An Easy Programming System

Joseph Weisbecker
1220 Wayne Av
Cherry Hill NJ 08002

This article describes a hexadecimal interpretive programming system which requires less hardware than high level languages such as BASIC, and which I feel is much easier to use than machine language. In my experience, hexadecimal interpretive programming is ideally suited to real time control, video graphics, games or music synthesis. It can be used with inexpensive computer systems consisting of a hexadecimal keyboard and only 1 K or 2 K of programmable memory. Expensive terminals and large memories aren't required. You can quickly and easily write useful programs that require five to ten times less memory than conventional high level languages without resorting to the tedious complexities of actual machine language.

Interpretive Programming

This programming approach isn't new, but surprisingly few people seem to be using it. The technique consists of designing a high level pseudomachine language that is more powerful for specific applications than conventional machine language. An interpretive program is then written to execute this new set of pseudoinstructions. Each pseudoinstruction is really just a code that specifies a machine language subroutine. This set of subroutines can be designed to perform any functions you might need for your application. By staying with a machine language format, and not using labels or English words for instruction codes, memory requirements are lower. By limiting the addressing range and number of variables, you can limit each pseudoinstruction code length to several bytes for further memory space savings. Interpretive programs for these powerful pseudomachine languages can require as few as 512 bytes of memory. It has seldom taken me more than a week to implement a new hexadecimal interpretive language, and I can then use it for years. The approach can be thought of as vertical microprogramming with the microprocessor machine language used as the microcode representation.

To illustrate the compactness of these types of programs, I wrote a video tic-tac-toe program using the CHIP-8 language described below. Only 500 program bytes were required versus 3000 bytes for an equivalent version written in BASIC. Besides saving memory, this also meant 2000 fewer keystrokes for initial program entry. In addition, the CHIP-8 interpreter was about eight times smaller than the BASIC interpreter. The CHIP-8 program ran on a 1.5 K memory system with a hexadecimal keyboard, while the BASIC program required an 8 K system with an ASCII keyboard and alphanumeric display. The CHIP-8 program took about 12 hours to design, hand code, enter and debug. I suspect that the BASIC version took at least as long on a much more expensive system.

This hexadecimal interpretive programming approach is important for two reasons. First, it reduces the cost of the hardware you need to get started in home computing. Second, it drastically reduces the amount of read only memory required in microprocessor based products such as controllers and video games. Read only memory cost is a significant factor in these types of products.

A detailed example will be used to illustrate the hexadecimal intepretive programming approach. The new RCA COSMAC VIP computer will be used for this example (see August 1977 BYTE, page 30, for a description of this computer). It is a low cost, single card computer containing 2048 bytes of programmable memory, a graphic video display, and a hexadecimal keyboard. I had this type of programming in mind when I incorporated features such as multiple program counters in the COSMAC (1802) microprocessor architecture.

The pseudomachine language used in my example will be one called CHIP-8, designed for use with the COSMAC VIP system. I will discuss using this language rather than describing the interpreter for it. Suffice it to say that the interpreter only requires 512 bytes and resides at memory locations 0000 to 01FF (hexadecimal). Programs written in the CHIP-8 language must start at memory location 0200 (hexadecimal). The sample program described will run on a 1024 byte memory system. This includes the CHIP-8 interpreter, the program, work area and video display refresh buffer. The program itself only requires 60 CHIP-8 instructions.

CHIP-8 Language

Table 1 describes the 31 CHIP-8 instructions provided in this pseudomachine language. Each instruction requires only two bytes (four hexadecimal digits). Memory addressing is limited to 4096 bytes so that only three hexadecimal digits are needed to specify a memory address. The number of variables has been limited to 16, labeled V0 to VF in this article. These are 1 byte variables or registers that can be modified or examined by CHIP-8 instructions. There is also a 2 byte memory address register called I, which is used by certain instructions. A real time clock or timer is provided. This timer can be set to any hexadecimal value between 00 and FF by the FX15 instruction. For example, if V2 contained hexadecimal 0A, an F215 instruction would set the timer to 0A. This timer is automatically decremented by one 60 times per second until it reaches 00. If the timer was set to 3C (decimal 60) it would reach 00 exactly 1 second later. This timer can be used to provide delays in game or control programs. A tone clock is also provided which can be set to cause a tone lasting from 1/60 to about 4 seconds.

An important feature of this type of language is that all variables (registers) are contained in memory. This means that debugging is generally lmited to examining memory locations, not internal microprocessor hardware registers. Astute readers will be wondering why I maintained a fixed 2 byte instruction length when variable instruction length was possible. Since absolute memory addresses are used, fixed 2 byte instructions avoid addressing confusion that increases programming errors. Also, any instruction can easily be replaced by a branch instruction of the same length for debugging breakpoints or program patching.

Graphic Display Approach

Before proceeding with a detailed programming example, readers will need to understand the video display system. Figure 1 shows the graphic display format used.

(Hexadecimal) Instruction	Operation
1MMM	Go to 0MMM
BMMM	Go to 0MMM + V0
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 = hexadecimal key (LSD)
EXA1	Skip next instruction if VX ≠ hexadecimal 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 = hexadecimal 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 = V0 : VX (I = I + X + 1)
FX65	Let V0 : VX = MI (I = I + X + 1)
00E0	Erase display (all 0s)
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)

Table 1: CHIP-8 instruction set. Note that invalid hexadecimal characters in the hexadecimal instructions listed are replaced by valid hexadecimal codes when a program is written. Thus B1000 might be a valid use of the BMMM instruction.

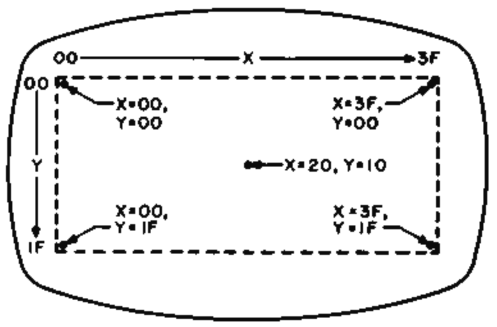


Figure 1: A drawing of the video display. The inner dashed square is the playing area. The range of X and Y is shown.

The dotted line indicates the area of the screen used for display. This display area consists of an array of spot positions 64 wide by 32 high. These spot positions represent bits in a 256 byte page of memory. When a memory bit is one, the spot position is on (white). The CHIP-8 language specifies spot positions on the screen by an XY coordinate system as shown in figure 1. The values of the X coordinate (horizontal spot position) can run from 00 to 3F (0 to 63 decimal). The values of the Y coordinate (vertical spot position) run from 00 to 1F (0 to 31 decimal). Any two variables (V0 to VF) can be used to specify the X and Y coordinates of a spot position on the screen.

The display instruction (DXYN) lets you show a pattern of spots on the screen. This pattern of spots can form a picture, letter, number, etc. Patterns are represented in memory by a list of one to 15 bytes. Suppose you want to display a rocket ship. You must first construct a rocket ship pattern on grid paper as illustrated in figure 2. The hexadecimal codes for this pattern can then be derived directly from the bit pattern.

To show this rocket ship on the screen with a DXYN instruction, you must first set I to the address of the rocket ship pattern byte list in memory. You must then set two variables to the X and Y coordinates at which you want the rocket ship pattern to appear on the screen. The X and Y coordinates specify the position of bit 7 of the first pattern byte on the screen. For example, the following short program would show the rocket pattern of figure 2 at the top left corner of the screen:

```
    Memory
    Address         Instruction
 (Hexadecimal)         Code                    Comments

     0200              6200         Set V2 = rocket X coordinate = 00
     0202              6300         Set V3 = rocket Y coordinate = 00
     0204              A20A         Set I = rocket pattern address = 020A
     0206              D236         Display 6 byte rocket pattern
     0208              1208         End loop
     020A              2070 )
     020C              70F8 }          Rocket pattern byte list
     020E              D888 )
```

The last hexadecimal digit of the display instruction (DXYN) must always specify the number of bytes in the pattern to be shown on the screen. The DXYN instruction compares each bit of the new pattern to be displayed with whatever is already displayed on the screen at the same spot positions. If a 1 bit is already displayed at the same position as a 1 bit in the new pattern to be displayed, a 0 will be shown on the screen at this spot position, and VF will be set to 01. In other words, the new pattern to be shown is combined with the pattern already showing on the screen via an EXCLUSIVE OR function. This means that after a pattern is shown on the screen it can be erased by showing the same pattern again with the same X and Y coordinates. Incrementing the X or Y coordinate and showing the pattern a third time would cause the illusion of motion. If the value of VF is 01 after showing the pattern on the screen, it means that the pattern touched or hit a previously displayed pattern.

The DXYN instruction permits displaying, erasing and moving individual patterns on the video screen. The ability to detect when one pattern meets another permits you to program chase, paddle and target games.

Decimal Digits and Random Bytes

Several instructions are provided to permit displaying decimal numbers on the video screen. These are useful for game scorekeeping, etc. An FX33 instruction converts the value of any variable (VX) to decimal form. Suppose I=0442 and V9=A7 (hexadecimal). An F933 would cause 01 to be stored in memory location 0422 (hexadecimal), 06 in 0423, and 07 in location 0424.

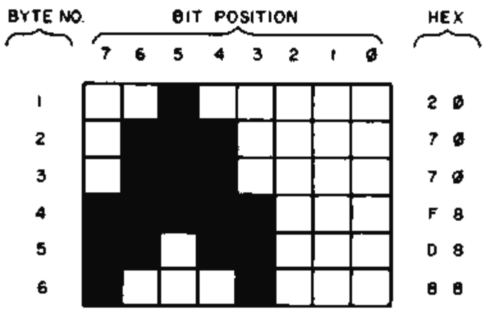


Figure 2: The definition of the rocket pattern is shown. The dark squares are encoded as a 1 bit in the appropriate byte. The actual value of each byte of the pattern is shown under the HEX column.

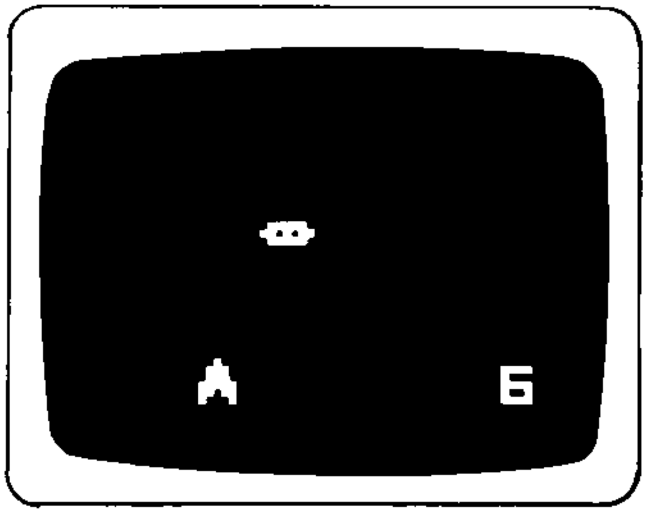


Photo 1: The actual video display of the game showing the rocket, UFO and score.

Since A7 in hexadecimal equals 167 in decimal, we see that the three bytes addressed by I represent the decimal equivalent of the value of V9. If I=0422, an F265 instruction could then be used to set V0, V1 and V2 to the values of the three bytes addressed by I above (01, 06 and 07). An FX29 instruction can then be used to set I to a 5 byte pattern representing any one of the three decimal digits. An F229 instruction would leave I addressing a 5 byte pattern for displaying the least significant decimal digit (7 in this example). A DXY5 instruction can then be used to display the decimal digit on the video screen at any desired position.

The above example illustrates the use of an FX65 instruction to transfer three memory bytes to three variables (V0 to V2). The FX55 instruction will store any number of variables in memory locations starting at the I address. These two instructions can be used to increase the number of variables by swapping sets of variables and memory bytes. Just remember that variables are always copied to or from memory in groups starting with V0 and ending with VX, inclusive.

It is often useful to generate random byte values. The CXKK instruction sets any variable (VX) to a random byte value. This random byte will have any bits matching 0 bit positions in KK (a 2 digit hexadecimal number) set to 0. For example, a C407 instruction would set V4 equal to a random byte value between hexadecimal values 00 and 07.

The remainder of the CHIP-8 instructions should be self-explanatory. The 2MMM instruction will transfer control to a subroutine which must be terminated by

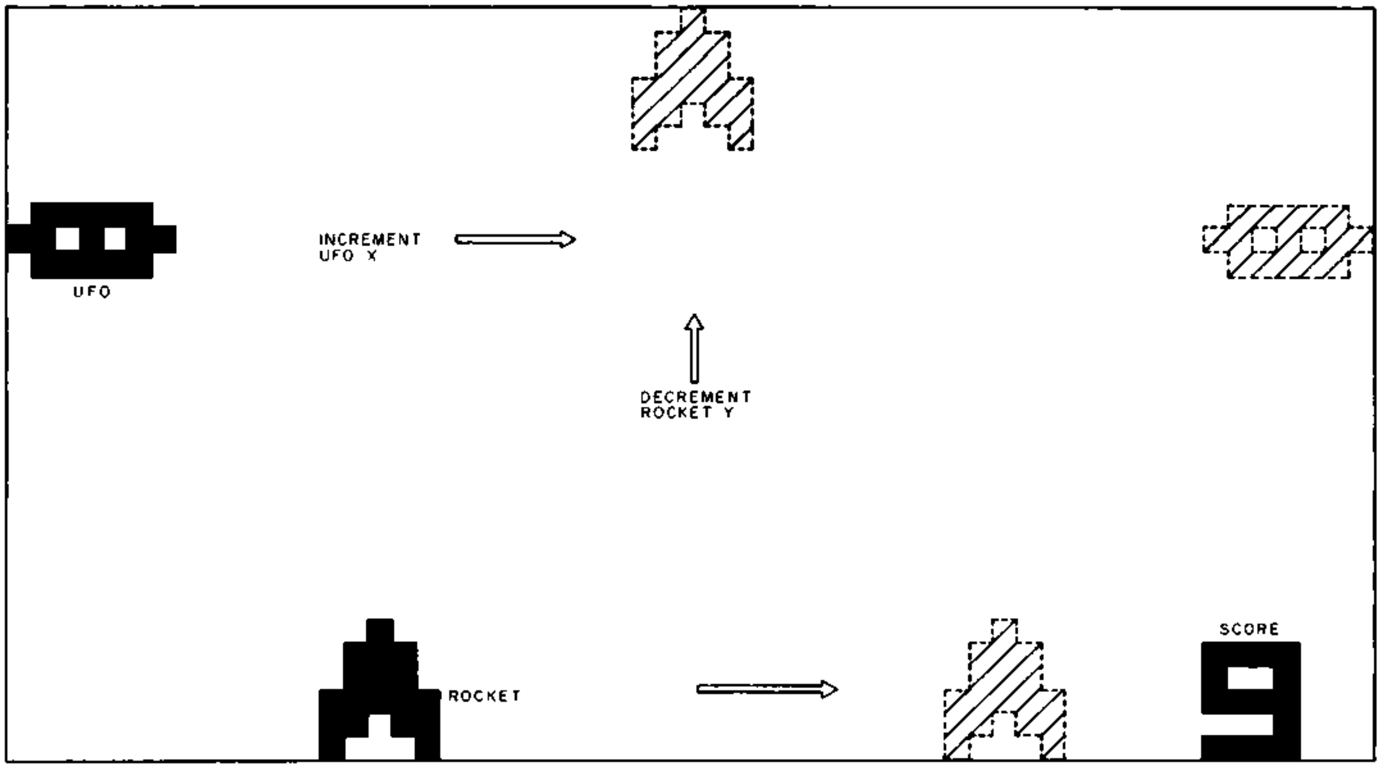


Figure 3: The range of rocket X values is from hexadecimal 0F to 2E. Rocket Y is decremented from hexadecimal 1A to 00. The UFO Y remains a constant hexadecimal 08, while the UFO X is incremented from hexadecimal 00 to 39.

instruction 00EE to return control to the instruction following the 2MMM. You can nest these subroutines. The 0MMM instruction permits a machine language subroutine to be inserted if required.

Designing a Video Game Program

A detailed example will illustrate how easily the CHIP-8 language can be used to program a real time video game. The first step always involves specifying the video display and the functions to be programmed. Figure 3 shows the display format chosen for this game. An enemy UFO will be constantly moving from left to right across the top of the screen. A single digit score will be displayed at the lower right. A rocket ship will appear at a random horizontal position along the bottom edge of the display area. You can launch this rocket by pressing key F on the hexadecimal keyboard. The rocket will then move vertically toward the top of the screen. When it reaches the top or hits the target UFO it will be erased and a new rocket will appear at the bottom of the screen. After nine rockets have been launched the game ends and no new rockets will appear. If you hit the UFO with a rocket the score is incremented by 1.

After specifying the positions of the various game patterns on the video screen as shown in figure 3, you must decide on how the 16 variables will be used in the program. Table 2 illustrates how we will use the variables in this example. Six variables (V3, V4, V5, V6, V7, V8) are needed to specify the X and Y coordinates of the three types of patterns involved (score, target UFO and rocket). We need two more variables (V1, V2) to keep track of the current score and number of rockets launched. V9 will be used as a flag that shows whether or not the current rocket has been launched. VA will be set to 01 if the rocket hits the UFO (ie: point scored) and V0 will be used for a working register in the program. VF is the hit flag and is automatically set to 01 when a hit occurs.

Flowcharting the Game

I believe you should always construct a detailed flowchart, such as the one shown in figure 4. Proper flowcharting will save hours of debugging and will simplify making future changes to your program. A flowchart also lets you see the major and minor loops in your program, allowing you to avoid timing bugs that can occur in real time situations such as video games.

Step 1 involves initializing the score and

V0–	Temporary variable
V1–	Score (00 at start)
V2–	Rocket counter (00 at start)
V3–	Score X (38)
V4–	Score Y (1B)
V5–	UFO X (00 at start)
V6–	UFO Y (08)
V7–	Rocket X (random, 0F to 2E)
V8–	Rocket Y (1A at start)
V9–	Rocket fired flag (00=no, 01=yes)
VA–	Score increment (00 or 01)
VF–	Hit flag (00 or 01)

Table 2: Rocket program variables. VB, VC, VD and VE are not used in this program.

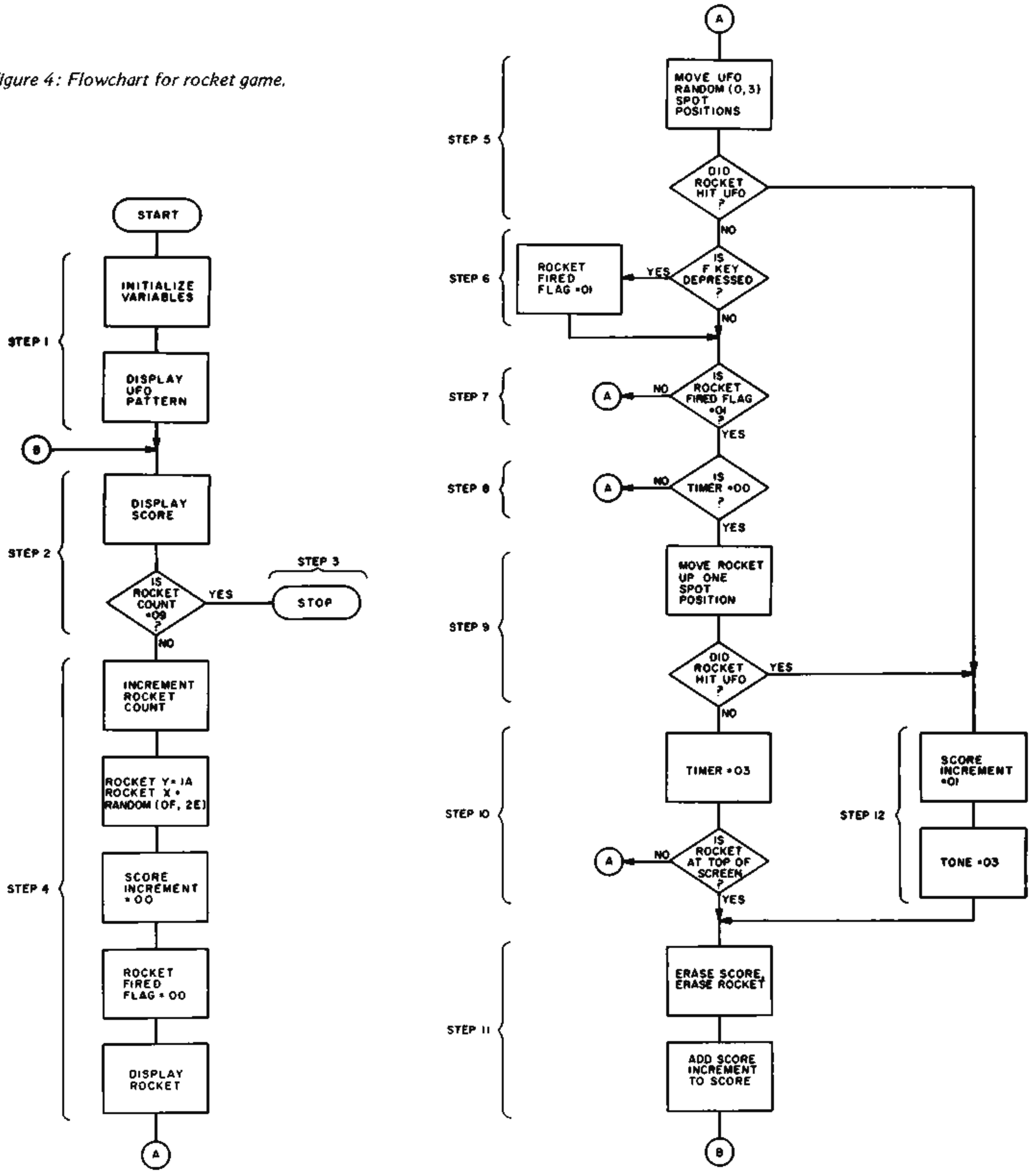
rocket counters, as well as the X and Y coordinates for the target UFO and on screen score digit. The UFO pattern is shown on the screen so that it can subsequently be moved. In step 2 the latest score is shown on the screen, and V2 is checked to see if the game should end because nine rockets have been fired.

Step 4 performs the operations required to show a new rocket at the bottom of the screen. The rocket count is incremented by 1 for each new rocket. The rocket pattern Y coordinate is set to hexadecimal 1A so that the rocket will appear at the bottom of the screen. The rocket X coordinate is set to a random value between hexadecimal 0F and 2E so that it will appear at a random horizontal position without interfering with the score digit. The flag V9 is set to 00 to indicate that the rocket has not yet been fired. The rocket is then shown on the screen and the program proceeds to the loop containing steps 5, 6 and 7.

This loop causes the target UFO to continuously move across the top of the screen while waiting for key F to be pressed. The UFO is randomly moved zero, one, two or three spot positions to the right each time the loop is executed. This gives it a rather fast, randomly varying rate of motion, making it harder to hit. The movement of the UFO merely involves incrementing its X coordinate (V5). When V5 is incremented past the right edge of the display area, wrap around automatically occurs so that the UFO reappears at the left edge of the display area. This wrap around automatically occurs when any X or Y coordinate of any display pattern is incremented or decremented past any edge of the 64 by 32 bit display area.

When key F is pressed to launch the rocket, step 6 causes V9 to be set to 01. Step 7 then causes step 8 to be included in the loop. Step 8 checks the value of the system real time clock (or timer) to see if it has reached 00 yet. When the timer reaches 00 the rocket is moved up one spot position, and the timer is reset to a value of 03 (1/20

Figure 4: Flowchart for rocket game.



```
 Address
(Hexadecimal)  Instruction  Pseudocode          Comments

0200           6100         V1=00               Step 1:  Score
0202           6200         V2=00                        Rocket count
0204           6338         V3=38                        Score X
0206           641B         V4=1B                        Score Y
0208           6500         V5=00                        UFO X
020A           6608         V6=08                        UFO Y
020C           A27E         I=027E                       UFO pattern
020E           D563         SHOW 3MI@V5V6                UFO
0210           226A         DO 026A             Step 2:  Show score
0212           4209         SKIP; V2 NE 09
0214           1214         GO 0214             Step 3:  End loop
0216           7201         V2+01               Step 4:
0218           681A         V8=1A                        Rocket Y
021A           6A00         VA=00
021C           C71F         V7=RND
021E           770F         V7+0F                        Rocket X
0220           6900         V9=00
0222           A278         I=0278                       Rocket pattern
0224           D786         SHOW 6MI@V7V8
0226           A27E         I=027E              Step 5:  UFO pattern
0228           D563         SHOW 3MI@V5V6                Erase UFO
022A           C003         V0=RND
022C           8504         V5=V5+V0                     Set VF
022E           D563         SHOW 3MI@V5V6
0230           3F00         SKIP; VF EQ 00
0232           1262         GO 0262                      Step 12 if hit
0234           600F         V0=0F               Step 6:
0236           E0A1         SKIP; V0 NE KEY
0238           6901         V9=01
023A           3901         SKIP; V9 EQ 01      Step 7:
023C           1226         GO 0226                      Step 5
023E           F007         V0=TIME             Step 8:
0240           3000         SKIP; V0 EQ 00
0242           1226         GO 0226                      Step 5
0244           A278         I=0278              Step 9:  Rocket pattern
0246           D786         SHOW 6MI@V7V8                Erase rocket
0248           78FF         V8+FF
024A           D786         SHOW 6MI@V7V8
024C           3F00         SKIP; VF EQ 00
024E           1262         GO 0262                      Step 12
0250           6003         V0=03               Step 10:
0252           F015         TIME=V0
0254           3800         SKIP; V8 EQ 00
0256           1226         GO 0226                      Step 12
0258           226A         DO 026A             Step 11: Erase score
025A           A278         I=0278                       Rocket pattern
025C           D786         SHOW 6MI@V7V8                Erase rocket
025E           81A4         V1=V1+VA                     Score+VA
0260           1210         GO 0210                      Step 2
0262           6A01         VA=01               Step 12:
0264           6003         V0=03
0266           F018         TONE=V0
0268           1258         GO 0258                      Step 11
026A           A2A0         I=02A0              SSS:     3 byte work area
026C           F133         MI=V1 (3DD)
026E           A2A2         I=02A2                       Least significant digit
0270           F065         V0: V0=MI
0272           F029         I=V0 (LSDP)
0274           D345         SHOW 5MI@V3V4
0276           00EE         RET
0278           2070                             ROCK:
027A           70F8                                      Rocket pattern
027C           D888
027E           7CD6                             UFO:     UFO Pattern
0280           7C00
```

Figure 5: The rocket program code in CHIP-8 hexadecimal interpretive language instructions. The steps specified relate directly to the flowchart given for the game in figure 4.

second). This timer reset value determines the speed at which the rocket moves upward. The larger the timer value, the slower the rocket moves. The loop comprising steps 5 thru 10 keeps both the target UFO and rocket moving on the screen until either they touch each other (a hit), or the rocket reaches the top of the screen without touching the UFO.

If the rocket and UFO touch each other, step 12 is executed. This sets VA to 01 and sounds a short tone so the player knows (s)he scored. Step 11 is then executed to erase the rocket and the current score on screen. Step 11 also increments the score variable (V1) by the 01 in VA. The program then returns to step 2 where the updated score is shown on the screen. If less than nine rockets have been fired, a new rocket is shown on the screen in step 4, and the step 5-6-7 loop is repeated as before.

If the rocket reaches the top of the screen without touching the UFO, step 10 will branch to step 11 causing the score and rocket to be erased. In this case VA will contain 00 (from step 4) so that the score will remain unchanged. Note that the entire program has now been designed. By double-checking the program in flowchart form, you can eliminate almost all program bugs before they occur. An extra hour spent on the flowchart can eliminate many hours of debugging later. All that remains now is to translate the flowchart into an appropriate sequence of CHIP-8 instructions.

Coding and Debugging the Final Program

The final program is shown in figure 5. To translate the flowchart into CHIP-8 instructions, start by listing even numbered hexadecimal memory addresses in the first column, as shown in figure 5. Fill in the third column with an abbreviated description of the function to be performed by each instruction. It is usually most convenient to locate subroutines and pattern byte lists at the end of the program. Labeling the appropriate program addresses with the flowchart step numbers will also prove helpful. The actual hexadecimal codes for the CHIP-8 instructions can then be written in column 2 and entered into the COSMAC VIP memory using the hexadecimal keyboard.

To debug the program, replace the 4209 instruction at memory location 0212 with a 1212 branch instruction. When the program runs, it will stop at location 0212 since the 1212 branch loops on itself. If the UFO and a 0 digit initial score show on the video screen, you know execution was proper up to location 0212. Replace the

1212 branch with the original 4209 instruction and put a similar idle loop branch further down in the program for the next test run. In this way you can identify which program steps are causing a problem. If you need to change any portion of the program, just insert a branch instruction to a patch added at the end. Designing, coding and debugging this simple game program required about eight hours. Actual coding and loading the program into memory required less than an hour of this time.

The sample program was kept simple for ease of understanding. Even in this simplified form it is a challenging game to play. The speeds of the rocket and UFO can be easily adjusted to make scoring more or less difficult. Adding multiple targets and 2 digit scoring is possible. Multiple rocket launch angles or after-launch steering could be incorporated. Exploding UFO patterns could be shown when one is hit.

Conclusions

Hexadecimal interpretive programming provides an easy way to program small computers. This approach requires fewer instructions and is much easier than machine language programming. On the other hand, hexadecimal interpretive programming requires much less hardware overhead cost than do high level languages such as BASIC.

A detailed example was provided to illustrate this interpretive approach for a real time video game. The RCA COSMAC VIP and CHIP-8 language were used in this example, although other hexadecimal interpretive languages are possible, and a similar approach can be used with other microcomputers. The steps required in programming with a language such as CHIP-8 are the same as required when using any language: you must specify the functions required, decide on variable and memory utilization, prepare a flowchart, check the flowchart, do the detailed coding, load the code, and debug to the extent required to get the program running properly. Only the last two steps involve using the hardware. Skipping any of the earlier steps will invariably lead to excessive machine debugging time no matter what language is used.

If you've never tried a language such as CHIP-8, you may be surprised at how easy it is to use. If you have a limited budget you will certainly appreciate the savings in hardware over conventional high level languages. Last but not least, you might even discover that designing your own hexadecimal interpretive language is also fun.■

Program faster, debug easier now

Flowchartrix™ a unique flowchart development tool from Stirling/Bekdorf, saves you time, space, and money, no matter what language you work with. Whether you program professionally or just for fun. The 78F2 Flowchartrix™ helps your thoughts move in logical steps, and lets you retrace logic easily when debugging.

When you use "top-down" programming methods, you can use the 78F2 to lay out your original logic concept blocks. Then by following the plan you lay out in words at the concept stage, you can write a finely detailed flowchart quite smoothly. Then it's easy to write actual code based on the flowchart.

54% more logic cells than other flowchart forms, so you get far more of your program on each page. Each Flowchartrix has a full 77 logic cells, not just 50. This not only saves paper, but also makes your finished flowcharts easier to understand. By seeing up to 27 extra steps of your program on each page, you comprehend program flow more clearly. That's important while writing the flowchart, more important when you write actual code. It's also extremely helpful when you debug, and indispensable when you come back months or years later to modify your original work. 78F2's higher matrix count makes your flowcharts quicker to debug because there are fewer pages to search for errors. Fewer pages also save you money and storage space.

Unique matrix can show your loops AS loops. The Flowchartrix 7 x 11 matrix gives you plenty of room to write loops laid out as sort of a squared circle. This makes loops and subroutines easier to recognize, because their form is readily apparent at a glance. Since they're easier to find, and may even be completed on a single page, they're also simpler to debug or modify.

Every matrix cell has a specific label to help you track branch points. Now it's far easier to follow your program from page to page, point to point. When you write program documentation, having a separate reference point for each cell makes your program much easier to describe clearly.

With Flowchartrix, you don't need a shape template to draw remarkably regular logic symbols. Guides for the most-used logic symbols are right in each matrix cell. They help you draw most standard flowchart symbols entirely free-hand. This saves all the time you'd otherwise spend hunting a shape template and positioning it to draw every symbol. Your train of thought need no longer be interrupted by template tedium. With 78F2, your pencil can fly as fast as you can write, without interruption. When a flash of insight strikes, now you can keep your pen on paper, flowing rapidly from one step to the next without a break.

78F2 is surface-engineered to take both pen and pencil without blotching. The tough 22= base stock is the same brilliant white opaque material used in our 78C1 Combination Coding/CRT Layout forms. Pure enough to use with magnetic ink scanners, heavy enough to withstand vigorous erasure, every Flowchartrix gives you crisp, sharp, characters and symbols. It takes ink without spreading, and accepts soft pencil lead with good contrast.

Order your supply today. Ask your local computer store for Stirling/Bekdorf™ 78F2 Flowchartrix™ To enjoy the world's most advanced program development aids most, use the entire Stirling/Bekdorf system: 78F2 Flowchartrix (for concept planning and flowcharting), 78C1 Combination Coding/CRT Layout forms (for coding and display planning in BASIC, OPUS, and other line-number languages), and 78P4 Print-Out Design Sheets (to design report printouts for easy coding). Our programming tools work together as a complete system to save time and reduce errors during every stage of program development, from concept to completion. Try them for yourself today. If your store is out-of-stock, use the coupon below to get yourself a supply on the way now.

YES! Please rush the programming aids indicated below:

78F2 Flowchartrix™
☐ two 50 sht pads $7 90 + $2 85 shpg
☐ ten 50 sht pads $34 35 + $6 45 shpg

78C1 Coding/CRT Combination
☐ two 50-sht pads $6.35 + $1.95 shpg.
☐ ten 50 sht. pads $26.85 + $3 35 shpg

78P4 Print Out Designer
☐ one 50-sht pad. $7.45 + $3.15 shpg. ☐ five 50-sht pads. $32.10 + $6.75 shpg

Texas residents please add 5 5% sales tax to base price

Enclosed is my check for $ ____________
Charge to: ☐ Master Charge ☐ Visa exp. date ____________
Card # ____________ Signature ____________

Dealer inquiries welcome

Name ____________
Address ____________ Phone ____________
City ____________ State ____________ Zip ____________

We ship UPS so P O Box address must give phone number

Stirling/Bekdorf™

b10 4407 Parkwood ▫ San Antonio, TX 78218 ▫ (512) 824-5643

Circle 352 on inquiry card.