FAMICOM PARTY

Making NES Games in Assembly

All works and copyrights belong to Kevin Zurawel with his website:

https://famicom.party/book

I prefer a book-like edition, so I made this:

https://github.com/zDragon117/book-Famicom-Party-Making-NES-Games-in-Assembly

A small change I made is mention of the modern NES game Micro Mages released in 2019 which is an excellent game.

And I am hoping that someday my NES game will be mentioned too.

❖ Changelog:

2nd edition:

- Edit the cover.
- Use font QT Korrin like the website.
- Update content with a new emulator (Mesen2) and new graphics tool (NEXXT)
- Add Part III Gameplay: Chapters 16 and 17.

Table of Contents

Part I: The Setup

Introduction

Sometime in the fall of 1990, my parents gave me a Nintendo Entertainment System Action Set: a grey, boxy Control Deck, two controllers, a bright orange "Zapper" light gun, and a cartridge that contained both *Super Mario Bros.* and *Duck Hunt*.

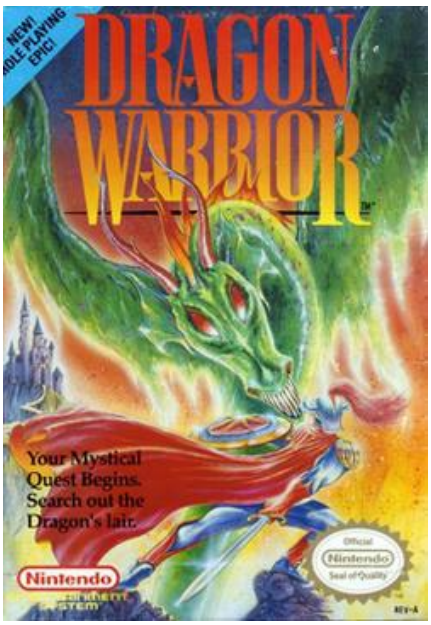
An NES console with one controller.

Photo by Evan Amos.

The Control Deck plugged into the big CRT TV we had in the basement via an RF switch, basically an antenna that fed video from the Control Deck to the TV when you tuned it to Channel 3. It was the first video game console we had ever owned, and I loved it.

I spent a lot of time in the basement that year. At first, my dad did too - he was working his way through *Super Mario Bros.*, learning where the secret Warp Zones were located and how to get past the menacing Hammer Bros. Eventually he defeated Bowser (or “the dragon”, as he called him), saved the princess, and never really played an NES game ever again.

I was hooked, though. We got a promotional letter in the mail offering a free copy of *Dragon Warrior* for subscribing to Nintendo Power magazine; signing up for a $20 magazine subscription to get a $50 game was a no-brainer.

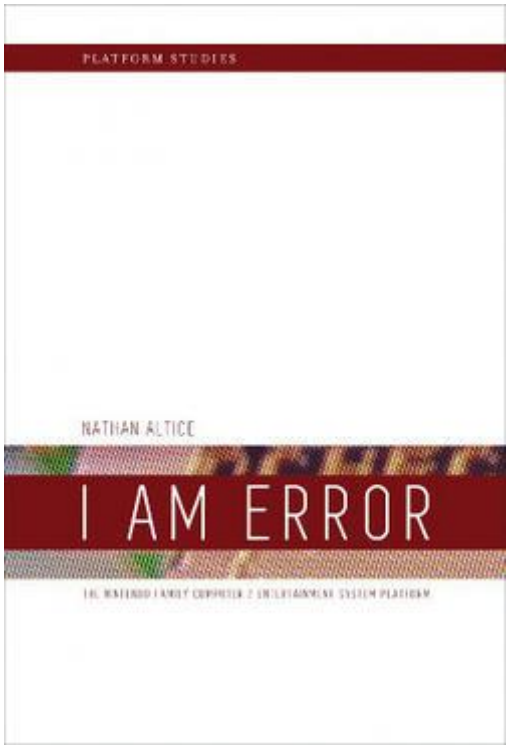


Dragon Warrior (US box art).

Dragon Warrior introduced me to RPGs in the same way that *Super Mario Bros.* had introduced me to platformers. There were plenty of other games that I spent time with, too: *Ducktales*, *Final Fantasy*, *Contra*, and *Mega Man III* were some of my favorites.

The NES ultimately set me on a career path. As a kid, I knew that video games were special, and that making video games was what I wanted to do as an adult. I had the (mistaken) belief that a career making video games meant a career **playing** video games. I started learning C++ once my family got a computer because that was what "real" programmers used. (Little did I know that NES programmers don't use C++, but in my defense I was young and naive.) I never made any actual games, though. Game programming always seemed more complicated than I was prepared for, and besides, there was no shortage of great games from other people waiting to be played.

That dream stuck with me, and after years of being a professional web developer I started learning NES development. (I got a strong nudge in the right direction from Nathan Altice's excellent *I Am Error*.)



I Am Error.

It was hard to know where to begin. There were plenty of resources around the internet, but they were all incomplete or inaccurate in some way. I got started with bunnyboy's "Nerdy Nights" series on the NintendoAge forums. Then I found tepples' NROM template on GitHub, and started learning the ca65 assembler. After months of struggling to understand PPU writes, attribute tables, and scroll registers, it all started to click. I'm glad that I had the experience of fighting with these concepts to learn on my own, but I wish that I could have had a guide that started from scratch and taught all of the essentials of NES development.

My hope is that the book you are now reading will serve as that guide.

1. A Brief Introduction to the NES

In broad terms, the NES ("Nintendo Entertainment System") is a home video game console designed to connect to a CRT television. It uses interchangeable, read-only cartridges ("Game Paks") to store games, since the system has no permanent means of storage. Input comes in the form of two controller ports on the front of the system; the standard NES controller has a 4-way, cross-shaped directional pad and four additional buttons, labelled Start, Select, A, and B.

An NES console with one controller.

Photo by Evan Amos.

The NES was first released in the U.S. in 1985; no one is sure exactly when, but most sources estimate it was in October of that year. Nintendo continued to sell the NES in the U.S. until 1995, well past the launch of its successor, the Super NES, in 1991. During that ten-year period, nearly 700 officially-licensed games were released for the NES, produced both by Nintendo directly and by a wide range of third-party developers like Konami and Capcom. It was the best-selling video game console of its generation, (David Sheff. *Game over: how Nintendo zapped an American industry, captured your dollars, and enslaved your children*. 1st edition. Random House New York, 1993. ISBN: 0679404694.) with "playing Nintendo" becoming synonymous with video games in the U.S. in the same way that "playing Atari" had been in the previous decade.

The NES did not spring fully-formed from a Nintendo R&D lab. A unique series of failures and innovations led to the system becoming a powerhouse in the home video game space. So, before we start to work with the NES, it is important to take a step back and discuss where the system came from. The historical background that led to the NES had important implications for the system's design and capabilities, which will ultimately impact the games that can (and can't) be made for the system.

■ *The American Experiment*

Nintendo began in 1889 in Kyoto, Japan, as a *hanafuda* playing cards manufacturer (*Sept. 23, 1889: Success is in the Cards for Nintendo*). When Hiroshi Yamauchi, the great-grandson of company founder Fusajiro Yamauchi, took over the business in 1949, he began to expand the company's product line, diversifying into new areas like taxi services, love hotels, and toys like the Ultra Hand (*As Nintendo Turns 125, 6 Things You May Not Know About This Gaming Giant)*. Nintendo would not find breakout success until it began creating electronic toys. Nintendo began producing the Color TV-Game series of home "Pong"-clone video game consoles in 1977, and in 1980, they released the "Game & Watch" series of handheld video games (*High Score!: The Illustrated History of Electronic Games)*.

Nintendo had released a few arcade games by this point, the most successful being 1979's *Radar Scope*, a space shooter game that was the #2 arcade game in Japan behind Namco's Pac-Man (*The Ultimate History of Video Games: From Pong to Pokemon - the Story Behind the Craze That Touched Our Lives and Changed the World*). By 1978, Namco had already found great success in American arcades, and Taito had licensed its wildly-popular Space Invaders to Midway for the U.S. market as well. Nintendo knew that the American market presented a huge opportunity, but it would need to compete with both its Japanese rivals as well as American arcade companies like Atari.

A promotional flyer for *Radar Scope*.

Radar Scope gameplay

Nintendo president Hiroshi Yamauchi thought Radar Scope would be an excellent way to start a Nintendo U.S. branch. In 1980, he gave his son-in-law, Minoru Arakawa, instructions to rent office and warehouse space for a new "Nintendo of America", and shipped him 3,000 Radar Scope machines from Japan.

Unfortunately, waiting nearly a year to begin shipment of Radar Scope machines—plus the extra two weeks it took for them to get from the west coast to Arakawa's New Jersey warehouse—meant that Radar Scope looked and felt dated, and American buyers were not particularly interested. Arakawa's sales team were only able to sell about 1,000 Radar Scope machines, with the rest sitting in the warehouse gathering dust.

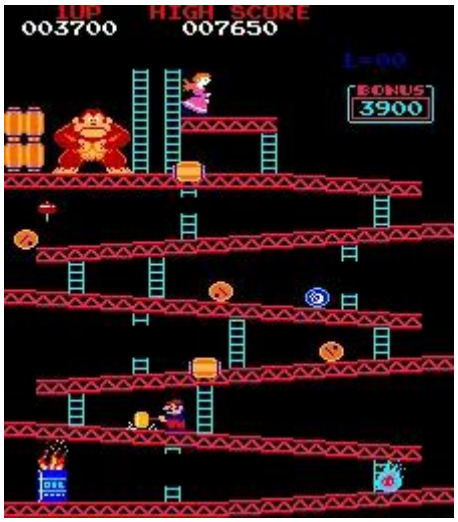
This was a serious problem for Nintendo. In the early 1980's, 2,000 arcade machines represented a substantial amount of money. For the sake of comparison, during this time period Atari would provide interested customers with a "Project Materials Cost Estimate" detailing the parts required for an arcade cabinet and their costs. A typical Atari arcade game like Missile Command contained $871 in parts, and would sell to an arcade or bar for $1,995. (For much more detail on the arcade business of the early '80's, see *Tales from the Arcade Factory Floors*). In 2018 dollars, that would be equivalent to about $2,695 in parts for a selling price of $6,170. 2,000 unsold arcade machines therefore represented over five million dollars (2018 equivalent) in sunk costs.

Hiroshi Yamauchi had an idea: what if Nintendo designed a new game that could re-use the majority of Radar Scope's components, something more appealing to American buyers? Yamauchi set aside $100,000 (1980 dollars) for the project, and handed responsibility for coming up with the new game to a young designer who had previously worked on the exterior shell of the Color TV-Game series. That designer was Shigeru Miyamoto, who would later become famous as the creator of Mario, Link, and many more of Nintendo's most famous game characters.

Shigeru Miyamoto, at E3 2013.

Photo by Jan Graber, licensed CC-BY-SA 3.0 de.

Miyamoto created a game that featured a fearless carpenter whose girlfriend had been captured by a large ape. His inspirations included the comic strip Popeye and the classic film King Kong. The result was *Donkey Kong*. To make the game more "American", Nintendo of America named the hero "Mario" (after warehouse landlord Mario Segale), and named the hero's girlfriend "Pauline". Donkey Kong was enormously popular, both in the U.S. and Japan, and it set up Nintendo as an arcade powerhouse.



Donkey Kong.

■ *The Home Market*

The success of the Radar-Scope-to-Donkey-Kong conversion project also led to another great idea: a home system which, like a Radar Scope cabinet, could run different game software on the same hardware. Home video game systems up to this point had been single-use machines, often built to play just one game (potentially with variants on a theme). Nintendo's own Color TV-Game could play six or fifteen varieties of Pong, depending on which model the user purchased, but nothing else. The most versatile system of the era was the Magnavox Odyssey, which could play over a dozen games by swapping out "game cards" included with the system. The cards did not contain any actual game data, though; they merely activated different parts of the system's internal circuitry, meaning the Odyssey's future expansion was extremely limited. The proposed Nintendo home system would operate the same way the arcade Donkey Kong machine did: game code would live on chips separate from the main processing components, with no pre-programmed game code on the system itself.

There was one major obstacle to creating such a system: cost. Video games, especially for the home, were marketed to families with children as a fun way to spend time together. If the machine were too expensive, parents would not consider buying it, no matter how good its games were. No parent would be willing to spend $871 for a home system, so Nintendo R&D had to find a way to make the Donkey Kong arcade cabinet for less. Yamauchi's goal was a system powerful enough to play Donkey Kong in the home, but able to sell for a mind-boggling ¥9,800 (equivalent to about $40 USD).

To meet that price target, Nintendo R&D started with the system's processor. The Donkey Kong arcade machine used, at its core, a Zilog Z80 processor , developed by a group of former Intel engineers who attempted to take all of the great features of the Intel 8080 series of processors and implement them in a smaller, faster, and cheaper package. The Z80 was everywhere in the Japanese computer market, powering the "MSX" standard for home computers.

A Zilog Z80 processor.

Nintendo was unable to find a manufacturing partner that would agree to its strict requirements, however. The only Japanese company desperate enough to sign a deal with Nintendo was Ricoh. Ricoh was utilizing only 10% of its chip manufacturing capability at the time, so to increase business it agreed to provide Nintendo with three million processors, purchased up-front. Nintendo was going all-in on their home console bet.

Ricoh, however, did not have a license to manufacture the Z80. Securing that license would be costly and take a long time, so Ricoh instead offered an alternative: why not use the MOS Technologies 6502 instead? Ricoh already had a license to manufacture the 6502, an 8-bit processor with a similar performance profile to the Z80. As an added bonus, it was relatively unknown in Japan (despite powering the Apple II, Commodore 64, Atari VCS, and many other computers popular in the West). Adopting the 6502 as the core of their new system would grant Nintendo a type of copy protection, since games written for Nintendo's system would not be easily portable to competitors' systems.

The MOS Technologies 6502.

Photo by Dirk Oppelt, CC-BY-SA.

The MOS Technologies 6502 was developed by a group of former Motorola engineers who attempted to take all of the great features from the Motorola 6800 series of processors and implement them in a smaller, faster, and cheaper package. (Sound familiar?) (Ricoh had a license to manufacture the 6502, but it did *not* have a license for the "binary-coded decimal" (BCD) functionality of the chip. To avoid running afoul of MOS Technologies, Ricoh's processor cut all electrical connections between the BCD module and the rest of the chip. When MOS Technologies was purchased by Commodore, the latter attempted to sue Nintendo for license violations but gave up when Commodore engineers discovered Ricoh's fix). You can still buy 6502-based processors today from Western Design Center (WDC), and they are commonly used in industrial applications. The 6502 became available for purchase in 1975, meaning it was already ten years old when Nintendo chose to use it in its new home system. Individual 6502 processors in 1975 sold for $25 each, and Nintendo's bulk order drove the per-processor price even lower. Ricoh paired its 6502-based processor with a custom "Picture Processing Unit" (PPU) to handle drawing graphics to the screen. The PPU allowed for sophisticated graphics, supporting a 64 color palette, a 256x240 pixel resolution display, 64 hardware sprites, and a hardware-scrolling background layer.

With the internals set, all that remained was to design the outer casing of the system and give it a name. Nintendo chose to call their new home system the "Family Computer", shortened to "Famicom". Its case was designed to make it look like a fun toy: red and white, with gold accents. The Famicom had two controllers hard-wired to the sides of the console, with a 15-pin expansion port on the front. Game cartridges were inserted vertically into a slot on top of the console.

The Nintendo Famicom.

Photo by Evan Amos.

The Famicom launched in July 1983 to great success, selling 500,000 consoles in two months despite only having three available games — Donkey Kong, Donkey Kong Jr., and Popeye. While Nintendo R&D was not able to hit Yamauchi's ambitious ¥9,800 target, the system did sell for ¥14,800 (equivalent to $65 USD), certainly cheap enough to appeal to parents.

■ *An American Renaissance*

With its popularity in Japan, bringing the Famicom to the U.S. was a no-brainer. But the American market was having issues of its own; 1983 was the year of the "Atari Crash" in the U.S., when retailers abandoned the idea of home video games after overproduction and a lack of quality control led to massive unsold inventories of Atari 2600 cartridges. U.S. retailers were not interested in buying inventory of a new video game system.

Nintendo spent two years re-designing the Famicom for a post-video-game market. It could not look like a video game machine, so its outer casing became a grey rectangle with a prominent front flap - a move meant to make it look like a VCR. The Famicom's small, colorful game cartridges became large, grey "Game Paks", mostly containing empty space but now large (and uniform) enough to bear a passing resemblance to VHS tapes. Finally, the system needed a name change. "Nintendo Entertainment System" positioned the system as part of a home theater setup, something that would fit in alongside stereo equipment.

The re-designed system was almost ready, but Nintendo felt it still needed something unique to make it a "must-buy" product. The "Deluxe Set" included a "Zapper" light gun, perfect for shooting games like *Duck Hunt*, and R.O.B., the "Robotic Operating Buddy". This was no mere video game system - it came with a *robot!* The cameras that formed R.O.B.'s "eyes" could detect coded patterns of flashing light from the TV, which would instruct R.O.B. to turn left or right or move its arms up or down. R.O.B. could be used with two games, *Gyromite* and *Stack-Up*, neither of which were particularly good. But games were secondary: R.O.B. existed solely to sell the NES to American parents who had already been burned by the Atari 2600.

R.O.B.

Photo by Evan Amos, CC-BY-SA 3.0.

There is one more piece of the Famicom's re-branding to discuss before we move on to programming the hardware. A key cause of the 1983 crash was a lack of quality control or licensing for Atari 2600 games. Video games were big business, and any company that could hire a programmer or two wanted to find a way to sell Atari games. The result was a flood of games that often barely worked. Most consumers, after being burned a few times by paying full price for terrible games, stopped buying them, and the video game market imploded.

Desperate to prevent this situation for the NES, Nintendo added a chip to the NES motherboard called "10NES" (sometimes referred to as "CIC", for "Checking Integrated Circuit").

The 10NES chip on an NES cartridge.

Photo by (you guessed it) Evan Amos.

The 10NES chip acted as a cryptographic lock and key. The console and cartridge each had their own CIC chip; the two chips would each calculate a value based on a specific algorithm, and the cartridge would send its answer to the console for comparison. If the results matched, the system would boot as normal. If they did not match, the CIC chip in the console would reset and try again. (Incidentally, this is why you would often have to "blow on the cartridge" to get an NES game to work. If the connectors on the cartridge did not make a secure connection to the console, the 10NES chip would be unable to transmit its code, and it would go into a reset loop. Blowing on the cartridge actually did nothing; what fixed the problem was removing and re-inserting the cartridge to re-seat the connectors.) The design of the 10NES chip as well as the specific algorithm running on it were patented by Nintendo, meaning that only Nintendo could legally manufacture 10NES chips. As a result, only Nintendo could manufacture cartridges that would work with the NES — unlicensed cartridges would not have the chip, and the NES console would refuse to boot them. Nintendo would have final say on what games were released for its platform.

2.Fundamental Concepts

What is a computer?

This sounds like a simple question, but it cuts to the heart of what we do as programmers. For now, let's just say that a "computer" is anything that executes a program. A "program" is just a series of instructions, and executing a program means starting at the beginning of the series of instructions and following them one-by-one. (If you read a program and follow the instructions yourself, congratulations! You are a computer!)

Every computer has a specific set of instructions that it knows how to follow. We call this set of instructions a computer's instruction set (very creative, I know). There are a number of ways to represent an instruction set, but for now let's assume that the instructions in the instruction set are represented by numbers. So, a program is just a list of numbers, each one defining a certain action to perform. Here's a hypothetical example instruction set:

- 1: move forward
- 2: turn left
- 3: turn right

A program running on a computer with this instruction set that moves forward three times, turns right, moves forward twice, turns left, and moves forward four times would look like this:

```
1 1 1 3 1 1 2 1 1 1 1
```

■ *Working with Data*

Often the instructions that the computer needs to execute take some form of data as well. A common thing that computers do is add numbers together; it is much simpler to have one instruction that adds rather than a whole set of instructions like this:

- 1: add 1
- 2: add 2
- 3: add 3
- 4: add 4

Or, alternatively, a single "add 1" instruction that you have to call multiple times, which would be equally difficult to use. A program that adds 1,000 to a number would take 1,000 times as much storage space (and take 1,000 times as long to execute) as a program that just added 1!

The data that goes with an instruction has to be part of the program, somewhere. Different programming languages take different approaches to this problem. Some programming languages require you to keep "code" (instructions) and data completely separate, while others combine the two. Each approach has its pros and cons, but for now let's look at combined instructions and data.

For our hypothetical "add some numbers" computer, the instruction set might look like this:

- 1: store next number as "first number"
- 2: add next number to first number, if stored

A program that adds the numbers 2 and 7 would look like this:

```
1 2 2 7
```

Stepping through the program one number at a time, we see the instruction "1", "store the next number as first number". The next number is "2", so 2 is stored as the first number. We then see the instruction "2", "add next number to first number". The next number is "7", so our program adds 7 to 2, with the result being 9. Here the data and instructions are intermixed. Seeing "1 2 2 7", it is impossible to know which "2"s are the instruction "add next number to first number" and which are the literal number "2" without starting at the beginning and stepping through the entire program.

Where does the result (9) live? How do we do anything with the result later in our program? And what does it mean to "store" something?

■ *Processor Registers*

As we have just seen, programs often need a place to temporarily store some data. Most computers accomplish this by providing *registers* - small places inside the processor that can each hold one value. "Values" here are really just numbers; as we've done with instructions in the instruction set, we can take any kind of value and represent it with a number as long as we have some kind of mapping between the numbers and the things they represent. As an example, Unicode represents every possible character, from every writing system on Earth, with a 32-bit number. (More on "bits" in just a bit.)Registers can be fully generic, or they can be tied to specific kinds of functionality. The NES' processor, for example, has a register called the *accumulator*, often abbreviated to "A", that handles all math operations. The 6502's instruction set has instructions that work like this:

- store next number in accumulator
- add next number to accumulator, result in accumulator
- put number from accumulator somewhere

This solves the problem of where to put numbers and how to access them. But there is still one open question: when we "put number from accumulator somewhere", where is "somewhere"? The 6502 in the NES only has three registers, so complicated programs like games can't use *only* registers for storing results.

■ *Memory*

Computers make available to programs some amount of (non-permanent) memory to store things temporarily, allowing the computer to have a small number of (expensive) registers while still allowing for a reasonable amount of values to be stored outside of the program itself. This memory is made available as a series of register-sized boxes, each holding one value and referred to by number. The NES provides your program with two kilobytes (2KB) of memory space, numbered from zero to 2,047 - a memory space's number is referred to as its *address*, just like a house number. So, the 6502 instruction set we looked at above is really more like this:

- store next number in accumulator
- add next number to accumulator, result in accumulator
- put number from accumulator into memory address of next number

■ *Representing Data*

This leads to our final question for this chapter - how are all of these numbers represented inside the computer?

Up to this point, we have been using "standard", decimal (base 10) numbers. These are the kinds of numbers that we use every day - numbers like "2" or "7" or "2,048". Computers, however, operate through electrical currents that can either be "on" or "off", with no in-between. These currents form the basis of all data inside the computer, and as a result computers use binary (base 2) numbers.

The smallest unit of information a computer can process is a "bit", or "binary digit". A bit stores one of two values - 0 or 1, "on" or "off". If we combine more than one bit as a single number, we can represent a larger range of values. Two bits, for example, can represent four different values:

```
00 01 10 11
```

Three bits let us represent eight different values:

```
000 001 010 011 100 101 110 111
```

Each bit we add allows us to represent twice as many values, in the same way that each decimal digit we add to a decimal number lets us represent ten times as many values (1 → 10 → 100 → 1,000). Eventually, we reach eight bits working together to represent a single value, which is so common that it has its own name: a *byte*. A byte can store one of 256 values. Since four bits is half of a byte, it is occasionally referred to as a *nybble*, which is incredibly cute. A nybble can hold one of 16 values.

Computers, including video game consoles, are often described as being a certain number of bits. Modern desktop/laptop computers are generally "64-bit", older versions of Windows like Windows XP are called "32-bit operating systems", and the NES is an "8-bit" system. What all of these are referring to is the *register size* of the computer - how many bits a single register can hold at one time. Slightly complicating things, the NES' *address bus* is actually 16 bits wide, meaning the NES can deal with 65,536 different memory addresses instead of just 256. Each memory address still only holds one byte, though. Since the NES is an "8-bit" computer, its registers each hold an 8-bit value (one byte). Additionally, each memory address can hold one byte.

How do we work with numbers larger than 255? It is not uncommon for someone playing *Super Mario Bros.* to get a score in the tens of thousands, far too large a number to represent in one byte. When we need to represent a value that is larger than what one byte can hold, we use more than one byte. Two bytes (16 bits) can hold one of 65,536 different values, and as we add more bytes our representational power increases sharply. Three bytes can store a value up to 16,777,215, and four bytes can store a value up to 4,294,967,295. When we use more than one byte in this way, we are still limited by the register size of the computer. To work with a 16-bit number on an 8-bit system, we need to fetch or store the number in two parts - the "low" byte on the right, and the "high" byte on the left.Dealing with these larger-than-one-register values is why processors have a defined *endianness* - i.e., which byte comes first when dealing with large numbers. "Little-endian" processors, like the 6502, take the low byte first followed by the high byte. "Big-endian" processors, like the Motorola 68000, do the opposite, expecting the high byte to come first followed by the low byte. Most modern processors are little-endian, since Intel's hugely popular x86 architecture is little-endian.

Since the 6502 that powers the NES works with eight bits of data at a time, smaller numbers still take eight bits to represent. This can be inefficient, so it is common to represent multiple smaller values in a single byte when needed. One byte can hold two four-bit numbers, or four two-bit numbers, or even eight individual on/off values (we call these "*flags*").

As an example, the byte `10110100` could represent:

- One 8-bit value: 180
- Two 4-bit values: 11 (`1011`) and 4 (`0100`)
- Four 2-bit values: 2 (`10`), 3 (`11`), 1 (`01`), and 0 (`00`)
- Eight on/off (or true/false) values: on, off, on, on, off, on, off, off
- Any other combination of bit lengths that add up to eight

To help us talk about these multiple-values-in-one-byte scenarios, it's common to number the bits inside a byte, much like how we can name the "low" and "high" bytes in a 16-bit value. The bit on the far right is "bit 0", and bits count up toward "bit 7" on the far left. Here's an example:

```
byte:  1 0 1 1 0 1 0 0
bit #: 7 6 5 4 3 2 1 0
```

■ *Making Data Human-Readable*

As we have seen, bytes are a very flexible way to represent a variety of data types in a computer system. The downside of using bytes, though, is that they are difficult to read. It takes work to see "10110100" and mentally translate it to the decimal number "180". When an entire program is represented as a series of bytes, the problem is made exponentially worse.

To solve this problem, most code is represented as *hexadecimal* numbers. "Hexadecimal" is a fancy way of saying "base 16"; a single hexadecimal ("hex") digit can hold one of sixteen values. Here are the numbers from zero to fifteen represented in hexadecimal:

```
0 1 2 3 4 5 6 7 8 9 a b c d e f
```

Hex notation is useful because a hex digit and a nybble store the same range of values. This means we can represent a byte using two hex digits, a much more compact and easy-to-read representation.

Decimal	Binary (1 byte)	Hex
0	00000000	00
7	00000111	07
31	00011111	1f
94	01011110	5e
187	10111011	bb
255	11111111	ff

Dealing with numbers that could be decimal, binary, or hexadecimal can be confusing. The number "10", for example, represents 10 if it is decimal, 2 if it is binary, or 16 if it is hexadecimal. To make clear what value we are referring to, the convention is to use prefixes. `10` is a decimal number, `%10` is a binary number, and `$10` is a hexadecimal number.

■ *Putting It All Together*

Now that we've covered a wide range of topics related to how computers (and programs) work, let's take one more look at the entire process of what happens when you run a program.

First, the program itself is represented as a series of bytes (what we call *machine code*). Each byte is either an instruction for the processor, or a piece of data that goes with an instruction.

The processor starts at the beginning of the program and repeatedly carries out a three-step process. First, the processor *fetches* the next byte from the program. The processor has a special register called the *program counter*, which keeps track of what byte number of the program is next. The program counter (or "PC") works together with a register called the *address bus*, responsible for retrieving and storing bytes from the program or from memory, to fetch bytes.

Next, the processor *decodes* the byte it fetched, figuring out which entry in its instruction set the byte corresponds to (or which instruction the data byte goes with). Finally, it *executes* the instruction, which will make changes to the processor registers or memory. The processor increments the program counter by one to fetch the next byte of the program, and the cycle begins again.

3. Getting Started

Let's get started actually programming for the NES! In this chapter, we're going to set up a development environment, installing all of the tools that you will need to work through this book, and then we will build and run the most basic game possible just to make sure everything is working.

■ *Setting Up Your Development Environment*

Here are all of the tools that we will be installing. Some of these will be used right away (and all the time), while others are more specialized and won't come into play until much later. For each category, I'm including the specific software I will be using in this book; there are many other choices, so feel free to experiment with other tools once you get comfortable with my recommendations.

- A text editor (your choice)
- An assembler/linker (ca65 and ld65)
- An emulator (Mesen2)
- A graphics tool that can read/save NES formatted images (NEXXT)
- A music composition tool (FamiStudio)

■ *Text Editor*

First, you will need a *text editor*. I assume that you have previous programming experience and that, as a result, you already have a favorite text editor. If not, here are a few programs that you may want to try.

- Sublime Text. Cross-platform, popular with web developers, easy to get started with and powerful tools once you get familiar with the basics.
- Atom. Basically GitHub's answer to Sublime Text. Cross-platform, highly configurable.
- Visual Studio Code. Microsoft's robust text editor platform. Tailored for web development but extensible for any kind of programming. Also cross-platform, not limited to Windows.
- Vim, emacs, nano, etc. Command-line text editors of yore. (I personally use Vim, but your mileage may vary.)

■ *Assembler and Linker*

An *assembler* compiles your assembly code (what we will be writing in this book) into machine code, the raw stream of bytes that the processor reads. A *linker* takes a group of files that have been run through the assembler and turns them into a single program file. Since each processor has its own machine code, assemblers usually target only one type of processor. There are many assemblers and linkers to choose from for the 6502, but for this book we will be using ca65 and ld65. They are open-source and cross-platform, and have some very useful features for developing larger programs. ca65 and ld65 are part of the larger "cc65" suite of programs, which include a C compiler and more.

❖ Mac

To install ca65 and ld65 on a Mac, first install Homebrew (https://brew.sh/), a Mac package manager. Copy and paste the command from the homepage into a terminal and press enter; follow the instructions and Homebrew will be ready to use. Once you have Homebrew, type `brew install cc65` and press enter.

❖ Windows

On Windows, you'll need to download ca65 and ld65 to a specific directory on your computer. Download the latest "Windows Snapshot" from the main cc65 project page (https://github.com/cc65/cc65). Unzip the contents to `C:\cc65`. You'll also need to update your system path to make ca65 and ld65 available from any directory. The process for doing this varies depending on which version of Windows you are using. On most newer versions of Windows, you can right-click "My Computer", select "Properties", then "Advanced System Settings" and finally "Environment Variables". You'll want to find the entry for `%PATH%` and add `C:\cc65\bin` to the end of it.

❖ Linux

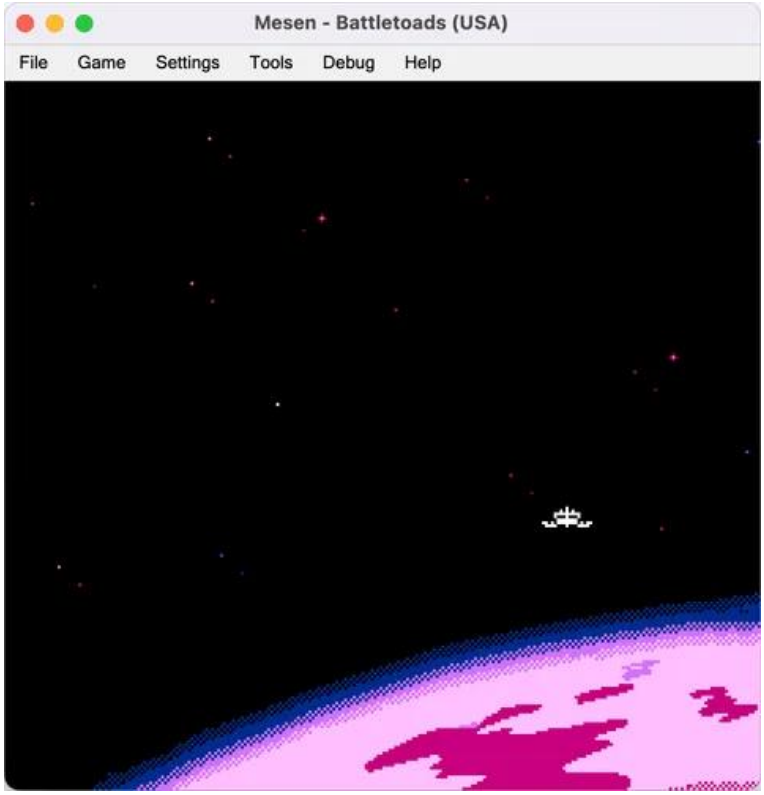
You will need to build cc65 from source. Thankfully, this is a fairly simple process. First, make sure you have git and a basic build environment - on Ubuntu, for example, `sudo apt-get install git build-essential` should do it. Then, navigate to the directory where you want to install cc65, clone the cc65 repository, and build it:

```
git clone https://github.com/cc65/cc65.git
cd cc65
make
```

Finally, make the cc65 programs available from any directory with `sudo make avail`. This will add symlinks from your cc65 folder to `/usr/local/bin`.

■ *Emulator*

An emulator is a program that runs programs intended for a different computer system. We will use an NES emulator to run the programs that we create on the same computer used to develop them, instead of requiring a hardware NES. There are a number of NES emulators available (and, once you have a solid grasp of NES development, it's fun to try to make your own!), but for this book we will be using Mesen (https:/github.com/SourMesen/Mesen2). It is cross-platform and features powerful debugging tools, which will be useful as we write programs.



Mesen2.

- Windows / Linux / Intel-based Mac

For these systems, download the latest development release (https://nightly.link/SourMesen/Mesen2/workflows/build/master), then unzip. You will need to install .NET Runtime v6 (https://dotnet.microsoft.com/en-us/download/dotnet/6.0) on all systems. Non-Windows systems will also need to install SDL2 through your OS' package manager or Homebrew on Mac.

- ARM-based Mac (M1, M2, etc.)

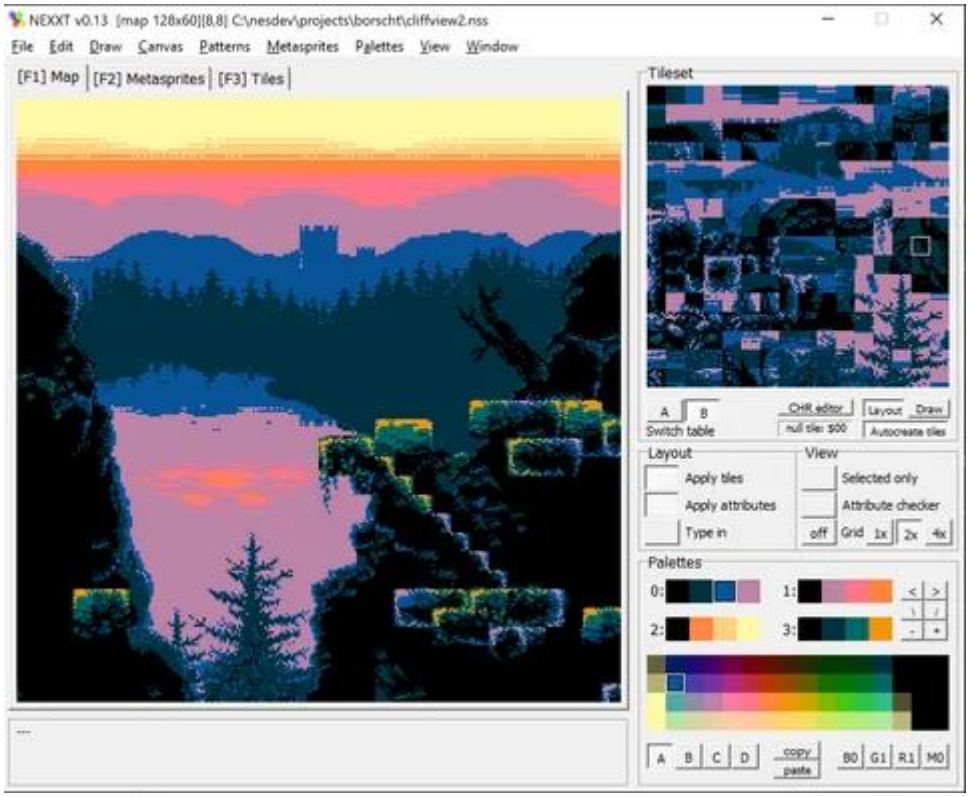
On these systems, you will need to build Mesen2 from source. First, install SDL2 via Homebrew (`brew install sdl2`), then install the .NET 6 SDK (https://dotnet.microsoft.com/en-us/download/dotnet/6.0). Download Mesen2's source code and run `make`:

```
git clone https://github.com/SourMesen/Mesen2.git
cd Mesen2
make
```

When the build process is complete, you will have `Mesen.app` in the bin/osx-arm64/Release/osx-arm64/publish directory inside of the `Mesen2` directory where you ran `make`. Move `Mesen.app` into your Applications folder.

■ *Graphics Tools*

The NES stores graphics in a very different format from common image types like JPEG or PNG. We will need a program that can work with NES images. There are plugins for large graphics packages like Photoshop or GIMP, but a smaller, purpose-built tool is often a better choice. For this book, we will be using NEXXT (https://frankengraphics.itch.io/nexxt), a derivative of NES Screen Tool (https://shiru.untergrund.net/software.shtml). NEXXT is a Windows-only program, but runs well on other platforms under WINE.



NEXXT.

❖ Windows

Download NEXXT from itch.io (https://frankengraphics.itch.io/nexxt). Unzip, and run NEXXT.exe.

❖ Linux

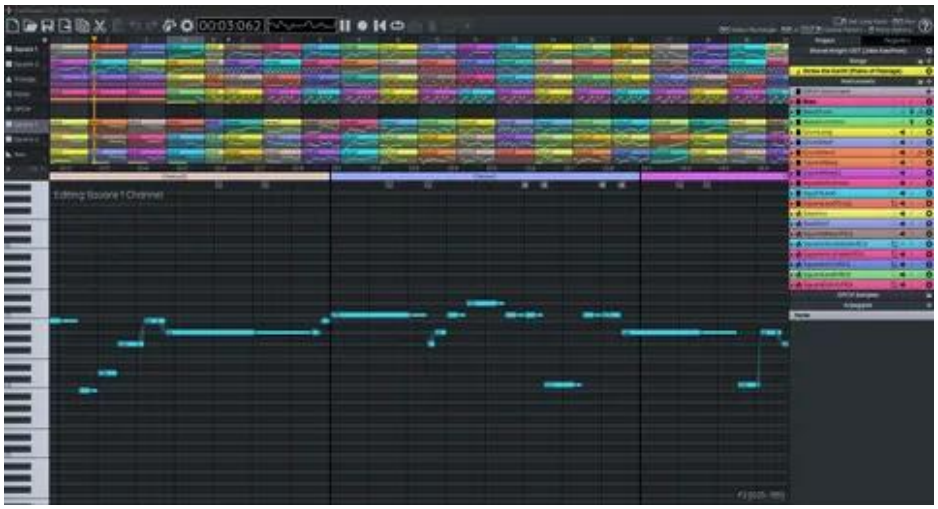
Since NEXXT is a 32-bit, Windows-only program, Linux users will have to run it via WINE (https://www.winehq.org/). Install WINE via your distribution's package manager, then run NEXXT.exe with it (`wine NEXXT.exe`).

❖ Mac

Newer Mac versions (starting with Catalina) are 64-bit only, and 32-bit software will not run even in a standard WINE install. Thankfully, 32-bit Windows programs can be run using the "Crossover" version of WINE, which is able to translate 32-bit code into 64-bit code on the fly. Install Crossover via Homebrew (`brew install --cask gcenx/wine/wine-crossover --no-quarantine`). Download NEXXT as above and unzip, then run `wine64 ./NEXXT.exe` in the directory where you unzipped it.

■ *Music Composition Tools*

As with graphics, NES audio is a set of instructions to an audio processor rather than something like an MP3. The most popular program for creating NES audio is FamiTracker (http://www.famitracker.com/), which is powerful but complex and Windows-only. For this book, we will be using FamiStudio (https://famistudio.org/), which is cross-platform, has a friendlier interface, and outputs directly into an easy-to-integrate format.



FamiStudio.

❖ Windows / Mac / Linux

Download the latest release from the FamiStudio website (https://famistudio.org/).

■ *Putting It All Together*

Now that you have all of the tools installed, it's time to make sure that they work. We are going to create the "Hello World" of NES games: filling the entire screen with one color.

Open your text editor and create a new file, `helloworld.asm`. Copy and paste the following code into the file:

```
1   .segment "HEADER"
2   .byte $4e, $45, $53, $1a, $02, $01, $00, $00
3
4   .segment "CODE"
5   .proc irq_handler
6     RTI
7   .endproc
8
9   .proc nmi_handler
10    RTI
11  .endproc
12
```

```
13  .proc reset_handler
14    SEI
15    CLD
16    LDX #$40
17    STX $4017
18    LDX #$FF
19    TXS
20    INX
21    STX $2000
22    STX $2001
23    STX $4010
24    BIT $2002
25  vblankwait:
26    BIT $2002
27    BPL vblankwait
28  vblankwait2:
29    BIT $2002
30    BPL vblankwait2
31    JMP main
32  .endproc
33
34  .proc main
35    LDX $2002
36    LDX #$3f
37    STX $2006
38    LDX #$00
39    STX $2006
40    LDA #$29
41    STA $2007
42    LDA #%00011110
43    STA $2001
44  forever:
45    JMP forever
46  .endproc
47
48  .segment "VECTORS"
49  .addr nmi_handler, reset_handler, irq_handler
50
51  .segment "CHARS"
52  .res 8192
53  .segment "STARTUP
```

Next, we need to use our assembler. In the directory where you saved helloworld.asm, run `ca65 helloworld.asm`. The result should be a new file, helloworld.o. This is an *object file* - machine code. But it is not in a format that is ready to plug into an emulator just yet. To do that, we need to run the linker. In the same directory, run `ld65 helloworld.o -t nes -o helloworld.nes`. This should result in a new file, helloworld.nes - a "ROM" file for the emulator.

Open Mesen2 and choose “Open” from the “File” menu. Select the helloworld.nes file you just created and click Open. The result should be a green screen.

The green screen here is an actual, running NES emulator in your browser! I am using binjnes (https://github.com/binji/binjnes) by Ben Smith. Whenever we compile a .nes file, I will include a running demo like this one. (It's hard to tell in this case, but the emulator is actually running at 60fps.)

■ *Next Steps*

If you were able to see the green screen in Mesen2, congratulations! Your development environment is ready to use. In the next chapter, we will discuss what the code you copied and pasted is actually doing, and learn a bit about how the NES hardware works.

4.NES Hardware

Before we look at assembly, let's start with an overview of the NES itself.

■ *The Console*

If you were to open up an NES console and take a look at its insides, you would see something like this:

An NTSC (US/Japan) NES motherboard. Photo by Evan Amos.

Visually, the NES motherboard is dominated by the cartridge connector at the top and two large chips. On the left side is this chip, labelled "RP2A03":

The Ricoh 2A03 CPU/APU.

And on the right side is this other chip, labelled "RP2C02":

The Ricoh 2C02 PPU.

Together, these two chips provide all of the NES' processing power. The first chip, the 2A03, is the NES' CPU - "central processing unit". The 2A03 is based on the MOS Technologies 6502 processor, with a few special tweaks by its producer, Ricoh. When I say the 2A03 is "based on" the 6502, there was really only one major difference between the two: the 2A03 lacks support for a feature in the 6502 called "binary coded decimal" (BCD) mode. BCD allows the binary numbers the CPU works with to act like decimal numbers when adding or subtracting. The BCD circuitry in the 6502 was covered by a separate licensing agreement that Ricoh was not party to, so Ricoh could not legally include BCD mode in its processors even though it had full schematics of how the mode was implemented in silicon. To get around this issue, Ricoh fabricated a "full" 6502 but cut all electrical connections between the BCD portion of the chip and the rest of the processor. Ricoh also included within the 2A03 a full APU - "audio processing unit" - that handles music and sound effects.

The second chip, the 2C02, is the NES' PPU - "picture processing unit" -, what you might think of nowadays as a "graphics card". The PPU receives instructions from the CPU and translates them into output for the screen. The CPU knows nothing about how televisions work; it leaves all of that processing to the PPU.

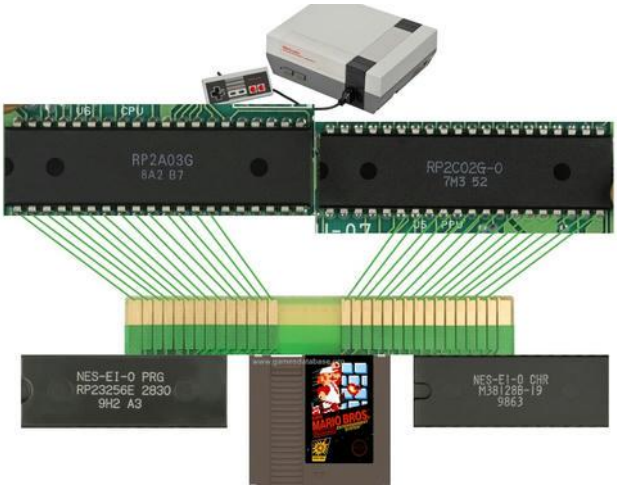
■ *Cartridges*

NES games are distributed via plastic cartridges (or "Game Paks", as Nintendo called them in the US). Inside each cartridge is a small circuit board that looks something like this:

The cartridge circuit board for *Tetris*. Photo by Evan Amos.

Much like the console motherboard, cartridge circuit boards are dominated by two large chips. The left-side chip is labelled "PRG", and the right-side chip is labelled "CHR". PRG-ROM is "program ROM", and it contains all of the game's code (machine code). CHR-ROM is "character ROM", which holds all of the game's graphics data. There are two additional chips on this board. To the right of the two main ROMs is the "CIC" (Checking Integrated Circuit) or "lockout" chip, an attempt to ensure that only Nintendo-produced cartridges would work in an NES. (The CIC chip is also the reason why your NES games often flicker repeatedly until you take them out and put them back in, hopefully *without* blowing on the contacts first.) Under the two ROM chips is the "MMC1", a memory management controller which we will cover in detail much later in this book.

These two ROM chips have their pins connected to a series of gold connectors at the edge of the cartridge board. When the cartridge is inserted into the console's cartridge slot, the gold connectors make electrical contact with an identical set of connectors in the console. The result is that the PRG-ROM is electrically wired directly to the 2A03 CPU, and the CHR-ROM is wired directly to the 2C02 PPU.



A diagram showing how the cartridge's ROM chips connect to the console's processors.

■ *What does this have to do with the test project?*

If you look back at the test project's assembly source, you'll notice several lines that start with `.segment`. These are *assembler directives* - something we'll explore in the next chapter - that dictate where in the finished ROM file certain pieces of code should go.

```
.segment "HEADER"
.segment "CODE"
.segment "VECTORS"
.segment "CHARS"
.segment "STARTUP"
```

Two of these segments are not part of the game's code per se. The `STARTUP` segment doesn't actually do anything; it's needed for C code compiled down to 6502 assembly, but we won't be using it. The `HEADER` segment contains information for emulators about what kind of chips are present in the cartridge.

The other segments line up to the PRG/CHR split. `CODE` is, of course, the game code that is stored in the PRG-ROM. `VECTORS` is a way to specify code that should appear at the very end of the PRG-ROM block (for reasons we will discuss later). And `CHARS` represents the entire contents of the CHR-ROM, generally included as a binary file.

So, what are the contents of our CHR-ROM chip, and therefore the graphics that our game will display?

```
.segment "CHARS"
.res 8192
```

`.res` is another assembler directive that tells the assembler to "reserve" a certain amount of blank space - in this case, 8,192 bytes. But if the entire CHR-ROM is empty, where does the green background that our test project displays come from?

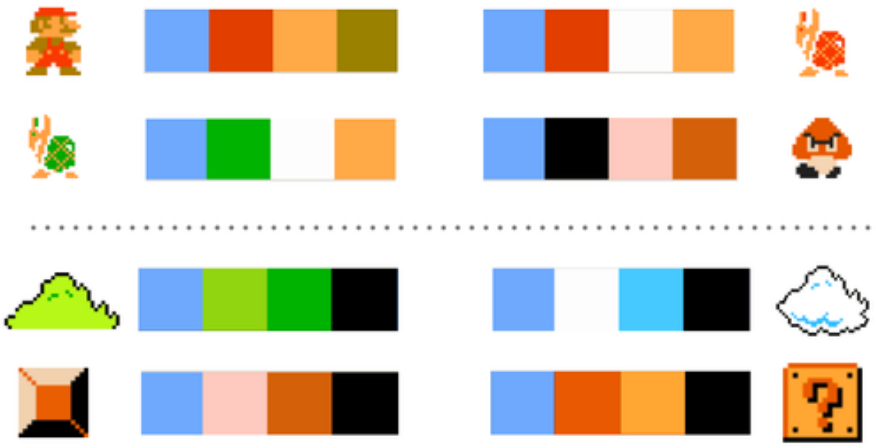
■ *Colors and Palettes*

When I said before that the CHR-ROM chip contains "all" of the graphics for our game, I was simplifying a bit. To be more precise, the CHR-ROM chip contains *patterns* of different colors to be displayed in the game. The colors themselves, however, are part of the PPU itself. The PPU knows how to display a fixed set of 64 colors. Observant readers may notice that the color palette image here only contains 56 colors, not 64. The reason for this is that eight of the 64 colors the PPU knows how to display are just black. This is due to a quirk in how NTSC CRT televisions display color, not a mistake in designing the hardware.

The NES color palette.

Due to hardware limitations, you can't use all 64 colors at the same time. Instead, you assign colors to the system's eight four-color *palettes*. Four of these palettes are used for the "background", and the other four are used for the "foreground".

The eight palettes have one additional limitation: the first color of each of the eight palettes must be the same. This first color is used as a "default" background color (when nothing else is being drawn to the background for a given pixel, the default color is used), and it serves as a transparency color for the foreground (foreground pixels drawn with the first color act as transparent, allowing the background pixel "behind" them to show through). These limitations mean that the NES can display a maximum of 25 colors at a time - the one "default" color, and eight palettes with three other colors each.



The eight palettes used in World 1-1 of Super Mario Bros. The top four palettes are used for the foreground and the bottom four palettes are used for the background. Notice that many elements, both foreground and background, are composed of the same patterns but different palettes - e.g. the different-colored turtles in the foreground and the bush/cloud graphics in the background. The blue first color of each palette is the color of the "sky" in the background of World 1-1.

The PPU refers to each color with a one-byte number. Palette information is stored in a specific location of the PPU's memory map (separate RAM that is only accessed by the PPU itself). The 32-byte region in PPU memory from $3f00 to $3f20 holds the contents of the eight palettes in sequence. The "default color", being the first color of each palette, is stored at $3f00, $3f04, $3f08, $3f0c, $3f10, $3f14, $3f18, and $3f1c.While it is common to repeat the first color for each palette manually, technically the PPU only cares about the value written to $3f00. Our test program takes advantage of this fact to cut down on the amount of code that needs to be written.

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

The NES color palette, overlaid with the byte the PPU uses to refer to each color.

■ *Back to the test project*

Referencing the color chart above, the green background color of our test project is $29. If we look back at our test project's code, we find this:

```
40      LDA #$29
```

This line of assembly code tells the processor to get ready to use the number $29, which corresponds to the green we display in the background. We'll look at the details of how this line (and much of the rest of the test project's source code) works in the next chapter.

5.Introducing 6502 Assembly

The "Hello World" program you created in Chapter 3 is probably unlike any other you have ever seen: 44 lines of code, mostly composed of three-letter acronyms and numbers, just to change the color of the screen. What did you copy and paste?

As you can probably guess from the title of this chapter, helloworld.asm contained assembly code. I prefer to use the .asm extension for assembly files, but the choice is pretty arbitrary. Some developers prefer to use .s for their assembly files. Ultimately, the choice is yours - ca65 will assemble any file extension as long as the contents are assembly code. Assembly is one layer of abstraction above machine code. Writing assembly requires detailed knowledge of a processor's instruction set, but it is far more readable than machine code's stream of bytes. Let's look at some example assembly code to see how it works:

```
1   .proc main
2     LDX PPUSTATUS
3     LDX #$3f
4     STX PPUADDR
5     LDX #$00
6     STX PPUADDR
7
8   copy_palettes:
9     LDA palettes,x
10    STA PPUDATA
11    INX
12    CPX #$20  ; 32 colors total
13    BNE copy_palettes
```

- Separation of instructions and data

The three-letter words in all caps on lines 2-6 and 9-13 are called *opcodes*. Each one represents an instruction from the processor's instruction set, but instead of referring to them by number, they now have names. `LDA`, for example, means "load accumulator". We'll be learning a few dozen opcodes over the course of this book; there are 56 "official" 6502 assembly opcodes in total.

Having instructions represented with short "words" means that we can now distinguish between instructions and data visually, instead of having to work through the code byte-by-byte. Anything to the right of an opcode is data that goes with that instruction. We call data for an instruction an *operand*. On line 3, for example, `LDX #$3f` is an instruction to "load the X register with the hex value $3f". `LDX` is the opcode, and `#$3f` is the operand. Here, and throughout this book, I will be presenting opcodes in all-caps, but this is a personal choice. The assembler will accept lowercase opcodes as well, and some developers prefer lowercase. The same applies to operand values - `$3f` can also be written as `$3F`.

❖ Constants and labels

While the separation of instructions and data is incredibly useful, assembly gives us several other tools as well. The operands on lines 2, 4, 6, and 10, in all-caps, are *constants* defined elsewhere. When the assembler (ca65) is run, it will replace the name of the constant (e.g. `PPUSTATUS`) with its value (e.g. `$2002`). You can make a constant by using the equal sign: `PPUSTATUS = $2002`.

Assembly also lets us set *labels* - named places in our code that can be referenced later. On line 8, a new label, `copy_palettes`, is defined. What makes a label a label is the colon (:) that it is followed by. On line 13, the label is used as an operand to the `BNE` opcode. We'll talk about the specifics of how it works later, but line 13 is essentially repeating the code from lines 8-13 again. When the assembler is run, it replaces labels in your assembly code with actual memory addresses in the machine code output.

❖ Comments

You can add comments to your assembly code using the semicolon (;) character. Anything from the semicolon to the end of the line is treated as a comment and will be stripped out when you run your code through an assembler. Comments are useful for reminding yourself what a certain piece of code is meant to do. In our example code above, you can see a comment on line 12.

❖ Assembler directives

Finally, assembly code gives us *directives* - instructions for the assembler that affect the conversion from assembly to machine code, rather than instructions for the processor where the machine code will be executed. Directives start with a period (.). The example code uses the `.proc` directive on line 1 to indicate a new lexical scope (more on that later). Another common directive is `.byte`, which indicates that the following bytes should be copied raw into the machine code output rather than trying to process them as opcodes or operands.

■ *Your First Opcodes: Data Movement*

Now that we have seen what 6502 assembly looks like, let's start learning it! I have broken out the opcodes that you will learn over the course of the book into seven major groups. The first group comprises instructions that move data between registers and memory.

❖ Loading data: `LDA, LDX, LDY`

The "LD" commands load data into a register. As a reminder, the 6502 has three registers to work with: "A" or "accumulator", which can do math, and "X" and "Y", the "index registers". `LDA` loads data into the accumulator, `LDX` loads data into the X register, and `LDY` loads data into the Y register.

There are two main sources for this data: loading from a memory address and loading a given value. Which source is used is based on how you write the operand for the opcode. If you use a 16-bit value (four hex digits), it will load the contents of that memory address. If you use a hash sign (#) followed by an 8-bit value (two hex digits), it will load the exact value you used. Here is an example:

```
LDA $3f00  ; load contents of memory address $3f00
           ; into the accumulator
LDA #$3f   ; load the value $3f into the accumulator
```

These different operand formats are called *addressing modes*. The 6502 can use eleven different addressing modes (though most opcodes are limited to a subset of them). In case you're curious, the eleven addressing modes are Accumulator, Immediate, Implied, Relative, Absolute, Zeropage, Indirect, Absolute Indexed, Zeropage Indexed, Indexed Indirect, and Indirect Indexed.The two modes we've just seen are *absolute* mode (providing a memory address) and *immediate* mode (providing an exact value). We will learn more addressing modes as needed over the course of the book.

When your code is run through the assembler, the different addressing modes actually translate to different entries in the instruction set. `LDA $3f00` becomes `ad 00 3f`. In the 6502 instruction set, `ad` is the instruction set number of the "LDA absolute mode" instruction, and $3f00 is placed in little-endian order. `LDA #$3f`, though, becomes `a9 3f`. Here, `a9` is the instruction set number for "LDA immediate mode". The assembler is smart enough to insert the correct instruction set number based on how you write the operand, so you don't need to worry about *which* `LDA` instruction you need to use.

- Storing data: `STA, STX, STY`

The "ST" opcodes do the reverse of the "LD" opcodes - they store the contents of a register to a memory address. STA stores the contents of the accumulator to the location specified by its operand, STX stores the contents of the X register, and STY stores the contents of the Y register. The ST instructions cannot use immediate mode, since it doesn't make sense to store the contents of a register into a literal number. After a store operation, the register you stored from keeps its same value, allowing you to store the same register value in a different location right away.

- Transferring data: `TAX, TAY, TXA, TYA`

Finally, the "T" instructions transfer data from one register to another. These opcodes are all read as "transfer from register to register" - `TAX`, for example, is "transfer from accumulator to X register". The "transfer" in these instructions is more accurately described as a "copy", since after one of these instructions, both registers will have the value of the first register.

❖ A small example

Now that you've learned your first ten opcodes (and two addressing modes), let's look at a small example that makes use of them.

```
LDA #$a7
TAY
STY $3f00
```

What does this code do? Let's take it line-by-line:

1. First, we load the value `$a7` into the accumulator. `LDA` is the opcode that loads data into the accumulator. The `#` in front of `$a7` indicates that we want to use immediate mode, so this is a number to put directly into the accumulator, not a memory address.
2. Next, we copy the value `$a7` from the accumulator to the Y register. Now both A and Y have the same value.
3. Finally, we store the value of the Y register (`$a7`) at memory address `$3f00`.

■ *Back to the Test Project*

You now understand enough assembly to work out what most of the test project is doing. Let's look specifically at the main portion, reproduced here:

```
34  .proc main
35    LDX $2002
36    LDX #$3f
37    STX $2006
38    LDX #$00
39    STX $2006
40    LDA #$29
41    STA $2007
42    LDA #%00011110
43    STA $2001
44  forever:
45    JMP forever
46  .endproc
```

Most of this code is made up of loads and stores. The memory addresses that we load from or store to are `$2001, $2002, $2006`, and `$2007`. The immediate-mode values that we load are `$3f, $00, $29`, and `%00011110` (a binary value, unlike the hex values we use everywhere else).

Looking more closely at the immediate-mode loads, notice that the first two are `$3f00`, the address in PPU memory where palettes begin, followed by `$29`, the green color that we are using in the background. This code is telling the PPU to store `$29` at address `$3f00`, but how?

❖ Memory-Mapped I/O

On the NES, addresses in the `$2000-$6000` range are reserved for use as *memory-mapped I/O* (or "MMIO") addresses. "I/O" is "input/output" - sending data between different devices. "Memory-mapped" means that these interfaces to other devices are mapped to memory addresses - in other words, certain memory addresses are not memory at all, but rather connections to other devices.

Memory addresses in the low `$2000`s correspond to connections to the PPU. If you want to learn more about the PPU's MMIO addresses (or any other NES topic, for that matter), check out the NESDev Wiki. While it is not the best resource to learn from, it is an invaluable reference to the system, backed by meticulous research from its contributors. There are four MMIO addresses in use in our code; let's take a look at what each one does (along with the name each address is commonly known by).

❖ `$2006`: PPUADDR and `$2007`: PPUDATA

The NES CPU (where your code is being executed) doesn't have direct access to the PPU's memory. Instead, CPU memory address `$2006` lets your code select an address in PPU memory, and `$2007` lets your code write a byte of data to that address. To set the address you want to write to, store two bytes of data to `$2006` - first the "high" (left) byte, followed by the "low" (right) byte. Here's how our test project does that:

```
36      LDX #$3f
37      STX $2006
38      LDX #$00
39      STX $2006
```

This code first stores the byte $3f to $2006, then the byte $00 to $2006 - in other words, it sets the address for any following writes to PPU memory to `$3f00`, which is the address of the first color of the first palette.

To store data at the selected PPU memory address, store a byte to `$2007`:

```
40      LDA #$29
41      STA $2007
```

This writes the byte `$29` (which represents "green") to the memory address we selected before (`$3f00`). Each time you store a byte to PPUDATA, the memory address for the next store is incremented by one. If the next lines of code in the program were `LDA #$21` and `STA $2007`, the byte `$21` would be written to PPU address `$3f01` even though we did not do anything with PPUADDR.

❖ `$2002`: PPUSTATUS

PPUSTATUS is a read-only MMIO address. When you load from `$2002`, the resulting byte gives you information about what the PPU is currently doing. Reading from PPUSTATUS has one useful side-effect, as well: it resets the "address latch" for PPUADDR. It takes two writes to PPUADDR to fully specify a memory address, and it is possible that your code might make one write but never get around to doing the second write. Reading from PPUSTATUS makes it so that the next write to PPUADDR will always be considered a "high" byte of an address.

In our test project, we read ("load") from PPUSTATUS before attempting to write an address to PPUADDR:

```
35      LDX $2002
36      LDX #$3f
37      STX $2006
38      LDX #$00
39      STX $2006
40      LDA #$29
41      STA $2007
```

- `$2001`: PPUMASK

There's still one more thing our test project has to do after it tells the PPU to use color `$29` as the first color of the first palette - it has to tell the PPU to actually start drawing things to the screen! PPUMASK allows your code to give the PPU instructions about what to draw, as well as set some tweaks to how colors are displayed. The byte that you store to PPUMASK is a set of eight *bit flags*, where each bit in the byte acts as an on/off switch for a particular property. Here is what each bit does. (Remember, the bits that make up a byte are numbered 0-7, with bit 0 all the way on the right, and bit 7 all the way on the left.)

Bit #	Effect
0	Greyscale mode enable (0: normal color, 1: greyscale)
1	Left edge (8px) background enable (0: hide, 1: show)
2	Left edge (8px) foreground enable (0: hide, 1: show)
3	Background enable
4	Foreground enable
5	Emphasize red
6	Emphasize green
7	Emphasize blue

Before we look at the test project, a few notes on these options. Bits 1 and 2 enable or disable the display of graphical elements in the left-most eight pixels of the screen. Some games choose to disable these to help avoid flicker during scrolling, which we will learn more about later. Bits 5, 6, and 7 allow your code to "emphasize" certain colors - making one of red, green, or blue brighter and making the other two colors darker. Using one of the emphasis bits essentially applies a color tint to the screen. Using all three at the same time makes the entire screen *darker*, which many games use as a way to create a transition from one area to another.

Let's look again at our test project's code. What value does our test project write to PPUMASK?

```
42      LDA #%00011110
43      STA $2001
```

Here is what options we are setting, bit-by-bit:

Bit #	Value	Effect
0	0	Greyscale mode disabled
1	1	Show leftmost 8px of background
2	1	Show leftmost 8px of foreground
3	1	Background enabled
4	1	Foreground enabled
5	0	No red emphasis
6	0	No green emphasis
7	0	No blue emphasis

Since our test project turns on background rendering in its write to PPUMASK, our green background shows up after this line.

■ *Wrapping Up the main Code*

Our test project has written a color to PPU memory and turned on display rendering, so it's all done, right?

Not so fast. Remember, the CPU fetches and executes instructions one at a time, continuously. Unless we write code to keep the processor busy, it will continue reading (empty) memory and "executing" it, which could lead to disastrous results. Thankfully, there's an easy solution to this problem:

```
44    forever:
45      JMP forever
```

We will learn the details of how this works later, but essentially these two lines of code create a *label* (`forever`), and then tell the CPU to fetch that label as the next instruction to execute. The CPU goes into an infinite loop, and your project happily does nothing but display its green background.

We've now covered everything in `main` (though we still haven't talked about what `.proc` and `.endproc` are doing), but there's much more in the test project to discuss. Next, we will learn about interrupt vectors and how our code initializes the NES when it first powers on.

■ *Homework*

Now that you know how the test project sets a background color, try modifying it to display a different color. I've included the color chart from Chapter 4 below. Don't forget to re-assemble and re-link your code before opening it in Mesen2.

The NES color palette.

6.Headers and Interrupt Vectors

Last chapter, we covered the "main" section of the test project, which sets the background color and then goes into an infinite loop. Yet that code only accounts for 13 lines out of the 44-line test project source code. In this chapter, we'll explore the remainder of the test project's source code and learn a few more opcodes.

■ *iNES Headers*

At the very beginning of the test project, we see this curious bit of assembly:

```
1   .segment "HEADER"
2   .byte $4e, $45, $53, $1a, $02, $01, $00, $00
```

`.segment`, since it begins with a period, is an assembler directive - an instruction for the assembler to follow when it is converting the code to machine code. `.segment`, as we discussed in Chapter 4, tells the assembler where to put your code in the final output. The `HEADER` segment, unsurprisingly, is placed at the very beginning of the resulting .nes file.

The second line uses another assembler directive that we've seen before. `.byte` tells the assembler to insert literal data bytes into the output, rather than trying to interpret opcodes from them. The three ASCII codepoints `$4e, $45, $53` are the string "NES" (the ASCII representations of "N", "E", and "S"). The next byte, `$1a`, represents the MS-DOS "end of file" character. These four bytes are a "magic number" that marks the output as an NES game.Most file types have a "magic number" at the start of the file to make it easier for an operating system to know what a file is with certainty. Java bytecode files begin with `$cafebabe`, PDF files begin with `$25504446` ("%PDF"), and zip files begin with `$504b` ("PK", since the format used to be called "PKZIP"). You can find a larger list of file type magic numbers on Wikipedia.

The "NES" magic number header specifically identifies the file as an "iNES" NES game. iNES was one of the first NES emulators, and the first to achieve widespread popularity. iNES' author, Marat Fayzullin, created a special header format that provides the emulator with information about the game itself, such as which region (NTSC/PAL) the game is from, how many PRG and CHR ROM banks it has, and more. The full iNES header is 16 bytes long.

Our test project's header, after "NES" and `$1a`, specifies that the "cartridge" for our game contains two 16KB PRG-ROM banks (32KB storage) and one 8KB CHR-ROM bank, and that it uses "mapper zero". NES cartridges come in many (hundreds) of variations, and the iNES header assigns each variant a number. Mapper zero, as you might guess, represents the simplest kind of NES cartridge, generally called "NROM". Some of the earliest NES games used NROM cartridges, including *Balloon Fight*, *Donkey Kong*, and *Super Mario Bros.*. Due to their relative simplicity, we'll be creating NROM cartridge games for most of this book. For more details on the iNES header format, see the article on the NESDev Wiki. Many of the things that can be set through the iNES header, like mirroring modes and PRG-RAM, will be discussed in greater detail later in this book.

■ *Isolating Procedures with .proc*

After the header, another `.segment` directive switches us to the "CODE" segment, which begins at 16 bytes into the file (i.e., after the 16-byte header). The first thing within the segment is a new assembler directive, `.proc`. This directive lets you create new lexical scopes in your code. Basically, this means that the labels you make inside of a `.proc` are unique to that proc. Here's an example:

```
.proc foo
  LDA $2002
some_label:
  LDA #$3f
.endproc

.proc bar
  LDA #$29
some_label:
  STA $2007
.endproc
```

Here, we use the label `some_label` twice, once in `.proc foo` and once in `.proc bar`. However, since they are inside of different procs, the two labels are treated as totally separate. Using `some_label` inside of `bar` will refer to `bar`'s version, and there is no way to access `foo`'s version of the label from inside of `bar`. Generally, we will wrap independent pieces of code in their own procs so they can use the same label names without clobbering each other. This might seem like a lot of effort just to be able to use the same label name in more than one place, which is true to an extent. The real power of using `.proc` comes when your code is composed of multiple files - some of which you might not have written yourself! Procs let you safely use a label without having to check through all of the code to see if you (or another developer) have used the same label before.

The test project uses four procs - `irq_handler`, `nmi_handler`, `reset_handler`, and `main`. We covered the `main` proc in the last chapter, but what are the other procs doing?

■ *Interrupt Vectors*

As discussed earlier, the processor in the NES repeatedly fetches and executes bytes in sequence. When certain events happen, though, we want to interrupt the processor and tell it to do something else. The events that can cause these interruptions are called *interrupt vectors*, and the NES/6502 has three of them:

- The *reset vector* occurs when the system is first turned on, or when the user presses the Reset button on the front of the console.
- The *NMI vector* ("Non-Maskable Interrupt") occurs when the PPU starts preparing the next frame of graphics, 60 times per second.

- The *IRQ vector* ("Interrupt Request") can be triggered by the NES' sound processor or from certain types of cartridge hardware.

When an interrupt is triggered, the processor stops whatever it is doing and executes the code specified as the "handler" for that interrupt. A handler is just a collection of assembly code that ends with a new opcode: RTI, for "Return from Interrupt". Since the test project doesn't need to make use of NMI or IRQ handlers, they consist of just RTI:

```
5   .proc irq_handler
6     RTI
7   .endproc
8
9   .proc nmi_handler
10    RTI
11  .endproc
```

`RTI` marks the end of an interrupt handler, but how does the processor know where the handler for a given interrupt begins? The processor looks to the last six bytes of memory - addresses `$fffa` to `$ffff` - to find the memory address of where each handler begins.

Memory address	Use
`$fffa-$fffb`	Start of NMI handler
`$fffc-$fffd`	Start of reset handler
`$fffe-$ffff`	Start of IRQ handler

Because these six bytes of memory are so important, ca65 has a specific segment type for them: `.segment "VECTORS"`. The most common way to use this segment is to give it a list of three labels, which ca65 will convert to addresses when assembling your code. Here is what our test project's "VECTORS" segment looks like:

```
39  .segment "VECTORS"
40  .addr nmi_handler, reset_handler, irq_handler
```

.addr is a new assembler directive. Given a label, it outputs the memory address that corresponds to that label. So, these two lines of assembly set bytes $fffa to $ffff of memory to the addresses of the NMI handler, reset handler, and IRQ handler - exactly the same order as in the table above. Each label on line 40 is the start of the .proc for that handler.When the NES first turns on, rather than starting from memory address $0000, the 2A03 follows a specific series of steps. It fetches the memory address stored in $fffc and $fffd (the address of the start of the reset handler). It places that address into the program counter, which makes the start of the reset handler the next instruction to be executed. Then it works its way through the reset handler, instruction by instruction.

❖ The Reset Handler

While the test project doesn't make use of the NMI or IRQ events, it does need a reset handler. The reset handler's job is to set up the system when it is first turned on, and to put it back to that just-turned-on state when the user hits the reset button. Here is the test project's reset handler:

```
13  .proc reset_handler
14    SEI
15    CLD
16    LDX #$40
17    STX $4017
18    LDX #$FF
19    TXS
20    INX
21    STX $2000
22    STX $2001
23    STX $4010
24    BIT $2002
25  vblankwait:
26    BIT $2002
27    BPL vblankwait
28  vblankwait2:
29    BIT $2002
30    BPL vblankwait2
31    JMP main
32  .endproc
```

A few things to note about this section of code. First, unlike the other interrupt handlers, it does not end in `RTI` - that's because when the system is first turned on, the processor wasn't in the middle of doing anything else, so there is nowhere to "return" to. Instead, it ends with `JMP main`. We saw `JMP` at the end of `main` last chapter, where `JMP forever` created our infinite loop. `JMP` stands for "jump", and it tells the processor to go somewhere else to fetch its next instruction. The operand for `JMP` is a full, two-byte memory address, but it is nearly always used with a label that the assembler will convert to a memory address at assemble time. `JMP main`, here, tells the processor to start executing the code in `main` once it is done with the reset handler.

Second, this code features several opcodes we haven't seen before. Let's learn about them by analyzing the reset handler line-by-line.

Lines 14 and 15 feature two opcodes that are, generally, only found in reset handlers. `SEI` is "Set Interrupt ignore bit". After an `SEI`, anything that would trigger an IRQ event does nothing instead. Our reset handler calls `SEI` before doing anything else because we don't want our code to jump to the IRQ handler before it has finished initializing the system. `CLD` stands for "Clear Decimal mode bit", disabling "binary-coded decimal" mode on the 6502.

Due to convoluted licensing issues surrounding the 6502 and Ricoh's legal ability to manufacture it, the 2A03 used in the NES has binary-coded decimal mode circuitry within it, but all electrical traces that would connect those circuits to the rest of the chip have been cut. `CLD` (and its counterpart, `SED`) do nothing on the NES as a result, but calling `CLD` as part of a reset handler just-in-case is considered best practice.

The next four lines disable audio IRQs (which are handled separately from the `SEI` we used earlier) and set up the "stack". I'm not going to go into detail on these yet; this reset code is something that you will likely copy verbatim into each project, so it's not something you will need to change on your own. I'll cover both of these much later.

The next three lines (after `INX`) go back to familiar loads and stores. We've seen `$2001` before - it's PPUMASK - but `$2000` is new. This address is commonly referred to as PPUCTRL, and it changes the operation of the PPU in ways more complicated than PPUMASK's ability to turn rendering on or off. We won't cover PPUCTRL in detail until later in this book, when we've seen more of how the NES PPU draws graphics. Like PPUMASK, it is a set of bit fields. For the purpose of initializing the NES, the main thing to point out is that bit 7 controls whether or not the PPU will trigger an NMI every frame. By storing `$00` to both PPUCTRL and PPUMASK, we turn off NMIs and disable rendering to the screen during startup, to ensure that we don't draw random garbage to the screen.

The remainder of the reset handler is a loop that waits for the PPU to fully boot up before moving on to our main code. The PPU takes about 30,000 CPU cycles to become stable from first powering on, so this code repeatedly fetches the PPU's status from PPUSTATUS (`$2002`) until it reports that it is ready. 30,000 cycles sounds like a long time, but the NES' 2A03 processor runs at 1.78 MHz, so 30,000 cycles is a tiny, tiny fraction of a second. I'm not going to cover `BIT` or `BPL` here, but rest assured that we will come back to them later.

Finally, with all of our setup complete, we jump to `.proc main` and execute our game's actual code.

❖ A Full Reset

The reset handler our test project uses is the most basic handler that will reliably start up the NES in a known-good state. There are many other tasks that the reset handler can take care of, like setting up the Audio Processing Unit (APU) and clearing out RAM. As we add more functionality to our games, we will expand the role of the reset handler as well.

We have now covered all of the code from the test project - congratulations! After the last few chapters, you might be thinking to yourself "this is incredibly complicated, why bother?" The next chapter is for you!

7.Why Would Anyone Do This?

We've spent six chapters together talking about how to get an over-30-year-old game system to display a static, green background. (Thank you, by the way, for sticking with me this long!) This is, to be fair, an incredible amount of work for such an unimpressive result. At this point, you may be wondering why anyone would bother going through so much work. Before we move on, I'd like to take the opportunity to remind you of why it's worth investing the time to develop for the Nintendo Entertainment System.

■ *It's a Classic*

First, the NES is an absolute classic. In both the US and Japan, it dominated the home video game market for most of the 1980's. In the aftermath of the 1983 "Atari crash", it single-handedly revitalized the US home video game market. An entire generation of gamers have fond memories of the system, and even young people with no prior experience with the system recognize its historical importance.

https://youtu.be/bDOZbvE01Fk

> I can't imagine a world without home video games. That's like saying TV wouldn't work, or a movie wouldn't work. ... *It's a classic.* Like a classic vinyl record. This [NES] is a classic vinyl video game.
>
> Ethan James (age 19), "Teens React to Nintendo (NES)"

■ *It's Good for Individuals and Small Teams*

Most commercial NES games were made by small teams. Consider *Donkey Kong*, the game that the NES was designed to play:

Role	Name
Programmer	Toshihiko Nakago
Course Designer	Kenta Usui
Original Game Designer	Shigeru Miyamoto
Music & Sound Effects	Yukio Taneoka
Producer	Masayuki Uemura

The entire Famicom game was made by a team of five people, only one of which was a "Programmer". Many of the best games for the NES were made by similarly small teams, like *Super Mario Bros.*:

Role	Name
Producer/Director/Designer	Shigeru Miyamoto
Assitant Director/Designer	Takashi Tezuka
Sound & Music	Koji Kondo
Programmer	Toshihiko Nakago
Programmer	Kazuaki Morita

Super Mario Bros. is quite likely the most iconic NES game of all time, and it, like *Donkey Kong*, was produced by a team of five people, two of whom were programmers.

■ *It's Powerful Enough to Be Interesting*

Compared to its predecessor in the home video game market, the Atari 2600, the NES offers a much greater range of expression to the developer. The Atari 2600 was designed to play a game called *Combat*:

https://youtu.be/3m86ftny1uY

Combat features no music, only sound effects, and a limited set of graphics to convey the "playfield" where the action takes place. The 2600 gave developers access to one background layer, two "sprite" objects (in this case, the tanks), and two "ball" objects (the bullets fired by the tanks). Any other graphics had to come from the player's own imagination.

Donkey Kong, the game the NES was designed to play, is a far different experience:

https://youtu.be/C_PrG8P5W8o

There is simple, repetitive music in addition to the sound effects of Mario walking and jumping over barrels. Multiple moving objects are on screen at one time (barrels, the flaming oil drum, the animated Donkey Kong at the top of the screen). Even in their use of text, the two games differ greatly. *Combat* displays only a single large score number for each player at the top; *Donkey Kong* shows the player's current score (using six digits), the current high score, the number of lives the player has remaining, and how many "bonus" points the player is able to earn by completing the stage.

In total, these changes allow the NES to display much more information on screen at one time, giving developers the freedom to create far more detailed and nuanced games than what was possible on the Atari 2600.

■ *It's Simple Enough to Be Approachable*

While the NES was a major step up from the capabilities of the Atari 2600, it pales in comparison to its own successor, the Super Famicom / Super NES. Here is a brief list of differences between the two consoles:

	NES	SNES
CPU	8-bit MOS 6502 derivative, 1.79 MHz	16-bit MOS 65c816 derivative, 3.58 MHz
Addressable memory	64KB	16MB
RAM	2KB	128KB
Graphics resolution	256x240	up to 512x478
Available colors	64	32,768
Background layers	1, up to 512x512	4, each up to 1024x1024
Audio output	5 fixed channels	8 fully-programmable channels
Largest released game	1MB	6MB

All of this additional power comes at a cost. The Super NES is much more difficult to program compared to the more limited NES. Having more space to store graphical assets means that designers are expected to produce far more output. The amazing audio processor on the Super NES requires a composer who is well-versed in its intricacies, since developers have to provide it with sound samples rather than using a built-in set of instruments. Super NES development is much less feasible as a solo developer or small team. Just compare the staff rolls of the NES games above to the team that created *Super Mario World*, the game the Super NES was designed to play:

Role	Name
Producer	Shigeru Miyamoto
Director	Takashi Tezuka
Sound Composer	Koji Kondo
Map Director	Hideki Konno
Area Director	Katsuya Eguchi
Programming Director	Toshihiko Nakago
Mario/System Programmer	Toshio Iwawaki
Object Programmer	Kazuaki Morita
Background Programmer	Shigehiro Kasamatsu
Background Programmer	Tatsunori Takakura
Map Programmer	Tatsuo Nishiyama
Area Data Input	Yoshihiro Nomoto
Area Data Input	Eiji Noto
Area Data Input	Satoru Takahata
Character Graphic Design	Shigefumi Hino

■ *People Still Make NES Games Today*

Nintendo ceased production of the NES in 1995, ten years after it was first released in the US. The last officially-licensed NES game released in North America was 1994's *Wario's Woods*. In the years that followed, as emulation (and therefore understanding) of the NES increased in quality, many independently-produced games were released for the system, often created by solo developers or very small teams. Here are just a few "homebrew" NES games that show what the system is capable of.

❖ ***Battle Kid: Fortress of Peril***, Sivak Games, 2010

https://youtu.be/yne04hukuyc

Produced by a solo developer, *Battle Kid: Fortress of Peril* helped popularize the idea of homebrew NES development.

❖ ***Kira Kira Star Night DX***, RIKI, 2013

https://youtu.be/h8kj7kytJp4

A Famicom exclusive, *Kira Kira Star Night DX* is essentially a great chiptune album with a beautiful (if simple) game attached. Its initial cartridge print run sold out in a single day.

❖ ***Twin Dragons***, Broke Studio, 2018

https://youtu.be/BOn6hEZuolU

Twin Dragons was a Kickstarter project by French developers Broke Studio. It raised over €30,000, easily meeting its campaign funding goals.

❖ ***Lizard***, Brad Smith, 2018

https://youtu.be/5RDAoN_qO9w

Lizard is a new NES game released both on a physical NES cartridge and digitally over Steam (wrapped in an NES emulator).

❖ ***Micro Mages, Morphcat Games***, 2019

https://youtu.be/VFX401vvKTQ

Micro Mages is a new platform game for the NES, play solo or with up to 4 players simultaneously. Fight to get the best score or cooperate to overcome challenging levels.

■ *It Has a Vibrant Community*

Thankfully, the developers who continue to create new NES games to this day are not isolated hermits. They are part of a broad community of hobbyists and NES players who continue to expand our knowledge of how the NES operates and how best to create games for it. The community has put together enormous amounts of reference documentation at the NESDev Wiki, and the NESDev Forums are a great place to get help or to see some of the latest techniques homebrew developers are using.

■ *...Continue?*

If, over the last few chapters, you were a bit frightened about what you were getting into, I hope that this chapter has re-invigorated you. The NES is a strong platform for development that sees new releases and new insight frequently, and I can't wait to continue introducing you to how it works. Let's keep going!

8.Refactoring

Before we move on to learning more about how the NES draws graphics, let's take some time to reflect on what we've already built. There are a number of improvements we can make now that will be useful in the future. By refactoring now, we will create a useful template for our upcoming projects.

■ *Constants*

There are several places in our code where we use a particular number that doesn't change, such as MMIO addresses for talking with the PPU. It's hard to tell what these numbers are referring to when looking at the code.

Thankfully, we can replace these abstract numbers with descriptive text, by declaring *constants*. A constant is essentially a name for one number, which can't be changed. Let's create constants for the PPU addresses we've used so far:

```
PPUCTRL   = $2000
PPUMASK   = $2001
PPUSTATUS = $2002
PPUADDR   = $2006
PPUDATA   = $2007
```

With these constants, our main code becomes much more readable:

```
34  .proc main
35    LDX PPUSTATUS
36    LDX #$3f
37    STX PPUADDR
38    LDX #$00
39    STX PPUADDR
40    LDA #$29
41    STA PPUDATA
42    LDA #%00011110
43    STA PPUMASK
44  forever:
45    JMP forever
46  .endproc
```

Where do we put these constants? The common approach is to make a separate constants file, which can be included into our main assembly file. We'll call the constants file `constants.inc`. Why does this file end in `.inc` instead of `.asm`? The constants file is not exactly assembly code; it doesn't have any opcodes. We will use the `.asm` extension for assembly code files, and `.inc` for files which are included into an assembly code file. Then, we include the constants file at the top of our `.asm` file like this:

```
.include "constants.inc"
```

■ *Header File*

We can do the same thing with the `.header` segment, since it will generally be the same from project to project. Let's make a `header.inc` file to hold our header content. Now would also be a good time to add some comments:

```
.segment "HEADER"
.byte $4e, $45, $53, $1a ; Magic string that always begins an
                           iNES header
.byte $02        ; Number of 16KB PRG-ROM banks
.byte $01        ; Number of 8KB CHR-ROM banks
.byte %00000001  ; Vertical mirroring, no save RAM, no mapper
.byte %00000000  ; No special-case flags set, no mapper
.byte $00        ; No PRG-RAM present
.byte $00        ; NTSC format
```

Now we can delete the `.segment "HEADER"` section of our main `.asm` file, and include our new header file. The top of our `.asm` file should now look like this:

```
.include "constants.inc"
.include "header.inc"

.segment "CODE"
```

When the assembler and linker run, they will take the contents of `header.inc` and put them in the correct place in the output ROM, exactly the same as if we had put it directly into the assembly file.

■ *ca65 Imports and Exports*

A full reset handler can become quite large, so it can be useful to put it into a separate file. But we can't just `.include` it, because we need a way to reference the reset handler in the `VECTORS` segment.

The solution is to use ca65's ability to import and export `.proc` code. We use the `.export` directive to inform the assembler that a certain proc should be available in other files, and the `.import` directive to use the proc somewhere else.

First, let's create `reset.asm`, including the `.export` directive:

```
.include "constants.inc"

.segment "CODE"
.import main
.export reset_handler
.proc reset_handler
  SEI
  CLD
  LDX #$40
  STX $4017
  LDX #$FF
  TXS
  INX
  STX $2000
  STX $2001
  STX $4010
  BIT $2002
vblankwait:
  BIT $2002
  BPL vblankwait
vblankwait2:
  BIT $2002
  BPL vblankwait2
  JMP main
.endproc
```

There are a few things I'd like to point out in this file. First, the file ends in `.asm`, because it contains opcodes. Second, we include the constants file so that it can be used here. Third, we need to specify which code segment this `.proc` belongs in, so the linker knows how to put everything together. Finally, note that we are importing `main`. This way, the assembler knows what memory address the `main` proc is located at, so the reset handler can jump to the correct address.

Now that we have a separate reset file, we'll use `reset_handler` inside our code:

```
4   .segment "CODE"
5   .proc irq_handler
6     RTI
7   .endproc
8
9   .proc nmi_handler
10    RTI
11  .endproc
12
13  .import reset_handler
14
15  .export main
16  .proc main
17    ; contents of main here
18  .endproc
19
20  .segment "VECTORS"
21  .addr nmi_handler, reset_handler, irq_handler
```

On line 13, where our `.proc reset_handler` used to be located, we now import the proc from an external file. Note that you do not need to specify which file the proc comes from - the assembler scans all `.asm` files for exports before it starts assembling, so it already knows what external procs are available and where they are located. (Note that this also means you can't export two procs with the same name - the assembler will have no way to tell which one you are referring to in an `.import`.) You may have noticed that `reset.asm` uses `.segment "CODE"`, and our main assembly file also uses `.segment "CODE"`. What happens when we assemble and link these files? The linker finds everything that belongs to the same segment and puts it together. The order does not particularly matter, since labels are converted into addresses at link time. We also have to export our `main` proc, so that the reset handler can import it and know where to jump to when it is finished.

■ *Custom Linker Configuration*

When we linked our sample project back in Chapter 3, we used this command:

```
ld65 helloworld.o -t nes -o helloworld.nes
```

The `-t nes` tells ld65 to use the default linker configuration for the NES. This is why we have the "STARTUP" segment, despite never using it. While the default configuration works for the sample project, it can lead to problems as our code becomes larger and more complicated. So, instead of using the default configuration, we will write our own linker configuration with only the segments and features that we need.

Our custom linker config will be in a file called `nes.cfg`, which will look like this:

```
MEMORY {
  HEADER: start=$00, size=$10, fill=yes, fillval=$00;
  ZEROPAGE: start=$10, size=$ff;
  STACK: start=$0100, size=$0100;
  OAMBUFFER: start=$0200, size=$0100;
  RAM: start=$0300, size=$0500;
  ROM: start=$8000, size=$8000, fill=yes, fillval=$ff;
  CHRROM: start=$0000, size=$2000;
}

SEGMENTS {
  HEADER: load=HEADER, type=ro, align=$10;
  ZEROPAGE: load=ZEROPAGE, type=zp;
  STACK: load=STACK, type=bss, optional=yes;
  OAM: load=OAMBUFFER, type=bss, optional=yes;
  BSS: load=RAM, type=bss, optional=yes;
  DMC: load=ROM, type=ro, align=64, optional=yes;
  CODE: load=ROM, type=ro, align=$0100;
  RODATA: load=ROM, type=ro, align=$0100;
  VECTORS: load=ROM, type=ro, start=$FFFA;
  CHR: load=CHRROM, type=ro, align=16, optional=yes;
}
```

The `MEMORY` section lays out the regions of memory that segments can be placed into, while the `SEGMENTS` section describes the segment names we can use in our code and which memory areas they should be linked into. I won't be going into detail on what each setting means, but you can find in-depth documentation in the ld65 docs.

To use this custom linker configuration, we first need to update the segment names in our code to match the config file's segment names. In our case, the only needed changes are moving `"CHARS"` to `"CHR"` and removing `"STARTUP"`.

■ *Putting It All Together*

Finally, we need to update the structure of our files a bit. We will move all of the `.asm` and `.inc` files into a sub-directory, `src`, with our new linker config at the top level. The code we have after all of our refactoring should now look like this:

```
08-refactoring
   |
   |-- nes.cfg
   |-- src
      |
      |-- constants.inc
      |-- header.inc
      |-- helloworld.asm
      |-- reset.asm
```

To assemble and link our code, we will use the following commands (run from the top-level `08-refactoring` directory):

```
ca65 src/helloworld.asm
ca65 src/reset.asm
ld65 src/reset.o src/helloworld.o -C nes.cfg -o helloworld.nes
```

To be clear, what we are doing here is first assembling each `.asm` file to create `.o` files. Once that is done, we pass all of the .o files to the linker. Instead of using the default NES linker config (`-t nes`), we use our new custom config (`-C nes.cfg`). The output from the linker is placed into the same `helloworld.nes` ROM file.

If you would like to download a copy of the files listed above, here is a ZIP file (https://famicom.party/book/projects/08-refactoring.zip) of everything so far. We'll be using this setup as a base for our future projects, so be sure that you are able to assemble, link, and run the code before moving on.

Part II: Graphics

9. The PPU

A "game" for the NES is made up of three components: graphics displayed on a screen, user input through some kind of controller, and audio for music and sound effects. The game uses the user's input to change the graphics it displays and the audio it plays, until the user turns off the system. In this set of chapters, we will look at each of these three components, beginning with how the NES displays graphics.

■ *Palettes*

As you may remember from Chapter 4, the NES uses a fixed set of 64 colors for all of its graphics.

The NES color palette.

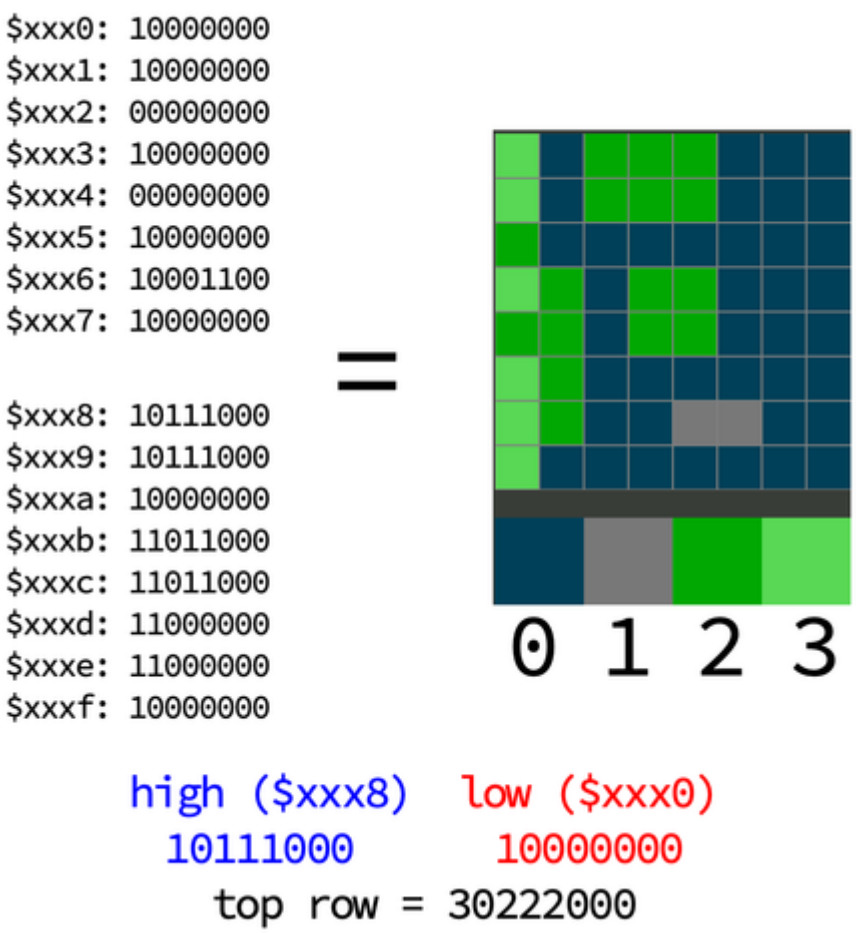
These colors are used to fill slots in eight four-color *palettes*. Four palettes are used to draw background objects, and the other four palettes are used to draw *sprites*, objects in the "foreground". Each thing drawn to the screen uses one of these palettes, limiting a single graphical object to four of the 64 available colors.

■ *Pattern Tables*

What exactly are these "graphical objects"? The NES does not let developers specify what to draw on a pixel-by-pixel basis. At a resolution of 256x240 pixels, each screen of graphics would require the specification of 61,440 pixels of color information, which would be far too much to fit into memory. Instead, the basic unit of NES graphics is the 8x8 pixel "tile". One screen of graphics is 32 tiles wide and 30 tiles tall (960 tiles).

The CHR-ROM in an NES cartridge holds two *pattern tables*, each of which holds 256 8x8 tiles. One pattern table is used for background graphics, and the other is used for sprite graphics. Each tile in the table is defined with two "bit planes", specifying which palette color (0-3) is used for each pixel of the tile. One bit plane defines the "low bit" of each pixel in the tile, and the other defines the "high bit". (Two bits, as you may recall, can represent four different values, corresponding to the four colors in a palette.) Each tile takes up 16 bytes of memory, so the CHR-ROM chip's 8KB of storage is just enough to fit the 512 tiles of the two pattern tables. By specifying only a palette index rather than an actual color, the tiles themselves take up less memory and can be re-used with different palettes as needed.

Another option is to use 8x16 tiles (8px wide and 16px tall), as seen in games like *The Legend of Zelda*. While this reduces the number of tiles that can be stored in the pattern tables, this has the advantage of allowing tiles from either table to be used as either background or sprite. Depending on your game's graphical needs, the advantage of being able to re-use tiles in both layers might outweigh the disadvantage of having fewer tiles available. These issues largely become moot with the use of mapper chips that allow for bank switching of pattern table graphics, which will be explored much later in this book.



An example pattern table tile. Bytes `$xxx0-$xxx7` provide the "low bit" for each pixel, and bytes `$xxx8-$xxxf` provide the "high bit" for each pixel.

Everything that an NES game draws to the screen is contained in its pattern tables. This is more complicated when CHR-RAM or bank switching are involved, though anything drawn is still, technically, present in the pattern tables at some point.There is no "system font"; if you want to draw text in your NES game, you need to create font tiles in a pattern table. (This is why most NES games, especially early NES games, tend to be all-caps, shouty affairs.) The limited space for tiles also means that efficient re-use of tiles is important. Being able to re-use a tile in multiple ways, or making clever use of palette swaps, can give a game a greater range of visual representations while still fitting within the memory limitations of CHR-ROM.

The sprite and background tables from *Super Mario Bros.*. Anything drawn during the course of the game is present in these two tables, including Mario himself (represented in the first five rows of the sprite table); all of the game's enemies; text and score graphics (the first three rows of the background table); and the many pipes, plants, and castles that appear throughout.

■ *Sprites*

Sprites represent the "foreground" layer of graphics. Each sprite is a single tile which can be positioned anywhere on the screen, down to the pixel. Sprites can also be flipped vertically or horizontally (but not rotated), and each sprite can specify which of the four sprite palettes it will use. This flexibility comes at a cost, though: memory and processing time constraints mean that the NES can only display 64 sprites at a time, and only eight sprites can be drawn on a scanline (a horizontal row of pixels).

If you've played many NES games, you have likely experienced "flickering", where some sprites appear and disappear rapidly in a way that does not seem intentional.

https://famicom.party/_app/immutable/assets/megaman2-flicker.YnEZPewF.mp4

An example of sprite flicker from *Mega Man II*.

This flicker is the result of too many sprites being drawn on one scanline. Since the PPU can only draw eight sprites on a scanline, any sprites beyond the first eight will not appear. The flickering effect, however, is a conscious choice on the part of the game developer, to help the player. Flicker results when the developer changes *which* eight sprites come first each scanline. In doing so, the player can see all of the sprites on the scanline, just not all at the same time. Otherwise, enemy sprites might be entirely invisible, which would be unfair.

When sprites are drawn to the screen, any pixel within the tile that uses the first color of the palette (index zero) is drawn as transparent, allowing the background layer to display at that location. This means that each sprite can only use three colors. By using transparency, it is possible to overlap sprites that use different color palettes over one another, creating graphics that use more than three colors.

Overlapping sprites used to create multicolor graphics. Mega Man's face is missing because its sprite is not being drawn due to scanline restrictions. Note that the missing area is not perfectly square, since transparency is being used to make the face and helmet sprites blend together.

■ *Backgrounds*

Sprites have great flexibility at the expense of only being able to cover a small portion of the screen. Backgrounds have the opposite trade-off. A background can cover the entire screen — 960 8x8 tiles — but background tiles must fit to a grid, and they suffer further limitations on palette use. The background layer can be scrolled in 1-pixel increments, but all the tiles move together. There is no way to scroll different parts of the screen differently without using tricky mid-frame updates (e.g. "Sprite Zero Hit", scanline IRQ).

❖ Nametables

Backgrounds are defined via *nametables*, which live in PPU memory. Each nametable is 960 bytes, and each of those bytes stores the tile number of one of the 256 tiles in the background pattern table. The PPU memory map has space for four nametables arranged in a square pattern, meaning that, in theory, you could set up four TV screens worth of background at once.

Screen 1 ($2000)	*Screen 2 ($2400)*
Screen 3 ($2800)	*Screen 4 ($2C00)*

The four nametables, showing the starting PPU memory address for each.

I say *in theory* because the Famicom was designed to be cheap, as we discussed in Chapter 1, and at the time, memory was very expensive. As a compromise, the Famicom/NES has enough physical memory for *two* nametables. These are "real" nametables that behave as expected. The memory ranges for the other two nametables act as "mirrors" of the real nametables, so that asking for a byte of memory from a mirror returns a byte from the corresponding real nametable. The developer can configure which two nametables are "real" and which two are "mirrored". On a hardware NES cartridge, this is done with a pad of solder over one of two contacts on the cartridge board. For emulators, a game's mirroring setting is part of its iNES header.

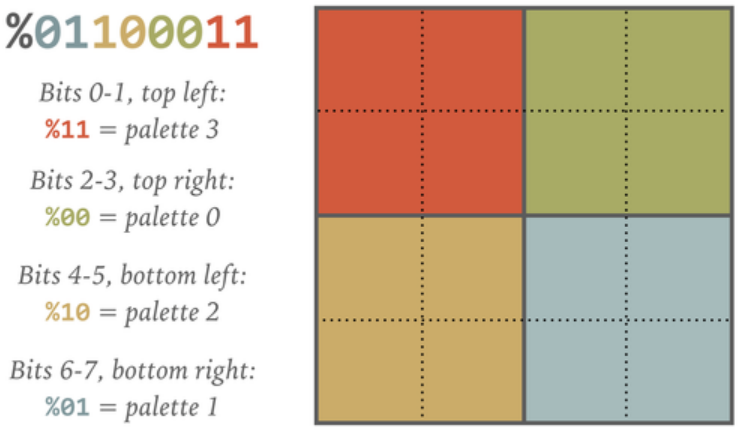
The internals of a *Balloon Fight* cartridge. The red rectangle shows the "V"/"H" contacts; whichever pair of contacts is soldered together will determine whether the game uses vertical or horizontal mirroring. Image from NES Cart Database.

Mirroring can be *vertical* or *horizontal*. In vertical mirroring, nametables 1 and 2 are "real", and 3 and 4 are mirrors. This gives the developer two screens in a left-to-right layout, perfect for horizontally-scrolling games. Horizontal mirroring, in contrast, makes nametables 1 and 3 the "real" nametables, and nametables 2 and 4 the mirrors. Horizontal mirroring results in two screens in a top-to-bottom layout, which is designed for vertically-scrolling games.While mirroring is hard-soldered in older NES games, later cartridges that add mapper chips give the developer the ability to change mirroring at any time. The MMC1 chip, for example, allows *Metroid* to switch between vertical and horizontal mirroring when the player moves through a doorway, allowing for a mix of horizontal and vertical scrolling sections.

❖ Attribute Tables

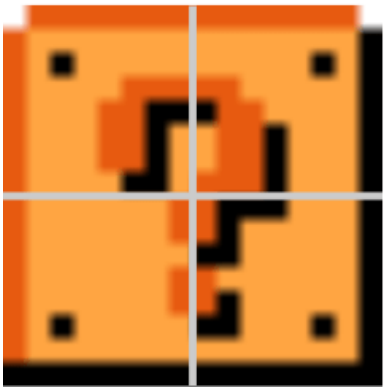
A nametable is just a list of tile numbers. In order to color each tile with a palette, we need a second type of table. At the end of each nametable is a 64-byte region called an *attribute table*, which specifies which palette to use for each tile of background. 960 + 64 = 1024 bytes, so each nametable / attribute table pair takes one kilobyte of memory.

Since the attribute table is only 64 bytes, there isn't enough space to specify a palette for each individual background tile. Instead, each byte of the attribute table specifies the palette colors for sixteen background tiles, in a four-by-four square. Bits 0-1 of each byte specify the palette used for the top left two-by-two portion of the four-by-four area, bits 2-3 the top right, bits 4-5 cover the bottom left and bits 6-7 select a palette for the bottom right. This means that in addition to background tiles being fixed to a grid, color information is tied to its own grid as well.



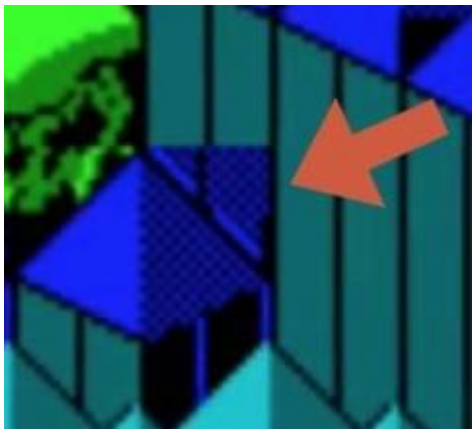
An illustration of how each byte of the attribute table determines palettes for sixteen background tiles.

As a consequence of attribute table limitations, background objects are generally drawn in 2x2 tile units. We call these larger objects *metatiles*.



The question mark (?) block from *Super Mario Bros.*, an example of a metatile.

The attribute table grid is also the reason why few NES games use an "isometric", or angled, display. Trying to draw backgrounds at an angle can cause color glitches when a large background section crosses attribute table boundaries.



Backgrounds from *Snake, Rattle 'n Roll* (1990). Attribute table boundaries result in one tile of the light blue "wall" appearing dark blue, since it is part of the same 2x2 palette block as the dark blue triangle.

10. Sprite Graphics

Now that we have talked about the PPU at a high level, we are ready to dive into the details of drawing sprites. By the end of this chapter, you will know how to create sprite tiles and draw them to the screen.

■ *Sprite Data*

Internally, the PPU uses 256 bytes of memory to store sprite information. It takes four bytes of data to describe a sprite, which is where the limit of 64 sprites at a time comes from. Those four bytes of data encode the following information:

1. Y position of the top left corner of the sprite (0-255)
2. Tile number from the sprite pattern table (0-255)
3. Special attribute flags (horizontal/vertical flipping, palette number, etc.)
4. X position of the top left corner of the sprite (0-255)

The third byte (attribute flags) uses its eight bits to store multiple pieces of information in a compact format. The eight bits control the following properties of a sprite:

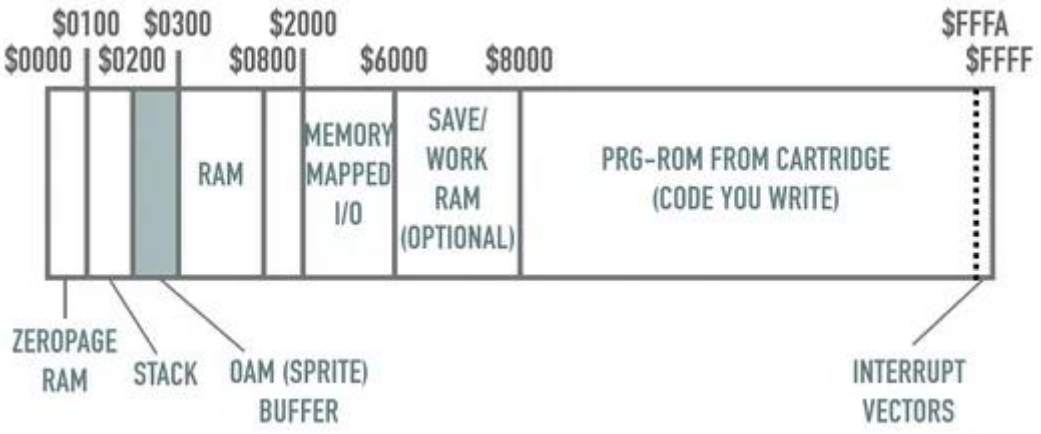
Sprite Attribute Flags

Bit #	Purpose
7	Flips sprite vertically (if "1")
6	Flips sprite horizontally (if "1")
5	Sprite priority (behind background if "1")
4-2	Not used
1-0	Palette for sprite

■ *Object Attribute Memory (OAM)*

The region in PPU memory where sprite data is stored is called "Object Attribute Memory", or "OAM". This region of memory is different in that there are special MMIO addresses that the CPU can use to update the contents of OAM all at once, at high speed. Being able to update OAM quickly is a necessity for fast-paced games, so that all 64 sprites can move smoothly every frame.

To use this high-speed copying, the CPU needs to have all of the sprite data ready to go in a contiguous page of memory. (A *page* is a block of 256 bytes.) Generally, this "sprite buffer" is placed in CPU memory addresses `$0200-$02ff`.A few things to note about the memory map image here. First, "RAM" is listed as extending from `$0300-$0800`. In reality, the entire range from `$0000-$0800` is RAM (2KB); the region from `$0300-$0800` is just the portion of RAM that is not commonly allocated for a specific purpose. The region from `$0800- $2000` in the memory map is empty space - writes to that region of memory will silently fail, and reads from that range have undefined behavior. Finally, note that the entire area from `$8000-$FFFF` comes from the PRG-ROM chip on the cartridge, including the six bytes that define the locations of interrupt handlers.



A common CPU memory layout for the NES. While it is possible to change where in memory some items like the stack or sprite buffer are located, this is the most common arrangement.

Within the sprite buffer (and in OAM itself), every four bytes defines one sprite. So, the first eight bytes of the sprite buffer look like this:

Memory address	Purpose
$0200	Y position of sprite 0 (first sprite)
$0201	Tile number of sprite 0
$0202	Attribute flags for sprite 0
$0203	X position of sprite 0
$0204	Y position of sprite 1 (second sprite)
$0205	Tile number of sprite 1
$0206	Attribute flags for sprite 1
$0207	X position of sprite 1

- `$2003`: OAMADDR and `$4014`: OAMDMA

Once we have set up all of the sprite data we want to transfer, we use two new MMIO addresses in our code to send all of the sprite data to the PPU. `OAMADDR` is used to set where in OAM we want to write to; for all of our projects (and for most commercial games), this will always be `$00`, the beginning of the OAM block. `OAMDMA` initiates the transfer of an entire page of memory into OAM. Writing the high byte of a memory address to `OAMDMA` will transfer that page.

While it might seem like we only need to write to OAM when something has changed, the OAM portion of PPU memory is implemented with "dynamic RAM", which means that it is highly unstable and needs to be continuously refreshed, even if nothing has changed. In practice, this means that we want to write to OAM once per frame of graphics (60 times per second).

❖ Non-Maskable Interrupts (NMI)

Fortunately, the NES has an easy-to-use system for running code once per frame: the Non-Maskable Interrupt (NMI). NMI is one of the three interrupt vectors the 6502 knows how to handle. The NMI event is triggered each time the PPU enters "vblank", which occurs at the end of each frame of graphics. "Vblank" stands for "vertical blank"; there is a similar "Hblank" or "horizontal blank" as well. To understand what these terms mean, we need to look at how the CRT televisions and monitors of the era work.

Until new technologies like plasma and LCD took over in the mid-2000's, most televisions used a technology called "cathode-ray tube" or CRT. CRTs work by shooting a beam of electrons through an "electron gun" to hit the inside of a phosphorescent screen, which absorbs the energy of the electrons and converts it into light. The electron gun sweeps across the screen continuously in horizontal lines from top to bottom, starting at the top-left corner and ending at the bottom-right corner before beginning again. The speed of these sweeps is determined by the video signal the television is designed to display. The NTSC standard used in the US and Japan calls for 60 frames per second, while the competing PAL standard used in Europe uses 50 frames per second.

When the electron gun is resetting itself — either to start a new horizontal line from the left edge or when moving from bottom right to top left to start a new frame — the stream of electrons is temporarily stopped, so as not to inadvertently cause graphical issues. These "blanking periods" are the only times when the display on screen is not changing. "Hblank" occurs at the end of each horizontal line and is incredibly brief, lasting only 10.9 microseconds for NTSC. "Vblank" is comparatively much longer, though still short: about 1250 microseconds, or 0.00125 seconds.

Since Vblank is one of the only times that nothing is being output to the screen, and since Hblank is far too short to do meaningful work, it is common practice to perform most graphical updates during Vblank, i.e. as part of the NMI handler. Currently, the NMI handler in the test project looks like this:

```
.proc nmi_handler
  RTI
.endproc
```

RTI, as discussed previously, is the opcode for Return from Interrupt. Let's update the NMI handler to copy the memory from $0200-$02ff into OAM each time it runs:

```
.proc nmi_handler
  LDA #$00
  STA OAMADDR
  LDA #$02
  STA OAMDMA
  RTI
.endproc
```

A quick review of the assembly we learned back in Chapter 5 is in order. On line 2, we load the *literal* value zero into the accumulator. In case you've forgotten: a number given to an instruction like `LDA` is, by default, a memory address. `LDA $00` means "load the accumulator with the value stored at memory address zero". Adding a "#" tells the assembler that this is a literal value, not a memory address.On line 3, we store (write) this zero to the OAMADDR address. This tells the PPU to prepare for a transfer to OAM starting at byte zero. Next, we load the literal value two into the accumulator, and write it to OAMDMA. This tells the PPU to initiate a high-speed transfer of the 256 bytes from $0200-$02ff into OAM.

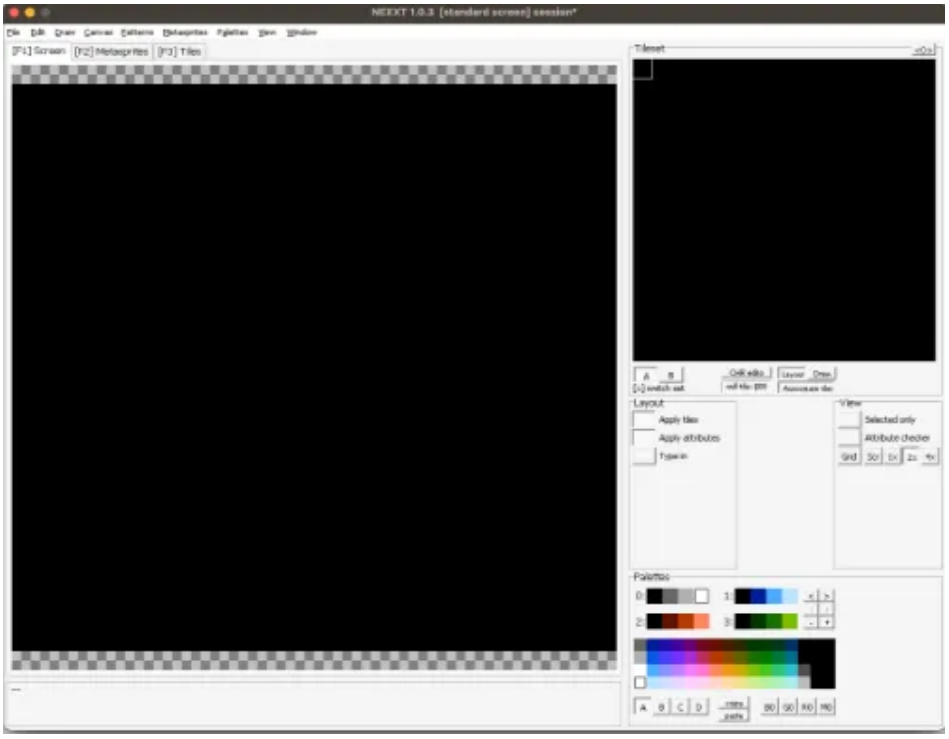
In order to use `OAMADDR` and `OAMDMA` in our code, we need to update our constants file to include these new constants. Here is the updated `constants.inc`:

```
PPUCTRL   = $2000
PPUMASK   = $2001
PPUSTATUS = $2002
PPUADDR   = $2006
PPUDATA   = $2007
OAMADDR   = $2003
OAMDMA    = $4014
```

We now have a reliable and automated way to keep OAM up to date. But where do our sprite graphics come from? As mentioned in the last chapter, the CHR-ROM chip in a cartridge holds two pattern tables, one for sprites and one for backgrounds. We will need to create our own pattern tables to display sprites on screen. This is where NEXXT comes in handy.

■ *Using NEXXT*

Open NEXXT. You should see a screen similar to this:

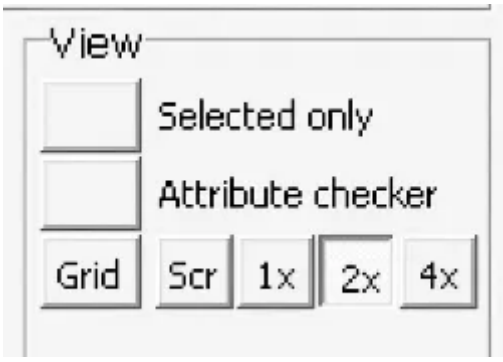


The main screen of NEXXT.

Before we dive into creating sprites, let's take a quick tour of how NES Lightbox is organized. The large area on the left half of the screen is a sort of canvas that can be used to draw backgrounds using tiles from a pattern table. We will not use this area until we start discussing background graphics. The right half of the interface is broadly divided into pattern tables ("Tileset") and palettes.

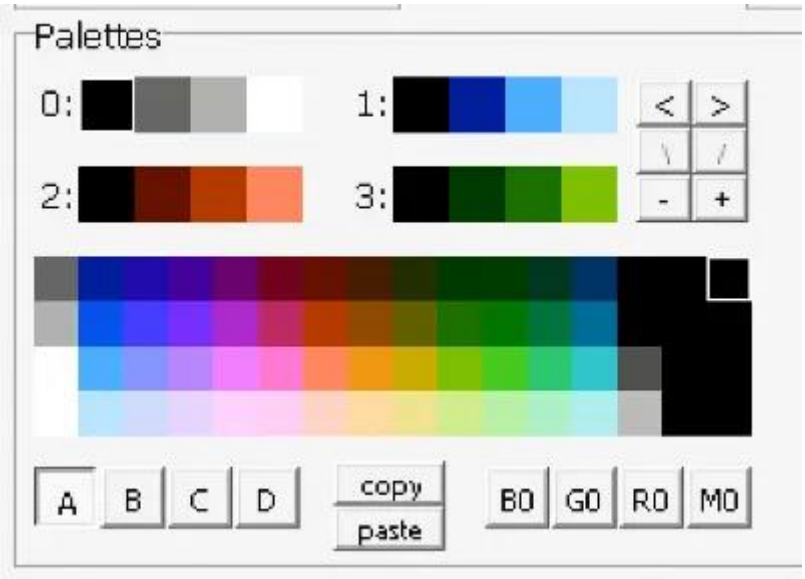
The "Tileset" area.

Within the “Tileset” area, the main element is a display of a pattern table. Just below the pattern table display is a toggle switch for “Bank A / Bank B”. As mentioned before, generally one pattern table is used for sprites, and the other is used for backgrounds. The A/B switch lets you flip between the two pattern tables. The “CHR editor” button will open a separate tile editor window, to make it easier to set the contents of each tile.



The "View" panel.

There is also a “View” panel, which provides the option to turn on and off a variety of grids showing the boundaries of each tile in the pattern table and in the large area on the left.



The "Palettes" area.

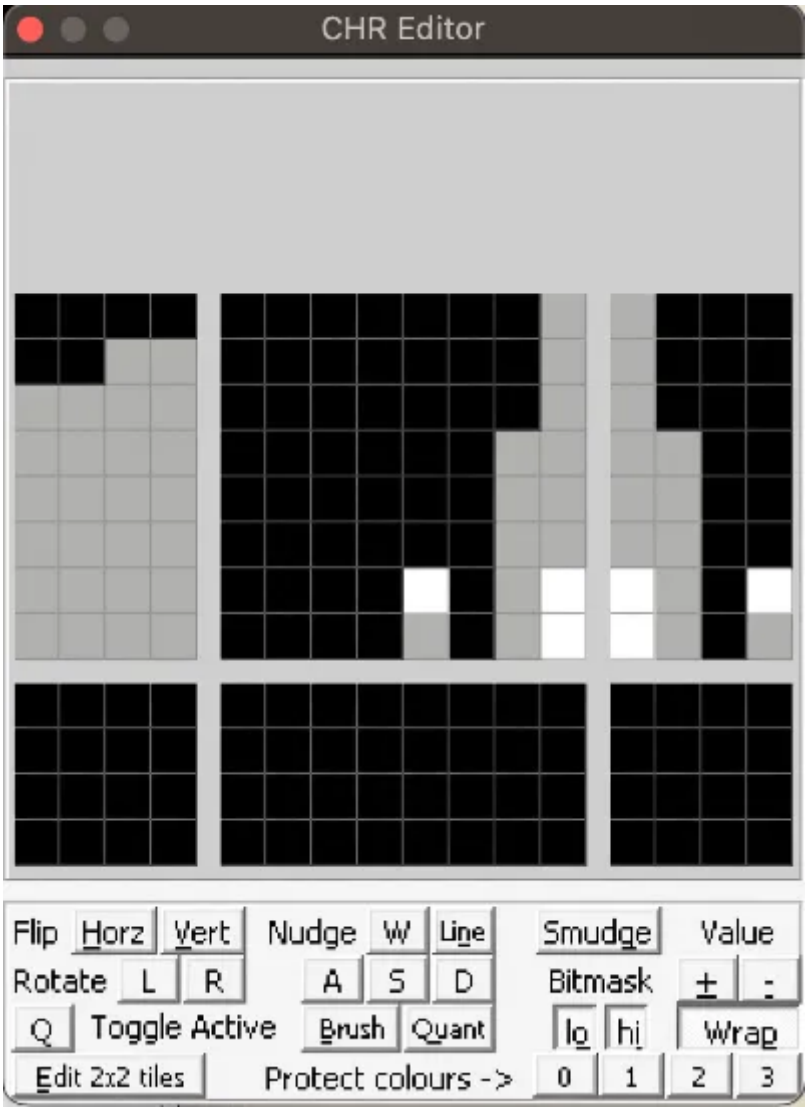
The "Palettes" area lets you preview how different palette choices will affect your tiles. There are four numbered palettes. Clicking any color will change the display of the sprites in the pattern table to the palette that color is part of. You can change a color by clicking a color in a palette, then choosing a color from the display of all possible colors at the bottom of this section. Hovering over any color in the pop-up will display the hexadecimal value the NES uses for that color, to use in your code.

Changing the first color of any palette in the palettes area will change the background color of the large canvas area as well as the background of the pattern table display.

❖ Making Your Own Tiles

To help you get started, I have created a starter .chr file that features some basic sprites and a full font in the background table. Download graphics.chr (https://famicom.party/book/projects/10-spritegraphics/src/graphics.chr) and open it in NEXXT ("Tilesets" → "Open CHR..." → select graphics.chr).

To edit or create tiles, click on the space in the pattern table display for the tile you wish to alter, then click the "CHR editor" button. This will open a separate tile-editing window.



NES Lightbox's "Edit Tile" window.

To edit a tile (or create new tiles!), select a palette index from the Palettes area, then click pixels in the CHR Editor window to set them to that palette index. The rotate and flip buttons, among others, allow you to easily make large-scale edits. Once you have created a set of tiles to work with, save your work by choosing “File” → “Patterns (.chr)” → “Save 8K (both tables)...“.

Note that clicking on different palettes in the main window changes all displayed colors to the colors from that palette. This is extremely useful for testing out what your tiles will look like in the palettes used by your game.

■ *Displaying Sprites In Your Game*

Download the full source code for this example: 10-spritegraphics.zip (https://famicom.party/book/projects/10-spritegraphics.zip)

In order to display our tiles in a game, we first need to make sure we load the .chr file that contains them. In our last project, we simply reserved 8,192 bytes of empty space for the CHR-ROM chip. Now that we have actual tiles, we will load them from a .chr file directly. Change the `.segment "CHR"` section to this:

```
.segment "CHR"
.incbin "graphics.chr"
```

`.incbin`, as you might expect, is a new assembler directive that instructs ca65 to include raw binary data (as opposed to `.include`, which is processed by the assembler). With our tiles in place, it's time to draw something. Here, we will use the four "spaceship" tiles from graphics.chr, but feel free to use your own tiles instead.

Next, we will need to fill out an entire palette, instead of just setting the first palette color to green (`$29`). We'll extend `.proc main` as follows:

```
20  .proc main
21    ; write a palette
22    LDX PPUSTATUS
23    LDX #$3f
24    STX PPUADDR
25    LDX #$00
26    STX PPUADDR
27    LDA #$29
28    STA PPUDATA
29    LDA #$19
30    STA PPUDATA
31    LDA #$09
32    STA PPUDATA
33    LDA #$0f
34    STA PPUDATA
```

Notice that we only have to set an address with `PPUADDR` once; each time we write to `PPUDATA`, the PPU memory address is automatically increased by one.

Next, we need to store the data for our sprites. We are going to start by drawing one sprite, the top-left "corner" of the spaceship. As discussed earlier, we will store all our sprite information in memory between `$0200-$02ff`, and copy it to the PPU using DMA transfer (in our NMI handler). Let's continue modifying `.proc main` to copy sprite data into `$0200-$02ff`:

```
36    ; write sprite data
37    LDA #$70
38    STA $0200 ; Y-coord of first sprite
39    LDA #$05
40    STA $0201 ; tile number of first sprite
41    LDA #$00
42    STA $0202 ; attributes of first sprite
43    LDA #$80
44    STA $0203 ; X-coord of first sprite
```

Finally, we need to make one more change to `.proc main`. In our previous examples, after writing palette data, we turn on the screen by writing to `PPUMASK`. However, now that we are using the NMI handler, we need to tell the CPU to generate NMI events. We can do that by writing to `PPUCTRL`, like this:

```
46 vblankwait:       ; wait for another vblank before continuing
47   BIT PPUSTATUS
48   BPL vblankwait
49
50   LDA #%10010000  ; turn on NMIs, sprites use first pattern table
51   STA PPUCTRL
52   LDA #%00011110  ; turn on screen
53   STA PPUMASK
```

For full details of the possible options that can be set via PPUCTRL and PPUMASK, see the NESDev wiki (https://www.nesdev.org/wiki/PPU_registers).

At this point, our code will draw a single sprite (the top-left of the spaceship) to the screen, at X `$80` and Y `$70` (near the middle of the screen). The sprite data that we wrote to CPU memory between `$0200-$02ff` will be copied to PPU memory once per frame by our NMI handler. Assembling, linking, and running this code gives the output seen below.

While this works, our code is very inefficient. To draw all 64 sprites this way, we would need 128 lines of code. To make our sprite code more manageable, we will store the sprite data separately from the code that writes it to memory, and use a loop to iterate over the data. This requires a few new assembly opcodes, which we will learn next chapter.

11. More Assembly: Branching and Loops

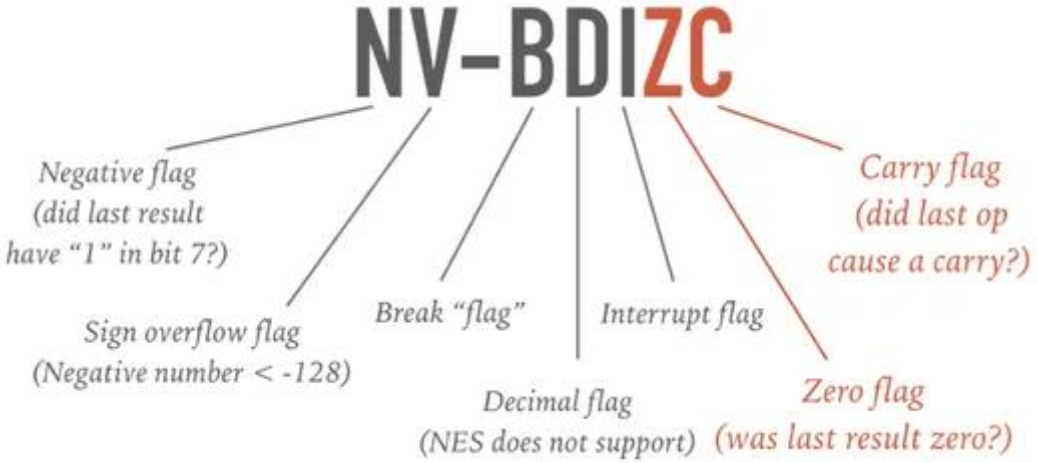
At the end of the last chapter, we successfully drew a single sprite to the screen, but doing so took a large amount of code. In this chapter, we will learn some new assembly opcodes that will help us separate data from logic and, as a side effect, make our code much more efficient and easier to read and reason about.

■ *Flow Control in Assembly*

With the exception of JMP, all of the assembly we have seen so far has been completely linear: the processor reads a byte from the next memory address and processes it, moving from the start of our ROM file to the end. *Flow control* refers to the ability to write code that can evaluate certain conditions and change which code will be executed next based on the result. For the 6502, there are two forms of flow control. JMP is an "unconditional" *jump* to a different point in the ROM (it does not perform any kind of test or evaluation). The other form of flow control is called "branching", because it performs a test and then moves to one of two different "branches" of code based on the result.

❖ The Processor Status Register

The key to branching is a part of the 6502 called the *processor status register*, often referred to as P. The status register, like all other 6502 registers, is eight bits in size. Unlike the other registers, the status register is not directly accessible to the programmer. Each time the processor performs an operation, the status register changes to reflect the results of that operation. Each bit in the status register gives information about a particular aspect of the last operation.



The eight bits of the processor status register (NV-BDIZC).

For our purposes, the two most important bits (or "flags") of the processor status register are the Z ("zero") and C ("carry") bits. The zero flag is *set* (1) if the result of the last operation was zero. The zero flag is *cleared* (0) if the result of the last operation was anything but zero. Similarly, the carry flag is set if the result of the last operation caused a "carry", and cleared otherwise.

❖ Carrying

While the zero flag is straightforward, the carry flag requires some additional explanation. Consider what happens when we add the decimal numbers 13 and 29. If we were to perform this addition by hand, we would first add the "3" from 13 to the "9" from 29. The result is 12, which is too large to fit into a single decimal digit. So, we write down a "2" and *carry over* the "1" into the next column to the left. Here, we add together the "1" from 13, the "2" from 29, and the "1" that we carried over. The result is 4, which we write down under that column, for a total of 42.

The carry flag on the 6502 performs the same function, but for bytes. An addition operation will result in the carry flag being set if the result of the addition is larger than what can fit in one byte (i.e. if the result is greater than 255). We will generally use an opcode that force-clears the carry flag before performing an addition, to avoid having a carry set by a previous operation persist to the current addition.

Why would you ever want to persist a previous carry flag before performing an addition? *Not* clearing the carry flag allows you to add together multi-byte numbers, by first adding the lowest bytes of the two numbers and letting that addition set the carry flag if needed. When you add the next-lowest bytes of the two numbers, the carry flag will automatically be added in.

Subtraction works similarly to addition, except that we will generally *set* the carry flag before performing a subtraction. A subtraction operation results in the carry flag being *cleared* if the number being subtracted is larger than the number it is being subtracted from. As an example, if we subtract 17 from 15, the result is -2. This number is less than zero, so in order to carry out the subtraction we need to "borrow" from the next-lowest column, using up the carry flag that we set before starting the subtraction.

It is important to note here that much like single-digit decimal numbers, bytes "wrap around" when adding beyond 255 or subtracting beyond zero. If a byte (or register, or memory value) is currently at 253 and you add 7, the result is not 260 — it is 4, with the carry flag set. Similarly, if a byte's value is 4 and you subtract 7, the result is 253 with a cleared carry flag, not -3.

We will cover how to actually perform addition and subtraction in a later chapter; for now, all you need to know is that the instructions the processor executes will change the values of the zero and carry flags in the processor status register.

■ *Branching*

Now that you know about the processor status register, we can use the results of operations to branch to different parts of our code. The simplest way to use branching is to build a loop that does something 256 times. In its most basic form, it looks like this:

```
  LDX #$00
LoopStart:
  ; do something
  INX
  BNE LoopStart
  ; end of loop
```

Before we look at how the loop actually works, there are two new opcodes here to discuss. `INX` stands for "increment X"; it adds one to the value of the X register and stores it back in the X register. `BNE` stands for "Branch if Not Equal to zero"; it changes the normal flow of code execution if the zero flag in the processor status register is cleared (i.e., the result of the last operation was not zero).

The loop begins by loading the immediate value `$00` into the X register. Next, we have a label (`LoopStart`), which will be used by our branching instruction at the end of the loop. After the label, we do whatever it is that we want our loop to perform, then we increment the X register. Like all math-related operations, this will update the flags in the processor status register. The final line of the loop checks the value of the zero flag. If the zero flag is set, nothing special happens — the program just keeps on running, with whatever comes after `; end of loop` being executed next. If the zero flag is cleared, then `BNE LoopStart` tells the processor to find the `LoopStart` label and execute whatever is located there next instead — in other words, running the next iteration of the loop.

In actual operation, this loop will run 256 times. On the first iteration of the loop, the value of the X register is zero. After `INX`, the value of the X register is one. Since the result of `INX` was not a zero, the zero flag will be cleared. When we get to `BNE LoopStart`, because the zero flag is cleared, the processor will go back to the `LoopStart` label and run through the loop again. This time, the X register will become two, which is still not zero, and the loop will run again. Eventually, the X register's value will be 255. When we run `INX` this time, the X register will "roll over" to zero, and the carry flag will be set. Note that in this case, we could have branched based on either the zero flag or the carry flag. When the X register "rolls over" from 255 to 0, both the zero and carry flags will be set. Now that the last operation resulted in a zero, `BNE LoopStart` will no longer be triggered, and the processor will continue on to whatever comes after the loop.

There is one more thing to note here before we move on. After running this code through our assembler and linker, all of our labels (like `LoopStart`) will be stripped out and replaced with actual memory addresses. To ensure that branches do not take up an undue amount of processor time, the data that follows a branch command is not a memory address but a signed one-byte number that is added to whatever memory address is in the program counter. As a result, the code you want to branch to must be less than 127 bytes before, or less than 128 bytes after, your branch instruction. If you need to branch to something that is further away, you'll need to `JMP` to that label instead. This is probably not going to be a common occurrence unless you are writing fairly complicated code, but it's an interesting implementation detail that might lead to a hard-to-track-down bug somewhere down the line.

■ *A Review of Looping/Branching Opcodes*

We've already seen `INX` and `BNE`, but these are only two of the opcodes that you are likely to use for creating loops. Let's look at the ten new opcodes that you should add to your toolset.

❖ Incrementing and Decrementing Opcodes

These opcodes allow you to add or subtract by one in a single opcode. There is no need to explicitly set or clear the carry flag before using one of these opcodes.

`INX` and `INY` will add one to ("increment") the X or Y register, respectively. In the opposite direction, `DEX` and `DEY` will subtract one from ("decrement") the X or Y register. Finally, you can use `INC` and `DEC` to increment or decrement the contents of a memory address. As an example, you could use `INC $05` to add one to whatever is stored at memory address `$05`, and store the result back in `$05`.

All of the increment/decrement opcodes will update the values of the zero and carry flags of the processor status register.

❖ Branching Opcodes

There are branching opcodes for each flag of the processor status register. Each flag has two opcodes - one that branches if the flag is set, and one that branches if the flag is cleared. For our purposes, the only branching opcodes you will need to use check the values of the zero and carry flags.

`BEQ` ("Branch if Equals zero") and `BNE` ("Branch if Not Equals zero") will change the flow of the program if the zero flag is set or cleared, respectively. `BCS` ("Branch if Carry Set") and `BCC` ("Branch if Carry Cleared") do the same for the carry flag. What follows each opcode should generally be a label for what code should be executed next if the branching opcode's conditions are met.

■ *Another Branching Example*

I'd like to present one more example of branching. This time, our loop will run eight times instead of 256.

```
  LDY #$08
LoopTwo:
  ; do something
  DEY
  BNE LoopTwo
  ; end of loop
```

As in the previous loop example, here we first set up the pre-conditions of our loop by setting the Y register to `$08`. Then we have the label that our branching opcode will use later. After we've done whatever it is that we want our loop to do on each iteration, we decrement the Y register and then branch back to the start of the loop if the zero flag is cleared.

In a more modern, C-like programming language (such as JavaScript), this entire loop could be re-written as follows:

```
for (y = 8; y != 0; y--) {
  // do something
}
```

■ *Making Comparisons*

While the loops we have seen so far are useful, they require some careful setup. The loops above rely on our loop counter becoming zero in order to end the loop. To make more flexible and powerful loops, we need the ability to make arbitrary comparisons. In 6502 assembly, the opcodes that let us do that are `CMP`, "Compare (with accumulator)", `CPX`, "Compare with X register", and `CPY`, "Compare with Y register".

Each of these opcodes works by performing a subtraction, setting the zero and carry flags as appropriate, and then discarding the result of the subtraction. Remember that when we perform a subtraction, we first set the carry flag. This means that we have three possible outcomes from a comparison, based on the register value and the value we are comparing it to:

1. *Register is larger than comparison value*: Carry flag set, zero flag clear
2. *Register is equal to comparison value*: Carry flag set, zero flag set
3. *Register is smaller than comparison value*: Carry flag clear, zero flag clear

We can use this information to create more complicated program logic. Consider the case where we load a value from memory and then check whether it is greater than, equal to, or less than `$80`.

```
  LDA $06
  CMP #$80
  BEQ reg_was_80
  BCS reg_gt_80
  ; neither branch taken; register less than $80
  ; do something here
  JMP done_with_comparison ; jump to skip branch-specific code
reg_was_80:
  ; register equalled $80
  ; do something here
  JMP done_with_comparison ; skip the following branch
reg_gt_80:
  ; register was greater than $80
  ; do something here
  ; no need to jump because done_With_comparison is next
done_with_comparison:
  ; continue with rest of the program
```

This kind of three-way branch is fairly common. Note the presence of a label to mark the end of all branch code, so that earlier code can `JMP` over branch-specific code that shouldn't be executed if the branch was not taken.

■ *Using Comparisons in Loops*

To close out this chapter, let's take a look at how comparisons can be used to create more sophisticated loops. Here is a loop that runs eight times, but counts up from zero instead of counting down from eight.

```
  LDX #$00
loop_start:
  ; do something
  INX
  CPX #$08
  BNE loop_start
  ; loop is finished here
```

Here, we set the X register to zero before starting our loop. After each loop iteration, we increment the X register and then compare X to `$08`. If the X register does not equal eight, the zero flag will not be set, and we will go back to `loop_start`. Otherwise, the `CPX` will set the zero flag (because eight minus eight equals zero), and the loop will be finished.

To review, in this chapter we learned the following opcodes:

- `INX`
- `INY`
- `INC`
- `DEX`
- `DEY`
- `DEC`
- `BNE`
- `BEQ`
- `BCC`
- `BCS`
- `CMP`
- `CPX`
- `CPY`

That's 13 more opcodes to add to your toolset!

Next chapter, we will re-factor our code to take advantage of loops and comparisons, in preparation for creating background graphics.

12. Practical Loops

Now that you know how to create loops for various purposes, it's time to put that knowledge to use cleaning up our existing code. Using loops will make your assembly code cleaner, more readable, and easier to extend in the future.

To make full use of loops, we will pair the looping opcodes we learned last chapter with a new *addressing mode*. Back in Chapter 5, when we first talked about opcodes, two addressing modes were introduced: *absolute* mode (e.g., `LDA $8001`) and *immediate* mode (e.g., `LDA #$a0`). Now, we will learn a third addressing mode: *indexed mode*.

■ *Indexed Mode*

Indexed mode combines a fixed, absolute memory address with the variable contents of an index register (hence the name "index register"). To use indexed addressing mode, write a memory address, a comma, and then a register name.

```
    LDA $8000,X
```

The example code above will fetch the contents of memory address (`$8000` + the value of the X register). If the current value of the X register is `$05`, then the command `LDA $8000,X` will fetch the contents of memory address `$8005`.

Using indexed mode allows us to perform actions across a range of memory addresses with ease. As a simple example, here is a code snippet that will set the 256 bytes of memory from `$3000` to `$30FF` to `$00`.

```
    LDA #$00
    TAX
  clear_zeropage:
    STA $3000,X
    INX
    BNE clear_zeropage
```

To review, line 1 above sets the accumulator to zero (`#$00`), and then line 2 copies that zero to the X register. Line 4 stores the zero from the accumulator to memory address (`$3000` plus X register), which will be `$3000` the first time through the loop. Line 5 increments the X register, and then line 6 checks the status of the zero flag in the processor status register. If the last operation was not equal to zero, we return to the label at line 3. When we increment the X register from zero to one, the result of the last operation is one, so the zero flag will not be set and the loop will repeat again. The next time through the loop, when we reach line 4, the zero from the accumulator will be stored again at memory address (`$3000` plus X register), which will now be memory address `$3001`. The loop will repeat until the X register is already `$ff` and the increment at line 5 changes the X register to `$00`.

■ *Loading Palettes and Sprites*

Now that you understand indexed mode, let's use it to simplify our existing code for loading palettes and sprites. In our existing code from Chapter 10, palette and sprite loading is tedious, repetitive, and error-prone. This is in large part because the code tightly mixes *data* and *logic*. By using loops and indexed addressing, we can separate the palette and sprite data from the code that sends that data to the PPU, making it easier to update the data without inadvertently breaking things.

Our code from Chapter 10 to load palette data looks like this:

```
21    ; write a palette
22    LDX PPUSTATUS
23    LDX #$3f
24    STX PPUADDR
25    LDX #$00
26    STX PPUADDR
27    LDA #$29
28    STA PPUDATA
29    LDA #$19
30    STA PPUDATA
31    LDA #$09
32    STA PPUDATA
33    LDA #$0f
34    STA PPUDATA
```

Let's separate out the palette values and store them somewhere else. The palette values here are read-only data, so we will store them in the `RODATA` segment and not in the current `CODE` segment. It will look something like this:

```
60  .segment "RODATA"
61  palettes:
62  .byte $29, $19, $09, $0f
```

We set a label (`palettes`) to easily identify the start of our palette data, and then we use the `.byte` directive to tell the assembler "what follows is a series of plain data bytes, do not try to interpret them as opcodes".

Next, we will need to adjust our palette-writing code to loop over the data in `RODATA`. We'll keep lines 21-26 above that set the PPU address to `$3f00`, but starting at line 27, we'll make use of a loop:

```
27  load_palettes:
28    LDA palettes,X
29    STA PPUDATA
30    INX
31    CPX #$04
32    BNE load_palettes
```

Instead of hard-coding each palette value, we load it as "the address of the `palettes` label plus the value of the X register". By incrementing the X register each time through the loop (`INX`), we can sequentially access all of the palette values.

Note that to end the loop, we are comparing against `#$04`. This ensures that we will run this loop for four, and only four, values. If we set the comparison operand to something larger, we could end up reading memory beyond what we intended as palette storage, which can have unpredictable effects.

Now that our palettes are loading in a cleaner fashion, let's turn our attention to sprite data. Just like with palettes, we can store our sprite data in `RODATA` and read it with a loop. The current sprite loading code looks like this:

```
36    ; write sprite data
37    LDA #$70
38    STA $0200 ; Y-coord of first sprite
39    LDA #$05
40    STA $0201 ; tile number of first sprite
41    LDA #$00
42    STA $0202 ; attributes of first sprite
43    LDA #$80
44    STA $0203 ; X-coord of first sprite
```

Following the same process as with the sprites, our new sprite loading code will look like this:

```
36    ; write sprite data
37    LDX #$00
38  load_sprites:
39    LDA sprites,X
40    STA $0200,X
41    INX
42    CPX #$04
43    BNE load_sprites
```

This code is subtly different from the palette loading code. Note that on line 40, instead of writing to a fixed address (`PPUDATA`), we use indexed mode to increment the address to *write to* as well as the address to *read from*.

One more step: we still need to move our sprite data into `RODATA`. Here is our sprite data, in a much more readable, one-line-per-sprite format:

```
63  sprites:
64  .byte $70, $05, $00, $80
```

■ *Homework*

Now that you have seen how to use loops and branching to make assembly code more readable and maintainable, it's time to try them out for yourself. Extend the existing code to load four full palettes (with colors of your choosing) and to draw at least four sprites to the screen. You'll need to modify the palette and sprite data in `RODATA` as well as change the loop counters in the palette-loading and sprite-loading loops. Don't forget to re-assemble your source files and link them into a new .nes file (see the end of Chapter 8 for a refresher).

All code from this chapter can be downloaded in a zip file (https://famicom.party/book/projects/12-practicalloops.zip).

13. Background Graphics

Before we start animating sprites around the screen, I'd like to introduce you to how background graphics work on the NES. We explored the mechanics behind background graphics in Chapter 9, but in this chapter we'll look at the actual code required to display backgrounds.

■ *The Background Pattern Table*

First, you'll need graphical tiles in the background pattern table. The PPU has two pattern tables for tiles, one for sprites and one for backgrounds. In the previous chapters, we added some tiles to the sprite pattern table. Now, we'll need to add tiles to the background table.

For this chapter, I've updated the CHR file (https://famicom.party/book/projects/13-backgroundgraphics/src/starfield.chr) to include some pre-made background tiles. There are tiles for numbers and letters (to allow us to display text to the player), solid-colored squares of each of the four palette colors, and a few "starfield" tiles. You can view (and edit) these tiles in NES Lightbox. Our long-term goal is going to be the creation of a vertically-scrolling shooter game in the style of *Star Soldier*.

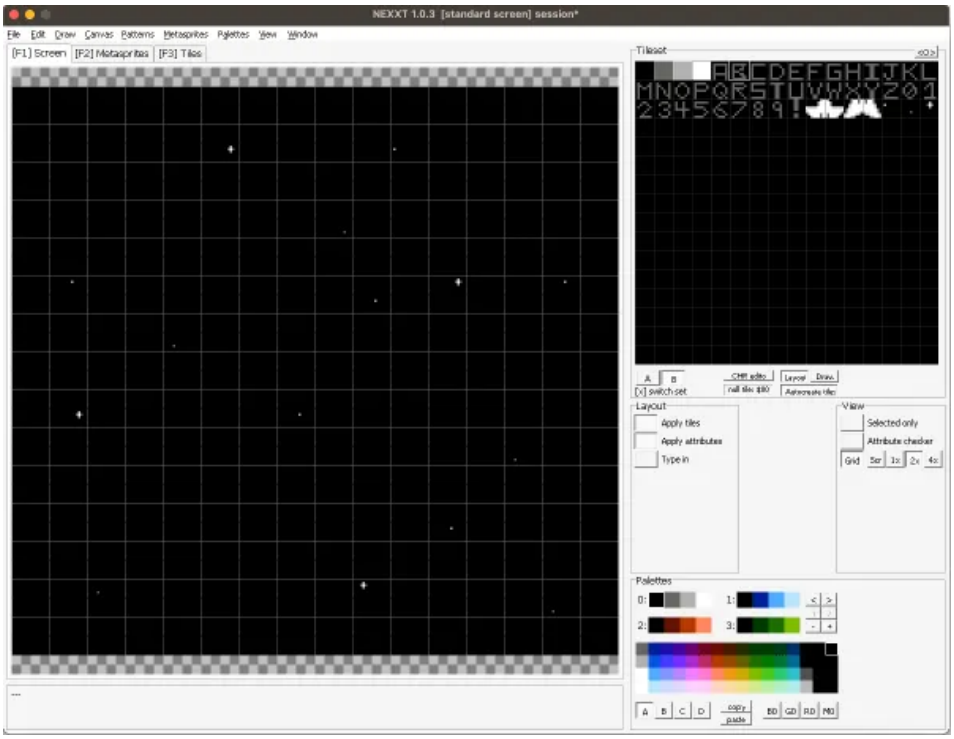
Star Soldier, developed by Hudson Soft and released for the Famicom in 1986 and the NES in 1988, is a vertical-scrolling shooter series.

The new background tiles in Bank B of "starfield.chr"

To make use of these new background tiles, we will need to write a nametable. By default, the attribute table will be all zeroes, meaning that every tile in the nametable will be drawn with the first palette. To use other palettes, we will need to write an attribute table as well.

■ *Writing a Nametable*

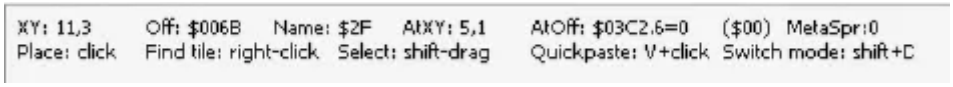
First, let's figure out where to put our new tiles in the nametable. We only need to write to nametable addresses where we intend to place a tile — everywhere else, the default background color from our palettes will be used. In NEXXT, click a tile from the right side of the screen, then click anywhere in the large area on the left to "draw" with the tile. Here is the example background that we will use:



Placing background tiles in NEXXT.

Once you have an arrangement that you like, you can save what you have so far to a `.nam` file. From NEXXT's menus, select "File" → "Canvas (.nam or .map)" → "Save as screen (32x30)", then choose a filename and a location to save the file. To reload your background later for editing, open NEXXT and select "File" → "Canvas (.nam or .map)" → "Open canvas (map or nametable)", and select the `.nam` file you saved earlier.

When you hover over any tile in the large left pane, the status bar at the bottom of NEXXT shows you useful information for writing nametables and attribute tables. As discussed in Chapter 9, the NES has four nametables, the first of which starts at PPU memory address `$2000`. The bottom status bar of NEXXT shows the "nametable offset" ("Off") and "attribute offset" ("AtOff") for each tile position — that is, how many bytes beyond the start of the nametable or attribute table it is. It also displays the current tile number as "Name".



The bottom status bar of NEXXT, displaying nametable and attribute table offsets.

We can use this information to fill out our nametable. As you might remember, the nametable is just a series of tile numbers; palette selection occurs in the attribute table. To create the nametable, we need to write the appropriate tile number to each nametable address as displayed in NEXXT. Let's start with the "large" star tile (tile number `$2f`, as you can see by hovering over the tile in the right-side "Tileset" window or the tile in the canvas view, and looking at the status bar at the bottom of the application window). From hovering over the places where the large star tile is used, we can see that we need to write the value `$2f` to the following addresses in PPU memory:

- `$206b`
- `$2157`
- `$2223`
- `$2352`

Each of these addresses is the start of the nametable ($2000) plus the offset ("Off") for each tile in the canvas.

The process is the same as what we have used before for palettes and sprites: read from `PPUSTATUS`, write the address you want to send data to to `PPUADDR` ("high"/left byte first, then "low"/right byte), and then send the actual data to `PPUDATA`. Previously, we have used loops to do this, taking advantage of the way sequential writes to `PPUDATA` automatically increase the address by one. This time, though, we need to write to totally non-linear memory addresses, so we'll need to repeat the process in full for each background tile. Here is the code to write the first "large star" to the nametable:

```
46      ; write a nametable
47      ; big stars first
48      LDX PPUSTATUS
49      LDX #$20
50      STX PPUADDR
51      LDX #$6b
52      STX PPUADDR
53      LDA #$2f
54      STA PPUDATA
```

The one thing we can do to save on the number of commands we need to enter is to store the tile number in a different register from what we use to read `PPUSTATUS` and write to `PPUADDR`; here we are using the X register for the tile number, so that subsequent writes of the same tile to the nametable will not require re-loading it.

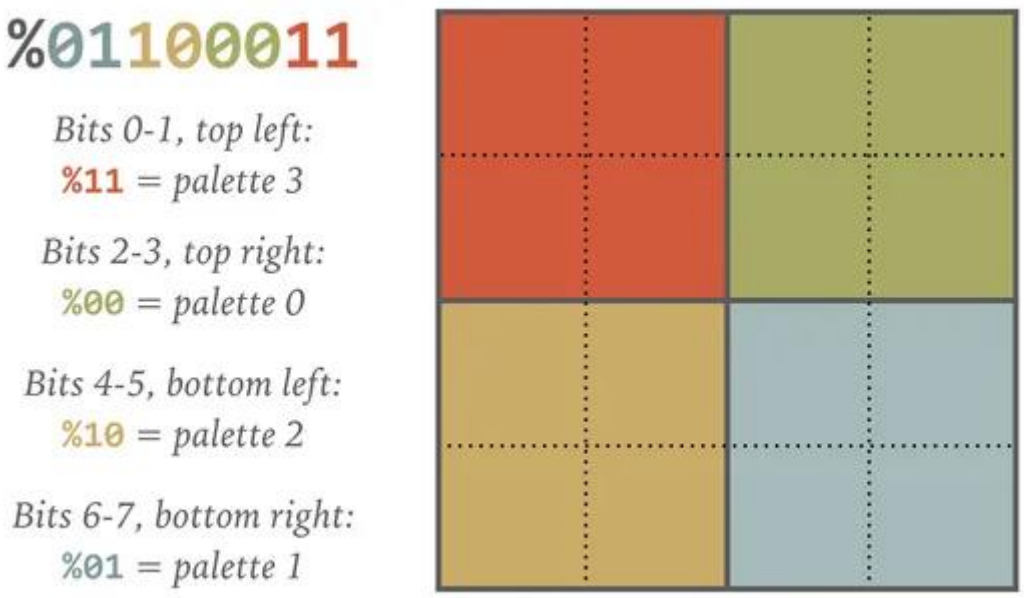
We can use the same procedure to add the two varieties of "small" star (`$2d` and `$2e`) to the nametable. To use different colors for the tiles, we will need to write an attribute table as well.

■ *The Attribute Table*

Background tile palettes are set via an attribute table, which uses one byte to store palette information for a 4x4 square of background tiles. To change the palettes used for a particular tile, first hover over that tile in the left-side nametable display and note the “AtOff” for the tile. Remember that an “AtOff” in NEXXT is an offset from the start of the nametable (e.g., `$2000`), not the start of an attribute table. Next, we need to figure out which 2x2 area of tiles (top left, top right, bottom left, or bottom right) the tile we want to change is part of. To help with finding the boundaries of an attribute table byte, “AtOff” will end with one of `.0`, `.2`, `.4`, or `.6`, which will tell you the starting bit of the current tile’s palette in the attribute table byte.

As an example, let’s change the palette used to draw the first “large star” in the nametable. By hovering over it in the nametable viewer, we can find the attribute offset (`$03C2.6`).

Each byte of the attribute table holds palette information for four 2x2 areas of background tiles, using two bits for each area. From left to right, the eight bits in an attribute table byte hold the palette number for bottom-right, bottom-left, top-right, and top-left 2x2 areas.



Layout of bits in an attribute table byte.

In this case, we want to change the palette for the bottom-right section of an attribute table byte, so we will need to change the leftmost two bits of the byte. By default, the attribute table is all zeroes, meaning every 2x2 area of the background uses the first background palette. Let's change our tile to use the second palette (`01`) instead of the first palette (`00`). To change just the bottom-right 2x2 area, and leave the other parts of this 4x4 area with the first palette, we would write the byte `%01000000` to the appropriate PPU memory address (`$23c2`). Here is what that would look like:

```
158     ; finally, attribute table
159     LDX PPUSTATUS
160     LDX #$23
161     STX PPUADDR
162     LDX #$c2
163     STX PPUADDR
164     LDA #%01000000
165     STA PPUDATA
```

Remember that this will set all tiles in that 2x2 region to use the second palette. In this case, the background tiles we are setting are relatively far apart, but if your backgrounds are more “busy” you will need to think carefully about where to change from one background palette to another. When you select a palette (by clicking any color within the palette) before placing a tile in the nametable display, NEXXT will update the underlying attribute table and update all tiles in the affected 2x2 area to use the new palette, which can help you identify potential attribute table clashes.

■ *Additional Changes*

As our games start getting more complicated, the simple reset and NMI handlers we started with will need some minor changes to prevent strange (and difficult to debug) graphical glitches. In our reset handler, I've added a loop that sets the Y-position of all sprites off the screen (i.e. any value larger than `$ef`). The state of CPU memory can be random at startup, which could result in portions of the OAM buffer at `$0200` having fake sprite data. This loop hides all sprites off the screen until we explicitly set them ourselves, which will prevent these "phantom" sprites from being visible to the player.

```
17      LDX #$00
18      LDX #$00
19    clear_oam:
20      STA $0200,X ; set sprite y-positions off the screen
21      INX
22      INX
23      INX
24      INX
25      BNE clear_oam
```

Second, our NMI handler will need to set the scroll position of the nametables. We will cover scrolling in detail in a later chapter, but for now just know that we are setting the scroll position to display the first nametable, with no scrolling. If we did not explicitly set this scroll position, we could accidentally display a combination of nametables at an unpredictable scroll point.

```
9     .proc nmi_handler
10      LDA #$00
11      STA OAMADDR
12      LDA #$02
13      STA OAMDMA
14      LDA #$00
15      STA $2005
16      STA $2005
17      RTI
18    .endproc
```

Finally, now that we are using both sprites and backgrounds, we need to set all eight palettes. To do so, I've expanded the palette definitions in the `RODATA` segment as follows:

```
192   palettes:
193   .byte $0f, $12, $23, $27
194   .byte $0f, $2b, $3c, $39
195   .byte $0f, $0c, $07, $13
196   .byte $0f, $19, $09, $29
197
198   .byte $0f, $2d, $10, $15
199   .byte $0f, $19, $09, $29
200   .byte $0f, $19, $09, $29
201   .byte $0f, $19, $09, $29
```

The first four palettes are background palettes, and the second set of four palettes are sprite palettes. Note that I have also changed the colors used in the first sprite palette to make the "spaceship" look better.

Since we now need to write more than just four palette bytes, I've changed the loop that writes palettes to use `CPX #$20` (16 values) instead of `CPX #$04`.

■ *Using NEXXT Sessions*

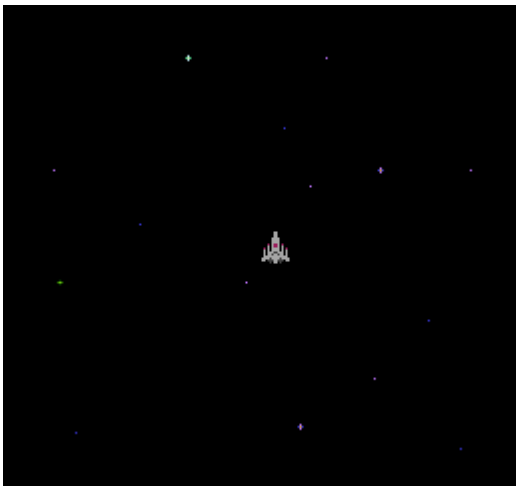
Before we move on to this chapter's homework, let's look at the "Session" functionality of NEXXT. A "session" is a list of files that make up a working environment (tileset, palettes, nametable) and all settings. The session file uses absolute paths to each file, which means that session files are unfortunately not portable between different computers.

To create a session file, select "File" → "Save Session As..." from NEXXT's menus. Choose a name and location for the session file and click "OK". Once you have saved (or loaded) a project, you can select "File" → "Save Session" to save the project's .chr, .nam, and .pal files with the current data in the application.

To restore your work environment later, select "File" → "Open Session..." and select the .nss file you saved earlier. If any of the project's files have changed location on disk, you can edit the .nss file with any text editor - it is simply a list of keys and values in plain text.

■ *Homework*

To help you get started, download all of the code from this chapter (https://famicom.party/book/projects/13-backgroundgraphics.zip). Here is what our project looks like so far running in an emulator:



For homework, add a new tile to the background pattern table ("Bank B" in NEXXT), use the new tile in a nametable, and update the attribute table for the new tile to use more than one palette. You can use the `.nam` file provided to get your NEXXT environment set up quickly. Be sure to save your modified `.chr` file and to re-build all .asm files each time you make changes.

14. Sprite Movement

In the last chapter, we created background graphics to display behind our sprites. While the addition of backgrounds helps make our game look more like an actual game, it is still entirely static - no different from a still picture. In this chapter, we'll look at how to move sprites around the screen. To do so, we will need to make some changes to how our game is drawing sprites.

First, we can no longer hard-code the positions of our sprites in cartridge ROM. There are many choices for where we could put our sprite's information, but the best location is "zero-page RAM".

As a reminder, "cartridge ROM" or PRG-ROM is the read-only part of your game, and in the NES' memory map, it is located from `$8000` to `$ffff`. All of your game's code is located there, though your game code can refer to memory addresses outside of PRG-ROM, like when we wrote sprite data to `$0200-$02ff`.

■ *Zero-Page RAM*

A "page" of memory on the NES is a contiguous block of 256 bytes of memory. For any memory address, the *high* byte determines the page number, and the *low* byte determines the specific address within the page. As an example, the range from `$0200` to `$02ff` is "page `$02`", and the range from `$8000` to `$80ff` is "page `$80`".

What, then, is "zero-page RAM"? Page zero is the range of memory from `$0000` to `$00ff`. What makes page zero useful for things like sprite positions is its speed. The 6502 processor has a special addressing mode for working with zero-page RAM, which makes operations on zero-page addresses much faster than the same operation on other memory addresses. To use zero-page addressing, use one byte instead of two when providing a memory address. Let's look at an example:

```
    LDA $8000 ; "regular", absolute mode addressing
              ; load contents of address $8000 into A

    LDA $3b   ; zero-page addressing
              ; load contents of address $003b into A

    LDA #$3b  ; immediate mode addressing
              ; load literal value $3b into A
```

Note that you *must* use just one byte in order to take advantage of zero-page addressing mode. The assembler does not know anything about the memory addresses passing through it. If you were to type `LDA $003b` instead of `LDA $3b` in your assembly code, the resulting machine code would use (slower) absolute mode, even though the memory address you are loading from is located in page zero.

So, using zero-page addressing gives us very fast access to 256 bytes of memory. Those 256 bytes are the ideal place to store values that your game will need to update or reference frequently, making them an ideal place to record things like the current score, the number of lives the player has, which stage or level the player is in, and the positions of the player, enemies, projectiles, etc.

Notice that I said the position of "the player", and not the positions of the individual tiles that make up the player. Any game that hopes to have more than a very small number of objects on screen at one time will need to carefully ration the use of zero-page addresses.

Let's start using zero-page RAM in our code. Because only addresses from $8000 and up are ROM (i.e., part of your actual cartridge / code that you write), we can't just write zero page values directly. Instead, we tell the assembler to *reserve* memory in page zero, like this:

```
.segment "ZEROPAGE"
player_x: .res 1
player_y: .res 1
```

Note the ":" after the name of each reserved byte - these look like the labels that we've already been using, and that's because they are! When we use the name of a reserved byte later in our code, we're telling the assembler to find the memory address that corresponds to that label and replace the name with the address, exactly the same as any other label in our code.

First, we tell the assembler that we want to reserve page zero memory by using the appropriate segment from our linker config file - in this case, `"ZEROPAGE"`. Then, for each memory range we want to reserve, we use the `.res` directive, followed by the number of bytes we want to reserve. Generally this will be "1" to reserve a single byte of memory, but being able to specify any number can be useful if, for example, you need to store a 16-bit number in page zero.

Now that we have reserved memory, we need to initialize it to a good starting value somewhere in our code. Two good options for this are either as part of the reset handler, or at the beginning of `main`. We'll opt for the reset handler approach here. In `reset.asm`, just before `JMP main`, add the following code:

```
    ; initialize zero-page values
    LDA #$80
    STA player_x
    LDA #$a0
    STA player_y
```

If you try to assemble this code, however (`ca65 src/reset.asm`), you will get an error:

```
Error: Symbol 'player_y' is undefined
Error: Symbol 'player_x' is undefined
```

Generally, reserved memory names are only valid in the same file where they are defined. In this case, we reserved `player_x` and `player_y` in our main file, but we were trying to use them in `reset.asm`. Thankfully, ca65 provides directives to export and import reserved zero-page memory so it can be shared between files. We'll just need to add an `.exportzp` directive in our main file:

```
.segment "ZEROPAGE"
player_x: .res 1
player_y: .res 1
.exportzp player_x, player_y
```

Then, in `reset.asm`, we can use an `.importzp` directive:

```
.segment "ZEROPAGE"
.importzp player_x, player_y
```

The `.importzp` directive should go inside of a `.segment "ZEROPAGE"`, even if you are not doing anything else with page zero values in that file.

When you assemble these files, ca65 will look through the other source files in the same directory looking for imports and exports and it will figure out what data should go where.

■ *Subroutines*

Since we only have a limited number of zero-page addresses (256) available, we need to ration them out carefully. Instead of storing the position of every player sprite tile individually (which would take 8 bytes of zero page just for x/y positions), we will store just an overall player X and Y coordinate and offload the drawing of the actual player sprites to a *subroutine*. Subroutines are assembly's version of functions - named, reusable code fragments.

To create a subroutine, make a new `.proc` in your code. The only requirement for a subroutine is that it *must* end with the opcode `RTS`, "Return from Subroutine". To *call* a subroutine, use the opcode `JSR`, "Jump to Subroutine", followed by the name of the subroutine (whatever follows `.proc`).

Before we go further, let's take a look at what actually happens when we call a subroutine. Here is some example code:

```
1    LDA #$80
2    JSR do_something_else
3    STA $8000
4
5  .proc do_something_else
6    LDA #$90
7    RTS
8  .endproc
```

When this code runs, the processor first puts the literal value `$80` into the accumulator. Then it calls the subroutine `do_something_else`. When the 6502 sees a `JSR` opcode, it pushes the current value of the program counter (the special register that holds the memory address of the next byte to be processed) onto the *stack*. A stack, in computer science, is a "last in, first out" data structure, similar to a stack of real-life plates. Adding something to the stack means putting it on top of the pile, and only the top-most element is available at any given time.

On the 6502, the stack is 256 bytes in size and is located at `$0300` to `$03ff`. The 6502 uses a special register, the "stack pointer" (often abbreviated "S"), to indicate where the "top" of the stack is. When the system is first initialized, the value of the stack pointer is `$ff`. Every time something is stored on the stack, it is written to `$0300` plus the address held in the stack pointer (e.g., the first write to the stack is stored at `$03ff`), and then the stack pointer is decremented by one. When a value is removed from the stack, the stack pointer is incremented by one.

Attempting to write more than 256 items to the stack at one time causes the stack pointer to wrap around from `$00` to `$ff`, meaning that further writes to the stack will overwrite already existing stack data. This generally-catastrophic scenario is called a *stack overflow* (though in the case of the 6502, it's more properly called a stack *underflow*).

So, on line 2, the processor sees a `JSR` opcode and stores the current value of the program counter on the stack. Then, it takes the operand of the `JSR` and puts that memory address into the program counter. Here, the processor skips from line 2 to line 6, and writes the literal value `$90` to the accumulator. The next opcode is an `RTS`. When the 6502 sees an `RTS`, it takes the "top" value from the stack (often referred to as "popping" and item off the stack) and puts it into the program counter. Given the way the stack works, this *should* be the address that was "pushed onto" the stack back when the processor saw a `JSR`. This pulls us back to whatever code is immediately after the `JSR`. Here, that means `STA $8000` - and the result will be writing `$90` to that memory address, not `$80`. Subroutines do not, by default, "save" the values of any registers either when they are called or when they return. In most higher-level programming languages, this is taken care of for you through concepts like "variable scope" or "lifetimes". In assembly, though, you must handle saving and restoring the state of all registers (including the processor status register!) if you need those values to remain the same when returning from a subroutine.

In general, it is good practice to always save and restore registers when subroutines are involved. Interrupts like NMI or IRQ can be called at any time - even while you are inside of another subroutine! - and it can be difficult to accurately predict what value is in a register if your subroutines / interrupt handlers are not written in a "defensive" manner.

■ *Subroutine Register Management*

To help you save and restore the contents of registers, the 6502 provides four opcodes: PHA, PHP, PLA, and PLP. PHA and PHP are used to "push" the accumulator ("A") and processor status register ("P"), respectively, onto the stack. In the other direction, PLA and PLP "pull" the top value off of the stack and place it into the accumulator or processor status register. There are no special opcodes for the X and Y registers; to push their values, you must first transfer them into the accumulator (with TXA / TYA), and to restore them you must pull into the accumulator and then transfer again (with TAX / TAY).

Let's look at an example subroutine that uses these new opcodes:

```
.proc my_subroutine
  PHP
  PHA
  TXA
  PHA
  TYA
  PHA

  ; your actual subroutine code here

  PLA
  TAY
  PLA
  TAX
  PLA
  PLP
  RTS
.endproc
```

When my_subroutine is called (with JSR my_subroutine), the first six opcodes preserve the state of the registers on the stack before doing anything else. PHP, storing the state of the processor status register, comes first, because the processor status register is updated after every instruction - if we waited until the end to store P, it would likely be modified by the results of instructions like TXA. With the processor status register stored away on the stack, we next push the value of the accumulator, and then transfer and push the values of the X and Y registers. With everything stored on the stack, we are free to use all of the 6502's registers without worrying about what the code that called our subroutine expects to find in them. Once the subroutine code is finished, we reverse all of the storing we did at the beginning. We restore everything in the opposite order of how we stored it, first pulling and transferring to the Y and X registers, then the accumulator, and then the processor status register. Finally, we end with RTS, which returns program flow to the point where we called the subroutine.

If you forget to include RTS at the end of your subroutine, the 6502 will not return to where the subroutine was called and will instead happily continue with the next byte after your subroutine code. The processor doesn't know anything about .procs, they are simply a tool to help *you* organize your code.

■ *Your First Subroutine: Drawing the Player*

Now that you've seen how subroutines work, it's time to create your own. Let's write a subroutine that draws the player's ship at a given location. To do that, we'll need to use the player_x and player_y zero-page variables we created earlier to write the appropriate bytes to memory range $0200-$02ff. Previously, we did this by storing the appropriate bytes in RODATA and copying them with a loop and indexed addressing, the same way we did with palettes. As a quick refresher, we need to write four bytes of data for each 8 pixel by 8 pixel sprite tile: the sprite's Y position, tile number, special attributes / palette, and X position. The tile number and palette for each of the four sprites that make up the player ship will not change, so let's start there. We will also save and restore the system's registers at the start and end of our subroutine.

```
.proc draw_player
  ; save registers
  PHP
  PHA
  TXA
  PHA
  TYA
  PHA

  ; write player ship tile numbers
  LDA #$05
  STA $0201
  LDA #$06
  STA $0205
  LDA #$07
  STA $0209
  LDA #$08
  STA $020d

  ; write player ship tile attributes
  ; use palette 0
  LDA #$00
  STA $0202
  STA $0206
  STA $020a
  STA $020e
```

```
    ; restore registers and return
    PLA
    TAY
    PLA
    TAX
    PLA
    PLP
    RTS
  .endproc
```

The player ship uses tiles `$05` (top left), `$06` (top right), `$07` (bottom left), and `$08` (bottom right). We write those tile numbers to memory addresses `$0201`, `$0205`, `$0209`, and `$020d`, respectively, because those correspond to "byte 2" of the first four sprites. All of the player ship's tiles use palette zero (the first palette), so the code to write sprite attributes is much shorter. `$0202`, `$0206`, `$020a`, and `$020e` are the bytes immediately following the previous tile number bytes, and so they hold the attributes for each of the first four sprites. Finally, we restore all of the registers, in the opposite order of how we stored them, and use `RTS` to end the subroutine.

What about the *location* of each tile on screen? For that, we will need to use `player_x, player_y`, and some basic math. Let's assume, to make things easier, that `player_x` and `player_y` represent the top left corner of the top left tile of the player's ship. In our reset handler, we positioned the top left corner of the top left player ship tile at (`$70, $a0`). Once we have placed the top left tile, we can add eight pixels to `player_x` and `player_y` to find the positions of the other three tiles.

In the past, we have used `INC` / `DEC` to add or subtract. When adding or subtracting more than 1, however, there are more efficient opcodes. `ADC` ("ADd with Carry")

Here's what that looks like (previous code reduced to just comments):

```
  .proc draw_player
    ; save registers
    ; store tile numbers
    ; store attributes

    ; store tile locations
    ; top left tile:
    LDA player_y
    STA $0200
    LDA player_x
    STA $0203
```

```
    ; top right tile (x + 8):
    LDA player_y
    STA $0204
    LDA player_x
    CLC
    ADC #$08
    STA $0207

    ; bottom left tile (y + 8):
    LDA player_y
    CLC
    ADC #$08
    STA $0208
    LDA player_x
    STA $020b

    ; bottom right tile (x + 8, y + 8)
    LDA player_y
    CLC
    ADC #$08
    STA $020c
    LDA player_x
    CLC
    ADC #$08
    STA $020f

    ; restore registers and return
  .endproc
```

Remember that when you want to perform addition, first call `CLC`, then use `ADC` (unless you're trying to add something to a 16-bit number, which will be rare for now). The result of the addition can be found in the accumulator; it does not get written to `player_y` or `player_x`.

■ *Putting It All Together*

With our subroutine written, it's time to make use of it. We already set up the initial values of `player_x` and `player_y` in the reset handler. Now, we'll call our new subroutine as part of the NMI handler, so it runs every frame:

```
14  .proc nmi_handler
15    LDA #$00
16    STA OAMADDR
17    LDA #$02
18    STA OAMDMA
19
20    ; update tiles *after* DMA transfer
21    JSR draw_player
22
23    LDA #$00
24    STA $2005
25    STA $2005
26    RTI
27  .endproc
```

Notice that we perform a DMA transfer of whatever is already in memory range `$0200-$02ff` *before* calling our subroutine. The amount of time you have available to complete your NMI handler is very short, so putting your DMA transfer first ensures that at least *something* will be drawn to the screen each frame.

Finally, we need to update `player_x` each frame so that our sprites will actually move around the screen. For this example, we will keep `player_y` the same, but we will modify `player_x` so that the player's ship moves to the right until it is near the right edge of the screen and then moves left until it is near the left edge of the screen. To make this easier, we'll need to store what direction the player's ship is moving in. Let's add another zero page variable, `player_dir`. A `0` will indicate that the player's ship is moving left, and a `1` will indicate that the player's ship is moving right.

```
.segment "ZEROPAGE"
player_x: .res 1
player_y: .res 1
player_dir: .res 1
.exportzp player_x, player_y
```

I did not export `player_dir` because other files do not (yet) need to access it. Now we can write the code to update `player_x`. We could write out this code as part of the NMI handler directly, but in anticipation of more complicated player movement in the future, let's put it into its own subroutine, `update_player`:

```
.proc update_player
  PHP
  PHA
  TXA
  PHA
  TYA
  PHA

  LDA player_x
  CMP #$e0
  BCC not_at_right_edge
  ; if BCC is not taken, we are greater than $e0
  LDA #$00
  STA player_dir    ; start moving left
  JMP direction_set ; we already chose a direction,
                    ; so we can skip the left side check
not_at_right_edge:
  LDA player_x
  CMP #$10
  BCS direction_set
  ; if BCS not taken, we are less than $10
  LDA #$01
  STA player_dir   ; start moving right
direction_set:
  ; now, actually update player_x
  LDA player_dir
  CMP #$01
  BEQ move_right
  ; if player_dir minus $01 is not zero,
  ; that means player_dir was $00 and
  ; we need to move left
  DEC player_x
  JMP exit_subroutine
move_right:
  INC player_x
exit_subroutine:
  ; all done, clean up and return
  PLA
  TAY
  PLA
  TAX
  PLA
  PLP
  RTS
.endproc
```

This subroutine makes heavy use of the branching and comparison opcodes we saw in Chapter 11. We first load `player_x` into the accumulator and compare with `$e0`. `CMP`, as we learned earlier, subtracts its own operand from the accumulator, but only sets the carry and zero flags. We can use the resulting processor status register flags to tell us whether the value in the accumulator (in this case, `player_x`) was greater than, equal to, or less than `CMP`'s operand. `BCC not_at_right_edge` tells the 6502 to skip ahead to `not_at_right_edge` if the carry flag is cleared. When performing a subtraction as part of a comparison, the 6502 first sets the carry flag, and it is only cleared if the accumulator is smaller than `CMP`'s operand. In this case, if the accumulator is smaller than `$e0`, we know we are not near the right edge of the screen, so we can skip ahead to `not_at_right_edge`. If the accumulator is greater than `$e0`, the carry flag will still be set and the 6502 will continue with the next line. In that case, we are near the right edge of the screen, so we will need to update `player_dir` with a zero (to signify "moving left"). Then we use `JMP` to skip over the checks for whether or not we are near the left edge of the screen, because we already know that's not possible.

If the result of the first comparison was that `player_x` is not near the right edge of the screen, it's time to test if `player_x` is near the left edge of the screen. We compare `player_x` with `$10`, and this time we use `BCS direction_set`. `BCS`, as explained above, will activate if the accumulator (`player_x`) was larger than the comparison value (`$10`). In that case, we are not near the left edge and can skip forward to actually updating `player_x`. Otherwise, we need to update `player_dir` to be `$01`, indicating "move right".

Note that, with the way `update_player` is structured, if the player's ship is not near *either* edge of the screen, `player_dir` will not be updated, but we will still increment or decrement `player_x` as appropriate.

Finally, it's time to actually use the results of our edge tests. We compare `player_dir` with `$01` and look to see if the result is zero. If it is, `BEQ move_right` activates and we increment `player_x`. Otherwise, we decrement `player_x`. Having performed our update, we restore all of the registers and return from the subroutine.

Let's call our new subroutine inside of the NMI handler to finish off our example project:

```
    ; update tiles *after* DMA transfer
    JSR update_player
    JSR draw_player
```

All that's left is to assemble and link the files into a NES ROM:

```
ca65 src/spritemovement.asm
ca65 src/reset.asm
ld65 src/reset.o src/spritemovement.o -C nes.cfg -o spritemovement.nes
```

If you open the resulting `.nes` file in an emulator, you should see this:

■ *Homework*

Now that you understand the basics of moving sprites around the screen, try these projects to explore and deepen your understanding.

- Make the player ship move faster. Currently, we move the player's ship one pixel per frame, or 60 pixels per second. How would you make that movement faster?
- Change `player_y` instead of (or in addition to) `player_x`. Keep in mind that sprite Y positions greater than `$e0` will be below the bottom of the viewable screen area.
- What happens if, instead of checking the left and right edges of the screen, you just `INC player_x` (or `DEC player_x`) every frame? It's simple enough that you could do it directly in the NMI handler, without even touching `update_player`.
- Add additional sprites and move them separately from the player ship sprite.

To help you get started, you can download all of the code from this chapter (https://famicom.party/book/projects/14-spritemovement.zip).

15. Background Scrolling

We have covered drawing and moving sprites, but so far we have only *drawn* backgrounds. The NES has the ability to scroll backgrounds by as little as one pixel per frame in a smooth motion. In contrast, the Atari 2600's background scrolling was much less fluid, as seen in *Vanguard*:

https://youtu.be/ewQDKtc1i-4

Vanguard (SNK, 1982) was a side-scrolling shooter game originally released in arcades and later ported to the Atari 2600.

The NES' scrolling abilities come courtesy of a few special registers in the PPU and the way that backgrounds are laid out in memory. As a reminder, the NES PPU's memory map has space for four nametables, but only enough physical RAM to hold two. Those two nametables can have either a *vertical layout*, in which `$2000` and `$2800` are "real", or a *horizontal layout*, in which `$2000` and `$2400` are "real". Which layout is used depends on how the cartridge is manufactured. Older cartridges feature "V" and "H" solder pads on the cartridge board; whichever one has solder on it is the layout that will be used.

The "V"/"H" solder pads on the cartridge board for *Donkey Kong* (Nintendo, 1983). Here, the "V" pad is soldered, meaning that this cartridge uses a vertical nametable layout (also known as horizontal mirroring).

■ *Using* PPUSCROLL

In most cases, scrolling is controlled via writes to the `PPUSCROLL` (`$2005`) memory-mapped I/O address during Vblank (i.e., in your NMI handler). Your NMI handler should write to `PPUSCROLL` twice per frame. The first write will determine the X scroll position, in pixels, and the second write will determine the Y scroll position, also in pixels. Writes to `PPUSCROLL` should always occur at the *end* of the NMI handler (or at least after any writes to other PPU registers like `PPUADDR`), because the PPU uses the same internal registers for both memory access and scroll information, meaning writes to other PPU registers can change the scroll position.

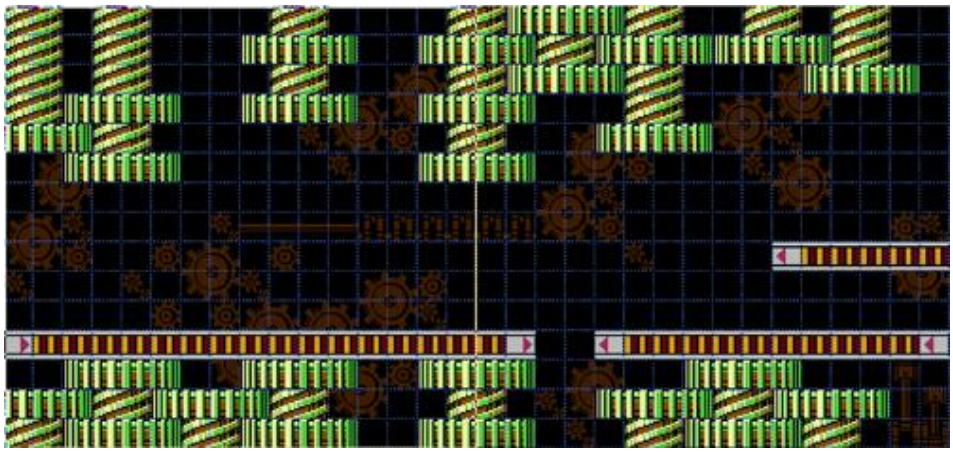
What are the "other" cases, where scrolling is handled differently? It is possible to create "split-screen" effects, where the top of the screen is at a different scroll position than the bottom, by carefully manipulating the internal working of the PPU. This requires writes to `PPUSCROLL` and `PPUCTRL` within a very short period of time, when the PPU has almost finished drawing the scanline before the split.

What the X and Y "scroll positions" mean varies based on the cartridge's nametable layout and what has been written to `PPUCTRL`. The PPU keeps track of a "current" or "base" nametable, which can be set via writes to `PPUCTRL`. The lowest two bits of a byte written to `PPUCTRL` set the base nametable, with `00` representing the nametable at `$2000`, `01` representing `$2400`, `10` representing `$2800`, and `11` representing `$2c00`. Once the base nametable has been set, the X and Y scroll positions are offsets from that base nametable.

Consider a standard horizontal-layout game. In this layout, there are two nametables: `$2000` and `$2400`. If we set the base nametable to `$2000`, and set both scroll positions to zero, the resulting background image displayed on screen will be the entirety of the nametable at `$2000`. Here is what that looks like:

```
; at end of NMI handler
LDA #$00
STA PPUSCROLL ; set X scroll
STA PPUSCROLL ; set Y scroll
```

Now let's say we want to move the "camera" twenty pixels to the right. With a horizontal layout, we would see all but the leftmost twenty pixels of the nametable at `$2000` lined up with the left edge of the screen, and the right side of the screen would be the leftmost twenty pixels of the nametable at `$2400`.



Side-by-side nametables in a horizontal layout, with the attribute grid visible in blue. As the player moves to the right, the X scroll position increases and the viewport "slides" across the two nametables. These are the first two screens of the Metal Man stage in *Mega Man 2* (Capcom, 1988).

In code, that looks like this:

```
; at end of NMI handler
LDA #20 ; decimal 20 because no "$"
STA PPUSCROLL ; set X scroll
LDA #$00
STA PPUSCROLL ; set Y scroll
```

What would this code do in a game that uses a vertical layout? When a scroll position is set so that the viewport would move beyond the bounds of the two "real" nametables, the viewport wraps around instead. However, most games do not use this functionality; generally, a game that uses a vertical layout will prevent horizontal scrolling and vice versa.

■ *Camera Systems*

Now that we have seen *how* to scroll the background, it's time to take a look at *when* to scroll the background. While it might feel natural for a character moving to the right to have the background scroll to the right with them, the NES leaves scrolling decisions entirely up to the programmer. Different games take very different approaches to the question of what should trigger a scroll. These approaches represent different *camera systems*, each with their own advantages and drawbacks. Let's look at a few basic techniques that were common on the NES.

The examples that follow are based on Itay Keren's seminal work on camera systems, "Scroll Back: The Theory and Practice of Cameras in Side-Scrollers", which goes into far greater detail than I will here. The terms used below (like "position-locking") are from the same article.

❖ Position Locking

The simplest camera system, *position locking* keeps the player in the same place on screen at all times, and scrolls the background every time the player "moves".

https://famicom.party/_app/immutable/assets/micromachines.4_nVjgU6.mp4

Position-locked camera in *Micro Machines* (CodeMasters, 1991). Here, the player vehicle is kept in the center of the screen at all times.

Position locking is useful when the player needs a consistent view distance around their character on screen. In *Micro Machines*, which mixes high speed racing with sudden turns, the center-locked camera gives the player a clear view of their surroundings at all times. Similarly, in side-scrolling games, position locking on the X axis ensures that the player can see a fixed distance to the left and right, so they will not be surprised by enemies coming from either side.

https://famicom.party/_app/immutable/assets/ducktales.XmPOUYHn.mp4

A horizontally position-locked camera in *Ducktales* (Capcom, 1989). Scrooge remains in the same X position at all times, but he can move vertically without the camera following him.

❖ Camera Windows

Position-locked cameras are extremely popular on the NES, but sometimes developers want to give the player more freedom to move around without constantly shifting the viewport. A *camera window* specifies a region of the screen in which the player can move without causing the screen to scroll; attempting to move outside of the window causes the screen to scroll instead.

https://famicom.party/_app/immutable/assets/crystalis.v7tQkrL-.mp4

A camera window in *Crystalis* (SNK, 1990). The camera window here is especially noticeable when the player character is moving left or right.

❖ Auto-scroll

Finally, the third camera system we will look at here is auto-scrolling. In an auto-scroll camera system, the player does not have any control over camera movement - the camera is constantly moving on its own, with the player either remaining in the same position while the background scrolls below or with the player sprites moving, by default, to match the movement of the background.

https://famicom.party/_app/immutable/assets/starsoldier.0K-9hYUp.mp4

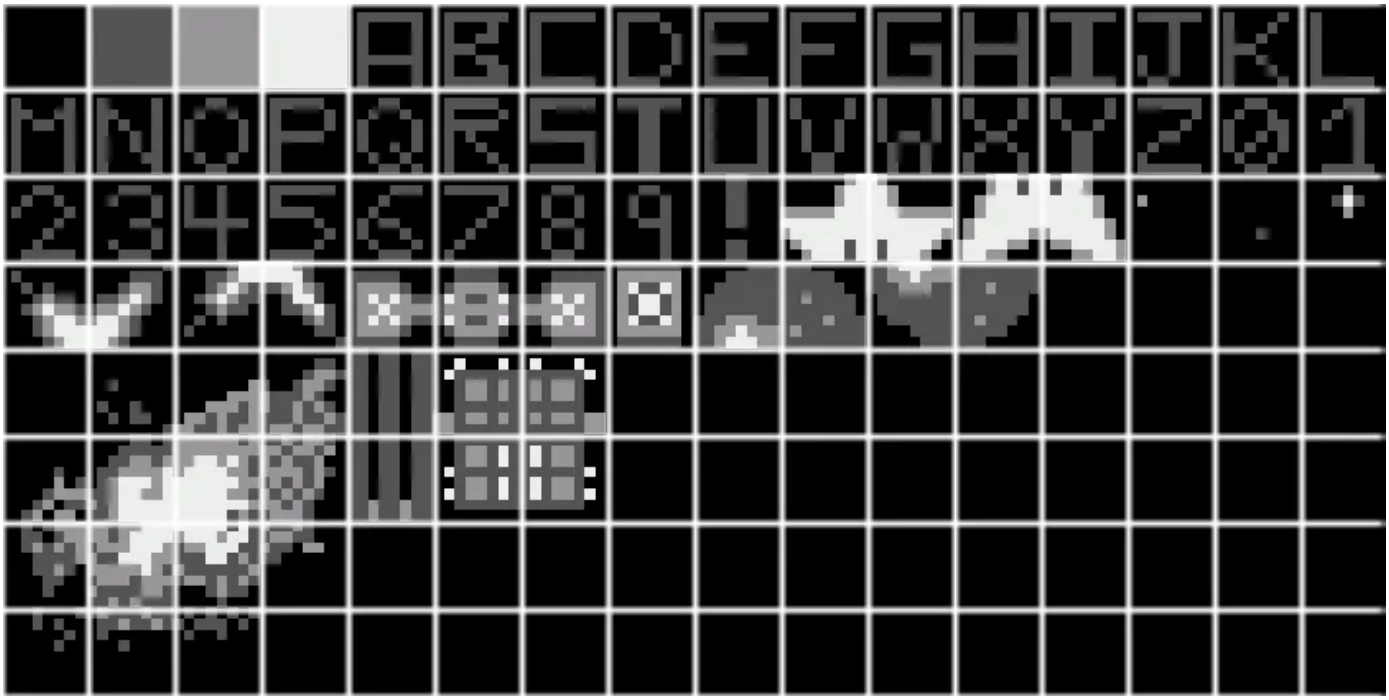
Auto-scroll in *Star Soldier* (Hudson Soft / Taxan, 1988). The player's ship is totally disconnected from the background starfield. If the player does not press any buttons, it remains in the same place on screen as where it was before.

https://famicom.party/_app/immutable/assets/smb3.QwSFlYDY.mp4

Auto-scroll in an airship level of *Super Mario Bros. 3* (Nintendo, 1990). Here, when the player is not pressing buttons on the controller, Mario moves along with the background. When the player reaches the left edge of the screen, they remain there but the background continues moving under them. Other games with auto-scrolling may count hitting the left edge of the screen as an instant death.

■ *Setting up backgrounds for scrolling*

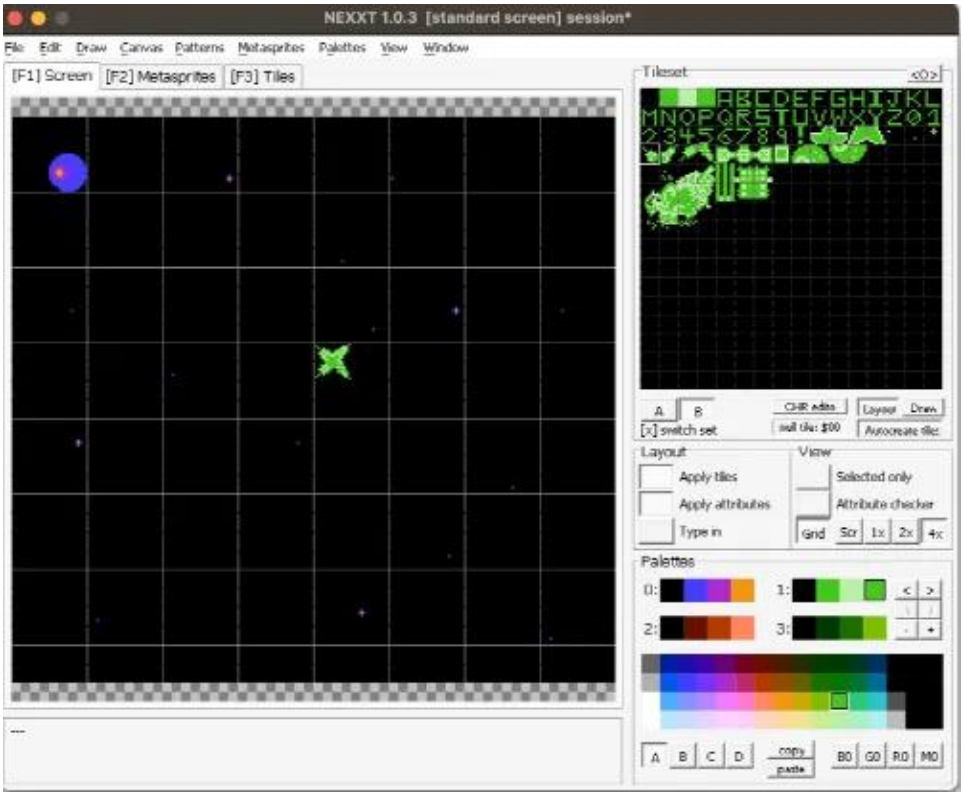
For our space shooter project, we will use a vertical layout with an auto-scrolling camera. To make our backgrounds more interesting, I've created some additional graphics tiles and put them into a new .chr file (`scrolling.chr`).



The new background tiles in `scrolling.chr`.

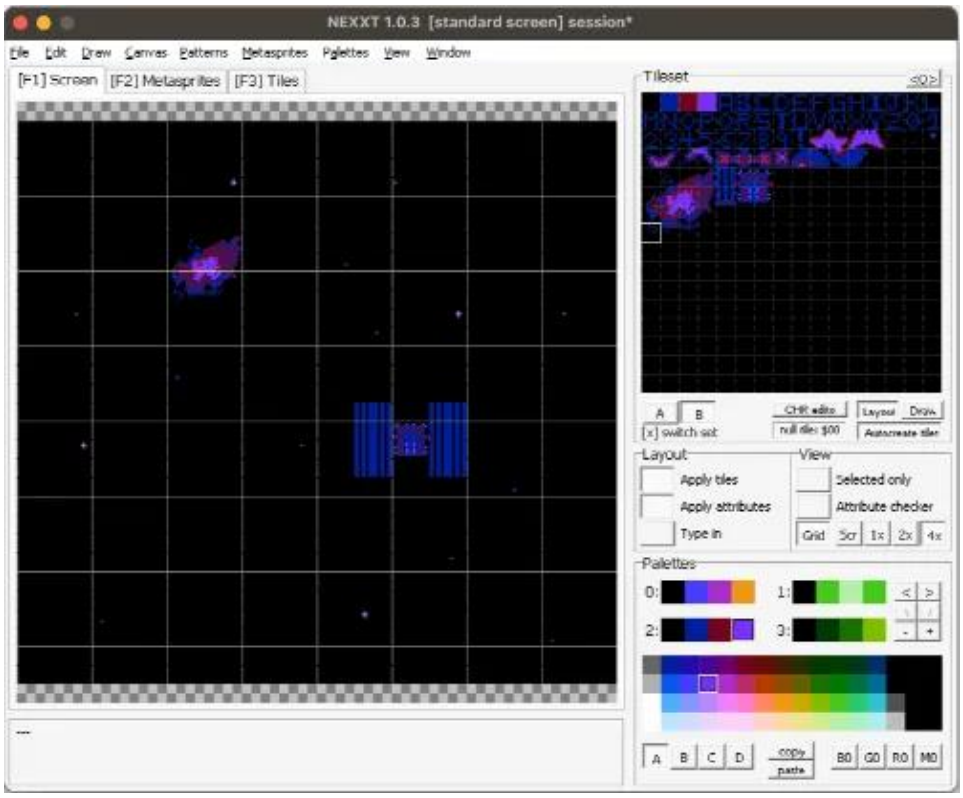
To finish out the chapter, we will set up two nametables worth of backgrounds in a horizontal mirroring arrangement, and then continuously scroll across them. We will start with the "bottom" nametable taking up the full screen, slowly scroll upwards until the "top" nametable takes up the full screen, and then wrap around to display the bottom nametable again.

Here is one of the nametables:



Minor additions to the nametable from last chapter, as seen in NEXXT.

And here is the other:



Another set of additions, as seen in NEXXT.

Both nametables use the same pattern of stars in the background, but with these new objects overlaid on top. To make things simpler, let's move our star-background-drawing code into a subroutine that we can call to draw stars for any of the four nametables, selected via whatever is stored in a register. I made a new file, `backgrounds.asm`, and added the following subroutine:

```
.include "constants.inc"

.segment "CODE"

.export draw_starfield
.proc draw_starfield
  ; X register stores high byte of nametable
  ; write nametables
  ; big stars first
  LDA PPUSTATUS
  TXA
  STA PPUADDR
  LDA #$6b
  STA PPUADDR
  LDY #$2f
  STY PPUDATA

  LDA PPUSTATUS
  TXA
  ADC #$01
  STA PPUADDR
  LDA #$57
  STA PPUADDR
  STY PPUDATA

; ...and much more, see the file for full listing

  ; finally, attribute table
  LDA PPUSTATUS
  LDA #$23
  STA PPUADDR
  LDA #$c2
  STA PPUADDR
  LDA #%01000000
  STA PPUDATA

  LDA PPUSTATUS
  LDA #$23
  STA PPUADDR
  LDA #$e0
  STA PPUADDR
  LDA #%00001100
  STA PPUDATA

  RTS
.endproc
```

To use this subroutine, first store the high byte of the nametable you want to draw stars to into the X register. In our case, since we are using horizontal mirroring, the two "real" nametables will be located at `$2000` and `$2800`, so the value we write to the X register will be either `#$20` or `#$28`. Notice that we use the Y register to load and store tile numbers, and the accumulator to write to PPUADDR. We need to keep that base nametable byte in the X register because we frequently need to load it into the accumulator (with `TXA`) and then add something to create the correct nametable tile address for each nametable. See, for example, lines 19-23 above. This particular byte of data needs to be written to either `$2157` or `$2957`, depending on which nametable it is in. By using `TXA` followed by `ADC #$01`, we get `$21` (if the X register had `$20`) or `$29` (if the X register had `$28`), giving us the correct value for both nametables.

Back in our main file, we can remove the existing "draw star background" code and replace it with two calls to our new subroutine:

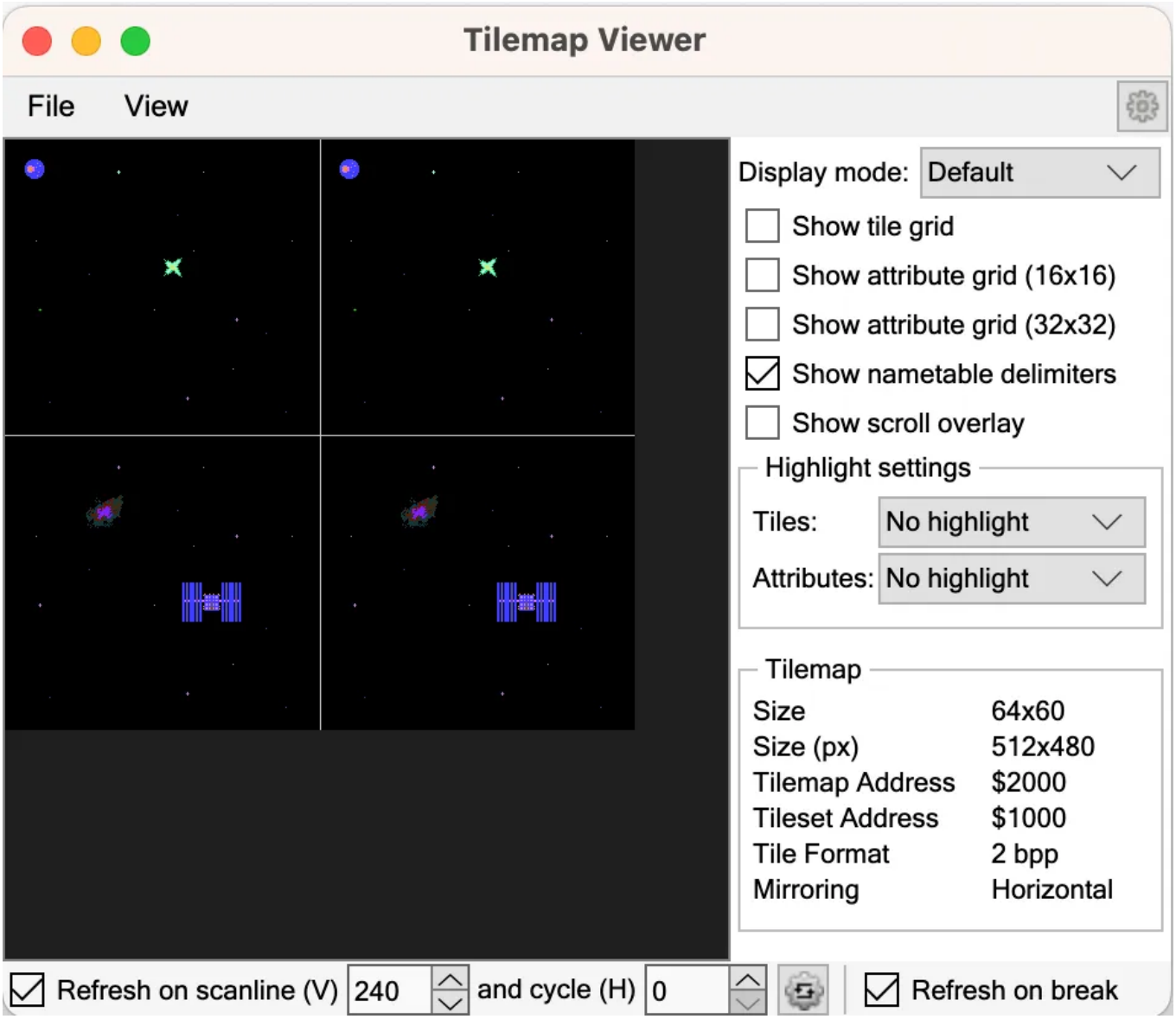
```
.import draw_starfield

; inside .proc main, after writing palettes:
  ; write nametables
  LDX #$20
  JSR draw_starfield

  LDX #$28
  JSR draw_starfield
```

The result, if you check the "Nametables" debug helper in Nintaco, is that both nametables will have a starfield background. Next, we'll need to add some new objects on top of the starfield. I chose to do this in another subroutine inside `backgrounds.asm`, this time called `draw_objects`:

```
152 .export draw_objects
153 .proc draw_objects
154   ; Draw objects on top of the starfield,
155   ; and update attribute tables
156
157   ; new additions: galaxy and planet
158   LDA PPUSTATUS
159   LDA #$21
160   STA PPUADDR
161   LDA #$90
162   STA PPUADDR
163   LDX #$30
164   STX PPUDATA
165   LDX #$31
166   STX PPUDATA
167
```

```
168     LDA PPUSTATUS
169     LDA #$21
170     STA PPUADDR
171     LDA #$b0
172     STA PPUADDR
173     LDX #$32
174     STX PPUDATA
175     LDX #$33
176     STX PPUDATA
177
178   ; ...and more, not listed here
179
180     ; finally, attribute tables
181     LDA PPUSTATUS
182     LDA #$23
183     STA PPUADDR
184     LDA #$dc
185     STA PPUADDR
186     LDA #%00000001
187     STA PPUDATA
188
189     LDA PPUSTATUS
190     LDA #$2b
191     STA PPUADDR
192     LDA #$ca
193     STA PPUADDR
194     LDA #%10100000
195     STA PPUDATA
196
197     LDA PPUSTATUS
198     LDA #$2b
199     STA PPUADDR
200     LDA #$d2
201     STA PPUADDR
202     LDA #%00001010
203     STA PPUDATA
204
205     RTS
206   .endproc
```

This subroutine does not collect any information from the X register (nor any other); the addresses are hard-coded. Calling this subroutine (with `JSR draw_objects` following our two calls to `draw_starfield`) will complete our two nametables.



The completed nametables as seen in Nintaco's "Nametables" viewer. Note that the left and right halves are identical - this is because we are using horizontal mirroring. If we were to scroll to the left or right, we would move into that mirrored area and the screen would appear to "wrap around".

■ *Implementing autoscroll*

As discussed previously, scroll positions get set in the NMI handler via two writes to `PPUSCROLL`. The first write sets the X scroll amount (how many pixels to the right the camera window should move) and the second write sets the Y scroll amount (how many pixels up the camera window should move). We want our game to have the background continuously scroll downward, so the X scroll amount will never change (always zero) and the Y scroll amount will start at the maximum value and decrease every frame. Remember that while an NES screen is 256 pixels wide, it is only 240 pixels tall. That means we will need to start our Y scroll at 239 (the maximum value), and when we need to scroll "less than zero", we will need to set it back to 239 instead of doing the normal wrap around to 255.

To accomplish this, we will need two additional zero-page variables. The first, `scroll`, will store the current Y scroll amount. The second, `ppuctrl_settings`, will

store the current settings that were sent to PPUCTRL, so that we can change the base nametable every time we hit scroll position 0.

```
4  .segment "ZEROPAGE"
5  player_x: .res 1
6  player_y: .res 1
7  player_dir: .res 1
8  scroll: .res 1
9  ppuctrl_settings: .res 1
10 .exportzp player_x, player_y
```

Let's also add a new constant to `constants.inc`, before we forget:

```
PPUSCROLL = $2005
```

We will need to make two changes to our `main` code to set things up. First, we'll need to set an initial value for `scroll`:

```
55 .proc main
56   LDA #239   ; Y is only 240 lines tall!
57   STA scroll
```

Second, we set PPUCTRL after drawing all of our nametables. We'll need to store the value that we send to PPUCTRL so that we can modify and re-use it later:

```
85   LDA #%10010000  ; turn on NMIs, sprites use first pattern table
86   STA ppuctrl_settings
87   STA PPUCTRL
```

The lowest (right-most) two bits set which nametable is the "base" that scroll offsets are applied to. In this case, those two bits are `00`, indicating the nametable at `$2000`. We will need to toggle between `$2000` (`00`) and `$2800` (`10`) as the base nametable in order to have smooth scrolling. If we did not change the base nametable, you would see a smooth scroll from one nametable to the other, but then the scroll would immediately snap back to the initial nametable instead of continuing to scroll smoothly.

With everything set up, let's take a look at the NMI handler, where the actual scroll positions will be set. I'll present the code here first and then explain it.

```
17 .proc nmi_handler
18   LDA #$00
19   STA OAMADDR
20   LDA #$02
21   STA OAMDMA
22   LDA #$00
23
24   ; update tiles *after* DMA transfer
25   JSR update_player
26   JSR draw_player
27
```

```
28   LDA scroll
29   CMP #$00 ; did we scroll to the end of a nametable?
30   BNE set_scroll_positions
31   ; if yes,
32   ; update base nametable
33   LDA ppuctrl_settings
34   EOR #%00000010 ; flip bit #1 to its opposite
35   STA ppuctrl_settings
36   STA PPUCTRL
37   LDA #240
38   STA scroll
39
40 set_scroll_positions:
41   LDA #$00 ; X scroll first
42   STA PPUSCROLL
43   DEC scroll
44   LDA scroll ; then Y scroll
45   STA PPUSCROLL
46
47   RTI
48 .endproc
```

The top part of this code is unchanged from the previous example. Our new scroll code begins at line 28. Note that you must *always* set scroll positions at the end of the NMI handler, right before `RTI`. If you set scroll positions earlier, other writes to PPU memory can interfere with how the PPU calculates scroll positions, leading to unexpected behavior.For a full explanation of how the PPU calculates scroll positions, including a discussion of the PPU's internal registers, see the NESDev Wiki page on PPU scrolling.

The first thing that our new code does is check the current scroll position (`scroll`) against zero. If it is zero, that means we are about to wrap to a new nametable, and we will need to change the base nametable. `BNE set_scroll_positions` skips the `PPUCTRL`-setting code if `scroll` is not zero. Assuming that we do need to change the base nametable, we load the saved PPUCTRL settings and use a logical filter (`EOR`) to flip a bit to its opposite.

■ *Logical Filters*

Before continuing on, let's take some time to learn about the three logical filter opcodes that the 6502 offers: `AND`, `ORA`, and `EOR`. Each opcode takes one byte as its operand, compares each bit of its operand with the corresponding bit of the accumulator, and changes individual bits of the value stored in the accumulator. Because logical filters make bit-by-bit comparisons, logical filter operands are usually expressed in binary.

`AND` compares each bit of its operand with the corresponding bit of the accumulator, and sets that bit of the accumulator to `1` if *both* bits were `1` to begin with.

Otherwise, it sets that bit of the accumulator to 0. Here is an example:

```
    LDA #%10101010
    AND #%00001111
```

After running the above code, the value in the accumulator will be #%00001010. Bits one and three are set to 1 because for those bits, both the initial value in the accumulator and the value in the AND operand were 1. All other bits of the accumulator were set to zero.

ORA ("OR with Accumulator") compares bit-by-bit like AND, but it sets bits of the accumulator to 1 if *either* the accumulator bit or the operand bit are 1. Accumulator bits are only set to zero if *both* accumulator and operand bits are zero. Here is an example:

```
    LDA #%10101010
    ORA #%00001111
```

After running the above code, the value in the accumulator will be #%10101111.

The final logical filter is EOR ("Exclusive OR", more commonly known as "XOR"). In logic, XOR returns "true" if either, but not both, of its inputs are true, and "false" otherwise. When EOR compares bits, if the bit from EOR's operand is 0, the corresponding accumulator bit is not changed. If the operand bit is 1, the corresponding accumulator bit is flipped to its opposite value. Here is an example:

```
    LDA #%10101010
    EOR #%00001111
```

After running the above code, the value in the accumulator will be #%10100101. The highest four bits (1010) were unchanged, because EOR's operand had zeroes there. The lowest four bits were flipped to 0101, because EOR's operand had ones there.

The three logical filters allow you to make precision edits to the bits in any byte. AND allows you to filter out only the specific bits you care about, with 0 bits in its operand setting accumulator bits to zero and 1 bits allowing values from the accumulator to pass through unchanged. ORA allows you to turn bits on by using a 1 in its operand. EOR allows you to flip specific bits in the accumulator to their opposites by using a 1 bit in the operand. They take some time to learn, but they are incredibly useful and we will see them repeatedly going forward.

■ *Wrapping up*

To return to our previous discussion, when we need to update the base nametable, we load ppuctrl_settings into the accumulator followed by EOR #%00000010. This leaves most of the accumulator value as-is, but flips bit 1 (second from the right) to its opposite. Bits 1 and 0 of the value we send to PPUCTRL control the base nametable, with 00 representing the nametable at $2000, 01 for $2400, 10 for $2800, and 11 for $2c00. By changing a single bit, we can flip between $2000 and $2800 as the base nametable. We then write the new value back to ppuctrl_settings, and write it to

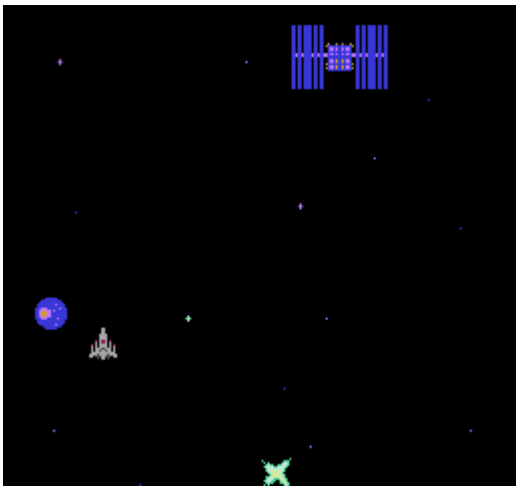
`PPUCTRL` as well. After changing the base nametable, we reset `scroll` to 240 (not 239, because we are about to decrement it in the code that follows).

With the base nametable set, we can set the actual scroll positions. We first write a zero to `PPUSCROLL` to set the X scroll amount. Then, `DEC scroll` subtracts one from `scroll` and stores the result back into `scroll`. We load `scroll` into the accumulator and then write it to `PPUSCROLL` to set the Y scroll amount. Having set both X and Y scroll amounts, we are done with the NMI handler and can call `RTI` to go back to main code.

Let's build and run our new project:

```
ca65 src/backgrounds.asm
ca65 src/scrolling.asm
ld65 src/backgrounds.o src/scrolling.o -C nes.cfg -o scrolling.nes
```

Here is our project running in an emulator:



■ *Homework*

Using the code from this chapter (https://famicom.party/book/projects/15-scrolling.zip), try the following exercises:

- Modify the backgrounds to display something different. There are several additional tiles in the CHR file for this project, experiment with different combinations and palettes.
- Change the speed of the background auto-scroll. How would you make the scroll twice as fast? What about half as fast?
- Reverse the direction of the auto-scroll.

Part III: Gameplay

16. Controller Input

With the basics of NES graphics covered, let's turn our attention to reading player input from the controller(s). In this chapter, we'll look at the history of input devices for the system, how to read the state of controller buttons, and how to use controller input to create actual gameplay.

■ *A History of Controllers*

The most successful home console prior to the NES was the Atari 2600 (or "Video Computer System" / VCS. The 2600's controller was a joystick with a single button, which meant players only had a limited ability to provide input.

The Atari 2600 joystick.

In place of a joystick, the Famicom/NES controllers use a directional pad (or "D-pad"), shaped like a plus sign. This setup, invented by Gunpei Yokoi, first made an appearance in Nintendo's "Game & Watch Multiscreen" Donkey Kong system in 1982.

https://youtu.be/t6HykFfFM1s?si=XScRsm0oQf84erPl

Japanese commercial for the Game & Watch Dual Screen

The “Game & Watch Multiscreen” edition of Donkey Kong.

Yokoi would also go on to produce Kid Icarus and Metroid for the NES, as well as designing the Virtual Boy. His practice of “lateral thinking with withered technology” continues to be the foundation of Nintendo’s designs.

The Famicom controllers, released only one year later in 1983, were an evolution of the Multiscreen design. The two controllers each feature a D-pad, two action buttons (labelled “A” and “B”), and two supplemental buttons (“Select” and “Start”). On the Famicom, the controllers are clearly labelled “I” and “II”, and player two’s controller features a microphone with its own volume slider.

A "player 2" controller from the Famicom, featuring microphone controls and a lack of Select and Start buttons.

Why did the Player 2 controller feature a microphone? At the time of the Famicom's release, karaoke had become very popular in Japan. The microphone would allow developers to create karaoke games for the Famicom, with the player's voice projected through their TV speakers. I'm not aware of any Famicom games that actually used that functionality, and the microphone was dropped when the controllers were revised for the NES. Some Famicom games do use the microphone as a sort of additional input by having the player scream into it at certain points. This functionality has been preserved in Mesen2, which allows you to set a key for microphone input when the Player 2 controller is set to "Famicom controller".

On the Famicom, the two controllers were hard-wired into the console. To facilitate the addition of future input methods, the Famicom console also had a front-facing, 15-pin expansion port, normally hidden behind a red plastic cover. This expansion port was where the Famicom's version of the "Zapper" light gun connected, as well as multiplayer accessories.

A close-up picture of the Famicom expansion port. Photo from NintendoWorldReport.

A schematic showing how the Famicom light gun attaches to the console, via the expansion port. The Famicom light gun has a more realistic, "Wild West" revolver style than the comparitively futuristic (and bright orange) US "Zapper" light gun.

When the Famicom was re-designed for its US release as the NES, the controllers underwent some minor, but significant, changes. The microphone was dropped, making the “Player 1” and “Player 2” controllers identical. Additionally, the controllers were now detachable, and connected to the console via a specially-designed plug. This made using alternative controllers (such as the Zapper) much easier, since players could simply disconnect a standard controller and plug in something else in its place.

The original NES controller.

Photo by Evan Amos.

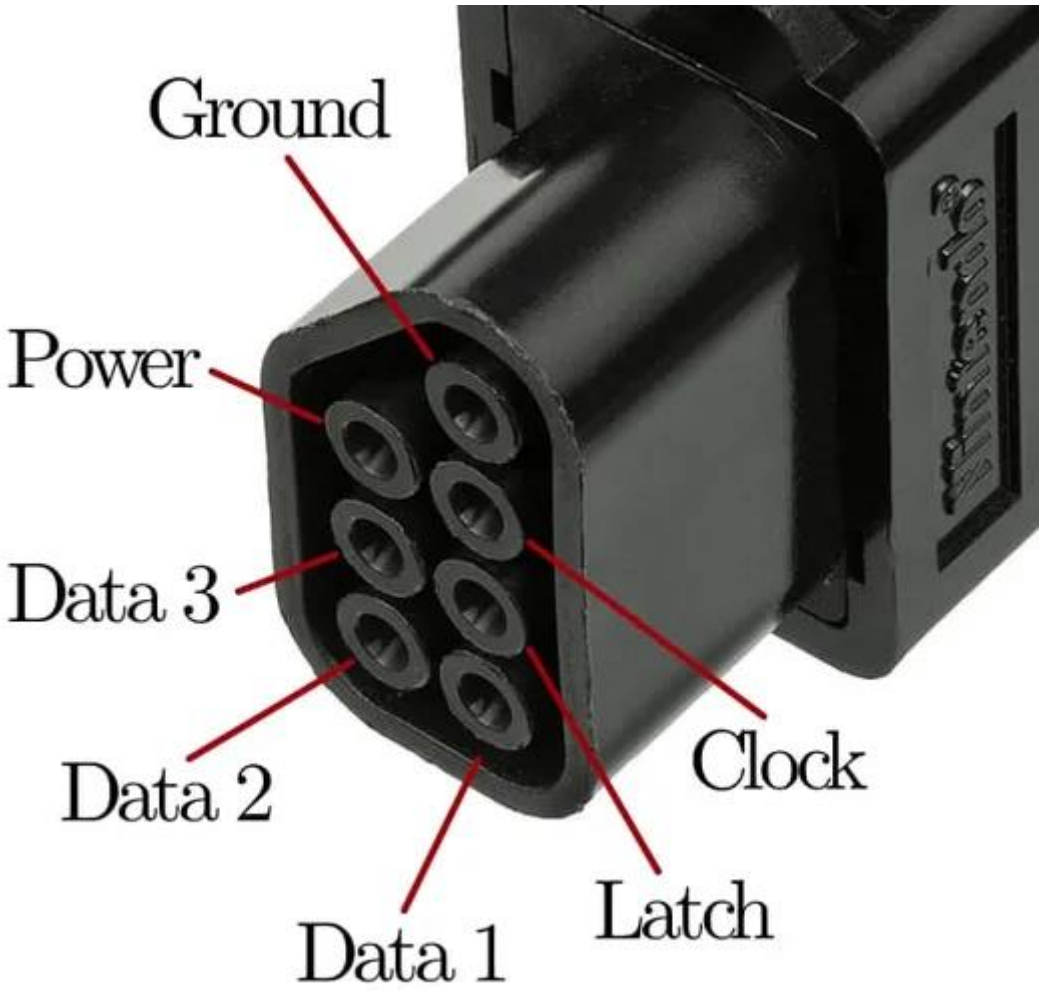
Nintendo (and its hardware partners) created several alternative controllers over the lifespan of the system. The NES Advantage was an arcade-style joystick, with the ability to set the A or B buttons to “Turbo” (automatic rapid button presses when held down), as well as a “slow motion” feature (in reality, a “Turbo” setting for the start button). The NES Max controller resembled more modern controller designs, with “wings” for better ergonomics and a sliding directional pad that looks similar to an analog stick, but is still digital input. Later controller releases included the NES Satellite and NES Four Score, each of which allowed four controllers to connect to the system at the same time and enabled simultaneous four-player gameplay in games that were coded to take advantage of it.

A collection of non-standard NES controllers. Clockwise, from top left: NES Advantage, NES Max, NES Satellite, NES Four Score.

Perhaps the most unusual alternative controller, and the one which likely did more to cement Nintendo's success in the US than anything else, was R.O.B., the "Robotic Operating Buddy". Launched as part of the NES "Deluxe Set" in 1985, R.O.B. was a battery-powered "robot" with photodiodes for "eyes", which could watch the TV for specially-coded flashes of light and move in response to them. Only two licensed games were ever released to take advantage of R.O.B.: Gyromite, a pack-in game in the Deluxe Set, and Stack-Up, also released in 1985. While R.O.B. was a failure in terms of driving development of new games, it was a cool-looking (for the time) robot that enticed many players/parents to buy the Deluxe Set.

■ *Controller Hardware*

An NES controller has eight inputs: four on the directional pad (Up, Down, Left, Right), and four actual buttons (Select, Start, A, B). Each input can be in one of two states: pressed, or not pressed. This sounds like the perfect opportunity to store controller state in a byte of memory, with one bit for the state of each button. From an electrical standpoint, though, sending eight bits of data at a time would require eight wires. The NES controller connector only has seven pins, making this strategy impossible.



An annotated photo of an NES controller plug. There are seven pins, only three of which are used for data. In general, only one data line is used at a time, with "Data 1" used for standard NES controllers and Data 2 / Data 3 being used by the Zapper.

Instead of sending all eight button states in parallel, the NES controller uses a "parallel-in, serial-out shift register". The controller has one eight-bit register inside of itself. When the controller receives a signal from the console on the "Latch" wire, it begins to continuously fill the register with the current state of the buttons. Turning off the Latch signal causes the shift register to go back to serial mode, saving the last button states before the signal was turned off. Then, when the controller receives a signal on the "Clock" wire, it outputs the state of a single button at a time (in a specific order) on the appropriate Data wire. So, to read the state of all buttons on the controller, your code must:

- Send a "1" to the controller latch pin
- Send a "0" to the controller latch pin
- Send eight Clock signals to the controller, and listen for eight bits of data over the Data pin

There are, as you might expect, special memory-mapped I/O addresses for performing these actions. Memory address `$4016` sets the latch state for both Controller 1 and Controller 2. Once the controllers' shift registers are filled, reading from `$4016` will return one bit of data (one button state) from Controller 1, and reading from `$4017` will return one bit of data from Controller 2.

On the NES Satellite or NES Four Score, the first eight reads of `$4016` return the button states for Controller 1, and the next eight reads return the button states for Controller 3. A final set of eight reads returns a "signature" pattern of bits, which allows the game to determine whether or not a four-player adapter is connected to the console. A similar process is used with `$4017` to read the states of Controller 2 and Controller 4.

To convert the flow above to assembly code, we first write a "1" to `$4016`, then write a "0" to the same address.

```
    LDA #$01
    STA $4016
    LDA #$00
    STA $4016
```

(Note: as with other MMIO addresses, it's common to set constant names for `$4016` and `$4017`. Later, I will use `CONTROLLER1` and `CONTROLLER2`.)

Reads from `$4016` each return a single button state, with bit 0 set if the button is pressed and cleared if it is not (all other bits are cleared). The button states are returned in this order: A, B, Select, Start, Up, Down, Left, Right. Storing these eight button states to eight separate zero-page addresses would be very inefficient, but thankfully, there is a simple technique that can store those eight button states in a single byte. In order to use it, though, we will need to learn a few more 6502 opcodes.

■ *Bit Shifts and Rotations*

A bit shift moves the bits within a byte left or right. The 6502 has two opcodes that can shift bits: ASL (Arithmetic Shift Left) and LSR (Logical Shift Right). ASL moves all bits in a byte one position to the left, dropping the leftmost bit (bit 7) into the carry flag and adding a zero to replace the now-empty rightmost bit (bit 0). LSR does the opposite, moving all bits in a byte one position to the right, dropping the rightmost bit into the carry flag and setting the leftmost bit to zero. Because of how binary numbers work, performing a left shift is the same as multiplying by two (so long as the result fits within a single byte), and a right shift is the same as dividing by two and rounding down.

The rotation opcodes (ROL "ROtate Left" and ROR "ROtate Right") shift bits just like ASL and LSR, but rather than filling empty bits with zeroes, they move whatever is stored in the carry flag into the empty bit position. When using a rotation opcode, the contents of the carry flag are shifted into the byte before one of the byte's bits are shifted into the carry flag.

Let's look at a few examples:

```
; Our starting byte - equivalent to decimal 15
LDA #%00001111
STA $8000

; Shift left.
ASL $8000
; Memory address $8000 now contains 00011110,
; equivalent to decimal 30.
; The carry flag contains 0, because
; that was the left-most bit.

; Shift back to the right.
LSR $8000
; Memory address $8000 is now back to 00001111.
; The carry flag still contains 0.

; Shift right again.
LSR $8000
; Memory address $8000 now contains 00000111,
; equivalent to decimal 7.
; Note that the carry flag is now 1 - when
; the rightmost bit was shifted right, it went
; into the carry flag.
```

```
    ; This time, let's rotate right.
    ROR $8000
    ; Memory address $8000 now contains 10000011,
    ; and the carry flag contains 1 again.
    ; What happened?
    ; The "1" from the carry flag moved into the
    ; leftmost bit position, and the "1" in the
    ; rightmost bit position dropped off into the
    ; carry flag.

    ; Let's do that a few more times:
    ROR $8000
    ; Memory address $8000: 11000001, carry flag: 1
    ROR $8000
    ; Memory address $8000: 11100000, carry flag: 1
    ROR $8000
    ; Memory address $8000: 11110000, carry flag: 0
    ROR $8000
    ; Memory address $8000: 01111000, carry flag: 0

    ; We can also shift or rotate the accumulator directly:
    LDA $8000
    ROL A
    LSR A
    ; The results of the rotate and shift are only in
    ; the accumulator, not stored back into $8000.
```

❖ Ring Counters

Now that we've looked at shifts and rotations, let's put them to use to store controller data in a single byte. Remember, asking for a button state (reading from `$4016`) returns the state of a button in bit 0, with the bit set if the button is pressed or cleared if the button is not pressed. A ring counter makes use of rotations to run a loop exactly eight times, transferring the results of eight reads from `$4016` into a single byte.

To set up the ring counter, we'll first need a byte of memory to store our controller data. Since controller data is updated frequently, a byte of zero-page RAM is ideal.

```
pad1: .res 1
```

Next, we will set the initial state of `pad1` to the byte `00000001`.

```
  LDA #%00000001
  STA pad1
```

Each time we read a button state from the controller, we will use shift and rotation opcodes to first transfer the bit that represents the button state into the carry flag, and then rotate it onto `pad1`. When the "1" from bit 0 rotates all the way left and falls off into the carry flag, we know that we have transferred eight button states and we can end the loop (by checking at the end of each loop iteration against `BCC`, Branch if Carry Clear).

Here's a look at the full code:

```
  ; write a "1", then a "0", to CONTROLLER1 ($4016)
  ; in order to lock in button states
  LDA #$01
  STA CONTROLLER1
  LDA #$00
  STA CONTROLLER1

  ; initialize pad1 to 00000001
  LDA #%00000001
  STA pad1

get_button_states:
  LDA CONTROLLER1       ; Get the next button state
  LSR A                 ; Shift the accumulator right one bit,
                        ; dropping the button state from bit 0
                        ; into the carry flag
  ROL pad1              ; Shift everything in pad1 left one bit,
                        ; moving the carry flag into bit 0
                        ; (because rotation) and bit 7
                        ; of pad1 into the carry flag
  BCC get_button_states ; If the carry flag is still 0,
                        ; continue the loop. If the "1"
                        ; that we started with drops into
                        ; the carry flag, we are done.
```

At the end of this loop, the eight bits of `pad1` will contain the state of all eight buttons on the controller, as follows:

Bit	Button
0	Right
1	Left
2	Down
3	Up
4	Start
5	Select
6	B
7	A

To capture the state of player 2's controller buttons, use the same ring counter, but substitute `CONTROLLER2` (`$4017`) for `CONTROLLER1`, and designate a second byte of zero-page RAM for storing button states (`pad2`).

■ *Using Controller Data*

Once you have captured the state of the controller's buttons, the next step is making use of that data in your game code. To do so, we can use the logical filters introduced in the last chapter to test whether or not the buttons we care about are set, and then branch based on the zero flag. To make that testing easier, I like to set constants for each button's position in `pad1`:

```
BTN_RIGHT   = %00000001
BTN_LEFT    = %00000010
BTN_DOWN    = %00000100
BTN_UP      = %00001000
BTN_START   = %00010000
BTN_SELECT  = %00100000
BTN_B       = %01000000
BTN_A       = %10000000
```

Once you have these constants, the checks themselves are fairly simple. Here is how we would test whether or not the start button is pressed:

```
  LDA pad1       ; Load button states into accumulator
  AND #BTN_START ; Must use "#"! Not a memory address!
  BNE start_pressed ; Branch to code you want to
                    ; run when start is pressed
```

As a quick refresher, `AND` lets you selectively filter out bits from the accumulator. Any bits that are "0" in `AND`'s operand will be set to zero in the accumulator; any bits that are "1" in `AND`'s operand will stay as they are in the accumulator. By using `00010000` as our operand for `AND`, we ensure that all bits except the bit that represents the state of the start button will be zero. That way, if the start button is not pressed, the result of our `AND` will be zero, regardless of how many other buttons on the controller are pressed.

Let's apply this to the game we've been building. Instead of having the player's ship automatically move left and right, we will instead read the controller and move the player's ship appropriately. We already isolated our player-updating code into its own subroutine (`.proc update_player`), which should make things a bit simpler. We will need to make the following changes from the last chapter:

- Read the state of the controller in NMI
- Update our `update_player` subroutine to test for different button presses and move the sprites accordingly

Let's start with reading the controller. We will make a new subroutine for the ring counter and call it from NMI. First, though, we will need some new constants in our `constants.inc`:

```
10  CONTROLLER1 = $4016
11  CONTROLLER2 = $4017
12
13  BTN_RIGHT   = %00000001
14  BTN_LEFT    = %00000010
15  BTN_DOWN    = %00000100
16  BTN_UP      = %00001000
17  BTN_START   = %00010000
18  BTN_SELECT  = %00100000
19  BTN_B       = %01000000
20  BTN_A       = %10000000
```

With that out of the way, let's create a new file for controller-related subroutines. This will make it possible to re-use the same controller code in multiple projects. Inside the file (I'm calling it `controllers.asm`), we'll write a subroutine that uses the ring counter technique we saw earlier:

```
1   .include "constants.inc"
2
3   .segment "ZEROPAGE"
4   .importzp pad1
5
6   .segment "CODE"
7   .export read_controller1
8   .proc read_controller1
9     PHA
10    TXA
11    PHA
12    PHP
13
```

```
14    ; write a 1, then a 0, to CONTROLLER1
15    ; to latch button states
16    LDA #$01
17    STA CONTROLLER1
18    LDA #$00
19    STA CONTROLLER1
20
21    LDA #%00000001
22    STA pad1
23
24  get_buttons:
25    LDA CONTROLLER1 ; Read next button's state
26    LSR A           ; Shift button state right, into carry flag
27    ROL pad1        ; Rotate button state from carry flag
28                    ; onto right side of pad1
29                    ; and leftmost 0 of pad1 into carry flag
30    BCC get_buttons ; Continue until original "1" is in carry flag
31
32    PLP
33    PLA
34    TAX
35    PLA
36    RTS
37  .endproc
```

One thing to note is the `.importzp pad1` on line 4 - we will need to make sure that we reserve a byte in zero-page with that name and export it with `.exportzp`. Here is the updated `ZEROPAGE` segment in our main file (now called `input.asm`):

```
4     .segment "ZEROPAGE"
5     player_x: .res 1
6     player_y: .res 1
7     scroll: .res 1
8     ppuctrl_settings: .res 1
9     pad1: .res 1
10    .exportzp player_x, player_y, pad1
```

We have a fair number of things that we are keeping track of, but we are only using five of the 256 zero-page addresses available to us. Even so, it's still important to use zero-page only for things that will be updating frequently, since later additions like audio or tracking a large number of enemies can take up a big chunk of zero-page addresses.

Next, let's update the NMI handler to read controller state once per frame:

```
18  .import read_controller1
19
20  .proc nmi_handler
21    LDA #$00
22    STA OAMADDR
23    LDA #$02
24    STA OAMDMA
25    LDA #$00
26
27    ; read controller
28    JSR read_controller1
```

Remember that you need to import any subroutines you want to use that are exported in a different file (line 18).

Everything is in place to work with controller data. Now, it's time to update the `update_player` subroutine. Our logic for updating the player position will work as follows (the exact order of checking directions is arbitrary):

- Check if the player pressed Left. If so, decrement `player_x`.
- Check if the player pressed Right. If so, increment `player_x`.
- Check if the player pressed Up. If so, decrement `player_y`.
- Check if the player pressed Down. If so, increment `player_y`.

While virtually all of this subroutine will be replaced, a portion of it that might still be useful is the code to detect collisions with the edge of the screen. To keep the logic simpler, this chapter will only focus on using controller input, but it's a good homework exercise to re-implement bounds checking on your own, essentially ignoring controller presses that would put the ship outside of a certain area.

This setup gives us a one-to-one mapping between controller presses and movement on screen, which will have a "stiff" feel. In a later chapter, we'll add rudimentary physics, but this simpler approach will let us focus on the controller.

The updated `update_player` subroutine is as follows:

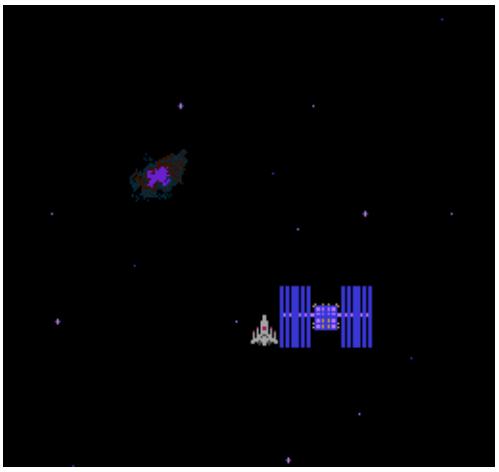
```
.proc update_player
  PHP  ; Start by saving registers,
  PHA  ; as usual.
  TXA
  PHA
  TYA
  PHA

  LDA pad1        ; Load button presses
  AND #BTN_LEFT   ; Filter out all but Left
  BEQ check_right ; If result is zero, left not pressed
  DEC player_x  ; If the branch is not taken, move player left
check_right:
  LDA pad1
  AND #BTN_RIGHT
  BEQ check_up
  INC player_x
check_up:
  LDA pad1
  AND #BTN_UP
  BEQ check_down
  DEC player_y
check_down:
  LDA pad1
  AND #BTN_DOWN
  BEQ done_checking
  INC player_y
done_checking:
  PLA ; Done with updates, restore registers
  TAY ; and return to where we called this
  PLA
  TAX
  PLA
  PLP
  RTS
.endproc
```

That rounds out the updates to the code. Let's build everything and verify that it works as expected, by running the following commands in the top-level directory for the project.

```
ca65 src/backgrounds.asm
ca65 src/controllers.asm
ca65 src/reset.asm
ca65 src/input.asm
ld65 src/backgrounds.o src/controllers.o src/input.o src/reset.o -C nes.cfg -o input.nes
```

The resulting .nes file should let you move the player ship freely using whatever your emulator has mapped to the D-pad inputs. Here it is running in an emulator in your browser! The emulator supports keyboard shortcuts, using the following keys (for QWERTY keyboards). You will need to first click on the emulator area or otherwise focus it in order for keypresses to be registered.

Controller	Keyboard
Right	Right
Left	Left
Down	Down
Up	Up
Start	Enter
Select	Tab
B	s
A	a



■ *Homework*

Here are some exercises to try now that we have covered the basics of controller input. You can download the code from this chapter (https://famicom.party/book/projects/16-input.zip) as a starting point.

Using the code from Chapter 14 as a guide, make it so the player's ship cannot wrap around the screen. When the ship approaches an edge, ignore controller input that would move it off screen.

Make it so the ship responds differently when multiple buttons are pressed at the same time. As an example, holding "B" while pressing a direction could make the player's ship move twice as fast as normal.

Let the player "pause" the game. When Start is pressed, stop scrolling the screen and ignore controller directional movement. When Start is pressed again, resume scrolling and listening to inputs.

17. Object Pools

With the ability to control the spaceship, our sample project is starting to feel like a real game and not just a tech demo. Let's continue to build on that momentum by introducing enemy spaceships. For now, we won't worry about detecting collisions - that will come in the next chapter. Instead, the main focus in this chapter will be object pools, a technique for handling multiple objects in a flexible way.

■ *Why Pooling?*

In the context of this book, we'll consider an object pool to be a fixed block of memory that is used to hold data about a particular type of entity in a game. Each entity needs to track certain information—what tiles it is drawn with, its X and Y positions, etc.—and that information will need to be stored somewhere in memory. In games that are not turn-based, entities are updated nearly every frame, meaning the best place to store an entity's data is in zero-page RAM.

In modern game development environments, adding a few more entities when needed is not a concern. Modern systems can easily spare a few bytes (or kilobytes, or megabytes) for additional entities. On the NES, though, zero-page space is limited and finite, so every byte has to be planned in advance. Setting up an object pool guarantees that the game will never use more memory than we allocated in advance, and lets us more easily plan out how the 256 bytes of zero-page space will be used.

Object pools also help from a gameplay perspective. If we use an object pool for player projectiles and give it three slots, then the player can only have three projectiles on screen at one time. Without that kind of limitation, a player could use a turbo controller (or tap rapidly) to fill the screen with projectiles, taking up all of the OAM slots, making a number of sprites invisible due to the eight-sprites-per-scanline limitation and likely causing massive slowdown due to the additional demands on collision detection.

■ *Designing Pools*

An object pool should be flexible enough to accommodate a range of entity types, so long as those entities have some shared role. It is common to put entities that can harm the player into a pool together, separate from entities that are neutral or helpful. An alternative approach could be keeping objects that participate in collision detection in one pool, and those that do not in another. There is no set way of deciding what goes into a pool together and what does not. In general, a good object pool design will make it easy to iterate through the entities in the pool to do common tasks, without needing to write logic that excludes certain entities. As an example, if you put both enemies and player projectiles into a single pool, collision detection would be extremely complicated, since you would need to constantly test whether any two items can actually affect one another. Separate pools for enemies and player projectiles let you cycle through the entities in one pool and compare them to entities from the other, greatly reducing the complexity.

Once you've decided which types of entities will be pooled together, the next step is establishing what data you will be storing for each entity. For reasons we'll get to soon, each entity in a pool will need to store the same data as every other entity in the pool. For our space shooter game, we will make an enemies pool, where each enemy has the following data:

- X position
- Y position
- X velocity
- Y velocity
- Bit flags:
  - Type (determines what sprites to draw, etc.)
  - Active flag (should this be updated / drawn / etc.?)

This is a good starting point, but there is plenty of other data you could choose to store. You could give each enemy a health counter, allowing enemies to take more than one attack to defeat. Enemies will likely need some kind of "AI" behavior; you could keep a list of behaviors somewhere else in code and reference the current behavior pattern with an ID number. Enemies could also have animations, which would require you to store the current animation state (animation type and frame number) for each entity. Note that even if we add all of these properties, the pool is still quite flexible. Giving pool entities a "type" value could translate into nearly anything, since the code that transforms a type number into sprites drawn on screen can live outside of the pool. This pool could easily handle enemies ranging from a single sprite tile to giant boss monsters without any changes.

Similarly, let's create a pool to store player projectiles. Player projectiles do not share a role with enemies, and since they don't vary much from one another, the pool design can be simpler as well:

- X position
- Y position

Here we assume that all projectiles have the same velocity and use the same graphics, so all we need to track is their position on screen. If you wanted to add different weapons (or weapon upgrades), you would need to add more data to the entities in the pool.

■ *Implementing object pools*

In most modern programming languages, if you wanted to track a set of similar entities, you would reach for something like an array of objects (or a list of instances, or a slice of structs; the terminology here varies by language). Here's an example of that approach in JavaScript:

```
let enemiesPool = [
  {
    x: 57,
    y: 12,
    type: 3,
    active: true
  },
  {
    x: 36,
    y: 12,
    type: 3,
    active: false
  }
];
```

This method makes it very easy to loop through the objects within the array, and then take action on each object in turn. On the NES, however, this approach is very inefficient. Using this sort of setup (which is called array of structures), NES assembly code to loop through all entities would need to:

1. Store a "current object address" and "current object index" somewhere, as well as the length of each field and the total length of each object.
2. Access each property of the current object by using a table of offsets from the start of the object (load the offset of the property into the X register, then e.g. `LDA current_obj_address, X`).
3. To get to the next object, add the object length to the current object's starting address.
4. Check each time you move to the next object that the current object index has not exceeded the total length of the array.

In contrast, due to the 6502's wealth of addressing modes, a more popular approach on the NES is the structure of arrays. To go back to our previous JavaScript example, a structure of arrays approach would look like this:

```
let enemiesPool = {
  x: [57, 36],
  y: [12, 12],
  type: [3, 3],
  active: [true, false]
};
```

While this approach might seem counter-intuitive at first, it works very well with the 6502's indexed addressing modes. Here is our hypothetical pool traversal code under structure of arrays:

1. Store a current object index in the X register.
2. To look up a property, use indexed addressing with the label that marks the start of that array, e.g. `LDA enemy_types, X` to get the type of the current enemy.
3. To go to the next object in the pool, increment the X register.
4. Check each time you increment if the current object index has exceeded the length of the pool.

The number of steps is the same, but each step is much simpler and also faster to execute.

Now that we have both a design and an implementation approach for the pool, let's actually create it. We'll need to add the following to our `.segment "ZEROPAGE":`

```
NUM_ENEMIES = 5

; enemy object pool
enemy_x_pos: .res NUM_ENEMIES
enemy_y_pos: .res NUM_ENEMIES
enemy_x_vels: .res NUM_ENEMIES
enemy_y_vels: .res NUM_ENEMIES
enemy_flags: .res NUM_ENEMIES

; player bullet pool
bullet_xs: .res 3
bullet_ys: .res 3

; export all of this
.exportzp enemy_x_pos, enemy_y_pos
.exportzp enemy_x_vels, enemy_y_vels
.exportzp enemy_flags
```

With this code, we've reserved 31 bytes of zero-page RAM across two pools, to store the state of our non-player objects in a structure of arrays format. The `enemy_flags` byte for each enemy is a series of bitfields; here, I'll use three bits to store the enemy type (giving us eight possible enemy types), one bit to store whether or not the enemy at this slot is 'active' (should be drawn to the screen), and four bits reserved for future use.

To make use of these pools, we will want to cycle through them each frame, making any updates as needed. For now, this will be relatively simple: creating new enemies if there are open slots, updating the positions of existing enemies, and freeing up enemy slots when they have exited the screen. In the future, this loop is where we will add collision detection, "AI" behavior, and animation updates.

Because there will be so much going on in this loop, there is actually one more thing to do before we implement it. Up to this point, our `main` code has set things up and then just waited for NMIs to occur, with all actual logic being part of the NMI handler. While our projects were simple, this was sufficient, but as we add more and more functionality, we will quickly run out of CPU time. Vblank (the time during which the NMI handler runs) lasts for about 2,250 CPU cycles, which is a decent amount but not a whole lot of time. In contrast, the CPU spends about 27,000 cycles outside of Vblank each frame. Currently, that time is wasted just waiting for the next Vblank. A better approach would be to spend that time setting things up for the next Vblank, and letting our NMI handler focus on sending those changes to the PPU.

Nearly all of this section is based on "The frame and NMIs", from the NESDev Wiki.

■ *Moving code from NMI to main*

To make the most of Vblank time, we'll want to limit the NMI handler to only handling things that occur every frame - even if the game is paused, or slowdown is occurring. Generally those items will include:

- Copying sprite data to OAM;
- Making any per-frame background updates;
- Updating the scroll registers;
- Playing music and sound effects.

Everything else should be handled as part of the main loop, since there is so much more CPU time available there. In fact, there is so much CPU time available that we will need to do something to keep the CPU busy while we wait for the next Vblank, or else our game logic will race far ahead of what is being displayed on the screen. To start, we need a way to track whether or not Vblank (an NMI) has occured yet. I'll use one byte of zeropage to track whether the non-Vblank code should "sleep" (wait for NMI), or run itself.

```
  sleeping: .res 1
```

If `sleeping` is non-zero, we are waiting for the next NMI before doing anything else. The end of our main loop can change from the current "jump forever":

```
  forever:
    JMP forever
```

To something that takes sleeping into account:

```
    INC sleeping
  sleep:
    LDA sleeping
    BNE sleep

    JMP mainloop
```

Here, we increment `sleeping` so it is not zero and then repeatedly load it, until a load results in zero. How does `sleeping` revert to zero? In our NMI handler. At the end of the NMI handler, we add:

```
    LDA #$00
    STA sleeping
```

So, the entire flow is now:

- Reset handler runs at power-on (or reset), and ends with `JMP main`
- Main code (outside of Vblank) runs until it hits the `sleep` loop, which sets `sleeping` to one and waits for NMI
- At Vblank time, NMI handler runs, and sets `sleeping` back to zero
- Main code, still in the `sleep` loop, sees that `sleeping` is now zero and jumps back to the beginning of the loop

Now, the main loop and the NMI handler pass control back and forth to one another, letting us make the most of both. Note that the end of our `main` proc jumps to a new label - mainloop - instead of jumping back to the beginning of `main`. `main` begins with some initialization that we don't want to repeat every frame, so we need to add a `mainloop` label within `main` at the point that we want to run repeatedly.

It's time to move code that is not essential to NMIs out into the main loop. Looking through the NMI handler from the last chapter, the following operations are worth keeping in NMI (because they can only be done during Vblank):

Copying sprite data to OAM

Setting PPUCTRL

Setting scroll values

The rest (including calculating what values to write to PPUCTRL and decrementing `scroll`) should be moved to the main loop. If we remove those items, our new, shorter NMI handler looks like this:

```
.proc nmi_handler
  PHP
  PHA
  TXA
  PHA
  TYA
  PHA

  ; copy sprite data to OAM
  LDA #$00
  STA OAMADDR
  LDA #$02
  STA OAMDMA

  ; set PPUCTRL
  LDA ppuctrl_settings
  STA PPUCTRL

  ; set scroll values
  LDA #$00 ; X scroll first
  STA PPUSCROLL
  LDA scroll
  STA PPUSCROLL

  ; all done
  LDA #$00
  STA sleeping

  PLA
  TAY
  PLA
  TAX
  PLA
  PLP
  RTI
.endproc
```

Notice that the NMI handler has a new beginning and ending. Previously, all of our game logic occurred in the NMI handler. Nothing else could ever interrupt our game logic or stealthily change the memory addresses we were working with, but now it's very possible for something in main code to change memory locations or registers that NMI expects to use, and vice-versa. (We literally just did that sort of thing, by having both NMI and `main` make use of `sleeping`.) This means that our NMI handler must now act like any other subroutine, saving the state of registers and restoring them when it is done.

Next, let's move the items we removed from the NMI handler into the main loop. Here is just the `mainloop` portion of our new `.proc main`:

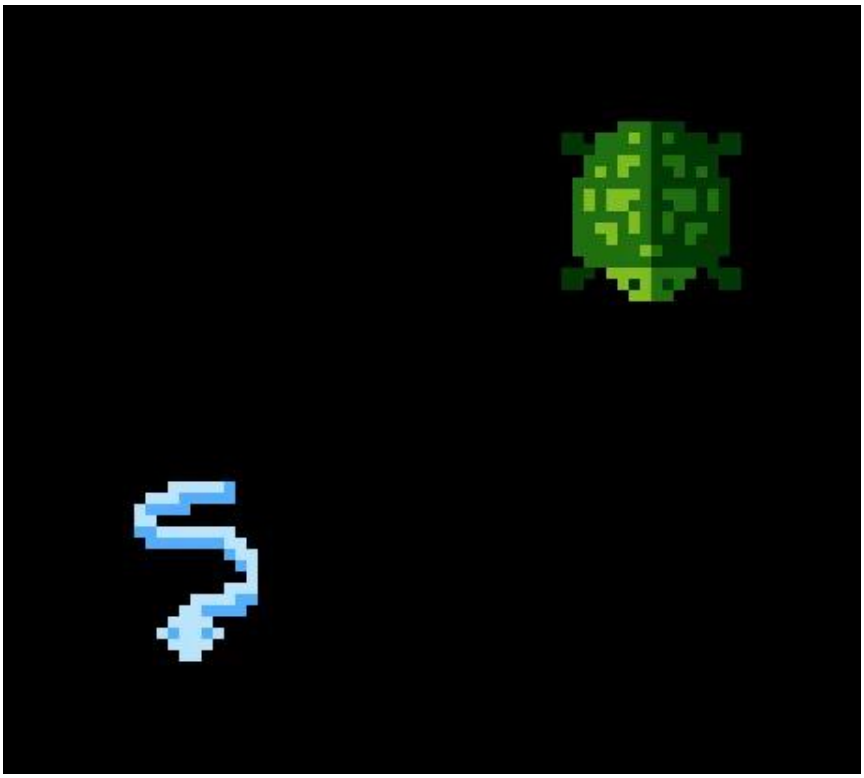
```
  mainloop:
    ; Read controllers.
    JSR read_controller1

    ; Update the player and prep to draw
    JSR update_player
    JSR draw_player

    ; Check if PPUCTRL needs to change
    LDA scroll ; did we reach the end of a nametable?
    BNE update_scroll
    ; if yes,
    ; Update base nametable
    LDA ppuctrl_settings
    EOR #%00000010 ; flip bit 1 to its opposite
    STA ppuctrl_settings
    ; Reset scroll to 240
    LDA #240
    STA scroll

  update_scroll:
    DEC scroll

    ; Done processing; wait for next Vblank
    INC sleeping
  sleep:
    LDA sleeping
    BNE sleep

    JMP mainloop
```

Most of this is copied as-is from the old NMI handler, but note that we are not writing to PPUCTRL here. Writes to the PPU can only occur during Vblank (in the NMI handler), or else you will get distracting visual artifacts on the screen.

With the main loop and NMI now separated, we have ample time to make use of our new object pools in the main loop to generate, draw, and manage enemies and projectiles in the game.

■ *Enemy routines*

We'll need some enemies to manage, of course. I've created graphics tiles for two enemy types, seen here:



Our two enemy types - space turtles and galactic snakes. I am not an artist but I'm trying my best.

I'm going to call the turtles "type 1" and the snakes "type 2", and we will use those values in the "type" field for each enemy in the pool. Those types will let us easily determine which subroutines to call in order to update a particular enemy and draw it. For better organization, I am going to put enemy subroutines into their own file, `enemies.asm`, and import them into the main file.

The drawing subroutine is the simpler of the two. If all of our enemy types can be drawn with 2x2 tile metasprites, they can share a single drawing routine that picks tiles to draw based on enemy type. (If the enemies are different sizes, then multiple drawing subroutines will be required. Instead of calling `draw_enemy` directly, the code that loops through the enemy pool will need to check the enemy's type to determine the appropriate draw function.). Using `current_enemy` as an index into various arrays, the drawing subroutine can:

- Find the appropriate place in OAM to write sprite data
- Write sprite X and Y positions with values from `enemy_x_pos` and `enemy_y_pos`
- Identify the tiles to write using `enemy_flags` and a lookup table of tile numbers
- Select an appropriate palette

The drawing subroutine will need to make use of both index registers: one to provide an offset into each enemy info array, and one to keep track of the offset into OAM as each byte is written. Additionally, the X register will need to do double duty, generally using `current_enemy` for offsets, but changing to the current enemy's type (stored in zeropage as `current_enemy_type`) when loading tile numbers. I am using bit 7 of the `enemy_flags` byte as a marker for "active" or "inactive". As discussed in a previous chapter, that bit can be turned on with `ORA #%10000000`, and turned off with `EOR #%10000000`. Here is the drawing subroutine (in enemies.asm):

```
126   .export draw_enemy
127   .proc draw_enemy
128     PHP
129     PHA
130     TXA
131     PHA
132     TYA
133     PHA
134
135     ; First, check if the enemy is active.
136     LDX current_enemy
137     LDA enemy_flags,X
138     AND #%10000000
139     BNE continue
140     JMP done
141
142   continue:
143     ; Find the appropriate OAM address offset
144     ; by starting at $0210 (after the player
145     ; sprites) and adding $10 for each enemy
146     ; until we hit the current index.
147     LDA #$10
148     LDX current_enemy
149     BEQ oam_address_found
150   find_address:
151     CLC
152     ADC #$10
153     DEX
154     BNE find_address
155
156   oam_address_found:
157     LDX current_enemy
158     TAY ; use Y to hold OAM address offset
159
```

```
160     ; Find the current enemy's type and
161     ; store it for later use. The enemy type
162     ; is in bits 0-2 of enemy_flags.
163     LDA enemy_flags, X
164     AND #%00000111
165     STA current_enemy_type
166
167     ; enemy top-left
168     LDA enemy_y_pos, X
169     STA $0200, Y
170     INY
171     LDX current_enemy_type
172     LDA enemy_top_lefts, X
173     STA $0200, Y
174     INY
175     LDA enemy_palettes, X
176     STA $0200, Y
177     INY
178     LDX current_enemy
179     LDA enemy_x_pos, X
180     STA $0200, Y
181     INY
182
183     ; enemy top-right
184     LDA enemy_y_pos, X
185     STA $0200, Y
186     INY
187     LDX current_enemy_type
188     LDA enemy_top_rights, X
189     STA $0200, Y
190     INY
191     LDA enemy_palettes, X
192     STA $0200, Y
193     INY
194     LDX current_enemy
195     LDA enemy_x_pos, X
196     CLC
197     ADC #$08
198     STA $0200, Y
199     INY
200
```

```
201     ; enemy bottom-left
202     LDA enemy_y_pos, X
203     CLC
204     ADC #$08
205     STA $0200, Y
206     INY
207     LDX current_enemy_type
208     LDA enemy_bottom_lefts, X
209     STA $0200,Y
210     INY
211     LDA enemy_palettes, X
212     STA $0200, Y
213     INY
214     LDX current_enemy
215     LDA enemy_x_pos, X
216     STA $0200, Y
217     INY
218
219     ; enemy bottom-right
220     LDA enemy_y_pos, X
221     CLC
222     ADC #$08
223     STA $0200, Y
224     INY
225     LDX current_enemy_type
226     LDA enemy_bottom_rights, X
227     STA $0200,Y
228     INY
229     LDA enemy_palettes, X
230     STA $0200,Y
231     INY
232     LDX current_enemy
233     LDA enemy_x_pos, X
234     CLC
235     ADC #$08
236     STA $0200, Y
237
238   done:
239     PLA
240     TAY
241     PLA
242     TAX
243     PLA
244     PLP
245     RTS
246   .endproc
```

Notice that on lines 139-140, a branch (`BNE`) skips over a `JMP` to near the end of the subroutine, rather than branching directly to that label. This is intentional; a branch command, when assembled into machine code, takes a relative movement as its operand, and because that operand is one (signed) byte, a branch can only move up to 128 bytes backward or 127 bytes forward. The `done` label is more than 127 bytes ahead of where we would branch, so only a `JMP` can be used here.

In keeping with the structure-of-arrays model, the tile numbers for each enemy type are stored in four arrays: `enemy_top_lefts`, `enemy_top_rights`, `enemy_bottom_lefts`, and `enemy_bottom_rights`. The enemy type serves as the index into each of these arrays, so e.g. the first enemy type (type zero) will be the first element of each array. Here, for reference, are those arrays, down in `.segment "RODATA"`. Each array lists the appropriate turtle tile, followed by the appropriate snake tile. These arrays are stored in `RODATA`.

```
248   .segment "RODATA"
249
250   enemy_top_lefts:
251   .byte $09, $0d
252   enemy_top_rights:
253   .byte $0b, $0e
254   enemy_bottom_lefts:
255   .byte $0a, $0f
256   enemy_bottom_rights:
257   .byte $0c, $10
```

For enemy behavior, I'll keep things simple for now with a single subroutine that increments the enemy's Y position by its Y velocity (from `enemy_y_vels`) and, if it is greater than 239 (the bottom of the screen), marks it as inactive. This subroutine is also in `enemies.asm`.

```
12    .export update_enemy
13    .proc update_enemy
14      PHP
15      PHA
16      TXA
17      PHA
18      TYA
19      PHA
20
21      ; Check if this enemy is active.
22      LDX current_enemy
23      LDA enemy_flags, X
24      AND #%10000000
25      BEQ done
26
```

```
27      ; Update Y position.
28      LDA enemy_y_pos, X
29      CLC
30      ADC enemy_y_vels, X
31      STA enemy_y_pos, X
32
33      ; Set inactive if Y >= 239
34      CPY #239
35      BCC done
36      LDA enemy_flags, X
37      EOR #%10000000
38      STA enemy_flags, X
39
40    done:
41      PLA
42      TAY
43      PLA
44      TAX
45      PLA
46      PLP
47      RTS
48    .endproc
```

With these subroutines written, we are ready (finally?) to begin spawning enemies.

■ *Spawning enemies via object pool*

There are many approaches to spawning enemies, but let's keep it simple for now. Each frame, we will use CPU time (outside of NMI) to loop through the enemies list. If an enemy slot is active, we will call `update_enemy`. If a slot is inactive, we will start a timer that counts down from 20 to zero, at which point we will spawn an enemy in the first inactive slot our loop finds. This gives us a new enemy 1/3 of a second (20 frames) after an enemy goes off screen. When not in use, the timer will store `$ff`, to give a clear difference between "counting down to zero" and "waiting to start counting down".

First, the code to loop through enemies and update them. Here is our new subroutine, `process_enemies`, in `enemies.asm`:

```
.export process_enemies
.proc process_enemies
  ; Push registers onto the stack
  PHP
  PHA
  TXA
  PHA
  TYA
  PHA

  ; Start with enemy zero.
  LDX #$00

enemy:
  STX current_enemy
  LDA enemy_flags, X
  ; Check if active (bit 7 set)
  AND #%10000000
  BEQ spawn_or_timer
  ; If we get here, the enemy is active,
  ; so call update_enemy
  JSR update_enemy
  ; Then, get ready for the next loop.
  JMP prep_next_loop
spawn_or_timer:
  ; Start a timer if it is not already running.
  LDA enemy_timer
  BEQ spawn_enemy ; If zero, time to spawn
  CMP #20 ; Otherwise, see if it's running
  ; If carry is set, enemy_timer > 20
  BCC prep_next_loop

  LDA #20
  STA enemy_timer
  JMP prep_next_loop
spawn_enemy:
   ; TODO!

prep_next_loop:
  INX
  CPX #NUM_ENEMIES
  BNE enemy
```

```
    ; Done with all enemies. Decrement
    ; enemy spawn timer if 20 or less
    ; (and not zero)
    LDA enemy_timer
    BEQ done
    CMP #20
    BEQ decrement
    BCS done
  decrement:
    DEC enemy_timer

  done:
    ; Restore registers, then return
    PLA
    TAY
    PLA
    TAX
    PLA
    PLP

    RTS
  .endproc
```

This is a large subroutine, but the pieces are fairly straightforward. The loop that processes enemies starts at the `enemy` label. Using the X register as the index of which enemy number is currently being processed, the loop first checks if the current enemy is active by testing its entry in the `enemy_flags` array. If the enemy is active, `update_enemy` is called and the rest of the loop logic is skipped. Otherwise, a new zeropage address (`enemy_timer`) is checked. A value of 255 means that the timer is not currently in use; when a timer starts, it is set to 20 and counts down to zero. If `enemy_timer` is zero, `BEQ spawn_enemy` branches to the portion of code that spawns an enemy (to be described next). If it is not zero, a second check tests if `enemy_timer` is greater than 20. If so, the timer is not currently in use, so a new timer can be started. Otherwise, the timer is already in use and we are done processing this enemy.

How should new enemies be spawned? Depending on the kind of game you are looking to make, a good approach might be to implement a random number generator to select the enemy type to spawn, or hard-coding particular patterns of enemies that appear during a stage. (The whole idea of having different "modes" in a game - whether that's different playstyles, different stages, or even simple things like a title screen or pause state - will be its own separate chapter.). Since this chapter is already quite long, I'm going to opt for something much simpler instead: specific slots are always specific enemy types. The first three slots will be turtles, which will move at a speed of 1px per frame, and the last two will be snakes, which will move twice as fast. Here is the setup code at the beginning of our `main` subroutine:

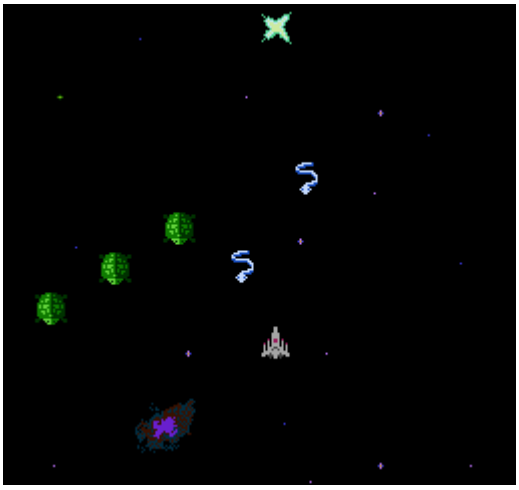
```
115     ; set up enemy slots
116     LDA #$00
117     STA current_enemy
118     STA current_enemy_type
119
120     LDX #$00
121   turtle_data:
122     LDA #$00 ; turtle
123     STA enemy_flags,X
124     LDA #$01
125     STA enemy_y_vels,X
126     INX
127     CPX #$03
128     BNE turtle_data
129     ; X is now $03, no need to reset
130   snake_data:
131     LDA #$01
132     STA enemy_flags,X
133     LDA #$02
134     STA enemy_y_vels,X
135     INX
136     CPX #$05
137     BNE snake_data
138
139     LDX #$00
140     LDA #$10
141   setup_enemy_x:
142     STA enemy_x_pos,X
143     CLC
144     ADC #$20
145     INX
146     CPX #NUM_ENEMIES
147     BNE setup_enemy_x
```

Notice also that each enemy slot has a specific X coordinate, so that they do not overlap with one another.

Now, when it's time to spawn a new enemy, the first inactive slot our loop finds is marked as active and has its Y position set to zero. Here is that `spawn_enemy` label from the `process_enemies` subroutine, with "`TODO`" replaced by actual code:

```
  spawn_enemy:
    ; Set this slot as active
    ; (set bit 7 to "1")
    LDA enemy_flags,X
    ORA #%10000000
    STA enemy_flags,X
    ; Set y position to zero
    LDA #$00
    STA enemy_y_pos,X
    ; IMPORTANT: reset the timer!
    LDA #$ff
    STA enemy_timer
```

When all of the code above is assembled and linked, the result shows off our new object pools in action!



Notice how our object pool limits the game to having no more than five enemies on screen at a time. Just having different Y velocities already introduces some measure of randomness (or at least "random-seeming") in how enemies appear on screen.

■ *Homework*

Here are a few changes you might like to try out. See how each of them affects the "feel" of the game, and experiment with other changes to what we've already done here. To get you started, here is a zip file with the source code from this chapter (https://famicom.party/book/projects/17-objectpools.zip).

- Change the number of enemies in the pool, and play with the ratio of turtles vs. snakes.
- Add a third enemy type. Create new graphics and update the arrays of tiles, and give the new enemy type its own Y velocity.
- Give at least one enemy type an X velocity, and update the update_enemy code to add X velocity to X position.
- Instead of hard-coding enemy X positions up front, when a new enemy spawns, set its X position to the player's X position at the time of spawning.
- (HARDER) Add the ability for the player to shoot bullets. Bullets should use the first inactive slot in the bullet pool to spawn, and should start at the player's X and Y position at the time they are spawned. You will need to create your own graphics tiles, drawing routine, etc. as well.