III. CHIP-8 Language Programming

CHIP-8 is an easy-to-learn programming language that lets you write your own programs. To use the CHIP-8 language, you must first store the 512-byte CHIP-8 language program at memory locations 0000 to 01FF. The CHIP-8 language program is shown in Appendix C in hex form so you can enter it directly in memory using the hex keyboard. You can then record it on a memory cassette for future use. Each CHIP-8 instruction is a two-byte (4-hex-digit) code. There are 31, easy-to-use CHIP-8 instructions as shown in Table I.

When using CHIP-8 instructions your program must always begin at location 0200. There are 16 one-byte variables labeled 0-F. VX or VY refers to the value of one of these variables. A 63FF instruction sets variable 3 to the value FF (V3=FF). I is a memory pointer that can be used to specify any location in RAM. An A232 instruction would set I=0232. I would then address memory location 0232.

Branch Instructions

There are several types of jump or branch instructions in the CHIP-8 language. Instruction 1242 would cause an unconditional branch to the instruction at memory location 0242. Instruction BMMM lets you index the branch address by adding the value of variable 0 to it before branching. Eight conditional skip instructions let you test the values of the 16 one-byte variables or determine if a specific hex key is being pressed. This latter capability is useful in video game programs. (Only the least significant hex digit of VX is used to specify the key.)

A 2570 instruction would branch to a subroutine starting at location 0570. 00EE at the end of this subroutine will return program execution to the instruction following the 2570. The subroutine itself could use another 2MMM instruction to branch to (or call) another subroutine. This technique is known as subroutine nesting. Note that all subroutines called (or branched to) by 2MMM instructions must end with 00EE. **Ignoring this rule will cause hard-to-find program bugs.**

How to Change and Use the Variables

The CXKK instruction sets a random byte value into VX. This random byte would have any bits matching 0 bit positions in KK set to 0. For example, a C407 instruction would set V4 equal to a random byte value between 00 and 07.

A timer (or real-time clock) can be set to any value between 00 and FF by a FX15 instruction. This timer is automatically decremented by one, 60 times per second until it reaches 00. Setting it to FF would require about 4 seconds for it to reach 00. This timer can be examined with a FX07 instruction. A FX18 instruction causes a tone to be sounded for the time specified by the value of VX. A value of FF would result in a 4-second tone. The minimum time that the speaker will respond to is that corresponding to the variable value 02.

A FX33 instruction converts the value of VX to decimal form. Suppose I=0422 and V9=A7. A F933 instruction would cause the following bytes to be stored in memory:

```
0422  01
0423  06
0424  07
```

Since A7 in hex equals 167 in decimal, we see that the

Table I – CHIP-8 Instructions

Instruction	Operation
1MMM	Go to 0MMM
BMMM	Go to 0MMM + VO
2MMM	Do subroutine at 0MMM (must end with 00EE)
00EE	Return from subroutine
3XKK	Skip next instruction if VX = KK
4XKK	Skip next instruction if VX ≠ KK
5XY0	Skip next instruction if VX = VY
9XY0	Skip next instruction if VX ≠ VY
EX9E	Skip next instruction if VX = Hex key (LSD)
EXA1	Skip next instruction if VX ≠ Hex key (LSD)
6XKK	Let VX = KK
CXKK	Let VX = Random Byte (KK = Mask)
7XKK	Let VX = VX + KK
8XY0	Let VX = VY
8XY1	Let VX = VX/VY (VF changed)
8XY2	Let VX = VX & VY (VF changed)
8XY4	Let VX = VX + VY (VF = 00 if VX + VY ≤ FF, VF = 01 if VX + VY > FF)
8XY5	Let VX = VX – VY (VF = 00 if VX < VY, VF = 01 if VX ≥ VY)
FX07	Let VX = current timer value
FX0A	Let VX = hex key digit (waits for any key pressed)
FX15	Set timer = VX (01 = 1/60 second)
FX18	Set tone duration = VX (01 = 1/60 second)
AMMM	Let I = 0MMM
FX1E	Let I = I + VX
FX29	Let I = 5-byte display pattern for LSD of VX
FX33	Let MI = 3-decimal digit equivalent of VX (I unchanged)
FX55	Let MI = VO : VX (I = I + X + 1)
FX65	Let VO : VX = MI (I = I + X + 1)
00E0	Erase display (all 0's)
DXYN	Show n-byte MI pattern at VX-VY coordinates. I unchanged. MI pattern is combined with existing display via EXCLUSIVE-OR function. VF = 01 if a 1 in MI pattern matches 1 in existing display.
0MMM	Do machine language subroutine at 0MMM (subroutine must end with D4 byte)

three RAM bytes addressed by I contain the decimal equivalent of the value of V9.

If I=0327, a F355 instruction will cause the values of V0, V1, V2, and V3 to be stored at memory locations 0327, 0328, 0329, and 032A. If I=0410, a F265 instruction would set V0, V1, and V2 to the values of the bytes stored at RAM locations 0410, 0411, and 0412. FX55 and FX65 let you store the values of variables in RAM and set the values of variables to RAM bytes. A sequence of variables (V0 to VX) is always transferred to or from RAM. If X=0, only V0 is transferred.

The 8XY1, 8XY2, and 8XY4, and 8XY5 instructions perform logic and binary arithmetic operations on two 1-byte variables. VF is used for overflow in the arithmetic operations.

Using the Display Instructions

An 00E0 instruction erases the screen to all 0's. When the CHIP-8 language is used, 256 bytes of RAM are displayed on the screen as an array of spots 64 wide by 32 high. A white spot represents a 1 bit in RAM, while a dark (or off) spot represents a 0 bit in RAM. Each spot position on the screen can be located by a pair of coordinates as shown in Fig. 1.

The VX byte value specifies the number of horizontal spot positions from the upper left corner of the display. The VY byte value specifies the number of vertical spot positions from the upper left corner of the display.

The DXYN instruction is used to show a pattern of spots on the screen. Suppose we wanted to form the

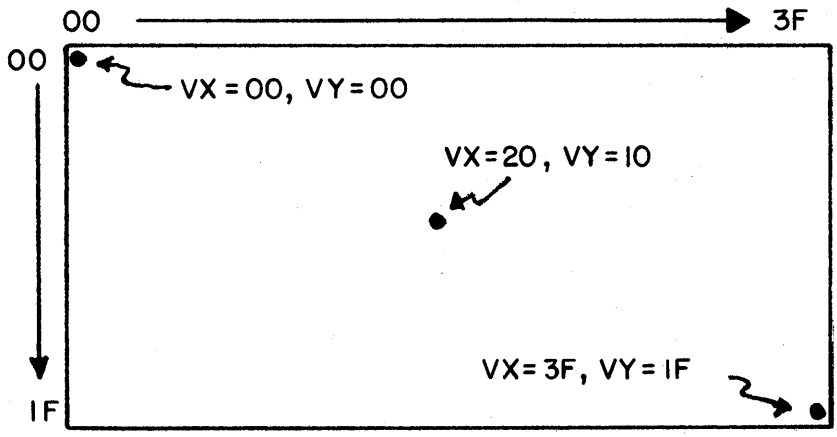

Fig. 1 — Display screen coordinate structure.

pattern for the digit "8" on the screen. First we make up a pattern of bits to form "8" as shown in Fig. 2.

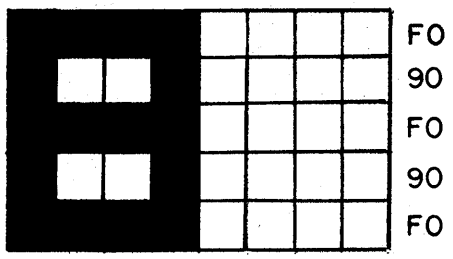

Fig. 2 — Pattern of bits forming digit 8.

In this example we made the "8" pattern five spots high by four spots wide. Patterns to be shown on the screen using the DXYN instruction must always be one byte wide and no more than fifteen bytes high. (Several small patterns can be combined to form larger ones on the screen when required). To the right of the "8" pattern in Fig. 2 are the equivalent byte values in hex form. We could now store this pattern as a list of five bytes at RAM location 020A as follows:

```
020A  F0
020B  90
020C  F0
020D  90
020E  F0
```

Suppose we now want to show this pattern in the upper left corner of the screen. We'll assign V1=VX and V2=VY. Now we let V1=V2=00 and set I=020A. If we now do a D125 instruction, the "8" pattern will be shown on the screen in the upper left corner.

You can write a program to show the "8" pattern on the screen as follows:

```
0200 A20A I=020A
0202 6100 V1=00
0204 6200 V2=00
0206 D125 SHOW 5MI@V1V2
0208 1208 GO 0208
020A F090
020C F090
020E F000
```

The first column of this program shows the memory locations at which the instruction bytes in the second column are stored. The third column indicates the function performed by each instruction in shorthand form. Only the bytes in the second column are actually stored in memory.

With the CHIP-8 interpreter stored at 0000-01FF, you can load the above program in memory and run it. Set V1 and V2 to different values to relocate the "8" pattern on the screen. The VX-VY coordinates always specify the screen position of the upper left-hand bit of your pattern. This bit can be either 0 or 1. The last digit of the DXYN instruction specifies the height of your patterns or the number of bytes in your pattern list.

When a pattern is displayed, it is compared with any pattern already on the screen. If a 1 bit in your pattern matches a 1 bit already on the screen, then a 0 bit will be shown at this spot position and VF will be set to a value of 01. You can test VF following a DXYN instruction to determine if your pattern

touched any part of a previously displayed pattern. This feature permits programming video games which require knowing if one moving pattern touches or hits another pattern.

Because trying to display two 1 spots at the same position on the screen results in a 0 spot, you can use the DXYN instruction to erase a previously displayed pattern by displaying it a second time in the same position. (The entire screen can be erased with a single 00E0 instruction.) The following program shows the "8" pattern, shows it again to erase it, and then changes VX and VY coordinates to create a moving pattern:

```
0200 A210 I=0210
0202 6100 V1=00
0204 6200 V2=00
0206 D125 SHOW 5MI@V1V2
0208 D125 SHOW 5MI@V1V2
020A 7101 V1+01
020C 7201 V2+01
020E 1206 GO 0206
0210 F090
0212 F090
0214 F000
```

The "8" pattern byte list was moved to 0210 to make room for the other instructions. Try changing the values that V1 and V2 are incremented by for different movement speeds and angles. A delay could be inserted between the two DXYN instructions for slower motion.

The FX29 instruction sets I to the RAM address of a five-byte pattern representing the least significant hex digit of VX. If VX=07, then I would be set to the address of a "7" pattern which could then be shown on the screen with a DXYN instruction. N should always be 5 for these built-in hex-digit patterns. Appendix C shows the format for these standard hex patterns. The following program illustrates the use of the FX29 and FX33 instructions:

```
0200 6300 V3=00
0202 A300 I=0300
0204 F333 MI=V3(3DD)
0206 F265 V0:V2=MI
0208 6400 V4=00
020A 6500 V5=00
020C F029 I=V0(LSDP)
020E D455 SHOW 5MI@V4V5
0210 7405 V4+05
0212 F129 I=V1(LSDP)
0214 D455 SHOW 5MI@V4V5
0216 7405 V4+05
0218 F229 I=V2(LSDP)
021A D455 SHOW 5MI@V4V5
021C 6603 V6=03
021E F618 TONE=V6
0220 6620 V6=20
0222 F615 TIME=V6
0224 F607 V6=TIME
0226 3600 SKIP;V6 EQ 00
0228 1224 GO 0224
022A 7301 V3+01
022C 00E0 ERASE
022E 1202 GO 0202
```

This program continuously increments V3, converts it to decimal form, and displays it on the screen.

The FX0A instruction waits for a hex key to be pressed, VX is then set to the value of the pressed key, and program execution continues when the key is released. (If key 3 is pressed, VX=03). A tone is heard while the key is pressed. This instruction is used to wait for keyboard input.

Applying CHIP-8

You should now be able to write some simple CHIP-8 programs of your own. Here are some things to try:

1. Wait for a key to be pressed and show it on the display in decimal form.

2. Show an 8-bit by 8-bit square on the screen and make it move left or right when keys 4 or 6 are held down.

3. Show an 8-bit square on the screen. Make it move randomly around the screen.

4. Show a single bit and make it move randomly around the screen leaving a trail.

5. Program a simple number game. Show 100 (decimal) on the screen. Take turns with another player. On each turn you can subtract 1-9 from the number by pressing key 19. The first player to reach 000 wins. The game is more interesting if you are only allowed to press a key which is horizontally or vertically adjacent to the last key pressed.

If you are unsure of the operation of any CHIP-8 instruction, just write a short program using it. This step should clear up any questions regarding its operation. In your CHIP-8 programs be careful not to write into memory locations 0000-01FF or you will

lose the CHIP-8 interpreter and will have to reload it. You can insert stopping points in your program for debugging purposes. Suppose you want to stop and examine variables when your program reaches the instruction at 0260. Just write a 1260 instruction at location 0260. Flip RUN down and use operating system mode A to examine variables V0-VF. The memory map in Appendix C shows where you can find them.

After the above practice you are ready to design more sophisticated CHIP-8 programs. **Always prepare a flowchart before actually writing a program.** The last 352 bytes of on-card RAM are used for variables and display refresh. In a 2048-byte RAM system you can use locations 0200-069F for your programs. This area is enough for 592 CHIP-8 instructions (1184 bytes). In a 4096-byte RAM system you can use locations 0200-0E8F. This area is equal to 1608-CHIP-8 instructions (3216 bytes).

Some Program Ideas

Here are a few ideas for programs to write using the CHIP-8 language:

1. INTOXICATION TESTER - Display a six-digit random number on the screen for several seconds. You must remember this number and enter it from the keyboard within ten seconds after the screen goes blank to prove that you're sober and score.

2. NUMBER BASE QUIZ - Display numbers in binary or octal on the screen. You must enter their decimal equivalent to score points.

3. DICE - Push any key to simulate rolling dice displayed on the screen.

4. PUPPETS - Show large face on the screen. Let small children move mouth and roll eyes by pushing keys.

5. BUSY BOX - Let small children push keys to make different object appear on the screen, move, and make sounds.

6. SHUFFLEBOARD - Simulate shuffleboard-type games on the screen.

7. COMPUTER ART - Design new programs to generate pleasing geometric moving patterns on the screen.

8. INVISIBLE MAZE - Try to move a spot through an invisible maze. Tones indicate when you bump into a wall.

9. LUNAR LANDING - Program a graphic lunar landing game.

10. COLLIDE - Try to maneuver a spot from one edge of the screen to the other without hitting randomly moving obstacles.

11. CAPTURE - Try to chase and catch randomly moving spots within a specified time limit.

12. LEARNING EXPERIENCES - Program graphic hand and eye coordination exercises for young children or those with learning disabilities.

13. NUMBER RECOGNITION - Show groups of objects or spots on the screen. Young child must press key representing number of objects shown to score.

14. WALL BALL - Program a wall-ball-type paddle game for one player.

15. FOOTBALL - Each player enters his play via the hex keyboard and the computer moves the ball on the screen.

16. BLACKJACK - Play "21" against the computer dealer.

17. HOLIDAY DISPLAYS - Design custom, animated displays for birthdays, Halloween, Christmas, etc.

18. METRIC CONVERSION - Help children learn metric by showing lengths on screen in inches and requiring centimeter equivalent to be entered to score.

19. TURING MACHINE - Simulate a simplifed Turing machine on the screen.

20. TIMER - Use the computer to time chess games, etc.

21. HEXAPAWN - Program Hexapawn so that the computer learns to play a perfect game.

22. NIM - Program Nim with groups of spots shown on the screen.

23. BLOCK PUZZLES - You can simulate a variety of sliding block-type puzzles on the screen.

24. BOMBS AWAY - Show a moving ship at the bottom of the screen. Try to hit the ship by releasing bombs from a moving plane at the top of the screen.

25. PROGRAMMED SPOT - Introduce children to programming concepts by letting them preprogram the movements of a spot or object on the screen.

The next section will discuss machine language programming. You can even combine machine language subroutines with CHIP-8 programs if desired.