Programming in CHIP-8 — a crash course

(what to do with your ETI-660 while you're learning how to do things with your ETI-660)

The 'language' used by our ETI-660 Learners' Microcomputer is called CHIP-8. It was developed by the designers of the 1802 microprocessor as an aid to writing small programs. Phil Cohen takes a look at CHIP-8 and gives a few programs for you to explore.

Phil Cohen

FIRSTLY, I should really tell you what CHIP-8 is *not*. It's not a high-level language like BASIC or even FORTRAN. Programs written in CHIP-8 are almost indecipherable to the casual reader. In that respect it is very much like 'machine code' — which is the 'language' that the microprocessor itself uses.

CHIP-8 is one step removed from that primitive machine code. What happens is this: you write a program in CHIP-8, and put it into an area of the ETI-660's RAM memory. Then you tell the microprocessor (by means of the built-in 'monitor' program located in IC11 that I'll also describe in this article) to go to the area of ROM memory that CHIP-8 resides in. The information in the CHIP-8 'interpreter' will tell the microprocessor how to do the CHIP-8 *program* that you've entered.

So why not write your programs in machine code in the first place and be done with it? Well, CHIP-8 offers some facilities (especially in input and output) that are difficult to design into a machine code program (especially for a first-timer).

If the last few paragraphs have 'lost' you, then don't worry. I'm just about to repeat myself in simpler terms.

When you turn on your ETI-660, the computer will be in the control of the 'monitor'. By operating the buttons on the unit, you can make the machine store a sequence of bytes of information in its memory.

Having done that, you then tell the monitor to hand over control to the CHIP-8 interpreter. The CHIP-8 interpreter follows the 'program' that you typed in using the monitor. The program can be anything from a TV game to a reaction timer.

How do you know what bytes of information to store in order to make the machine perform a particular series of actions? — that's easy. You read the rest of this article!

The monitor

Before I go on to the CHIP-8 interpreter proper, I'll just quickly cover the various commands that the monitor program is capable of understanding.

On the ETI-660, there is a RESET button. Now, no matter what happens, pressing the RESET button will get you back to the state where you were when the machine was first turned on. If after the monitor has passed control to the CHIP-8 interpreter, things start to go wrong (the thing gets into an endless loop, for example), then all you have to do is to press the RESET button and you'll be back in the monitor again.

Now, the first command that we'll cover is the **'Memory Inspect and Modify'** (MI&M for short). This command is started by pressing key **0**. The machine must just have been RESET.

When you press RESET, four digits will appear on the screen. This four-digit number is an 'address' — the number corresponds to a 'location' of RAM. Each 'pigeon hole' in the ETI-660 RAM memory has a four-digit number allocated to it.

In order to change the number on the screen, press key 0, followed by four digits.

Once you have done that, you can find out what the contents of the memory location are by pressing one of the STEP buttons (either one — they're connected in parallel). The two-digit number that appears is the value held in the location whose address you entered.

Pressing the STEP button again will show you the next location. Pressing it repeatedly will 'step' you through memory.

You can change the contents of any location by putting its address on the screen, and then entering the value that you want using the keyboard.

You can enter a series of values into successive memory locations by pressing STEP between entries. So to enter 12 into location 0600, 34 into 0601, and 56 into 0602, you would press RESET, then 0, then 0600, then STEP, then 12, STEP, 34, STEP, 56.

You can get out of MI&M mode by pressing RESET — this will not affect the values that you've put into memory.

The numbering system used is base 16, or hexadecimal ('hex' for short), rather than the more usual base 10 (which we were all taught at school). If you don't understand base 16, wait until the 'simple' series of articles on programming which will follow this 'crash course'.

So, after you've worked out what the various bytes of your CHIP-8 program are to be, you can enter them into memory using MI&M.

Once you have entered them, you can **pass control to the CHIP-8 interpreter** by pressing key **8** (after the machine has been RESET).

The CHIP-8 interpreter will look for the first CHIP-8 instruction at memory location 0600. The locations available to you to use for a CHIP-8 program are from 0600 to 07FF — that's over 500 locations.

Storing on cassette

Now, if you turn the machine off, you will lose the program that you just typed in. This is not good, especially if it happens by accident.

You can **store your program onto cassette** by the use of key number **2**. What you do is this: put the location at which the program starts into locations 0400 and 0401.

All your CHIP-8 programs will start at location 0600, so that means setting location 0400 to 06, and location 0401 to 00.

Similarly, set locations 0402 and 0403 to the end location of your program. 07FF should cover it. Press RESET, then start your cassette running and press button 2. As the ▶

STORING ON/LOADING FROM CASSETTE

To Store: Having typed in your program and got it running, you'll want to store (or save) it on cassette. Say the program runs from 0600 (in RAM) to 0725 — like the 'Target Practice' example later. First, press RESET, then press 0. Now,
press

'0400'
'STEP'
'06'
'STEP'
'00'
'STEP'

The address section in the bar at the bottom of the screen should now show '0402'. Continue with:

'07'
'STEP'
'25'

As a precaution, to avoid 'chopping off' the end of your program, you could make the last entry 'FF' if you like.

Then, press RESET. Connect up your cassette recorder — TAPE OUT to the 'mic' jack and TAPE IN to the 'ear' jack, and insert a fresh cassette. Set your cassette deck recording and:

press '2'

The speaker in the 660 will emit a series of outrageous noises. The TV screen will go blank, too. When the noise stops, STOP your cassette recorder. The TV screen will come back to life. Rewind your tape. Turn the power to the ETI-660 off briefly, then on again. Now you can try the next bit.

To load: Rewind your tape, if it isn't already rewound, or set it to the vicinity of the tape prior to where you recorded the program. You have to set the memory location 0400 to the start of the program (usually 0600) and 0402 to the end of the program — or beyond that (if you want to be safe). Taking the 'Target Practice' program:
press

'RESET'
'0'
'STEP'
'0400'
'06'
'STEP'
'00'
'STEP'
'07'
'25'
(or you could put 'FF' here)
'RESET'
'4'

The TV screen will go blank. Now start the cassette deck in PLAY. Set the volume control well up. Some experimentation with the volume control may be necessary to achieve a successful load. As the program is loaded, the speaker in the 660 will make funny noises. The TV display will come back on and the sound will stop when the load is finished. STOP the cassette deck. Press RESET and then 8 and your program is up and running!

Troubles: Your cassette deck should be in good condition, with clean unmagnetised heads properly aligned. If it cannot adequately record or reproduce 10 kHz then have it serviced or get another cassette deck. We recommend you also read 'Reliable Cassette Recorder For Your Computer' by Graham Wideman, September '81 ETI.

If all is OK with the deck, the record level may be too low. Listen to the tape once you've stored a program. The signal should be loud and with little or no background noise. If the record level is too low then reduce the value of R25. About 2k7 should more than cover it.

Most troubles with the cassette interface may be traced to head alignment or magnetisation problems, and unreliable connectors.

program loads, you'll hear an outrageous noise from the speaker. When the noise stops, stop the tape.

Putting your program onto tape is fine — but how do you get it off? Simple. Use key **4**. This **loads the taped program into memory**.

Press the RESET key, and then press key 4. Then set the cassette going. As the program is loaded in, the tone being put out by the ETI-660 will change. When the TV display comes back on, it's finished. Stop the tape.

There is a further command, key **6**, and this allows you to **execute a machine code program** at the address shown on the TV. However, the use of this is a little difficult. In other words, I haven't got room in this article to explain the 1802's instruction set, so I'll leave that one till later.

CHIP-8 proper

Fine. Now you're quite happy about what to do with your CHIP-8 program once you've written it. All you have to do now is to find out how CHIP-8 works. You use key 0 to enter it, then press key 8 to run it.

Each CHIP-8 instruction is two bytes long.

Take a look at Table 1. Don't expect to understand it, just take a look at it.

Looks complicated, doesn't it? Well, at first it is. But we'll split it into easily digestable chunks.

Before I tell you how the instruction set works, I'd better tell you how the table works. Each four-digit code in the leftmost column is one of the instructions. Now each instruction is two bytes long. That's four hexadecimal digits.

Taking the first instruction first (makes sense?), the code is 0MMM. Now this could be anything from 0000 to 0FFF. What I'm trying to get over by putting the Ms in is that where an M appears, you could have anything from 0 to F.

The mnemonic is just a way of showing the same thing in a more readable form. So the mnemonic for 0123 would be CALL 123. This would make the machine jump to a machine code subroutine at 0123 (the locations that it will jump to will always start with 0 because of the small size of the system).

So if you wanted to make the machine jump to a machine code subroutine (doesn't matter that you don't know what a subroutine is just yet) at location 0456, you would write CALL 456 down when you were planning your program, and when the time came to enter it into the machine, you would put in 0456.

By the way, as each instruction in CHIP-8 takes up two bytes, it takes two operations of the STEP key to input each full instruction.

So if the first instruction in the program (which, remember, starts at location 0600) was to be a CALL 456, then what you would do would be to press the RESET button, then press 0, then enter 0600, then press STEP, then enter 04, then press STEP again, then enter 56. And so on for the rest of the program.

Now I'll start describing the instruction set. I'll put the name of each instruction in **bold** type as I come to it, so that you can find it quickly later.

The first instruction is, of course, the **0MMM**. I've more or less covered this one — all that it does is to send the machine to a *machine code* routine. As I haven't told you how to write machine code routines, I'll leave it at that.

The next instruction is **1MMM**. This causes the CHIP-8 interpreter to jump to location 0MMM for the next instruction. Now normally, the interpreter will do the instructions in the order in which it finds them in memory, starting at location 0600, then 0602 (two locations per instruction, remember), then 0604, etc. But say you want it to go into a 'loop', repeating the instructions in locations 0600 to 0608 endlessly?

The answer would be to put an instruction in 0608 that causes the normal sequence of the 'execution' of the instructions to be changed. If you put a 1600 in location 0608 (and the mnemonic for this would be GOTO 600), then the machine would do the instruction in location 0600 at the start of the program, then 0602, then 0604, then 0606, then it would come to 0608. Following the GOTO instruction, the next one it would do would be 0600 again, followed by 0602, 0604, 0606, 0608, then back to 0600, 0602 ... and so on.

Of course, the GOTO instruction can not only make the interpreter jump *back* in the program — it can also make it jump forward.

Subroutines

The next instruction we come to is one of the most complicated. It's the **2MMM** instruction — the DO MMM.

Now what this tells the machine to do is similar to the effect of the GOTO MMM instruction. The difference is that, whereas when it got to a GOTO MMM, the computer goes where it's told, with a *DO* MMM it remembers where it jumped from. So for example, you might have a segment of program somewhere in memory that does something useful that you want to do quite often from different parts of the program. A program segment like that is called a subroutine (for historical reasons).

The DO MMM instruction allows you to send the machine out of the main part

of the program to the start of that subroutine, and a special instruction at the *end* of the subroutine will send it *back* once the subroutine is finished.

The special instruction is 00EE. So, for example, the following sequence will cause the subroutine to be done twice at different parts of the main program (the dots '...' mean various program instructions that aren't important to what I'm showing you):

location	code	mnemonic	comments
0600	...		
...	...		
0610	2640	D0 640	first 'call'; **main program**
...			
0624	2640	D0 640	second 'call'
...	...		
...	...		
...	...		
0640	...		
...	...		**subroutine**
0652	00EE	RET	sends it back

When the machine got to 0610, it would immediately jump to 0640, then it would do 0640, 0642, 0644, ... and when it got to 0652, it would jump back to 0612 (the instruction *after* the one that 'called' it).

It would then go 0614, 0616, ... and when it got to 0624, it would again jump to 0640, go through the subroutine again, and when it got to 0652, it would jump back to 0626 — the instruction after the one that it was called from.

You can see that, by using this sort of thing, you can put in subroutines (sometimes called 'routines' for short) to do all sorts of useful things. They would of course have to be saved on tape with the main program.

TABLE 1 — CHIP-8 INSTRUCTION SET

CODE	MNEMONIC	FUNCTION
0MMM	CALL MMM	jump to a machine code subroutine at location 0MMM. The subroutine must end in a RET (D4)
1MMM	GOTO MMM	jump to location 0MMM (at which the next CHIP-8 instruction is to be found)
2MMM	DO MMM	jump to a CHIP-8 subroutine at location 0MMM. The subroutine must end in 00EE (which is the CHIP-8 equivalent of RET)
3XKK	SKIP IFVX=VKK or SKF VX=KK	skip the next instruction if variable VX holds the value KK
4XKK	SKIP IF NOT VX=KK or SKF VX=KK	skip the next instruction if variable VX does *not* hold the value KK
5XY0	SKIP IF VX=VY or SKF VX—VY	skip the next instruction if variable VX holds the same value as variable VY
6XKK	VX=KK	set variable VX to value KK
7XKK	VX=VX+KK	add KK to the value at present in VX
8XY0	VX=VY	set the value of VX to that in VY
8XY1	VX= VX OR VY	logical OR VX with VY, put the result into VX. Logical OR is like putting the two binary values through an OR gate bit by bit
8XY2	VX= VX AND Y	same as 8XY1, but logical AND
8XY3	VX= VX XOR VY	same, but with logical exclusive OR
8XY4	VX= VX + VY	add VX and VY, put the contents into VX. If there is an overflow, VF becomes 01, otherwise it becomes 00
8XY5	VX= VX − VY	subtract VY from VX and put the result into VX. If there is an underflow, VF becomes 00, otherwise it becomes 01
9XY0	SKIP IF NOT VX=VY or SKF VX=VY	skip the next instruction if the value of VX is not the same as that of VY
AMMM	I=MMM	set the value of the special 'memory pointer' variable I to 0MMM.
BMMM	GOTO MMM + V0	the value in variable V0 is added to 0MMM, and the machine will jump to the location whose address is the result of the addition
CXKK	VX = RND AND KK	produce a random byte, logical AND it with KK, and put the result into VX
DXYN	SHOW N AT VX, VY	starting at the address held in the 'memory pointer' variable I, or SHOW N@VXVY take N bytes out of memory and put them on the screen at a position VX from the left and VY from the top
EX9E	SKIP IF VX=KEY or SKF VX=KEY	skip the next instruction if VX is equal to the key being pressed
EXA1	SKIP IF NOT VX=KEY or SKF VX=KEY	opposite of EX9E
FX00	PITCH = VX	set the pitch(i.e: frequency)of the tone generator (beeper) to VX
FX07	VX = TIME	set VX to the current timer value
FX0A	VX = KEY	wait for a key to be pressed, then put the value of the key in VX
FX15	TIME = VX	set the value of the timer to VX
FX18	TONE = VX	set the duration of the tone burst (length of beep) to VX times 20 milliseconds (when VX is 50, it will be one second)
FX1E	I = I + VX	increase the value of the memory pointer by VX
FX29	I = DSPLY VX or I = DSP, VX	set the value of the memory pointer I so that when the next DXYN instruction is executed, the pattern shown will be the number that is the least significant digit of VX.
FX33	M(I)= DECML VX or MI=DEQ, VX	convert the value in VX to a three-digit decimal (base 10) number, and store the three digits in memory locations starting at location I (i.e: I, I+1 and I+2)
FX55	M(I)=V0:VX	put the values of variables V0 through VX into memory, starting with V0 at location I, V1 at location I+1, etc ... up to VX. Then increase the value of I so that it is I+X+1 (i.e: so that it points to the next location above the ones used for the variables)
FX65	V0:VX=M(I)	take the values of variables V0 through VX *out* of memory in the same way as the FX55 instruction. I becomes I+X+1. V0 comes out of location I.

Variables and loops

The next instruction is the 3XKK. Before I go too far into this one, I'll explain about the interpreter's **variables**.

Now the names of these variables are V0, V1, V2, V3 ... up to VF (there are sixteen altogether, and the second digit of the name of each of them is a hexadecimal one).

Each of these variables can 'hold' a value one byte long. That is, each of them can hold a value from 00 to FF.

Now the value of each of these variables can be changed at any time during a program. If you like, they're like sixteen little blackboards, each with a number written on it (from 00 to FF).

Each of the blackboards has a name — the first one is V0, the next V1 ... and so on up to VF.

Some of the instructions that the machine can execute allow new numbers to be written into the variables. Of course, when you write a new number in, the one that was there before will disappear (each blackboard can only hold one number at a time — I said they were *little* blackboards).

Other instructions allow the values in the variables to be read, added together, subtracted, etc.

So let's take a look at the **3XKK** instruction. What it says is "skip the next instruction if the value in VX is KK". The mnemonic is SKIP IF VX=KK, or if you want a shorter one (and a mnemonic is only for *your* convenience, the writer of the program), then another one is SKF VX=KK. ▶

Now the 'X' in 3XKK tells the machine *which* of the sixteen variables you're talking about. So 3245 would mean "skip the next instruction if variable V2 was equal to 45 (hex)".

The usefulness of this sort of instruction is in 'loops'. If you want to repeat a series of steps in a program a definite number of times, then you could set V2 (for example) to 0 at the start of the program, then do the sequence you wanted to repeat, then have an instruction that increased V2 by 1 (and we'll come to such an instruction later), then have a 3205 instruction.

The 3205 would skip the next instruction if V2 held the value 05. The instruction *after* the 3205 would be a GOTO MMM instruction which would send the machine back to the start of the sequence that was to be repeated.

So the first time the machine went through the sequence, V2 would be 00 until it got to the instruction to add 1 to it. It would then be 01, and when it got to the 3205, it would look at the value of V2 and see if it was 05. It wouldn't be (we've just shown that it would be 01), and so it would do the next instruction, which would be the GOTO.

That would send it back to the start of the repeated instructions (we say "back round the loop"). When it came to the instruction that added 1 to V2, it would leave V2 with the value 02.

See what's happening? Each time it goes round the loop, V2 increases in value by 1, until finally it gets to the value 05. *This* time, when it comes to the 3205 instruction, it would look at the value of V2, see that it *was* 05, and skip the next instruction (which is the GOTO). So it would *not* go round the loop again.

Effectively, what the 3205 instruction is doing is to tell the machine to repeat the loop 5 times.

Anyway, that's what the 3XKK instruction does.

The **4XKK** instruction does much the same, except that it will skip if VX is *not* equal to KK.

The **5XY0** compares the values of two variables, VX and VY. So a 5670 would skip the next instruction if the value in V6 was equal to the value in V7.

The **6XKK** sets VX to the value KK. If you like, this tells the machine to write the value KK on blackboard VX. 6789 would tell the machine to write the value 89 in variable V7. Another way of saying that is to say "set V7 to 89".

Increment/decrement

The **7XKK** is the one we mentioned earlier (honest) that allows you to *increase* the value held in VX by KK. So a 7201 would add 1 to the value stored in V2, and put the result back into V2. In other words, it would increase the value of V2 by 1. Another name for this is 'incrementing' (no, that's nothing to do with the police interrogations). We can say that the instruction *increments* V2.

The opposite of incrementing, by the way (*decreasing* the value of a variable by 1) is *decrementing*, and not excrementing.

Manipulations

The **8XY0** instruction sets variable VX to the value that is stored in VY. So if V2 was 03 and V4 was 89, an 8240 instruction would leave V2 at 89 and V4 at 89 also. (It's worth sitting down and thinking about that one).

The **8XY1**, **8XY2** and **8XY3** instructions are very similar. Each one of them takes the values in VX and VY, does something with them, and puts the result back into VX.

The 8XY1 does a logical OR on the two values. Now in order to explain this one, you have to convert the hex values that we usually work with back into binary.

Let's say that the value in V1 is 34, and the value in V2 is 9A. That gives them the following binary values:

variable	hex value	binary value
V1	34	00111000
V2	9A	10011010

Now what the 'logical OR' does is to take corresponding bits of the values of V1 and V2 and put them through an OR gate. Starting from the left, 0 OR 1 would give 1; 0 OR 0 would give 0; 1 OR 0 would give 1; and 1 OR 1 would give 1; and so on.

So the result would be 10111010, which in hex is BA. That's how it works. The usefulness of it is that for example, 40 OR 08 is equal to 48.

The logical AND (8XY2) works in much the same way, except that it puts the individual bits through an AND gate (metaphorically). The usefulness of that is that 48 AND F0 is 40; 48 AND 0F is 08.

The last one, 8XY3, gives a rather weird function (again bit by bit) called 'exclusive OR', or 'XOR' for short. What this says is that the result will be 1 if the inputs are different. So 1 XOR 0 is 1, but 1 XOR 1 is 0.

Plus and minus

The instruction **8XY4** is quite straightforward — the value in VX is added to the value in VY, and the result ends up in VX. But what happens when you try to add F0 to 42? The result (if you work it out by hand) would be 132 (hex). Now the result is going to go into VX, and VX will only hold two hex digits. So VX becomes 32, and VF (always VF, no matter what the values of X and Y are) is set to 01.

Of course, if the result is less than 100, VF is set to 00.

8XY5 is the same as 8XY4, except that the value in VY is *subtracted* from the value in VX. If the result is less than 00, VF is set to 00, but if the result is not less than 00, VF is made 01.

Skip to my loo

9XY0 is simply "skip the next instruction if the value in VX is not the same as the value in VY" — the opposite of the 5XY0 instruction.

You and 'I'

Now comes the time to introduce the 'memory pointer', I. This is the same as the rest of the variables V0 through VF, except that it can hold three hex digits (while they can hold only two).

It needs the extra room because it is used to hold the address of memory locations, and three hex digits is the smallest useful space in which that can be done.

The **AMMM** instruction simply sets the value in I to MMM, so that A123 would leave I holding the value 0123.

We'll come back to the I variable later.

Branches

BMMM is a strange one. What it says is "branch to the memory location which has the address of 0MMM plus the value of V0". So if V0 had the value 05, the instruction B610 would cause the machine to go to the instruction at location 0615.

This instruction is useful for accessing a series of short program segments arranged in a 'table' starting at 0MMM. As the value in V0 is changed, the jump will take the machine to different parts of the table. V0 is sometimes called the 'offset' and MMM the 'base' in this context.

Randomise

CXKK is a nice one. What it does is to produce a random byte (which the interpreter does all by itself), and then to AND this random byte with the value KK and put the result into variable VX.

So C234 will generate this random variable, from 00 to FF, then AND the random byte with 34. It's worth working out for yourself, but I can tell you right away that the *maximum* value of the result of that AND operation will be 34. The result is then put into V2.

So what the C234 instruction says is "set V2 to a random value between 0 and 34 (hex)".

Output and the screen

Now we come to some of the output instructions. These deal mainly with outputting information to the screen. The CHIP-8 program can, of course,

output to the screen directly by altering the memory locations which store the information that is to be displayed directly. (Each dot on the screen is one bit of memory, and the whole screen is stored in memory in a block starting at 0480). However, the CHIP-8 interpreter allows a much greater control over what's going on.

In the CHIP-8 interpreter, the screen 'locations' are split into 3F horizontally and 3F vertically. I'll explain that a bit further.

The **DXYN** instruction causes the machine to take N bytes out of memory, starting at the memory location whose address is given by the I pointer. So far, so good.

Now the DXYN causes the machine to put an image of these bytes onto the screen. What I mean by that is this: assume that the screen is blank to start with. Let's say that the DXYN is D321. So the number of bytes to be taken from the locations starting at I is 1.

Let's further say that the value of that one byte is 83. The binary for that is 10000011. The machine would put that binary image onto the screen by turning on one dot, then leaving the next five off, then turning on the next two.

By 'turning on a dot', I mean that a small part of the screen will go bright. The size of each dot is a few square millimetres (in fact, the numbers that you see when you turn the computer on are made up of these dots (also called 'pixels')).

Which part of the screen would this pattern appear in? That's where the XY part of the DXYN instruction comes in. X and Y refer to two variables. The first gives a number of dot widths across the screen, and the second gives a number of dot heights down the screen.

So let's say that the instruction was D321, and that V3 held the value 05 and V2 held the value 08. The first dot in the image would be five dots to the right and eight dots down from the extreme top left of the screen.

Now all this may seem unnecessarily complicated, but consider this: once you have set I to point at the image that you want (and set the relevant bytes of memory so that the dots you want in that image are right), changing the values held in VX and VY will make the image 'move' across the screen.

Another point I might bring to your attention is that, where the instruction is about to turn on a dot that's already on, it will turn it off. In fact, rather than saying 'turn the dot on', what I should have said was 'change the state of the dot'.

This means that the image that you put on the screen can be removed (set back to its original blank state) by executing the same DXYN instruction.

EX9E is an easy one. It skips the next instruction if the number on the key that you press is the same as the value in VX. This means that you can wait for a particular key to be pressed, ignoring all other keys, by making the instruction after the EX9E a jump back to the EX9E instruction (think about it).

EXA1 is the same as EX9E, except that it will skip if the key you press is *not* the one whose value appears in VX.

Play it, Sam

Another nice feature of the CHIP-8 interpreter is that it has a musical tone generator — the speaker in the machine can be made to emit a variety of musical notes.

The **FX00** instruction sets the pitch (frequency) of this tone. The X refers to one of the variables. So F200 would set the pitch of the tone generator to the contents of variable V2. The higher the value of the contents of the variable, the *lower* the pitch of the tone.

Once the PITCH variable (and it behaves just like a variable) has been set, all tones from the generator will be at that pitch until another PITCH=VX instruction is encountered (or the program ends).

Period

Yet another nice feature offered by the interpreter is the timer. This is like another variable, in that the one-byte value in it can be set to a given value, and the value in it can be used to set another V-type variable.

The difference is that the value in the timer variable will reduce by one every 20 milliseconds (one-fiftieth of a second) or so.

Having set the timer with a particular value (the time that you want the program to pause for, for example), you can then arrange to check to see if the period has ended by using the **FX07** instruction. This sets VX to the current timer value.

It's then a simple matter to use a 3XKK instruction to check the value in VX to see if it's zero, looping back to the FX07 instruction each time if it is not. The "RANDOM PITCH AND BLOCKS" example program later in the article uses this trick.

Pounding the keys

For keyboard input, the interpreter makes life really easy. The **FX0A** instruction waits for a key to be pressed, and then sets VX to the number on the key. Simple.

Remember I said that you could set the value of the timer? Well, **FX15** is how you do it. It sets the timer to the value of VX.

Having set the pitch of the tone, how do you make the thing sound? Well, the **FX18** does this. Not only does it set the tone off, it also allows you to set the length of time that it sounds for. One word of warning, however — the program will *not* wait for the end of the beep. That means that, if you want to produce a series of beeps (as in the "RANDOM PITCH AND BLOCKS" program), you have to arrange a timer loop so that the program will wait for the beep to finish before trying to do another beep. (If it does try to do another beep, the buzzer will be reset, and the second beep will override the first, making the first one very short).

Numbers

Remember the memory pointer? Well **FX1E** increases the value of I by the value of VX. This is useful if you want to display a large area of memory on the screen. You can move the I pointer through memory in steps of the value of VX.

The **FX29** instruction allows you to draw numbers on the screen without really thinking about it too much. What it does is to set the I pointer so that it just happens to be pointing at a block of memory which holds the *shape* of the number which is the right-hand digit of the value of VX.

To give an example, say that you want to display the number 9 at the top left hand corner of the screen. The following program segment will do just that:

mnemonic	code	comments
V1 = 09	6109	it's the right-hand digit we're interested in
V2 = 0	6200	
V3 = 0	6300	
I = DSPLY V1	F129	I will be pointing at the digit 9
SHOW 5 AT V2, V3	D235	It takes 5 bytes to show a number on the screen. This will show it at screen location 0, 0 — the top left hand corner.

You might like to try this segment on your machine; make the last code (after the D235) at 0000 — this will send it back to the monitor once it's finished. Try changing the values in V1, V2 and V3 to see what effect it has.

All this is fine — but what about games? Most people like to see scores, etc, in decimal, rather than in hex!

What the **FX33** instruction does is to take the value of VX and convert it to a three-digit decimal number (somewhere between 0 and 255). It then puts the result into the three memory locations which start with the one pointed to by I. (I is not altered). So location I becomes the 'hundreds', location I+1 becomes 'tens' and I+2 becomes 'units'. How do we get the values out of there? Later. ▶

Now comes the time to introduce the concept of a 'stack'.

Stacks

When you jump to a subroutine (remember subroutines?) you might want to 'save' the values that are in the variables at present, so that you can go on using the same variables in the main program when the machine finishes the subroutine. So there should be some way of automatically saving the values of the variables in memory while the subroutine is going on, and then retrieving them afterwards.

A stack is a place in memory where the values in variables are kept for storage (at least, that's one of the uses for it).

The important thing about a stack is that, if *during the subroutine* you want to store some more variables, the stack can accommodate them.

It works like this: first, you set I to the start of an area of memory that you know is available (0700 is a good one). Then when you execute the **FX55** instruction, the machine will start by putting the value of V0 into memory at location I.

The value in I will then be increased by one. That's important — I'll tell you why later.

The next variable, V1, will be put into memory at the location *now* pointed at by I, then I is increased again. In fact, all variables starting at V0 and finishing at VX will be stored. So if the instruction is F355, variables V0, V1, V2 and V3 will be stored.

Notice that, at the end of this instruction, I will *still* be pointing at an *available* memory location (0703, in this example).

So if you now changed the values of some of the variables, you could do another FX55 and put these new values into memory at the locations after the old values. That's why it's called a 'stack' — the values are thrown into memory in the same way that a stack of magazines will pile up — one after the other.

Now in order to get the variables out, you would use the **FX65** instruction. First you would have to set I back to the start of the 'stack', then the FX65 would pull the variables out in the same order that the FX55 pushed them in.

The value of the memory pointer I is increased by the FX65 instruction in the same way that it is increased by the FX55 instruction. In fact, the FX65 is the way that we would get out of memory the results of the FX55 instruction.

The following example will take the value that V3 is set to, and display it *in decimal* at the top left of the screen:

mnemonic	code	comments
V3 = 80	6380	this is the value we're trying to display
V4 = 00	6400	
V5 = 00	6500	V4, V5 are the screen position — 00, 00 is the top left
I = 0700	A700	points to a free area of memory
M(I) = DECML V3	F333	this is the actual 'guts' of the program, and it puts the three 'results' digits at 0700, 0701 and 0702. I is still 0700.
V0:V2 = M(I)	F265	this takes the results out and puts them into V0 through V2 (which is why we used V3 to hold the input at the first line).
I = DSPLY V0	F029	ready to put the first digit on the screen
SHOW 5 AT V4, V5	D455	as per the last example
I = DSPLY V1	F129	next digit
V4 = V4 + 04	7408	moves the screen location along so that the next digit will not be on top of the first one
SHOW 5 AT V4, V5	D455	next digit is now on the screen
I = DSPLY V2	F229	last digit
V4 = V4 + 04	7404	move screen position again
SHOW 5 AT V4, V5	D455	last digit — finished!
	0000	returns control to monitor.

TARGET PRACTICE

```
0600  67 19  V7=19
0602  68 00  V8=00
0604  26 C2  DO 06C2
0606  26 DE  DO 06DE
0608  65 25  V5=25
060A  66 0D  V6=0D
060C  26 E6  DO 06E6
060E  D5 65  SHOW 5MI@V5V6
0610  CD 01  VD=RND
0612  3D 01  SKF VD=01
0614  6D 07  VD=07
0616  27 0C  DO 070C
0618  64 10  V4=10
061A  63 0B  V3=0B
061C  83 D4  V3=V3+VD
061E  A6 BF  I=06BF
0620  D3 41  SHOW 1MI@V3V4
0622  6C 00  VC=00
0624  6B 80  VB=80
0626  60 03  V0=03
0628  E0 A1  SKF V0≠KEY
062A  6B FF  VB=FF
062C  60 06  V0=06
062E  E0 A1  SKF V0≠KEY
0630  6B 00  VB=00
0632  60 09  V0=09
0634  E0 A1  SKF V0≠KEY
0636  6B 01  VB=01
0638  3B 80  SKF VB=80
063A  26 9A  DO 069A
063C  A6 B3  I=06B3
063E  D5 65  SHOW 5MI@V5V6
0640  85 94  V5=V5+V9
0642  86 A4  V6=V6+VA
0644  45 20  SKF V5≠20
0646  26 F6  DO 06F6
0648  45 3B  SKF V5≠3B
064A  26 FA  DO 06FA
064C  46 00  SKF V6≠00
064E  26 FE  DO 06FE
0650  46 1B  SKF V6≠1B
0652  27 08  DO 0708
0654  D5 65  SHOW 5MI@V5V6
0656  3F 00  SKF VF=00
0658  16 80  GO TO 0680
065A  4C 00  SKF VC≠00
065C  16 24  GO TO 0624
065E  A6 BF  I=06BF
0660  D3 41  SHOW 1MI@V3V4
0662  33 3E  SKF V3=3E
0664  16 6E  GO TO 066E
0666  47 00  SKF V7≠00
0668  16 68  GO TO 0668
066A  27 0C  DO 070C
066C  16 10  GO TO 0610
066E  73 02  V3+02
0670  44 00  SKF V4≠00
0672  6B 01  VB=01
0674  44 1D  SKF V4≠1D
0676  6B FF  VB=FF
0678  84 B4  V4=V4+VB
067A  D3 41  SHOW 1MI@V3V4
067C  4F 00  SKF VF≠00
067E  16 3C  GO TO 063C
0680  60 02  V0=02
0682  F0 18  TONE=V0
0684  A6 BF  I=06BF
0686  D3 41  SHOW 1MI@V3V4
0688  A6 B3  I=06B3
068A  D5 65  SHOW 5MI@V5V6
068C  26 DE  DO 06DE
068E  78 0A  V8+0A
0690  26 DE  DO 06DE
0692  47 00  SKF V7≠00
0694  16 94  GO TO 0694
0696  27 0C  DO 070C
0698  16 08  GO TO 0608
069A  6C 01  VC=01
069C  60 07  V0=07
069E  F0 18  TONE=V0
06A0  26 C2  DO 06C2
06A2  77 FF  V7+FF
06A4  26 C2  DO 06C2
06A6  00 EE  RET
06A8  01 7C
06AA  7C FE
06AC  7C 7C
06AE  70 7C
06B0  38 7F
06B2  7F 7C
06B4  7C 7C
06B6  7C 7C
06B8  38 38
06BA  38 38
06BC  38 3E
06BE  E0 80
06C0  00 D4
06C2  A7 F8  I=07F8
06C4  F7 33  MI=V7(3DD)
06C6  6E 1B  VE=1B
06C8  6F 10  VF=10
06CA  F2 65  V0:V2=MI
06CC  F0 29  I=DSP,V0
06CE  DF E5  SHOW 5MI@VFVE
06D0  6F 15  VF=15
06D2  F1 29  I=DSP,V1
06D4  DF E5  SHOW 5MI@VFVE
06D6  6F 1A  VF=1A
06D8  F2 29  I=DSP,V2
06DA  DF E5  SHOW 5MI@VFVE
06DC  00 EE  RET
06DE  A7 F8  I=07F8
06E0  F8 33  MI=V8(3DD)
06E2  6E 00  VE=00
06E4  16 C8  GO TO 06C8
06E6  C9 01  V9=RND
06E8  39 01  SKF V9=01
06EA  69 FF  V9=FF
06EC  CA 01  VA=RND
06EE  3A 01  SKF VA=01
06F0  6A FF  VA=FF
06F2  A6 B3  I=06B3
06F4  00 EE  RET
06F6  69 01  V9=01
06F8  16 EC  GO TO 06EC
06FA  69 FF  V9=FF
06FC  16 EC  GO TO 06EC
06FE  6A 01  VA=01
0700  C9 01  V9=RND
0702  39 01  SKF V9=01
0704  69 FF  V9=FF
0706  00 EE  RET
0708  6A FF  VA=FF
070A  17 00  GO TO 0700
070C  6E 08  VE=08
070E  A6 A9  I=06A9
0710  DD EF  SHOW FMI@VDVE
0712  7E 0F  VE+0F
0714  A6 B8  I=06B8
0716  DD E6  SHOW 6MI@VDVE
0718  6E 10  VE=10
071A  60 08  V0=08
071C  80 D4  V0=V0+VD
071E  8F 00  VF=V0
0720  A6 BE  I=06BE
0722  DF F2  SHOW 2MI@VFVE
0724  00 EE  RET
```

Shoot by pressing key 3 (up), 6 (straight), or 9 (down) to hit the moving target. The bottom number shows the number of shots. Each hit scores 10 points (tip number).

You can try playing about with the values in V3, and also in V4 and V5 — remember that V4 is increased from its *original* value, so making it a different value to start with will move all three digits (which is just what we want).

Miscellaneous

We've covered (phew!) all of the instructions in the table so far — but there are a few others which are actually 'machine code' subroutines built into the ROM (but don't let that stop you using them):

code	function
0000	return to monitor
00E0	clear the screen
00EE	return from CHIP-8 subroutine
00F8	turn the screen on
00FC	turn the screen off (i.e: display a blank)
00FF	do nothing

RANDOM PITCH AND BLOCKS

address	mnemonic	code	comments
0600	V0 = V8	6008	this is the variable which holds the timer initial value
0602	V1 = RND AND FF	C1FF	sets V1 to a random value between 00 and FF
0604	PITCH = V1	F100	sets the frequency of the tone to the random value
0606	TIME = V0	F015	load timer with 08
0608	TONE = V0	F018	make a tone of the same length as the timer period
060A	V2 = TIME	F207	sets V2 to the current timer value so that it can be kept an eye on by the next instruction
060C	SKF V2 = 00	3200	skip the next instruction if the timer has reached 00
060E	GOTO 060A	160A	loop back to wait for the timer if it's not finished yet
			Now for the blocks on the screen ...
0610	VA = RND AND 37	CA37	sets VA to somewhere between 00 and 37
0612	VB = RND AND 37	CB37	sets VB to somewhere between 00 and 37
0614	I = 61A	A61A	sets I to the end of the program (where there is spare memory)
0616	SHOW 7 @ VA, VB	DAB7	puts the block at I onto the screen at the random position VA, VB
0618	GOTO 0600	1600	go and do it all again ...
061A ...	0620:		these locations will hold the 'pattern' that will end up on the screen. (Try filling them with FFs).

SONG IN THE KEY OF YALE

address	mnemonic	code	comments
0600	I = 0622	A622	set the memory pointer to the 'data' area
0602	V1 = 0F	610F	
0604	V0 = FF	60FF	
0606	V0 = V0 + 01	7001	this will set V0 to 00 the first time through (the result of the addition will be 100, and the first digit will end up in VF).
0608	V0 = V0 AND V1	8012	this makes sure that V0 is between 00 and 0F (if you like, it 'masks' out the left-hand digit.
060A	SKF V0 = KEY	E09E	V0 is busy being the numbers from 00 to 0F one by one — when you press a key, this loop will try all of the possible values until V0 matches the key.
060C	GOTO 0606	1606	loop back and try the next value of V0
060E	V2 = 08	6208	
0610	I = I + V0	F01E	points I at one of the locations between 0622 and 0632, depending on what V0 is (and therefore on what key was pressed)
0612	V0:V0 = M(I)	F065	this loads the value in the memory location pointed to by I into V0
0614	PITCH = V0	F000	... and this is why — each location from 0622 to 0630 holds the pitch which corresponds to that key
0616	TIME = V2	F215	
0618	TONE = V2	F218	set up the timer and the beeper for a beep of length V2 (which is 08).
061A	V2 = TIME	F207	set V2 to keep an eye on the time
061C	SKF V2 = 05	3205	let the timer go until it gets to 5, then look for the next note (if there is none, it will sound for 100 milliseconds after the key is released).
061E	GOTO 061A	161A	keep looking at the timer ...
0620	GOTO 0600	1600	next key
0622	data	8071	these locations hold the required pitch settings for C major — perhaps you can change them to get sharps and flats?
0624		655F	
0626		544B	
0628		433F	
062A		3832	
062C		2F2A	
062E		2521	
0630		1F1C	

A word of explanation about these — **00FC** will cause the screen to go blank until a **00F8** is executed. This allows you time to write things on to the screen — and then turn it on all of a sudden.

However, **00E0** clears all of the dots from the screen — makes it blank until you turn some of the dots back on.

0000 is a good one — it just finishes the program and sends you back to the monitor. For programs which do something and then stop, this is very useful.

Now, **00FF** may seem a bit useless. But it's an instruction that you will probably end up using more than any other. The reason for this is that, with a long program, if you want to put in one or two extra instructions to make the program work the way you want to, you will have to change the locations of all the program from there on. However, if you put in a few 00FFs at various points in the program, you can get away with re-entering only a small part of the program.

Similarly, any instructions that you want to take out can be replaced by 00FFs. All a 00FF does is to send the machine to the next instruction.

Examples

The first example is instructive rather than useful. What it does is to write blocks of dots on the screen, at the same time playing a sequence of notes of random pitch and duration.

If you run this program, you will find that the 'blocks' do not appear at a truly random position, but that they fall into 12 areas on the screen. This is because finding a random value and then ANDing it with the limiting value (37, in this case) does not provide a completely linear distribution. Work that one out for yourselves — maybe you could find a way of putting the blocks on the screen randomly? (Clue: you might have to use the 8XY5 instruction).

The next example is a little more advanced — what it does is to turn your keyboard into a piano! As you press the keys, you will hear notes played in the key of C major.

A game — and colour

Included here is another, longer, example — a fully-fledged game called 'Target Practice'. It hasn't got the same level of commenting as the previous examples, so that leaves you to work out for yourself how it works internally.

The fully-optioned ETI-660 is capable of producing colour on screen — I haven't covered that this time because of lack of space, but we'll get to it later.●