MULTIGESTURE.NET

Subscribe RSS | Email

Subscribe

A multitouch and HCI research blog Twitter LinkedIn Youtube

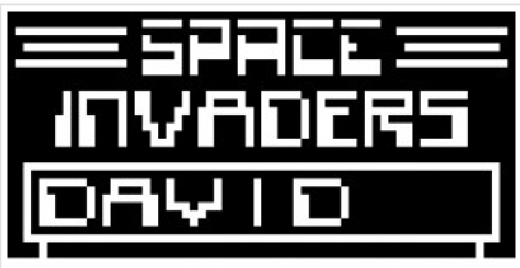
How to write an emulator (CHIP-8 interpreter)

This guide is intended to give a brief introduction to the world of emulation and will also teach you how to write one yourself from scratch.

Personally I have been excited about emulators since the late 90's. As I didn't own a console back in the days (only had a C64), I was pleasantly surprised when I learned that you could use an emulator to run console games on the PC. I still remember playing Super Mario 3 on the PC using a SNES/Super Famicom emulator Snes9x and a few years later completing Metal Gear Solid using Bleem! (PSX emulator).

These days however I'm more focussed on providing support to emulator projects of recent consoles such as: PCSX2 (Sony Playstation 2), Dolphin-emu (Nintendo Gamecube and Wii) and nullDC (Sega Dreamcast).

While this guide expects you to have some basic knowledge of computer systems and assumes you know a program language, it should also be an interesting read for people who are interested in emulation in general.



#define emulator

I think it's important to first understand what an emulator is and isn't.

An emulator is a computer program that mimics the internal design and functionality of a computer system (System A). It allows users to run software designed for this specific system (Sytem A) on a totally different computer system or architecture (System B).

Often people confuse a simulator with an emulator and vice versa. Just remember that these words aren't synonyms.

Let's take a look at the following example:
Pong is a 2D tennis game which was developed by Atari and ran on their own hardware. However, the game wasn't just available on Atari systems, but also on rival platforms such as Amstrad, Amiga and the C64.

Since not every Pong game was licensed by Atari to run on these platforms, it also meant that not every game was running the code from Atari. Basically what happened is that people created their own implementation (clones) of the game Pong. In this case they *simulated* the looks and game behavior of Pong.

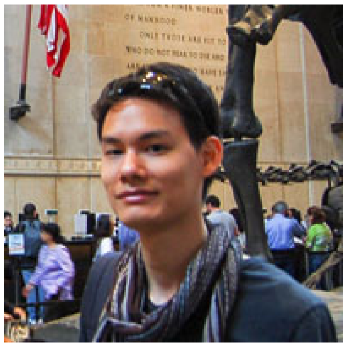
Search

Search

About

I'm Laurence Muller (M.Sc.), a former Fellow of the Scientists' Discovery Room Lab (SDR Lab) at Harvard University / School of Engineering and Applied Sciences (SEAS), where I worked on innovative scientific software for multi-touch devices and display wall systems.

I'm also the founder of Epic Windmill, a software company that develops apps and games for mobile devices.



Twitter

RT @maddiestone v2 of my free Intro to Android App Reverse Engineering workshop is here! I've added 3 new exercises, walk-through videos for all 7 exercises, a new module on obfuscation, & exercises on vuln hunting rather than just malware. I hope it helps! maddiestone.github.io/Android… pic.twitter.com/0iPNoJ3p7y

About 2 months ago from Laurence Muller's Twitter

Follow @LaurenceMuller 345 followers

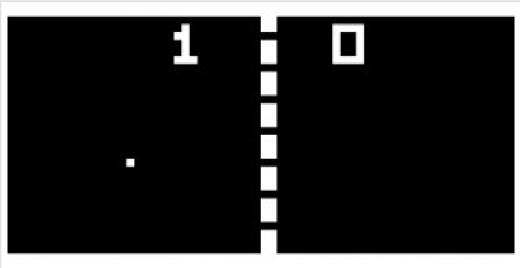
Random quotes

- Never assume. It makes an "ass" out of "u" and "me".

HCI Conferences

- ACM Tabletop 2010 (Saarbrucken, Germany)
- ACM Tabletop 2011 (Japan)
- ACM Tabletop 2012 (Cambridge, MA, USA)
- CHI 2009 (Boston, MA, USA)
- CHI 2010 (Atlanta, GA, USA)
- CHI 2011 (Vancouver, BC, CA)
- CHI 2012 (Austin, TX, USA)

In case of an emulator, we choose not to re-implement the game Pong for our native system. Instead, we re-create the environment with a computer program which allows us to run the original machine code of Pong. A benefit of this is that it won't just allow us to run Pong, but also any other application developed for that platform.

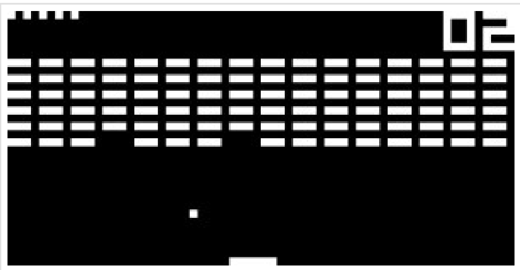


What is a CHIP-8?

The Chip 8 actually never was a real system, but more like a virtual machine (VM) developed in the 70's by Joseph Weisbecker. Games written in the Chip 8 language could easily run on systems that had a Chip 8 interpreter.

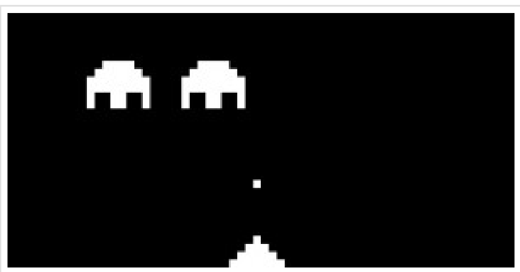
Why start with a CHIP-8 emulator?

Writing a Chip 8 emulator is probably the easiest emulation project you can undertake. Due to small number of opcodes (35 in total for Chip 8) and the fact that a lot of instructions are used in more advanced CPUs, a project like this is educational (get a better understanding of how the CPU works and how machine code is executed), manageable (small number of opcodes to implement) and not too time consuming (project can be finished in a few days).



Before you start…

- Pick a programming language you're familiar with (C/C++ or Java are common).
  The examples below will use C/C++
- Don't use this project as a way to learn how to program.
  (If bitwise operations confuse you, study them first)
- You will probably need to use 3rd party libraries to handle audio / video output and user input (GLUT / SDL / DirectX)
- OK GO!



CPU Specifications

When you start writing an emulator, it is important that you find as much information as possible about the system you want to emulate. Try to find out how much memory and registers are used in the system, what architecture it is using and see if you can get hold of technical documents that describe the instruction set.

In the case of the Chip 8, I would recommend taking a look at the Chip 8 description on Wikipedia.

- IDC 2009 (San Jose, CA, USA)
- IEEE Tabletop 2008 (Amsterdam, NL)
- IEEE Tabletop 2009 (Banff, Canada)
- SuperComputing 2008 (Austin, TX, USA)
- TEI 2010 (Cambridge, MA, USA)
- TEI 2011 (Funchal, Portugal)

Multitouch

- David Wallin (whitenoise)
- Harry van der Veen (gravano)
- Jimi Hertz
- Johannes Hirche (xwolf)
- Joshua Blake
- Justin Ireland
- Lynn Marentette
- Mathieu Virbel (Tito)
- Paul D'Intino (Fairlane)
- Ralph Das
- Richard Monson-Haefel (clevermonkey)
- Sebastian Hartman
- Seth Sandler (cerupcat)
- Sharath Patali
- SOCO Amsterdam
- Taha Bintahir
- Thomas Hansen
- Tim Roth
- Touchlib

Multitouch (commercial)

- Apple iPad
- Apple iPhone
- MERL DiamondTouch
- MS Surface
- Perceptive Pixel

Science

- Dr. C. Shen
- Dr. R.G. Belleman
- Harvard SEAS
- Harvard University SDR Lab
- Prof. Peter Sloot
- Scientific Visualization and Virtual Reality Research Group
- Universiteit van Amsterdam

Archive

August 2012 (1)
April 2012 (1)
February 2012 (2)
November 2011 (1)
March 2011 (1)
December 2010 (4)
November 2010 (2)
April 2010 (2)
February 2010 (1)
December 2009 (1)

Category

Android
Application
BlackBerry
Blogroll
Featured
Howto
Infovis
Multitouch

I'll give you a brief overview of the Chip 8 system and some hints on how to implement the essential parts:

- The Chip 8 has 35 opcodes which are all two bytes long. To store the current opcode, we need a data type that allows us to store two bytes. An unsigned short has the length of two bytes and therefor fits our needs:
- ```
  unsigned short opcode;
  ```
- The Chip 8 has 4K memory in total, which we can emulated as:
- ```
  unsigned char memory[4096];
  ```
- CPU registers: The Chip 8 has 15 8-bit general purpose registers named V0,V1 up to VE. The 16th register is used for the 'carry flag'. Eight bits is one byte so we can use an unsigned char for this purpose:
- ```
  unsigned char V[16];
  ```
- There is an Index register I and a program counter (pc) which can have a value from `0x000` to `0xFFF`
- ```
  unsigned short I;
  unsigned short pc;
  ```
- The systems memory map:
- ```
  0x000-0x1FF - Chip 8 interpreter (contains font set in emu
  0x050-0x0A0 - Used for the built in 4x5 pixel font set (0-
  0x200-0xFFF - Program ROM and work RAM
  ```
- The graphics system: The chip 8 has one instruction that draws sprite to the screen. Drawing is done in XOR mode and if a pixel is turned off as a result of drawing, the VF register is set. This is used for collision detection.
- The graphics of the Chip 8 are black and white and the screen has a total of 2048 pixels (64 x 32). This can easily be implemented using an array that hold the pixel state (1 or 0):
- ```
  unsigned char gfx[64 * 32];
  ```
- Interupts and hardware registers. The Chip 8 has none, but there are two timer registers that count at 60 Hz. When set above zero they will count down to zero.
- ```
  unsigned char delay_timer;
  unsigned char sound_timer;
  ```
- The system's buzzer sounds whenever the sound timer reaches zero.

It is important to know that the Chip 8 instruction set has opcodes that allow the program to jump to a certain address or call a subroutine. While the specification don't mention a stack, you will need to implement one as part of the interpreter yourself. The stack is used to remember the current location before a jump is performed. So anytime you perform a jump or call a subroutine, store the program counter in the stack before proceeding. The system has 16 levels of stack and in order to remember which level of the stack is used, you need to implement a stack pointer (sp).

- ```
  unsigned short stack[16];
  unsigned short sp;
  ```

Finally, the Chip 8 has a HEX based keypad (`0x0-0xF`), you can use an array to store the current state of the key.

- ```
  unsigned char key[16];
  ```

Game Loop

To give you an idea on how to design your emulator, I made a small example of a layout. It does not teach you how to use GLUT or SDL to handle graphics and input but merely shows you how the flow of your emulator should be.

```
#include
#include   // OpenGL graphics and input
#include "chip8.h" // Your cpu core implementation

chip8 myChip8;

int main(int argc, char **argv)
```


August 2009 (1)
July 2009 (3)
February 2009 (1)
December 2008 (1)
November 2008 (1)
September 2008 (2)
July 2008 (1)
May 2008 (3)
March 2008 (1)
February 2008 (1)
January 2008 (1)
November 2007 (2)
September 2007 (5)
July 2007 (1)
June 2007 (1)
May 2007 (5)
April 2007 (3)
```

```
 8 {
 9   // Set up render system and register input callbacks
10   setupGraphics();
11   setupInput();
12
13   // Initialize the Chip8 system and load the game into the memo
14   myChip8.initialize();
15   myChip8.loadGame("pong");
16
17   // Emulation loop
18   for(;;)
19   {
20     // Emulate one cycle
21     myChip8.emulateCycle();
22
23     // If the draw flag is set, update the screen
24     if(myChip8.drawFlag)
25       drawGraphics();
26
27     // Store key press state (Press and Release)
28     myChip8.setKeys();
29   }
30
31   return 0;
32 }
```

- Line 3-5: In this example we assume you will create a separate class to handle the opcodes.
- Line 10-11: Setup the graphics (window size, display mode, etc) and input system (bind callbacks)
- Line 14: Clear the memory, registers and screen
- Line 15: Copy the program into the memory
- Line 21: Emulate one cycle of the system
- Line 24: Because the system does not draw every cycle, we should set a draw flag when we need to update our screen. Only two opcodes should set this flag:
  - `0x00E0` – Clears the screen
  - `0xDXYN` – Draws a sprite on the screen
- Line 28: If we press or release a key, we should store this state in the part that emulates the keypad

Emulation cycle

Next we will look into the emulation cycle.

```
void chip8::initialize()
{
  // Initialize registers and memory once
}

void chip8::emulateCycle()
{
  // Fetch Opcode
  // Decode Opcode
  // Execute Opcode

  // Update timers
}
```

Every cycle, the method *emulateCycle* is called which emulates one cycle of the Chip 8 CPU. During this cycle, the emulator will Fetch, Decode and Execute one opcode.

Fetch opcode
During this step, the system will fetch one opcode from the memory at the location specified by the program counter (pc). In our Chip 8 emulator, data is stored in an array in which each address contains one byte. As one opcode is 2 bytes long, we will need to fetch two successive bytes and merge them to get the actual opcode.

To demonstrate how this works we will be using opcode `0xA2F0`.

```
// Assume the following:
memory[pc]     == 0xA2
memory[pc + 1] == 0xF0
```

In order to merge both bytes and store them in an `unsigned short` (2 bytes datatype) we will use the bitwise OR operation:

```
opcode = memory[pc] << 8 | memory[pc + 1];
```

So what did actually happen?
First we shifted `0xA2` left 8 bits, which adds 8 zeros.

```
0xA2       0xA2 << 8 = 0xA200   HEX
10100010   1010001000000000     BIN
```

Next we use the bitwise OR operation to merge them:

```
1010001000000000 | // 0xA200
        11110000 = // 0xF0 (0x00F0)
------------------
1010001011110000   // 0xA2F0
```

Decode opcode
As we have stored our current opcode, we need to decode the opcode and check the opcode table to see what it means. We will continue with the same opcode:

```
0xA2F0 // Assembly: mvi 2F0h
```

If we take a look at the opcode table, it tells us the following:

- ANNN: Sets I to the address NNN

We will need to set index register I to the value of NNN (0x2F0).

Execute opcode
Now that we know what to do with the opcode, we can execute the opcode in our emulator. For our example instruction `0xA2F0` it means that we need to store the value `0x2F0` into index register I. As only 12 bits are containing the value we need to store, we use a bitwise AND operator (&) to get rid of the first four bits (nibble):

```
1010001011110000 & // 0xA2F0 (opcode)
0000111111111111 = // 0x0FFF
------------------
0000001011110000   // 0x02F0 (0x2F0)
```

Resulting code:

```
I = opcode & 0x0FFF;
pc += 2;
```

Because every instruction is 2 bytes long, we need to increment the program counter by two after every executed opcode. This is true unless you jump to a certain address in the memory or if you call a subroutine (in which case you need to store the program counter in the stack). If the next opcode should be skipped, increase the program counter by four.

Timers
Besides executing opcodes, the Chip 8 also has two timers you will need to implement. As mentioned above, both timers (delay timer and sound timer) count down to zero if they have been set to a value larger than zero. Since these timers count down at 60 Hz, you might want to implement something that slows down your emulation cycle (Execute 60 opcodes in one second).

Getting started

Now that you know the basics of emulation and how the system works, it is time to put all pieces together and start coding the emulator.

Initialize system
Before running the first emulation cycle, you will need to prepare your system state. Start clearing the memory and resetting the registers to zero. While the Chip 8 doesn't really have a BIOS or firmware, it does have a basic fontset stored in the memory. This fontset should be loaded in memory location `0x50 == 80` and onwards. More details about how the fontset works can be found at the end of this guide.
Another important thing to remember is that the system expects the application to be loaded at

memory location `0x200` . This means that your program counter should also be set to this location.

```
void chip8::initialize()
{
  pc     = 0x200;  // Program counter starts at 0x200
  opcode = 0;      // Reset current opcode
  I      = 0;      // Reset index register
  sp     = 0;      // Reset stack pointer

  // Clear display
  // Clear stack
  // Clear registers V0-VF
  // Clear memory

  // Load fontset
  for(int i = 0; i < 80; ++i)
    memory[i] = chip8_fontset[i];

  // Reset timers
}
```

Loading the program into the memory

After you have initialized the emulator, load the program into the memory (use fopen in binary mode) and start filling the memory at location: `0x200 == 512` .

```
for(int i = 0; i < bufferSize; ++i)
  memory[i + 512] = buffer[i];
```

Start the emulation

Our system is now ready to execute its first opcode. As mentioned above, we should fetch, decode and execute the opcode. In this example we start by reading the first 4 bits of the current opcode to find out what the opcode is and what the emulator needs to do:

```
void chip8::emulateCycle()
{
  // Fetch opcode
  opcode = memory[pc] << 8 | memory[pc + 1];

  // Decode opcode
  switch(opcode & 0xF000)
  {
    // Some opcodes //

    case 0xA000: // ANNN: Sets I to the address NNN
      // Execute opcode
      I = opcode & 0x0FFF;
      pc += 2;
    break;

    // More opcodes //

    default:
      printf ("Unknown opcode: 0x%X\n", opcode);
  }

  // Update timers
  if(delay_timer > 0)
    --delay_timer;

  if(sound_timer > 0)
  {
    if(sound_timer == 1)
      printf("BEEP!\n");
    --sound_timer;
  }
}
```

In some cases we can not rely solely on the first four bits to see what the opcode means. For example, `0x00E0` and `0x00EE` both start with `0x0` . In this case we add an additional switch and compare the last four bits:

```
// Decode opcode
switch(opcode & 0xF000)
{
  case 0x0000:
    switch(opcode & 0x000F)
    {
```

```
      case 0x0000: // 0x00E0: Clears the screen
        // Execute opcode
      break;

      case 0x000E: // 0x00EE: Returns from subroutine
        // Execute opcode
      break;

      default:
        printf ("Unknown opcode [0x0000]: 0x%X\n", opcode);
    }
  break;

  // more opcodes //
}
```

Opcode examples

Lets take a look at some more opcodes that might look daunting at first.

Example 1: Opcode `0x2NNN`

This opcode calls the subroutine at address NNN. Because we will need to temporary jump to address NNN, it means that we should store the current address of the program counter in the stack. After storing the value of the program counter in the stack, increase the stack pointer to prevent overwriting the current stack. Now that we have stored the program counter, we can set it to the address NNN. Remember, because we're calling a subroutine at a specific address, you should not increase the program counter by two.

```
case 0x2000:
  stack[sp] = pc;
  ++sp;
  pc = opcode & 0x0FFF;
break;
```

Example 2: Opcode `0x8XY4`

This opcode adds the value of VY to VX. Register VF is set to 1 when there is a carry and set to 0 when there isn't. Because the register can only store values from 0 to 255 (8 bit value), it means that if the sum of VX and VY is larger than 255, it can't be stored in the register (or actually it starts counting from 0 again). If the sum of VX and VY is larger than 255, we use the carry flag to let the system know that the total sum of both values was indeed larger than 255. Don't forget to increment the program counter by two after executing the opcode.

```
case 0x0004:
  if(V[(opcode & 0x00F0) >> 4] > (0xFF - V[(opcode & 0x0F00) >> 8])
    V[0xF] = 1; //carry
  else
    V[0xF] = 0;
  V[(opcode & 0x0F00) >> 8] += V[(opcode & 0x00F0) >> 4];
  pc += 2;
break;
```

Example 3: Opcode `0xFX33`

Stores the Binary-coded decimal representation of VX at the addresses I, I plus 1, and I plus 2

I have to confess that I couldn't to figure out how to implement this opcode, so I used TJA's solution.

```
case 0x0033:
  memory[I]     = V[(opcode & 0x0F00) >> 8] / 100;
  memory[I + 1] = (V[(opcode & 0x0F00) >> 8] / 10) % 10;
  memory[I + 2] = (V[(opcode & 0x0F00) >> 8] % 100) % 10;
  pc += 2;
break;
```

Handling graphics and input

Drawing pixels

The opcode responsible for drawing to our display is `0xDXYN`. The Wikipedia description tells us the following:

- Draws a sprite at coordinate (VX, VY) that has a width of 8 pixels and a height of N pixels. Each row of 8 pixels is read as bit-coded starting from memory location I; I value doesn't change after the execution of this instruction. As described above, VF is set to 1 if any screen pixels are flipped from set to unset when the sprite is drawn, and to 0 if that doesn't happen.

As the description of the opcode is telling us, the Chip 8 actually draws on the screen by drawing sprites. It will give us the location of where the sprite needs to be drawn (the opcode tells us which V register we need to check to fetch the X and Y coordinates) and the number of rows (N). The width of each sprite is fixed (8 bits / 1 byte). The state of each pixel is set by using a bitwise XOR operation. This means that it will compare the current pixel state with the current value in the memory. If the current value is different from the value in the memory, the bit value will be 1. If both values match, the bit value will be 0.

```
01000101 ^
11110011 =
----------
10110110
```

Lets assume it the opcode was `0xD003`. This means it wants to draw a sprite at location 0,0 which is 3 rows high. At memory location I, the following values were set:

```
memory[I]     = 0x3C;
memory[I + 1] = 0xC3;
memory[I + 2] = 0xFF;
```

How do these 3 bytes represent a sprite? Take a look at the binary values of each byte:

```
HEX    BIN        Sprite
0x3C   00111100     ****
0xC3   11000011   **    **
0xFF   11111111   ********
```

You should use the binary representation to fill your array (`gfx[]`). However, before setting the value in `gfx[]` using the XOR operator, you will also need to check if any of the pixels changed from 1 to 0. If this is the case, you should set the VF register to 1 (This is basically a test for collision detection).

Example of the implementation of opcode `0xDXYN`

```
 1 case 0xD000:
 2 {
 3   unsigned short x = V[(opcode & 0x0F00) >> 8];
 4   unsigned short y = V[(opcode & 0x00F0) >> 4];
 5   unsigned short height = opcode & 0x000F;
 6   unsigned short pixel;
 7
 8   V[0xF] = 0;
 9   for (int yline = 0; yline < height; yline++)
10   {
11     pixel = memory[I + yline];
12     for(int xline = 0; xline < 8; xline++)
13     {
14       if((pixel & (0x80 >> xline)) != 0)
15       {
16         if(gfx[(x + xline + ((y + yline) * 64))] == 1)
17           V[0xF] = 1;
18         gfx[x + xline + ((y + yline) * 64)] ^= 1;
19       }
20     }
21   }
22
23   drawFlag = true;
24   pc += 2;
25 }
26 break;
```

- Line 3-4: Fetch the position and height of the sprite
- Line 5: Pixel value
- Line 8: Reset register VF
- Line 9: Loop over each row
- Line 11: Fetch the pixel value from the memory starting at location I
- Line 12: Loop over 8 bits of one row

- Line 14: Check if the current evaluated pixel is set to 1 (note that `0x80 >> xline` scan through the byte, one bit at the time)
- Line 16-17: Check if the pixel on the display is set to 1. If it is set, we need to register the collision by setting the VF register
- Line 18: Set the pixel value by using XOR
- Line 23: We changed our `gfx[]` array and thus need to update the screen.
- Line 24: Update the program counter to move to the next opcode

Input

The Chip 8 system uses a simple HEX keypad that allows users to interact with the system. For our emulator this means we need to implement a method that will set the state of each key in the variable that handles the key states. Every cycle you should check the key input state and store it in `key[]`.

It actually doesn't matter what value you store, because opcode `0xEX9E` and `0xEXA1` only check if a certain key is pressed or isn't pressed. Opcode `0xFX0A` only waits for a key press, and when it receives one, it stores the key name in the register and not the key state.

```
case 0xE000:
  switch(opcode & 0x00FF)
  {
    // EX9E: Skips the next instruction
    // if the key stored in VX is pressed
    case 0x009E:
      if(key[V[(opcode & 0x0F00) >> 8]] != 0)
        pc += 4;
      else
        pc += 2;
    break;
```

Below you'll find an example of the original keypad layout. It does not really matter how you implement the key mapping, but I suggest something as on the right side.

```
Keypad                   Keyboard
+-+-+-+-+                +-+-+-+-+
|1|2|3|C|                |1|2|3|4|
+-+-+-+-+                +-+-+-+-+
|4|5|6|D|                |Q|W|E|R|
+-+-+-+-+       =>       +-+-+-+-+
|7|8|9|E|                |A|S|D|F|
+-+-+-+-+                +-+-+-+-+
|A|0|B|F|                |Z|X|C|V|
+-+-+-+-+                +-+-+-+-+
```

CHIP-8 fontset

This is the Chip 8 font set. Each number or character is 4 pixels wide and 5 pixel high.

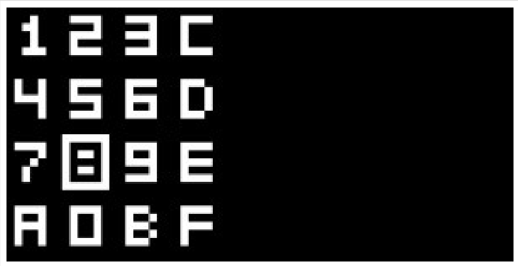
```
unsigned char chip8_fontset[80] =
{
  0xF0, 0x90, 0x90, 0x90, 0xF0, // 0
  0x20, 0x60, 0x20, 0x20, 0x70, // 1
  0xF0, 0x10, 0xF0, 0x80, 0xF0, // 2
  0xF0, 0x10, 0xF0, 0x10, 0xF0, // 3
  0x90, 0x90, 0xF0, 0x10, 0x10, // 4
  0xF0, 0x80, 0xF0, 0x10, 0xF0, // 5
  0xF0, 0x80, 0xF0, 0x90, 0xF0, // 6
  0xF0, 0x10, 0x20, 0x40, 0x40, // 7
  0xF0, 0x90, 0xF0, 0x90, 0xF0, // 8
  0xF0, 0x90, 0xF0, 0x10, 0xF0, // 9
  0xF0, 0x90, 0xF0, 0x90, 0x90, // A
  0xE0, 0x90, 0xE0, 0x90, 0xE0, // B
  0xF0, 0x80, 0x80, 0x80, 0xF0, // C
  0xE0, 0x90, 0x90, 0x90, 0xE0, // D
  0xF0, 0x80, 0xF0, 0x80, 0xF0, // E
  0xF0, 0x80, 0xF0, 0x80, 0x80  // F
};
```

It might look just like an array of random numbers, but take a close look at the following:

```
DEC   HEX    BIN         RESULT    DEC   HEX    BIN         RESULT
240   0xF0   1111 0000    ****     240   0xF0   1111 0000    ****
144   0x90   1001 0000    *  *      16   0x10   0001 0000       *
144   0x90   1001 0000    *  *      32   0x20   0010 0000      *
```

144	0x90	1001 0000	* *	64	0x40	0100 0000	*
240	0xF0	1111 0000	****	64	0x40	0100 0000	*

Look at the left example were we are drawing the number 0. As you can see it see it consists out of 5 values. Of every value, we use the binary representation to draw. Note that only the first four bits (nibble) are used for drawing a number or character.



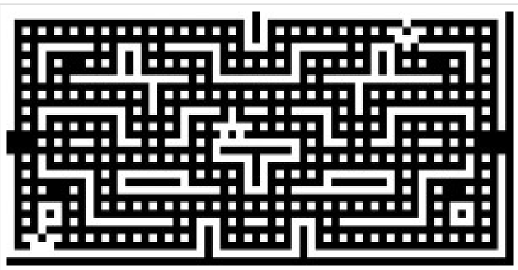
Conclusion

Hopefully this guide provided you enough information to get you started with your own emulator project. At least you should now have a basic understanding of how emulation works and perhaps a better understanding of how a CPU executes opcodes.

I have included my own implementation of a Chip 8 interpreter below which you can use as a reference. The zip file contains a binary for Windows but also includes the full source code of the emulator. Because the full source code is supplied, I recommend only looking at `chip8.cpp` file as a last resort to see how I implemented a particular opcode. The file `chip8.h` and `main.cpp` should be safe to view without spoiling too much. Actually, `main.cpp` mostly contains GLUT code which you can reuse in other (non-emulator related) projects as well.

- myChip8 (27887 downloads) – Latest (Windows binary + Source code)
- Early version of myChip8 from 2003 (contains a nice debugger)
- An Android port I did in 2008

Let me know if you find this guide useful! If you have questions or think that essential parts are missing, please use the comment section ☺ !



Credits
Special thanks to the following persons (many only known by their pseudonym) who have helped me greatly with my own emulation projects in the past and present.

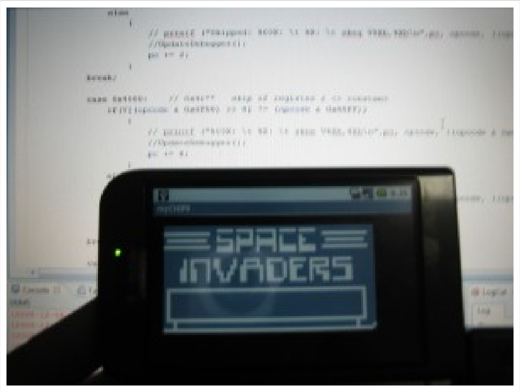
- ector & F|RES (Dolphin-emu team)
- Saqib, zenogais, Absolute0, Shadow, gigaherz, Florin, Goldfinger (PCSX2 team)
- drk||Raziel & ZeZu (nullDC team)
- Hacktarux (Mupen64)
- Muad (nSX2)
- pSXAuthor (pSX)
- Shadowprince, Linker, Aprentice, SculleatR, ShizZy, Dave2001

Suggestions

After you have completed your first Chip 8 emulator, you might want to try one of the following things:

- Add Super CHIP-8 Opcodes
- Use function pointers instead of a giant switch statement
- Improve graphics by adding filters (Hqx)
- Port it to other platforms (Mobile?) or languages

- Move on to a more complex project for example emulating a Gameboy (Z80 processor)



Resources

- http://en.wikipedia.org/wiki/CHIP-8 – System information
- Cowgod's Chip-8 Technical Reference v1.0 – Recommended
- Chip8 tutorial – Goldroad.co.uk (mirror)
- (S)Chip 8 instruction set – Goldroad.co.uk (mirror)
- David Winter's CHIP-8 emulation page – Contains some games
- (S)CHIP-8 instruction set overview – (Erik Bryntse)
- Chip8 emulator topic on Emutalk
- Chip8 emulator topic on NGemu

Advanced emulator resources

- Zilmar's Emubook
  - Emubook CPU emulation chapter
  - Emubook memory emulation chapter

- Zenogais Emulation Tutorials – (Offline, but available through Archive.org)
  - Zenogais' Emulation Tutorials – (mirror)
    - Dynamic Recompiler – (mirror)
    - Array of Function Pointers – (mirror)
    - Introduction to Emulation Part 1 – (mirror)
    - Introduction to Emulation Part 2 – (mirror)
    - Laying the Ground For An Emulator – (mirror)

Bookmark & Share

Share

84 Comment

Martin Ennemoser

Are you sure that you display opcode is correctly implemented?

Take for example the rom from here (but I think it can be reproduced with you provided rom 'pong2.c8' as well):
https://github.com/badlogic/chip8/blob/master/roms/pong.rom

A valid opcode is 'DAB6'. Lets assume V[B] = 0x1e (decimal 30)

This actually overflows you gfx buffer since it starts at y = 30 until y = 35. But as you know your screen just has 32 rows. I think it doesn't crash because its a C++ program and that the memory after gfx is still unused. However, take the C# implementation from Patrick and you will see that Pong will crash as soon as you press 2 on the keyboard and the left "Pong-Bar" reaches the top of the screen. But you can also notice it visually in your implementation because the "Pong-Bars" do not break correctly when they are overflowing the screen.

To fix this, write the inner loop as:

```
for (byte xline = 0; xline > xline)) != 0) // if sprite pixel is set
{
byte posX = (byte) ((x + xline) % WIDTH);
byte posY = (byte)((y + yline) % HEIGHT);

ushort posPixel = (ushort) (posX + ((posY) * 64));

if (gfx[posPixel] == 1)
V[0xF] = 1; // set vf register
gfx[posPixel] ^= 1;
}
}
```

28 FEB 2018 | REPLY

Landon

Hello, I'd like some clarification on a section.

" Before running the first emulation cycle, you will need to prepare your system state. Start clearing the memory and resetting the registers to zero. While the Chip 8 doesn't really have a BIOS or firmware, it does have a basic fontset stored in the memory. This fontset should be loaded in memory location 0x50 == 80 and onwards. More details about how the fontset works can be found at the end of this guide.
Another important thing to remember is that the system expects the application to be loaded at memory location 0x200. This means that your program counter should also be set to this location."

In the example you give for loading the fontset, you don't offset memory[] by 80. However, in the example for loading the game, you do. What's with that?

24 JUN 2018 | REPLY

CDF

Sprites should wrap around, so:
gfx[x + xline + ((y + yline) * 64)] ^= 1;
doesn't work if the sprite position > 64*32 – 8
part of the sprite would be outside of the array changing data from other variables (the stack in my case)

gfx[(x + xline + ((y + yline) * 64)) % (64 * 32)] ^= 1;
Works

15 NOV 2018 | REPLY

Prox

Question, the article says to decode the first 4 bits of the OPCode but only uses the last 4 bits of it. why are the terms switched?

0xF0 means the last 4 bits are 1111 as it represents (end> 11110000 00001111 <start),.

01 FEB 2019 | REPLY

New top story on Hacker News: How to write an emulator (CHIP-8 interpreter) – Tech + Hckr News
John Jason Fallows » Blog Archive New top story on Hacker News: How to write an emulator (CHIP-8 interpreter) - John Jason Fallows
New top story on Hacker News: How to write an emulator (CHIP-8 interpreter) – ÇlusterAssets Inc.,
Porting a CHIP-8 emulator to Rust – NTCore

« Older Comments

Leave a reply

Submit

Tags

adobe air Android application chat conference diy emulation facebook flash games google earth gsoc hacking harvard hci howto infovis IRC kinect latency mma multitouch museum nasa nuigroup projects showcase sparkon surface computing tabletop technology thesis touchlib touch tracer tutorial tv uva

Recent Posts

- NodeBeat featured on the BlackBerry Dev blog
- PlayBook add-on (ofxQNX)
- Rewriting and porting fIRC
- Epic Windmill
- NodeBeat, openFrameworks and Android

Recent Comments

- Hung Hoang on How to repair a damaged PCB board – WD1200
- Porting a CHIP-8 emulator to Rust – NTCore on How to write an emulator (CHIP-8 interpreter)
- Prox on How to write an emulator (CHIP-8 interpreter)
- nickma on How to compile Unison for a Synology DS212+
- nickma on How to compile Unison for a Synology DS212+

About

The website usage statistics for multigesture.net are tracked by Google Analytics.