4504 SYSTEM PROGRAMMERS MANUAL

22nd October, 2015

CONTENTS:

CHAPTER I: 4504 CENTRAL PROCESSING UNIT

SECTION I: CPU OVERVIEW

The 4504 forms an extended version of the SuperCHIP-8 system, along with being fully backwards compatible with CHIP-8 and SuperCHIP, it also extends to capabilities of the original CHIP-8 design, whilst maintaining the essence of being a small and a rather easy CPU to emulate, compared to the relatively humongous architectures of today.

The 4504 is, for the most part, an 8-bit CPU, all operations done are on 8-bit registers. However, since 8-bit addressing only has the ability to access up to 256 bytes maximum, using 8-bit registers for memory access would rather be limited. Hence, the CPU has a 12-bit Data Index (I) register, which is used for accessing memory. This gives a total of 0x1000 or 4096 bytes of memory to be accessed by a 4504 program at once. Do note that the total addressing capability of the 4504 is a total of 64 kibibytes, or 0xFFFF bytes of memory, this is done using a system of segmentation, which is explained later in this document. The Instruction Pointer register in 4504, just like the Data Index register is 12-bit, giving 4504 programs access to a total 4 Kibibytes (4096 bytes) of code and data at once.

As for the hardware, in addition to the CPU, a storage device, a display device, and a hex keyboard are usually present.

Nesting calls are also supported in 4504, just like the original CHIP-8. However, the 4504 has 256 (0xFF) levels of nesting, compared to CHIP-8's mere 16.

SECTION II: CPU SPECIFICATIONS

As described before the 4504 is an 8-bit CPU based upon and backwards compatible with the CHIP-8 design. The 4504 has the following specifications:

- 16 8-bit General Purpose registers: V0, V1, V2, V3, V4, V5, V6, V7, V8, V9, VA, VB, VC, VD, VE, VF.
  - VF is often used as a carry flag by some instructions
- 1 12-bit Instruction Pointer: IP
  - Used along with CS to find the next operation code (opcode) to execute.
- 1 12-bit Code Index: CI
  - Used for computing offsets for far jumps and setting interrupt handlers.
- 1 12-bit Data Index: I
  - Used for memory access (loading or storing)
- 1 12-bit Data Stack Pointer: SP
  - Index for the last item pushed to stack.
- 1 8-bit Code Segment: CS
  - Code Segment to fetch the next opcode from. The value of the address of the next opcode is computed linearly using the formula:
    - `Code_Linear = ((CS << 12) | IP)`
- 1 8-bit Data Segment: DS
  - Data segment, generally used to compute the linear address from the Data Index for data access, using a similar formula as above:
    - `Data_Linear = ((DS << 12) | I)`
- 1 8-bit Stack Segment: SS

    - Stack Segment, used along with SP to compute the linear address of stack in memory. Using a similar formula given above:
        - `Stack_Linear = ((SS << 12) | SP)`

- 512 bytes (256 levels) of nesting calls
    - Used for storing return addresses along with the return segment when a CALL is done. Return is accomplished by the RET instruction.
- 16 interrupt handlers
    - Interrupts are generally used to handle exceptions and communicate with several pieces of hardware.
- 1 8-bit Delay Timer Register: DT
    - DT is decremented by 1 every 1/60th of a second until DT = 0.
- 1 8-bit Sound Timer Register: ST
    - ST is decremented by 1 every 1/60th of a second until ST = 0. For every decrement, a standard beep (A440Hz) is played.
- 16 key input through a hexadecimal keyboard
    - Used to receive input, key codes range from 0-F. Can be mapped to other keys on modern keyboard to suit their layout.
- 65536 bytes of memory space
    - Used to store program code and data.
- 256 8-bit Ports
    - Used by hardware devices to send, and receive information.

The CPU runs at a constant frequency of 3 Megahertz, and decrements both the delay and the sound timers every 1/60th of a second, that is, at the rate of 60hz.

SECTION III: 4504 Instruction Set

Most 4504 operation codes are exactly 2 bytes in length, containing both the operation code and the immediates or operands. This way of instruction encoding is very compact, although poses some limitations.

In the upcoming table, the following must be taken note of:

- *NNN, NN, N:* These are constants, or immediates. NNN is 12-bit, NN is 8-bit, N is 4-bit. They represent hexadecimal nibbles.
- *X, Y:* Represent the index of registers, X is often the first register argument index and Y is the latter register argument index. They are also hexadecimal nibbles. X and Y can range from 0-F.
- *0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F:* Represent hexadecimal nibbles of an operation code. Also, '0' is put in places for nibbles that are ignored in operation codes.
- All operation codes are prefixed with '0x' to denote hexadecimal notation.
- Instructions which have a '*' suffix are SuperCHIP instructions, while those having a '**' are exclusively 4504 instructions. Note that the asterisks **do not** form a part of the mnemonic and are just there for reference.

Mnemonic	Operation Code	Description
NOP**	0x0000	No operation code. Skip.
OUT** VX, VY	0x0X9Y	Output register VX to port, whose index is in VY.
IN** VX, VY	0x0XAY	Input into register VX from port, whose index is in VY.
HLNDR** N	0x00BN	Set interrupt handler N's CS:IP to DS:CI. There are only 16 interrupt handlers, the higher nibble of VX is ignored.
SCD N	0x00CN	Scroll screen down N pixels.
INT** N	0x00DN	Throw interrupt with index 'N'. Pushes CS:IP to stack and jumps to handler's CS:IP.
CLS	0x00E0	Clears the screen buffer.
MOV** DS, VX	0x0XE1	Set DS to value of register VX. The higher nibble of VX is ignored.
MOV** VX, DS	0x0XE2	Set register VX to value of DS.
MOV** VX, CS	0x0XE3	Set register VX to value of CS.
MOV** DS, CS	0x00E4	Set value of DS to CS.
MOV** SS, VX	0x0XE5	Set value of SS to

		register VX. Higher nibble of VX is ignored.
MOV** SP, I	0x00E6	Set value of SP to I.
PUSH** VX	0x0XE7	Push value of register VX to SS:SP (Linear: (SS << 12) | SP), and decrement SP by 1.
POP** VX	0x0XE8	Pop value of register VX from SS:SP (Linear: (SS << 12) | SP), and increment SP by 1.
MOV** CI, I	0x00E9	Set value of CI to I.
MOV** I, CI	0x00EA	Set value of I to CI.
PUSH** I	0x00EB	Push Index register to SS:SP, and decrement by 2.
POP** I	0x00EC	Pop Index register from SS:SP, and increment by 2.
MOV** I, SP	0x00ED	Set value of SP to I.
RET	0x00EE	Pop CS:IP from return stack (not data stack which is SS:SP), and jump to the return address.
SCR*	0x00FB	Scroll display 4 pixels right.
SCL*	0x00FC	Scroll display 4 pixels left.
EXIT*	0x00FD	Exit execution.
LOW*	0x00FE	Switch to low

		resolution mode. (64x32)
HIGH*	0x00FF	Switch to high resolution mode. (128x64)
JMP NNN	0x1NNN	Set IP to 0xNNN.
CALL NNN	0x2NNN	Push CS:IP to return address stack, and set IP to NNN.
SKIPE VX, NN	0x3XNN	Skip next instruction if VX equals NN.
SKIPNE VX, NN	0x4XNN	Skip next instruction if VX is not equal to NN.
SKIPE VX, VY	0x5XY0	Skip next instruction if register VX equals VY.
MOV VX, NN	0x6XNN	Set register VX to NN.
ADD VX, NN	0x7XNN	Add register VX to NN. Carry flag VF is **not** modified.
MOV VX, VY	0x8XY0	Set register VX to register VY.
OR VX, VY	0x8XY1	Do a bitwise OR on VX by VY bits, and store result in VX.
AND VX, VY	0x8XY2	Do a bitwise AND on VX by VY bits, and store result in VX.
XOR VX, VY	0x8XY3	Do a bitwise XOR on VX by VY bits, and store result in VX.
ADDC VX, VY	0x8XY4	Add VX to VY, store

		result in VX, and set carry flag VF if overflow.
SUB VX, VY	0x8XY5	Subtract VX from VY, store result in VX, and set carry flag.
SHR1 VX	0x8X06	Shift register right VX by 1, store result in VX.
…	0x8XY7	
SHL** VX, VY	0x8XY8	Shift register VX left by VY bits, and store result in VX.
SHR** VX, VY	0x8XY9	Shift register VX right by VY bits, and restore result in VX.
ROL** VX, VY	0x8XYA	Rotate register VX left by VY bits and store result in VX.
ROR** VX, VY	0x8XYB	Rotate register VX right by VY bits and store result in VX.
XCHG** VX, VY	0x8XYC	Set VX to VY, and set VY to VX.
NOT** VX	0x8X0D	Do a bitwise not (!) on register VX, store result in VX.
NEG** VX	0x8X1D	Do a bitwise neg (~) on register VX, store result in VX.
SHL1 VX	0x8X0E	Shift register VX by 2, store result in VX.
SKIPNE VX, VY	0x9XY0	Skip next instruction if register VX is not

		equal to register VY.
MOV I, NNN	0xANNN	Set I to NNN.
JMP0 NNN	0xBNNN	Set IP to NNN + V0.
RAND VX, NN	0xCXNN	Set VX to a random number ranging from 00 to NN.
DRW VX, VY, N	0xDXYN	Draw a sprite which is located at DS:I, on position VX, VY with height N.
SKIPDN VX	0xEX9E	Skip next instruction if key, whose key code is in VX is pressed i.e. down.
SKIPUP VX	0xEXA1	Skip next instruction if key, whose key code is in VX is not pressed, i.e. up.
MOV** CI, VX, VY	0xFXAY	Set CI to (VX << 8) | VY.
JMPF** VX, VY	0xFXBY	Set IP to VX:CI+VY.
CALLF** VX, VY	0xFXCY	Push CS:IP to return address stack, set IP to VX:CI+VY.
MOV** CI, NNN	0xF0D0 0x0NNN	Set CI to NNN. This is a 4-byte opcode. The only one of it's kind.
MOV VX, DT	0xFX07	Set register VX to DT (Delay Timer)
WAITKEY VX	0xFX0A	Keep idling until key, whose keycode is in VX is pressed.
MOV DT, VX	0xFX15	Set Delay Timer to register VX.

MOV ST, VX	0xFX18	Set Sound Timer to register VX.
ADD I, VX	0xFX1E	Set I to I + VX.
ADD CI, VX	0xFX1F	Set CI to CI + VX
GETSPR VX	0xFX29	Get offset of sprite of digit in lower nibble of VX. The segment is guaranteed to be 0.
GETSPREX VX	0xFX30	Get offset of sprite to be used in high resolution mode of digit in lower nibble of VX. The segment is guaranteed to be 0.
STOREBCD	0xFX33	Store binary coded decimal representation of VX into DS:I with the most signification digit at I, second most at I+1, and least at I+2.
STORE N	0xFN55	Store registers V0 to VN in DS:I, starting from V0 at I.
LOAD N	0xFN65	Load registers V0 to VN from DS:I, starting from V0 at I.
HP48S N	0xFN75	Store registers V0 to VN to HP48 flags. (N <= 7)
HP48L N	0xFN85	Load registers V0 to VN in HP48 flags. (N <= 7)

SECTION III: MEMORY ACCESS (SEGMENTATION)

Memory access in 4504 is done through a rudimentary process of segmentation, which give 16-bit **linear** addresses. The least significant **12-bits** are the offset, and the **most signification 4-bits are the segment**. In total, thus, the 4504 can access upto 64 KiB of memory. Segments are in the range of 0x0-0xF, and offsets are in the range of 0x000-0xFFF.

The segmentation "algorithim" is thus,

```
Address_Linear = (SEG << 12) | OFF
```

The 4504 provides 3 segment registers for this purpose: CS, DS, and SS. Namely, the **code segment**, **data segment**, and **stack segment**, respectively.

The code segment is used to refer to the segment where the current operation code is executing. The offset to this segment is IP, thus the current opcode is retrieved through: `*(U16*)MEMORY[(CS << 12) | IP]`.

The CS and IP registers cannot be modified directly, however, they can be modified through JMP, CALL, JMP0, JMPF, and CALLF instructions.

JMP: Linear jump, jumps to an immediate 12-bit offset.
e.g: `JMP 0x123`
CALL: Linear call, pushes current CS and IP to return address stack, and jumps to address. The callee can then return back to the caller using the "RET" instruction which pops CS and IP, and resumes execution to the caller.
e.g: `CALL 0x123`

JMP0: Same as "JMP" - except the offset is added to register V0.
e.g:

```
MOV V0, 10
JMP0 0x550 ; Jumps to 0x55A
```

JMPF: Intersegmental jump. Jumps to VX:CI+VY.
e.g:

```
MOV CI, 0x500
MOV V0, 0xF
MOV V1, 0xA
JMPF V0, V1 ; Jumps to 0xF:0x500+0xA, CS:IP is 0xF:0x50A.
Linear 0xF50A
```

CALLF: Intersegmental CALL. Same as CALL but jumps to VX:CI+VY instead.
e.g:

```
MOV CI, 0x500
MOV V0, 0xE
MOV V1, 0
CALLF V0, V1 ; Calls 0xE:0x500 + 0x0 or 0xE:0x500. Linear
E500. RET can be used to return back to this segment.
```

The data segment represents the segment of the index register offset. It is used to load and store data, using the instructions: LOAD, STORE, STOREBCD, GETSPR, GETSPREX, and HLNDR.

The stack segment is used for access to the stack, the stack pointer serves as an offset to the stack segment. The following code will set up a proper stack on the 4504:

```
MOV I, 0x000
MOV SP, I
MOV V0, 0x0F
MOV SS, V0
; SS:SP = 0x0F:0x000, Linear: 0xF000
PUSH V1
; SS:SP = 0xF:0xFFF, Linear: 0xFFFF
PUSH V2
```

```
; SS:SP = 0xF:0xFFE, Linear: 0xFFFE
```

The “PUSH” and “POP” instructions can be used to store and load values to the Stack Pointer.

The Stack Pointer is truncated if the stack exceeds it’s limit, hence there is unfortunately, no proper way to detect a stack overflow.

Do note that the offsets returned from the GETSPR and GETSPREX cannot be used until the data segment is set to 0, as these only return the offsets of the font characters located at segment 0.

On initialisation, the 4504 starts in compatiblity mode with segment set to 0, and offset set 0x200, where the program is loaded to being execution. Hence, CS:IP is set 0:0x200, SS:SP is set to 0x0:0x0, and DS:I is set to 0x0:0x0, while CI is set to 0x0.

SECTION IV: INTERRUPTS

The 4504 has 16 interrupt handlers ranging from 0x0-0x0F, these can be fired by the CPU or by the program depending upon the situation.

Interrupts are generally used to call intersegmental routines that are supposed to be called quite a few times, as calling the routines the “normal” way would not be very size efficient.

Setting an interrupt

The interrupt handlers can be set using the HLNDR instruction. The HLNDR instruction expects one argument which ranges from 0x0-0x0F, the code segment of the interrupt handler must be passed in DS, while the address of the interrupt handler must be in CI.

The follwing snippet sets the interrupt handler 0xF to the desired handler:

```
; Assume int0xF_handler is in the same code segment.
MOV DS, CS
MOV CI, int0xF_handler
HLNDR 0xF
..
int0xF_handler:
    ...
```

Returning from an Interrupt

Interrupts return in the same way as callees, using the ‘RET’ instruction.

Calling an interrupt

You can call interrupts using the ‘INT’ instruction, for instance, as in above, to call the int0xF_handler, use:

```
INT 0xF
```

Reserved Interrupts

The following is a list of interrupts that are reserved by the system, the interrupts are 0x0, 0x01, and 0x2. They are *invalid opcode interrupt*, fired when the CPU encounters and invalid opcode, *invalid operation interrupt*, fired when the CPU encounters an attempt to do an invalid operation, and 0x02, which is the *breakpoint interrupt*.

CHAPTER II: THE 4504 DISPLAY UNIT

SECTION I: DISPLAY UNIT OVERVIEW

The 4504's display unit is a monochrome 128x64 display, and works exactly the same way as the CHIP-8's display unit. It supports two resolutions, 64x32 (which is set using the LOW instruction) and 128x64 (which is set using the HIGH instruction). The one and only instruction used for drawing sprites on screen is the '0xDXYN' instruction, which will be explained further.

The display unit works by using bitmapped sprites, every turned on bit represents a white (or *foreground* colour) pixel, while every turned off bit represents a black (or *background* color) pixel. For instance, in order to display character 'A', usually in low resolution modes, the sprite array to be used looks like this:

```
0xF0, 0x90, 0xF0, 0x90, 0x90
```

Now, representing this in binary form we get,

```
11110000, 10010000, 11110000, 10010000, 10010000
```

Arranging these one, above the another,

```
11110000
10010000
11110000
10010000
10010000
```

Replacing '1's with '*', and '0's with spaces, we see:

```
****
*   *
****
*   *
*   *
```

Hence, the shape formed looks like a rudimentary letter ‘A’ pixel map.

If you are running in low resolution mode, sprites can have a maximum length of 8 pixels, and a maximum height of 15 pixels, high resolution mode however, supports 16x16 pixel sprites.

[This manual is under construction. Last signed: 18/12/2015]