VIPER

September - October 1983
Volume 5, Number 3

Journal of the VIP Hobby Computer Assn.

The VIPER was founded by ARESCO, Inc. in June 1978

****

PIPS for VIPS IV - Part 3

And once again VIPER continues with the serialization of Tom Swan's PIPS for VIPS IV. This issue deals with Page Switching and Graphics in machine language, both very useful techniques for game programs for 1802/1861 computers.

Also in this issue is a letter from Bob Toegel (WA2EGP) on p. 17 and his review of Paul Moews' ELFISH interpretive language for the ELF machines.
There was an advertisement for ELFISH in an earlier VIPER (Vol. 4, I think).

The pages called "appendix" are designed to be placed at the end of your PIPS volume, once you have received all the material.

Cassettes of PIPS IV are now available and the cost for the ten programs on the tape is $5. You may send your check to VIPHCA at the usual address. Each program on the tape is complete, with the appropriate CHIP-8 interpreter, where necessary. The program titles are:

1. Cos Melodeon	6. Move on
2. Holiday tree	7. Box
3. Attack of Micromen	8. Shape maker
4. Line up	9. Graphics #1
5. Sweeper	10. Graphics #2

ELFISH - an Interpretive Development System
A review by Robert J. Toegel

As a Super Elf owner, I've always been jealous of the VIP and the CHIP-8 language. Now Paul Moews has not only come to my rescue but has gone a few steps further. I recently had the pleasure of "playing with" a copy of ELFISH and an editor-assembler written in ELFISH.

ELFISH is a 1K, relocatable (and ROMable) interpretive language that bears a close similarity to CHIP-8, but there are important differences. Many commands are changed and new ones have been added to handle 16-bit variables, 8 and 16-bit logic and arithmetic operations, three display buffers (hex, decimal, bit-pattern), and labels for assembly. A 1K editor-assembler is included with ELFISH. It is written in ELFISH and allows you to replace, insert, delete data, scan memory in both directions, go to a particular address, change addresses, execute the program and assemble the program. The display consists of four lines, each showing the address and two bytes of memory. It uses the normal 1861 capabilities as does ELFISH (64 x 32).

Unfortunately, my experiences with the tape copy of ELFISH were not 100% positive. I couldn't load either copy from the tape. From the sound of the data, it appeared to be a head alignment problem. This is the first time my Radio Shack CTR-41 wouldn't load a tape, but then I haven't checked my own recorder's alignment.

After entering by hand the program from the listing in the ELFISH manual and double checking, I found the assembly not to be complete even when using the sample in the manual, although no error code appeared. Whether I made a mistake entering the program or there was an error in the listing, I have no idea, but all other functions work perfectly. I used the editor-assembler to recheck the program. (Warning: don't try to modify the program with itself---it's a nice way to bomb.)

The manual was a little difficult for me to understand, notably the section on ELFISH. I felt that the author assumes the reader has a good working knowledge of CHIP-8. For me, the one sample program of ELFISH was not sufficient to demonstrate the capabilities of the language. Maybe a different format would be better, especially for those of us who take longer to catch on than most.

One command that is not included is the equivalent to the CHIP-8 command CXKK, the random number routine. Unless you write a machine code random number subroutine, some game programs are not going to be as much fun as they would be in CHIP-8. Another command that would be nice is one that would cycle the Q line to produce a tone similar to the FX18 tone instruction in CHIP-8. These changes would make it easier to change CHIP-8 programs to ELFISH, but then I'm being lazy and picky!

In spite of the problems I had with loading and assembly, plus the "missing" commands, I can still recommend this package to any Super ELF owner. It is easy to use and makes programming in both machine language and ELFISH a breeze. I would gladly spend the $10 for the program and it would be an important and often-used software package for my Super ELF.

PAGE SWITCHING

I have always been fascinated by cartoons. Bugs Bunny and the Road Runner still manage to capture as much of my attention as of any well seasoned six-year old on a Saturday morning. Some Walt Disney (or was it a Woody Woodpecker?) show once ran a feature on how a cartoon is produced. I remember it as if it was yesterday and am still impressed by the process. Thousands of individual still frames are flashed one by one fooling the viewer into seeing the animation of feathered little innocents escaping the menacing jaws of treacherous, but always inept, feline monsters.

With this revelation I would prepare tablet after tablet of "cartoons" which could be animated by rapidly thumbing the pages. You can use the same idea for producing sophisticated computer graphics on a video screen. Here's how.

Even though your COSMAC VIP is a very fast computer, when complicated images are graphically animated, the images must usually be done in parts or sections. For example, the CHIP-8 instruction, DXYN, is only capable of displaying 8 x 15 bit blocks at a time. To construct a larger figure, many display instructions must be combined.

Most of you have probably written programs which first display a figure, maybe just a simple single dot ball, then erase the same figure, adjust its display coordinates and redisplay the figure in a new position. Rapidly repeating the process causes the figure to appear to move.

Extending the process to complex patterns is unfortunately disappointing. Instead of smooth motion, each segment of the larger figure appears to move disjointedly. Though computer operations may sometimes appear to occur simultaneously, the human eye is often capable of detecting even the slightest of timing differences during the construction of a complex shape. The result is a jerky display that may be difficult for the eye to follow.

The same is true when displaying many different figures on the screen. Rather than moving all at once, each figure may appear to change position sequentially. Also an annoying flashing or rolling effect may be seen.

These problems may be eliminated by using page switching, a technique that only permits the eye to see a completely finished page at one time. Just as with cartoons, the entire screen appears animated and no flicker of the individual objects is seen.

There are tradeoffs (aren't there always?). Speed is not a problem, the page switching mechanics taking negligible execution time. But you now must keep two separate and distinct display refresh memory areas. For the four page CHIP-8 programs that follow, this means that eight precious memory pages, half of a 4K VIP must be dedicated to the display.

The following modifications may be made to a copy of Hi-res CHIP-8 such as presented in a previous issue of VIPER or such as provided with the COS-MELODEON program in this book. Normally, the memory area from $0800 to $0BFF will be display page #1 and $0C00 to $0FFF will be display page #2. Programs may occupy the space from locations $0500 to $07FF, admittedly not much room. By installing an additional 4K board on your VIP, however, both display pages may be moved to higher memory locations and CHIP-8 programs placed within the addresses from $0500 to $0FAF. More on this later. (Locations $0FB0 to $0FFF are still reserved for use by the VIP operating system and cannot be used by programs <u>except</u> as intermediate storage locations.)

The modifications presented here change certain functions in the ERASE SCREEN and the DXYN instruction subroutines. Each of these subs is altered to operate only on the display pages <u>not</u> being shown on the video screen. Therefore when you execute an 0200 ERASE command, nothing will appear to happen. It is the invisible display which is erased. When a DXYN instruction is performed, the display bits are inserted into the off screen page while the viewer is still seeing the other one. The page on display is always "static" that is it never changes before your eyes.

Following the routines in your program that construct all the needed display information, calling the machine language subroutine (MLS) at location 0216 (with the CHIP-8 instruction 0216) will bring the off screen page into view. The viewer sees a completed page of graphics with the mechanics of constructing those graphics occurring invisibly.

A couple of things need careful programming attention. First, the PROTECT sub of the interpreter will no longer function. PROTECT was used to selectively erase 1,2,3, or 4 pages of the display by setting V0 equal to the number of pages less than 5, calling 0216 PROTECT and then 0200 ERASE. The PAGE SWITCHING sub replaces PROTECT.

The DXYN instruction will now not automatically cut off a figure at the bottom of the display screen. When viewing display page #2, a figure displayed too low on the screen will appear in part in the top of the page being viewed. Worse, when displaying page #1, a figure may be displayed past page #2. Due to the mimicking of memory locations not containing RAM, such a display could overwrite part of the CHIP-8 interpreter. It is the programmer's responsibility not to display figures past the bottom edge of the display screen. There is no similar problem or restriction with the left or right edges.

Finally, the display pages are not automatically erased by the interpreter on switching to run. To clear both display pages and begin execution of a program, the following sequence should be programmed:

```
0500 BEGIN: 0200   ERASE -- Clear off screen page
  02        0216   FLOP  -- Switch display pages
  04        0200   ERASE -- Clear off screen page
```

These three instructions should be the first in any program using page switching animation. Both display pages will be erased and the viewer will be seeing a blank screen following their execution.

When using higher memory for display pages, a machine language subroutine must be written and called once at the beginning of the program. The purpose of this MLS is to set the display page pointer RB.1 (that refers to one of the 1802's machine registers) to the new display page pair. There is room for this routine at location 021B in the interpreter, or it may be incorporated into your program somewhere else. For example, the following will work.

```
021B  F8 NN     NEWPG: LDI  DISP.1 ;NN=New display page
  1D  BB               PHI  RB     ;Put in RB.1
  1E  D4               SEP  R4     ;Return to CHIP-8 program
```

You must insert the address of either display page pair in place of the "NN" in the above program. If you wish to use \$1000 - \$13FF and \$1400 - \$17FF for the display for example, you would replace NN above with either \$10 or \$14, it doesn't matter which. Display page pairs may be located anywhere in memory, but must be either in the top 2K or the bottom 2K of any 4K block. Therefore, \$1200 - \$15FF and \$1600 - \$19FF are <u>not</u> valid display page possibilities, but \$2800 - \$2BFF and \$2C00 - \$2FFF are allowed.

When using off card RAM, the following CHIP-8 sequence should be used at the start of a program:

```
0500  BEGIN: 021B   MLS   -- Call MLS to change RB.1
  02         0200   ERASE -- Erase one display page
  04         0216   FLOP  -- Switch display pages
  06         0200   ERASE -- Erase the other display page
```

This assumes that you had previously entered the machine language sub NEWPG as just described and inserted a valid display page value in place of "NN."

* * *

Using page switching is not complicated if you remember two things.

1) Erasing and displaying figures will not be immediately seen following execution of 0200 and DXYN instructions.
2) Perform an 0216 instruction (a call to the FLOP sub) to see the effects of previously executed erasing and displays.

One feature of the page switching modifications is that the flip flop from page #1 to page #2 is completely automatic. The program never needs to know which page is being displayed -- it all happens by itself. All CHIP-8 conventions are the same as before.

Here are the additions to Hi-res CHIP-8.

PAGE SWITCHER FOR HI-RES CHIP-8

```
004E D4             SEP  R4    ;Cancel pre-erase command

0082 30 DC          BR   $DC   ;New branch to patch in DXYN

00DA 30 B3          BR   $B3   ;Cancel end of page exit in DXYN

00DC AC             PLO  RC    ;See Jan. '80 VIPER
  DD 93             GHI  R3    ;  for comments
  DE 7E             SHLC
  DF BC             PHI  RC    ;This is part of the
00E0 8C             GLO  RC    ;  array mutiplication
  E1 FE             SHL        ;  to position RC on the
  E2 F1             OR         ;  display page for the
  E3 AC             PLO  RC    ;  DXYN instruction.  It is
  E4 9C             GHI  RC    ;  the same routine moved up
  E5 7E             SHLC       ;  a few bytes
  E6 52             STR  R2
  E7 9B             GHI  RB    ;Flop RB.1 to the off
  E8 FB 04          XRI  $04   ;  screen display buffer
  EA 30 84          BR   $84   ;Return from patch

00EC - Available
00ED - Available

0200 9B             GHI  RB    ;Find last page of
  01 FB 07          XRI  $07   ;  off screen buffer
```

SWITCH DISPLAY PAGE SUB

```
0216 9B       FLOP: GHI  RB    ;Switch display pages
  17 FB 04          XRI  $04   ;The XRI $04 will cause
  19 BB             PHI  RB    ;  RB.1 to flop back and
  1A D4             SEP  R4    ;  forth between pages
```

GRAPHICS IN MACHINE LANGUAGE

The routines that follow form the backbone of a machine language graphics package for VIP owners. Because most 1802 systems use the 1861 video display chip, the subroutines have been written to work with that particular hardware circuitry. However, no VIP operating system routines are called by the graphics subs so they should run without modification on Elfs and homebrew set ups. I can't <u>promise</u> you they will run on other systems, but I <u>believe</u> they will.

Why program in machine language (or assembly language if you have an assembler program)? The best reason I can think of is speed. No programming language can execute instructions as fast as individual machine codes. It's like running a stripped down race car. No extras. Nothing fancy. Just the bare necessities. Naturally, speed is attractive for computer graphics. You can always put on the brakes if you want.

Another reason for developing programs in machine language is to satisfy that bottomless curiosity many of us feel about computers. What <u>really</u> is going on in the binary depths of a processor's wizardry? If you're like me, you <u>need</u> to answer that question. The most satisfying programming to me comes out of constructing a program from the ground up. When done, you have a product of which you intimately know every little nut and bolt and have personally tightened each one down to your own standards of satisfaction.

* * *

Here are some of the VIP graphics capabilities you will have under machine language control:

1) Plot a point anywhere on the display
2) Clear the display page
3) Connect any two points with a straight line
4) Display a shape from a set of points
5) Display a set of points (plot)
6) Change display resolutions
7) FLOP display for animations (4K minimum required)

Because the graphics subs sit in less than two memory pages, even the smallest VIP system with 2K of memory may be programmed to experiment with hi-resolution graphics. The system is designed to take advantage of advanced graphics techniques which you may find published in magazines and textbooks. If you have only been using one page CHIP-8 up to now, you will be amazed (I sure was!) at what that little VIP can really do. And we've only scratched the surface.

Before going into how to use the subroutine package, here is a brief explanation of your computer's graphics capabilities thanks to a little chip of electronics, the RCA 1861.

WHAT IS AN INTERRUPT?

The graphics display chip, the RCA 1861, works by regularly outputting memory bits one by one to show up on a properly connected TV screen with brightened dots for bits equal to one and darkened dots for zero bits. In order for a program to display graphics, it may insert binary values of the proper combinations of ones and zeros into memory. Pictures may be drawn, shaped, animated and erased by selectively setting and resetting bits in a memory area known as the "display refresh page." (When "page" is used here as a graphics term, it refers to the entire display refresh even though that memory area may be composed of up to four 256 byte memory pages.)

A key word in the definition of the 1861 chip function is "regular." Once every 1/60 seconds the entire display refresh page is sent to the video screen. Though the display may appear stationary, it is really being "refreshed" or updated 60 times a second. If this were not done at these regular intervals, the phosphors in the TV tube (known jargonistically as a "CRT" for Cathode Ray Tube) could not retain the display information except for short moments until fading away.

This timing matches that of most all video sets and is at a rate too fast for the eye to see intervals between the picture frames. To take photos of a television screen, by the way, the camera exposure must last for at least 1/60 of a second but may be set to multiples of 1/60 such as 1/30, 1/15, etc. to be sure of capturing at least one full frame. I throw this in as you may wish to someday photograph your computer display and these are the only settings guaranteed to produce consistent results. Animations may be captured on still photos by calculating the number of frames to expose. Movies may be made that use computer graphics -- in fact the briefing room sequence of the movie STAR WARS was created on a video display and filmed a frame at a time. A similar technique for computer animations will be presented a little later.

It is the responsibility of the 1861 video chip to request the information it needs to refresh the display. It is the program's responsibility to respond to that request and insure that the proper memory bytes will be sent to the video circuits.

The VIP's 1802 microprocessor receives the video chip's timed information requests on a line called the "interrupt line." Inside the processor is a flag capable of being set to 1 or 0. This flag is the Interrupt Enable Flag (IE) and when set to zero, interrupt requests are ignored. When IE = 1, however, the computer will go into a very special sequence every time an interrupt signal is received. This process, also in "jargonese," is called "servicing an interrupt request."

Because interrupts may occur at any time during a program run, all appropriate registers must be saved so that the running of the interrupt routine will not disturb the calculations the program may be making. You might think of the process simply as an <u>external</u> subroutine call.

Another line in the 1802 permits a device to directly access memory with the processor functioning only as a blind director via register R0. Bytes are sent directly from memory at locations addressed by R0 <u>without</u> those bytes entering the D register. This action, called DMA for <u>D</u>irect <u>M</u>emory <u>A</u>ccess, is an extremely fast means for input and output of memory blocks.

Interrupts and DMA action are basic processes for VIP graphics. The 1861 video chip sends 1024 requests for direct memory access at specific intervals following its request for an interrupt. These DMA "bursts," which come in groups of 8, are what cause the display refresh bytes to be sent to the video screen. The purpose, then, of the interrupt routine is to set up and control register R0 for the byte transfers and to sense via the flagline EF1 when the output cycle is nearing completion. All this takes but a moment. (Note: The EF1 signal is ignored for Hi-res displays.)

Various amounts of memory may be sent via DMA to the video set by manipulating the pointer during the interrupt. There is no reason why even separated 8-byte memory locations could not be used except that it is more natural to use a sequential portion of memory for the display refresh.* The 1861 video chip always outputs a resolution of 128 lines of 8 bytes each (64 bits horizontally). However, by repeating each eight byte segment a number of times, the apparent resolution of the display may be altered vertically. For single memory page displays, the normal CHIP-8 format, each eight byte segment in those 256 bytes is repeated four times resulting in a display resolution of 32 vertical bits (128/4) by 64 horizontal bits. The horizontal resolution cannot be changed under program control.

DMA action automatically increments R0 following each byte transfer. If the interrupt routine merely sets up that pointer before the first DMA request, a full 1024 sequential bytes will be sent to the video circuits. This results in the highest possible resolution with the simplest of interrupt routines.

When writing such an interrupt routine, there are a few things that must be accomplished. First, sometime before the

---

*NOTE: Though I have not tried this, organizing the display refresh in a non-sequential manner could improve routines such as CLEAR display. Rather than erasing bottoms up, for example, the effect would be a visually neater venetian blind result. However, this would also complicate the subroutines in this book that plot and draw lines on the screen.

first DMA burst, a three cycle instruction must be executed to compensate for the single interrupt cycle. A C4 NOP instruction at the beginning of the interrupt routine will accomplish this requirement. Then, 29 cycles later (including the three cycle NOP), the first burst of eight DMA requests will occur and will continue to occur every six cycles from that point 128 times total. It is important to realize that the DMA bursts occur between the execution of instruction codes.

The two interrupt routines supplied here may be used for resolutions of 64 x 64 and 128 x 64 and will work equally well with the graphics subs in this chapter. Register RB.1 is used to hold the display page address (the address of the first memory page of the refresh buffer) though this was arbitrarily done to mirror the format of the CHIP-8 interpreter. These interrupts may not be used in CHIP-8 games, however. (If you are using a VIP you may set R1 to the address of the ROM interrupt at $8146 for a resolution of 32 x 64 which will also work with the following subs except LINE. In that case, however, R9 and R8 may not be used by your program as these are changed by the ROM interrupt routine.)

It is very important that the main body of the program contains no other of the three cycle type of instructions. This includes all of the long branches and long skips of the hex code form $CN. Their use may (but not always) hold up an interrupt request and momentarily desyncronize the critical 1/60 second timing. If this happens, the display will jitter occasionally though nothing will happen to damage the computer or the program. If you don't care about the jitter, you may use three cycle instructions. (That's one of the great things about computers. You can make all the programming errors you want without physically hurting the machine.)

With all resolutions less than the full 1024 bytes, DMA bursts are received while the interrupt routine is running. Your program will be delayed for the time this takes to happen. Using the Four Page High-Resolution format, however, sets up the DMA pointer R0 then returns before the first DMA bursts occur. Therefore, the output of the display refresh to the video screen will occur in between instructions of the main program, and not during the interrupt routine.

This interlacing of the two functions -- your program and the display -- causes a great increase in speed when using the highest resolution. Both are being executed in effect simultaneously. However, the output may also occur during portions of a program that are responsible for inserting bytes into the display refresh. If this should happen, the display may be caught in the process of being animated and appear somewhat less than smooth. In such cases, page switching, will eliminate the problem and as with CHIP-8 page switching, the following subs were designed to take advantage of this important graphics technique.

Reader I/O

Dear Ray,

Thank goodness for another source of programs and information on the 1802. In fact, you even convinced me to get a VIP. (Now I have three!) My Super Elf is getting worried. Keep up the good work!

Another reason for writing is that I'm looking for information from any Super Elf owners who have had trouble with the parallel port on the expansion board. From what I can determine there is a software-hardware catch-22 which prevents the port from being used. I have a temporary fix but I would like to find a more permanent solution (hardware mod). My temporary fix works in most cases but makes it impossible to break Super Basic from the ASCII keyboard. Any help or other sufferers?

Very late vote: I would also like to see Elf articles. The 1802 needs support against the more advanced "whoopee-bang number-cruncher" micros. I think users of VIPs, Elfs, Super Elfs and homebrewed microprocessors can benefit from any 1802 based program. Although this is a VIP user newsletter, we can all learn from articles about other 1802 systems. Maybe we can give the appliance operators of those kilobuck machines a lesson in real hobby computing without the "My micro is better than your micro!" attitude that I've run into from non-1802 users. Its up to us to stick together. (I'm an 1802 snob.)

Also I would like thank Paul Piescik for the help he gave me with Starfight. Paul you were right, my local electronic store sold me a cable with an intermittant.

Keep up the good work everybody and don't let anybody 68 you!

Robert J. Voegel

THE SUBROUTINES

There are six subroutines in the graphics set plus two interrupt routines. An additional subroutine permits you to change display resolutions under software control without having to first turn off the video. Also included are the mechanics of the Standard Call and Return Technique (SCRT) and the necessary initialization of registers, etc. A controller for page switching is also included, but some changes must be made first before using it. (More on this later.)

All of the subroutines may be placed in Read Only Memory (ROM) and all are page relocatable except for subroutine calls among them (nesting). In other words, a sub with a starting address of $0130 will function equally well if it is relocated at $1030 or $FF30 except that calls to that sub will have to be changed to the new address.

The SCRT, described in the RCA 1802 Programming Manual, is a versatile means for using subroutines. Some time is lost due to the mechanics of the technique, but this is made up by a flexibility not possible with other methods. To use the technique and to call a subroutine, the program simply executes a D4 SEP R4 instruction followed by the two byte address of the subroutine. The three bytes D4 01 30, for example would cause the subroutine at $0130 to begin running. In the following listings "CALL" is equivalent to "SEP R4."

Return addresses are saved sequentially on a stack addressed by R2. Each subroutine must end with the instruction D5 SEP R5 causing a return to the memory location just after the most recent CALL. In the listings, "RETN" is equivalent to "SEP R5." (Do not confuse RETN for RET which is a different instruction!)

Some advantages of the technique are: 1) The ability to call subroutines anywhere in memory from anywhere else; 2) X is set equal to 2 by issuing either a CALL or a RETN and careful programming may take advantage of this; 3) The stack pointer R2 is always addressing a free memory location. You do not have to decrement R2 before pushing a byte onto the stack but you must be sure to have R2 free before CALLing another sub; 4) R3 is the program counter for all routines and subroutines; 5) Unlimited nesting may be programmed subject only to the size of the stack.

Some disadvantages are: 1) A small loss in speed -- much more noticeable in nested loops that call subs; 2) Registers R4, R5 and R6 are not available for use by your program. (These may be pushed onto the stack temporarily provided there are no CALLS or RETNS issued before restoring their original values.); 3) Parameters may not be passed to subs in D or on the stack. Only registers and memory locations may be used to "give" a subroutine a value or for a routine to "return" a value to the calling program.

After you use this technique for subroutines, I believe you will become hooked on its power. To assure compatibility with future 1802 systems, the standard RCA implementation has been used and I suggest you refer to RCA's 1802 manual for further comments on the design and use of the SCRT.

* * *

On the tape supplied with this book you will find two graphics subs packages. If you do not have the tape, you may enter the routines exactly as shown here duplicating Graphics Subs Package #1. Package #2 is the same set but with page switching enabled. For the modifications you must make to enable page switching, see the description of the FLOP sub. For relocating, be careful not to cross page boundaries. Some of the subs require lookup tables that must exist on the same memory page.

Each of these graphics subs will be explained in full and I suggest you read this section before attempting to use your new machine language graphics capabilities. After these explanations, a reference is provided giving the requirements of each routine plus its CALLing address.

The listing is half machine/half assembly and differs from the format used in earlier books. I think this will be more readable, but there are a few idiosyncracies that need to be explained.

As always, a number preceded by "$" means that the value is expressed in hexadecimal digits. Labels are used to refer to memory addresses so that LDI STACK.1 would mean to load the D register with the high byte of the value called "STACK." This value would have been previously defined as the base address of the stack memory area. Unlike a real assembly listing, some labels such as STACK are used more for reference of obvious intent. All branch destinations have been properly labeled, however. Very few VIP'ers have access to assemblers and I assume most of you will either be hand loading or simply using the tape.

When a jump is made to a local address -- meaning an address within the same routine -- a special local labeling system has been used. All local jumps are to addresses labeled with a number between 1 and 9 and the letter "H" meaning Here. For example "1H" and "3H" could be used as local labels specifying where branches are to go. Branch instructions refer to these numbers adding either the letter "B" or "F" meaning Back or Forward. A jump always procedes to the nearest "H" number of the same value used by the branch instruction. Therefore there may be several "1H's" or "2H's" in the same routine without a conflict. Here is an example which should help make this a little clearer.

```
0100        BR    1F   ;Branch Forward to $0103 to start
  02  2H:   INC   RF   ;Count = count + 1
  03  1H:   LDA   RE   ;Test byte @ M(R(E))
  04        XRI   $FF  ;Check for end of table
  06        BNZ   2B   ;If  not end, branch Back to $0102
  08        GLO   RF   ;Test count
  09        BZ    1F   ;If = 0, branch Forward to $010C
  0B        RETN       ;Return normal
  0C  1H:   SEQ        ;Signal empty
  0D        RETN       ;Return with Q = 1
```

This routine won't do anything if you load it into your computer, but it might be used in a larger program to find the size of a table ending with an $FF byte. Don't worry about comprehending its purpose. Notice, however, that there are two "1H:" labels. The branches always procede in the indicated direction to the nearest label with the same number. There is no conflict.

When the program is assembled into hex codes, the proper branch addresses will be inserted. The 1H's, 2B's, etc. don't actually go into the computer -- they are only a convenience to make the assembly listing easier to read. Actual hex values are always marked with a dollar sign.

If you are new to assembly programming, you will probably find a lot of this unintelligible. That's perfectly normal, and you should not let that discourage you. Believe me, I <u>know</u> how that feels. Just stay with it and you'll find clear spaces in the clouds before long. The sample graphics programs will help give you a footing with machine language and you may want to experiment with them first if this is all new to you. <u>Using</u> the graphics subs may be considerably easier at first than understanding <u>how</u> they work. Fiddling with other peoples programs is a good way to discover on your own what certain instructions do. I hope you'll get into your work duds, crawl under all this and loosen a few screws to see what drops out. There's no better way to learn.

One caution: don't load the graphics subs into your computer and flip the run switch. Alone, until you tell them what to do (Like you have to tell the CHIP-8 interpreter what instructions to perform) they won't do anything at all. After the program listing you will find many programs and suggestions for hours of graphics fun in 1802 machine language.

WHAT'S THE POINT?

The most important subroutine of the lot is POINT beginning at $0100. Similar to CHIP-8's DXYN instruction, POINT allows a program to turn on individual bits in the display memory refresh according to their X,Y coordinates.

Each of the 1024 points on the display may be referenced by a unique pair of coordinates. These are sometimes referred to as "Cartesian coordinates" and the set of all points together as "Cartesian space," though mathematically speaking, the display exists only in one quadrant because there are no negative values for X or Y. The subroutine's job is to take a pair of X,Y values and turn them into the address of a byte and the location of a bit in that byte. That single bit may then be turned on or off resulting in a white or a dark point on the display screen.

On the display, it <u>looks</u> as though there are rows (horizontal) and columns (vertical) of dots. Actually these memory bits exist in bytes at sequential addresses -- it is the programmer who decides to view those memory bytes as if they existed in a matrix with rows and columns matching that of the display.

Because there are exactly eight bytes in a row, giving the display its 64 bits horizontal resolution, the first byte in any row may be found at position 8*N where N is the Y or vertical coordinate. Note that Y = 0 is the position of the first vertical row starting at the top of the display. Therefore, multiplying Y by eight and adding the result to the address of the first display byte locates the proper row in the matrix.

The column position is more difficult to calculate. First we need to find which of the eight bytes in the row corresponds to the X coordinate, move a pointer to that byte, then find which bit in the byte exactly corresponds to X. Because there are 64 horizontal bits in eight bytes, dividing X by eight locates the proper byte again with 0 being the first byte position. The remainder of that division can only be between 0 and 7 ("modulus 8" meaning the remainder following division by eight). This remainder is used to locate the individual bit in the byte and ends the display process for one point.

Sometimes "talking" a formula out like this helps to understand it more than reading a cold undocumented equation. Any matrix may be treated the same way. It is convenient, however, that the multiplication of Y and the division of X be by powers of two. Shifting may then take the place of more time consuming math subs. Don't let anyone tell you that complex arrays of various dimensions aren't possible in machine language, however. It ain't true!

The memory matrix looks like this for all 1024 bytes in the display refresh.

X coordinates

Y							
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
_	_	_	_	_	_	_	_
1016	1017	1018	1019	1020	1021	1022	1023

This diagram illustrates a very important point in working with addressing schemes, matrices, lists etc. Byte number one is located in <u>position</u> number zero. This convention helps keep addressing simple. The address of the first byte plus zero (the first X coordinate for example) of course leaves that address unchanged.

Once the proper bit position is discovered, it is turned on by POINT. This is accomplished by the OR instruction at $012D. Unlike CHIP-8 displays, redisplaying a bit over another will <u>not</u> turn that bit off. Only by clearing the entire screen may figures be erased. You may change location $012D to $F3 XOR giving the POINT sub the same selective erasing capability as CHIP-8. Depending on the animation, either XOR or OR will result in a nicer looking display. Try both to see which one suits

APPENDIX

NOTES ON THE CHIP-8 INTERPRETER

Both COS-MELODEON and A BINARY TREE FOR THE HOLIDAYS use the following hi-resolution CHIP-8 interpreter which is a modified version of the original low resolution CHIP-8 interpreter developed by Joe Weisbecker. Though compatible with most CHIP-8 games, some of the original CHIP-8 instructions will not work and have been replaced with new commands. Do not use the old commands in CHIP-8 programs. They are likely to cause wild and hairy program bugs.

Old Instructions		New Instructions
00E0	ERASE SCREEN	0200
BMMM	GOTO 0MMM+V0	NONE

All other CHIP-8 instructions are valid. Notice that there is no replacement for the BMMM command, a little (if ever) used instruction anyway. In addition to these changes, some new capabilities have been added. These are:

New Instructions

- BXA7 - Output VX to the VIP I/O port (For X≠F!)
- BXA9 - Input VX from VIP I/O port
- BXAB - Wait for keypress then input VX from I/O port
- BXB1 - Strobe (sets and resets Q)
- 0216 - Protect. V0 = # pages to erase with 0200 instruction. V0 must be either 1,2,3 or 4. Other values may cause a program to crash.

Besides all this, the program ATTACK OF THE MICROMEN adds page switching capability, removing the 0216 PROTECT instruction. These changes were detailed in that chapter of this book. If you are loading the interpreter by hand, the page switching modifications would go in after you load the following interpreter.

The CHIP-8 memory map on page 36 of the VIP Instruction Manual is the same in format for the modified interpreter. However, the interpreter now extends from $0000 to $029F; the stack work area and variables are on page 2 for <u>all</u> size VIPs; and $0300 to $04FF contains the ASCII character set. Programs may begin at $0500 and extend up to the display page, the last four memory pages in your computer ($0C00 to $0FFF for a 4K system, $0800 to $0BFF for 3K)

The interpreter is presented only in its hexadecimal form rather than a documented listing as all of the following code has been disassembled in one place or another. For those of you who are interested in the anatomy of the interpreter, here is a reference list where explanations of the code may be found.

```
Original CHIP-8 disassembly -- VIPER Vol 1 Issue 2 Aug. '78
Hi-res Modifications        -- VIPER Vol 2 Issue 6 Jan. '80
Messager                    -- PIPS FOR VIPS Vol 1
Adding I/O to Hi-res CHIP-8 -- VIPER Vol 2 Issue 9 Apr. '80
```

If you have less than 4K of memory in your computer, the following location in the interpreter must be changed to tell the DXYN sub where the end of the display is. If you only have 2K, you may not use the interpreter in its present form.*

```
00DB  0C/10 -- 2K=08*/3K=0C/4K=10
```

---

*If you enter 1300 at $00FE, you may use the interpreter in a 2K system with your programs starting at $0300. However, you may not use the Messager capability in that case.

Enter the appropriate value for your system. If you add memory on board your VIP, remember to change this value to the correct one or strange things will begin to go bump in your memory bits!

USING MESSAGER

MESSAGER is a program which I wrote for the first PIPS FOR VIPS. It is the same program included with the interpreter here. If you have PIPS Volume 1, you may want to refer to page 100-104 for the listing and instructions on how to use MESSAGER. Do not, however, make the suggested modification at 0211 as detailed on page 102 of PIPS I.

MESSAGER may be easily used in your own hi-resolution programs. Words, numbers -- any characters -- may be printed on the video screen by simply keeping those characters in their ASCII hex code form. ASCII means American Standard Code for Information Interchange and gives each of 128 characters a unique binary code. The 1802 assembly language reference card that came with your VIP contains an ASCII chart. All of the characters in that chart except for three are included in the ASCII character set below following the Hi-res CHIP-8 hexadecimal listing. These three characters could not be constructed in the 3x8 area used by MESSAGER for a single character.

Right Character	ASCII Code	My Character
#	$23	≠
%	$25	pc
&	$26	÷

The & ampersand was the real devil so I decided on the divide symbol. If you are not familiar with ASCII, you may wonder what all the two and three letter codes are for the ASCII hex codes 00 - 1F. These are called "control characters" because they are usually entered at the keyboard (not the hexpad) by pressing the control key and a letter key at the same time. They are usually not printed on the display and so these codes are blank in the following character set. The 95 printable codes are $20 - $7E. (Remember, the dollar sign means "hexadecimal")

A series of characters (even a series of zero or one character) is called a "string" because that series is often treated as a unit. A string of ASCII characters may form a word or a group of words or a sentence. There are enough strings in this book to go fly a kite, which you may consider telling me to do after reading that comment.

To print an ASCII string in CHIP-8 programs, simply enter the ASCII codes for each letter into an area of memory not to be used elsewhere by your program. All ASCII strings must be terminated with a null character which is jargonette for "ended with a zero." For example, the following string spells out the words "Oh go fly a kite":

```
0600  4F 68 20 67 6F 20 66 6C 79 20 61 20 6B 69 74 65 00
```

To cause the above string to be printed on the display, the following CHIP-8 program may be used. (Don't forget the 00 null character at the end of the string!)

```
0500  6C00  ;VC=00 -- VC is always VX coordinate
  02  6D30  ;VD=30 -- VD is always VY coordinate
  04  A600  ;SET I -- Point "I" to the ASCII string
  06  0244  ;MESGR -- CALL MESSAGER
  08  DCD8  ;PRINT -- Used by MESSAGER program
  0A  150A  ;STOP
```

Notice that you have the full upper and lower case letter set, 16 characters per line and up to 21 lines possible though 16 lines are far more readable. VC and VD are always used as the X and Y coordinates and the 0244, DCD8 sequence must always be programmed in that order with no intervening instructions.

Of course the above program needs the Hi-res CHIP-8 interpreter in order to work.

HI-RES CHIP-8 INTERPRETER with INPUT/OUTPUT and MESSAGER

```
0000  91 BB F8 02 B2 B6 F8 CF A2 F8 02 B1 F8 25 A1 90
  10  B4 F8 1B A4 F8 01 B5 F8 FA A5 D4 96 B7 E2 94 BC
  20  45 AF F6 F6 F6 F6 32 44 F9 50 AC 8F FA 0F F9 F0
  30  A6 05 F6 F6 F6 F6 F9 60 A7 4C B3 8C FC 0F AC 0C
  40  A3 D3 30 1B 8F FA 0F B3 45 30 40 22 69 12 C0 02
  50  00 01 01 01 01 01 01 01 01 01 01 01 01 00 01 01
  60  00 7C 75 83 8B 95 B4 B7 BC 91 EB A4 D9 70 99 05
  70  06 FA 07 BE 06 FA 3F F6 F6 F6 22 52 07 FA 7F E2
  80  FE FE 30 E0 74 BC 45 FA 0F AD A7 F8 D0 A6 93 AF
  90  87 32 F3 27 4A BD 9E AE 8E 32 A4 9D F6 BD 8F 76
  A0  AF 2E 30 98 9D 56 16 8F 56 16 30 8E 00 EC F8 D0
  B0  A6 93 A7 8D 32 FC 06 F2 2D 32 BE F8 01 A7 46 F3
  C0  5C 02 FB 07 32 D1 1C 06 F2 32 CE F8 01 A7 06 F3
  D0  73 16 8C FC 08 AC 9C 7C 00 BC FB 10 3A B3 30 FC
  E0  AC 93 7E BC 8C FE F1 AC 9C 7E 52 9B 30 84 42 B5
  F0  42 A5 D4 8D A7 87 32 AC 2A 27 30 F5 F8 FF A6 87
0100  56 12 D4 00 00 45 A3 98 56 D4 F8 81 BC F8 95 AC
  10  22 DC 12 56 D4 06 B8 D4 06 A8 D4 64 0A 01 E6 8A
  20  F4 AA 3B 28 9A FC 01 BA D4 F8 81 BA 06 FA 0F AA
  30  0A AA D4 E6 06 BF 93 BE F8 1B AE 2A 1A F8 00 5A
  40  0E F5 3B 4B 56 0A FC 01 5A 30 40 4E F6 3B 3C 9F
  50  56 2A 2A D4 00 22 86 52 F8 F0 A7 07 5A 87 F3 17
  60  1A 3A 5B 12 D4 22 86 52 F8 F0 A7 0A 57 87 F3 17
```

```
0170  1A 3A 6B 12 D4 15 85 22 73 95 52 25 45 A5 86 FA
  80  0F B5 D4 45 E6 F3 3A 82 15 15 D4 45 E6 F3 3A 88
  90  D4 45 07 30 8C 45 07 30 84 E6 62 26 45 A3 36 88
  A0  D4 3E 88 D4 E6 45 A3 63 D4 6B D4 3F AB 6B 37 AE
  B0  D4 7B 7A D4 45 56 D4 45 E6 F4 56 D4 45 FA 0F 3A
  C0  C4 07 56 D4 AF 22 F8 D3 73 8F F9 F0 52 E6 07 D2
  D0  56 F8 FF A6 F8 00 7E 56 D4 19 89 AE 93 BE 99 EE
  E0  F4 56 76 E6 F4 B9 56 45 F2 56 D4 45 AA 86 FA 0F
  F0  BA D4 F8 00 B4 F8 1B A4 12 D4 02 3F 00 4B 15 00
0200  9B FC 03 BF F8 FF AF F8 04 AE EF 94 73 8F 3A 0B
  10  2E 8E 3A 0B 5F D4 F8 F0 A6 F8 02 BF F8 08 AF 06
  20  5F D4 7A 72 70 C4 22 78 22 52 19 E2 9B B0 F8 00
  30  A0 98 32 38 AB 2B 8B B8 88 32 22 7B 28 30 23 9B
  40  FF 03 BB D4 22 15 93 B4 F8 4C A4 D4 F8 FC A6 A7
  50  17 06 BF 9A 73 8A 52 4A 32 95 FE FE AC F8 03 7C
  60  00 BC 1C 1C 1C 94 BA F8 EF AA F8 04 AF 2A 0C FE
  70  FE FE FE 5A 2A 0C FA F0 5A 2C 2F 8F 3A 6D B3 F8
  80  70 A3 D3 E2 25 72 AA F0 BA 1A F8 FC A6 A7 17 06
  90  FC 04 56 30 53 15 9F 56 12 F8 01 B3 F8 F2 A3 D3
```

ASCII CHARACTER SET

```
0300 - 037F - Not important - control characters 00-1F
0380  00 00 00 00 22 20 20 00 55 00 00 00 17 27 40 00
  90  27 43 70 00 66 44 32 30 20 70 20 00 24 00 00 00
  A0  12 22 21 00 42 22 24 00 05 25 00 00 02 72 00 00
  B0  00 00 24 00 00 30 00 00 00 00 40 00 10 20 40 00
  C0  65 55 30 00 26 22 70 00 71 74 70 00 71 31 70 00
  D0  45 71 10 00 74 73 70 00 74 75 70 00 77 11 10 00
  E0  75 75 70 00 75 71 70 00 00 20 20 00 00 10 12 00
  F0  12 42 10 00 07 07 00 00 42 12 40 00 71 20 20 00
0400  07 54 70 00 77 57 50 00 65 75 60 00 76 66 70 00
  10  73 33 70 00 76 76 70 00 76 76 60 00 74 57 70 00
  20  55 75 50 00 72 27 70 00 11 57 70 00 45 66 50 00
  30  44 47 70 00 57 75 50 00 47 77 50 00 77 55 70 00
  40  77 57 40 00 77 55 60 00 75 76 50 00 76 73 70 00
  50  77 72 20 00 55 57 70 00 55 55 20 00 55 77 50 00
  60  55 25 50 00 55 22 20 00 71 24 70 00 32 22 30 00
  70  40 20 10 00 62 22 60 00 25 00 00 00 00 00 00 70
  80  21 00 00 00 02 57 50 00 44 75 70 00 00 76 70 00
  90  11 75 70 00 00 75 67 00 32 72 20 00 00 75 71 70
  A0  44 75 50 00 02 02 20 00 02 02 26 00 44 56 50 00
  B0  02 22 20 00 00 57 50 00 00 07 50 00 00 75 70 00
  C0  00 75 74 40 00 75 71 10 00 76 60 00 00 32 60 00
  D0  02 72 20 00 00 55 70 00 00 55 20 00 00 57 50 00
  E0  00 52 50 00 00 55 71 70 00 62 30 00 32 42 30 00
  F0  22 02 20 00 62 12 60 00 00 63 00 00 00 00 00 00
```