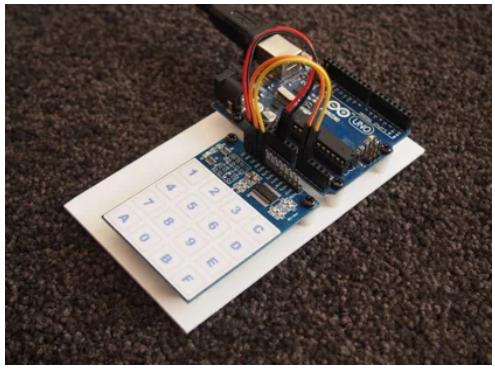
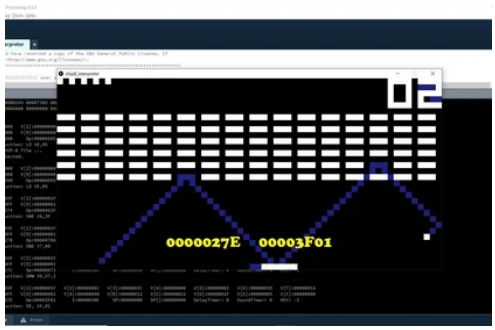
AUTODESK Instructables

CHIP-8 Computer

By lingib in CircuitsArduino

Introduction: CHIP-8 Computer





CHIP-8 is an interpreted programming language for a virtual graphics computer. **[1]**

It was developed by RCA engineer Joseph Weisbecker in the mid-1970's to allow video games to be more easily programmed for the COSMAC VIP 8-bit microcomputer.

Many of these games are in the public domain. **[2]**

This instructable explains how to make a virtual CHIP-8 graphics computer that can play these games using Processing 3 (freeware), an Arduino UNO R3, and a hexadecimal keypad. The keypad, which is optional, allows two players at the same time.

This instructable also explains how to create your own games using CHIP-8

Apart from games my version of CHIP-8 is great for learning how to program as:

- breakpoints may be set
- single-stepping is supported
- all register contents are displayed when single-stepping
- all instructions are disassembled when single stepping

The estimated cost of parts is less than $20.

Images

- The video shows the CHIP-8 virtual computer in action. The game “Jumping Sprite” is developed in this instructable. Deceptively simple ... difficult to master. A public domain game called “BRIX” is also demonstrated in which the idea is to knock out as many bricks as possible before missing the ball with the paddle.
- The CHIP-8 computer is shown in the cover photo.
- Photo 2 shows the external hexadecimal keypad. Construction is simple ... a baseboard, four jumper wires and a wire link. **[3]**
- Photo 3 is a screen shot of the game “BRIX” in single step mode. In this mode the register contents are displayed and the next instruction is disassembled. The blue trace can be turned off in the CHIP-8 header ... but is helpful when debugging.

Notes

[1]

CHIP-8 is an acronym for:

Comprehensive **H**exadecimal **I**nterpretive **P**rogramming – **8**-Bit

CHIP-8 details may be found at https://en.wikipedia.org/wiki/CHIP-8

[2]

There are many public domain CHIP-8 programs ... just Google CHIP-8 to find them.

One such games pack is available from https://www.zophar.net/pdroms/chip8/chip-8-games-...

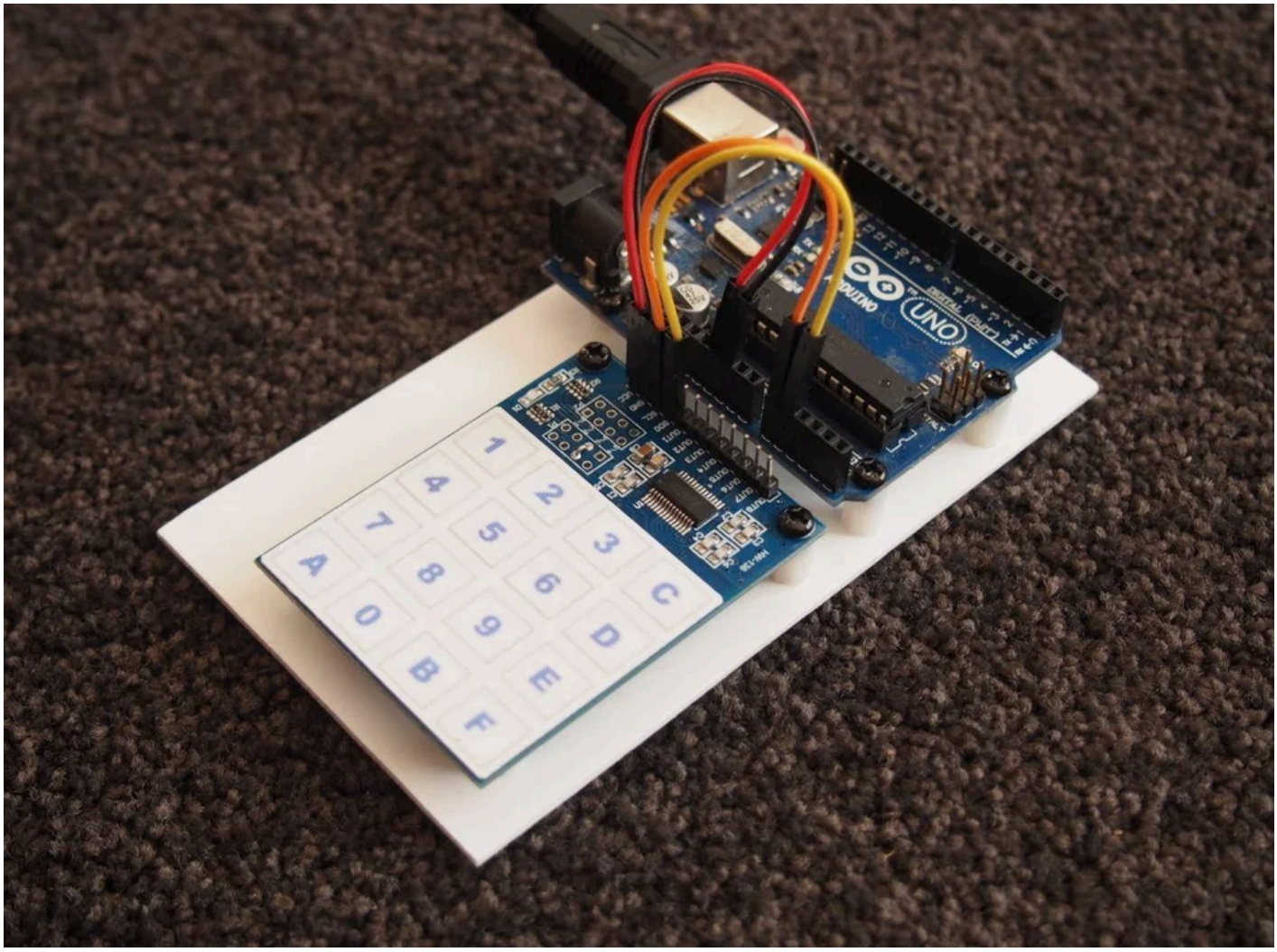
Be aware that programs written for later versions of CHIP-8 may not run as some have extra instructions and a different starting address.

[3]

Construction details for the hexadecimal keypad are described in my instructable https://www.instructables.com/id/Touch-Keypad/

The Arduino keypad software for talking to CHIP-8 is included in this instructable.

Step 1: Parts List



My CHIP-8 software supports two players.

Option 1 (single player)

Nothing is required for this option other than keypad stickers.

CHIP-8 reconfigures your PC keyboard as follows when it is running:

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

The cost for this option is the price of some (optional) keypad stickers

Option 2 (two players)

The external keyboard (photo 1) offers the following advantages:

- two players can play as both keyboards are recognised
- the keypad layout is rectangular rather than slanted.
- you aren't leaning over the CPU keyboard
- you can sit back to watch the screen

The following parts are required for this option:

- 1 onlyArduino UNO R3
- 1 only TTP229 capacitive touch keypad
- 4 only male-female Arduino jumper leads
- 8 only M3 x 9mm theaded nylon spacers
- 12 only M3 x 5mm bolts
- 1 only USB cable
- 1 only piece of scrap plastic for the baseboard

The estimated cost of parts for this option is less than $20.

Installing the hex keypad

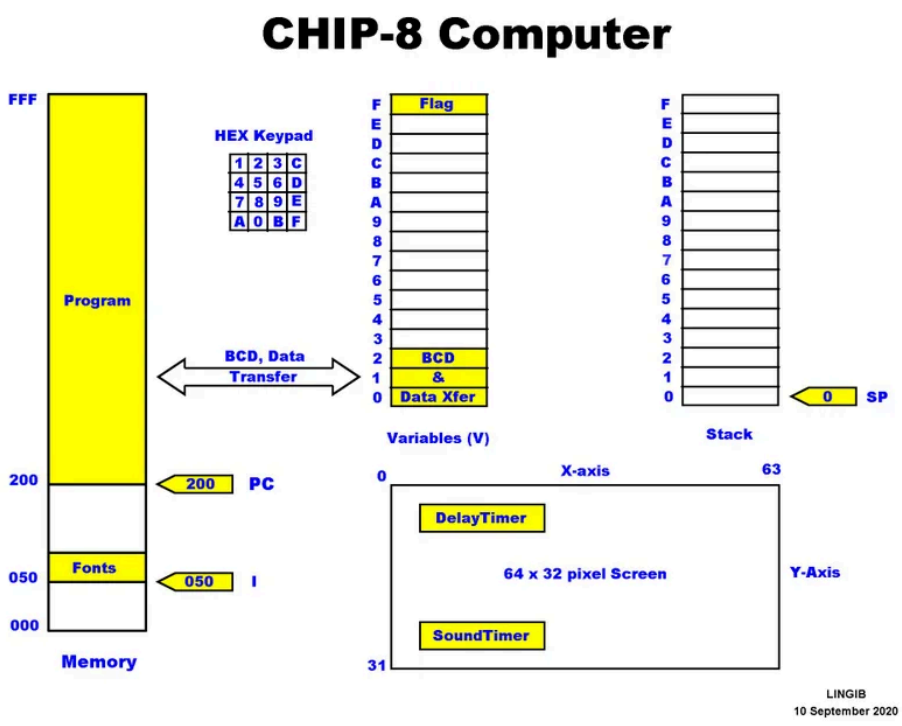
Apart from the four jumper wires you also need to install a wire link.

Construction details are in my instructable https://www.instructables.com/id/Touch-Keypad/.

To activate this keypad

- Download and install the attached file “chip8_keyboard.ino” to your Arduino
- The keypad is recognised by setting “ExternalKeypad = true;” in the CHIP-8 header.

Step 2: CHIP-8 Computer



```
// ----- chip8 Screen Font
int[] Font =
  {
  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
};
```

Photo 1 shows the memory map for my version of the CHIP-8 computer.

CHIP-8 is an imaginary computer that has the following (software) components:

- **ALU** ... arithmetic and logic unit
- **PC** ... program counter
- **SP** ... stack pointer
- **I** ... index register
- **V**[0..F] ... sixteen registers for holding variables

The following hardware components are also emulated in software

- **ROM** read only memory
- **RAM** random access memory
- **Stack** for nesting up to 16 subroutines
- **DelayTimer** for general purpose delays up to 6 seconds
- **SoundTimer** for controlling the sound (beep) duration

Display

Each of the above components are explained in more detail below:

RAM (random access memory)

The random access memory is created in software using a 4096 byte array ... Memory[4096]

This memory is required for

- your programs
- storing registers
- converting hexadecimal numbers to BCD (binary coded decimal)

ROM (read only memory)

This is defined as the first 512 memory locations from **Memory[0]** to Memory[511] or **Memory[1FF]** in hexadecimal.

ROM is normally used for the computer operating system. But CHIP-8 doesn't have an operating system as such because we are using a Processing 3 interpreter.

The only items in this ROM area are the in-built screen fonts. **The fonts start at Memory[050] hexadecimal**

Stack

I have chosen to use a separate array **Stack[32]** (decimal) for the stack rather use Memory[]

The stack pointer **SP,** which **always points to Stack[0] at startup**, is automatically incremented and decremented with each JSR (jump to subroutine) and RET (return from subroutine command.

Display

The CHIP-8 display has 64 horizontal and 32 vertical pixels (picture elements) which means the graphics are somewhat chunky.

The display is refreshed between 50Hz and 60Hz (approx **20mS**)

Timers

CHIP-8 has two 8-bit timers that decrement each time the screen is refreshed and stop when their counts reach zero.

- The **delay timer** is used for general delays up to 5 seconds. (255*20mS)
- The **sound timer** works the same way except that it beeps while the count is above zero.

Fonts

The fonts comprise chunky graphics as shown by the number 7 in photo 2. The screen fonts begin at Memory[050] in hexadecimal.

Each hexadecimal digit is comprises a pattern of 4 x 5 pixels.

Photo 3 shows the HEX (hexadecimal) font patterns for the digits [0-9], [A-F].

If you refer to photo 3 you will see that the pattern for the number '7' comprises the following 5 bytes where each binary '1' represents a screen pixel:

```
0xF0    ****....
0x10    ...*....
0x20    ..*.....
0x40    .*......
0x40    .*......
```

Index Register

The 12-bit index register can point anywhere in memory.

It is used for:

- register to memory data trasfer
- memory to register data transfer
- pointing to fonts
- BCD conversion

Important ... point the index register (I) at unused memory before doing BCD conversion

Keypad

The original CHIP-8 computer used a hexadecimal keypad.

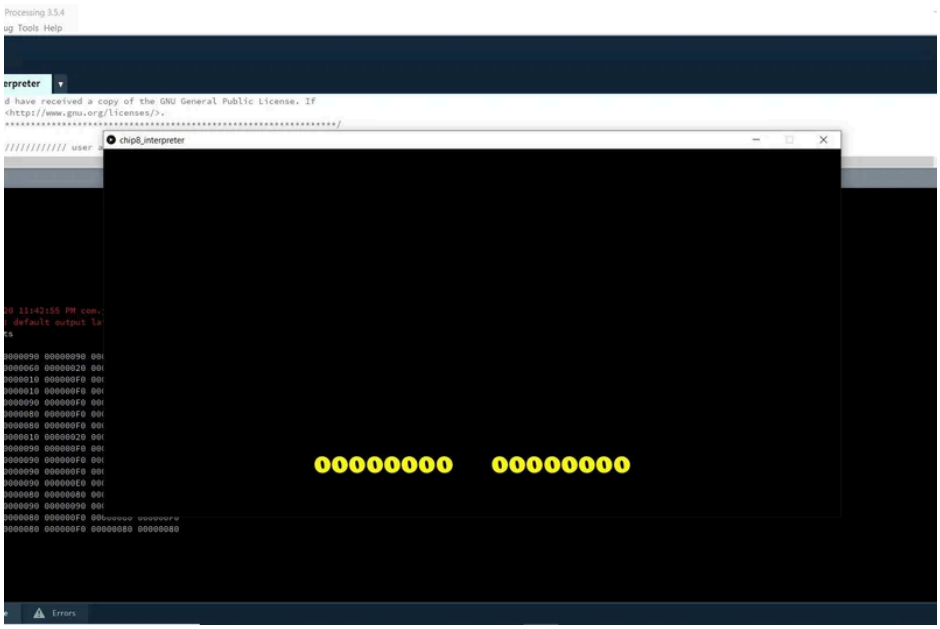
My version of CHIP-8 remaps the following keys on your keyboard to form a HEX keypad.

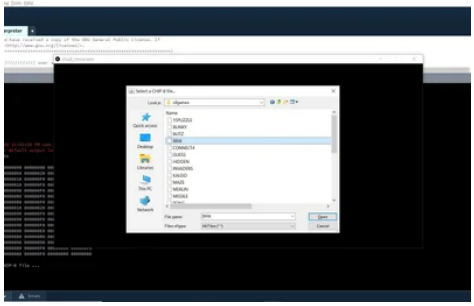
```
1 2 3 4  ===>  1 2 3 C
Q W E R  ===>  4 5 6 D
A S D F  ===>  7 8 9 E
Z X C V  ===>  A 0 B F
```

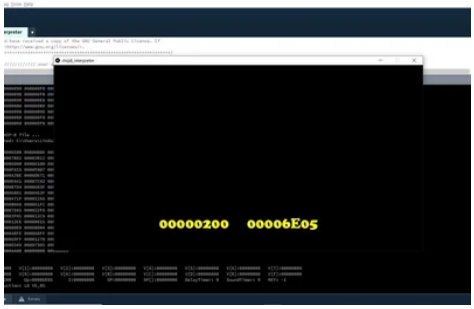
Your PC key layout reverts to normal when you close CHIP-8

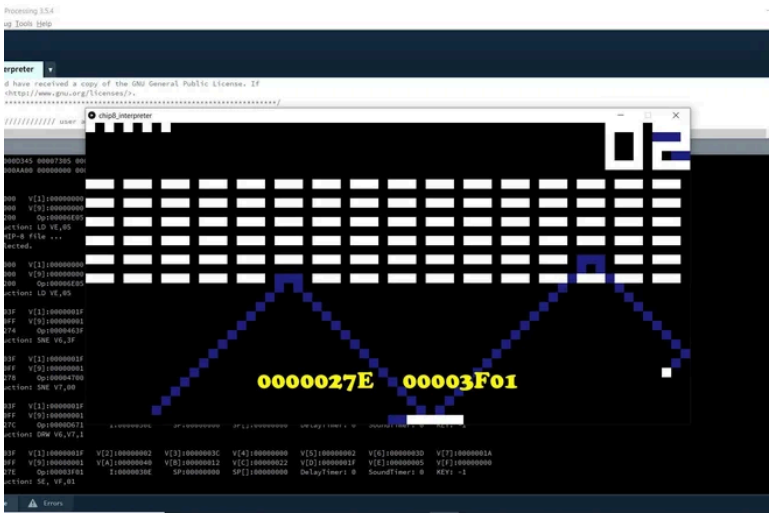
The software also accepts an external keypad should you have two players or wish to sit back from the screen.

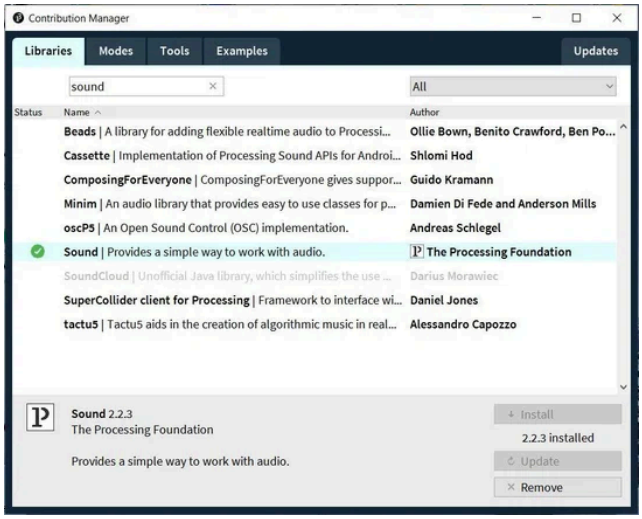
Step 3: CHIP-8 Software











To run CHIP-8 games you need the following software

- Processing 3 from https://processing.org/download/.
- chip8_interpreter.pde .... attached to this instructable
- a games pack from https://www.zophar.net/pdroms/chip8/chip-8-games-...

Each of the above packages are freeware.

Installing the games pack

- Unzip the games pack to your desktop

Installing Processing 3

Processing 3 is similar to Arduino except Processing has a draw() loop .

- Download and install the Processing version for your computer.
- Add a sound library by clicking "Sketch->Import Library -> Add Library"
- Type "Sound" into the "Filter" box
- Now click " Sound -> Install" ... (see photo 5)

Installing CHIP-8

- Run Processing
- Open “chip8_interpreter.pde”
- Open a new sketch
- Copy the contents of “chip8_interpreter.pde” into the sketch and save it using the same name but without the quotes or the file extension.

Running CHIP-8

- Run Processing
- Open “chip8_interpreter.pde”
- Left-click the top-left arrow in Processing ... a screen similar to photo 1 will appear.

Your keyboard has now been remapped:

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

The following keys control the program

- **J** =load file
- **K** = start program
- **L** = stop program

Let’s run a CHIP-8 game from the games pack.

- Press “J” ... this brings up a menu
- Navigate your way to the games pack
- Select BRIX ..... photo 2

The screen will resemble photo 3 when BRIX has loaded

- Press “K” to start the game .... (photo 4)
- Press “L” to stop the game

Notes

The “Q” and “E’ keys control the paddle for BRIX

- Whenever the “L” key is pressed the program stops and displays all of the the register contents. Pressing “L” multiple times allows you to single step through a program
- To restart the game press “K”
- The “blue” trace can be turned off in the CHIP-8 header.
- The “speed” can also be adjusted in the CHIP-8 header

Important ... Always click the CHIP-8 screen once ... this ensures that you don’t accidentally overwrite the CHIP-8 program when you press your keyboard

Step 4: CHIP-8 Instruction Set

CHIP-8 has 35 opcodes, which are all two bytes long and stored big-endian. **[1]**

A detailed instruction set may be down loaded from https://en.wikipedia.org/wiki/CHIP-8

This instruction set was useful when writing the CHIP-8 interpreter but is difficult to use when programming as all of the opcodes are listed in alpha-numeric order.

For ease of coding the attached “**Chip-8 Instruction Set.pdf**” has the instructions grouped by function.

The most-significant-nibble (4 bits) determines the instruction type which leaves a maximum of 12 bits for addresses. The maximum amount of memory is therefore 2^12=4096 bytes.

Depending on the instruction the remaining nibbles are grouped as follows:

- **NNN** a 12-bit pointer address
- **NN** an 8-bit number
- **N** a 4-bit number
- **X** the register number (V[X]) holding the X-axis screen coordinate
- **Y** the register number (V[Y]) holding the Y-axis screen coordinate

(Note: I use the term register and variable V interchangeably)

An instruction such as DXYN means:

- **D** = what to do ... in this case Draw a sprite
- **X** = X is the register holding the X-screen coordinate
- **Y** = Y is the register holding the Y-screen coordinate
- **N** =N is the number of bytes to display from where the index register is pointing

Reserved Registers

The following registers should be treated as reserved

- V[0] used for BCD (Binary Coded Decimal) and register to memory transfer
- V[1] used for BCD (Binary Coded Decimal) and register to memory transfer
- V[2] used for BCD (Binary Coded Decimal) and register to memory transfer
- V[F] collision detector and flag register for arithmetic operations

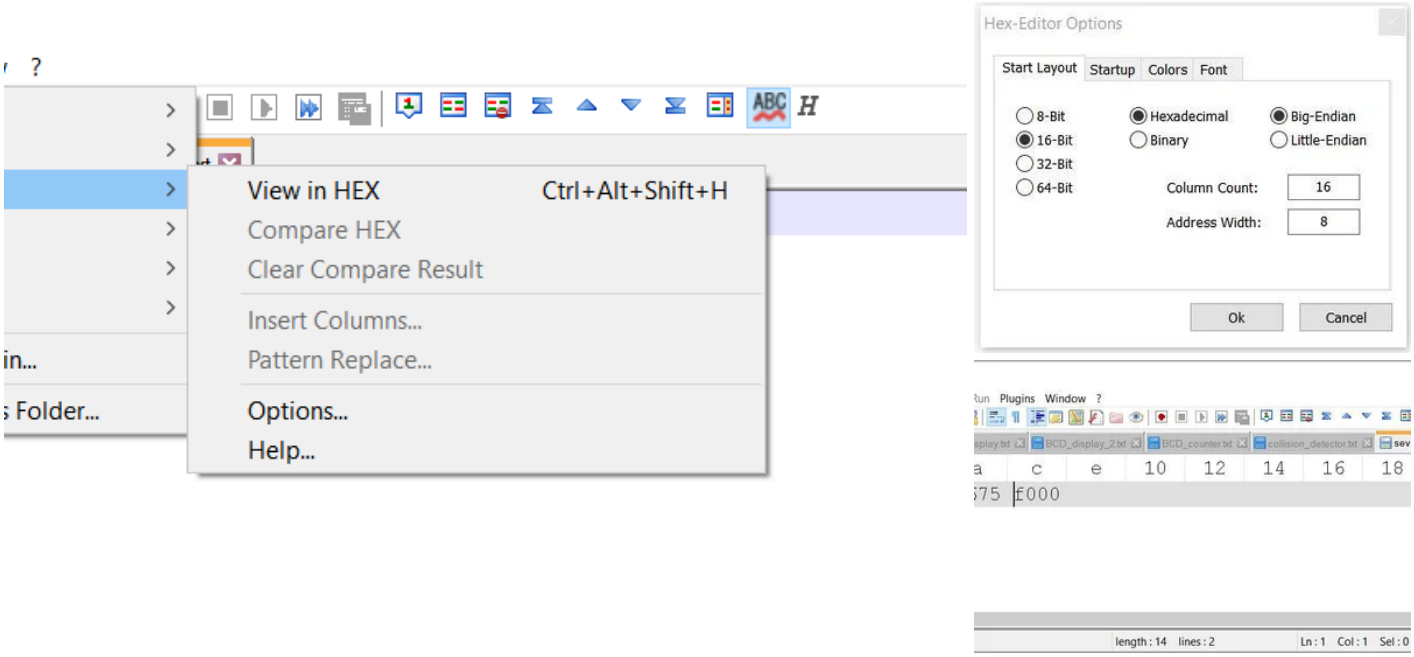
In general it is better to use the high numbered registers as any transfer of bytes from memory starts with V[0]

Note

[1]

Big-endian is an order in which the "big end" (most significant value in the sequence) is stored first (at the lowest storage address). Little-endian is an order in which the "little end" (least significant value in the sequence) is stored first.

Step 5: HEX Editor



A hexadecimal editor is required if you wish to write your own software.

Installing the editor software

- Download and install **Notepad++** from https://notepad-plus-plus.org/downloads/
- Download, and unzip, the file " HexEditor_0.9.6_x64.zip" from https://sourceforge.net/projects/npp-plugins/files...
- The file that we are interested in is called **HexEditor.dll**
- Open Notepad++
- Left-click "Plugins >>Open Plugins Folder" ..... (photo1)
- Create a new folder called HexEditor using "right-click >> new>>folder"
- Copy & paste HexEditor.dll into the HexEditor folder
- Restart Notepad ++

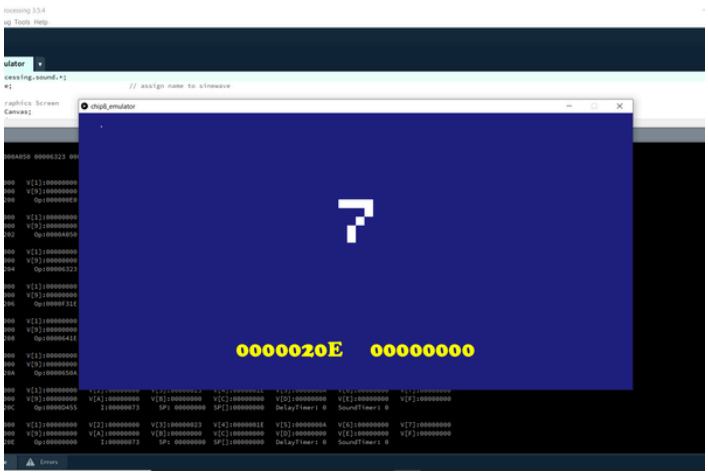
Configuring the Plugin

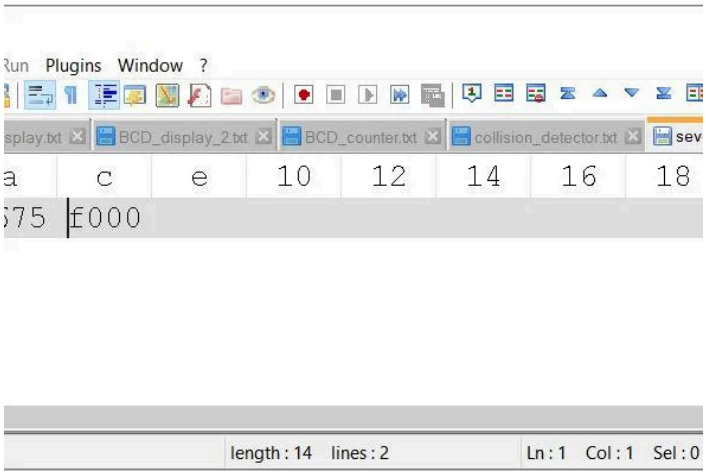
- Left-click "Plugins>>HEX-Editor>>Options"
- Duplicate the settings shown in photo2
- CHIP-8 opcode will now appear in groups of four HEX digits as shown in photo 3.

Notes

- Notepad++ defaults to a plain text editor whenever a file is opened.
- To view/edit/create a HEX file you must left-click "Plugins>>HEX-Editor>>View In HEX"
- The HEX editor has one minor quirk ... everything is displayed in lower case !!

Step 6: Creating Your First CHIP-8 Program





Let’s create the image shown in photo 1.

- Enter the numbers shown in photo 2
- Save the file as seven.txt

These number have the following meaning:

```
00e0    clear the screen
661e    put the hex number 1e into register 6 ... the X screen coordinate is now 30 decimal
670a    put the hex number 0a into register 7 ... the Y screen coordinate is now 10 decimal
6807    put the hex number 7 into register 8 ... this is the number we want to display
f829    point the index register at the graphic representing the number in register 8
d675    display 5 graphic lines (i.e. the 7 sprite) at the location stored in registers 6 and 7
f000    stop
```

Running your own code is exactly the same as running the BRIX software in Step 3:

- Press “J” and select “seven.txt”
- Press “K” to run
- Press “L” to stop

Photo 1 shows the resulting screen pattern ... amazing for only seven instructions !!!

The attached file "seven.txt" contains the above code

A number of common code examples follow.

Step 7: Keypad Test

This program demonstrates:

- the random number generator
- the keyboard skip-if- equal instruction
- the use of jump instructions and
- how to erase a sprite

The screen number will change whenever your key-press matches the on-screen number.

The annotated code to achieve this is shown below:

```
200     00e0            Clear the screen
202     601C            Store the X coordinate in register V0
204     610D            Store the Y coordinate in register V1
206     C20F <------+   Generate a random number between 0 .. F in register V2
208     F229        |   Point the I register at the sprite-shape for this number
20A     D015        |   Display the number by flipping the screen pixels
20C     E29E <--+   |   Skip the next instruction if your keypress matches the number
20E     120C ---+   |   Jump to the program instruction at address 20C
210     D015        |   Erase the number by flipping all the pixels back to their original state
202     1206 -------+   Jump to the random number generator at address 206
```

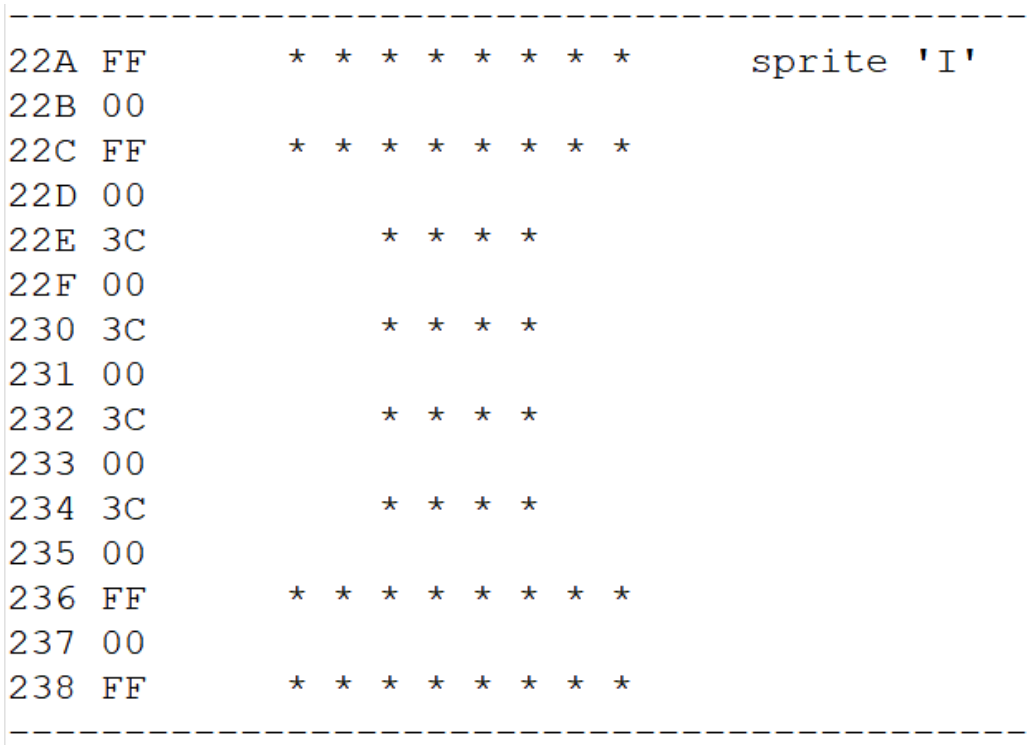
This program is far more complex than the “7” program yet only requires 10 instructions !!!

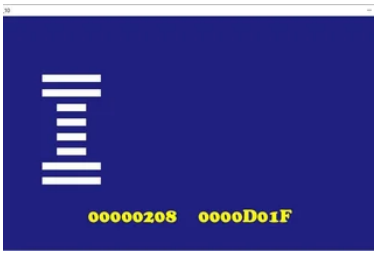
The attached file "keypad test.tx" contains the above code.

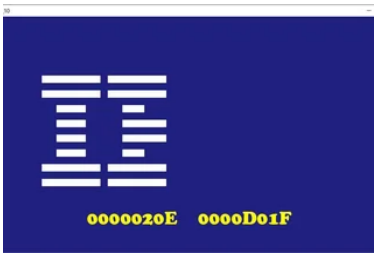
Notes

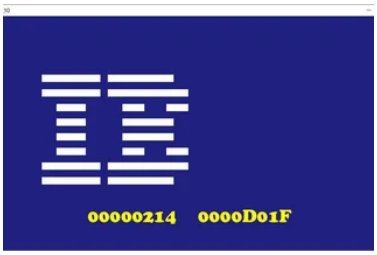
- Since the program is in an infinite loop it doesn’t require a stop instruction
- The F in the instruction at address 206 is a “mask” that restricts the numbers to the range 0..F
- The program has two jump instructions ... one “polls” the keyboard until a matching key is pressed ... the other restarts the number generator
- The instruction at address 210 erases the number by displaying it a second time

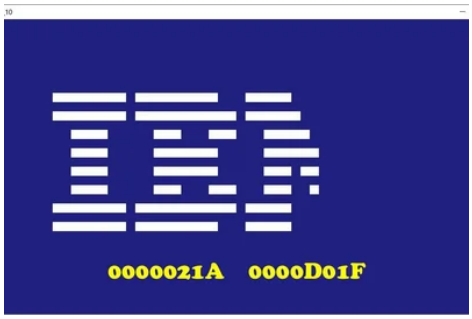
Step 8: Sprites

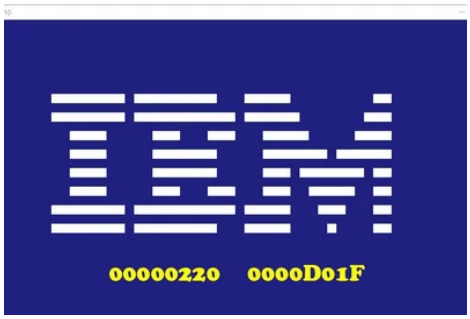


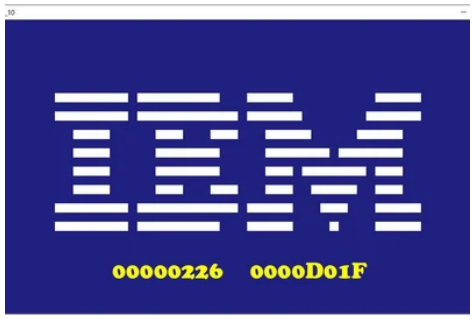












Everything you see on a CHIP-8 screen is created using “sprites”**[1]**

- The letter “I” shown in photo 2 was created using the 8 x 15 sprite-pattern shown in photo 1
- Photos 3 through 7 show how adjacent sprites can be used to create an image.

Important

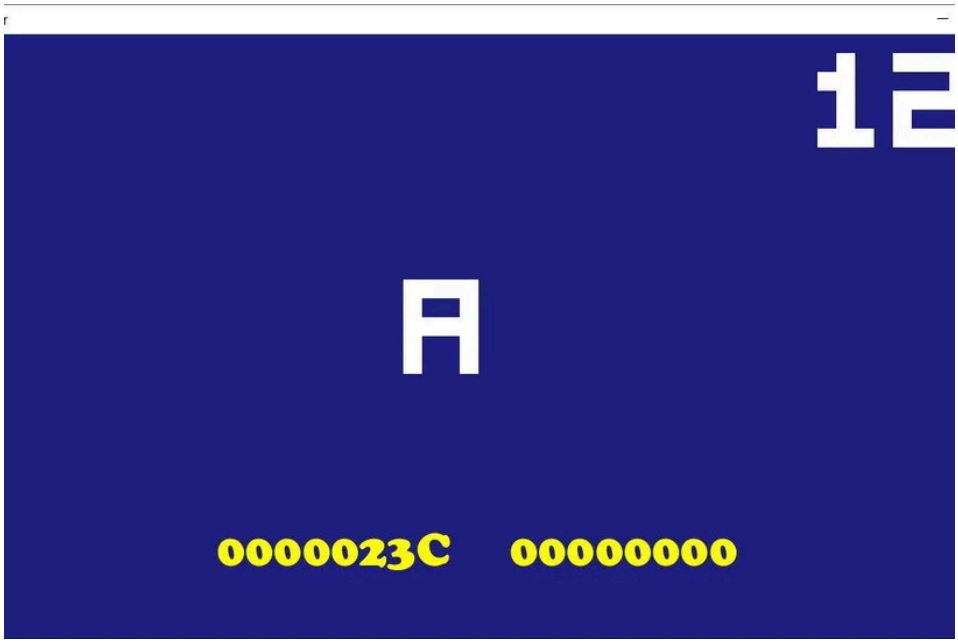
- Each 1-bit in a sprite flips the screen pixel beneath.
- To remove a sprite image simply display it again and all of the screen pixels will flip back to their original state.

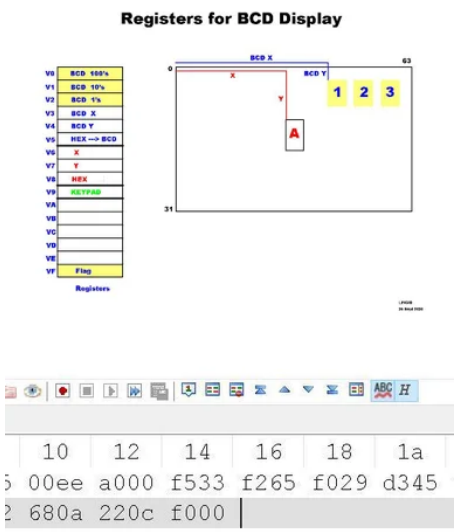
Notes

[1]

A sprite is a graphic image that a user can interact with and move around. The term was first used by Danny Hillis at Texas Instruments in the late 1970s as it floats over the screen in a fairy-like manner.

Step 9: BCD Display





This step introduces:

- BCD (binary-coded-decimal) conversion
- subroutines

This example converts hexadecimal 7B to 123 decimal and display each of the three digits in the top right corner of the display.

It also displays the hexadecimal digit A in the screen center

Images:

- Photo 1 shows the screen we wish to draw
- Photo 2 shows the memory map needed to achieve our goal
- Photo 3 shows the hexadecimal code

We are free to place our variables in any register except V0,V1,V2, & VF

- V0,V1,V2 are required for the BCD (binary coded decimal) conversion
- V0 is used for the 100's
- V1 is used for the 10's
- V2 is used for the units
- VF is reserved for flags which we will use in the next exercise.

The annotated code to achieve this follows:

```
/////////////// this is similar to setup() in Arduino /////////////////
200     00E0            clear the screen
20A     122C            jump past the following two subroutines

/////////////// display random number subroutine ///////////////////
206     661C            put RND X into register V6
208     670D            put RND Y into register V7
20C     F829            point to sprite
20E     D675            display sprite
210     00EE            return from subroutine

/////////////// display BCD numbers subroutine ////////////////////
212     A000            point to conversion area   ..................... (see note 1)
```

```
214     F533            convert the number in register V5 to BCD ........ (see note 1)
216     F265            fill registers V0 to V2 inclusive with BCD

202     6331            put screen X coordinate into register V3 (31 hex = 49 decimal)
204     6401            put screen Y coordinate into register V4
218     F029            point 100's sprite ............................. (see note 2)
21A     D345            display 100's sprite ............................ (see note 2)

21C     7305            move BCD X to the 10's coordinate by adding 5
21E     F129            point to the 10's sprite ........................ (see note 2)
220     D345            display the 10's sprite ......................... (see note 2)

222     7305            move BCD X to the 1's coordinate by adding 5
224     F229            point to the 1's sprite ......................... (see note 2)
226     D345            display the 1's sprite .......................... (see note 2)
22A     00EE            return from subroutine

////////////////// display 123 ////////////////////////////
22C     657B            put hex 7B (123 decimal) into register V5
22E     2212            convert and display using subroutine at address 212

//////////////////// display A (hexadecimal) ////////////////////////
230     680A            put hex 0A into register V8
232     220C            display 0A using subroutine at address 20C

////////////////////// task complete ////////////////////////////////////
234     F000            stop ... enter debug mode
```

The above code is contained in the attached file "BCD_test.txt:

Notes

[1]

To erase the numbers we just need to replace the last instruction with

```
234     2212            redraw the BCD numbers using the subroutine at address 212
236     220C            redraw the number 0A using the subroutine at address 220C
238     F000            stop ... enter debug mode
```

The reason this works is that the numbers stored in registers V0,V1,V2, and register V8 have not changed.

This modified code is contained in the file "BCD_test_2.txt"

(Hint: you may wish to single-step through this program using the “L” key.)

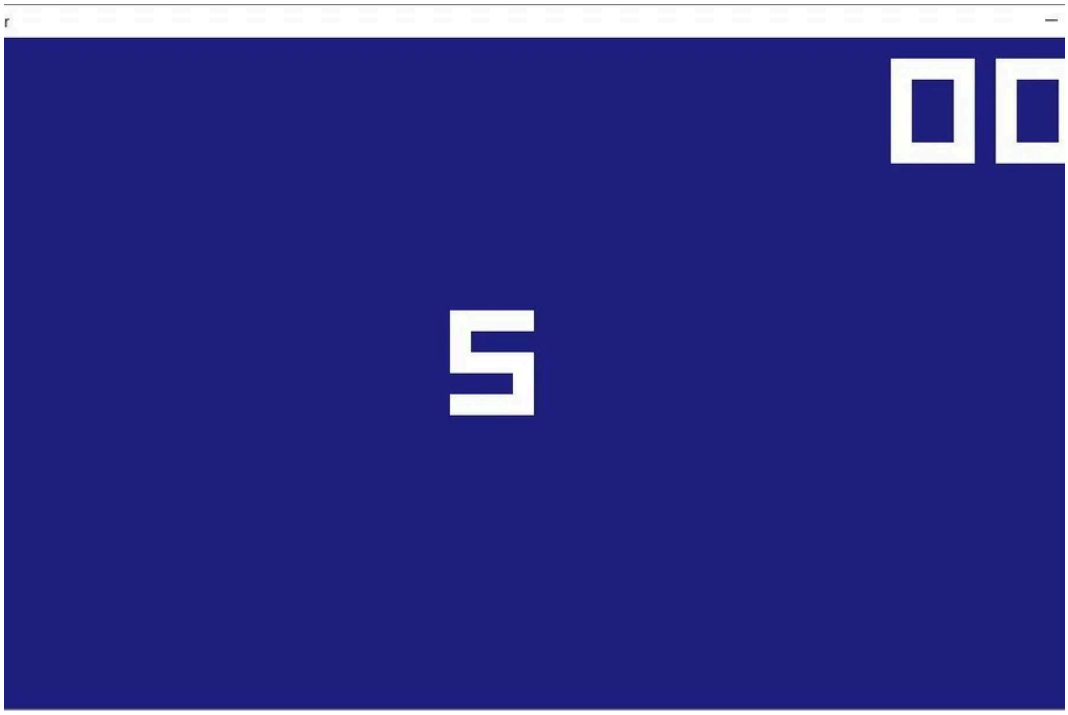
[2]

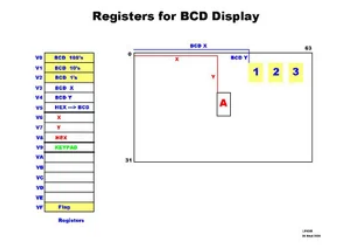
We don’t want to be rewriting our subroutine code every time we enter a program.

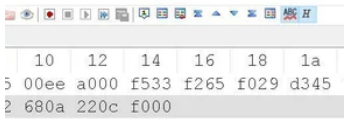
Placing them near the start and jumping around them means that you only have to edit the last few lines of code when creating a new program which means that you will wind up with a library full of useful subroutines.

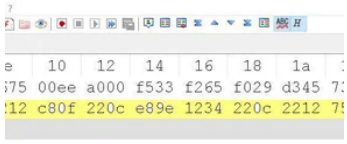
We demonstrate this in the next example.

Step 10: BCD Counter









Download and run the attached CHIP-8 file “BCD_counter.txt”

This example adds 1 to the counter each time you press a key that matches the central number.

It also demonstrates

- random numbers
- erasing sprites
- the use of existing code

Images:

- Photo 1 shows the screen we wish to draw ... the counter is set to zero
- Photo 2 shows the memory map needed to achieve our goal
- Photo 3 shows the code from the previous example
- Photo 4 shows the the code for this example. Apart from the text in yellow highlight, the code is the same as the previous example. This is possible because we placed our code at the beginning.

The annotated code for this example is shown below:

```
/////////////// display the counter ////////////////////
22C     6500    reset the counter by changing V5 to zero
22E     2212    convert and display V5 using the subroutine at address 212

/////////////// display a random number ////////////////////
230     C80F    generate a random number in register V8.
232     220C    convert and display V8 using the subroutine at address 20C

/////////////// look for a matching key press ////////////////////
234     E89E    skip the next instruction if the key press equals the number in V8
236     1234    jump back to address 234 if the key isn’t the same as the number in V8

/////////////// erase all digits ////////////////////
238     220C    erase the random number by displaying it again
23A     2212    erase the three counter digits by displaying them again

/////////////// increment the counter ////////////////////
```

```
23C     7501    add 1 to the counter value held in register V5
23E     122E    jump to address 22E which displays the BCD contents of register V5
```

Step 11: A Simple Game (1) ... the Rules

Let’s make a simple game using two sprites.

At the end of this project you should have a better understanding of:

- sprites
- random numbers
- masks
- delay timers
- sound timers
- BCD counters
- keyboard logic
- collision flags

The attached file **"Jumping Sprite.txt"** contains the code for this game should you wish to try it now.

Game Rules

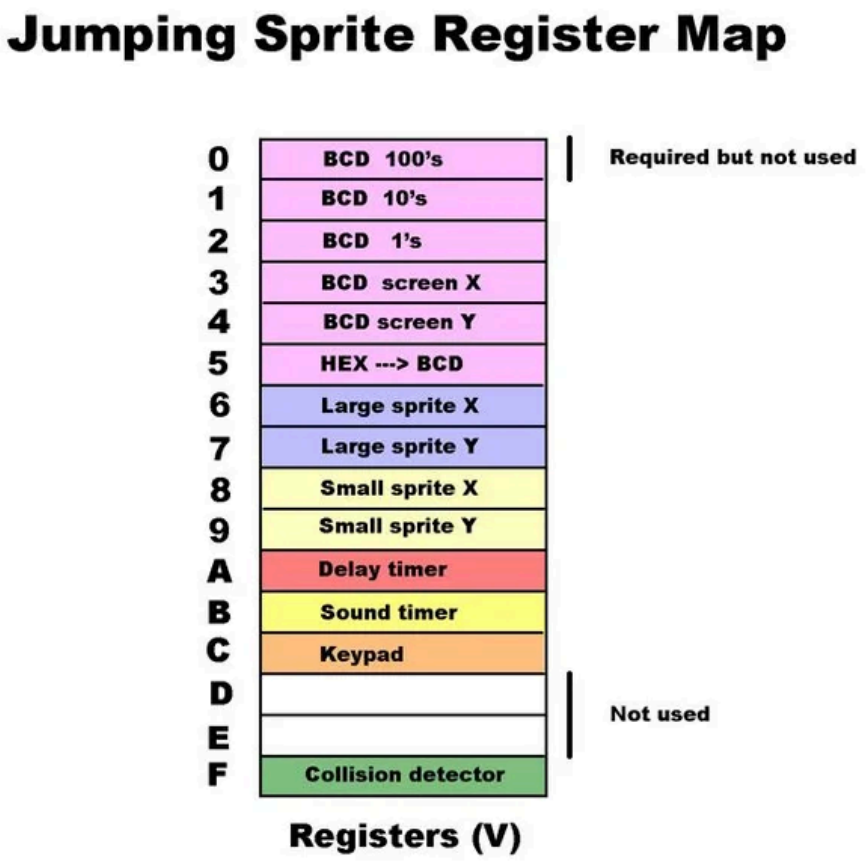
- The position of a large sprite is controlled by a random number generator.
- The large sprite moves to a different position every 5 seconds.
- The position of a smaller sprite is controlled by your keypad:
- 2=up
- 4=left
- 6=right
- 8=down

Your task is to make the smaller sprite collide with the larger sprite using the keys on your keypad

If a collision occurs:

- the large sprite disappears
- the sound timer emits a beep
- 1 is added to your counter and
- the large sprite then reappears in another position

Step 12: A Simple Game (2) ... Getting Ready



Register Map:

- Before starting any project you need to allocate a register to every task.
- The register map for our "Jumping Sprite" program is shown in photo 1

We also need to understand how the timers work

The sound timer:

- The sound timer requires the use of a dedicated register ... we will use register C
- Numbers sent to register C must be copied to the SoundTimer in Processing
- A beep is emitted when we send a number to the SoundTimer
- The number in the SoundTimer is automatically decremented every 20mS.
- The beep stops when the SoundTimer reaches zero
- A number of 0x06 produces a 6*20mS =120mS beep
- The contents of register C are not altered

The delay timer:

- The delay timer requires a dedicated delay register ... we will use register D
- Numbers sent to register D must be copied to the DelayTimer in Processing
- The DelayTimer in Processing is decremented every 20mS
- Unlike the SoundTimer there is no beep.
- The DelayTimer must be “polled” to determine if it has reached zero
- A number of 0x64 produces a 100*20mS= 2 second delay
- The number in rgeister D is not altered.

We are now ready to code:

To understand the game let's split "Jumping Sprite.txt" up into five parts where each part is a a collection of subroutines.

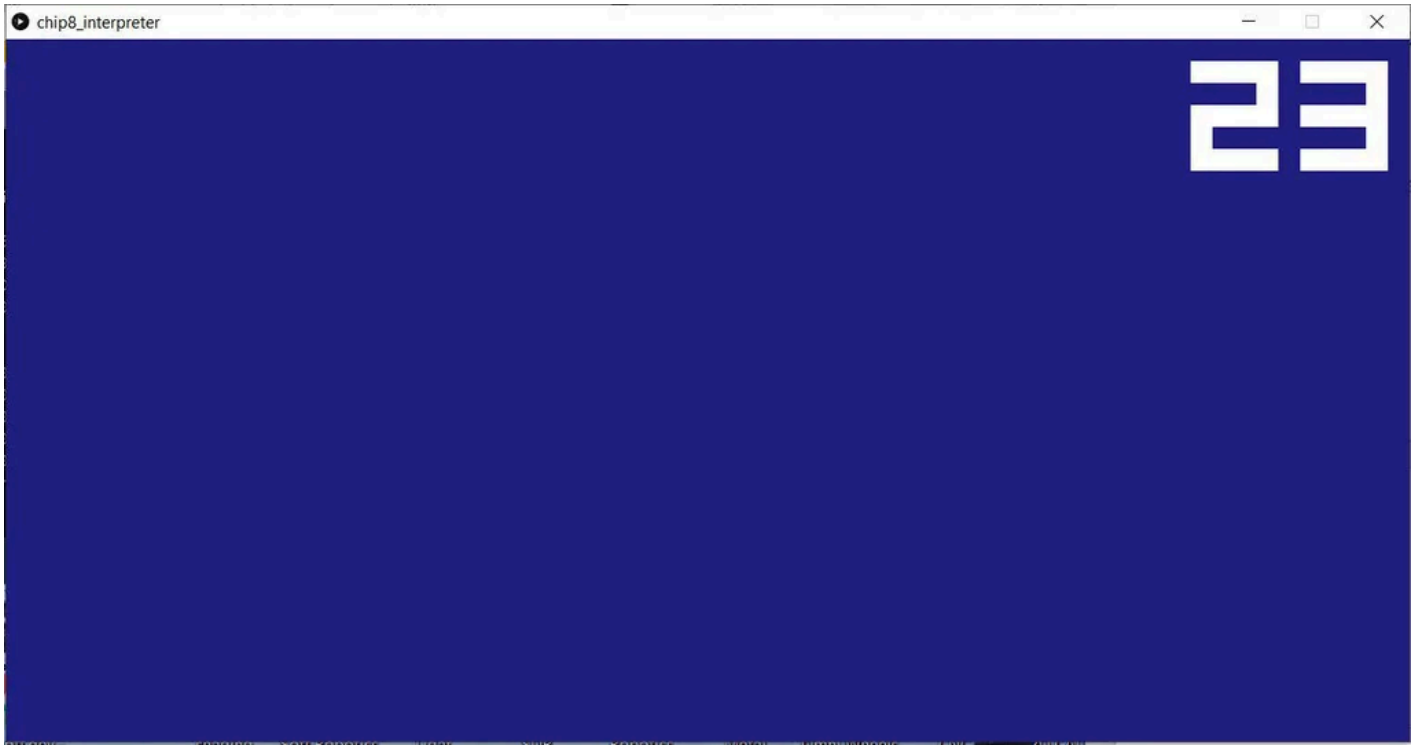
In each of the following steps we add and test more subroutines until our program is complete.:

Method:

To save a lot of typing:

- open a fresh copy of the file "Jumping Sprite.txt" with your hexcecimal text editor
- overwrite the Jumping Sprite code with the "test code" in each of the four parts starting at the address shown in code line 202 ... don't worry about the code you haven't overwritten.
- you must also overwrite code line 202 with the new start address in shown in each part.
- save your modified code using a different "filename"
- now run your new filename using the "JKL" keys in Processing

Step 13: A Simple Game (3) ... the BCD Display



The BCD display

Warning

Be aware that the BCD instruction **FX33** expects the index register **I** to be pointing at a free area in memory for making calculations.

If you forget to point the index register **I** to a safe area in memory then the FX33 instruction may overwrite parts of your program code.

This is an extremely difficult bug to find ...

Lets convert hexadecimal number 7B into its BCD form (123 decimal) .

The annotated code for this is:

```
200     00E0    clear screen
202     121A    jump to start

/////////////// display BCD numbers subroutine ////////////////////
*  The user is responsible for pointing the index register I at clear memory *
204     6336    set screen X coordinate for 10' s digit ..... (36 hex = 54 decimal)
206     6401    set screen Y coordinate
208     A000    .......... point register I at scratch-pad memory
20A     F533    .......... uses scratchpad to do BCD conversion
20C     F265    ...........fills registers V0 to V2 inclusive with BCD from scratchpad
20E     F129    ........... automatically points register I to the 10' s sprite
210     D345    display the 10' s sprite
212     7305    move BCD X to the 1' s coordinate by adding 5
214     F229    ........... automatically points register I to the 1' s sprite
216     D345    display the 1' s sprite
218     00EE    RTS ... return from subroutine

////////////////// test code //////////
/* we remove this test code once the above code has been tested */
21A     657B    store 123 decimal into register V5
21C     2204    JSR 204 ... display last 2 digits of 123 .... BCD turns on
```

```
21E     2204    JSR 204 ... display last 2 digits of 123 .... BCD turns off
220     F000    stop
```

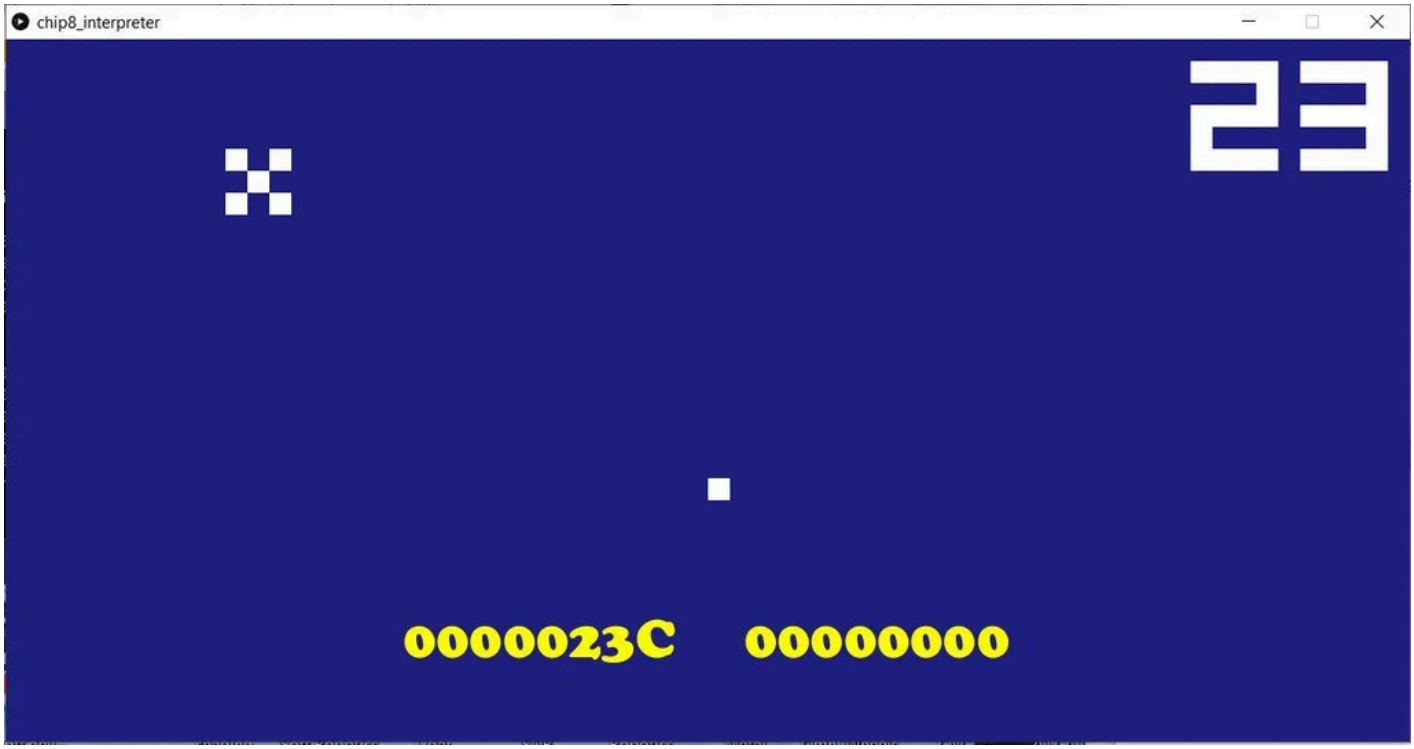
Your screen should look like photo 1 when you run the above code.

Notes:

- The index register **I** points to three different areas of memory
- Code line 208 **POINTS** the index register to a safe area of memory **BEFORE** we use the **FX33** instruction
- Code line 21E erases the BCD display by displaying it a second time.
- Use the “L” single-step key to see this in action.

In the next step we create our own sprites.

Step 14: A Simple Game (4) ... Creating Sprites



Sprites

Let's now create some sprites.

Instead of writing new code we replace the test code in part 1 with our new subroutine(s) then change code line 202 to point at our new test code.

The annotated code to achieve this is shown below:

```
200     00E0    clear screen
202     122A    jump to start of test code

/////////////// display BCD numbers subroutine ////////////////////
/*
   The user is responsible for pointing the index register I at clear memory
*/
204     6336    set screen X coordinate for 10's digit ..... (36 hex = 54 decimal)
206     6401    set screen Y coordinate
208     A000    .......... point register I at scratch-pad memory
20A     F533    .......... convert V[5] content to BCD
20C     F265    ...........fills registers V0 to V2 inclusive with BCD from scratchpad
20E     F129    ........... automatically points register I to the 10's sprite
210     D345    display the 10's sprite
212     7305    move BCD X to the 1's coordinate by adding 5
214     F229    ........... automatically points register I to the 1's sprite
216     D345    display the 1's sprite
218     00EE    RTS ... return from subroutine

//////////////////// large sprite pattern ///////////////////
21A     A0      *.*....         large sprite pattern
21B     40      .*.....
21C     A0      *.*.....

//////////////////// small sprite pattern ///////////////////////
21D     80      *.......        small sprite pattern

//////////////////// display large sprite ///////////////////////
21E     A21A    point register I at the large sprite
220     D673    display 3 lines of the large sprite at coordinate V[6],V[7]
```

```
222     00EE            RTS ... return from subroutine

////////////////////// display the small sprite ////////////////
224     A21D    point register I at the small sprite
226     D891    display 1 line of the small sprite at coordinate V[8],V[9]
228     00EE    RTS ... return from subroutine

////////////////////// test code ///////////////////////
/* Note:
    - that code line 202 now points to the first line of this test code
    - no other code lines have changed.
*/
// ----- convert 7B hex to 123 BCD
22A     657B    store 123 decimal into register V5
22C     2204    JSR ... display last 2 digits of 123</p><p>// ------ draw large sprite
22E     660A    put the X coordinate for the large sprite into register V6
230     6705    put the Y coordinate for the large sprite into register V7
232     221E    JSR ... draw large sprite </p><p>// ----- draw small sprite
234     6820    put the X coordinate for the small sprite into register V8
236     6914    put the Y coordinate for the small sprite into register V9
238     2224    JSR ... draw the small sprite
23A     F000    stop
```

Your screen should look like photo 1 when you run the above code.

Step 15: A Simple Game (5) ... Make the Large Sprite Jump

This code block demonstrates the use of:

- The **delay timer** to make the large sprite change location every 5 seconds.
- **Masks** to prevent the large sprite from disappearing off the screen

```
200     00E0    clear screen
202     123E    jump to start of test code

/////////////// display BCD numbers subroutine ////////////////////
/*
   The user is responsible for pointing the index register I at clear memory
*/
204     6336    set screen X coordinate for 10's digit ..... (36 hex = 54 decimal)
206     6401    set screen X coordinate for 1's digit
208     A000    .......... point register I at scratch-pad memory
20A     F533    .......... convert V[5] content to BCD
20C     F265    ...........fills registers V0 to V2 inclusive with BCD from scratchpad
20E     F129    ........... automatically points register I to the 10's sprite
210     D345    display the 10's sprite
212     7305    move BCD X to the 1's coordinate by adding 5
214     F229    ........... automatically points register I to the 1's sprite
216     D345    display the 1's sprite
218     00EE    RTS ... return from subroutine

////////////////////// large sprite pattern ///////////////////
21A     A0      *.*....         large sprite pattern
21B     40      .*.....
21C     A0      *.*.....

////////////////////// small sprite pattern ///////////////////////
21D     80      *.......        small sprite pattern

////////////////////// display large sprite ///////////////////////
21E     A21A    point register I at the large sprite
220     D673    display 3 lines of the large sprite at coordinate V[6],V[7]
222     00EE            RTS ... return from subroutine

////////////////////// display the small sprite ////////////////
224     A21D    point register I at the small sprite
226     D891    display 1 line of the small sprite at coordinate V[8],V[9]
228     00EE    RTS ... return from subroutine

//////////////////  random sprite move  //////////////
22A     FA07    read DelayTimer into register VA
22C     3A00    skip next instruction if VA == 0
22E     00EE    RTS  ... return from subroutine

230     221A    redraw the large sprite ... erases large sprite

232     C63D    set random X coordinate  (3D mask keeps numbers in range 0..61)
234     C71D    set random Y coordinate  (1D mask keeps numbers in range 0..29)
236     221A    draw new large sprite

238     6AFF    put new delay length into register VA
23A     FA15    set DelayTimer to register VA contents
23C     00EE    RTS ... return from subroutine

////////////////////// test code ///////////
23E     657B    store 123 decimal into register V5
240     2204    JSR ... display last 2 digits of 123

242     6820    put the X coordinate for the small sprite into register V8
244     6914    put the Y coordinate for the small sprite into register V9
246     2224    JSR .... draw the small sprite
```

```
248     C63D    set random X coordinate  (3D mask means 0..60 + 2 sprite columns)
24A     C71D    set random Y coordinate  (1D mask means 0..28 + 2 sprite rows)
24C     221E    JSR ... draw large sprite

24E     6AFF    put new delay length into register VA
250     FA15    set DelayTimer to register VA contents

252     222A    JSR ... draw random sprite
254     1252    infinite random sprite loop
```

The sprite should now jump to a new location every five seconds

Points to note:

- the mask of 0x3D (61 decimal) in code lines 230, 246 prevent the sprite going beyond the right-hand screen edge
- The mask 0f 0x1D (29 decimal) in code lines 232, 248 keeps the sprite above the bottom screen edge
- You should now begin to see a pattern ... make a small section of your code work then add some more.

Step 16: A Simple Game (6) ... Moving the Small Sprite

Let's add a keypad routine to the code in step 3 where:

- keypad 2 = moves the small sprite up
- keypad 4 = moves the small sprite down
- keypad 6 = moves the small sprite left
- keypad 4 = moves the small sprite right

The annotated code follows:

```
200     00E0    clear screen
202     1290    jump to start of test code

/////////////// display BCD numbers subroutine ////////////////////
/*
   The user is responsible for pointing the index register I at clear memory
*/
204     6336    set screen X coordinate for 10's digit ..... (36 hex = 54 decimal)
206     6401    set screen X coordinate for 1's digit
208     A000    .......... point register I at scratch-pad memory
20A     F533    .......... convert V[5] content to BCD
20C     F265    ...........fills registers V0 to V2 inclusive with BCD from scratchpad
20E     F129    ........... automatically points register I to the 10's sprite
210     D345    display the 10's sprite
212     7305    move BCD X to the 1's coordinate by adding 5
214     F229    ........... automatically points register I to the 1's sprite
216     D345    display the 1's sprite
218     00EE    RTS ... return from subroutine

////////////////////// large sprite pattern ///////////////////
21A     A0      *.*....         large sprite pattern
21B     40      .*.....
21C     A0      *.*.....

////////////////////// small sprite pattern ///////////////////////
21D     80      *.......        small sprite pattern

////////////////////// display large sprite ///////////////////////
21E     A21A    point register I at the large sprite
220     D673    display 3 lines of the large sprite at coordinate V[6],V[7]
222     00EE            RTS ... return from subroutine

////////////////////// display the small sprite ////////////////
224     A21D    point register I at the small sprite
226     D891    display 1 line of the small sprite at coordinate V[8],V[9]
228     00EE    RTS ... return from subroutine

//////////////////  random sprite move  ///////////////
22A     FA07    read DelayTimer into register VA
22C     3A00    skip next instruction if VA == 0
22E     00EE    RTS  ... return from subroutine

230     221A    redraw the large sprite ... erases larghe sprite

232     C63D    set random X coordinate  (3D mask keeps numbers in range 0..61)
234     C71D    set random Y coordinate  (1D mask keeps numbers in range 0..29)
236     221A    draw new large sprite

238     6AFF    put new delay length into register VA
23A     FA15    set DelayTimer to register VA contents
23C     00EE    RTS ... return from subroutine

//////////////////  move small sprite with keypad ///////////////
23E     6C02    keypad = 2
240     ECA1    skip next instruction if keypad != 2
242     1258    jump up arrow
```

```
244     6C08    keypad = 8
246     ECA1    skip next instruction if keypad != 8
248     1266    jump down arrow

24A     6C04    keypad = 4
24C     ECA1    skip next instruction if keypad != 4
24E     1274    jump left arrow

250     6C06    keypad = 6
252     ECA1    skip next instruction if keypad != 6
254     1282    jump right arrow

256     00EE    RTS ... key not found

258     2224    JSR draw small sprite (ERASE) ... up arrow
25A     6C01    Keypad =1
25C     89C5    Y = Y-Keypad ... decrease Y
25E     6C1F    Keypad = 0x1F ... mask to restrict Y to range 0..31
260     89C2    apply the mask
262     2224    JSR draw small sprite (NEW SPRITE)
264     00EE    RTS

266     2224    JSR draw small sprite (ERASE) ... down arrow
268     6C01    Keypad =1
26A     89C4    Y= Y+Keypad ... increase Y
26C     6C1F    Keypad = 0x1F ... mask to restrict Y to range 0..31
26E     89C2    apply the mask
270     2224    JSR draw small sprite (NEW SPRITE)
272     00EE    RTS

274     2224    JSR draw small sprite (ERASE) ... left arrow
276     6C01    Keypad = 1
278     88C5    X = X-Keypad ... decrease X
27A     6C3F    Keypad = 0x3F ... mask to restrict Y to range 0..63
27C     88C2    apply the mask using AND logic
27E     2224    JSR draw small sprite (NEW SPRITE)
280     00EE    RTS

282     2224    JSR draw small sprite (ERASE) ... right arrow
284     6C01    Keypad = 1
286     88C4    X = X+Keypad ... increase X
288     6C3F    Keypad =0x3F ... mask to restrict Y to range 0..63
28A     88C2    apply the mask using AND logic

28C     2224    JSR draw small sprite (NEW SPRITE)
28E     00EE    RTS

////////////////// test code /////////
290     6500            reset BCD counter (V5)
292     2204            JSR ... display last 2 digits of 123

294     6820            put the X coordinate for the small sprite into register V8
296     6910            put the Y coordinate for the small sprite into register V9
298     2224            JSR .... draw the small sprite

29A     C63D            set random X coordinate  (3D mask keeps numbers in range 0..61)
29C     C71D            set random Y coordinate  (1D mask keeps numbers in range 0..29)
29E     A21A            point register I at the large sprite
2A0     D673            draw large sprite at coordinate V[6],V[7]

2A2     6AFF            put new delay length into register VA
2A4     FA15            set DelayTimer to register VA contents

2A6     222A            JSR ... draw random sprite

2A8     223E            JSR ... scan keypad

2B0     12A6            infinite loop
```

We are now able to move the small sprite up, down, left, and right

Points to note:

- a “mask” of 0x3F (63 decimal) at code lines 27A, 288 restricts the sprite X position to between 0..63
- a “mask” of 0x1F (31 decimal) at code lines 25E, 26C restricts the sprite Y position to between (0..32)
- Code lines 204 thru 28E never change and are the equivalent of an **Arduino “library”**
- Code lines 290 thru 2A0 are the equivalent of an **Arduino setup().**
- Code lines 2A2 thru 2B0 are the equivalent of an **Arduino loop()**

Coding is extremely easy once you have created your first “library”.

Step 17: A Simple Game (7) ... Collision Detector

Now that our sprites can move we need some way of:

- detecting a collision
- making a “beep” whenever a collision occurs
- adding 1 to the score counter

Collision detection is built into the DXYN instruction ... register V[F] is set to a 1 if a collision occurs otherwise it is set to a 0.

The computer a makes beep whenever the sound timer is greater than than zero. The larger the number ... the longer the beep.

The score counter needs no explanation ... just add 1 to the value in register V5

These changes are shown in the following code:

```
200     00E0    clear screen
202     12A2    jump to start of test code

/////////////// display BCD numbers subroutine ////////////////////
/*
   The user is responsible for pointing the index register I at clear memory
*/
204     6336    set screen X coordinate for 10’s digit ..... (36 hex = 54 decimal)
206     6401    set screen Y coordinate
208     A000    .......... point register I at scratch-pad memory
20A     F533    .......... convert V[5] content to BCD
20C     F265    ...........fills registers V0 to V2 inclusive with BCD from scratchpad
20E     F129    ........... automatically points register I to the 10’s sprite
210     D345    display the 10’s sprite
212     7305    move BCD X to the 1’s coordinate by adding 5
214     F229    ........... automatically points register I to the 1’s sprite
216     D345    display the 1’s sprite
218     00EE    RTS ... return from subroutine

//////////////////// large sprite pattern ///////////////////
21A     A0      *.*....         large sprite pattern
21B     40      .*......
21C     A0      *.*.....

////////////////////// small sprite pattern ////////////////////
21D     80      *.......        small sprite pattern

////////////////////// display large sprite ////////////////////
21E     A21A    point register I at the large sprite
220     D673    display 3 lines of the large sprite at coordinate V[6],V[7]
222     00EE            RTS ... return from subroutine

////////////////////// display the small sprite ///////////////
224     A21D    point register I at the small sprite
226     D891    display 1 line of the small sprite at coordinate V[8],V[9]
228     00EE    RTS ... return from subroutine

//////////////////  random sprite move  //////////////
22A     FA07    read DelayTimer into register VA
22C     3A00    skip next instruction if VA == 0
22E     00EE    RTS  ... return from subroutine

230     221A    redraw the large sprite ... erases larghe sprite

232     C63D    set random X coordinate  (3D mask keeps numbers in range 0..61)
234     C71D    set random Y coordinate  (1D mask keeps numbers in range 0..29)
236     221A    draw new large sprite
```

```
238     6AFF    put new delay length into register VA
23A     FA15    set DelayTimer to register VA contents
23C     00EE    RTS ... return from subroutine

//////////////////  move small sprite with keypad //////////////
23E     6C02    keypad = 2
240     ECA1    skip next instruction if keypad != 2
242     1258    jump up arrow

244     6C08    keypad = 8
246     ECA1    skip next instruction if keypad != 8
248     1266    jump down arrow

24A     6C04    keypad = 4
24C     ECA1    skip next instruction if keypad != 4
24E     1274    jump left arrow

250     6C06    keypad = 6
252     ECA1    skip next instruction if keypad != 6
254     1282    jump right arrow

256     00EE    RTS ... key not found

258     2224    JSR draw small sprite (ERASE) ... up arrow
25A     6C01    Keypad =1
25C     89C5    Y = Y-Keypad ... decrease Y
25E     6C1F    Keypad = 0x1F ... mask to restrict Y to range 0..31
260     89C2    apply the mask
262     2224    JSR draw small sprite (NEW SPRITE)
264     00EE    RTS

266     2224    JSR draw small sprite (ERASE) ... down arrow
268     6C01    Keypad =1
26A     89C4    Y= Y+Keypad ... increase Y
26C     6C1F    Keypad = 0x1F ... mask to restrict Y to range 0..31
26E     89C2    apply the mask
270     2224    JSR draw small sprite (NEW SPRITE)
272     00EE    RTS

274     2224    JSR draw small sprite (ERASE) ... left arrow
276     6C01    Keypad = 1
278     88C5    X = X-Keypad ... decrease X
27A     6C3F    Keypad = 0x3F ... mask to restrict Y to range 0..63
27C     88C2    apply the mask using AND logic
27E     2224    JSR draw small sprite (NEW SPRITE)
280     00EE    RTS

282     2224    JSR draw small sprite (ERASE) ... right arrow
284     6C01    Keypad = 1
286     88C4    X = X+Keypad ... increase X
288     6C3F    Keypad =0x3F ... mask to restrict Y to range 0..63
28A     88C2    apply the mask using AND logic

28C     2224    JSR draw small sprite (NEW SPRITE)
28E     00EE    RTS

////////////////// collision detector ////////////////////
290     3F01    skip next instruction if VF == 1
292     00EE    RTS ... there was no collision

294     6B06    put number in sound register (6*20mS=120mS)
296     FB18    send sound register contents to sound timer

298     2230    JSR ... erase/create large sprite

29A     2204    erase BCD
29C     7501    add 1 to V5
29E     2204    display new BCD count
```

```
2A0     00EE    RTS

////////////////// test code //////////
2A2     6500    reset BCD counter (V5)
2A4     2204    JSR ... display last 2 digits of 123

2A6     6820    put the X coordinate for the small sprite into register V8
2A8     6910    put the Y coordinate for the small sprite into register V9
2AA     2224    JSR .... draw the small sprite

2AC     C63D    set random X coordinate  (3D mask keeps numbers in range 0..61)
2AE     C71D    set random Y coordinate  (1D mask keeps numbers in range 0..29)
2B0     A21A    point register I at the large sprite
2B2     D673    draw large sprite at coordinate V[6],V[7]

2B4     6AFF    put new delay length into register VA
2B6     FA15    set DelayTimer to register VA contents

2B8     222A    JSR ... draw random sprite

2BA     223E    JSR ... scan keypad

2BC     2290    JSR ... collision detector

2BE     12B8    infinite loop
```

Guess what ... our game is complete :)

The file "Jumping Sprite.txt" contains the above code and is "ready to run".

Note

- Code lines 2B8, 2BA, 2BC, and 2BE are the equivalent of an **Arduino loop()**.

Step 18: Debugging

Sometimes your programs don't behave as expected in which case you need to "debug" your software **[1]**

The BCD subroutine in the above example misbehaved ... every so often the BCD score counter was getting corrupted.

- Photo 1 shows what the BCD counter should look like
- Photo 2 shows the corrupted BCD counter
- Photo 3 shows a screen shot after I single stepped through the subroutine
- Photo 4 shows the problem ... I had forgotten to point the index register at unused memory.

Explanation:

- By default all registers get set to zero when CHIP-8 starts ... the index register (I) was therefore pointing at unused memory.
- The first BCD conversion worked perfectly but when the score changed the second BCD conversion didn't !!!!
- The reason was that the index register **(I)** was left pointing at the last font that was displayed ... the second conversion was therefore overwriting the fonts.

To locate the "bug" I stopped the program just before the fault appeared by pressing the "L" key.

I then single-stepped through the code (photo 4) looking at the registers to see what changed (or didn't change) **[2]**

The disassembler really helps in situations like this.

Notes:

[1]

The terms "bug" and "debugging" are popularly attributed to Admiral Grace Hopper in the 1940s. While she was working on a Mark II computer at Harvard University, her associates discovered a moth stuck in a relay and thereby impeding operation, whereupon she remarked that they were "debugging" the system.

https://en.wikipedia.org/wiki/Debugging

[2]

The use of a spread-sheet is not normally required, but in this case it made it easier to see what was causing the problem.

Step 19: Summary

This instructable explains how to make a CHIP-8 virtual computer that can run existing public domain games using Processing 3

An external keypad allows two players at the same time. It also allows you to sit back from your computer screen when playing solo.

Arduino software that supports the external keyboard described in my instructable https://www.instructables.com/id/Touch-Keypad/ is included.

Unlike other CHIP-8 interpreters my version includes the following tools for debugging:

- breakpoints
- single-stepping
- disassembler
- all registers displayed

As such it is ideal for learning how to program.

To assist programming I have provided an instruction sheet with the CHIP-8 instructions grouped by function rather than alpha-numeric order.

Finally, a simple graphics game is developed to demonstrate how to:

- make counters
- configure your keypad
- use the delay timer
- use the sound timer
- create sprites (graphics)
- move sprites
- use “masks” to set boundary conditions
- use a “flag” to detect collisions
- create a subroutine library

The estimated cost of parts for the external keyboard version is less than $20

The single player version only requires some keyboard stickers.

Click here to view my other instructables.