2 bytes):

```
  .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
```

Reset Code
The reset vector was set to the label RESET, so when the processor starts up it will start from RESET: Using the .org directive that code is set to a space in game ROM. A couple modes are set right at the beginning. We are not using IRQs, so they are turned off. The NES 6502 processor does not have a decimal mode, so that is also turned off. This section does NOT include everything needed to run code on the real NES, but will work with the FCEUXD SP emulator. More reset code will be added later.

```
    .bank 0
    .org $C000
RESET:
    SEI        ; disable IRQs
    CLD        ; disable decimal mode
```

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 program 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 NESASM3, then double click on background.bat. That will run NESASM3 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.

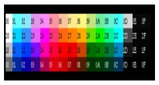
You can start the Debug... from the Tools menu in FCEUXD SP to watch your code run. Hit the Step Into button, choose Reset from the NES menu, then keep hitting Step Into to run one instruction at a time. On the left is the memory address, next is the hex opcode that the 6502 is actually running. This 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. So far there isn't much code, but the debugger will be very helpful later.

NEXT WEEK: more PPU details, start of graphics

This Week: now that you can make and run a program, time to put something on screen!

Palettes

Before putting any graphics on screen, you first need to set the color palette. There are two separate palettes, each 16 bytes. One palette is used for the background, and the other for sprites. The byte in the palette corresponds to one of the 64 base colors the NES can display. $0D is a bad color and should not be used. These colors are not exact and will look different on emulators and TVs.



The palettes start at PPU address $3F00 and $3F10. To set this address, PPU address 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 to high
    LDA #$3F
    STA $2006    ; write the high byte of $3F10 address
    LDA #$10
    STA $2006    ; write the low byte of $3F10 address
```

That code tells the PPU to set its address to $3F10. Now the PPU data port at $2007 is ready to accept data. The first write will go to the address you set ($3F10), then the PPU will automatically increment the address ($3F11, $3F12, $3F13) after each read or write. You can keep writing data and it will keep incrementing. This sets the first 4 colors in the palette:

```
    LDA #$32   ;code for light blueish
    STA $2007  ;write to PPU $3F10
    LDA #$14   ;code for pinkish
    STA $2007  ;write to PPU $3F11
    LDA #$2A   ;code for greenish
    STA $2007  ;write to PPU $3F12
    LDA #$16   ;code for redish
    STA $2007  ;write to PPU $3F13
```

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,$0F,$35,$36,$37,$0F,$39,$3A,$3B,$0F,$3D,$3E,$0F  ;background palette data
    .db $0F,$1C,$15,$14,$0F,$02,$38,$3C,$0F,$1C,$15,$14,$0F,$02,$38,$3C  ;sprite palette data
```

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 both palettes at once which is 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
```

Once that code finishes, the full color palette is ready. One byte or the whole thing can be changed while your program is running.

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 on the screen. Generally objects are made from multiple sprites next to each other. Examples would be Mario and any of the enemies like Goombas and Bowser. The PPU has enough internal memory for 64 sprites. This memory is separate from all other video memory and cannot be expanded.

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 sprite memory. 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
```

Once the second write is done the DMA transfer will start automatically. All data for the 64 sprites will be copied. Like all graphics updates, this needs to be done at the beginning of the VBlank period, so it will go in the NMI section of your code.

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

1 - 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.

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

3 - Attributes - this byte holds color and displaying information:

```
  76543210
  |||   ||
  |||   ++- Color Palette of sprite.  Choose which set of 4 from the 16 colors to use
  |||
  ||+------ Priority (0: in front of background; 1: behind background)
  |+------- Flip sprite horizontally
  +-------- Flip sprite vertically
```

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

Those 4 bytes repeat 64 times (one set per sprite) to fill the 256 bytes of sprite memory. If you want to edit sprite 0, you change bytes $0200-0203. Sprite 1 is $0204-0207, sprite 2 is $0208-020B, 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 every frame. This is done with the PPU port $2000. The Pattern Table 0 is also selected to choose sprites from. Background will come from Pattern Table 1 when that is added later.

```
  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 vblank (0: off; 1: on)
```

And the new code to set up the sprite data:

```
LDA #$80
STA $0200        ;put sprite 0 in center ($80) of screen vertically
STA $0203        ;put sprite 0 in center ($80) of screen horizontally
LDA #$00
STA $0201        ;tile number = 0
STA $0202        ;color palette = 0, no flipping
LDA #%10000000   ; enable NMI, sprites from Pattern Table 0
STA $2000
LDA #%00010000   ; no intensify (black background), 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 sprites.asm, mario.chr, and sprites.bat are all in the same folder as NESASM3, then double click sprites.bat. That will run NESASM3 and should produce the sprites.nes file. 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 (0 to 255), or to change the color palette for the sprite (0 to 3). You can choose the PPU viewer in FCEUXD SP to see both Pattern Tables, and both Palettes.

Next Week: multiple sprites, reading controllers

This Week: one sprite is boring, so now we add many more! Also move that sprite around using the controller.

Multiple Sprites

Last time there was only 1 sprite loaded so we just used a few LDA/STA pairs to load the sprite data. This time we will have 4 sprites on screen. Doing that many load/stores just takes too much writing and code space. Instead a loop will be used to load the data, like was used to load the palette before. First the data bytes are set up using the .db directive:

```
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 #$10              ; Compare X to hex $10, decimal 16
 BNE LoadSpritesLoop   ; Branch to LoadSpritesLoop if compare was Not Equal to zero
                ; if compare was equal to 16, continue 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.

Once the sprites have been loaded into RAM, you can modify the data there.

Controller Ports

The controllers are accessed through memory port 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, everything else erased
```

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  ; erase everything but 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 all 4 sprites together.

Finally try changing the code to use the dpad instead of the A and B buttons. Left/right should change the X position of the sprites, and up/down should change the Y position of the sprites.

NEXT WEEK: Backgrounds, attribute table

This Week: Now that you have a basic understanding of the NES tile graphics, we learn how to display one static non scrolling background.

Backgrounds

There are three components used to generate backgrounds on the NES. First is the background color palette, used to select the colors that will be used on screen. Next is the nametable that tells the layout of the graphics. Finally is the attribute table that assigns the colors in the palette to areas on screen.

Background Palette

Like the sprites there are 16 colors in the **background palette**. Our previous apps were already loading a background palette but it was not being used yet. You can use the PPU Viewer in FCEUXD SP to see the color palettes.

Nametables

Like the sprites, background images are made up from 8x8 pixel tiles. The screen video resolution is 32x30 tiles, or 256x240 pixels. PAL systems will show this full resolution but NTSC crops the top 8 and bottom 8 rows of pixels for a final resolution of 256x224. Additionally TV's on either system can crop another few rows on the top or bottom.

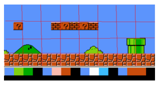
One screen full of background tiles is called a **nametable**, and the NES has enough internal RAM connected to the PPU for two nametables. Only one will be used here. The nametable has one byte (0-255) for which 8x8 pixel graphics tile to draw on screen. The nametable we will use starts at PPU address $2000 and takes up 960 bytes (32x30). You can use the Nametable viewer in FCEUXD SP to see all the nametables.

Attribute Tables

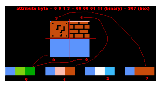
The attribute tables may be the most difficult thing to understand, and sets many of the graphics limitations. Each nametable has an **attribute table** that sets which colors in the palette will be used in sections of the screen. The attribute table is stored in the same internal RAM as the nametable, and we will use the one that starts at PPU address $23C0 ($2000+960).

First the screen is divided into a 32x32 pixel grid, or 4x4 tiles. Each byte in the attribute table sets the color group (0-3) in the background palette that will be used in that area. That 4x4 tile area is divided again into 4 2x2 tile grids. Two bits of the attribute table byte are assigned to each 2x2 area. That is the size of one block in SMB. This limitation means that only 4 colors (one color group) can be used in any 16x16 pixel background section. A green SMB pipe section cannot use the color red because it already uses 4 colors.

When looking at a sample SMB screen, first the 4x4 tile grid is added and the palette is shown at the bottom:



You can see there are 8 grid squares horizontally, so there will be 8 attribute bytes horizontally. Then each one of those grid squares is split up into 2x2 tile sections to generate the attribute byte:



No 16x16 area can use more than 4 colors, so the question mark and the block cannot use the greens from the palette.

Uploading the data

To set the background graphics your data has to be defined in your ROM using the .db directive, then copied to the PPU RAM. Some graphics tools will generate this data but here it will just be done manually. To keep it shorter only a few rows of graphics will be created. The same CHR file from SMB will be used here too. First the nametable data is defined, with each graphics row split into two 16 byte sections to keep lines shorter:

```
nametable:
  .db $24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$24  ;;row 1
  .db $24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$24  ;;all sky ($24 = sky)

  .db $24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$24  ;;row 2
  .db $24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$24  ;;all sky

  .db $24,$24,$24,$24,$45,$45,$24,$24,$45,$45,$45,$45,$45,$45,$24,$24  ;;row 3
  .db $24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$53,$54,$24,$24  ;;some brick tops

  .db $24,$24,$24,$24,$47,$47,$24,$24,$47,$47,$47,$47,$47,$47,$24,$24  ;;row 4
  .db $24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$24,$55,$56,$24,$24  ;;brick bottoms
```

Then the attribute table data is defined. Each byte covers 4x4 tiles, so only 8 bytes are needed here. Binary is used so editing the 2 bits per 2x2 tile area is easier:

```
attribute:
  .db %00000000, %00010000, %0010000, %00010000, %00000000, %00000000, %00000000, %00110000
```

And finally the same color palette as SMB is used:

```
palette:
  .db $22,$29,$1A,$0F,  $22,$36,$17,$0F,  $22,$30,$21,$0F,  $22,$27,$17,$0F
```

Just like our previous palette loading, a loop is used to copy a specific number of bytes from a memory location to the PPU. First the PPU address is set to the beginning of the nametable at $2000. Then our 128 bytes of background data are copied. Next the PPU address is set to the beginning of the attribute table at $23C0 and 8 bytes are copied.

```
LoadBackground:
  LDA $2002             ; read PPU status to reset the high/low latch
  LDA #$20
  STA $2006             ; write the high byte of $2000 address
  LDA #$00
  STA $2006             ; write the low byte of $2000 address
  LDX #$00              ; start out at 0
LoadBackgroundLoop:
  LDA background, x     ; load data from address (background + the value in x)
  STA $2007             ; write to PPU
  INX                   ; X = X + 1
  CPX #$80              ; Compare X to hex $80, decimal 128 - copying 128 bytes
  BNE LoadBackgroundLoop  ; Branch to LoadBackgroundLoop if compare was Not Equal to zero
                ; if compare was equal to 128, keep going down

LoadAttribute:
  LDA $2002             ; read PPU status to reset the high/low latch
  LDA #$23
  STA $2006             ; write the high byte of $23C0 address
  LDA #$C0
  STA $2006             ; write the low byte of $23C0 address
  LDX #$00              ; start out at 0
LoadAttributeLoop:
  LDA attribute, x      ; load data from address (attribute + the value in x)
  STA $2007             ; write to PPU
  INX                   ; X = X + 1
  CPX #$08              ; Compare X to hex $08, decimal 8 - copying 8 bytes
  BNE LoadAttributeLoop
```

The final changes are to tell the PPU to use the Pattern Table 0 graphics for sprites, and Pattern Table 1 for background:

```
  LDA #%10010000 ;enable NMI, sprites from Pattern 0, background from Pattern 1
  STA $2000
```

Enable the background rendering:

```
  LDA #%00011110 ; enable sprites, enable background
  STA $2001
```

And to tell the PPU that we are not doing any scrolling at the end of NMI:

```
  LDA #$00
  STA $2005
  STA $2005
```

Putting It All Together

Download and unzip the background2.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 the background. Set it to PAL Emulation so you get to see the whole screen.

Any background areas that you did not write to will still be using tile 0, which happens to be the number 0 in the SMB graphics. Try adding more nametable and attribute data to the .db sections, then changing the loops so they copy more bytes to the PPU RAM. You can also try changing the starting PPU address of the nametable and attribute table writes to move the rows further down.

NEXT WEEK: Subroutines, game structure, states

This Week: Most of this lesson is about how to organize and structure your game. Subroutines and game states help arrange the code for easier reading and reuse of code.

Variables

As covered in week 1, variables are data stored in RAM that you can change any time. The sprite data in RAM is all variables. You will need more variables for keeping track of things like the score in the game. To do that you first need to tell NESASM where in RAM to put the variable. This is done using the .rsset and .rs directives. First .rsset is used to set the starting address of the variable. Then .rs is used to reserve space. Usually just 1 byte is reserved, but you can have as much as you want. Each time you do a .rs the address gets incremented so you don't need to do .rsset again.

```
 .rsset $0000    ;put variables starting at 0
 score1   .rs 1  ;put score for player 1 at $0000
 score2   .rs 1  ;put score for player 2 at $0001
 buttons1 .rs 1  ;put controller data for player 1 at $0002
 buttons2 .rs 1  ;put controller data for player 2 at $0003
```

Once you set the address for the variable, you do not need to know the address anymore. You can just reference it using the variable name you created. You can insert more variables above the current ones and the assembler will automatically recalculate the addresses.

Constants

Constants are numbers that you do not change. They are just used to make your code easier to read. In Pong an example of a constant would be the position of the outer walls. You will need to compare the ball position to the walls to make the ball bounce, but the walls do not change so they are good constants. Doing a compare to LEFTWALL is easier to read and understand than a comparison to $F6.

To declare constants you use the = sign:

```
 RIGHTWALL      = $02 ; when ball reaches one of these, do something
 TOPWALL        = $20
 BOTTOMWALL     = $D8
 LEFTWALL       = $F6
```

The assembler will then do a find/replace when building your code.

Subroutines

As your program gets larger, you will want subroutines for organization and to reuse code. Instead of progressing linearly down your code, a subroutine is a block of code located somewhere else that you jump to, then return from. The subroutine can be called at any time, and used as many times as you want. Here is what some code looks like without subroutines:

```
RESET:
 SEI          ; disable IRQs
 CLD          ; disable decimal mode

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

clrmem:
 LDA #$FE
 STA $0200, x
 INX
 BNE clrmem

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

Notice that the vblankwait is done twice, so it is a good choice to turn into a subroutine. First the vblankwait code is moved outside the normal linear flow:

```
vblankwait:      ; wait for vblank
 BIT $2002
 BPL vblankwait

RESET:
 SEI          ; disable IRQs
 CLD          ; disable decimal mode

clrmem:
 LDA #$FE
 STA $0200, x
 INX
 BNE clrmem
```

Then that code needs to be called, so the JSR (Jump to SubRoutine) instruction is where the vblankwait code used to be:

```
RESET:
 SEI          ; disable IRQs
 CLD          ; disable decimal mode

 JSR vblankwait  ;;jump to vblank wait #1

clrmem:
 LDA #$FE
 STA $0200, x
 INX
 BNE clrmem

 JSR vblankwait  ;; jump to vblank wait again
```

And then when the subroutine has finished, it needs to return back to the spot it was called from. This is done with the RTS (ReTurn from Subroutine) instruction. The RTS will jump back to the next instruction after the JSR:

```
   vblankwait:      ; wait for vblank  <--------
    BIT $2002                                \
    BPL vblankwait                            |
 ----- RTS                                     |
/                                  |
|  RESET:                                     |
|   SEI          ; disable IRQs               |
|   CLD          ; disable decimal mode       |
|                                 |
|    JSR vblankwait  ;;jump to vblank wait #1 --/
|
\--> clrmem:
    LDA #$FE
    STA $0200, x
    INX
    BNE clrmem

    JSR vblankwait  ;; jump to vblank wait again, returns here
```

Better Controller Reading

Now that you can set up subroutines, you can do much better controller reading. Previously the controller was read as it was processed. With multiple game states, that would mean many copies of the same controller reading code. This is replaced with one controller reading subroutine that saves the button data into a variable. That variable can then be checked in many places without having to read the whole controller again.

```
ReadController:
  LDA #$01
  STA $4016
  LDA #$00
  STA $4016
  LDX #$08
ReadControllerLoop:
  LDA $4016
  LSR A           ; bit0 -> Carry
  ROL buttons     ; bit0 <- Carry
  DEX
  BNE ReadControllerLoop
  RTS
```

This code uses two new instructions. The first is LSR (Logical Shift Right). This takes each bit in A and shifts them over 1 position to the right. Bit 7 is filled with a 0, and bit 0 is shifted into the Carry flag.

```
bit number      7 6 5 4 3 2 1 0  carry
original data   1 0 0 1 1 0 1 1  0
           \ \ \ \ \ \ \  \
            \ \ \ \ \ \ \  \
shifted data    0 1 0 0 1 1 0 1  1
```

Each bit position is a power of 2, so LSR is the same thing as divide by 2.

The next new instruction is ROL (ROtate Left) which is the opposite of LSR. Each bit is shifted to the left by one position. The Carry flag is put into bit 0. This is the same as a multiply by 2.

These instructions are used together in a clever way for controller reading. When each button is read, the button data is in bit 0. Doing the LSR puts the button data into Carry. Then the ROL shifts the previous button data over and puts Carry back to bit 0. The following diagram shows the values of Accumulator and buttons data at each step of reading the controller:

```
              Accumulator                               buttons data
bit:             7  6  5  4  3  2  1  0  Carry           7  6  5  4  3  2  1  0  Carry
read button A    0  0  0  0  0  0  0  A  0               0  0  0  0  0  0  0  0  0
LSR A            0  0  0  0  0  0  0  0  A               0  0  0  0  0  0  0  0  A
```

```
ROL buttons      0  0  0  0  0  0  0  0  A             0  0  0  0  0  0  0  A  0

read button B    0  0  0  0  0  0  0  B  0             0  0  0  0  0  0  0  A  0
LSR A            0  0  0  0  0  0  0  0  B             0  0  0  0  0  0  0  A  B
ROL buttons      0  0  0  0  0  0  0  0  0             0  0  0  0  0  0  A  B  0

read button SEL  0  0  0  0  0  0  0 SEL 0             0  0  0  0  0  0  0  A  0
LSR A            0  0  0  0  0  0  0  0 SEL            0  0  0  0  0  0  0  A SEL
ROL buttons      0  0  0  0  0  0  0  0  0             0  0  0  0  0  A  B SEL 0

read button STA  0  0  0  0  0  0  0 STA 0             0  0  0  0  0  0  0  A  0
LSR A            0  0  0  0  0  0  0  0 STA            0  0  0  0  0  0  0  A STA
ROL buttons      0  0  0  0  0  0  0  0  0             0  0  0  0  A  B SEL STA 0
```

The loop continues for a total of 8 times to read all buttons. When it is done there is one button in each bit:

```
bit:        7     6     5     4     3     2     1     0
button:     A     B   select start  up   down  left right
```

If the bit is 1, that button is pressed.

Game Layout

The Pong game engine will use the typical simple NES game layout. First all the initialization is done. This includes clearing out RAM, setting up the PPU, and loading in the title screen graphics. Then it enters an infinite loop, waiting for the NMI to happen. When the NMI hits the PPU is ready to accept all graphics updates. There is a short time to do these so code like sprite DMA is done first. When all graphics are done the actual game engine starts. The controllers are read, then game processing is done. The sprite position is updated in RAM, but does not get updated until the next NMI. Once the game engine has finished it goes back to the infinite loop.

```
Init Code -> Infinite Loop -> NMI -> Graphics Updates -> Read Buttons -> Game Engine --\
             ^                                                                          |
              \------------------------------------------------------------------------/
```

Game State

The use of a "game state" variable is a common way to arrange code. The game state just specifies what code should be run in the game engine each frame. If the game is in the title screen state, then none of the ball movement code needs to be run. A flow chart can be created that includes what each state should do, and the next state that should be set when it is done. For Pong there are just 3 basic states.

```
 ->Title State              /--> Playing State           /-->  Game Over State
/ wait for start button --/     move ball              /      wait for start button -\
|                    move paddles           |                              \
|                    check for collisions   /                              |
|                    check for score = 15 -/                               |
 \                                                    /
  \------------------------------------------------------------------------/
```

The next step is to add much more detail to each state to figure out exactly what is needed. These layouts are done before any significant coding starts. Some of the game engine like the second player and the score will be added later. Without the score there is no way to get to the Game Over State yet.

```
Title State:
 if start button pressed
  turn screen off
  load game screen
  set paddle/ball position
  go to Playing State
  turn screen on


Playing State:
 move ball
  if ball moving right
   add ball speed x to ball position x
   if ball x > right wall
    bounce, ball now moving left

  if ball moving left
   subtract ball speed x from ball position x
   if ball x < left wall
    bounce, ball now moving right

  if ball moving up
   subtract ball speed y from ball position y
   if ball y < top wall
    bounce, ball now moving down

  if ball moving down
   add ball speed y to ball position y
    if ball y > bottom wall
     bounce, ball now moving up

 if up button pressed
  if paddle top > top wall
   move paddle top and bottom up

 if down button pressed
  if paddle bottom < bottom wall
   move paddle top and bottom down

 if ball x < paddle1x
  if ball y > paddle y top
   if ball y < paddle y bottom
    bounce, ball now moving left


Game Over State:
 if start button pressed
  turn screen off
  load title screen
  go to Title State
  turn screen on
```

Putting It All Together

Download and unzip the pong1.zip sample files. The playing game state and ball movement code is in the pong1.asm file. Make sure that file, mario.chr, and pong1.bat is in the same folder as NESASM3, then double click on pong1.bat. That will run NESASM3 and should produce pong1.nes. Run that NES file in FCEUXD SP to see the ball moving!

Other code segments have been set up but not yet completed. See how many of those you can program yourself. The main parts missing are the paddle movements and paddle/ball collisions. You can also add the intro state and the intro screen, and the playing screen using the background information from the previous week.

This Week: The NES is an 8 bit machine, but sometimes you need more! Learn to handle 16+ bit numbers, and use them for bigger loops.

16 Bit Math

Doing 16 bit addition and subtraction is fairly simple because of the carry flag that we had previously been clearing. First the normal add is done using the clc/adc pair. This add is for the lower 8 bits of the 16 bit number. For the upper 8 bits the adc instruction is used again, but without the clc. You want to keep the carry from the first add, in case it overflowed. To only add in the carry the second adc value is just 0.

Here are some examples in decimal. One digit column is added at a time. The carry (1) is added to the next column as needed.

```
      0 3
+     0 4
      0 7    (no carry needed, top digit = 0)
```

```
      0 4
+     0 8
      1 2    (carry only, top digit = 1)
```

```
      2 2
+     1 9
      4 1    (carry plus 2 plus 1, top digit = 4)
```

And the code to do it on the NES, adding 1 to a 16 bit number:

```
  LDA lowbyte      ; load low 8 bits of 16 bit value
  CLC              ; clear carry
  ADC #$01         ; add 1
  STA lowbyte      ; done with low bits, save back
  LDA highbyte     ; load upper 8 bits
  ADC #$00         ; add 0 and carry from previous add
  STA highbyte     ; save back
```

The same process of adding 0 without clearing the carry can be continued to do 24 bit, 32 bit, or higher numbers. It is also the same process to do 16 bit subtraction:

```
  LDA lowbyte      ; load low 8 bits of 16 bit value
  SEC              ; set carry
  SBC #$01         ; subtract 1
  STA lowbyte      ; done with low bits, save back
  LDA highbyte     ; load upper 8 bits
  SBC #$00         ; subtract 0 and carry from previous sub
  STA highbyte     ; save back
```

Pointers and Indirect Indexed Mode

Previously when loading background tiles the x register was used as an 8 bit offset. Now that we can handle 16 bit numbers a different addressing mode can be used. The 16 bit address is saved into two 8 bit variables, which are then used as a "**pointer**" which points to the background data we want. The LDA instruction then uses the "**Indirect Indexed**" addressing mode. This takes the 16 bit variable inside the brackets and uses it as an address. For the address to be correct, the low byte must be first and the high byte must come immediately after. Then the value in the Y register is added to the address. This forms the final address to load from. Both variables must also be in the first 256 bytes of RAM, called "**Zero Page**", and the X register cannot be used with this addressing mode.

```
    .rsset $0000       ; put pointers in zero page
pointerLo  .rs 1   ; pointer variables are declared in RAM
pointerHi  .rs 1   ; low byte first, high byte immediately after

    LDA #$D0
    STA pointerHi
    LDA #$12
    STA pointerLo       ; pointer now says $D012

    LDY #$00            ; no offset from Y
    LDA [pointerLo], y  ; load data from the address pointed to by the 16 bit pointer variable plus the value in the Y
register
```

That last line is the same as

LDA $D012, y

Because we kept Y = 0, that is the same as

LDA $D012

Copy Loops

Now using your 16 bit math the pointer address can be incremented. Instead of being limited to 256 background tiles like when using the x offset, the whole background can be copied in one loop. First the address of the background data is put into the pointer variable. The high and low bytes of the address are each copied individually. Then the number of tiles to copy is put into the loop counter, which will count down to 0. Each time through the loop one byte will be copied, the 16 bit pointer address will be incremented, and the 16 bit loop counter will be decremented. The Y offset is always kept at 0, because the pointer always points to the correct byte. When the loop counter reaches 0 everything is done.

```
    LDA #LOW(background)
    STA pointerLo       ; put the low byte of the address of background into pointer
    LDA #HIGH(background)
    STA pointerHi       ; put the high byte of the address into pointer

    LDA #$00
    sta counterLo       ; put the loop counter into 16 bit variable
    LDA #$04
    sta counterHi       ; count = $0400 = 1KB, the whole screen at once including attributes


    LDY #$00            ; put y to 0 and don't change it
LoadBackgroundLoop:
    LDA [pointerLo], y
    STA $2007           ; copy one background byte

    LDA pointerLo
    CLC
    ADC #$01
    STA pointerLo
    LDA pointerHi
    ADC #$00
    STA pointerHi       ; increment the pointer to the next byte


    LDA counterLo
    SEC
```

```
  SBC #$01
  STA counterLo
  LDA counterHi
  SBC #$00
  STA counterHi       ; decrement the loop counter


  LDA counterLo
  CMP #$00
  BNE LoadBackgroundLoop
  LDA counterHi
  CMP #$00
  BNE LoadBackgroundLoop  ; if the loop counter isn't 0000, keep copying
```

That is a lot of code to copy just one byte!

Nested Loops

To avoid using so much code, we can use both the X and Y registers as loop counters. By putting one loop inside another loop we create a "nested loop". First the inside loop counts all the way up. Then the outside loop counts up once, and the inside loop counts all the way again. Normally using only X or Y would only give a maximum of 256 times through a loop like we have previously done. With nested loops using both X and Y the maximum is the inside counter multiplied by the outside counter, or 256*256 = 65536.

```
  LDX #$00
  LDY #$00
OutsideLoop:

InsideLoop:
  ;
  ;  this section runs 256 x 256 times
  ;

  INY                 ; inside loop counter
  CPY #$00
  BNE InsideLoop      ; run the inside loop 256 times before continuing down

  INX
  CPX #$00
  BNE OutsideLoop     ; run the outside loop 256 times before continuing down
```

First the Inside Loop runs and Y will count from 0 to 256. When that finishes X will count 0 to 1, and branch back to the beginning of the loops. Then the Inside Loop runs again, Y 0 -> 256. X now goes 1 -> 2 and the process continues. Everything ends when both X and Y have each counted to 256.

When we are using nested loops to copy entire backgrounds we want 256 x 4 = 1KB. The Y code from above can be unchanged, but the X code is changed to CPX #$04.

Because we are changing the Y register our previous pointer copying code also needs to be modified. Instead of incrementing the pointer every time, the incrementing Y register is doing the same thing. The low byte of the pointer will be kept at 0. This means your background data needs to be aligned to where the low byte of the address is $00. However the high byte of the pointer still needs to change. By always making the inside loop count 256 times, that will end at the same time that the high byte needs to change. This time 16 bit math isn't needed because only the high byte is incremented.

No loop counter is used because X and Y are used instead. If you cannot align your data so the low byte of the address is $00, you will have to use the CopyLoop above.

```
  LDA #$00
  STA pointerLo       ; put the low byte of the address of background into pointer
  LDA #HIGH(background)
```

```
    STA pointerHi       ; put the high byte of the address into pointer

    LDX #$00            ; start at pointer + 0
    LDY #$00
OutsideLoop:

InsideLoop:
    LDA [pointerLo], y  ; copy one background byte from address in pointer plus Y
    STA $2007           ; this runs 256 * 4 times

    INY                 ; inside loop counter
    CPY #$00
    BNE InsideLoop      ; run the inside loop 256 times before continuing down

    INC pointerHi       ; low byte went 0 to 256, so high byte needs to be changed now

    INX
    CPX #$04
    BNE OutsideLoop     ; run the outside loop 256 times before continuing down
```

Putting It All Together

Download and unzip the background3.zip sample files. All the code 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 background3.nes. Run that NES file in FCEUXD SP to see the full background.

The new nested loop is used to copy a whole background to the screen instead of only 128 bytes. The background is aligned using the .org directive so the low address byte is $00. The attributes are also placed directly after the background data so it is are copied at the same time.

Your task is to separate out the code that sets the pointer variables from the code that copies the loop. That way you can have multiple backgrounds that use different pointer loading code, but the same copy code.

If you are using a different assembler, the Indirect Indexed mode may use () instead of []. The LOW() and HIGH() syntax may also be different.

This Week: NES uses binary and hex, but your gamers want to read in decimal? Here are two solutions for displaying scores and other numbers in a readable way.

BCD Mode

The 6502 processor has a mode called BCD, or Binary Coded Decimal, where the adc/sbc instructions properly handle decimal numbers instead of binary numbers. The NES is not a full 6502 processor and does not include this mode. Be careful when you are searching for code to not copy any that uses that mode, or you will get incorrect results. If the code is doing a SED instruction, it is enabling the decimal mode and you should not use it. Instead you get to do all the decimal handling yourself!

Storing Digits

The first method uses more code, but may be easier to understand. Say your score is a 5 digit number. You will make 5 variables, one for each digit. Those variables will only count from 0 to 9 so you need to write code to handle addition and subtraction. Super Mario uses this method. It's lowest digit is always 0, so that isn't actually stored in a variable. Instead it is just a permanent part of the background.

We will start with just incrementing a 3 digit number to see how its done:

```
IncOnes:
   LDA onesDigit     ; load the lowest digit of the number
   CLC
   ADC #$01          ; add one
   STA onesDigit
   CMP #$0A          ; check if it overflowed, now equals 10
   BNE IncDone        ; if there was no overflow, all done
IncTens:
   LDA #$00
   STA onesDigit     ; wrap digit to 0
   LDA tensDigit     ; load the next digit
   CLC
   ADC #$01          ; add one, the carry from previous digit
   STA tensDigit
   CMP #$0A          ; check if it overflowed, now equals 10
   BNE IncDone        ; if there was no overflow, all done
IncHundreds:
   LDA #$00
   STA tensDigit     ; wrap digit to 0
   LDA hundredsDigit ; load the next digit
   CLC
   ADC #$01          ; add one, the carry from previous digit
   STA hundredsDigit
IncDone:
```

When the subroutine starts, the ones digit is incremented. Then it is checked if it equals $0A which is decimal 10. That number doesn't fit in just one digit, so the ones digit is set to 0 and the tens digit is incremented. The tens digit is then checked in the same way, and the chain continues for as many digits as you want.

The same process is used for decrementing, except you check for underflow (digit=$FF) and wrap the digit to $09.

Adding two numbers is the same idea, except other than checking if each digit equals $0A you need to check if the digit is $0A or above. So instead of BEQ the opcode will be BCC.

```
AddOnes:
   LDA onesDigit      ; load the lowest digit of the number
   CLC
   ADC onesAdd         ; add new number, no carry
   STA onesDigit
   CMP #$0A           ; check if digit went above 9. If accumulator >= $0A, carry is set
   BCC AddTens         ; if carry is clear, all done with ones digit
                              ;   carry was set, so we need to handle wrapping
   LDA onesDigit
   SEC
```

```
    SBC #$0A          ; subtract off what doesnt fit in 1 digit
    STA onesDigit     ; then store the rest
    INC tensDigit     ; increment the tens digit
AddTens:
    LDA tensDigit     ; load the next digit
    CLC
    ADC tensAdd       ; add new number
    STA tensDigit
    CMP #$0A          ; check if digit went above 9
    BCC AddHundreds   ; no carry, digit done
    LDA tensDigit
    SEC
    SBC #$0A          ; subtract off what doesnt fit in 1 digit
    STA tensDigit     ; then store the rest
    INC hundredsDigit ; increment the hundreds digit
AddHundreds:
    LDA hundredsDigit ; load the next digit
    CLC
    ADC hundredsAdd   ; add new number
    STA hundredsDigit
AddDone:
```

When that code is all done, the ones/tens/hundreds digits will hold the new value. With both code samples there is no check at the end of the hundreds digit. That means when the full number is 999 and you add one more, the result will be wrong! In your code you can either wrap around all the digits to 0, or set all the digits to 999 again for a maximum value. Of course if your players are hitting the max they likely want more digits!

Binary to Decimal Conversion

The second method of handling number displays uses less code, but could use much more CPU time. The idea is to keep you numbers in plain binary form (8 or 16 bit variables) for the math, then convert them to decimal for displaying only. An 8 bit binary value will give you 3 decimal digits, and a 16 bit binary will give 5 decimal digits.

This first example is coded to be understandable, not fast or small. Each step compares the binary value to a significant decimal value (100 and then 10). If the binary is larger, that value is subtracted from the binary and the final decimal digit is incremented. So for a text example:

initial binary: 124
initial decimal: 000

1: compare to 100
2: 124 greater than 100, so subtract 100 and increment the decimal hundreds digit
3: repeat hundreds again

current binary: 024
current decimal: 100

1: compare to 100
2: 024 less than 100, so all done with hundreds digit

current binary: 024
current decimal: 100

1: compare to 10
2: 024 greater than 10, so subtract 10 and increment the decimal tens digit
3 repeat tens again

current binary: 014

current decimal: 110

1: compare to 10
2: 014 greater than 10, so subtract 10 and increment the decimal tens digit
3 repeat tens again

current binary: 004
current decimal: 120

etc for ones digit

You can see this will transfer the binary to decimal one digit at a time. For numbers with large digits (like 249) this will take longer than numbers with small digits (like 112). Here is the code:

```
HundredsLoop:
    LDA binary
    CMP #100             ; compare binary to 100
    BCC TensLoop         ; if binary < 100, all done with hundreds digit
    LDA binary
    SEC
    SBC #100
    STA binary           ; subtract 100, store whats left
    INC hundredsDigit    ; increment the digital result
    JMP HundredsLoop     ; run the hundreds loop again

TensLoop:
    LDA binary
    CMP #10              ; compare binary to 10
    BCC OnesLoop         ; if binary < 10, all done with hundreds digit
    LDA binary
    SEC
    SBC #10
    STA binary           ; subtract 10, store whats left
    INC tensDigit        ; increment the digital result
    JMP TensLoop         ; run the tens loop again

OnesLoop:
    LDA binary
    STA onesDigit        ; result is already under 10, can copy directly to result
```

This code can be expanded to 16 bit numbers, but the compares become harder. Instead a more complex series of loops and shifts with a table is used. This code does shifting of the binary value into the carry bit to tell when to add numbers to the final decimal result. I did not write this code, it came from a post by Tokumaru at http://nesdev.parodius.com/bbs/viewtopic.php?p=10824&am... There are many more examples of different conversion styles at that forum thread.

Notice there are no branches other than the loop running 16 times (one for each binary input bit), so the conversion always takes the same number of cycles.

```
    tempBinary - 16 bits input binary value
    decimalResult - 5 bytes for the decimal result

BinaryToDecimal:
      lda #$00
      sta decimalResult+0
      sta decimalResult+1
      sta decimalResult+2
      sta decimalResult+3
      sta decimalResult+4
      ldx #$10
BitLoop:
      asl tempBinary+0
      rol tempBinary+1
      ldy decimalResult+0
      lda BinTable, y
      rol a
      sta decimalResult+0
      ldy decimalResult+1
      lda BinTable, y
      rol a
      sta decimalResult+1
      ldy decimalResult+2
      lda BinTable, y
      rol a
      sta decimalResult+2
      ldy decimalResult+3
      lda BinTable, y
      rol a
      sta decimalResult+3
      rol decimalResult+4
      dex
      bne BitLoop
      rts
BinTable:
      .db $00, $01, $02, $03, $04, $80, $81, $82, $83, $84
```

Displaying Numbers

Once you have your numbers in decimal format you need to display them on the screen. With the code above all the results have 00000 = $00 $00 $00 $00 $00. If your background tiles for digits start at tile 0 then that will work fine. However if you are using ASCII you will need to add an offset to each digit. The ASCII code for the digit 0 is $30, so you just add $30 to each digit before writing it to the background. If your code uses the first method of compare/wrapping digits, then you could compare to $3A and wrap to $30 to automatically handle this. You would just need to make sure you set each digit to $30 instead of $00 when clearing the number to 00000. You have control over where background tiles are located, so the offset for the digit tiles can be whatever you choose.

Putting It All Together

Download and unzip the pong2.zip sample files. The playing game state and ball movement code is in the pong2.asm file. Make sure that file, mario.chr, and pong2.bat is in the same folder as NESASM3, then double click on pong1.bat. That will run NESASM3 and should produce pong2.nes. Run that NES file in FCEUXD SP to see the score! Right now the score just increments every time the ball bounces off a side wall.

Try making two scoring variables and drawing them both. You can also use the other binary to decimal converters to add more than 1 to the score each time. In the DrawScore you can also check the score digits and not draw any leading zeros. Instead replace them with spaces when you are drawing to the background.

To do the advanced lessons you should have already finished Pong.

This Week: As you complete a full game you may find the NROM memory limits to be too small. To enable more ROM on carts many forms of "bank switching" were used. This article deals with just one type of CHR switching, used on CNROM carts. CNROM is easy to use and very cheap to manufacture. The ReproPak, PowerPak, and PowerPak Lite all support CNROM completely so it is easy to get your code running on real hardware. If you are using donor carts you can look up games that use CNROM at BootGod's NES Cart Database.

CHR Bank Switching

Bank switching is exchanging one chunk of ROM for a different chunk, while keeping everything in same address range. It is not making a copy, so it happens instantly. You can switch between different banks whenever you want. The size and memory range of the banks depends on the mapper. For the CNROM mapper used in this article the bank size is 8KB of CHR ROM. The whole 8KB range of PPU memory $0000-1FFF is switched at once. This means the graphics for all background tiles and sprite tiles will be swapped. In your game you may have some tiles duplicated in multiple banks so they do not appear to change on screen. PRG is not bank switched, so it remains at the NROM limit of 32KB.

Set Mapper Number

The first part of adding bank switching is changing the mapper number your .NES file uses. At the top of your code has previously been:

```
 .inesmap 0 ; mapper 0 = NROM, no bank swapping
```

The new line is:

```
 .inesmap 3 ; mapper 3 = CNROM, 8KB CHR ROM bank swapping
```

This line in the header just tells the emulator to use CNROM to play your game. A list of other iNES mapper numbers can be seen at the wiki at http://nesdevwiki.org/.

Set CHR Size

The next part is to increase the size of your CHR ROM. Change the .ineschr value from 1 to 2, showing that there are now two 8KB banks. CNROM can handle 32KB of CHR ROM or four 8KB banks but this example will only use two.

Add CHR Data

The third part adds the data for the next bank into your game. Just make a new .bank statement below your current one for CHR, giving it the next sequential number. In your code when you set which bank to switch to this is the number used. PRG bank numbers are ignored so your original CHR bank will be #0 and the new one will be #1.

Bank Switching Code

The final part it to write your bank switching code. This subroutine will take a bank number in the A register and switch the CHR bank to it immediately. The actual switch is done by writing the desired bank number anywhere in the $8000-FFFF memory range. The cart hardware sees this write and changes the CHR bank.

```
... your game code ...
 LDA #$01 ;;put new bank to use into the A register
 JSR Bankswitch ;;jump to bank switching code
... your game code ...


Bankswitch:
 STA $8000 ;;new bank to use
 RTS
```

Bus Conflicts

When you start running your code on real hardware there is one catch to worry about. For basic mappers, the PRG ROM does not care if it receives a read or a write command. It will respond to both like a read by putting the data on the data bus. This is a problem for bank switching, where the CPU is also trying to put data on the data bus at the same time. They electrically fit in a "bus conflict". The CPU could win, giving you the right value. Or the ROM could win, giving you the wrong value. This is solved by having the ROM and CPU put the same value on the data bus, so there is no conflict. First a table of bank numbers is made, and the value from that table is written to do the bank switch.

```
... code ...
 LDA #$01 ;;put new bank to use into A
 JSR Bankswitch ;;jump to bank switching code
... code ...


Bankswitch:
 TAX ;;copy A into X
 STA Bankvalues, X ;;new bank to use
 RTS


Bankvalues:
 .db $00, $01, $02, $03 ;;bank numbers
```

The X register is used as an index into the Bankvalues table, so the value written by the CPU will match the value coming from the ROM.

Putting It All Together

Download and unzip the chrbanks.zip sample files. This set is based on the previous Week 5 code. Make sure that file, mario0.chr, mario1.chr, and chrbanks.bat is in the same folder as NESASM3, then double click on chrbanks.bat. That will run NESASM3 and should produce chrbanks.nes. Run that NES file in FCEUXD SP to see small Mario.

Inside the LatchController subroutine a new section is added to read the Select and Start buttons from the controller. The Select button switches to CHR bank 0, and the Start button switches to CHR bank 1. Graphics of CHR bank 1 have been rearranged so Mario will change to a beetle. The tile numbers are not changed, but the graphics for those tiles are.

Open the PPU Viewer from the Tools menu in FCEUXD SP and try hitting the buttons. You can see all the graphics changing at once when the active bank switches.

This Week: The MMC1 is the first advanced mapper made by Nintendo. It is used for many games including top titles like The Legend of Zelda. The main benefits are mirroring control, up to 256KB of PRG ROM, 128KB of CHR RAM or ROM, and 8KB of WRAM. The WRAM can be battery backed for saved games. This tutorial will cover all features of the MMC1 and how to use them. You should be comfortable with the normal Nerdy Nights series before starting. Another more simple lesson for bankswitching is Advanced Nerdy Nights #1. If you only need one or two of the banking features then you may want to consider more simple and cheaper mappers instead such as UNROM or CNROM.

Carts using the MMC1 will have the S*ROM board code, like SNROM and SGROM. BootGod's NesCartDB database can be searched for which games use which boards. The ReproPak MMC1 board can also be used to build carts.

Shift Registers

The MMC1 uses a 5 bit shift register to temporarily store the banking bits. Shift registers were covered in Week 7. When writing to the register data comes in from data bit 0 only. This is similar to the controller reading where data outputs to data bit 0. Every time a write happens the current bits are shifted and D0 is inserted. The first bit you write eventually becomes to lowest bank bit. On the 5th write when the shift register is full the 5 bit value gets copied to the banking register. At this point the bank switch happens immediately without any delays. To load a bank register the LSR instruction is used for shifting:

```
  LDA banknumber
  STA bankreg    ; load bank bit 0 to shift register from data bit 0
  LSR A          ; shift in next data bit to position 0
  STA bankreg    ; load bank bit 1 from data bit 0
  LSR A
  STA bankreg    ; bank bit 2
  LSR A
  STA bankreg    ; bank bit 3
  LSR A
  STA bankreg    ; bank bit 4, bank register loaded, bank switch happens here
```

Unlike other simple mappers like UNROM and CNROM, there are no bus conflicts. The ROM is not enabled while you are writing so you do not have to make the data you are writing match.

Data bit 7 is also connected to the MMC1. When a write happens to any banking register with D7=1 the shift register is reset back to position 0. It will then take another 5 writes to fully load the next value. All other bits are ignored and D0 is not loaded into the shift register. The PRG bits of the control register are also reset to their default values as shown in the next section. Usually you will only reset the MMC1 at the very beginning of your program:

```
  LDA #%10000000
  STA $8000
```

Config Register at $8000-9FFF

To load the config register, do 5 writes to the $8000-9FFF range. The config bits are:

```
43210
-----
CPRMM
|||||
|||++- Mirroring (0: one-screen, lower bank; 1: one-screen, upper bank;
|||                2: vertical; 3: horizontal)
||+--- PRG swap range (0: switch 16 KB bank at $C000; 1: switch 16 KB bank at $8000;
||                        only used when PRG bank mode bit below is set to 1)
|+---- PRG size (0: switch 32 KB at $8000, ignoring low bit of bank number;
|                  1: switch 16 KB at address specified by location bit above)
+----- CHR size (0: switch 8 KB at a time; 1: switch two separate 4 KB banks)
```

Mirroring Config

Your program can change the mirroring at any point using these bits. You do not need to wait for vblank to change them. When using the MMC1 the .inesmir directive bit is ignored. You must set it through your code. Mirroring set to 0 and 1 are single screen mirroring modes. Only 1KB is used for all nametables. When scrolling the screen will wrap both vertically and horizontally. Mirroring set to 2 is the typical vertical mirroring, and 3 is horizontal mirroring.

PRG Bank Size Config
The MMC1 swaps PRG ROM in either 16KB or 32KB chunks. By default this bit is set to 1 for 16KB banks. Clearing it to 0 enables 32KB banks. Notice these are not the same size as the 8KB NESASM banks so the bank numbers will be different. When using 16KB banks the MMC1 banks are twice as big, so you must divide your NESASM bank number by 2 when writing it to the bank register. When using 32KB banks you must divide by 4. 16KB banks is most commonly used, with the bulk of the code in the fixed bank and data/graphics/music in the swappable banks.

PRG Swap Range Config
When using 16KB banks set above, the PRG address range that gets swapped can be configured. If 32KB banks are used this bit is ignored and the entire $8000-FFFF range is swapped at once.

By default this bit is set to 1, making the $8000-BFFF range swappable while the $C000-FFFF range is fixed to the last bank Of PRG. This matches the PRG swapping of the UNROM mapper and is most commonly used. Clearing this bit to 0 changes this so $8000-BFFF is fixed and $C000-FFFF is swappable.

Changing the range or bank size can be useful for swapping audio samples but you have to be careful to put IRQ/reset/NMI vectors in all banks that are loaded into the vector area at $FFFA-FFFF.

CHR Bank Size Config
Like the PRG the CHR bank size can be configured to either 4KB or 8KB banks. With 8KB banks the whole $0000-1FFF range is one bank. With 4KB banks there are two banks at PPU $0000-0FFF and $1000-1FFF. This can be used with background in one bank and sprites in another. Then, for example, all sprites could be swapped and the background could stay.

CHR Bank 0 Register at $A000-BFFF

This is the register for CHR bank 0. To set it do 5 writes to the $A000-BFFF range. When in 4KB CHR mode it selects a bank for PPU $0000-0FFF. The full 5 bit value is used so there are 32 possible banks. Each bank is 4KB making it 128KB CHR maximum. When in 8KB CHR mode this register controls the full PPU $0000-1FFF. The bottom bit is ignored so there are 16 possible banks. Each bank is now 8KB which is still 128KB max.

4KB mode	**8KB mode**
controls $0000-0FFF	controls $0000-1FFF bottom bit is ignored

CHR Bank 1 Register at $C000-DFFF

This is the register for CHR bank 1. To set it do 5 writes to the $C000-DFFF range. When in 4KB CHR mode it selects a bank for PPU $1000-1FFF. When in 8KB CHR mode it is completely ignored.

4KB mode	**8KB mode**
controls $1000-1FFF	register ignored

PRG Bank Register at $E000-FFFF

This is the register for PRG banking. To set it do 5 writes to the $E000-FFFF range. The bits are:

```
43210
-----
WPPPP
|||||
|++++-  Select a PRG ROM bank (low bit ignored in 32 KB mode)
+-----  WRAM chip enable (0: enabled; 1: disabled)
```

In 16KB PRG mode it selects a 16KB PRG bank for the current swappable address range. Only the 4 lower bits are used for 16 possible PRG banks. That is 256KB maximum. In 32KB PRG mode it selects a 32KB bank for the $8000-FFFF range. Only bits 3-1 are used for 8 possible banks. Bit 0 is ignored.

16**KB mode** swap=0	16**KB mode** swap=1	32**KB mode**
controls $C000-FFFF	controls $8000-BFFF (default setting)	controls $8000-FFFF low bit ignored

WRAM
Bit 5 of the PRG Bank register also controls WRAM access. Clear this bit to enable WRAM access. Setting the bit to 1 disables the WRAM. If the WRAM is used for saved games it is usually disabled when it is not being accessed to prevent unwanted writes from corrupting the saves when the console is reset.

To use WRAM in your program nothing needs to be changed in the iNES header. The emulator will assume there is WRAM based on the mapper number. Next you need to enable the WRAM in the PRG Bank register. With that bit at 0 you can now use the WRAM. It is just plain RAM that you can read or write at the $6000-7FFF range. Like the console RAM the contents are unknown when the console is powered on. All RAM is cleared to unknown values when power is removed. However the RAM is not cleared when just the reset button is pushed. Power is still going to the cart so the RAM is still valid. This can be useful for telling if the console was just turned on or was only reset.

To add a battery to the WRAM, set the .inesmir directive to 2 in the iNES header. Now read and write to the WRAM normally. When power is removed the RAM contents will remain. The emulator will create an 8KB .sav file for the WRAM, however some emulators will not do this unless you have done some WRAM access.

Banking Routines

Keeping track of all the register addresses and bits can get confusing, so a few simple routines are used instead. In general subroutines should be used for even simple bank switching so the mapper can be changed more easily later. Only the most commonly used Config and PRG Bank register routines are shown here, it is your job to write the others.

```
ConfigWrite:        ; make sure this is in a fixed PRG bank so the RTS doesn't get swapped away
    LDA #$80
    STA $8000        ; reset the shift register
    LDA #%00001110    ; 8KB CHR, 16KB PRG, $8000-BFFF swappable, vertical mirroring
    STA $8000        ; first data bit
    LSR A           ; shift to next bit
    STA $8000        ; second data bit
    LSR A           ; etc
    STA $8000
    LSR A
    STA $8000
    LSR A
    STA $8000        ; config bits written here, takes effect immediately
    RTS
```

Putting It All Together

Download and unzip the cyoammc1.zip sample files. The CYOA code has been changed to the MMC1 mapper. Running the cyoammc1.bat file will create cyoammc1.nes, which will run in an emulator. This sample uses 8KB of CHR RAM so no CHR banking has been included. The changes from UNROM PRG mapping are minimal. Some of the variables have been moved to the WRAM area to show how to use it. An example of using WRAM to detect reset is also included in resetCount. Try changing the battery info in the iNES header to see how it makes resetCount change between power and resets.

This Week: Time to learn how to do horizontal background scrolling, like Super Mario Bros. Hopefully it is explained with the most easy to understand code. There is no compression, no buffers, and no metatiles, so only the ideas of scrolling are presented. Once you understand the scrolling part you should look into those other topics to save code/data space and increase performance if needed.

Nametable Review

Before starting the scrolling you must fully understand how nametables work. One nametable is 32x30 background tiles, which covers exactly one visible screen. Including the attribute table, each screen needs 1KB of PPU RAM. The NES PPU has the address space for 4 nametables ($2000, $2400, $2800, $2C00) in a 2x2 grid:

```
+-----------+-----------+
|           |           |
|           |           |
|   $2000   |   $2400   |
|           |           |
|           |           |
+-----------+-----------+
|           |           |
|           |           |
|   $2800   |   $2C00  |
|           |           |
|           |           |
+-----------+-----------+
```

However the NES only has 2KB of PPU RAM inside the console, so there are only two actual nametables. The other two nametables are copies of those actual ones. Your mirroring settings determine the layout of the actual nametables and which ones are copies.

Vertical mirroring means the nametables stacked vertically are the same data. 0 ($2000) is a mirror of 2 ($2800), and 1 ($2400) is a mirror of 3 ($2C00). 0 and 1 are next to each other and have different data. This is what we want for horizontal scrolling. When you are looking at nametable 0 and scroll to the right, nametable 1 will be in view. Typically your mirroring setting is the opposite of the scrolling direction. To set the iNES header:

```
  .inesmir 1  ;;VERT mirroring for HORIZ scrolling
```

Scroll registers

Before scrolling we will fill both nametables 0 ($2000) and 1 ($2400). The same data will be copied into both, except the attribute table will be different. By setting the second nametable attributes to another color palette the two screens will have a very visible difference.

```
FillNametables:
  LDA $2002             ; read PPU status to reset the high/low latch
  LDA #$20
  STA $2006             ; write the high byte of $2000 address (nametable 0)
  LDA #$00
  STA $2006             ; write the low byte of $2000 address
  LDY #$08
  LDX #$00              ; fill 256 x 8 bytes = 2KB, both nametables all full
  LDA #$7F
FillNametablesLoop:
  STA $2007
  DEX
  BNE FillNametablesLoop
  DEY
  BNE FillNametablesLoop


FillAttrib0:
  LDA $2002             ; read PPU status to reset the high/low latch
  LDA #$23
  STA $2006             ; write the high byte of $23C0 address (nametable 0 attributes)
  LDA #$C0
  STA $2006             ; write the low byte of $23C0 address
```

```
    LDX #$40             ; fill 64 bytes
    LDA #$00             ; palette group 0
FillAttrib0Loop:
    STA $2007
    DEX
    BNE FillAttrib0Loop


FillAttrib1:
    LDA $2002            ; read PPU status to reset the high/low latch
    LDA #$27
    STA $2006            ; write the high byte of $27C0 address (nametable 1 attributes)
    LDA #$C0
    STA $2006            ; write the low byte of $27C0 address
    LDX #$40             ; fill 64 bytes
    LDA #$FF             ; palette group 3
FillAttrib1Loop:
    STA $2007
    DEX
    BNE FillAttrib1Loop
```

The scroll registers are at $2005. Like some other PPU registers you need to write to it twice. The first write is the horizontal scroll count, the second write is the vertical scroll count. The scroll sets which pixel of the nametable for the start of the left side of the screen. Previously we have set the scroll to 0 so the left side of the screen is aligned with the left edge of the nametable. The scroll registers are both 8 bit registers, making the scroll range 0 to 255. The screen is 256 pixels wide so the horizontal scroll register covers one full screen wide.

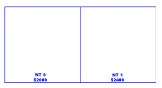
This sample code just increments the horizontal scroll register ($2005) by 1 on every frame. You can see when the first nametable scrolls off the screen, the second one comes on screen. The previously set colors make the split between nametables obvious. As the scroll register wraps from 255 to 0 the first nametable becomes completely visible again. You can also see the sprites are not affected by the scroll registers. They have their own separate x and y position data.

```
NMI:
    LDA #$00
    STA $2003
    LDA #$02
    STA $4014       ; sprite DMA from $0200

    ;  run other game graphics updating code here

    LDA #$00
    STA $2006       ; clean up PPU address registers
    STA $2006
```

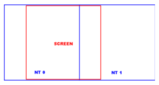
The full code and compiled .NES file is available from the download link at the bottom of this tutorial. scrolling1.asm includes everything up to this point.

Nametable Register

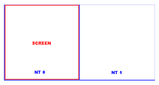
The problem with just the scroll register is that it isn't big enough. In the previous example the scroll wrapped from 255 to 0, so the second nametable is never shown on the left side. Both nametables together is 512 pixels wide but the scroll can only count 256 pixels. The solution is to switch which nametable is on the left side of the screen at the same time the scroll register wraps to 0.



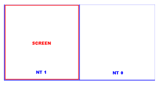
Vertical mirroring means nametables are arranged horizontally



Scrolling shows nametable 0 and 1 (blue) on the screen (red)



When the scroll register wraps, nametable 0 is displayed again



Swap which nametable is on the left when the wrap happens to display nametable 1

To set the starting nametable, change bit 0 of the PPU control register at $2000. Clearing it to 0 will put nametables 0 and 2 on the left side of the screen with 1 and 3 to the right. Setting it to 1 will put 1 and 3 on the left, and 0 and 2 on the right.

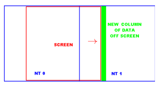
This sample code has the same scroll incrementing, but swaps the nametables at the same time the scroll wraps from 255 to 0. Instead of the background jumping it continuously scrolls from one nametable to the next. When the scroll wraps again the nametables are swapped again and the scrolling keeps going.

NMI:

The full code and compiled .NES file is available from the download link at the bottom of this tutorial. scrolling2.asm includes everything up to this point.

Drawing New Columns

For just two screens of graphics the code above is fine. Games like Super Dodgeball use this method. Both nametables are filled and scrolled between. For games like SMB where the levels are wider than two screens some new background data will have to be inserted. The solution is to draw a new vertical column of tiles somewhere off the visible screen, before it is scrolled into the visible area. As long as the new column is drawn ahead of the visible area, calculated by the current scroll and nametable, it will appear continuous. The tricky part is figuring out which column to draw, and where it is to be placed. If we always use the opposite nametable and the same scroll point we will be drawing the column that is about to come on screen.



>**When to Draw**

We will draw a new column anytime the scroll register becomes a multiple of 8, meaning the scroll is aligned to the tiles. Some bit masking and testing can calculate when this happens. First any part of the scroll not 0 to 7 is thrown away. Then if the result equals 0 the scroll count is a multiple of 8.

```
  LDA scroll
  AND #%00000111     ; throw away higher bits
  BEQ DrawNewColumn  ; see if lower bits = 0
```

Where to Draw
Now that we know when to draw, we need to calculate the starting PPU address of the new column. The scroll register counts in pixels, but we want to count in tiles for which column to draw. Each tile is 8 pixels wide, so we divide the scroll by 8 to get the tile number. That number is the low bits of the address.
LDA scroll LSR A LSR A LSR A ; shift right 3 times = divide by 8 STA columnLow ; $00 to $1F, screen is 32 tiles wide

The high bits of the address will come from the current nametable. First the low bit is inverted, to get the off screen nametable number. Then the number is shifted up and added to the base address.

```
  LDA nametable
  EOR #$01         ; invert low bit, A = $00 or $01
  ASL A            ; shift up, A = $00 or $02
  ASL A            ; $00 or $04
  CLC
  ADC #$20         ; add high byte of nametable base address ($2000)
  STA columnHigh   ; now address = $20 or $24 for nametable 0 or 1
```

Now the scroll count and nametable have been used to make the full column address to start copying new background data. It will be at the top of the nametable that is off screen. As the scroll and nametable are changed, that calculation will still give the correct starting address.

How to Draw
Previously when we have been copying data to the background the PPU is set to auto increment the address by 1. That helps with the copying because a whole row of data can be copied while only writing the PPU address once. Incrementing by 1 goes to the next horizontal tile. In this case we want to go to the next vertical tile because we are copying a column instead of a row. We want it to increment by 32 which will jump down instead of across. There are 32 tiles per row, so adding 32 will always go down to the next row in the same column. The PPU has an increment 32 mode, set using bit 2 in the PPU control register at $2000. When bit 2 is set to 0 the increment mode is +1. When bit 2 is set to 1 the increment mode is +32. By setting the increment mode to +32 and copying 30 bytes of background tiles we can draw one column at a time.

```
DrawColumn:
  LDA #%00000100      ; set to increment +32 mode, don't care about other bits
  STA $2000

  LDA $2002           ; read PPU status to reset the high/low latch
  LDA columnHigh
  STA $2006           ; write the high byte of column address
  LDA columnLow
  STA $2006           ; write the low byte of column address
  LDX #$1E            ; copy 30 bytes
  LDY #$00
DrawColumnLoop:
  LDA columnData, y
  STA $2007
  INY
  DEX
  BNE DrawColumnLoop
```

By using the when/where/how we can draw a new column of data off screen before it becomes visible. The full code and compiled .NES file is available from the download link at the bottom of this tutorial. scrolling3.asm includes everything up to this point. It will be best to watch in an emulator where you can see everything that is off screen. First open the scrolling3.nes file in the FCEUXDSP emulator. Then choose "Name Table Viewer..." from the "Tools" menu. Reset the emulator and watch the new columns being drawn off the visible screen area.

Drawing Real Background Data
The last example drew new columns, but it wasn't any real data. This example adds another counter to keep track of how far along into the level a player is. By incrementing this counter every time a new column is drawn the correct next column is easy to find. The DrawNewColumn function has been updated to use the counter to load real

background data. It can also be used at the beginning of the game initialization to populate the starting nametable data instead of using the fill loops.

The full code and compiled .NES file is available from the download link at the bottom of this tutorial. scrolling4.asm includes 4 screens (128 columns) of real background ripped from SMB.

Updating the Attributes

The final piece of the scrolling puzzle is the attribute table. Updating it is the same process as the background, where the attributes are updated while they are off screen. Again the scroll and nametable registers will be used to calculate the correct attribute bytes to update. Each attribute byte covers a 4x4 tile area. 4 tiles wide is 32 pixels, so the attributes must be updated anytime the scroll register is a multiple of 32. The column numbers already calculated could be used instead of the scroll variables to do the calculations.

```
  LDA scroll
  AND #%00011111     ; check for multiple of 32
  BEQ NewAttrib      ; if low 5 bits = 0, time to write new attribute bytes
```

Only 8 attribute bytes will need to be changed each time. However they are not sequential, so the PPU increment +1 or +32 modes will not work. The PPU address needs to be changed for every attribute byte updated. The starting address is the base attribute table at $20C0. Like the background address the nametable bit is shifted up and added in. Then the scroll register is divided by 32 to get the attribute byte offset. All that is calculated together to find the PPU address of the first attribute byte. After that 8 is added to the address for each of the next bytes.

```
  LDA nametable
  EOR #$01          ; invert low bit, A = $00 or $01
  ASL A             ; shift up, A = $00 or $02
  ASL A             ; $00 or $04
  CLC
  ADC #$20          ; add high byte of attribute base address ($20C0)
  STA columnHigh    ; now address = $20 or $24 for nametable 0 or 1

  LDA scroll
  LSR A
  LSR A
  LSR A
  LSR A
  LSR A
  CLC
  ADC #$C0
  STA columnLow     ; attribute base + scroll / 32
```

The full code and compiled .NES file is available from the download link at the bottom of this tutorial. scrolling5.asm has the same incrementing scroll, but now draws the new column and attribute bytes. Use the Name Table Viewer again to check out the attributes being updated. You can see the attribute update change the color of the off screen clouds before that column of tiles is changed. The same thing is why you see graphical glitches on the sides of SMB3 while it is scrolling. To use this in your own game you will need to expand columnNumber to a bigger value.

Once you have understood everything here, there are some more advanced concepts to check out:

Meta Tiles - This idea is to store your backgrounds as bigger blocks instead of individual tiles. Things like the question blocks would be stored as one byte in the ROM and then decoded into the 4 tiles when it is being drawn. Mostly this saves huge amounts of data space and could make updating attributes easier.

Buffers - A section of RAM can be reserved to act as a buffer for the data to draw to the PPU later. Outside of vblank where the is more processing time the next graphics updates would be calculated and stored in a buffer. Then during vblank those buffers can be dumped right to the PPU, saving time.

Compression - Packing the background data into simple compression formats like RLE can save even more data space. Combine that with meta tiles and buffers to have a full scrolling engine.

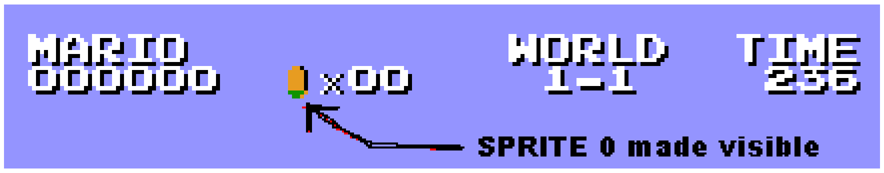
Putting It All Together

Download and unzip the scrolling.zip sample files. Each of them adds a small step, so go through them one at a time. Try expanding the background data to add more columns, making the scroll speed variable, or making the scrolling controlable.

This Week: After scrolling this tutorial should be pretty simple. Sprite 0 has a special PPU flag associated with it. Here it will be used to do split screen scrolling to enable a static status bar on the top of the screen.

Sprite 0 Hit Flag

Sprite 0 has a special flag in the PPU status register at bit 6. When a non transparent pixel of sprite 0 overlaps a non transparent pixel of the background, the flag is set. In the SMB example, sprite 0 is placed at the bottom of the coin icon. That is one part of the status bar that does not move.



In our example we first set the scroll registers to 0 for the static status bar. The nametable is also set to 0. That makes sure that the background and sprite 0 will overlap in the correct place.
NMI: Next we make sure the sprite 0 hit flag is clear, to avoid it being tripped from the previous frame. The flag is cleared at the end of vblank, so once it equals 0 you know the next frame has started.
WaitNotSprite0: lda $2002 and #%01000000 bne WaitNotSprite0 ; wait until sprite 0 not hit
Now we wait until the sprite 0 is hit. How long this takes depends on how far down the screen your sprite 0 is placed.
WaitSprite0: lda $2002 and #%01000000 beq WaitSprite0 ; wait until sprite 0 is hit
When that loop finishes, the PPU is drawing the first pixels of sprite 0 that overlap pixels on the background. We add a small wait loop so the rest of the status bar is drawn, and then change the scroll registers. The rest of the screen down is drawn using those settings.
ldx #$10 WaitScanline: dex bne WaitScanline LDA scroll STA $2005 ; write the horizontal scroll count register LDA #$00 ; no vertical scrolling STA $2005 LDA #%10010000 ; enable NMI, sprites from Pattern Table 0, background from Pattern Table 1 ORA nametable ; select correct nametable for bit 0 STA $2000
So the order is:

1 – set scroll to 0 for status bar
2 – wait for sprite 0 hit = 0
3 – wait for sprite 0 hit = 1
4 – delay so scanline finishes drawing
5 – set scroll for level background

The only other change is to make sure your graphics updating code does not draw over the status bar. The previous DrawNewColumn function handles the graphics updates so it has a few small differences. The starting address is increased by $80 to skip the first 4 rows of background. Then the source address is increased by $04 for the same reason.

Putting It All Together

Download and unzip the sprite0.zip sample files. sprite0.asm is the same as the previous scrolling5.asm file plus the changes covered here. This is another good one to watch in an emulator.

This lesson is largely unrelated to programming, but instead focuses on ROM editing. Using the recently released Exerion 2 ROM we will edit the title screen text to show your own message. The instructions here will work for many games but not all. Some use text/graphics compression or do not have all the letters available for use.

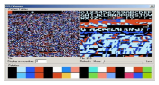
You will need:

- FCEUXDSP emulator with graphics viewer. Sorry Mac/Linux users, the best development emulators are Windows only.
- Hex editor application
- Text editor application
- 10 minutes

Step 1: Backup!
Make a copy of the ROM. Work with this copy. In case you mess up you can go back to the original.

Step 2: Finding the graphics
Load up the game in FCEUXDSP and go to the screen you want to edit. When you are there choose the Debug... menu option and hit the Step Into button in the window that opens. That will tell the emulator to pause so you can keep working without it changing.

Next choose the PPU Viewer... menu option. On one side will be the sprite graphics, and on the other the background graphics. The text is made of background graphics tiles.



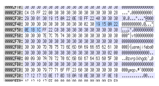
If you put your mouse over one of the letter graphics the tile number will update. Use those tile numbers to make a chart of which number corresponds to which tile number. For Exerion 2, that chart will start like:

A = 0A
B = 0B
C = 0C
D = 0D
E = 0E
F = 0F
G = 10
H = 11
I = 12
etc

Step 3: Finding the text
Use that chart to write out what text you are looking for. For Exerion 2 we are going to edit the "PLAYERS" text for the 2 players option. The hex characters we will be looking for are:

TEXT	P	L	A	Y	E	R	S
HEX	19	15	0A	22	0E	1B	1C

Now that you know what to look for, open up the ROM in your hex editor application. Do a hex search for the string you just figured out. Hopefully it will appear just once in the ROM. In Exerion 2 the 2 PLAYERS text is around hex address 2F40.

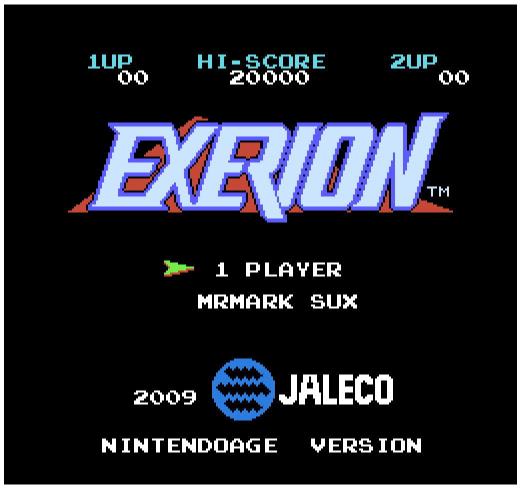


Step 4: Replacing the text
Use your chart again to make a new string, the same length as the previous text. This is very important. The size of the ROM must stay exactly the same size so you cannot add extra characters. If your new string is shorter than the old one then you must add space characters to make it the same length.

For Exerion 2 I want to add a message longer than just the PLAYERS text. Looking before the P in players there is a space, a 2, and lots more spaces. I can safely replace some of those too, as long as I don't add or subtract from the ROM size.

OLD TEXT		2		P	L	A	Y	E	R	S
OLD HEX	30	02	30	19	15	0A	22	0E	1B	1C
NEW TEXT	M	R	M	A	R	K		S	U	X
NEW HEX	16	1B	16	0A	1B	14	30	1C	1E	21

Just select the old hex and delete it, then paste in the new hex. Save your ROM and load it up in the emulator to make sure it worked. If the game doesn't load in the emulator then you likely changed the size of the ROM.



Now that your game is done, test it on real hardware with your PowerPak or make a physical copy with the ReproPak at http://www.retrousb.com

The APU

Music and sound effects on the NES are generated by the **APU** (Audio Processing Unit), the sound chip inside the CPU. The CPU "talks" to the APU through a series of I/O ports, much like it does with the PPU and joypads.

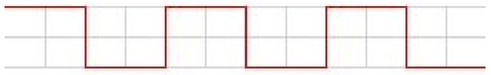
PPU: $2000-$2007
Joypads: $4016-$4017
APU: $4000-$4015, $4017

Channels

The APU has 5 channels: Square 1, Square 2, Triangle, Noise and DMC. The first four play waves and are used in just about every game. The DMC channel plays samples (pre-recorded sounds) and is used less often.

Square

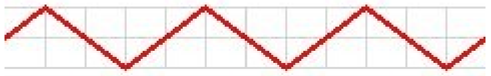
The square channels produce square waveforms. A square wave is named for its shape. It looks like this:



As you can see the wave transitions instantaneously from its high point to its low point (where the lines are vertical). This gives it a hollow sound like a woodwind or an electric guitar.

Triangle

The triangle channel produces triangle waveforms. A triangle wave is also named for its shape. It looks like this:



The sound of a triangle wave is smoother and less harsh than a square wave. On the NES, the triangle channel is often used for bass lines (in low octaves) or a flute (in high octaves). It can also be used for drums.

Noise

The noise channel has a random generator, which makes the waves it produces sound like.. noise. This channel is generally used for percussion and explosion sounds.

DMC

The DMC channel plays samples, which are pre-recorded sounds. It is often used to play voice recordings ("Blades of Steel") and percussion samples. Samples take up a lot of ROM space, so not many games make use of the DMC channel.

Enabling Channels

Before you can use the channels to produce sounds, you need to enable them. Channels are toggled on and off via port $4015:

```
APUFLAGS ($4015)

76543210
   |||||
   ||||+- Square 1 (0: disable; 1: enable)
   |||+-- Square 2
   ||+--- Triangle
   |+---- Noise
   +----- DMC
```

Here are some code examples using $4015 to enable and disable channels:

```
  lda #%00000001
  sta $4015 ;enable Square 1 channel, disable others
  lda #%00010110
  sta $4015 ;enable Square 2, Triangle and DMC channels.  Disable Square 1 and Noise.

  lda #$00
  sta $4015 ;disable all channels

  lda #$0F
  sta $4015 ;enable Square 1, Square 2, Triangle and Noise channels.  Disable DMC.
        ;this is the most common usage.
```

Try opening up some of your favorite games in FCEUXD SP and set a breakpoint on writes to $4015. Take a look at what values are getting written there. If you don't know how to do this, follow these steps:

1. Open FCEUXD SP
2. Load a ROM
3. Open up the Debugger by pressing F1 or going to Tools->Debugger
4. In the top right corner of the debugger, under "BreakPoints", click the "Add..." button
5. Type "4015" in the first box after "Address:"
6. Check the checkbox next to "Write"
7. Set "Memory" to "CPU Mem"
8. Leave "Condition" and "Name" blank and click "OK"

Now FCEUX will pause emulation and snap the debugger anytime your game makes a write (usually via STA) to $4015. The debugger will tell you the contents of the registers at that moment, so you can check what value will be written to $4015. Some games will write to $4015 every frame, and some only do so once at startup. Try resetting the game if your debugger isn't snapping.

What values are being written to $4015? Can you tell what channels your game is using?

Square 1 Channel

Let's make a beep. This week we'll learn how to produce a sound on the Square 1 channel. The Square channels are everybody's favorites because you can control the volume and tone and perform sweeps on them. You can produce a lot of interesting effects using the Squares.

Square 1 is controlled via ports $4000-$4003. The first port, $4000, controls the duty cycle (ie, tone) and volume for the channel. It looks like this:

```
SQ1_ENV ($4000)

76543210
||||||||
||||++++- Volume
|||+----- Saw Envelope Disable (0: use internal counter for volume; 1: use Volume for volume)
||+------ Length Counter Disable (0: use Length Counter; 1: disable Length Counter)
++------- Duty Cycle
```

For our purposes, we will focus on Volume and Duty Cycle. We will set Saw Envelope Disable and Length Counter Disable to 1 and then forget about them. If we leave Saw Envelopes on, the volume of the channel will be controlled by an internal counter. If we turn them off, WE have control of the volume. If WE have control, we can code our own envelopes (much more versatile). Same thing with the Length Counter. If we disable it, we have more control over note lengths. If that didn't make sense, don't worry. It will become clearer later. For now we're just going to disable and forget about them.

Volume controls the channel's volume. It's 4 bits long so it can have a value from 0-F. A volume of 0 silences the channel. 1 is very quiet and F is loud.

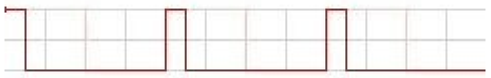
Duty Cycle controls the tone of the Square channel. It's 2 bits long, so there are four possible values:

00 = a weak, grainy tone. Think of the engine sounds in RC Pro-Am. (12.5% Duty)
01 = a solid mid-strength tone. (25% Duty)
10 = a strong, full tone, like a clarinet or a lead guitar (50% Duty)
11 = sounds a lot like 01 (25% Duty negated)

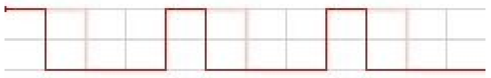
The best way to know the difference in sound is to listen yourself. I recommend downloading FamiTracker and playing with the different Duty settings in the Instrument Editor.

For those interested, Duty Cycle actually refers to the percentage of time that the wave is in "up" position vs. "down" position. Here are some pictures:

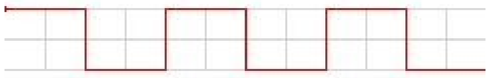
12.5%



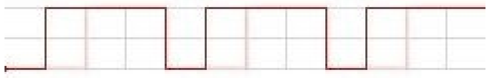
25%



50%



25% negated



Don't sweat it if graphs and waves aren't your thing. Use your ears instead.

Here's a code snippet that sets the Duty and Volume for the Square 1 channel:

```
  lda #%10111111; Duty 10 (50%), volume F (max!)
  sta $4000
```

$4001 controls sweeps for Square 1. We'll skip them for now.

Setting the Note

$4002 and $4003 control the period of the wave, or in other words what note you hear (A, C#, G, etc). Periods are 11-bits long. $4002 holds the low 8-bits and $4003 holds the high 3-bits of the period. We'll get into more detail in a future tutorial, but for now just know that changing the values written to these ports will change the note that is played.

```
SQ1_LO ($4002)

76543210
||||||||
++++++++- Low 8-bits of period

SQ1_HI ($4003)

76543210
||||||||
|||||+++- High 3-bits of period
+++++---- Length Counter
```

The Length Counter, if enabled, controls how long the note is played. We disabled it up in the $4000 section, so we can forget about it for now.

Here is some code that will produce an eternal beep on the Square 1 channel:

```
  lda #%00000001
  sta $4015 ;enable square 1

  lda #%10111111 ;Duty 10, Volume F
  sta $4000
  lda #$C9    ;0C9 is a C# in NTSC mode
  sta $4002
  lda #$00
  sta $4003
```

Putting It All Together
Download and unzip the square1.zip sample files. All the code above is in the square1.asm file. Make sure square1.asm and square1.bat are all in the same folder as NESASM3, then double click square1.bat. That will run NESASM3 and should produce the square1.nes file. Run that NES file in FCEUXD SP to listen to your beep! Edit square1.asm to change the Volume (0 to F), or to change the Duty Cycle for the square wave. Try changing the period to produce different notes.

Last Week: APU Overview and Square 1 Basics

This week: We will learn how to makes sounds with the Square 2 and Triangle channels.

Square 2

Last time we produced a beep on the Square 1 channel by making writes to $4000-$4003. Now we'll learn how to do it with Square 2. This is very easy because Square 1 and Square 2 are almost identical. We control the Square 2 channel with ports $4004-$4007, and they more or less mirror Square 1's $4000-4003.

SQ2_ENV ($4004)

```
76543210
||||||||
||||++++- Volume
|||+----- Saw Envelope Disable (0: use internal counter for volume; 1: use Volume for volume)
||+------ Length Counter Disable (0: use Length Counter; 1: disable Length Counter)
++------- Duty Cycle
```

SQ2_SWEEP ($4005)
Skip this for now. This port, incidentally, is where Square 2 differs from Square 1.

SQ2_LO ($4006)

```
76543210
||||||||
++++++++- Low 8-bits of period
```

SQ2_HI ($4007)

```
76543210
||||||||
|||||+++- High 3-bits of period
+++++---- Length Counter
```

To produce a sound, first we enable the channel via $4015:

```
  lda #%00000010 ;enable Square 2
  sta $4015
```

Then we write to the Square 2 ports:

```
  lda #%00111000 ;Duty Cycle 00, Volume 8 (half volume)
  sta $4004

  lda #$A9   ;$0A9 is an E in NTSC mode
  sta $4006

  lda #$00
  sta $4007
```

Except for sweeps, the Square 2 channel works just like the Square 1 channel.

Triangle

The Triangle channel produces triangle waveforms which have a smooth sound to them. Think of the flute-like melody in the Dragon Warrior overland song. That's the Triangle.

Unlike the Square channels, we have no control over the Triangle channel's volume or tone. It makes only one type of sound and it's either on (playing) or off (silent). We manipulate the Triangle channel via ports $4008-$400B.

```
TRI_CTRL ($4008)

76543210
||||||||
|+++++++- Value
+-------- Control Flag (0: use internal counters; 1: disable internal counters)
```

The triangle channel has two internal counters that can be used to automatically control note duration. We are going to disable them so that we can control note length manually. We will set the **Control Flag** to 1 and forget about it.

When the internal counters are disabled, **Value** controls whether the channel is on or off. To silence the channel, set Value to 0. To turn the channel on (ie, unsilence), set Value to any non-zero value. Here are some examples:

```
  lda #%10000000 ;silence the Triangle channel
  sta $4008

  lda #%10000001 ;Triangle channel on
  sta $4008

  lda #%10001111 ;Triangle channel on
  sta $4008

  lda #%11111111 ;Triangle channel on
  sta $4008
```

Note that the last three examples are functionally the same. Any non-zero value in Value makes the Triangle channel play.

Unused Port
$4009 is unused

Setting the Note
$400A and $400B control the period of the wave, or in other words what note you hear (A, C#, G, etc). Like the Squares, Triangle periods are 11-bits long. $400A holds the low 8-bits and $400B holds the high 3-bits of the period. We'll learn more about periods next week, but for now just know that changing the values written to these ports will change the note that is played.

```
TRI_LO ($400A)

76543210
||||||||
++++++++- Low 8-bits of period

TRI_HI ($400B)

76543210
||||||||
|||||+++- High 3-bits of period
+++++---- Length Counter
```

The Length Counter, if enabled, controls how long the note is played. We disabled it up in the $4008 section, so we can forget about it for now.

Here is some code to play an eternal beep on the Triangle channel:

```
  lda #%00000100 ;enable Triangle channel
  sta $4015

  lda #%10000001 ;disable counters, non-zero Value turns channel on
  sta $4008

  lda #$42   ;a period of $042 plays a G# in NTSC mode.
  sta $400A

  lda #$00
  sta $400B
```

Multiple Beeps

We now know how to use the Square 1, Square 2 and Triangle channels to make sound. It doesn't take too much extra work to make them all play at the same time. We just have to enable all three channels via $4015 and then write to the ports. Here's some code that will play a C#m chord (C# E G#) using the knowledge we have gained up to now:

```
  lda #%00000111  ;enable Sq1, Sq2 and Tri channels
  sta $4015

  ;Square 1
  lda #%00111000  ;Duty 00, Volume 8 (half volume)
  sta $4000
  lda #$C9        ;$0C9 is a C# in NTSC mode
  sta $4002       ;low 8 bits of period
  lda #$00
  sta $4003       ;high 3 bits of period

  ;Square 2
  lda #%01110110  ;Duty 01, Volume 6
  sta $4004
  lda #$A9        ;$0A9 is an E in NTSC mode
  sta $4006
  lda #$00
  sta $4007

  ;Triangle
  lda #%10000001  ;Triangle channel on
  sta $4008
  lda #$42        ;$042 is a G# in NTSC mode
  sta $400A
  lda #$00
  sta $400B
```

Putting It All Together
Download and unzip the triad.zip sample files. All the code above is in the triad.asm file. Make sure triad.asm and triad.bat are all in the same folder as NESASM3, then double click triad.bat. That will run NESASM3 and should produce the triad.nes file. Run that NES file in FCEUXD SP to listen to your C#m chord! Edit triad.asm to change the Volume and Duty Cycle for the square waves. Try changing the Periods to produce different notes.

Try to silence the various channels by either disabling them via $4015 or silencing them via $4000/$4004/$4008.

Finally try writing some code that will silence/unsilence the individual channels based on user input, like so:

A: toggle Square 1 channel on/off
B: toggle Square 2 channel on/off
Select: toggle Triangle channel on/off

Next Week: Periods and Lookup Tables

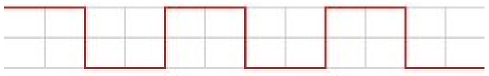
Last Week: Square 2 and Triangle Basics

This week: We will learn about periods and build a period lookup table that spans 8 octaves.

Periods

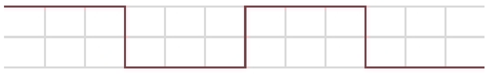
In the last two lessons, I've been giving you the values to plug into the 11-bit periods for the Square and Triangle channels. I haven't been giving you an explanation of what a period is, or where I got those numbers. So this week we're going to learn about periods.

What is a period?
A period refers to the length of a wave, or rather the *time* length of the *repeating* part of a wave. Take a look at this square wave (x-axis is time):

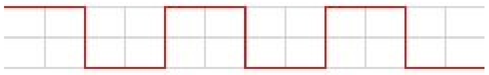


Notice how it is repeating. It starts high and remains high for 2 time units. Then it goes low and remains low for 2 time units. Then it repeats. When we say period, we are talking about the horizontal time length of this repeating wave. In this case, the period is 4 time units. The longer a period is, the lower the note will sound. Conversely, the shorter a period is, the higher the note will sound. Look at these 3 Square waves:

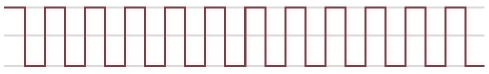
Period = 6 time units



Period = 4 time units



Period = 1 time unit



The top wave has the longest period (6 time units) and it will sound the lowest. The bottom wave has a short period (1 time unit) and will sound higher than the other two.

On the NES, we write an 11-bit period to the APU ports. The smaller the number, the shorter the period, the higher the note. Larger numbers = longer periods = lower notes. Look at the following code snippets that write an 11-bit period to the Square 1 ports:

```
  lda #$C9
  sta $4002
  lda #$05
  sta $4003 ;period $5C9: large number = long period = low note

  ;----

  lda #$09
  sta $4002
  lda #$00
  sta $4003 ;period $009: small number = short period = very high note
```

Periods -> Notes
So how do we know which 11-bit period values correspond to which notes? The magic forumla is:

P = C/(F*16) - 1

P = Period
C = CPU speed (in Hz)

F = Frequency of the note (also in Hz).

The value of C differs between NTSC and PAL machines, which is why a game made for NTSC will sound funny on a PAL NES, and vice-versa.

To find the period values for notes, we will have to look up note frequencies and plug them into the formula. Or we can cross our fingers and hope somebody has already done the work for us and put the answers in an easy-to-read table. Lucky for us a cool fellow named Celius has done just that, for both NTSC and PAL. Here are the charts:

http://www.freewebs.com/the_bott/NotesTableNTSC.txt
http://www.freewebs.com/the_bott/NotesTablePAL.txt

Lookup Tables

It is fairly common practice to store period values in a lookup table. A **lookup table** is a table of pre-calculated data stored in ROM. Like an answer sheet. Lookup tables are used to cut down on complicated, time-consuming calculations. Let's look at a trivial example. Let's say you want a subroutine that takes a value in A and returns 3^A. If you took the brute-force approach, you might write something like this:

```
multiplier .rs 1

; takes a value (0-5) in A and returns 3^A
three_to_the_a:
   bne .not_zero
   lda #$01        ;3^0 is 1
   rts
.not_zero:
   tay
   lda #$03
.loop:
   sta multiplier
   dey
   beq .done
   clc
   adc multiplier
   adc multiplier
   jmp .loop
.done:
   rts
```

It works, but it's not very pretty. Here is how we would do it with a lookup table:

```
;lookup table with pre-calculated answers
powers_of_3:
   .byte 1, 3, 9, 27, 81, 243

three_to_the_a:
   tay
   lda powers_of_3, y
   rts
```

Easier to code. Easier to read. And it runs faster too.

NESASM3 Tip#1: Local Labels
You may have noticed in the above example that I put a period in front of some labels: **.done**, **.loop**, **.not_zero**. NESASM3 treats these as local labels. There are two types of labels: global and local. A **global label** exists across the whole program and must be unique. A **local label** only exists between two global labels. This means that we can reuse the names of local labels - they only need to be unique within their scope. Using local labels saves you the trouble of having to create unique names for common case labels (like looping). I tend to use local labels for all labels that occur within subroutines. To make a label local, stick a period in front of it.

Note Lookup Table

Let's take Celius's tables and turn them into a note lookup table. Period values are 11 bits so we will need to define our lookup table using words. Note that .word is the same as .dw. Here is a note_table for NTSC:

```
;Note: octaves in music traditionally start from C, not A.
;    I've adjusted my octave numbers to reflect this.
note_table:
  .word                                                                  $07F1, $0780, $0713 ; A1-B1
($00-$02)
  .word $06AD, $064D, $05F3, $059D, $054D, $0500, $04B8, $0475, $0435, $03F8, $03BF, $0389 ; C2-B2
($03-$0E)
  .word $0356, $0326, $02F9, $02CE, $02A6, $027F, $025C, $023A, $021A, $01FB, $01DF, $01C4 ; C3-B3
($0F-$1A)
  .word $01AB, $0193, $017C, $0167, $0151, $013F, $012D, $011C, $010C, $00FD, $00EF, $00E2 ; C4-B4
($1B-$26)
  .word $00D2, $00C9, $00BD, $00B3, $00A9, $009F, $0096, $008E, $0086, $007E, $0077, $0070 ; C5-B5
($27-$32)
  .word $006A, $0064, $005E, $0059, $0054, $004F, $004B, $0046, $0042, $003F, $003B, $0038 ; C6-B6
($33-$3E)
  .word $0034, $0031, $002F, $002C, $0029, $0027, $0025, $0023, $0021, $001F, $001D, $001B ; C7-B7
($3F-$4A)
  .word $001A, $0018, $0017, $0015, $0014, $0013, $0012, $0011, $0010, $000F, $000E, $000D ; C8-B8
($4B-$56)
  .word $000C, $000C, $000B, $000A, $000A, $0009, $0008                                    ; C9-F#9
($57-$5D)
```

Notice that at the highest octaves, some notes have the same value (C9 and C#9 for example). This is due to rounding. We lose precision the higher we go, and a lot of the highest notes will sound out of tune as a result. So in songs we probably wouldn't use octaves 8 and 9. These high notes could be utilized for sound effects though, so we'll leave them in.

Once we have a note lookup table, we use the note we want as an index into the table and pull the period values from it, like this:

```
  lda #$0C            ;the 13th entry in the table (A2)
  asl a               ;multiply by 2 because we are indexing into a table of words
  tay
  lda note_table, y   ;read the low byte of the period
  sta $4002           ;write to SQ1_LO
  lda note_table+1, y ;read the high byte of the period
  sta $4003           ;write to SQ1_HI
```

To make it easier to know which index to use for each note, we can create a list of **symbols**:

```
;Note: octaves in music traditionally start at C, not A

;Octave 1
A1 = $00    ;"1" means octave 1.
As1 = $01   ;"s" means "sharp"
Bb1 = $01   ;"b" means "flat".  A# == Bb
B1 = $02

;Octave 2
C2 = $03
Cs2 = $04
Db2 = $04
D2 = $05
;...
A2 = $0C
As2 = $0D
Bb2 = $0D
B2 = $0E

;Octave 3
C3 = $0F
;... etc
```

Now we can use our new symbols instead of the actual index values:

```
  lda #A2             ;A2.  #A2 will evaluate to #$0C
  asl a               ;multiply by 2 because we are indexing into a table of words
  tay
  lda note_table, y   ;read the low byte of the period
  sta $4002           ;write to SQ1_LO
  lda note_table+1, y ;read the high byte of the period
  sta $4003           ;write to SQ1_HI
```

And if later we want to have a series of notes, symbols are much easier to read and alter:

```
sound_data:
  .byte C3, E3, G3, B3, C4, E4, G4, B4, C5 ; Cmaj7 (CEGB)

sound_data_no_symbols:
  .byte $0F, $13, $16, $1A, $1B, $1F, $22, $26, $27 ;same as above, but hard to read. Cmaj7 (CEGB)
```

Low Notes On Squares (Sweep Unit)
One last thing needs to be mentioned. It's very important. It has to do with the Square channels' sweep units. The sweep units can silence the square channels in certain situations (Periods >= $400, our lowest notes), *even when disabled*. We'll have to take a quick look at the sweep unit ports to solve this problem.

```
SQ1_SWEEP ($4001), SQ2_SWEEP ($4005)

76543210
||||||||
|||||+++- Shift
||||+---- Negate
|+++----- Sweep Unit Period
+-------- Enable (1: enabled; 0: disabled)
```

I'm not going to go into how it works now, but the unwanted silencing of low notes can be circumvented by **setting the negate flag**:

```
  lda #$08    ;set Negate flag on the sweep unit
  sta $4001   ;or $4005 for Square 2.
```

If you really want to know why, check the Sweep Unit section of blargg's NES APU Sound Hardware Technical Reference.

What about PAL?
For simplicity, these tutorials are going to use NTSC numbers. Once we finish our sound engine I'll try to whip up a tutorial about adding PAL support.

Putting It All Together
Download and unzip the periods.zip sample files. Make sure periods.asm, periods.chr, note_table.i and periods.bat are all in the same folder as NESASM3, then double click periods.bat. That will run NESASM3 and should produce the periods.nes file. Run that NES file in FCEUXD SP. Use the d-pad to select and play any note from our note table on the Square 1 channel. Controls are as follows:

Up - Play selected note
Down - Stop note
Left - Move selection down a note
Right - Move selection up a note

Homework: Edit periods.asm and add support for the Square 2 and Triangle channels. Allow the user to select between channels and play different notes on all three of them.

Homework #2: Read Disch's document The Frame and NMIs. Pay special attention to the "Take Full Advantage of NMI" section. We are going to use this style of NMI handler with our sound engine. In fact, periods.asm already uses it.

Next Week: Starting our sound engine.
Last Week: Periods, Lookup Tables

This Week: Sound Engine Basics. We will setup the framework to get our sound engine running.

Sound Engine

Now that we know how to get notes to play we can start thinking about our sound engine. What do we want it to be able to do? How will the main program interact with it?

It's good practice to separate the different pieces of your program. The sound engine shouldn't be messing with main program code and vice-versa. If you mix them, your code becomes harder to read, the danger of variable conflicts increases and you open yourself up to hard-to-find bugs. If you keep the different pieces of your program separate, you get the opposite: your code reads well, you avoid variable conflicts, and bugs are easier to trace. Separation also improves your ability to reuse code. If your sound engine only accesses its own internal routines and variables, it makes it that much easier to pull it out from one game and plug it into another.

There has to be some communication between the main program and the sound engine of course. The main program needs to be able to tell the sound engine to do things like: "Play song 2" or "shut up". But we don't want the main program sticking its nose in the sound engine's business. We only want it to issue commands. The sound engine will handle the rest on its own.

To set this up, we will create a small set of subroutines that the main program can use to invoke the sound engine and give it commands. I'll call these subroutines "**entrances**". We want as few entrances into the sound engine as possible. The sound engine itself will have several internal subroutines it can work with, but the main program will only use the entrances.

Entrances
So what will our entrance subroutines be? We need to think about what the main program would need to tell the sound engine to do. Here is a list of entrances we might want for our sound engine:

-Initialize sound engine	(sound_init)
-Load new song/sfx	(sound_load)
-Play a frame of music/sfx	(sound_play_frame)
-Disable sound engine	(sound_disable)

The names in paranthesis are what I'm going to call the subroutines in code. I prefixed them with sound_ for readability. You can tell at a glance that they are sound routines. Here is a rundown of what our commands will do:

sound_init will enable channels and silence them. It will also initialize sound engine variables.

sound_load will take a song/sfx number as input. It will use that song number to index into a table of pointers to song headers. It will read the appropriate header and set up sound engine variables. If that didn't make sense, don't worry. We'll be covering this stuff next week.

sound_play_frame will advance the sound engine by one frame. It will run the note timers, read from the data streams (if necessary), update sound variables and make writes to the APU ports. This stuff will also be covered in future weeks.

sound_disable will disable channels via $4015 and set a disable flag variable.

We already know enough to knock out two of those, sound_init and sound_disable. Let's write them now. We'll write skeleton code for the other entrance subroutines as well. A few things to mention before we do that though:

RAM
A sound engine requires a lot of RAM. A large sound engine might even take up a full page of RAM. For this tutorial, we'll stick all our sound engine variables on the $300 page of RAM. There is nothing magic about this number. I chose $300 for convenience. $000 is your zero-page RAM. $100 is your stack. If you completed the original Nerdy Nights series, $200 will be your Sprite OAM. So $300 is next in line.

ROM
The sound engine itself won't require a lot of ROM space for code, but if you have a lot of music your song data might take up a lot of space. For this reason, I'm going to change our header to give us two 16k PRG-ROM banks, like this:

```
  .inesprg 2 ;2x 16kb PRG code
```

Now we have twice as much ROM space, just in case we need it. BTW, this is the maximum amount of ROM we can have without using a mapper.

Noise Channel
I purposely haven't covered the Noise channel yet. We will want to silence it in our init code though, so I will go ahead

and teach that much. Noise channel volume is controlled via port $400C. It works the same as $4000/$4004 does for the Square channels, except there is no Duty Cycle control:

```
NOISE_ENV ($400C)

76543210
  ||||||
  ||++++- Volume
  |+----- Saw Envelope Disable (0: use internal counter for volume; 1: use Volume for volume)
  +------ Length Counter Disable (0: use Length Counter; 1: disable Length Counter)
```

Like the Squares, we will silence the Noise channel by setting both disable flags, and setting the Volume to 0.

Skeleton Sound Engine

Let's write the entrance subroutines to our sound engine. Most of this code should be very familiar to you if you completed the first three tutorials in this series.

```
   .rsset $0300 ;sound engine variables will be on the $0300 page of RAM

sound_disable_flag  .rs 1   ;a flag variable that keeps track of whether the sound engine is
disabled or not.
                ;if set, sound_play_frame will return without doing anything.

   .bank 0
   .org $8000  ;we have two 16k PRG banks now.  We will stick our sound engine in the first one,
which starts at $8000.

sound_init:
   lda #$0F
   sta $4015   ;enable Square 1, Square 2, Triangle and Noise channels

   lda #$30
   sta $4000   ;set Square 1 volume to 0
   sta $4004   ;set Square 2 volume to 0
   sta $400C   ;set Noise volume to 0
   lda #$80
   sta $4008   ;silence Triangle

   lda #$00
   sta sound_disable_flag  ;clear disable flag

   ;later, if we have other variables we want to initialize, we will do that here.

   rts

sound_disable:
   lda #$00
   sta $4015   ;disable all channels
   lda #$01
   sta sound_disable_flag  ;set disable flag
   rts

sound_load:
   ;nothing here yet
   rts

sound_play_frame:
   lda sound_disable_flag
   bne .done       ;if disable flag is set, don't advance a frame
   ;nothing here yet
.done:
   rts
```

Driving the Sound Engine

We have the framework setup for our sound engine to run. The main program now has subroutines it can call to issue commands to the sound engine. Most of them don't do anything yet, but we can still integrate them into the main program. First we will want to make a call to sound_init somewhere in our reset code:

```
RESET:
  sei
  cld
  ldx #$FF
  txs
  inx

  ;... clear memory, etc

  jsr sound_init

  ;... more reset stuff
```

Next we need something to drive our sound engine. Music is time-based. In any piece of music, assuming a constant tempo, each quarter note needs to last exactly as long as every other quarter note. A whole note has to be exactly as long as four quarter notes. If our sound engine is going to play music, it needs to be time-based as well. We have a subroutine, sound_play_frame, that will advance our sound engine a frame at a time. Now we need to ensure it gets called repeatedly at a regular time interval.

One way to do this is to stick it in the NMI. Recall that when enabled, the NMI will trigger at the start of every vblank. Vblank is the only safe time to write to the PPU, so the NMI is typically full of drawing code. We don't want to waste our precious vblank time running sound code, but what about after we are finished drawing? If we stick our call to sound_play_frame at the end of NMI, after the drawing code, we are set. sound_play_frame gets called once per frame, and we avoid stepping on the PPU's toes. And since sound_play_frame doesn't write to the PPU registers, it doesn't matter if our sound code spills out of vblank.

Let's setup the NMI to drive our sound engine:

```
NMI:
  pha     ;save registers
  txa
  pha
  tya
  pha

  ;do sprite DMA
  ;update palettes if needed
  ;draw stuff on the screen
  ;set scroll

  jsr sound_play_frame    ;run our sound engine after all drawing code is done.
                ;this ensures our sound engine gets run once per frame.

  lda #$00
  sta sleeping            ;did you do your homework and read Disch's document last week?
                ;http://nesdevhandbook.googlepages.com/theframe.html

  pla     ;restore registers
  tay
  pla
  tax
  pla
  rti
```

.include
To further separate our sound engine from the main program, we can keep all our sound engine code in a separate file. NESASM3 gives us a directive .include that we can use to copy a source file into our main program. We actually used this directive last week to include the note_table.i file, which contained our period lookup table.

Using .include to copy a source file into our code is very similar to how we use .incbin to import a .chr file. Assuming our sound engine code is saved in a file called sound_engine.asm, we will add the following code to our main program:

```
  .include "sound_engine.asm"
```

We will continue to include note_table.i, but since it is part of our sound engine we will stick the .include directive in the sound_engine.asm file.

It's not bad practice to use includes a lot. You can pull your joypad routines out and stick them in their own file. You can have separate files for your gamestate code, for your PPU routines and for just about anything else you can think of. Breaking up your code like this will make it easier to find things as your program gets larger and more complicated. It also makes it easier to plug your old routines into new programs.

Putting It All Together
Download and unzip the skeleton.zip sample files. Make sure skeleton.asm, sound_engine.asm, skeleton.chr, note_table.i, sound_data.i and skeleton.bat are all in the same folder as NESASM3, then double click skeleton.bat. That will run NESASM3 and should produce the skeleton.nes file. Run that NES file in FCEUXD SP.

I've hardcoded sound_load and sound_play_frame to play a little melody on the Square 1 channel. It uses a simple frame counter to control note speed. The data for the music is found in the sound_data.i file. Use the controller to interact with the sound engine. Controls are as follows:

A: Play sound from the beginning (sound_load)
B: Initialize the sound engine (sound_init)
Start: Disable sound engine (sound_disable)

Try editing sound_engine.asm to change the data stream that sound_play_frame reads from. The different data streams available are located in sound_data.i. Try adding your own data stream to sound_data.i too. Use the note symbols we made last week and terminate your data stream with $FF.

Homework: Write two new sound engine entrance subroutines for the main program to use:
1. sound_pause: pauses playback of the sound, but retains the current position in the data stream.
2. sound_unpause: if the sound is currently paused, resumes play from the saved position.

Then modify handle_joypad to allow the user to pause/unpause the music.

Homework #2: If the ideas presented in Disch's The Frames and NMIs document are still fuzzy in your head, read it again.

Extra Credit: See if you can understand how my drawing buffer works. Use Disch's document to help you. I won't cover drawing buffers in these sound tutorials, for obvious reasons, but it is definitely worth your time to learn how to use them.

Next week: Sound Data, Pointer Tables, Headers

Last Week: Sound Engine Basics, Skeleton sound engine

This Week: Sound Data, Pointer Tables, Headers

Designing Sound Data

We have a skeleton sound engine in place. Time to pack it with flesh and organs. Before we can play a song, we will have to load a song. Before we can load a song, we will need song data. So our next step is to decide how our sound data will look. We'll need to design our data format, create some test data and then build our engine to read and play that data.

Data Formats
So how do we go about designing a sound data format? A good place to start would be to look at what we are aiming to play. We know that our sound engine will have two basic types of sound data:

1. Music
2. Sound Effects (SFX)

Music plays in the background. It uses the first 4 channels, has a tempo, and usually loops over and over again.

Sound Effects are triggered by game events (eg, ball hitting a paddle) and don't loop indefinitely.

Sound effects have the job of communicating to the player what is going on right now, so they have priority over music. If there is music playing on the Square 2 channel, and a sound effect is also using the Square 2 channel, the sound effect should play instead of the music.

Depending on the game, some sound effects may have higher priority than others. For example, in a Zelda-like game the sound of the player taking damage would have priority over the sound of the player swinging their sword. The former communicates critical information to the player while the latter is just for effect.

Streams
As mentioned above, a sound effect will have to share a channel (or channels) with the music. This is unavoidable because music typically uses all the channels at once, all the time. So when a sound effect starts playing, it has to steal a channel (or more) away from the music. The music will continue to play on the other channels, but the shared channel will go to the sound effect. This creates an interesting problem: if we stop music on a channel to play a sound effect, how do we know where to resume the music on that channel when the sound effect is finished?

The answer is that we don't actually stop the music on the shared channel. We still advance it frame by frame in time with the other music channels. We just don't write its data to the APU ports when a sound effect is playing.

To do this, we will need to keep track of multiple streams of sound data. A data **stream** is a sequence of bytes stored in ROM that the sound engine will read and translate into APU writes. Each stream corresponds to one channel. Music will have 4 data streams - one for each channel. Sound effects will have 2 streams and the sfx themselves will choose which channel(s) they use. So 6 streams total that could potentially be running at the same time. We will number them like this:

```
MUSIC_SQ1 = $00 ;these are stream number constants
MUSIC_SQ2 = $01 ;stream number is used to index into stream variables (see below)
MUSIC_TRI = $02
MUSIC_NOI = $03
SFX_1     = $04
SFX_2     = $05
```

Each stream will need it's own variables in RAM. An easy way to organize this is to reserve RAM space in blocks and use the stream number as an index:

```
;reserve 6 bytes each, one for each stream
stream_curr_sound .rs 6     ;what song/sfx # is this stream currently playing?
stream_channel .rs 6        ;what channel is it playing on?
stream_vol_duty .rs 6       ;volume/duty settings for this stream
stream_note_LO .rs 6        ;low 8 bits of period for the current note playing on the stream
stream_note_HI .rs 6        ;high 3 bits of the note period
;..etc
```

Here we have 6 bytes reserved for each variable. Each stream gets its own byte, for example:

stream_vol_duty+0:	MUSIC_SQ1's volume/duty settings
stream_vol_duty+1:	MUSIC_SQ2's volume/duty
stream_vol_duty+2:	MUSIC_TRI's on/off
stream_vol_duty+3:	MUSIC_NOI's volume
stream_vol_duty+4:	SFX_1's volume/duty
stream_vol_duty+5:	SFX_2's volume/duty

In our sound_play_frame code we will loop through all of the streams using the stream number as an index:

```
  ldx #$00    ;start at stream 0 (MUSIC_SQ1)
.loop:
  ;read from data stream in ROM if necessary
  ;update stream variables based on what we read

  lda stream_vol_duty, x  ;the value in x determines which stream we are working with
  ;do stuff with volume

  lda stream_note_LO, x
  ;do stuff with note periods

  ;do more stuff with other variables

  inx         ;next stream
  cpx #$06    ;loop through all six streams
  bne .loop
```

The music streams will always be running, updating the APU ports with their data frame by frame. When a sound effect starts playing, one or both of the sfx streams will start running. Because our loop processes the SFX streams last, they will write to the APU last and thus overwrite the shared-channel music streams. Our channel conflict is taken care of automatically by the order of our loop!

Music

We now have an idea of how our stream data will be stored in RAM, but there are still many unanswered questions. How do we load a song? How do we know where to find the data streams in ROM? How do we read from those data streams? How do we interpret what we read from those streams?

To answer these questions, we need to make a data format. Let's start with music. What should our music data look like? Most NES music data is divided into three types:

1. **Note** - what note to play: A3, G#5, C2, etc
2. **Note Length** - how long to play the notes: eighth note, quarter note, whole note, etc
3. **Opcodes** - opcodes tell the engine to perform specific tasks: loop, adjust volume, change Duty Cycle for squares, etc
*3.5. **Arguments** - some opcodes will take arguments as input (e.g. how many times to loop, where to loop to).

Ranges
We will need to design our data format to make it easy for the sound engine to differentiate between these three types of data. We do this by specifying ranges. For example, we might say that byte values of $00-$7F represent Notes. $80-$9F are Note Lengths, and $A0-$FF are opcodes. I just made those numbers up. It really doesn't matter what values we use. The important thing is that we have ranges to test against to determine whether a byte is a note, note length or opcode. In our engine code we will have something like this:

```
fetch_byte:
  lda [sound_pointer], y       ;read a byte from the data stream
  bpl .note                    ;if < #$80, it's a Note
  cmp #$A0
  bcc .note_length             ;else if < #$A0, it's a Note Length
.opcode:                        ;else it's an opcode
  ;do Opcode stuff
.note_length:
  ;do Note Length stuff
.note:
  ;do Note stuff
```

This code reads a byte from the sound data and then tests that byte to see which range it falls into. It jumps to a different section of code for each possible range. Almost any data format you create will be divided into ranges like this, whether it be sound data, map data, text data, whatever.

BPL and BMI
These two branch instructions are worth learning if you don't know them already. After BEQ, BNE, BCS and BCC they are the most common branch instructions. They are often used in range-testing.

BPL tests the Negative (N) flag and will branch if it is clear. Think of BPL as Branch if PLus. The N flag will be clear if the last instruction executed resulted in a value less than #$80 (ie, *bit7 clear*).

```
  lda #%01101011
    ; |
    ; +-------- bit7 is clear.  This will clear the N flag.
  bpl .somewhere  ;N flag is clear, so this will branch

  lda #%10010101
    ; |
    ; +-------- bit7 is set.  This will set the N flag
  bpl .somewhere  ;N flag is set, so this will not branch.
```

BMI is the opposite. It tests the N flag and will branch if it is set. Think of BMI as Branch if MInus. The N flag will be set if the last instruction executed resulted in a value greater than or equal to #$80 (ie, *bit7 set*).

```
  lda #%01101011
    ; |
    ; +-------- bit7 is clear.  This will clear the N flag.
  bmi .somewhere  ;N flag is clear, so this will not branch

  lda #%10010101
    ; |
    ; +-------- bit7 is set.  This will set the N flag
  bmi .somewhere  ;N flag is set, so this will branch to the label .somewhere.
```

In the range-testing code above, I used BPL to check if a byte fell into the Note range (00-7F). Go back and check it.

Song Headers

Music on the NES is typically composed of four parts: a Square 1 part, a Square 2 part, a Triangle part and a Noise part. When you want to play a song, you will have the main program issue a command to the sound engine telling it what song you want to play. It will look something like this:

```
  lda #$02
  jsr sound_load  ;load song 2
```

Somehow our sound_load subroutine will have to take that "2" and translate it into a whole song, complete with Square 1, Square 2, Triangle and Noise parts. How does that little number become 4 streams of data? Well, that number is an index into a pointer table, a table of pointers to song headers. The song headers themselves will contain pointers to the individual channels' data streams.

Pointer Tables

A pointer table is a special kind of lookup table. Only instead of holding regular old numerical data a pointer table holds **addresses**. These addresses "point" to the start of data. Addresses on the NES are 16-bit ($0000-$FFFF), so pointer tables are always tables of words. Let's look at an example:

```
pointer_table:
  .word $8000, $ABCD, $CC10, $DF1B
```

Here we have a pointer table. It's four entries long. Each entry is a 16-bit address. Presumably there is data at these four addresses that we will want to read sometime in our program. To read this data we will need to index into the pointer table, grab the address and store it in a zero-page pointer variable and then read using indirect mode:

```
  .rsset $0000

ptr1 .rs 2  ;a 2-byte pointer variable.
       ;The first byte will hold the LO byte of an address
       ;The second byte will hold the HI byte of an address

  .org $E000  ;somewhere in ROM

  lda #$02    ;the third entry in the pointer table ($CC10)
  asl a       ;multiply by 2 because we are indexing into a table of words
  tay
  lda pointer_table, y    ;#$10 - little endian, so words are stored LO-byte first
  sta ptr1
  lda pointer_table+1, y  ;#$CC
  sta ptr1+1

  ;now our pointer is setup in ptr1.  It "points" to address $CC10.  Let's read data from there.

  ldy #$00
  lda [ptr1], y   ;indirect mode.  reads the byte at $CC10
  sta some_variable
  iny
  lda [ptr1], y   ;reads the byte at $CC11
  sta some_other_variable
  iny
  ;... etc
```

This code takes an index and uses it to read an address from our pointer_table. It stores this address in a variable called ptr1 (LO byte first). Then it reads from this address by using indirect mode. We specify indirect mode by putting []'s around our pointer variable. Look at this instruction:

```
  lda [ptr1], y
```

It means "Find the address ptr1 is pointing to. Add Y to that address. Load the value at that address into A".

This is very versatile because we can stick any address we want into our ptr1 variable and read from anywhere! A pointer table is just a lookup table of places we want to read from.

Of course you usually won't know where exactly in the ROM your data will be. So instead of declaring addresses explicitely ($8000, $ABCD, $CC10, etc), you will use labels instead:

```
pointer_table:
  .word data1, data2, data3       ;these entries will evaluate to the 16-bit addresses of the labels
below

data1:
  .byte $FF, $16, $82, $44        ;some random data

data2:
  .byte $0E, $EE, $EF, $16, $23

data3:
  .byte $00, $01
```

Song Header Pointer Table
Our songs work the same way. When the main program tells the sound engine to play a song, it will send the song number with it. This song number is actually an index into a pointer table of song headers:

```
song_headers:
  .word song0_header
  .word song1_header
  .word song2_header
  ;..etc

sound_load:
  asl a                    ;multiply by 2.  we are indexing into a table of pointers (words)
  tay
  lda song_headers, y      ;read LO byte of a pointer from the pointer table.
  sta sound_ptr            ;sound_ptr is a zero page pointer variable
  lda song_headers+1, y    ;read HI byte
  sta sound_ptr+1

  ldy #$00
  lda [sound_ptr], y
  sta some_variable
  iny
  ;...read the rest of the header data
```

Header Data
So what will our song header data look like? At the very least it should tell us:

How many data streams we have (songs will usually have 4, but sfx will have fewer)
Which streams those are (which stream index to use)
Which channels those streams use
Where to find those streams (ie, pointers to the beginning of each stream).
Initial values for those streams (for example, initial volume)

As we add more features to our sound engine, we may expand our headers to initialize those features. Let's start simple. Our headers will look like this:

```
main header:
--------+----------------
```

byte #	what it tells us
00	number of streams
01+	stream headers (one for each stream)

stream headers:

byte #	what it tells us
00	which stream (stream number)
01	status byte (see below)
02	which channel
03	initial volume (and duty for squares)
04-05	pointer to data stream

The **status byte** will be a bit-flag that tells us special information about the stream. For now we will just use bit0 to mark a stream as enabled or disabled. In the future we may use other bits to store other information, such as stream priority.

Stream Status Byte

```
76543210
       |
       +- Enabled (0: stream disabled; 1: enabled)
```

Sample Header

Here is some code showing a sample header:

```
SQUARE_1 = $00 ;these are channel constants
SQUARE_2 = $01
TRIANGLE = $02
NOISE = $03

MUSIC_SQ1 = $00 ;these are stream # constants
MUSIC_SQ2 = $01 ;stream # is used to index into stream variables
MUSIC_TRI = $02
MUSIC_NOI = $03
SFX_1     = $04
SFX_2     = $05

song0_header:
  .byte $04           ;4 streams

  .byte MUSIC_SQ1     ;which stream
  .byte $01           ;status byte (stream enabled)
  .byte SQUARE_1      ;which channel
  .byte $BC           ;initial volume (C) and duty (10)
  .word song0_square1 ;pointer to stream

  .byte MUSIC_SQ2     ;which stream
  .byte $01           ;status byte (stream enabled)
  .byte SQUARE_2      ;which channel
  .byte $38           ;initial volume (8) and duty (00)
  .word song0_square2 ;pointer to stream

  .byte MUSIC_TRI     ;which stream
  .byte $01           ;status byte (stream enabled)
  .byte TRIANGLE      ;which channel
  .byte $81           ;initial volume (on)
  .word song0_tri     ;pointer to stream

  .byte MUSIC_NOI     ;which stream
  .byte $00           ;disabled.  We will have our load routine skip the
                ;  rest of the reads if the status byte disables the stream.
                ;  We are disabling Noise because we haven't covered it yet.
```

```
;these are the actual data streams that are pointed to in our stream headers.
song0_square1:
  .byte A3, C4, E4, A4, C5, E5, A5 ;some notes.  A minor

song0_square2:
  .byte A3, A3, A3, E4, A3, A3, E4 ;some notes to play on square 2

song0_tri:
  .byte A3, A3, A3, A3, A3, A3, A3 ;triangle data
```

Sound Engine Variables
The last thing we need before we can write our sound_load routine is some variables. As mentioned, our sound engine will have several streams running simultaneously. Four will be used for music (one for each tonal channel). Two will be used for sound effects. So we will declare all variables in blocks of 6. Based on our header data, we will need the following variables:

```
stream_curr_sound .rs 6     ;reserve 6 bytes, one for each stream
stream_status .rs 6
stream_channel .rs 6
stream_vol_duty .rs 6
stream_ptr_LO .rs 6
stream_ptr_HI .rs 6
```

sound_load
Now let's write some code to read our header. Pay special attention to the X register. I recommend tracing through the code using the sample header above. Here is our sound_load routine:

```
;-------------------------------------
; load_sound will prepare the sound engine to play a song or sfx.
;  input:
;    A: song/sfx number to play
sound_load:
  sta sound_temp1         ;save song number
  asl a                   ;multiply by 2.  We are indexing into a table of pointers (words)
  tay
  lda song_headers, y     ;setup the pointer to our song header
  sta sound_ptr
  lda song_headers+1, y
  sta sound_ptr+1

  ldy #$00
  lda [sound_ptr], y      ;read the first byte: # streams
  sta sound_temp2         ;store in a temp variable.  We will use this as a loop counter
  iny
.loop:
  lda [sound_ptr], y      ;stream number
  tax                     ;stream number acts as our variable index
  iny

  lda [sound_ptr], y      ;status byte.  1= enable, 0=disable
  sta stream_status, x
  beq .next_stream        ;if status byte is 0, stream disabled, so we are done
  iny

  lda [sound_ptr], y      ;channel number
  sta stream_channel, x
  iny


  lda [sound_ptr], y      ;initial duty and volume settings
  sta stream_vol_duty, x
  iny

  lda [sound_ptr], y      ;pointer to stream data.  Little endian, so low byte first
  sta stream_ptr_LO, x
```

```
    iny

    lda [sound_ptr], y
    sta stream_ptr_HI, x
.next_stream:
    iny

    lda sound_temp1         ;song number
    sta stream_curr_sound, x

    dec sound_temp2         ;our loop counter
    bne .loop
    rts
```

Now our sound_load routine is ready. If the main program calls it, like this:

```
    lda #$00 ;song 0
    jsr sound_load
```

Our sound_load routine will take the value in the A register and use it to fill our music RAM with everything we need to get our song running!

Reading Streams
Once we have our header loaded, we are ready to rock. All of our active streams have pointers to their data stored in their stream_ptr_LO and stream_ptr_HI variables. That's all we need to start reading data from them.

To read data from our data stream, we will first copy the stream pointer into a zero-page pointer variable. Then we will read a byte using indirect mode and range-test it to determine whether it is a note, note length or opcode. If it's a note, we will read from our note_table and store the 11-bit period in RAM. Finally, we will update our stream pointer to point to the next byte in the stream.

First we will need to declare some new variable blocks for the note periods:

```
stream_note_LO .rs 6    ;low 8 bits of period
stream_note_HI .rs 6    ;high 3 bits of period
```

Here is our se_fetch_byte routine (se_ stands for "sound engine"):

```
;--------------------------
; se_fetch_byte reads one byte from a sound data stream and handles it
;  input:
;    X: stream number
se_fetch_byte:
    lda stream_ptr_LO, x    ;copy stream pointer into a zero page pointer variable
    sta sound_ptr
    lda stream_ptr_HI, x
    sta sound_ptr+1

    ldy #$00
    lda [sound_ptr], y      ;read a byte using indirect mode
    bpl .note               ;if <#$80, we have a note
    cmp #$A0                ;else if <#$A0 we have a note length
    bcc .note_length
.opcode:                     ;else we have an opcode
    ;nothing here yet
    jmp .update_pointer
.note_length:
    ;nothing here yet
    jmp .update_pointer
.note:
    asl                     ;multiply by 2 because we are index into a table of words
    sty sound_temp1         ;save our Y register because we are about to destroy it
    tay
    lda note_table, y       ;pull low 8-bits of period and store it in RAM
    sta stream_note_LO, x
```

```
  lda note_table+1, y     ;pull high 3-bits of period from our note table
  sta stream_note_HI, x
  ldy sound_temp1         ;restore the Y register

 ;update our stream pointers to point to the next byte in the data stream
 .update_pointer:
  iny                     ;set index to the next byte in the data stream
  tya
  clc
  adc stream_ptr_LO, x    ;add Y to the LO pointer
  sta stream_ptr_LO, x
  bcc .end
  inc stream_ptr_HI, x    ;if there was a carry, add 1 to the HI pointer.
.end:
  rts
```

Look at the part that updates the stream pointer. After we finish all our reads, Y will hold the index of the last byte read. To be ready for the next frame, we will want to update our pointer to point to the next byte in the data stream. To do this, we increment Y and add it to the pointer. But we have to be careful here. What if our current position is something like this:

stream_ptr: $C3FF
Y: 1

The next position here should be $C400. But ADC only works on the 8-bit level, so if we add 1 to the low byte of the pointer we will get this instead:

stream_ptr: $C300

The FF in the low byte becomes 00, but the high byte remains the same. We need to increment the high byte manually. But how do we know when to increment it and when to leave it alone? Lucky for us, ADC sets the carry flag whenever it makes a FF->00 transition. So we can just check the carry flag after our addition. If it is set, increment the high byte of the pointer. If it is clear, don't increment it. That's what our code above does.

Playing Music

We've loaded our header. We've set up our stream pointers in RAM. We've written a routine that will read bytes from the streams and turn them into notes. Now we need to update sound_play_frame. sound_play_frame will loop through all 6 streams. It will check the status byte to see if they are enabled. If enabled, it will advance the stream by one frame. Here's the code:

```
sound_play_frame:
  lda sound_disable_flag
  bne .done   ;if sound engine is disabled, don't advance a frame

  inc sound_frame_counter
  lda sound_frame_counter
  cmp #$08    ;***change this compare value to make the notes play faster or slower***
  bne .done   ;only take action once every 8 frames.

  ldx #$00                ;our stream index.  start at MUSIC_SQ1 stream
.loop:
  lda stream_status, x    ;check bit 0 to see if stream is enabled
  and #$01
  beq .next_stream        ;if disabled, skip to next stream

  jsr se_fetch_byte       ;read from the stream and update RAM
  jsr se_set_apu          ;write volume/duty, sweep, and note periods of current stream to the APU
ports

.next_stream:
  inx
  cpx #$06                ;loop through all 6 streams.
  bne .loop
```

```
    lda #$00
    sta sound_frame_counter ;reset frame counter so we can start counting to 8 again.
.done:
    rts
```

And here is se_set_apu which will write a stream's data to the APU ports:

```
se_set_apu:
    lda stream_channel, x   ;which channel does this stream write to?
    asl a
    asl a                   ;multiply by 4 so Y will index into the right set of APU ports (see below)
    tay
    lda stream_vol_duty, x
    sta $4000, y
    lda stream_note_LO, x
    sta $4002, y
    lda stream_note_HI, x
    sta $4003, y

    lda stream_channel, x
    cmp #TRIANGLE
    bcs .end        ;if Triangle or Noise, skip this part
    lda #$08        ;else, set negate flag in sweep unit to allow low notes on Squares
    sta $4001, y
.end:
    rts
```

Writing to the APU ports directly like this is actually bad form. We'll learn why in a later lesson.

One thing to pay attention to is how we get our APU port index. We take the channel and multiply it by 4. Recall that we declared constants for our channels:

```
SQUARE_1 = $00 ;these are channel constants
SQUARE_2 = $01
TRIANGLE = $02
NOISE = $03
```

If our stream_channel is $00 (SQUARE_1), we multiply by 4 to get $00. y = 0
    $4000, y = $4000
    $4001, y = $4001
    $4002, y = $4002
    $4003, y = $4003
If our stream_channel is $01 (SQUARE_2), we multiply by 4 to get $04. y = 4
    $4000, y = $4004
    $4001, y = $4005
    $4002, y = $4006
    $4003, y = $4007
If our stream_channel is $02 (TRIANGLE), we multiply by 4 to get $08. y = 8
    $4000, y = $4008
    $4001, y = $4009 (unused)
    $4002, y = $400A
    $4003, y = $400B
If our stream_channel is $03 (NOISE), we multiply by 4 to get $0C. y = C
    $4000, y = $400C
    $4001, y = $400D
    $4002, y = $400E
    $4003, y = $400F

See how everything lines up nicely?

Putting It All Together
Download and unzip the headers.zip sample files. Make sure the following files are in the same folder as NESASM3:

    headers.asm
    sound_engine.asm

headers.chr
note_table.i
song0.i
song1.i
song2.i
song3.i
headers.bat

Double click headers.bat. That will run NESASM3 and should produce the headers.nes file. Run that NES file in FCEUXD SP.

Use the controller to select songs and play them. Controls are as follows:

Up: Play
Down: Stop
Right: Next Song
Left: Previous Song

Song0 is a silence song. It is not selectable. headers.asm "plays" song0 to stop the music when you press down. See song0.i to find out how it works.
Song1 is an evil sounding series of minor thirds.
Song2 is a short sound effect on the Sq2 channel. It uses the SFX_1 stream. Try playing it over the other songs to see how it steals the channel from the music.
Song3 is a simple descending chord progression.

Try creating your own songs and sound effects and add them into the mix. To add a new song you will need to take the following steps:

1) create a song header and song data (use the included songs as reference). Note that data streams are terminated with $FF
2) add your header to the song_headers pointer table at the bottom of sound_engine.asm
3) update the constant NUM_SONGS to reflect the new song number total (also at the bottom of sound_engine.asm)

Although not necessary, I recommend keeping your song data in a separate file like I've done with song0.i, song1.i, song2.i and song3.i. This makes it easier to find the data if you need to edit your song later. If you do this, don't forget to .include your file.

Last Week: Sound Data, Pointer Tables and Headers

This Week: Tempo, Note Lengths, Buffering and Rests

Timing

Last week we put together a huge chunk of the sound engine. We finally got it to play something that resembled a song. But we had big limitations when it came to timing and note lengths. We were using a single frame counter to keep time across all 6 streams of the sound engine. This is a problem because it imposes the music's speed on our sound effects. If you were to use such a system in a real game, your sound effects would speed up or slow down whenever you change to a faster or slower song.

We also made the mistake of advancing the sound engine when our frame counter hit its mark, but skipping it when it didn't. What happens if a game event triggers a sound effect one or two frames after the counter hits its mark? The sound effect won't start until the next time our counter reaches its mark - we have to wait for it! There will be a delay. Not good.

Worst of all, our frame counter also doesn't allow for variable note lengths. Unless every song you write is going to consist of only 32nd notes, this is a problem. It becomes apparent then that we need a more complex timing system.

Tempo

We'll correct the first two problems by ripping out the universal counter and giving each stream it's own private counter. We'll also change our method of counting. Our old method of counting frames and taking action when we reach a certain number is very limited. For example, let's say that we have a song and our frame counter is taking action every 4 frames. Maybe the song sounds a tad faster than we want, so we slow it down by changing the speed to update once every 5 frames. But now the song sounds too slow. The speed we really want is somewhere in between 4 and 5, but we can't get there with our frame counting method. Instead we'll use a ticker.

Ticker
The ticker method involves taking a number (a **tempo**) and adding it to a total, frame by frame. Eventually, that total will wraparound from FF->00 and when it does the carry flag will be set (a **tick**). This carry flag tick will be the signal we look for to advance our stream.

For example, let's say our tempo value is $40 and our total starts at $00. After one frame we will add our tempo to the total. $00 + $40 = $40. Now our total is $40. Another frame goes by (2). We add our tempo to the total again. $40 + $40 = $80. Our total is $80. Another frame goes by (3). $80 + $40 = $C0. Another frame goes by (4). $C0 + $40 = $00. Carry flag is set. TICK! A tick tells us that it is time to advance this stream. When we finish updating, we start adding again until we get another tick.

As you can see, a tempo value of $40 will advance our stream once every 4 frames. If you do some math (256 / 5), you will discover that a tempo of $33 will advance the stream roughly every 5 frames. If $40 is too fast for your song and $33 is too slow, you still have the values $34-$39 to experiment with. Much more versatile! To see why this works, let's see what happens with a tempo value of say $36:

```
$00 + $36 + $36 + $36 + $36 + $36 = $0E (Tick in 5 frames)
$0E + $36 + $36 + $36 + $36 + $36 = $1C (Tick in 5 frames)
$1C + $36 + $36 + $36 + $36       = $02 (Tick in 4 frames)
$02 + $36 + $36 + $36 + $36 + $26 = $10 (Tick in 5 frames)
```

A tempo of $36 produces a tick every 5 frames most of the time, but sometimes it only takes 4 frames. You might think that this disparity would make our song sound uneven, but really a single frame only lasts about 1/60 of a second. Our ears won't notice. It will sound just right to us.

Here is some code that demonstrates how to implement a ticker:

```
stream_tempo .rs 6          ;the value to add to our ticker total each frame
stream_ticker_total .rs 6   ;our running ticker total.

sound_play_frame:
  lda sound_disable_flag
  bne .done   ;if disable flag is set, don't advance a frame

  ldx #$00
.loop:
  lda stream_status, x
  and #$01
  beq .endloop    ;if stream disabled, skip this stream

  ;add the tempo to the ticker total.  If there is a FF-> 0 transition, there is a tick
  lda stream_ticker_total, x
  clc
  adc stream_tempo, x
  sta stream_ticker_total, x
  bcc .endloop    ;carry clear = no tick. if no tick, we are done with this stream

  jsr se_fetch_byte   ;else there is a tick, so do stuff
  ;do more stuff
.endloop:
  inx
  cpx #$06
  bne .loop
.done:
  rts
```

Initializing
Anytime we add a new feature to our sound engine we will want to ask ourselves the following questions:

1) Is this a feature that needs to be initialized for each song/sfx?
2) If so, are the values we use to initialize the feature variable (ie, not necessarily the same for every song/sfx)?

If the answer to question #1 is yes, we will have to update sound_load to initialize the feature.
If the answer to question #2 is also yes, we will have to add a field to the song header format. The values to plug into the initialization are different for each song, so the songs' headers will need to provide those values for us.

In the case of our new timing scheme, we have two variables that need to be initialized: sound_ticker_total and sound_tempo. Of the two, only sound_tempo will be variable. Different songs will have different tempos, but they won't need to have different starting sound_ticker_totals. So we will have to add one new field to our song header format for tempo:

```
main header:
--------+----------------
byte #  | what it tells us
--------+----------------
00      |number of streams
01+     |stream headers (one for each stream)

stream headers:
--------+----------------
byte #  | what it tells us
--------+----------------
00      |which stream (stream number)
01      |status byte
02      |which channel
03      |initial volume (and duty for squares)
04-05   |pointer to data stream
06      |initial tempo
```

Then we will need to edit sound_load to read this new byte for each stream and store it in RAM. We'll also want to initialize stream_ticker_total to some fixed starting value, preferably a high one so that the first tick will happen without a delay. Finally, we will have to update all of our songs to include tempos in their headers.

Note Lengths

We still have the problem of note lengths. Songs are made up of notes of variable length: quarter notes, eighth notes, sixteenth notes, etc. Our sound engine needs to be able to differentiate between different note lengths. But how? We will use note length counters.

Note Length Counters
Think of the fastest note you'd ever need to play, say a 32nd note. Since that will be our fastest note, we'll give it the smallest count possible: $01. The next fastest note is a 16th note. In music, a 16th note equals two 32nd notes. In other words, a 16th note lasts twice as long as a 32nd note. So we will give it a count value that is twice the count value of our 32nd note: $02. The next fastest note is an 8th note. An 8th note equals two 16th notes. It is twice as long as a 16th note. So its count value will be twice that of the 16th note: $04. Going all the way up to a whole note, we can produce a lookup table like this:

```
note_length_table:
  .byte $01   ;32nd note
  .byte $02   ;16th note
  .byte $04   ;8th note
  .byte $08   ;quarter note
  .byte $10   ;half note
  .byte $20   ;whole note
```

We'll add more entries later for things like dotted quarter notes, but for now this is sufficient to get us started.

To play different note lengths, we will give each stream a note length counter:

```
stream_note_length_counter .rs 6
```

When a note is played, for example an 8th note, its count value will be pulled from the note_length_table and stored in the stream's note length counter. Then every time a tick occurs we will decrement the counter. When the note length counter reaches 0, it will signal to us that our note has finished playing and it is time for the next note. To say it another way, a note's count value is simply how many ticks it lasts. An eighth note is 4 ticks long. A quarter note is 8 ticks long. A half note is 16 ticks long ($10).

Note Lengths in Data
Now we need to add note lengths to our sound data. Recall that we specified that byte values in the range of $80-$9F were note lengths:

```
se_fetch_byte:
  lda stream_ptr_LO, x
  sta sound_ptr
  lda stream_ptr_HI, x
  sta sound_ptr+1

  ldy #$00
  lda [sound_ptr], y
  bpl .note                ;if < #$80, it's a Note
  cmp #$A0
  bcc .note_length         ;else if < #$A0, it's a Note Length
.opcode:                    ;else it's an opcode
  ;do Opcode stuff
.note_length:
  ;do note length stuff
.note:
  ;do note stuff
```

So the first byte value that we can use for note lengths is $80. We are going to be reading from a lookup table (note_length_table above), so we should assign the bytes in the same order as the lookup table.

byte	note length
$80	32nd note
$81	16th note
$82	8th note
$83	quarter note
$84	half note
$85	whole note

Now we can use these values in our sound data to represent note lengths:

```
;music data for song 0, square 1 channel
song0_sq1:
  .byte $82, C3 ;play a C eighth note
  .byte $84, D5 ;play a D half note
```

Of course, memorizing which byte value corresponds to which note length is a pain. Let's create some aliases to make it easier on us when we are creating our sound data:

```
;note length constants
thirtysecond = $80
sixteenth = $81
eighth = $82
quarter = $83
half = $84
whole = $85

song0_sq1:
  .byte eighth, C3     ;play a C eighth note
  .byte half, D5       ;play a D half note
```

Pulling from the table
There is a small problem here. Lookup tables index from 0.   This wasn't a problem for note values (C5, D3, G6) because our note range started from 0 ($00-$7f). But our note length data has a range of $80-$9F. Somehow we will need to translate the note length byte that comes from the data stream into a number we can use to index into our table. In other words, we need to figure out a way to turn $80 into $00, $81 into $01, $82 into $02, etc. Anything come to mind?

If you thought "just subtract $80 from the note length value", give yourself a cookie. If you thought "just chop off the 7-bit", give yourself two cookies. Both solutions work, but the second solution is a little bit faster and only takes one instruction to perform:

```
se_fetch_byte:
  lda stream_ptr_LO, x
  sta sound_ptr
  lda stream_ptr_HI, x
  sta sound_ptr+1

  ldy #$00
.fetch:
  lda [sound_ptr], y
  bpl .note                ;if < #$80, it's a Note
  cmp #$A0
  bcc .note_length         ;else if < #$A0, it's a Note Length
.opcode:                    ;else it's an opcode
  ;do Opcode stuff
.note_length:
  ;do note length stuff
  and #%01111111          ;chop off bit7
  sty sound_temp1         ;save Y because we are about to destroy it
  tay
  lda note_length_table, y    ;get the note length count value
  sta stream_note_length_counter, x   ;stick it in our note length counter
  ldy sound_temp1         ;restore Y
  iny                     ;set index to next byte in the stream
  jmp .fetch              ;fetch another byte
.note:
  ;do note stuff
```

Notice that we jump back up to .fetch after we set the note length counter. This is so that we can read the note that will surely follow the note length in the data stream. If we simply stop after setting the note length, we'll know how long to play, but we won't know which note to play!

Here's an updated sound_play_frame routine that implements both the ticker and the note length counters. Notice how the note length counter is only decremented when we have a tick, and we only advance the stream when the note length counter reaches zero:

```
sound_play_frame:
  lda sound_disable_flag
  bne .done   ;if disable flag is set, don't advance a frame

  ldx #$00
.loop:
  lda stream_status, x
  and #$01
  beq .endloop    ;if stream disabled, skip this stream

  ;add the tempo to the ticker total.  If there is a FF-> 0 transition, there is a tick
  lda stream_ticker_total, x
  clc
  adc stream_tempo, x
  sta stream_ticker_total, x
  bcc .endloop    ;carry clear = no tick.  if no tick, we are done with this stream

  dec stream_note_length_counter, x   ;else there is a tick. decrement the note length counter
  bne .endloop    ;if counter is non-zero, our note isn't finished playing yet

  jsr se_fetch_byte   ;else our note is finished.  Time to read from the data stream
  ;do more stuff. set volume, note, sweep, etc
.endloop:
  inx
  cpx #$06
  bne .loop
.done:
  rts
```

We have one last change to make. When we load a new song we will want it to start playing immediately, so we should initialize the stream_note_length_counter in the sound_load routine to do just that. Our sound_play_frame routine decrements the counter and takes action if the result is zero. Therefore, to ensure that our song starts immediately, we should initialize our stream_note_length_counter to $01:

```
  ;somewhere inside the loop of sound_load
  lda #$01
  sta stream_note_length_counter, x
```

And now our engine supports note lengths. But there is still room for improvement. What if we want to play a series of 8th notes? Not an uncommon thing to have in music. Here is how our data would have to look now:

```
sound_data:
  .byte eighth, C5, eighth, E5, eighth, G5, eighth, C6, eighth, E6, eighth, G6, eighth, C7 ;Cmajor
```

That's a lot of "eighth" bytes. Wouldn't it be better to just state "eighth" once, and assume that all notes following it are eighth notes? Like this:

```
sound_data:
  .byte eighth, C5, E5, G5, C6, E6, G6, C7 ;Cmajor
```

That saved us 6 bytes of ROM space. And if you consider that a game may have 20+ songs, each with 4 streams of data, each with potentially several strings of equal-length notes, this kind of change might save us hundreds, maybe even thousands of bytes! Let's do it.

To pull this off, we will have to store the current note length count value in RAM. Then when our note length counter runs to 0, we will refill it with our RAM count value.

```
stream_note_length .rs 6    ;note length count value

;-------

se_fetch_byte:
  lda stream_ptr_LO, x
  sta sound_ptr
  lda stream_ptr_HI, x
  sta sound_ptr+1

  ldy #$00
.fetch:
  lda [sound_ptr], y
  bpl .note                ;if < #$80, it's a Note
  cmp #$A0
  bcc .note_length         ;else if < #$A0, it's a Note Length
.opcode:                    ;else it's an opcode
  ;do Opcode stuff
.note_length:
  ;do note length stuff
  and #%01111111          ;chop off bit7
  sty sound_temp1         ;save Y because we are about to destroy it
  tay
  lda note_length_table, y    ;get the note length count value
  sta stream_note_length, x   ;save the note length in RAM so we can use it to refill the counter
  sta stream_note_length_counter, x   ;stick it in our note length counter
  ldy sound_temp1         ;restore Y
  iny                     ;set index to next byte in the stream
  jmp .fetch              ;fetch another byte
.note:
  ;do note stuff

;--------

sound_play_frame:
  lda sound_disable_flag
  bne .done   ;if disable flag is set, don't advance a frame

  ldx #$00
```

```
.loop:
   lda stream_status, x
   and #$01
   beq .endloop     ;if stream disabled, skip this stream

   ;add the tempo to the ticker total.  If there is a FF-> 0 transition, there is a tick
   lda stream_ticker_total, x
   clc
   adc stream_tempo, x
   sta stream_ticker_total, x
   bcc .endloop     ;carry clear = no tick.  if no tick, we are done with this stream

   dec stream_note_length_counter, x    ;else there is a tick. decrement the note length counter
   bne .endloop     ;if counter is non-zero, our note isn't finished playing yet
   lda stream_note_length, x    ;else our note is finished. reload the note length counter
   sta stream_note_length_counter, x

   jsr se_fetch_byte    ;Time to read from the data stream
   ;do more stuff
.endloop:
   inx
   cpx #$06
   bne .loop
.done:
   rts
```

Adding those 4 lines of code just saved us hundreds of bytes of ROM space. A nice tradeoff for 6 bytes of RAM. Now our data will be made up of "strings", where we have a note length followed by a series of notes:

```
sound_data:
   .byte eighth, C5, E5, G5, C6, E6, G6, quarter, C6 ;six 8th notes and a quarter note
```

Easy to read and easy to write.

Other Note Lengths
Now that everything is setup, we can add more note lengths to our note_length_table. Dotted notes are very common in music. Dotted notes are equal in length to the note plus the next fastest note. For example, a dotted quarter note = a quarter note + an 8th note. A dotted 8th note = an 8th note + a 16th note. Let's add some dotted notes to our table:

```
note_length_table:
   .byte $01   ;32nd note
   .byte $02   ;16th note
   .byte $04   ;8th note
   .byte $08   ;quarter note
   .byte $10   ;half note
   .byte $20   ;whole note
         ;---dotted notes
   .byte $03   ;dotted 16th note
   .byte $06   ;dotted 8th note
   .byte $0C   ;dotted quarter note
   .byte $18   ;dotted half note
   .byte $30   ;dotted whole note?
```

The actual order of our note_length_table doesn't matter. We just have to make sure our aliases are in the same order as the table:

```
;note length constants (aliases)
thirtysecond = $80
sixteenth = $81
eighth = $82
quarter = $83
half = $84
whole = $85
d_sixteenth = $86
d_eighth = $87
```

```
d_quarter = $88
d_half = $89
d_whole = $8A   ;don't forget we are counting in hex
```

Your music will determine what other entries you'll need to add to your note length table. If one of your songs has a really really long note, like 3 whole notes tied together, add it to the table ($60) and make an alias for it (whole_x3). If your song contains a note that is seven 8th notes long (a half note plus a dotted quarter note tied together), add it to the table ($1C) and make an alias for it (seven_eighths).

Buffering APU Writes

Before, we've been writing to the APU one stream at a time. If two different streams shared a channel, they would both write to the same APU ports. If three streams were to share a channel, which is possible if there are two different sound effects loaded into SFX_1 and SFX_2, all three would write to the same APU ports in the same frame. This is bad practice. It can also cause some unwanted noise on the square channels.

A better method is to buffer our writes. Instead of writing to the APU ports directly, each stream will instead write its data to temporary ports in RAM. We'll keep our loop order, so sfx streams will still overwrite the music streams. Then when all the streams are done, we will copy the contents of our temporary RAM ports directly to the APU ports all at once. This ensures that the APU ports only get written to once per frame max. To do this, we first need to reserve some RAM space for our temporary port variables:

```
soft_apu_ports .rs 16
```

We reserved 16 bytes for our temporary ports. Each one corresponds to an APU port:

```
soft_apu_ports+0  -> $4000      ;Square 1 ports
soft_apu_ports+1  -> $4001
soft_apu_ports+2  -> $4002
soft_apu_ports+3  -> $4003

soft_apu_ports+4  -> $4004      ;Square 2 ports
soft_apu_ports+5  -> $4005
soft_apu_ports+6  -> $4006
soft_apu_ports+7  -> $4007

soft_apu_ports+8  -> $4008      ;Triangle ports
soft_apu_ports+9  -> $4009 (unused)
soft_apu_ports+10 -> $400A
soft_apu_ports+11 -> $400B

soft_apu_ports+12 -> $400C      ;Noise ports
soft_apu_ports+13 -> $400D (unused)
soft_apu_ports+14 -> $400E
soft_apu_ports+15 -> $400F
```

Let's implement this by working backwards. First we will edit sound_play_frame and pull our call to se_set_apu out of the loop. We do this because we only want to write to the APU once, after all the streams are done looping:

```
;--------------------------
; sound_play_frame advances the sound engine by one frame
sound_play_frame:
  lda sound_disable_flag
  bne .done   ;if disable flag is set, don't advance a frame

  ldx #$00
.loop:
  lda stream_status, x
  and #$01    ;check whether the stream is active
  beq .endloop  ;if the channel isn't active, skip it

  ;add the tempo to the ticker total.  If there is a FF-> 0 transition, there is a tick
  lda stream_ticker_total, x
  clc
  adc stream_tempo, x
  sta stream_ticker_total, x
  bcc .endloop    ;carry clear = no tick.  if no tick, we are done with this stream
```

```
    dec stream_note_length_counter, x   ;else there is a tick. decrement the note length counter
    bne .endloop    ;if counter is non-zero, our note isn't finished playing yet
    lda stream_note_length, x   ;else our note is finished. reload the note length counter
    sta stream_note_length_counter, x

    jsr se_fetch_byte
    ;snip

.endloop:
    inx
    cpx #$06
    bne .loop

    jsr se_set_apu
.done:
    rts
```

Next we will modify se_set_apu to copy the temporary APU ports to the real APU ports:

```
se_set_apu:
    ldy #$0F
.loop:
    cpy #$09
    beq .skip   ;$4009 is unused
    cpy #$0D
    beq .skip   ;$400D is unused

    lda soft_apu_ports, y
    sta $4000, y
.skip:
    dey
    bpl .loop   ;stop the loop when Y is goes from $00 -> $FF

    rts
```

Now we have to write the subroutine that will populate the temporary APU ports with a stream's data. This part will get more complicated as we add more features to our sound engine, but for now it's quite simple:

```
se_set_temp_ports:
    lda stream_channel, x
    asl a
    asl a
    tay

    lda stream_vol_duty, x
    sta soft_apu_ports, y       ;vol

    lda #$08
    sta soft_apu_ports+1, y     ;sweep

    lda stream_note_LO, x
    sta soft_apu_ports+2, y     ;period LO

    lda stream_note_HI, x
    sta soft_apu_ports+3, y     ;period HI
    rts
```

We will make the call to se_set_temp_ports after our call to se_fetch_byte, where the old se_set_apu call was before we snipped it out of the loop. Notice that we don't bother to check the channel before writing the sweep. se_set_apu takes care of this part for us. There's no harm in writing these values to RAM, so we'll avoid branching here to simplify the code.

Crackling Sounds
Writing to the 4th port of the Square channels ($4003/$4007) has the side effect of resetting the sequencer. If we

write here too often, we will get a nasty crackling sound out of our Squares. This is not good.

The way our engine is setup now, we call se_set_apu once per frame. se_set_apu writes to $4003/$4007, so these ports will get written to once per frame. This is too often. We need to find a way to write here less often. We will do this by cutting out redundant writes. If the value we want to write this frame is the same as the value written last frame, skip the write.

First we will need to keep track of what was last written to the ports. This will require some new variables:

```
sound_sq1_old .rs 1  ;the last value written to $4003
sound_sq2_old .rs 1  ;the last value written to $4007
```

Whenever we write to one of these ports, we will also write the value to the corresponding sound_port4_old variable. Saving this value will allow us to compare against it next frame. To implement this, we will have to unroll our loop in se_set_apu:

```
se_set_apu:
.square1:
  lda soft_apu_ports+0
  sta $4000
  lda soft_apu_ports+1
  sta $4001
  lda soft_apu_ports+2
  sta $4002
  lda soft_apu_ports+3
  sta $4003
  sta sound_sq1_old       ;save the value we just wrote to $4003
.square2:
  lda soft_apu_ports+4
  sta $4004
  lda soft_apu_ports+5
  sta $4005
  lda soft_apu_ports+6
  sta $4006
  lda soft_apu_ports+7
  sta $4007
  sta sound_sq2_old       ;save the value we just wrote to $4007
.triangle:
  lda soft_apu_ports+8
  sta $4008
  lda soft_apu_ports+10
  sta $400A
  lda soft_apu_ports+11
  sta $400B
.noise:
  lda soft_apu_ports+12
  sta $400C
  lda soft_apu_ports+14
  sta $400E
  lda soft_apu_ports+15
  sta $400F
  rts
```

Now we have a variable that will keep track of the last value written to a channel's 4th port. The next step is to add a check before we write:

```
se_set_apu:
.square1:
  lda soft_apu_ports+0
  sta $4000
  lda soft_apu_ports+1
  sta $4001
  lda soft_apu_ports+2
  sta $4002
  lda soft_apu_ports+3
  cmp sound_sq1_old       ;compare to last write
  beq .square2            ;don't write this frame if they were equal
  sta $4003
  sta sound_sq1_old       ;save the value we just wrote to $4003
.square2:
  lda soft_apu_ports+4
  sta $4004
  lda soft_apu_ports+5
  sta $4005
  lda soft_apu_ports+6
  sta $4006
  lda soft_apu_ports+7
  cmp sound_sq2_old
  beq .triangle
  sta $4007
  sta sound_sq2_old       ;save the value we just wrote to $4007
.triangle:
  lda soft_apu_ports+8
  sta $4008
  lda soft_apu_ports+10   ;there is no $4009, so we skip it
  sta $400A
  lda soft_apu_ports+11
  sta $400B
.noise:
  lda soft_apu_ports+12
  sta $400C
  lda soft_apu_ports+14   ;there is no $400D, so we skip it
  sta $400E
  lda soft_apu_ports+15
  sta $400F
  rts
```

Finally we have to consider initialization. The only case we really have to worry about is the first time a song is played in the game. Consider what happens if we initialize the sound_sq1_old and sound_sq2_old variables to $00. We are essentially saying that on startup (RESET) the last byte written to $4003/$4007 was a $00, which isn't true of course. On startup, no write has ever been made to these ports. If we initialize to $00, and if the first note of the first song played has a $00 for the high 3 bits of its period, it will get skipped. That is not what we want. Instead, we should initialize these variables to some value that will never be written to $4003/$4007, like $FF. This ensures that the first note(s) played in the game won't be skipped.

```
sound_init:
  lda #$0F
  sta $4015   ;enable Square 1, Square 2, Triangle and Noise channels

  lda #$00
  sta sound_disable_flag  ;clear disable flag
  ;later, if we have other variables we want to initialize, we will do that here.
  lda #$FF
  sta sound_sq1_old
  sta sound_sq2_old
se_silence:
  lda #$30
  sta soft_apu_ports      ;set Square 1 volume to 0
  sta soft_apu_ports+4    ;set Square 2 volume to 0
  sta soft_apu_ports+12   ;set Noise volume to 0
  lda #$80
  sta soft_apu_ports+8     ;silence Triangle

  rts
```

Rests

The final topic we will cover this lesson is rests. A **rest** is a period of silence in between notes. Like notes, rests can be of variable length: quarter rest, half rest, whole rest, etc. In other words a rest is a silent note.

So how will we implement it? We will handle rests by considering a rest to be special case note. We will give the rest a dummy period in our note table. Then, when we fetch a byte from the data stream and determine the byte to be a note, we will add an extra check to see if that note is a rest. If it is, we will make sure that it shuts up the stream.

First let's add the rest to our note table. We will give it a dummy period. It doesn't really matter what value we use. I'm going to give it a period of $0000. We will also want to add the rest to our list of note aliases:

```
note_table:
  .word $07F1, $0780, etc...
  ;....more note table values here
  .word $0000 ;rest.  Last entry

;Note: octaves in music traditionally start at C, not A
A1 = $00    ;the "1" means Octave 1
As1 = $01   ;the "s" means "sharp"
Bb1 = $01   ;the "b" means "flat"  A# == Bb, so same value
B1 = $02
;..... other aliases here
F9 = $5c
Fs9 = $5d
Gb9 = $5d
rest = $5e
```

Now we can use the symbol "rest" in our music data. "rest" will evaluate to the value $5E, which falls within our note range ($00-$7F). When our sound engine encounters a $5E in the data stream, it will pull the period ($0000) from the note table and store it in RAM. A period of $0000 is actually low enough to silence the square channels, but the triangle channel is still audible at this period so we have more work to do.

Checking for a rest

When we encounter a rest, we will want to tell the sound engine to shut this stream up until the next note. The rest functions differently from all the other notes, so we will need to make a special check for it in our code. We will make a subroutine se_check_rest to do this for us:

```
se_fetch_byte:
  lda stream_ptr_LO, x
  sta sound_ptr
  lda stream_ptr_HI, x
  sta sound_ptr+1

  ldy #$00
.fetch:
  lda [sound_ptr], y
  bpl .note                 ;if < #$80, it's a Note
  cmp #$A0
  bcc .note_length          ;else if < #$A0, it's a Note Length
.opcode:                     ;else it's an opcode
  ;do Opcode stuff
.note_length:
  ;do Note Length stuff
.note:
  ;do Note stuff
  sty sound_temp1      ;save our index into the data stream
  asl a
  tay
  lda note_table, y
  sta stream_note_LO, x
  lda note_table+1, y
  sta stream_note_HI, x
  ldy sound_temp1      ;restore data stream index

  ;check if it's a rest
  jsr se_check_rest
```

```
.update_pointer:
   iny
   tya
   clc
   adc stream_ptr_LO, x
   sta stream_ptr_LO, x
   bcc .end
   inc stream_ptr_HI, x
.end:
   rts
```

se_check_rest will check to see if the note value is equal to $5E or not. If it is, we will need to tell the sound engine to silence the stream. If the note isn't equal to $5E, we can go on our merry way.

How will we silence our stream then? This is actually a little complicated. Recall that the stream's volume (stream_vol_duty) is set in the song's header. se_set_temp_ports copies the value of stream_vol_duty to soft_apu_ports. If we have se_check_rest modify the stream_vol_duty variable directly (set it to 0 volume), the old volume value disappears. We won't know what to restore it to when we are done with our rest. Oh no!

What we will want to do instead is leave stream_vol_duty alone. We will copy it into soft_apu_ports every frame as usual. Then, after the copy we will check to see if we are currently resting. If we are, we will make another write soft_apu_ports with a value that will set the volume to 0. Make sense?

stream_status
To do this we will need to keep track of our resting status in a variable. If our sound engine encounters a $5E in the data stream, we'll turn our resting status on. If it's not, we'll turn our resting status off. There are only two possibilities: on or off. Rather than declare a whole new block of variables and waste six bytes of RAM, let's assign one of the bits in our stream_status variable to be our rest indicator:

```
Stream Status Byte
76543210
      ||
      |+- Enabled (0: stream disabled; 1: enabled)
      +-- Rest (0: not resting; 1: resting)
```

Our new subroutine se_check_rest will be in charge of setting or clearing this bit of the status byte:

```
se_check_rest:
   lda [sound_ptr], y  ;read the note byte again
   cmp #rest           ;is it a rest? (==$5E)
   bne .not_rest
   lda stream_status, x
   ora #%00000010      ;if so, set the rest bit in the status byte
   bne .store          ;this will always branch.  bne is cheaper than a jmp.
.not_rest:
   lda stream_status, x
   and #%11111101      ;clear the rest bit in the status byte
.store:
   sta stream_status, x
   rts
```

Then we modify se_set_temp_ports to check the rest bit and silence the stream if it is set:

```
se_set_temp_ports:
   lda stream_channel, x
   asl a
   asl a
   tay

   lda stream_vol_duty, x
   sta soft_apu_ports, y       ;vol

   lda #$08
   sta soft_apu_ports+1, y     ;sweep
```

```
  lda stream_note_LO, x
  sta soft_apu_ports+2, y     ;period LO

  lda stream_note_HI, x
  sta soft_apu_ports+3, y     ;period HI

  ;check the rest flag. if set, overwrite volume with silence value
  lda stream_status, x
  and #%00000010
  beq .done       ;if clear, no rest, so quit
  lda stream_channel, x
  cmp #TRIANGLE   ;if triangle, silence with #$80
  beq .tri
  lda #$30        ;else, silence with #$30
  bne .store      ;this will always branch.  bne is cheaper than a jmp.
.tri:
  lda #$80
.store:
  sta soft_apu_ports, y
.done:
  rts
```

That's it. Now our engine supports rests! They work just like notes, so their lengths are controlled with note lengths:

```
song_data:  ;this data has two quarter rests in it.
  .byte half, C2, quarter, rest, eighth, D4, C4, quarter, B3, rest
```

Putting It All Together
Download and unzip the tempo.zip sample files. Make sure the following files are in the same folder as NESASM3:

tempo.asm
sound_engine.asm
tempo.chr
note_table.i
note_length_table.i
song0.i
song1.i
song2.i
song3.i
song4.i
song5.i
tempo.bat

Double click tempo.bat. That will run NESASM3 and should produce the tempo.nes file. Run that NES file in FCEUXD SP.

Use the controller to select songs and play them. Controls are as follows:

Up: Play
Down: Stop
Right: Next Song/SFX
Left: Previous Song/SFX

Song0 is a silence song. Not selectable. tempo.asm "plays" song0 to stop the music when you press down. See song0.i to find out how it works.
Song1 is last week's evil sounding series of minor thirds, but much faster now thanks to tempo settings.
Song2 is the same short sound effect from last week.
Song3 is a simple descending chord progression. We saved some bytes in the triangle data using note lengths (compare to last week's file)
Song4 is a new song that showcases variable note lengths and rests.
Song5 is a short sound effect. It plays 10 notes extremely fast. Play it over songs and see how it steals the SQ2 channel from the music.

Try creating your own songs and sound effects and add them into the mix. To add a new song you will need to take the following steps:

1) create a song header and song data (use the included songs as reference). Don't forget to add tempos for each

stream in your header. Data streams are terminated with $FF.
2) add your header to the song_headers pointer table at the bottom of sound_engine.asm
3) update the constant NUM_SONGS to reflect the new song number total (also at the bottom of sound_engine.asm)

Although not necessary, I recommend keeping your song data in a separate file like I've done with song0.i, song1.i, song2.i and song3.i. This makes it easier to find the data if you need to edit your song later. If you do this, don't forget to .include your file.

Last Week: Tempo, Note Lengths, Buffering and Rests

This Week: Volume Envelopes

Volume Envelopes

This week we will add volume envelopes to our engine. A **volume envelope** is a series of volume values that are applied to a note one frame at a time. For example, if we had a volume envelope that looked like this:

```
F E D C 9 5 0
```

Then whenever we played a note, it would have a volume of F on the first frame, a volume of E on the second frame, then D, then C, then 9, then 5 until it is finally silenced with a volume of 0 on the 7th frame. Applying this volume envelope on our notes would give them a sharp, short staccato feel. Conversely, if we had a volume envelope that looked like this:

```
1 1 2 2 3 3 4 4 7 7 8 8 A A C C D D E E F F F
```

Each note would start very quietly and fade in to full volume. Look at this volume envelope:

```
D D D C B 0 0 0 0 0 0 0 0 6 6 6 5 4 0
```

Here we start at a high volume (D) and let it ring for 5 frames. Then we silence the note for 8 frames. Then the note comes back at a very low volume for 5 frames. Notes using this volume envelope would sound like they had an faint echo.

As you can see, volume envelopes are pretty cool. We can get a lot of different sounds out of them. Let's add them in.

Channels
Volume envelopes are best suited for the square and noise channels where we have full control of the volume. The triangle channel on the other hand doesn't allow much volume control. It only has two settings: full blast and off. We can still apply volume envelopes in a limited way though. Consider these two volume envelopes:

```
0F 0E 0D 0C 09 05 00
```

```
04 04 05 05 06 06 07 07 08 08 09 09 0A 0A 00
```

These two envelopes would have a vastly different sound on the square channels, but to the triangle they look like this:

```
On On On On On On Off
On On On On On On On On On On On On On On Off
```

You don't get the subtle shifts in volume, but you do get a different length. We can use volume envelopes that end in 00 to control when the triangle key-off occurs. Not as cool as full volume control, but still useful.

Defining volume envelopes
First let's define some volume envelopes so we have some data to work with. We'll use some of the examples from above:

```
se_ve_1:
  .byte $0F, $0E, $0D, $0C, $09, $05, $00
  .byte $FF
se_ve_2:
  .byte $01, $01, $02, $02, $03, $03, $04, $04, $07, $07
  .byte $08, $08, $0A, $0A, $0C, $0C, $0D, $0D, $0E, $0E
  .byte $0F, $0F
  .byte $FF
se_ve_3:
  .byte $0D, $0D, $0D, $0C, $0B, $00, $00, $00, $00, $00
  .byte $00, $00, $00, $00, $06, $06, $06, $05, $04, $00
  .byte $FF
```

Notice that I terminated each envelope with $FF. We need some terminator value so the engine will know when we've reached the end of the envelope. We could have used any value, but $FF is pretty common.

Next we will make a pointer table that holds the addresses of our volume envelopes:

```
volume_envelopes:
  .word se_ve_1, se_ve_2, se_ve_3
```

Declaring variables
In order to apply a volume envelope to a particular stream, we will need a variable that tells us which one to use. We will also need an index variable that tells us our current position within the volume envelope:

```
stream_ve .rs 6         ;current volume envelope
stream_ve_index .rs 6   ;current position within the volume envelope
```

stream_ve will tell us which volume envelope to use. Code-wise, it will act as an index into our pointer table so we know where to read from. Sound familiar? It works the same way as "song number" did for loading a song. We aren't there yet, but here's a peek at how we will use these variables to read from the volume envelopes. (x holds the stream number):

```
  sty sound_temp1             ;save y because we are about to destroy it.

  lda stream_ve, x            ;which volume envelope?
  asl a                       ;multiply by 2 because we are indexing into a table of addresses
(words)
  tay
  lda volume_envelopes, y     ;get the low byte of the address from the pointer table
  sta sound_ptr
  lda volume_envelopes+1, y   ;get the high byte of the address
  sta sound_ptr+1

  ldy stream_ve_index, x      ;our current position within the volume envelope.
  lda [sound_ptr], y          ;grab the value.
  ;check against $FF (our termination value)
  ;set the volume
  ;increment stream_ve_index
  ;etc
```

Compare this code to the beginning of the sound_load routine. Are you starting to see a pattern?

Initializing
Whenever we add a new feature, we need to consider how we should initialize it. Every stream in our music data will potentially have a different volume envelope, so we should add a volume envelope field to our header. Volume envelopes will deprecate our old "initial volume" field, but we will still need to have duty cycle info, so we'll just rename that field:

```
main header:
--------+----------------
byte #  | what it tells us
--------+----------------
00      |number of streams
01+     |stream headers (one for each stream)

stream headers:
--------+----------------
byte #  | what it tells us
--------+----------------
00      |which stream (stream number)
01      |status byte
02      |which channel
```

```
03    | initial duty (for triangle, set the 7bit)
04    | volume envelope
05-06 | pointer to data stream
07    | initial tempo
```

To read this data from the header, we will have to insert the following code into our sound_load routine (after reading the duty):

```
  lda [sound_ptr], y      ;the stream's volume envelope
  sta stream_ve, x
  iny
```

Notes will always start from the beginning of the volume envelope, so we can just initialize stream_ve_index to 0:

```
  lda #$00
  sta stream_ve_index, x
```

Now we just need to make sure to assign volume envelopes to all the streams in our song data and we're ready to go:

```
song5_header:
  .byte $01           ;1 stream

  .byte SFX_1         ;which stream
  .byte $01           ;status byte (stream enabled)
  .byte SQUARE_2      ;which channel
  .byte $70           ;initial duty (01).  Initial volume deprecated.
  .byte $00           ;the first volume envelope (se_ve_1)
  .word song5_square2 ;pointer to stream
  .byte $FF           ;tempo..very fast tempo
```

Remember that you can always create descriptive aliases for your volume envelopes if you don't want to remember which number is which:

```
;volume envelope aliases
ve_short_staccato = $00
ve_fade_in = $01
ve_blip_echo = $02

song5_header:
  .byte $01           ;1 stream

  .byte SFX_1         ;which stream
  .byte $01           ;status byte (stream enabled)
  .byte SQUARE_2      ;which channel
  .byte $7F           ;initial duty (01).  Initial volume deprecated.
  .byte ve_short_staccato   ;the first volume envelope (se_ve_1)
  .word song5_square2 ;pointer to stream
  .byte $FF           ;tempo..very fast tempo
```

Using aliases is a good idea because the assembler will give you an error if you mistype your alias. If you mistype your number, and it is still a valid number, the assembler won't know there's a problem and will assemble it. This kind of bug in your data can be hard to trace.

Implementing Volume Envelopes

To implement volume envelopes, we need to modify the code where we set the volume. Instead of using a fixed value like we were doing before, we need to read from our current position in the volume envelope and use that value instead. Our volume code is starting to get a little complicated, so let's pull it out into its own subroutine. This will make our code easier to follow:

```
;----------------------------------------------------
; se_set_temp_ports will copy a stream's sound data to the temporary apu variables
;    input:
;       X: stream number
```

```
se_set_temp_ports:
  lda stream_channel, x
  asl a
  asl a
  tay

  jsr se_set_stream_volume    ;let's stick all of our volume code into a new subroutine
                   ;less cluttered that way

  lda #$08
  sta soft_apu_ports+1, y     ;sweep

  lda stream_note_LO, x
  sta soft_apu_ports+2, y     ;period LO

  lda stream_note_HI, x
  sta soft_apu_ports+3, y     ;period HI

  rts
```

What should our new subroutine se_set_stream_volume do? First it needs to read a value from our stream's volume envelope. Then it needs to modify the stream's volume using that value. Then we need to update our position within the volume envelope. Finally it needs to check to see if we are resting, and silence the stream if we are (we wrote this code last week). It looks something like this (new code in red):

```
se_set_stream_volume:
  sty sound_temp1             ;save our index into soft_apu_ports (we are about to destroy y)

  lda stream_ve, x            ;which volume envelope?
  asl a                       ;multiply by 2 because we are indexing into a table of addresses
(words)
  tay
  lda volume_envelopes, y     ;get the low byte of the address from the pointer table
  sta sound_ptr               ;put it into our pointer variable
  lda volume_envelopes+1, y   ;get the high byte of the address
  sta sound_ptr+1

.read_ve:
  ldy stream_ve_index, x      ;our current position within the volume envelope.
  lda [sound_ptr], y          ;grab the value.
  cmp #$FF
  bne .set_vol                ;if not FF, set the volume
  dec stream_ve_index, x      ;else if FF, go back one and read again
  jmp .read_ve                ;  FF essentially tells us to repeat the last
                   ; volume value for the remainder of the note
.set_vol:
  sta sound_temp2             ;save our new volume value (about to destroy A)
  lda stream_vol_duty, x      ;get current vol/duty settings
  and #$F0                    ;zero out the old volume
  ora sound_temp2             ;OR our new volume in.

  ldy sound_temp1             ;get our index into soft_apu_ports
  sta soft_apu_ports, y       ;store the volume in our temp port
  inc stream_ve_index, x      ;set our volume envelop index to the next position

.rest_check:
  ;check the rest flag. if set, overwrite volume with silence value
  lda stream_status, x
  and #%00000010
  beq .done                   ;if clear, no rest, so quit
  lda stream_channel, x
  cmp #TRIANGLE               ;if triangle, silence with #$80
  beq .tri                    ;else, silence with #$30
```

```
    lda #$30
    bne .store                  ;this always branches.  bne is cheaper than a jmp
.tri:
    lda #$80
.store:
    sta soft_apu_ports, y
.done:
    rts
```

After we read a value from our volume envelope, we AND stream_vol_duty with #$F0. This has the nice effect of clearing the old volume while preserving our squares' duty cycle settings. But we need to be careful here. Recall that the triangle channel's on/off status is controlled by the low 7 bits of the port:

```
TRI_CTRL ($4008)

76543210
||||||||
|+++++++- Value
+-------- Control Flag (0: use internal counters; 1: disable internal counters)
```

If any of those Value bits are set, the triangle channel will be considered on. Consider what happens if bit 4, 5 or 6 happen to be set. In this case, ANDing with #$F0 won't turn the triangle channel off. If the volume we pull from the volume envelope is 0, it won't silence our triangle channel because bit 4, 5 or 6 will still be set. If we are careful not to set these bits in our song headers, the problem should never come up. But for completeness we should fix it:

```
se_set_stream_volume:
    sty sound_temp1             ;save our index into soft_apu_ports (we are about to destroy y)

    lda stream_ve, x            ;which volume envelope?
    asl a                       ;multiply by 2 because we are indexing into a table of addresses
(words)
    tay
    lda volume_envelopes, y     ;get the low byte of the address from the pointer table
    sta sound_ptr               ;put it into our pointer variable
    lda volume_envelopes+1, y   ;get the high byte of the address
    sta sound_ptr+1

.read_ve:
    ldy stream_ve_index, x      ;our current position within the volume envelope.
    lda [sound_ptr], y          ;grab the value.
    cmp #$FF
    bne .set_vol                ;if not FF, set the volume
    dec stream_ve_index, x      ;else if FF, go back one and read again
    jmp .read_ve                ;  FF essentially tells us to repeat the last
                    ; volume value for the remainder of the note
.set_vol:
    sta sound_temp2             ;save our new volume value (about to destroy A)

    cpx #TRIANGLE
    bne .squares                ;if not triangle channel, go ahead
    lda sound_temp2
    bne .squares                ;else if volume not zero, go ahead (treat same as squares)
    lda #$80
    bmi .store_vol              ;else silence the channel with #$80
.squares:
    lda stream_vol_duty, x      ;get current vol/duty settings
    and #$F0                    ;zero out the old volume
    ora sound_temp2             ;OR our new volume in.

.store_vol:
    ldy sound_temp1             ;get our index into soft_apu_ports
    sta soft_apu_ports, y       ;store the volume in our temp port
    inc stream_ve_index, x      ;set our volume envelop index to the next position
```

```
.rest_check:
   ;check the rest flag. if set, overwrite volume with silence value
   lda stream_status, x
   and #%00000010
   beq .done                   ;if clear, no rest, so quit
   lda stream_channel, x
   cmp #TRIANGLE               ;if triangle, silence with #$80
   beq .tri                    ;else, silence with #$30
   lda #$30
   bne .store                  ;this always branches.  bne is cheaper than a jmp
.tri:
   lda #$80
.store:
   sta soft_apu_ports, y
.done:
   rts
```

New notes
The last thing we need to consider is new notes. When an old note finishes and we start playing a new note, we will want to reset the volume envelope back to the beginning. This is as easy as setting stream_ve_index to 0 when we read a new note:

```
se_fetch_byte:
   ;...snip... (setup pointers, read byte, test range, etc)
.note:
   ;do Note stuff
   sty sound_temp1     ;save our index into the data stream
   asl a
   tay
   lda note_table, y
   sta stream_note_LO, x
   lda note_table+1, y
   sta stream_note_HI, x
   ldy sound_temp1     ;restore data stream index

   lda #$00
   sta stream_ve_index, x  ;reset the volume envelope.

   ;check if it's a rest and modify the status flag appropriately
   jsr se_check_rest

   ;...snip... (update pointer)
```

And now we have volume envelopes.

Putting It All Together
Download and unzip the envelopes.zip sample files. Make sure the following files are in the same folder as NESASM3:

envelopes.asm
sound_engine.asm
envelopes.chr
note_table.i
note_length_table.i
vol_envelopes.i
song0.i
song1.i
song2.i
song3.i
song4.i
song5.i
envelopes.bat

Double click envelopes.bat. That will run NESASM3 and should produce the envelopes.nes file. Run that NES file in FCEUXD SP.

Use the controller to select songs and play them. Controls are as follows:

Up: Play
Down: Stop
Right : Next Song/SFX
Left : Previous Song/SFX

Song0 is a silence song. Not selectable.
Song1 is a boss song from The Guardian Legend, almost the same as the original.
Song2 is the same short sound effect from last week.
Song3 is a song from Dragon Warrior, very close to the original.
Song4 is the same song4 as last week, but volume envelopes allow us to save some bytes by reducing rests.
Song5 is a short sound effect, same as last week.

Try creating your own songs and sound effects and add them into the mix. To add a new song you will need to take the following steps:

1) create a song header and song data (use the included songs as reference). Don't forget to select a volume envelope for each stream in your header. Data streams are terminated with $FF.
2) add your header to the song_headers pointer table at the bottom of sound_engine.asm
3) update the constant NUM_SONGS to reflect the new song number total (also at the bottom of sound_engine.asm)

Try making your own volume envelopes too. To do so you will need to modify vol_envelopes.i. Remember that volume envelopes are terminated with $FF.

Next Week: Opcodes, Looping

Last Week: Volume Envelopes

This Week: Opcodes and Looping

Opcodes

So far our sound engine handles two type of data that it reads from music data streams: notes and note lengths. This is enough to write complex music but of course we are going to want more features. We will want control over the sound of our notes. What if we want to change duty cycles midstream? Or volume envelopes? Or keys? What if we want to loop one part of the song four times? Or loop the entire song continuously? What if we want to play a sound effect as part of a song?

All of these types of features, features where you are issuing commands to the engine, are going to be done through opcodes (also called control codes or command codes). An **opcode** is a value in the data stream that tells the engine to run a specific, specialized subroutine or piece of code. Most opcodes will have **arguments** sent along with them. For example, an opcode that changes a stream's volume envelope will come with an argument that specifies which volume envelope to change to.

We've actually been using an opcode for weeks, I just haven't mentioned it. It's the opcode that ends a sound, and we've been encoding it in our data streams as $FF. Here is the code we've been using:

```
se_fetch_byte:

   ;---snip--- (fetch a byte and range test)

.opcode:                        ;else it's an opcode
   ;do Opcode stuff
   cmp #$FF
   bne .end
   lda stream_status, x     ;if $FF, end of stream, so disable it and silence
   and #%11111110
   sta stream_status, x     ;clear enable flag in status byte

   lda stream_channel, x
   cmp #TRIANGLE
   beq .silence_tri         ;triangle is silenced differently from squares and noise
   lda #$30                 ;squares and noise silenced with #$30
   bne .silence
```

```
.silence_tri:
   lda #$80                 ;triangle silenced with #$80
.silence:
   sta stream_vol_duty, x  ;store silence value in the stream's volume variable.
   jmp .update_pointer     ;done

   ;---snip--- (do note lengths and notes, update the stream's pointer)
   rts
```

Here we check if the byte read has a value of $FF. If so we turn the stream off and silence it. That's an opcode.

It would be pretty messy if every opcode we had was just written straight out like this. Normally we would pull this code into its own subroutine, like this:

```
se_fetch_byte:

   ;---snip--- (fetch a byte and range test)

.opcode:                     ;else it's an opcode
   ;do Opcode stuff
   cmp #$FF                 ;end sound opcode
   bne .end
 jsr se_op_endsound       ;call the endsound subroutine
   iny
   jmp .fetch               ;grab the next byte in the stream.

   ;---snip--- (do note lengths and notes, update the stream's pointer)
   rts

se_op_endsound:
   lda stream_status, x    ;end of stream, so disable it and silence
   and #%11111110
   sta stream_status, x    ;clear enable flag in status byte

   lda stream_channel, x
   cmp #TRIANGLE
   beq .silence_tri        ;triangle is silenced differently from squares and noise
   lda #$30                 ;squares and noise silenced with #$30
   bne .silence
.silence_tri:
   lda #$80                 ;triangle silenced with #$80
.silence:
   sta stream_vol_duty, x  ;store silence value in the stream's volume variable.
   rts
```

The .opcode branch is much shorter now. If we wanted to add more opcodes, we could just add some more compares:

```
.opcode:
   ;do Opcode stuff
   cmp #$FF             ;is it the end sound opcode?
   bne .not_FF
   jsr se_op_endsound  ;if so, call the end sound subroutine
   jmp .end             ;and finish
.not_FF:
   cmp #$FE             ;else is it the loop opcode?
   bne .not_FE
   jsr se_op_loop       ;if so, call the loop subroutine
   jmp .opcode_done
.not_FE:
   cmp #$FD             ;else is it the change volume envelope opcode?
   bne .not_FD
   jsr se_op_change_ve ;if so, call the change volume envelope subroutine
   jmp .opcode_done
.not_FD:
```

```
.opcode_done:
   iny                  ;update index to next byte in the data stream
   jmp .fetch           ;go fetch another byte
```

This will work, but it's ugly. The more opcodes we add to our engine, the more checks we need to make. What if we have 20 opcodes? Do we really want to do that many compares? It's a waste of ROM space and cycles.

Tables

Anytime you find yourself in a situation where you are doing a lot of CMPs on one value, the answer is to use a **lookup tabl**e. It will simplify everything! We've done it already with notes, note lengths, song numbers and volume envelopes. Could you imagine trying to get a note's period without using the lookup table? It would look like this:

```
Is the note an A1?  If so, use this period, else
Is the note an A#1?  If so, use this period, else
Is the note a B1?  If so, use this period, else
Is the note a C2?  If so, use this period, else
... (about 100 more checks)
Is the note an F#9? If so, use this period, else
Is the note a rest?  If so, use this period
```

That's just crazy. It would be hundreds of lines of unreadable code and you'd run into branch-range errors too. When we use a lookup table, the code is simplified to this:

```
.note:
   ;do Note stuff
   sty sound_temp1      ;save our index
   asl a
   tay
   lda note_table, y
   sta stream_note_LO, x
   lda note_table+1, y
   sta stream_note_HI, x
   ldy sound_temp1      ;restore data stream index
```

Much cleaner. Again, I can't stress it enough: **if you find yourself doing lots of CMPs on a single value, use a table instead!**

With notes and note lengths we used a straight *lookup table* of values. With song numbers and volume envelopes we used a special type of lookup table called a *pointer table*, which stored data addresses. For opcodes we have two choices. We can use something called a **jump table** or we can use an **RTS table**. They are almost the same and the difference in performance between the two methods is negligible so for most programmers it's a matter of personal preference.

I prefer RTS tables myself, but we're going to use jump tables because they are easier to explain and understand.

Jump Tables

Ok, here's our problem: Our sound engine has opcodes. A lot of them, let's say 10 or more. Each opcode has its own subroutine. When our sound engine reads an opcode byte from the data stream, we want to avoid a long list of CMP and BNE instructions to select the right subroutine. How do we do that? We use a jump table.

A jump table is similar to a pointer table: it is a table of addresses. But whereas a pointer table holds addresses that point to the start of data, a **jump table** holds addresses that point to the start of *code* (ie, the start of subroutines). For example, suppose we have some subroutines:

```
sub_a:
   lda #$00
   ldx #$FF
   rts

sub_b:
   clc
   adc #$03
   rts

sub_c:
   sec
   sbc #$03
   rts
```

Here is how a jump table would look using these subroutines:

```
sub_jump_table:
  .word sub_a, sub_b, sub_c
```

Hey, that's pretty easy. We just use the subroutine label and the assembler will translate that into the address where the subroutine starts. Let's make a jump table for our sound opcode subroutines:

```
se_op_endsound:
  ;do stuff
  rts

se_op_infinite_loop:
  ;do stuff
  rts

se_op_change_ve:
  ;do stuff
  rts

;etc..  more subroutines

;this is our jump table
sound_opcodes:
  .word se_op_endsound
  .word se_op_infinite_loop
  .word se_op_change_ve
  ;etc, one entry per subroutine
```

Cool. We have a jump table now. So how do we use it?

Indirect Jumping
The 6502 let's us do some cool things. One of those things is called an indirect jump. An **indirect jump** let's you stick a destination address into a zero-page pointer variable and jump there. It works like this:

```
  .rsset $0000
;first declare a pointer variable somewhere in the zero-page
jmp_ptr .rs 2  ;2 bytes because an address is always a word

  lda #$00
  sta jmp_ptr
  lda #$80
  sta jmp_ptr+1
  jmp [jmp_ptr] ;will jump to $8000
```

Here we stick an address ($8000, lo byte first) into our **jmp_ptr** variable. Then we do an indirect jump by using the JMP instruction followed by a pointer variable in brackets:

```
  jmp [jmp_ptr] ;indirect jump
```

This instruction translates into English as "Jump to the address that is stored in jmp_ptr and jmp_ptr+1". It's extrememly useful. We can stick any address we want in there:

```
  lda #$00
  sta jmp_ptr
  lda #$C0
  sta jmp_ptr+1
  jmp [jmp_ptr] ;will jump to $C000
```

We could read an address from ROM and use that if we wanted to, for example our reset vector:

```
  lda $FFFC
  sta jmp_ptr
  lda $FFFD
  sta jmp_ptr+1
  jmp [jmp_ptr] ;will jump to our reset routine
```

And we can use it in combination with our jump table:

```
  lda sound_opcodes, y    ;read low byte of address from jump table
  sta jmp_ptr
  lda sound_opcodes+1, y  ;read high byte
  sta jmp_ptr+1
  jmp [jmp_ptr]   ;will jump to whatever address we pulled from the table.
```

Pretty powerful. We can dynamically jump to any section of code we want!

Implementation
So we know how to build a jump table and we know how to do an indirect jump. Let's tie it all together and stick it into our sound engine. Let's start with **se_fetch_byte**. se_fetch_byte reads a byte from the data stream and range-checks it to see if it is a note, note length or opcode. Recall that notes have a byte range of $00-$7F. Note lengths have a range of $80-$9F. The opcode byte range is $A0-$FF:

```
se_fetch_byte:
  lda stream_ptr_LO, x
  sta sound_ptr
  lda stream_ptr_HI, x
  sta sound_ptr+1

  ldy #$00
  lda [sound_ptr], y
  bpl .note                ;if < #$80, it's a Note
  cmp #$A0
  bcc .note_length         ;else if < #$A0, it's a Note Length
.opcode:                    ;else ($A0-$FF) it's an opcode
  ;do Opcode stuff
.note_length:
  ;do note length stuff
.note:
  ;do note stuff
```

So we need to assign our opcodes to values between $A0 and $FF. Just as with notes and note lengths, the opcode byte we read from the data stream will be used as a table index (after subtracting $A0), so we will assign our opcodes in the same order as our table:

```
sound_opcodes:
  .word se_op_endsound           ;this should be $A0
  .word se_op_infinite_loop      ;this should be $A1
  .word se_op_change_ve          ;this should be $A2
  ;etc, 1 entry per subroutine

;these are aliases to use in the sound data.
endsound = $A0
loop = $A1              ;be careful of conflicts here.  this might be too generic.  maybe song_loop
is better
volume_envelope = $A2
```

Now let's alter se_fetch_byte to take care of our opcodes:

```
se_fetch_byte:
  lda stream_ptr_LO, x
  sta sound_ptr
  lda stream_ptr_HI, x
  sta sound_ptr+1

  ldy #$00
```

```
.fetch:
  lda [sound_ptr], y
  bpl .note                ;if < #$80, it's a Note
  cmp #$A0
  bcc .note_length         ;else if < #$A0, it's a Note Length
.opcode:                    ;else ($A0-$FF) it's an opcode
  ;do Opcode stuff
  jsr se_opcode_launcher   ;launch our opcode!!!
  iny                      ;next position in the data stream
  lda stream_status, x
  and #%00000001
  bne .fetch               ;after our opcode is done, grab another byte unless the stream is
disabled
  rts                      ; in which case we quit  (explained below)
.note_length:
  ;do note length stuff
.note:
  ;do note stuff
```

I added a call to a subroutine called se_opcode_launcher and a little branch. Not a big change is it? But there's an important detail here. se_opcode_launcher will be a short, simple subroutine that will read from the jump table and perform an indirect jump. It looks like this:

```
se_opcode_launcher:
  sty sound_temp1         ;save y register, because we are about to destroy it
  sec
  sbc #$A0                ;turn our opcode byte into a table index by subtracting $A0
                ;  $A0->$00, $A1->$01, $A2->$02, etc.  Tables index from $00.
  asl a                   ;multiply by 2 because we index into a table of addresses (words)
  tay
  lda sound_opcodes, y    ;get low byte of subroutine address
  sta jmp_ptr
  lda sound_opcodes+1, y  ;get high byte
  sta jmp_ptr+1
  ldy sound_temp1         ;restore our y register
  iny                     ;set to next position in data stream (assume an argument)
  jmp [jmp_ptr]           ;indirect jump to our opcode subroutine
```

Short and simple. So why did I wrap this code in its own subroutine? Why not just stick this code as-is in the .opcode branch of se_fetch_byte? Because we need a place to return to.

The JSR and RTS instructions work as a pair. They go hand in hand. They need each other. Without going into too much detail, this is what goes on behind the scenes:

JSR sticks a return address on the stack and jumps to a subroutine. One way to look at it is to think of JSR as a JMP that remembers where it started from.
RTS pops the return address off the stack and jumps there.

So JSR leaves a treasure map for RTS to pick up and follow later. The **key point** here is that **RTS expects a return address to be waiting for it on the stack**.

Now our opcode subroutines all end in an RTS instruction. Do you see the potential problem here?

We call our opcode subroutines using an indirect jump. This requires us to use a JMP instruction, not a JSR instruction. A JMP instruction doesn't remember where it started from. No return address is pushed onto the stack with a JMP instruction. So when we jump to our opcode subroutine and hit the RTS instruction at the end, there is no return address waiting for us! The RTS will pull whatever random values happen to be on the stack at the time and jump there. We'll end up somewhere random and our program will surely crash!

To fix this, we wrap our indirect jump in a subroutine, se_opcode_launcher. We call it with a JSR instruction, completing the JSR/RTS pair:

```
  jsr se_opcode_launcher  ;this jsr will let us remember where we came from
```

This JSR instruction will stick a return address on the stack for us. Then inside se_opcode_launcher we perform our indirect jump to our desired opcode subroutine. Now when we hit that RTS instruction at the end of the opcode subroutine we have a return address waiting for us on the stack. Our program returns back to where we started. We are safe.

Opcode Subroutines

With our opcode launcher written, we are all set up to make opcodes. We already have one written: the **endsound** opcode. This is the opcode we will use to terminate sound effects. Sound effects don't loop continuously like songs do, so they need to be stopped. Let's take a look again:

```
se_op_endsound:
  lda stream_status, x    ;end of stream, so disable it and silence
  and #%11111110
  sta stream_status, x    ;clear enable flag in status byte

  lda stream_channel, x
  cmp #TRIANGLE
  beq .silence_tri        ;triangle is silenced differently from squares and noise
  lda #$30                ;squares and noise silenced with #$30
  bne .silence            ; (this will always branch.  bne is cheaper than a jmp)
.silence_tri:
  lda #$80                ;triangle silenced with #$80
.silence:
  sta stream_vol_duty, x  ;store silence value in the stream's volume variable.

  rts
```

This opcode is special. It's the reason for the check after the call to se_opcode_launcher:

```
se_fetch_byte:
  ;---snip---
.opcode:                       ;else ($A0-$FF) it's an opcode
  ;do Opcode stuff
  jsr se_opcode_launcher
  iny                         ;next position in the data stream
  lda stream_status, x
  and #%00000001
  bne .fetch                  ;after our opcode is done, grab another byte unless the stream is
disabled
  rts                         ; in which case we quit  (explained below)

  ;---snip---
```

Normally, we want se_fetch_byte to keep fetching bytes until it hits a note. Recall that with note lengths we jumped back to .fetch after setting the new note length. This is because after setting the length of the note, we needed to know WHAT note to play. So we fetch another byte. The same thing is true of opcodes. If we change the volume envelope with an opcode, great! But we still need to know what note to play next. If we use an opcode to switch our square's duty cycle, great! But we still need to know what note to play next. If we use an opcode to loop back to the beginning of the song, that's great! But we still need to read that first note of the song. This is why we jump back to fetch a byte after we run an opcode.

The ONE exception to this rule is when we end a sound effect. We are terminating the sound effect completely, so there is no next note. We don't want to fetch something that isn't there, so we need to skip the jump. That's why we check the status byte after we run the opcode. If the stream is disabled by the endsound opcode, we are finished. Otherwise, fetch another byte.

Looping

The next opcode in our list is the **loop** opcode. This is the opcode that we will stick at the end of every song to tell the sound engine to play the song again, and again and again. It is actually quite easy to implement. It takes a 2-**byte argument**, which is **the address to loop back to**. The subroutine looks like this:

```
se_op_infinite_loop:
  lda [sound_ptr], y      ;read LO byte of the address argument from the data stream
  sta stream_ptr_LO, x    ;save as our new data stream position
  iny
  lda [sound_ptr], y      ;read HI byte of the address argument from the data stream
```

```
    sta stream_ptr_HI, x    ;save as our new data stream position data stream position

    sta sound_ptr+1         ;update the pointer to reflect the new position.
    lda stream_ptr_LO, x
    sta sound_ptr
    ldy #$FF                ;after opcodes return, we do an iny.  Since we reset
                 ;the stream buffer position, we will want y to start out at 0 again.
    rts
```

The first thing to notice about this subroutine is that it reads two bytes from the data stream. This is the address argument that gets passed along with the opcode. To make it clear, let's look at some example sound data:

```
song1_square1:
    .byte eighth ;set note length to eighth notes
    .byte C5, E5, G5, C6, E6, G6, C5, Eb5, G5, C6, Eb6, half, G6 ;play some notes
    .byte loop          ;this alias evaluates to $A1, the loop opcode
    .word song1_square1 ;this evaluates to the address of the song1_square1 label
               ;ie, the address we want to loop to.
```

After the "loop" opcode comes a word which is the address to loop back to. In this example I chose to loop back to the beginning of the stream data.

So what does our loop opcode do? It reads the first byte of this address argument (the low byte) and stores it in stream_ptr_LO. Then it reads the second byte of the address argument (the high byte) and stores it in stream_ptr_HI. These are the variables that keep track of our data stream position! The loop opcode just changes these values to some address that we specify. Not too complicated at all. The last step is to update the actual pointer (**sound_ptr**) so that the next byte we read from the data stream will be the first note we looped back to.

In the example sound data above I looped back to the beginning of the stream data, but there's nothing stopping me from looping somewhere else:

```
song1_square1:
;intro, don't loop this part
    .byte quarter
    .byte C4, C4, C4, C4
.loop_point:    ;this is where we will loop back to.
    .byte eighth ;set note length to eighth notes
    .byte C5, E5, G5, C6, E6, G6, C5, Eb5, G5, C6, Eb6, half, G6
    .byte loop          ;this alias evaluates to $A1, the loop opcode
    .word .loop_point ;this evaluates to the address of the .loop_point label
               ;ie, the address we want to loop to.
```

Technically we can also "loop" to a forward position, in which case it's actually more like a jump than a loop. That's all a loop is really: a jump... backwards.

Changing Volume Envelopes
Let's write the opcode subroutine to change volume envelopes. This one is even easier. It takes **one argument**, which will be **which volume envelope to switch to**:

```
se_op_change_ve:
    lda [sound_ptr], y      ;read the argument
    sta stream_ve, x        ;store it in our volume envelope variable
    lda #$00
    sta stream_ve_index, x  ;reset volume envelope index to the beginning
    rts
```

That's it!

Changing Duty Cycles
Now let's add an opcode that will change the duty cycle for a square stream. This one also takes **one argument: which duty cycle to switch to**.

```
se_op_duty:
    lda [sound_ptr], y      ;read the argument (which duty cycle to change to)
    sta stream_vol_duty, x  ;store it.
    rts
```

Done! Now we have the subroutine, but we still need to add it to our jump table:

```
sound_opcodes:
    .word se_op_endsound            ;this should be $A0
    .word se_op_loop                ;this should be $A1
    .word se_op_change_ve           ;this should be $A2
    .word se_op_duty                ;this should be $A3
    ;etc, 1 entry per subroutine

;these are aliases to use in the sound data.
endsound = $A0
loop = $A1
volume_envelope = $A2
duty = $A3
```

And it's ready to use:

```
song0_square1:
;intro, don't loop this part
    .byte quarter
    .byte C4, C4, C4, C4
.loop_point:                            ;this is where we will loop back to.
    .byte duty, $B0                     ;change the duty cycle
    .byte volume_envelope, ve_blip_echo ;change the volume envelope
    .byte eighth                        ;set note length to eighth notes
    .byte C5, E5, G5, C6, E6, G6        ;play some notes

    .byte duty, $30                     ;change the duty cycle
    .byte volume_envelope, ve_short_staccato    ;change volume envelope
    .byte C5, Eb5, G5, C6, Eb6, half, G6    ;play some eighth notes and a half note

    .byte loop                          ;loop to .loop_point
    .word .loop_point
```

Readability
sound_engine.asm is getting pretty bulky with all these subroutines. It will only get bigger as we add more opcodes. It's nice to have all of our opcodes together in one place, but it's annoying to have to scroll around to find them. So let's pull all of our opcodes into their own file: **sound_opcodes.asm**. Then, at the bottom of sound_engine.asm, we can .include it:

```
    .include "sound_opcodes.asm" ;our opcode subroutines, jump table and aliases
    .include "note_table.i" ;period lookup table for notes
    .include "note_length_table.i"
    .include "vol_envelopes.i"
    .include "song0.i"  ;holds the data for song 0 (header and data streams)
    .include "song1.i"  ;holds the data for song 1
    .include "song2.i"
    .include "song3.i"
    .include "song4.i"
    .include "song5.i"
    .include "song6.i"  ;oooh.. new song!
```

I gave it the extension .asm because it contains code as well as data, and I like to be able to tell at a glance what files have what in them. Now whenever we want to add new opcodes, or tweak old ones, we have them nice and compact in their own file.

Updating Sound Data
Whenever we add new things to our sound engine, we have to think about how it will affect our old sound data. This week we added opcodes, which will change our songs and sound effects terminate. Before we were terminating them with $FF. This won't work anymore because $FF doesn't do anything. For songs, we should terminate with "loop" followed by an address to loop to. With sound effects we should terminate with the opcode "endsound". See the included songs and sound effects for examples.

RTS Tables

We talked about jump tables and indirect jumping this week. Another method for doing the same thing involves something called an **RTS table** and the **RTS Trick**. I won't cover it in these tutorials, but if you are curious to know how this works you can read this nesdev wiki article I wrote about the RTS Trick.

Putting It All Together
Download and unzip the opcodes.zip sample files. Make sure the following files are in the same folder as NESASM3:

opcodes.asm
sound_engine.asm
sound_opcodes.asm
opcodes.chr
note_table.i
note_length_table.i
vol_envelopes.i
song0.i
song1.i
song2.i
song3.i
song4.i
song5.i
song6.i
opcodes.bat

Double click opcodes.bat. That will run NESASM3 and should produce the opcodes.nes file. Run that NES file in FCEUXD SP.

Use the controller to select songs and play them. Controls are as follows:

Up: Play
Down: Stop
Right: Next Song/SFX
Left: Previous Song/SFX

Song0 is a silence song. Not selectable.
Song1 is a boss song from The Guardian Legend. Now it loops!
Song2 is the same short sound effect from last week. Terminated with endsound.
Song3 is a song from Dragon Warrior. Now it loops!
Song4 is the same song4 as last week, but now it loops!
Song5 is a short sound effect, terminated with the endsound opcode.
Song6 should be familiar to readers of this forum. Do you recognize it? It utilizes opcodes for changing duty cycles and volume envelopes. Plus it loops!

Try adding your own songs and sound effects in. Try to add your own opcodes too. Here's some ideas for opcodes:

1. Trigger a sound effect mid-song
2. Implement duty cycle envelopes (similar to volume envelopes). Then make an opcode that allows you to change it.
3. Finite loops

Next Week: more opcode fun. Finite Loops, Changing Keys and Autom... .

Last Week: Opcodes and Looping

This Week: More opcodes: Finite Loops, Key Changes, Chord Progressions

Opcodes
Last week we learned how to use opcodes. Opcodes allow a song's streams to call a subroutine mid-play. This is a very powerful tool. We learned some of the most common opcodes: infinite loop (really a jump), change volume envelopes and change duty cycles. Today we are going to expand on opcodes and learn some cool opcode tricks that can save us a lot (!) of bytes and time.

Finite Looping
Last week we added the infinite loop opcode, which was really just an unconditional jump back to an earlier part of the song. Today we're going to add a finite loop opcode. A finite loop opcode tells the sound engine to repeat a particular section of a song X times, where X is some number defined by you. In the Battle Kid theme song I added last week there is a passage that looks like this:

```
  .byte sixteenth
  .byte A3, C4, E4, A4, A3, C4, E4, A4, A3, C4, E4, A4, A3, C4, E4, A4
  .byte A3, C4, E4, A4, A3, C4, E4, A4, A3, C4, E4, A4, A3, C4, E4, A4
  .byte A3, C4, E4, A4, A3, C4, E4, A4, A3, C4, E4, A4, A3, C4, E4, A4
  .byte A3, C4, E4, A4, A3, E4, E3, E2
```

This is really just the same 4 notes repeated over and over again. Wouldn't it be cooler if we could do something like this instead:

```
  .byte sixteenth
  .byte A3, C4, E4, A4
  .byte loop_13_times_please
  .byte A3, E4, E3, E2
```

That saves a lot of bytes. We go from 56 bytes all the way down to around 10. The Battle Kid song actually plays this same phrase on both square channels, so really we go from 100+ bytes down to 20 or so. That's a big deal! If we consider how common repetitions of 4 or 8 occur in music, we can easily see that having a finite loop opcode could potential save us hundreds if not thousands of bytes in our sound data.

Finite Looping?
So what is a finite loop really? We saw that with an infinite loop it was really more like an unconditional jump. When the sound engine hits the infinite loop opcode, it jumps back, always, no matter what, no questions asked.   A finite loop on the other hand is a conditional jump. It checks a counter. If the counter isn't 0 it jumps. If it is 0, it doesn't jump.

Loop Counter
First things first we need a loop counter. Each stream will have the ability to loop, so each stream will need its own loop counter:

```
stream_loop1 .rs 6  ;loop counter variable (one for each stream)
```

We will want to initialize this to 0 in our sound_load code:

```
  lda #$00
  sta stream_loop1, x
```

Next we will need a way to set this counter to some value. Some games bundle this up together in the finite loop opcode, but I prefer to make it its own opcode:

```
;-------------------------------------------------------------------
;this is our JUMP TABLE!
sound_opcodes:
  .word se_op_endsound            ;$A0
  .word se_op_infinite_loop       ;$A1
  .word se_op_change_ve           ;$A2
  .word se_op_duty                ;$A3
  .word se_op_set_loop1_counter   ;$A4
  ;etc, one entry per subroutine

;these are aliases to use in the sound data.
endsound = $A0
loop = $A1
volume_envelope = $A2
duty = $A3
set_loop1_counter = $A4

se_op_set_loop1_counter:
  lda [sound_ptr], y      ;read the argument (# times to loop)
  sta stream_loop1, x     ;store it in the loop counter variable
  rts
```

Now we have an easy way to set the loop counter any time we want, like this:

```
  ;somewhere in sound data:
  .byte set_loop1_counter, $04    ;repeat 4 times
```

Looping With The Counter
Our finite loop opcode will work like the infinite loop opcode, with two changes:

1) it will decrement the loop counter
2) it will check the result and only jump on a non-zero result

Let's write it:

```
;--------------------------------------------------------------------
;this is our JUMP TABLE!
sound_opcodes:
   .word se_op_endsound            ;$A0
   .word se_op_infinite_loop       ;$A1
   .word se_op_change_ve           ;$A2
   .word se_op_duty                ;$A3
   .word se_op_set_loop1_counter   ;$A4
   .word se_op_loop1               ;$A5
   ;etc, one entry per subroutine

;these are aliases to use in the sound data.
endsound = $A0
loop = $A1
volume_envelope = $A2
duty = $A3
set_loop1_counter = $A4
loop1 = $A5

se_op_loop1:
   dec stream_loop1, x      ;decrement the counter
   lda stream_loop1, x      ;and check it
   beq .last_iteration      ;if zero, we are done looping
.loop_back:
   lda [sound_ptr], y       ;read ptr LO from the data stream
   sta stream_ptr_LO, x     ;update our data stream position
   iny
   lda [sound_ptr], y       ;read ptr HI from the data stream
   sta stream_ptr_HI, x     ;update our data stream position

   sta sound_ptr+1          ;update the pointer to reflect the new position.
   lda stream_ptr_LO, x
   sta sound_ptr
   ldy #$FF                 ;after opcodes return, we do an iny.  Since we reset
                 ;the stream buffer position, we will want y to start out at 0 again.
   rts
.last_iteration:
   iny                      ;skip the first byte of the address argument
                 ; the second byte will be skipped automatically upon return
                 ; (see se_fetch_byte.  There is an "iny" after "jsr se_opcode_launcher")
   rts
```

Now we can loop. To use the Battle Kid example above, we go from this (56 bytes):

```
   .byte A3, C4, E4, A4, A3, C4, E4, A4, A3, C4, E4, A4, A3, C4, E4, A4
   .byte A3, C4, E4, A4, A3, C4, E4, A4, A3, C4, E4, A4, A3, C4, E4, A4
   .byte A3, C4, E4, A4, A3, C4, E4, A4, A3, C4, E4, A4, A3, C4, E4, A4
   .byte A3, C4, E4, A4, A3, E4, E3, E2
```

to this (13 bytes):

```
   .byte set_loop1_counter, 13 ;repeat 13 times.
.intro_loop:                    ;make sure our loop point is AFTER we set the counter!
   .byte A3, C4, E4, A4        ;the phrase to repeat.
   .byte loop1                 ;finite loop opcode
   .word .intro_loop           ;address to jump back to
```

```
  .byte A3, E4, E3, E2        ;the last 4 notes
```

Pretty nice savings. Chances are we will be using this opcode set a lot.

Bonus
We can save a few more bytes here. You may have noticed that the code in the .loop_back section of our finite loop opcode is identical to the infinite loop code:

```
se_op_loop1:

  ;---snip---

.loop_back:
  lda [sound_ptr], y      ;read ptr LO from the data stream
  sta stream_ptr_LO, x    ;update our data stream position
  iny
  lda [sound_ptr], y      ;read ptr HI from the data stream
  sta stream_ptr_HI, x    ;update our data stream position

  sta sound_ptr+1         ;update the pointer to reflect the new position.
  lda stream_ptr_LO, x
  sta sound_ptr
  ldy #$FF                ;after opcodes return, we do an iny.  Since we reset
              ;the stream buffer position, we will want y to start out at 0 again.
  rts

  ;---snip---
```

Compare with:

```
se_op_infinite_loop:
  lda [sound_ptr], y      ;read ptr LO from the data stream
  sta stream_ptr_LO, x    ;update our data stream position
  iny
  lda [sound_ptr], y      ;read ptr HI from the data stream
  sta stream_ptr_HI, x    ;update our data stream position

  sta sound_ptr+1         ;update the pointer to reflect the new position.
  lda stream_ptr_LO, x
  sta sound_ptr
  ldy #$FF                ;after opcodes return, we do an iny.  Since we reset
              ;the stream buffer position, we will want y to start out at 0 again.
  rts
```

Why have identical code in two places? Let's cut out the whole .loop_back section and replace it with a "jmp se_op_infinite_loop":

```
se_op_loop1:
  dec stream_loop1, x     ;decrement the counter
  lda stream_loop1, x     ;check the counter
  beq .last_iteration     ;if zero, we are done looping
  jmp se_op_infinite_loop ;if not zero, loop back
.last_iteration:
  iny                     ;skip the first byte of the address argument
              ; the second byte will be skipped automatically upon return
              ; (see se_fetch_byte after "jsr se_opcode_launcher")
  rts
```

Multiple Finite Loops
You may have been wondering why I named the finite loop opcode "loop1". Why stick a 1 on the end there? This is because sometimes one finite loop opcode isn't enough. Consider the following song structure. Assume each letter represents a long series of notes:

```
  A A A B C
  A A A B C
  A A A B C
  A A A B C
```

With one finite loop opcode you could reduce it to this:

```
  (A A A B C)x4
```

But if you had two finite loop opcodes available, you could nest them to reduce it even further:

```
  (Ax3 B C)x4
```

If the music you write has a lot of patterns like this, it may be worth your while to have two or more finite loop opcodes available to you so that you can nest them. To add another finite loop opcode you need to:

1) declare another loop counter variable block in RAM (stream_loop2 .rs 6)
2) initialize the new loop counter to 0 in the sound_load routine.
3) add a new opcode for setting the new loop counter (se_op_set_loop2_counter)
4) add a new opcode to check the new counter and loop (se_op_loop2)
5) make sure to add the new opcodes to the jump table and give them an alias (set_loop2_counter, loop2).

Each finite loop opcode you add requires 6 bytes of RAM (a limited resource!), so please consider carefully if it is worth the tradeoff. It all depends on your music data.

Changing Keys

Another useful feature to have is the ability to change keys. Imagine you write a song and you have it all done. Then at the last minute you decide you want it to be in another key, say a step (2 notes) lower. Rather than rewrite the whole song by hand (it takes forever), wouldn't it be nice if there was an opcode that you could set to automatically subtract two from every note? What if you have a song pattern that gets played in more than one key (a rhythm track for a Blues song, for example)? We could save lots of bytes if we can figure out a way to write the pattern once, and then loop it while changing keys each iteration. Let's do it.

Note Offset
We will implement keys by having a note offset variable:

```
stream_note_offset .rs 6    ;note offset
```

The note offset is a value that gets added to the note value before pulling the period out of the note_table. We will initialize stream_note_offset to 0 so that the default behavior is to add 0 to the note (resulting in no change). However, if we set stream_note_offset to some value via an opcode, it will change the notes. Here is an updated se_fetch_byte that demonstrates how this works:

```
se_fetch_byte:
  ;...snip...
.note:
  ;do Note stuff
  sty sound_temp1      ;save our index into the data stream
  clc
  adc stream_note_offset, x    ;add note offset
  asl a
  tay
  lda note_table, y
  sta stream_note_LO, x
  lda note_table+1, y
  sta stream_note_HI, x
  ldy sound_temp1      ;restore data stream index

  ;...snip...
```

Imagine what would happen if we have stream_note_offset set to 2. Say we read a C4 note from the data stream:

1. A C4 note is equivalent to hex value #$1b (see aliases in note_table.i)

2. we add stream_note_offset to this value. #$1b + #$02 = #$1d.
3. hex value #$1d is equivalent to a D4 note (see note_table.i)
4. wow, we raised the note up a step!

Using the same value for stream_note_offset, if we had a string of notes like this:

```
C4, E4, G4, B4, C5, E5, G5, E5, B5, C6 ;Cmaj7
```

it would get translated to:

```
D4, Fs4, A4, C#5, D5, Fs5, A5, C#6, D6 ;Dmaj7
```

Using stream_note_offset we can easily transpose entire sections of music into other keys. As mentioned above, we will initialize a stream's stream_note_offset to zero:

```
sound_load:
  ;---snip---
  lda #$00
  sta stream_note_offset, x
  ;---snip---
```

Set Note Offset
Now let's make an opcode that will set stream_note_offset to a specific value:

```
;-----------------------------------------------------------------------
;this is our JUMP TABLE!
sound_opcodes:
  .word se_op_endsound            ;$A0
  .word se_op_infinite_loop       ;$A1
  .word se_op_change_ve           ;$A2
  .word se_op_duty                ;$A3
  .word se_op_set_loop1_counter   ;$A4
  .word se_op_loop1               ;$A5
  .word se_op_set_note_offset     ;$A6

;these are aliases to use in the sound data.
endsound = $A0
loop = $A1
volume_envelope = $A2
duty = $A3
set_loop1_counter = $A4
loop1 = $A5
set_note_offset = $A6

se_op_set_note_offset:
  lda [sound_ptr], y          ;read the argument
  sta stream_note_offset, x      ;set the note offset.
  rts
```

Now we can set the note offset anytime we want in the data stream:

```
;oops, after writing the song, I realized I wanted it to be in D instead.  No problem.
sound_data:
  .byte set_note_offset, 2
  .byte C2, C3, C4, C5, ;etc.. more notes in the key of C.
```

Adjust Note Offset
Setting the note offset to a specific value has very limited application. It's like a one-time keychange. More often we will want to set the note offset to some relative value. For example, instead of setting stream_note_offset to 2, we might want to set stream_note_offset to "the current offset + 2". If we had an opcode that let us adjust stream_note_offset by a relative value, we could use it together with loops. First let's write the opcode:

```
;-----------------------------------------------------------------------
;this is our JUMP TABLE!
sound_opcodes:
  .word se_op_endsound            ;$A0
```

```
    .word se_op_infinite_loop       ;$A1
    .word se_op_change_ve           ;$A2
    .word se_op_duty                ;$A3
    .word se_op_set_loop1_counter   ;$A4
    .word se_op_loop1               ;$A5
    .word se_op_set_note_offset     ;$A6
    .word se_op_adjust_note_offset  ;$A7

;these are aliases to use in the sound data.
endsound = $A0
loop = $A1
volume_envelope = $A2
duty = $A3
set_loop1_counter = $A4
loop1 = $A5
set_note_offset = $A6
adjust_note_offset = $A7

se_op_adjust_note_offset:
    lda [sound_ptr], y          ;read the argument (what value to add)
    clc
    adc stream_note_offset, x   ;add it to the current offset
    sta stream_note_offset, x   ;and save.
    rts
```

Let's look at this opcode in use. Say we have a long arpeggiated line like this:

```
C2, E2, G2, B2, C3, E3, G3, B3, C4, E4, G4, B4, C5, E5, G5, B5, C6, E6, G6, B6, C7 ;Cmaj7 (21 bytes)
```

This passage just repeats the same 4 notes (C E G B) over 5 octaves.

```
    .byte set_loop1_counter, 5      ;loop 5 times
.loop
    .byte C2, E2, G2, B2            ;these are the 4 notes to loop
    .byte adjust_note_offset, 12    ;each iteration add 12 to the offset (ie, go up an octave)
    .byte loop1
    .word .loop

    .byte C2                        ;will be a C7.  Cmaj7 (12 bytes)
```

The first time through the loop it will play C2, E2, G2, B2. The second time through the loop it will play C3, E3, G3, B3. The third time through will be C4, E4, G4, B4, etc. Using our opcodes, we reduce the size of our data from 21 bytes to 12 bytes. That's almost 50% savings.

Battle Kid
To take a better example, let's look at the bassline to the Battle Kid theme song. Last week, it looked like this:

```
song6_tri:
    .byte eighth
    .byte A3, A3, A4, A4, A3, A3, A4, A4
    .byte G3, G3, G4, G4, G3, G3, G4, G4            ;down a step (-2)
    .byte F3, F3, F4, F4, F3, F3, F4, F4            ;down a step (-2)
    .byte Eb3, Eb3, Eb4, Eb4, Eb3, Eb3, Eb4, Eb4    ;down a step (-2)
    .byte loop
    .word song6_tri
    ;36 bytes
```

We have a pattern here: X3, X3, X4, X4, X3, X3, X4, X4, where X = some note. It just so happens that each new X is just the previous X minus 2. Using our new opcode, we can rewrite the bassline like this:

```
song6_tri:
    .byte eighth
    .byte set_loop1_counter, 4              ;repeat 4 times
.loop:
    .byte A3, A3, A4, A4, A3, A3, A4, A4    ;series of notes to repeat
```

```
  .byte adjust_note_offset, -2           ;go down a step
  .byte loop1
  .word .loop

  .byte set_note_offset, 0               ;after 4 repeats, reset note offset to 0.
  .byte loop                             ;infinite loop
  .word song6_tri
  ;21 bytes
```

We drop from 36 bytes to 21 bytes of ROM space. About 40% savings!

Loopy Sound Effects
We can produce some cool sound effects if we combine loops and key changes at high tempos. Look at this one (tempo is $FF):

```
song7_square2:
  .byte set_loop1_counter, $08    ;repeat 8 times
.loop:
  .byte thirtysecond, D7, D6, G6  ;play two D notes at different octaves and a G.  Pretty random
  .byte adjust_note_offset, -4    ;go down 2 steps
  .byte loop1
  .word .loop
  .byte endsound
```

This sound effect plays a simple 3-note pattern in descending keys super fast. The sound data is only 12 bytes, but it produces a pretty complex sound effect. Listen to song7 in this week's sample files to hear it. By experimenting with loops like this we can come up with some sounds that would be difficult to compose by hand.

Complex Chord Progressions

We made some good savings percentage-wise on the bassline to Battle Kid. But we were lucky. The chord progression went down in consistent steps: -2, -2, -2. It was possible to loop this because we adjust the note_offset by the same value (-2) each time. But what if we had a pattern that was repeated in a more complicated way? We do. Let's look at the rhythm pattern for our Guardian Legend boss song:

```
song1_square1:
  .byte eighth
  .byte A2, A2, A2, A3, A2, A3, A2, A3
  .byte F3, F3, F3, F4, F3, F4, F3, F4            ;+8 (A2 + 8 = F3)
  .byte A2, A2, A2, A3, A2, A3, A2, A3            ;-8
  .byte F3, F3, F3, F4, F3, F4, F3, F4            ;+8
  .byte E3, E3, E3, E4, E3, E4, E3, E4            ;-1
  .byte E3, E3, E3, E4, E3, E4, E3, E4            ;+0
  .byte Ds3, Ds3, Ds3, Ds4, Ds3, Ds4, Ds3, Ds4    ;-1
  .byte D3, D3, D3, D4, D3, D4, D3, D4            ;-1
  .byte C3, C3, C3, C4, C3, C4, C3, C4            ;-2
  .byte B2, B2, B2, B3, B2, B3, B2, B3            ;-1
  .byte As2, As2, As2, As3, As2, As3, As2, As3    ;-1
  .byte A2, A2, A2, A3, A2, A3, A2, A3            ;-1
  .byte Gs2, Gs2, Gs2, Gs3, Gs2, Gs3, Gs2, Gs3    ;-1
  .byte G2, G2, G2, G3, G2, G3, G2, G3            ;-1
  .byte loop                                      ;+2 (loop back to A2)
  .word song1_square1
```

Here we have another pattern: Xi, Xi, Xi, Xi+1, Xi, Xi+1, Xi, Xi+1, where X = some note and i = some octave. Cool. A pattern means we have an opportunity to save bytes by looping. But wait. Unlike Battle Kid, this pattern jumps around in an inconsistent way. What should we do?

Super TGL Transposition Trick
I learned this trick from The Guardian Legend, so I call it the **TGL Transposition Trick**. What we do is we loop the pattern, and then use the loop counter as an index into a lookup table. The lookup table contains note offset values. Because the loop counter decrements, our lookup table will be sequentially backwards.

Wait, what? Let's looks at our example:

```
song1_square1:
  .byte eighth
  .byte set_loop1_counter, 14             ;repeat 14 times
.loop:
  .byte A2, A2, A2, A3, A2, A3, A2, A3
  ;pull a value from lookup_table and
  ; add it to stream_note_offset
  .byte loop1                             ;finite loop (14 times)
  .word .loop

  .byte loop                              ;infinite loop
  .word song1_square1

.lookup_table:
  .byte 2, -1, -1, -1, -1, -1, -2
  .byte -1, -1, 0, -1, 8, -8, 8       ;14 entries long, reverse order
```

I'm going to break it down in a second here, but first let me tell you that the part highlighted in red above will be covered by a single opcode, transpose. The transpose opcode takes a 2-byte argument, so altogether that commented section will be replaced with 3 bytes of data. So if we count up all of the bytes in our rhythm sound data we get 34 bytes. The original was 116 bytes. By using the TGL Transposition Trick, we save 82 bytes. That's 70%!

```
song1_square1:
  .byte eighth
  .byte set_loop1_counter, 14             ;repeat 14 times
.loop:
  .byte A2, A2, A2, A3, A2, A3, A2, A3
  .byte transpose                         ;the transpose opcode take a 2-byte argument
  .word .lookup_table                     ;which is the address of the lookup table

  .byte loop1                             ;finite loop (14 times)
  .word .loop

  .byte loop                              ;infinite loop
  .word song1_square1

.lookup_table:
  .byte 2, -1, -1, -1, -1, -1, -2
  .byte -1, -1, 0, -1, 8, -8, 8       ;14 entries long, reverse order

;*** altogether 34 bytes ***
```

The transpose opcode will set up a pointer variable to point to the lookup table. Then it will take the loop counter, subtract 1, and use the result as an index into the table. We subtract 1 because the tables index from zero. If we loop 14 times, our table will have 14 entries numbered 0-13. Once the transpose opcode has its index, it will pull a value from the table. This value will be added to stream_note_offset.

Before we write the opcode, let's trace through the data to see how it works. We'll start at the very first byte of song1_square1:

1) set note length to eighth notes
2) set the loop counter to 14

(.loop iteration 1)
3) play a series of notes: A2, A2, A2, A3, A2, A3, A2, A3
4) transpose opcode. Setup a pointer to lookup_table. Use our loop counter, minus one, as an index. The loop counter is 14 now, so we will pull out .lookup_table+13, which is an 8. Add 8 to the current stream_note_offset: 0 + 8 = 8.
5) decrement the loop counter (14->13) and loop back to the .loop label

(iteration 2)
6) our new string of notes with the +8: F3, F3, F3, F4, F3, F4, F3, F4.
7) transpose opcode. Loop counter is 13. Grab .lookup_table+12, which is -8. Add -8 to stream_note_offset: 8 + -8 = 0.
8) decrement loop counter (13->12) and loop back to .loop label

(iteration 3)
9) our new string of notes with the +0: A2, A2, A2, A3, A2, A3, A2, A3
10) transpose opcode. Loop counter is 12. Grab .lookup_table+11, which is 8. Add 8 to stream_note_offset: 0 + 8 = 8.
11) decrement loop counter (12->11) and loop back to .loop label

(iteration 4)
12) our new string of notes with the +8: F3, F3, F3, F4, F3, F4, F3, F4.
13) transpose opcode. Loop counter is 11. Grab .lookup_table+10, which is -1. Add -1 to stream_note_offset: 8 + -1 = 7.
14) decrement loop counter (11->10) and loop back to .loop label

(iteration 4)
15) our new string of notes with the +7: E3, E3, E3, E4, E3, E4, E3, E4.
16) transpose opcode. Loop counter is 10. Grab .lookup_table+9, which is 0. Add 0 to stream_note_offset: 7 + 0 = 7.
17) decrement loop counter (10->9) and loop back to .loop label

etc. On the last iteration our loop counter is 1. We grab .lookup_table+0 and add it to stream_note_offset. Then we decrement the loop counter (1->0). Our loop counter is now 0, so our loop breaks. Pretty cool, no? Let's write it.

```
;-------------------------------------------------------------------
;this is our JUMP TABLE!
sound_opcodes:
   .word se_op_endsound             ;$A0
   .word se_op_infinite_loop        ;$A1
   .word se_op_change_ve            ;$A2
   .word se_op_duty                 ;$A3
   .word se_op_set_loop1_counter    ;$A4
   .word se_op_loop1                ;$A5
   .word se_op_set_note_offset      ;$A6
   .word se_op_adjust_note_offset   ;$A7
   .word se_op_transpose            ;$A8

;these are aliases to use in the sound data.
endsound = $A0
loop = $A1
volume_envelope = $A2
duty = $A3
set_loop1_counter = $A4
loop1 = $A5
set_note_offset = $A6
adjust_note_offset = $A7
transpose = $A8

se_op_transpose:
   lda [sound_ptr], y           ;read low byte of the pointer to our lookup table
   sta sound_ptr2               ;store it in a new pointer variable
   iny
   lda [sound_ptr], y           ;read high byte of pointer to table
   sta sound_ptr2+1

   sty sound_temp               ;save y because we are about to destroy it
   lda stream_loop1, x          ;get loop counter, put it in Y
   tay                          ;   this will be our index into the lookup table
   dey                          ;subtract 1 because indexes start from 0.

   lda [sound_ptr2], y          ;read a value from the table.
   clc
   adc stream_note_offset, x    ;add it to the note offset
   sta stream_note_offset, x

   ldy sound_temp               ;restore Y
   rts
```

There is a new pointer variable here, sound_ptr2. Actually, what I really did was rename jmp_ptr to sound_ptr2. The new name let's me know it's for sound engine use only. Since we finish with jmp_ptr as soon as we jump, there are no pointer conflicts here.

Conclusion
This is just an example of how clever use of opcodes and looping can save you lots of bytes. Keep in mind that this transpose opcode is only useful if you write music that has repeating patterns in the rhythm section. If you don't, then save yourself some bytes and cut the opcode from your sound engine.

Putting It All Together
Download and unzip the opcodes2.zip sample files. Make sure the following files are in the same folder as NESASM3:

opcodes2.asm
sound_engine.asm
sound_opcodes.asm
opcodes2.chr
note_table.i
note_length_table.i
vol_envelopes.i
song0.i
song1.i
song2.i
song3.i
song4.i
song5.i
song6.i
song7.i
opcodes2.bat

Double click opcodes2.bat. That will run NESASM3 and should produce the opcodes2.nes file. Run that NES file in FCEUXD SP.

Use the controller to select songs and play them. Controls are as follows:

Up: Play
Down: Stop
Right : Next Song/SFX
Left : Previous Song/SFX

Song0 is a silence song. Not selectable.
Song1-Song6 are the same as last week, but they take up less ROM-space now
Song7 is a new sound effect created by looping a key change at high tempo.

As usual, try adding your own songs and sound effects in using the new opcodes. Experiment.
Last Week: Finite Looping, Key Changes, Chord Progressions

This Week: Simple Drums

Drums

This week we are going to take a look at drums. I saved them until now because they use the **Noise** channel, which operates much differently from the other 3 channels. With the Square and Triangle channels we manually set the waveform period to choose what note to play. Our note lookup table was just a table of periods to plug into the channel ports. The Noise channel on the other hand produces random noise. We don't use a note table at all.

Noise Channel

The noise channel doesn't produce notes, it produces noise. We communicate with the noise channel through 3 ports: $400**C**, $400**E**, $400**F**. Note that port $400D is *unused*. Let's take a closer look at the Noise ports.

Volume - $400C
$400C let's you control the volume of the Noise channel. It works just like the Square channels, except there is no Duty Cycle:

```
NOI_ENV ($400C)
```

76543210

```
||||||
||++++- Volume
|+----- Saw Envelope Disable (0: use internal counter for volume; 1: use Volume for volume)
+------ Length Counter Disable (0: use Length Counter; 1: disable Length Counter)
```

For our purposes, we will set Saw Envelope Disable and Length Counter Disable and forget about them. This will allow us to have full control of "note" length and volume via Volume Envelopes. We did the same thing for the Square channels.

Random Generator - $400E
$400E let's us control the settings for the random generator. It looks like this:

```
NOI_RAND ($400E)

76543210
|  ||||
|  ++++- Sound Type
+-------- Mode
```

Mode sets the mode. There are two modes, **Mode-0** and **Mode-1**. Mode-0 sounds dull and breathy, like static sssh. Mode-1 sounds more sharp, robotic and buzzy. There's really no good way to describe in words, so the best way to know the difference in sound is to listen to both modes. (see below)

Each mode has 16 possible sounds, selected by **Sound Type**. This means that the Noise channel really only gives us 32 possible sounds total! More complex sound effects on the Noise channel are created by playing combinations of these 32 noises one after another. To hear what each of the 32 sounds sound like, listen to **Song 8** in this week's sample program. It plays the 16 Mode-0 sounds followed by the 16 Mode-1 sounds.

Note: In this tutorial I will assign each of the 32 Noise channel sounds a number 00-1F. The left number (0 or 1) refers to the *Mode*. The right number (0-F) refers to the *Sound Type*. For example, sound "04" means "Mode 0, Sound Type 4". Sound "1E" means "Mode 1, Sound Type E".

Length Counter - $400F
$400F is the Noise channel's length counter. We disabled the length counter in $400C, so we can ignore this port completely!

Simple Noise Drums

The simplest way to make a drum sound on the Noise channel is to play a single Sound Type under a volume envelope that decays to 0 (silence). Many games use this kind of drum exclusively. The Guardian Legend for example only uses two drum sounds throughout the whole game: 04 and 06. Battle Kid makes heavy use this simple drum style too. Lots of games do.

So how will we represent simple drums in the sound data? Recall that our sound engine distinguishes between Notes, Note Lengths and Opcodes using ranges:

```
.fetch:
   lda [sound_ptr], y
   bpl .note                 ;if < #$80, it's a Note
   cmp #$A0
   bcc .note_length          ;else if < #$A0, it's a Note Length
.opcode:                      ;else it's an opcode
   ;do opcode stuff
   ;range A0-FF
.note_length:
   ;do note length stuff
   ;range 80-9F
.note:
   ;do note stuff
   ;range 00-7F
```

Although we won't use our note table for the Noise channel, we will treat drums like notes as far as ranges are concerned. So we need our drum data values to be in the range of $00-$7F. That's easy. We'll assign the Mode-0 sounds to $00-$0F and Mode-1 sounds to $10-$1F. Some drum data might look like this then:

```
example_drum_data:
  .byte eighth, $04
  .byte sixteenth, $1E, $1E, $1F
  .byte d_eighth, $04
  .byte sixteenth, $06, $06, $08, $08
  .byte eighth, $17, $07
  .byte loop
  .word example_drum_data
```

Since we are not using the note table for Noise, we will need to alter our Note code to check the channel and branch to different code if we are processing the Noise channel (new stuff in red):

```
.note:
  ;do Note stuff
  sta sound_temp2              ;save the note value
  lda stream_channel, x        ;what channel are we using?
  cmp #NOISE                   ;is it the Noise channel?
  bne .not_noise
  jsr se_do_noise              ;if so, JSR to a subroutine to handle noise data
  jmp .reset_ve                    ;and skip the note table when we return
.not_noise:                      ;else grab a period from the note_table
  lda sound_temp2     ;restore note value
  sty sound_temp1     ;save our index into the data stream
  clc
  adc stream_note_offset, x   ;add note offset
  asl a
  tay
  lda note_table, y
  sta stream_note_LO, x
  lda note_table+1, y
  sta stream_note_HI, x
  ldy sound_temp1     ;restore data stream index

  ;check if it's a rest and modify the status flag appropriately
  jsr se_check_rest
.reset_ve:
  lda #$00
  sta stream_ve_index, x
.update_pointer:
  iny
  tya
  clc
  adc stream_ptr_LO, x
  sta stream_ptr_LO, x
  bcc .end
  inc stream_ptr_HI, x
.end:
  rts
```

This code checks to see if the channel is the Noise channel. If so it JSRs to a special Noise subroutine. Upon return, it jumps over the note_table code and updates the volume envelope and stream pointer like normal.

So what does our special Noise subroutine, `se_do_noise`, look like? The job of `se_do_noise` will be to take the note value and convert it into something we can write to $400E (NOI_RAND). Recall that $400E expects the Mode number in bit7 and the Sound Type in bits 0-3:

```
NOI_RAND ($400E)

76543210
|   ||||
|   ++++- Sound Type
+-------- Mode
```

Mode-0 sounds don't need to be converted at all. We represent them with the values $00-$0F, which correspond exactly to the values we need to write to $400E.

Mode-1 sounds need to be tweaked a bit. We represent Mode-1 sounds with the values $10-$1F, or in binary %00010000-%00011111. Notice we identify the Mode number using bit4. Port $400E expects the Mode number in bit7 though, not bit4, so we will need to set bit7 ourselves:

```
se_do_noise:
    lda sound_temp2      ;restore the note value
    and #%00010000       ;isolate bit4
    beq .mode0           ;if it's clear, Mode-0, so no conversion
.mode1:
    lda sound_temp2      ;else Mode-1, restore the note value
    ora #%10000000       ;set bit 7 to set Mode-1
    sta sound_temp2
.mode0:
    lda sound_temp2
    sta stream_note_LO, x    ;temporary port that gets copied to $400E
    rts
```

Now everything is set. Note values of $00-$0F will get written to `stream_note_LO` directly. Note values of $10-$1F will get converted to $90-$9F first and then get written to `stream_note_LO`. Note that we do not bother clearing bit4 for Mode-1 sounds. Bit4 has no effect on $400E so we can just leave it as it is.

Drum Decay

The only thing left to add is a volume envelope for the simple drums to use. It should be short and decay to zero (silence):

```
volume_envelopes:
    .word se_ve_1
    .word se_ve_2
    .word se_ve_3
    .word se_ve_tgl_1
    .word se_ve_tgl_2
    .word se_battlekid_loud
    .word se_battlekid_loud_long
    .word se_battlekid_soft
    .word se_battlekid_soft_long
    .word se_drum_decay

se_drum_decay:
    .byte $0E, $09, $08, $06, $04, $03, $02, $01, $00  ;7 frames per drum.  Experiment to get the length and attack 
you want.
    .byte $FF

ve_drum_decay = $09
```

You can of course make multiple volume envelopes for the drums to choose from.

Conclusion

Now we can add drum data to our songs. Here is the drum data for The Guardian Legend boss song we've been using:

```
;in the song header, after the header info for Squares and Triangle:
    .byte MUSIC_NOI      ;which stream
    .byte $01            ;status byte: enabled
    .byte NOISE          ;which channel
    .byte $30            ;initial volume_duty value (disable length counter and saw envelope)
    .byte ve_drum_decay  ;volume envelope
    .word song1_noise    ;pointer to the sound data stream
    .byte $53            ;tempo

song1_noise:
    .byte eighth, $04                     ;this song only uses drum 04 (Mode-0, Sound Type 4) for a 
snare
```

```
    .byte sixteenth, $04, $04, $04
    .byte d_eighth, $04
    .byte sixteenth, $04, $04, $04, $04
    .byte eighth, $04, $04
    .byte loop
    .word song1_noise
```

Try adding some drums to your own songs!

Putting It All Together
Download and unzip the drums.zip sample files. Make sure the following files are in the same folder as NESASM3:

drums.asm
sound_engine.asm
sound_opcodes.asm
drums.chr
note_table.i
note_length_table.i
vol_envelopes.i
song0.i
song1.i
song2.i
song3.i
song4.i
song5.i
song6.i
song7.i
song8.i
drums.bat

Double click drums.bat. That will run NESASM3 and should produce the drums.nes file. Run that NES file in FCEUXD SP.

Use the controller to select songs and play them. Controls are as follows:

Up: Play
Down: Stop
Right : Next Song/SFX
Left : Previous Song/SFX

Song0 is a silence song. Not selectable.
Song1-Song7 are the same as last week, but a few of them (1, 4 and 6) have drums now.
Song8 is a new "song" that plays each of the Noise channel's 32 sounds, in order from 00-1F. First it plays them with a sustained volume envelope so that you can hear how they sound drawn out. Next they are played using the 7-frame ve_drum_decay volume envelope we made so you can hear how they sound as simple drum sounds.

Next Week: More Complex Drums, Noise SFX
