May 2021

Operation Guide

PAL-1

EPROM Expansion Card

1. BACKGROUND

In the PAL-1's memory map, there was previously a 16K portion of unused memory between $A000 and $DFFF. This lies between the upper most part of the 32K RAM expansion ($2000-$9FFF) and the 'mirrored' portion of the bottom 8K ($0000-$1FFF mirrored at $E000-$FFFF).

The ROM expansion board allows the use of this 16K block to permanently store programs or other information. The widely available 27C256 32K EPROM chip is used in the design.

So why a 32K EPROM chip if we only have a 16K memory window in memory?

This arrangement allows us to have two separate 16K ROM images that we can manually choose between. (For the technically inclined, this selection is made by forcing the A14 address line of the 27C256 either low or high.)

Details on the software burned onto the EPROM is provided here, but it is also possible for the user to develop their own custom software.

2. ROM BANK 0

The lower 16K of the ROM includes a menu driven ROM loader and a few other pieces of software. Maybe the easiest way to think of a ROM loader is as something similar to a cassette tape or punched paper tape interface that can quickly copy a program into RAM for use.

Note: *Use of some of the software on the ROM loader assumes the presence of the 32K memory expansion. There is not sufficient memory available for some of these programs to load without the expansion. If you don't have the RAM expansion, there are still some programs of interest over in ROM Bank 1.*

To get started with the ROM loader, follow these steps:

1. Set the bank selection jumper so that A14 is held low (logic zero).
2. Power up the system, hit reset, and then press enter until you are in the PAL-1's monitor.
3. Type `A000` and the space bar. This should set the monitor to the $A000 location.
4. Press `G` (for Go).

If things are set up, you should see a prompt like the following:

```
KIM
8600 EA A000<space>
A000 4C G
PAL-1 ROM Loader v1.0

(B)ASIC, (F)ORTH, eWo(z) Monitor, or e(X)it? _
```

At this menu, you can use upper or lower case characters. Typing an `X` will exit the ROM loader program and put you back into the PAL-1's native monitor. Any character that is not related to a valid option will cause the prompt to repeat.

There are three listed and one unlisted options. We will discuss each of these below.

2.1 ORIGINAL MICROSOFT BASIC

The original version of 9-digit Microsoft BASIC for the KIM-1 is widely available and probably the most commonly used high-level language for retro-computing enthusiasts using KIM-1 and it's later incarnations.

The ROM contains an image and by pressing `B` at the menu the user initiates a sequence of copying the image down into RAM (starting at $2000 as usual). There is a brief message that BASIC is staring (typically taking less than a second) and then BASIC is started (by clearing the decimal mode and jumping to $4065). You will have the usual prompts from the BASIC cold start routine as part of the startup.

```
KIM
8600 EA A000<space>
A000 4C G
PAL-1 ROM Loader v1.0

(B)ASIC, (F)ORTH, eWo(z) Monitor, or e(X)it? B
Starting MS BASIC...

MEMORY SIZE? <enter>
TERMINAL WIDTH? <enter>
WANT SIN-COS-TAN-ATN? Y<enter>

 24510 BYTES FREE

MOS TECH 6502 BASIC V1.1
COPYRIGHT 1977 BY MICROSOFT CO.


OK
_
```

If you want to go straight to BASIC from the PAL's monitor bypassing the menu, that is possible. At the start of the ROM loader there is a set of jumps that can be used to do just this. From the PAL-1's monitor, typing `A003<space>G` will load and start BASIC without any additional steps by the user.

This version of BASIC has been patched to allow use of the backspace key. (Details of the patch can be found in the source code.)

2.2 FIG FORTH

FORTH was a language that continues to be popular among some vintage microcomputer hobbyists. At the time the KIM-1 was popular, one of the most common versions of FORTH to encounter was from the FORTH Interest Group (FIG).

Typing `F` at the ROM menu prompt will load a version of FIG FORTH into memory at $2300 and do a warm start.

If you haven't used FORTH before, the following example can help you verify things are working as they should as you get started. (Contents between parentheses are options comments that don't need to be typed in.)

```
KIM
8600 EA A000<space>
A000 4C G
PAL-1 ROM Loader v1.0

(B)ASIC, (F)ORTH, eWo(z) Monitor, or e(X)it? F
Starting FIG FORTH...

fig-FORTH  1.1
2 2 + . ( ADD TWO PLUS TWO AND PRINT )
4 OK
: COUNT5 ( COUNT TO FIVE ) 0 BEGIN DUP . CR 1+ DUP 5 > UNTIL DROP ;
OK
COUNT5
0
1
2
3
4
5
OK
_
```

FIG FORTH can also be started directly from the PAL-1's monitor by typing `A006<space>G`.

NOTE: *There were some significant changes to the FORTH language as standards were developed beginning back in the 1980s. Books and tutorials from the earlier days of the language will be better references if you want to learn FIG FORTH. Some later material will not be compatible with this implementation.*

2.3 ENHANCED WOZ (EWOZ) MONITOR

The monitor program originally written by Steve "Woz" Wozniak for the original Apple 1 is affectionately known as the Woz Monitor. eWoz is a slightly enhanced version of this that has been implemented on vintage 6502-based systems and some of the more recent clones and new designs. While not fancy, the original was impressively implemented in only one page (256 bytes) of memory.

On the PAL-1, this program is a bit different than the previous one because it actually runs from ROM instead of being copied into RAM. Typing `Z` at the prompt will enter the eWoz Monitor which will display a backslash to let you know that it is ready to go.

```
KIM
8600 EA A000<space>
A000 4C G
PAL-1 ROM Loader v1.0

(B)ASIC, (F)ORTH, eWo(z) Monitor, or e(X)it? Z
Starting eWoz Monitor...

Welcome to eWoz 1.0P
\
9FFF<enter>
9FFF: 24
.A01F<enter>
A000: 4C 0F A0 4C 32 A0 4C 60 A0 4C F7 A1 4C 54 A4 D8
A010: A2 FF 9A A9 1C 85 E0 A9 A1 85 E1 20 0A A1 A9 36
B000.B01F<enter>
B000: 01 C8 20 8E 32 20 DA 35 20 27 29 4C 3C 2B 20 6B
B010: 3B A5 0F 20 0F 29 20 C6 00 F0 07 C9 2C F0 03 4C
A000 A003 A006 A009<enter>
A000: 4C
A003: 4C
A006: 4C
A009: 4C
_
```

In this example, we are seeing how the monitor can be used to examine memory. Typing in a hexadecimal address and pressing enter will display the address and its current contents. Skipping ahead a bit, you can see that if you type a start and end address with a period between them, eWoz will display the contents of that range.

eWoz does have a few features that are useful but maybe not intuitive. Going back up, if we type a period followed by an ending address we see we also get a hex dump. The starting address is the previous address we looked at incremented by one. So since we looked at $9FFF, the internal pointer had incremented the address to $A000 and that is used as the start address in the first dump.

At the end you can see how multiple addresses can be specified on a single line. In this example, these are the 6502 JMP (Jump) instructions for the vector table at the start of ROM Bank 0.

Below is an example of using the deposit function and then printing the results using the examine function.

```
A00: 00<enter>
0A00: F0
A10: 0 1 2 3 4 5 6 7 8 9 A B C D E F<enter>
0A10: 0F
:F E D C B A 9 8 7 6 5 4 3 2 1 0<enter>
A00.A2F<enter>
0A00: 00 F0 F0 F0 F0 F0 F0 F0 F0 F0 F0 F0 F0 F0 F0 F0
0A10: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0A20: 0F 0E 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00
_
```

In this example, typing `A00:<space>00` first displayed the current contents of address $0A00 and then changed the contents to $00.

On the second line, we use the same command but provide 16 bytes of data that is loaded into memory. Notice that leading zeros are not necessary.

Finally, on the third line, we didn't specify a start address but, similar to the examine command, the deposit command had incremented the last address we used and deposites the 16 bytes of data we provided into memory.

And at the end we use the examine command to look at the block of memory we have been working in and verify the results are what we expect.

One useful hint is that the output of the examine instructions is in the proper format to be loaded back into the monitor with the deposit function. If we are using a terminal emulator program, we can cut and paste these to save and load small programs. There are also programs that can convert larger assembly language programs into this same format for loading. While this is handy if we have nothing other than the Woz monitor available, eWoz has one additional feature that I use a lot during development work when transferring programs to the PAL-1.

The Intel Hex format is a text file format used to transfer binary data such as ROM images or programs in text format. Many assemblers can produce output in Intel Hex format that is ready to be loaded into RAM or ROM on the target machine. Even for those assemblers that don't have native support, there are a number of tools to help create and tailor Hex files.

While it is handy that you could take a hex dump from the Woz monitor and turn around and reload that information back in at some later time, the fact is that most development today is "cross-platform". That is, while in the past you might have developed a program using hand assembly or a small assembler actually running on the target machine, today most projects of any significant size or complexity are developed using an assembler running on a modern machine to create programs targeted for the older or less capable machine.

Recognizing this, eWoz has added the capability of loading an Intel Hex format file into memory. Since Hex files include information on both the address and the contents of that address, the process is fairly simple. An example is shown below.

```
Welcome to eWoz 1.0P
\
L<enter>
Start Intel Hex Transfer.
..........
Intel Hex Imported OK.
\
_
```

A dot is printed for each line of Intel Hex successfully read to show progress. When loading a large file, you probably want to have the automatic line wrap on in your terminal emulator.

See the later section for more details on using the serial connection, but for this I have found 4800 Baud, 7 Bits, No Parity, 1 Stop Bit, along with a 5 millisecond delay per character and a 500 millisecond delay per line to work reliably. (eWoz uses a line buffer that it fills and then processes. Because of that, the delay between lines is a bit more important than it is with some other applications.)

If you are using Linux, you should probably use the `unix2dos` utility to ensure lines end with a carriage return and line feed instead of just the line feed Linux typically uses for text files.

During development of the ROM contents, I was routinely assembling and uploading files of over 8K quickly and flawlessly with eWoz. While the built-in Paper Tape format reader of the KIM-1 Monitor is certainly usable, the use of Intel Hex format made development and debugging easier.

One final command is the Run command. This will start execution at the specified location. Below is an example of restarting the ROM loader by running from $A000.

```
PAL-1 ROM Loader v1.0

(B)ASIC, (F)ORTH, eWo(z) Monitor, or e(X)it? Z
Starting eWoz Monitor...

Welcome to eWoz 1.0P
\
A000R<enter>
A000: 4C
PAL-1 ROM Loader v1.0

(B)ASIC, (F)ORTH, eWo(z) Monitor, or e(X)it? _
```

Unlike the original KIM-1 Monitor, this does not reset flags or preload register values. It assumes that programs initialize themselves, but does enter the program with the 6502's decimal mode cleared and with the stack pointer initialized to the upper part of Page One (the stack should be clear, but it never hurts to initialize it if you plan to use it).

eWoz can also be started directly from the PAL-1's monitor by typing `A009<space>G`.

2.4 MEMORY TEST

There is one menu option that isn't listed. Typing `M` will load a memory test program into Page Zero and dump you into the PAL-1's monitor.

```
PAL-1 ROM Loader v1.0

(B)ASIC, (F)ORTH, eWo(z) Monitor, or e(X)it? M
Copying memory test (from First Book of KIM) to Page Zero...

KIM
0002 A9 G
KIM
A000 4C _
```

For details on how the memory test works and how to interpret the results, check out a copy of the First Book of KIM or look at the source code for the PAL-1 Memory Test on the W4JBM github site.

You should be able to press `G` in the monitor and if you have a properly functioning 32K expansion memory card, your session should look something like what you see above. If you see an address other than $A000 in the monitor when you exit, it might indicate a memory error occurred.

3. ROM BANK 1

In the beginning, ROM Bank 0 was primarily focused on making it easier to run a few KIM-1 classics out of RAM. On the other hand, ROM Bank 1 focuses on programs that run directly from the ROM and brings you some programs that are either new or not typically associated with the KIM-1.

ROM Bank 1 is not menu driven, meaning that you need to jump to individual programs from the PAL-1's monitor to start them. For quick reference, here is a list of the software and the location in memory:

Apple 1 Integer BASIC	$A000
eWoz Monitor	$B000
PBUG Monitor	$B400
VTL-02	$D700

3.1 APPLE 1 INTEGER BASIC

In the history of hobby computing, some things are legendary. For almost any 6502 or early Apple enthusiast, one of those things is the original version of Integer BASIC written by Steve "Woz" Wozniak for the original Apple 1 computer. (From now on, we'll call it A1B which is short for Apple 1 BASIC.)

While it proved a bit of a challenge, here it is running on the PAL-1!

To start, make sure the right ROM bank is chosen and from the PAL-1 monitor monitor type `A000<space>G`. This will jump to $A000 where there is a copy of the original Apple 1 BASIC (A1B) ROM (relocated from the original address of $E000) with only a few (11 bytes worth) patches to make it run on the PAL-1.

If you have never used an Apple 1 (or, more likely these days, a clone or an emulator), just a bit of context.

The Apple 1 had as flashing @ as the cursor. It had no backspace. Literally. Also there was no delete. Instead, there was a key labeled RUB OUT that generated an underscore ("_") character that was printed to indicate that the previous character had been deleted. So, for example, if you mistyped something you might have a screen that looked like:

```
>PRINT "HELO_LO WR_ORLD!"<enter>
HELLO WORLD!

>@
```

Also, as I understand it, the original keyboard only generated uppercase characters.

Since I had to patch the input and output routines (to work with the PAL-1) anyway, I took a bit of liberty here. By default, lower case alpha characters are converted to uppercase. Also, the BACKSPACE key on most terminal emulators will actually backspace. (So will typing an underscore character.)

WARNING: *Please don't type the underscore ("_") character while in this default mode! Integer BASIC uses this as its end-of-line marker and doesn't expect to encounter it in any context other than as a RUB OUT character or at the end of an input line in the line buffer. You almost certainly WILL break something if you insist on using it in print statements or anywhere else.*

For those wanting the authentic experience, you can revert to the original behavior of the underscore key and character by typing `POKE 6116,0`. If you want to return to the more intuitive (for modern users) style, type `POKE 6116,255`.

Currently the workspace space only goes from $0800 to $1000 (just like on the original Apple 1), so there are 2,048 bytes (2K) for both the program and any variables. If you have the 32K RAM Expansion Card for your PAL-1, you can open up more memory than early Apple 1 users could have imagined using the HIMEM and LOMEM commands in A1B. Here's an example:

```
Apple 1 Integer BASIC by Steve Wozniak
>PRINT PEEK(74),PEEK(75)
0       8

>PRINT PEEK(76),PEEK(77)
0       16

>HIMEM=32767

>LOMEM=8192

>SCR

>PRINT PEEK(74),PEEK(75)
0       32

>PRINT PEEK(76),PEEK(77)
255     127

>_
```

WARNING: *Do not try to print the value of HIMEM or LOMEM. The results are not accurate and it seems to ~~sometimes~~ always foul up the HIMEM pointer.*

The PEEK commands show the contents of the pointers to LOMEM and HIMEM. (Remember that the 6502 is little endian, so in the printout above LOMEM was originally at $0800 and HIMEM was originally at $2000.) The above commands give us 24K of memory for our program and variables. We can squeeze a bit more out by using a negative value for HIMEM (that is left as an exercise for the reader, as one of my college professors was fond of saying in a textbook he wrote) or poking in values, but realistically this should be more than adequate for any A1B programs you are likely to come across.

On the value we used for LOMEM in the example above, we need to make sure that HIMEM is above the PAL-1's I/O and ROM area (since LOMEM is also above it). Conversely, when we first started A1B, both HIMEM and LOMEM were below those areas.

There was no load or save command in A1B. You could exit to the monitor and save the proper portion of memory to a cassette tape if you had one. (To be successful, you had to save the right portion of the program space as well as the right values from Page Zero.) The easiest thing to do these days is to use the text cut and paste (or capture and replay) features of your terminal emulator.

If you do want to exit to the PAL-1's monitor, you can type `CALL 7202` and then press the enter key once or twice (just like after a reset) to get back to the monitor prompt.

Now, let's talk about a bit of history…

A1B was developed on hand-written pages Woz kept in a notebook starting back in 1975. It was hand assembled and the source code was not entered into a computer--only the object code (the hex values that Woz mentally converted the instructions to) was entered one byte at a time. (In the very early days, Woz and Steve Jobs didn't even have a cassette interface and had to do this entry manually each time they powered up a machine.) Even the expanded version of Integer BASIC that came on the early Apple][computers was hand assembled.

There are a few listings of the code floating around today, but they were all "reverse engineered"; typically starting with a disassembly of the ROM and working backwards to figure bits of it out.

One result of this is that there are certain "artifacts" in the code: places where there wasn't room to do something so there is a jump to a location where there happened to be some free memory and also some strings of bytes that aren't used but where it wasn't worth the effort to recode other parts to make use of some space that was freed up.

I have written significant chunks of assembly code both as a hobby and as a profession. To me, A1B is probably the apex of the hand assembly era of the hobby microcomputer. (A very sharp contrast to Microsoft who did early development with cross assembly tools on Harvard's PDP-10 and, later, a time share system.)

In the research on relocating and porting A1B, no mention of it ever having been previously ported to the KIM-1 was found. Given the timeframe, it is likely that most people with a KIM-1 would have either used Tom Pittman's Tiny BASIC or one of the flavors of Microsoft BASIC available for the KIM. There simply weren't ROM image files of the A1B ROM floating around and, even if there had been, it would not have been trivial to relocate it without using some recreated source code.

The bottom line is that this is likely the first widely and openly available version of A1B for the KIM-1 and its posterity. (And maybe the first and only ever.)

3.2 ENHANCED WOZ (EWOZ)

This was described in section 2.3 and a copy is included in both banks for two reasons.

First, it fits with the inclusion of Apple 1 Integer BASIC. In fact you may find some BASIC programs that are available as "memory images" in the format used by the Woz Monitor and which can be loaded using eWoz. (As long as they stuck to the bottom 4K of RAM like the early machines should have, this should work in theory. I can't say that I've actually tried it though…)

Second, it is one of the easiest ways to load Intel Hex files into memory. (Although we are about to see another program with that capability.)

3.3 PBUG (THE PAL-1 DEBUGGER)

PBUG is an adaption of Jeff Tranter's JMON monitor tailored for the PAL-1's ROM Card.

PBUG includes the types of features you would expect in a monitor along with both a disassembler and assembler. To get started, make sure the right ROM bank is chosen and from the PAL-1 monitor type `B400<space>G`. This will jump to $B400 where PBUG starts.

You should see a prompt that looks like `PAL> _` and can type `?` to get a list of available commands.

```
KIM
0000 00 B400<space>
B400 D8 G
PBUG for the PAL-1
PAL> ?
PBUG for the PAL-1

Derived from JMON (C) by J.Tranter
PAL-1 hardware by Liu Ganning
Adaptation by Jim McClanahan W4JBM

Assemble    A <address>
Breakpoint  B <n or ?> <address>
Copy        C <start> <end> <dest>
Dump        D <start>
Fill        F <start> <end> <data>...
Go          G <address>
Write Hex   H <start> <end>
Checksum    K <start> <end>
Load Hex    L
Info        N
Registers   R
Search      S <start> <end> <data>...
Unassemble  U <start>
Write       W <address> <data>...
Monitor     $
Trace       .
Help        ?
PAL> _
```

We will briefly go through the functions available, but the explanations do assume a certain level of familiarity with the 6502 microprocessor and the PAL-1/KIM-1 implementation.

There are several input routines used by the various PBUG commands. Addresses do require any leading zeros to be entered as do data bytes. In most situations, if you forget this or make a mistake in typing, pressing the ESCAPE key will allow you to exit back to the command prompt.

Let's start with the memory DUMP command because that is one that most people are at least somewhat familiar with. The spaces in commands is automatically inserted, but the command line will be something like this:

```
D <start_addr>
```

Where the start address is given in hex. For example, type DA000 and you will get a hex dump of memory contents starting at $A000. (By the way, commands can be either upper or lower case.)

```
PAL> D A000
A000 4C 00 B3 4C 0E B3 10 FB AD 10 D0 60 8A 29 20 F0  L..L.......`.) .
A010 23 A9 A0 85 E4 4C C9 A3 A9 20 C5 24 B0 0C A9 8D  #....L... .$....
A020 A0 07 20 C9 A3 A9 A0 88 D0 F8 A0 00 B1 E2 E6 E2  .. ............
A030 D0 02 E6 E3 60 20 15 A7 20 76 A5 A5 E2 C5 E6 A5  ....` .. v......
A040 E3 E5 E7 B0 EF 20 6D A0 4C 3B A0 A5 CA 85 E2 A5  ..... m.L;......
A050 CB 85 E3 A5 4C 85 E6 A5 4D 85 E7 D0 DE 20 15 A7  ....L...M.... ..
A060 20 6D A5 A5 E4 85 E2 A5 E5 85 E3 B0 C7 86 D8 A9   m..............
A070 A0 85 FA 20 2A A0 98 85 E4 20 2A A0 AA 20 2A A0  ... *.... *.. *.
A080 20 1B A5 20 18 A0 84 FA AA 10 18 0A 10 E9 A5 E4   .. ............
A090 D0 03 20 11 A0 8A 20 C9 A3 A9 25 20 1A A0 AA 30  .. ... ...% ...0
A0A0 F5 85 E4 C9 01 D0 05 A6 D8 4C CD A3 48 84 CE A2  .........L..H...
A0B0 AD 86 CF C9 51 90 04 C6 CF E9 50 48 B1 CE AA 88  ....Q.....PH....
A0C0 B1 CE 10 FA E0 C0 B0 04 E0 00 30 F2 AA 68 E9 01  ..........0..h..
A0D0 D0 E9 24 E4 30 03 20 F8 AF B1 CE 10 10 AA 29 3F  ..$.0. .......)?
A0E0 85 E4 18 69 A0 20 C9 A3 88 E0 C0 90 EC 20 0C A0  ...i. ....... ..
A0F0 68 C9 5D F0 A4 C9 28 D0 8A F0 9E 20 18 A1 95 50  h.]...(.... ...P
A100 D5 78 90 11 A0 2B 4C E0 A3 20 34 AE D5 50 90 F4  .x...+L.. 4..P..
A110 20 E4 AF 95 78 4C 23 A8 20 34 AE F0 E7 38 E9 01   ...xL#. 4...8..
A120 60 20 18 A1 95 50 18 F5 78 4C 02 A1 A0 14 D0 D6  ` ...P..xL......
A130 20 18 A1 E8 B5 50 85 DA 65 CE 48 A8 B5 78 85 DB   ....P..e.H..x..
A140 65 CF 48 C4 CA E5 CB B0 E3 A5 DA 69 FE 85 DA A9  e.H........i....
A150 FF A8 65 DB 85 DB C8 B1 DA D9 CC 00 D0 0F 98 F0  ..e.............
A160 F5 68 91 DA 99 CC 00 88 10 F7 E8 60 EA A0 80 D0  .h.........`....
  <Space> to continue or <ESC> to stop_
```

The output of the dump command (as well as several of the other commands) will pause once the typical terminal display is full and will prompt asking whether to continue or stop. Pressing

the spacebar will display another 23 lines of memory while pressing escape will return you to the `PAL> _` prompt.

The WRITE command will let us modify the contents of memory. Here is an example of it in use (with some of the memory dump truncated for clarity).

```
PAL> D 0400
0400 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
...
PAL> W 0400 12 34 56 'A' 'B' 'C' <enter>
PAL> D 0400
0400 12 34 56 41 42 43 00 00 00 00 00 00 00 00 00 00  .4VABC..........
...
PAL> _
```

We started with all zeros at $0400, but then used the WRITE command to enter $12, $34, $56. The spaces are automatically inserted in the command line and we only had to only type the hexadecimal values. Then we used a shortcut to enter ASCII values for A, B, and C. In this case, we type the opening single quote and the character we want to enter. The closing single quote and a space are automatically displayed and we can continue entering either the next hexadecimal value or use a single quote again to enter another ASCII value.

What if you want to put a certain value in a range of addresses? The FILL command allows you to do just this in the form:

```
F <start_addr> <end_addr> <data>...
```

The data you provide can be a single byte that is repeated (like filling with all $00). You can also enter a series of bytes such as `F 8000 80FF 00 01 02 03<enter>` which will put the sequence $00, $01, $02, and $03 in memory over and over until it hits the end of the specified range.

You can also use a single quote for ASCII values in the data portion of the FILL command.

Another command is GO which will let you specify an address where the monitor should jump and begin execution.

```
G <start_addr>
```

If you want to exit from PBUG back to the PAL-1/KIM-1 built-in monitor program, you can type `$`.

There is a powerful set of features available in PBUG including an assembler, support for breakpoints, the ability to trace through single steps of a program, the ability to examine and alter register values (during single steps), and a disassembler.

Below is an example of using the assembler and disassembler.

```
PAL> F 0400 04FF 00
PAL> A 0400
0400: LDX #ED
0402: LDY #C0
0404: JSR BE28
0407: JMP 0407
040A: <escape>
PAL> D 0400
0400 A2 ED A0 C0 20 28 BE 4C 07 04 00 00 00 00 00 00  .... '.L........
0410 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
...
  <Space> to continue or <ESC> to stop
PAL> U 0400
0400   A2 ED       LDX   #$ED
0402   A0 C0       LDY   #$C0
0404   20 27 BE    JSR   $BE28
0407   4C 07 04    JMP   $0407
040A   00          BRK
040B   00          BRK
...
041C   00          BRK
  <Space> to continue or <ESC> to stop
PAL> B 1 0407
PAL> G 0400
PBUG for the PAL-1

Breakpoint 1 at $0407
A-00 X-ED Y-C0 S-0140 P-32 ..-B..Z.
0407   4C 07 04    JMP   $0407
PAL> <press period>
A-00 X-ED Y-C0 S-0140 P-32 ..-B..Z.
0407   4C 07 04    JMP   $0407
PAL> <press period>
A-00 X-ED Y-C0 S-0140 P-32 ..-B..Z.
0407   4C 07 04    JMP   $0407
PAL> _
```

This program loaded pointers to the PBUG welcome message, jumped to the subroutine that prints that message, and then jumps into an endless loop.

But we have also set Breakpoint #1 to $0407, so when it first hits the endless loop the breakpoint is triggered and the register values, program counter, and a disassembly of the instruction at the breakpoint location (that is, the next instruction to be executed) is dumped to the console.

Pressing `.` will advance past the breakpoint (which, in this case, is in a loop so that doesn't do much for us). The `R` command will let you change register values or even the program counter as you step through a program.

Using the write command to create a null-terminated message and the code above to jump to a print routine should help you get started building an understanding of how PBUG can help you create and troubleshoot simple programs.

For loading and saving files, the `L` command loads an Intel Hex file and `H` will let you create a hex file for a given range of memory.

NOTE: Timing constants are important. PBUG handles each character as it comes in, so there is the need for a delay of around 25 ms between characters. This is much higher than what eWoz requires, but that is because much of the processing is happening as data is received in PBUG while eWoz just grabs the data, stores it in a buffer, and then processes when a carriage return is encountered. You could probably back down the end of line delay from the 500 ms required by eWoz, but I haven't tinkered with that.

You can copy blocks of memory using `C`. The routine will properly handle situations where you are overwriting part of the source area. For example, if we type `C 0400 04FF 0480`, the copy routine is smart enough to start copying from $04FF to $057F and then work it's way down until it ends with a final copy from $0400 to $0480.

If you had typed `C 0400 04FF 0380`, the routine realizes it should start copying from $0400 down to $0380 and continue working its way up until it copies $04FF to $047F.

The `K` command will generate a four digit checksum for a range of memory. I don't use this much, but it is a handy way to check that the memory contents of an address range aren't changing. For example, when I was debugging PBUG I would load it in RAM at $2000 and generate a checksum from $2000 to $66FF. After tinkering for a half hour with various commands, I could generate the checksum again and would expect it to match unless I had somehow corrupted something in memory.

One final command is `N` which will provide information on the system.

```
PAL> N
         Computer: PAL-1/KIM-1
         CPU type: 6502
RAM detected from: $0000 to $13FF
RAM detected from: $1780 to $17FF
RAM detected from: $2000 to $9FFF
  2nd RIOT Board : Not Found
       NMI vector: $1C1C
     RESET vector: $1C22
   IRQ/BRK vector: $1C1F
PAL> _
```

One example of where this information could be useful would be if a user encounters strange behavior from a program that works for others. This might tell us that instead of an NMOS 6502, that person was using a CMOS 65C02 processor which behaves differently in certain situations.

Part of the reason I wanted a monitor as robust as PBUG is to allow easy experimentation with code fragments. The included assembler, disassembler, breakpoints, and register dumps can let me enter a small portion of code and walk through it one step at a time to make sure the registers and the flags are doing what I expect them to.

And now a bit more history…

Early minicomputers like DEC's PDP-11 typically had a debugger that let you examine and change memory locations, look at registers, run portions of code, and other things you often needed to do as you got your code functioning as you wanted.

There were different tools available, but the DDT (apparently given the 'backronym' of Dynamic Debugging Technique) on the DEC platforms along with DDT (actually short for Dynamic Debugging Tool) in CP/M were early examples.

On the Motorola 6800, the initial debuggers were the MIKBUG and the MINIBUG (discussed in Motorola Engineering Note 100). Later another 6800 vendor, Southwest Technical Products Company (SWTPC) had SWTBUG. Even today, many of the hobby monitors for the 6800 and 6809 are called something-BUG.

(There were were the BUGBOOKs on various topics, but that doesn't seem related. There was also the Fairchild F8 with FAIR-BUG that may have been influenced by the 6800.)

Then we get to the 6502…

Instead of debuggers, the same piece of software seemed to be more frequently referred to as a monitor on 6502 systems. (The first I used was actually 65V on an Ohio Scientific C1P, but there were enhanced monitors available along with ROMs such as SYNMON and CEGMON.)

But the Commodore 64 really saw the proliferation of something-MONs: TEDMON, SuperMON64, SMON, and MON-64 among them.

One of the best 6502 monitors floating around and widely used by hobbyists today is JMON written by Jeff Tranter. It supports a number of platforms including the KIM-1. After a few weeks of tinkering with the PAL-1, I knew I needed a better debugger / monitor and JMON was the starting point.

At the same time, I always found the "bug" names a bit endearing and the monitor eventually evolved into the PAL-1 Debugger or PBUG.

3.4 VERY TINY LANGUAGE (VTL-02)

VTL is short for Very Tiny Language. Some of the details are a bit difficult to nail down, but VTL was written by Frank McCoy and Gary Shannon in 1976 for the 6800-based Altair 680 and later ported to the 8080-based Altair 8800. Memory of any type (RAM or ROM) was at a premium and VTL was able to fit into less than 1K.

Later, VTL-2 was developed for the Altair 680 Computer system. It included over thirty new features and fit into 768 bytes of ROM! That meant it could be installed in three empty PROM sockets on the Altair 680's CPU board.

In 2012, Mike Barry ported VTL-2 to the 6502 and called it VTL-02. (The "oh-two" being, it seems, a nod to the sixty-five oh-two processor… There was mention of a VTL-09 for the 6809 in Carl Warren's The MC6809 Cookbook, but so far as I know neither the source nor object code for this was ever released.)

There are a few different versions of VTL-02 (some optimized for size, some for speed, etc.) and the 6502 version has some features (like bit-wise operators) that the 6800 version lacked.

It is often described as being BASIC-like. (In fact, I have made that observation myself.) But other than line numbers and the question mark being the print command (not a shorthand for PRINT like on many BASICs), there aren't actually a lot of obvious similarities.

The manuals are available online, but the easiest way to get started (for many anyway) is to write a simple program that will actually do something. The example session below is a simple example of a VTL-02 program being created, listed, and run.

```
0000 00 D700<space>
D700 A9 G
6502 VTL-02C for the PAL-1

OK
10 X=1<enter>
20 #=(X=6)*100<enter>
30 ?=X<enter>
40 ?=""<enter>
50 X=X+1<enter>
60 #=20<enter>
0<enter>
10 X=1
20 #=(X=6)*90
30 ?=X
40 ?=""
50 #=20

OK
```

```
#=1<enter>
1
2
3
4
5

OK
<press RS button for reset><enter>
0000 00 D715<space>
D715 D8 G
OK
0<enter>
10 X=1
20 #=(X=6)*90
30 ?=X
40 ?=""
50 X=X+1
60 #=20

OK
#=1<enter>
1
2
3
4
5

OK
_
```

Let's go through the program line by line…

Line 10 initializes the variable X to a value of 1.

Line 20 is best understood from the middle where the conditional comparison (X=6) will return a 0 if it is not true or a 1 if it is true. That is then multiplied by 100, so we have either 0 if X<>6 or 100 if X==6.

Now we get to one of the quirky (yet cute) things in VTL-02…

The variable `#` is the current line number. Setting it to 0, by definition of the language, does nothing. (Or, put another, `#=0` does not alter the value of `#`.) Setting it to any other value (like 100) will go to that line or, if that line number does not exist, the next highest. Since there is no line 100 and nothing higher than that in the program, Line 20 effectively does a comparison of X

to 6, continues execution if it is not true, or jumps past the rest of the program and ends execution if it is true.

Line 30 prints the value of X to the console and Line 40 ends the current line (print a carriage return and linefeed) of print,

Line 50 increments X by 1 and Line 60 loops back up to Line 20 where the comparison logic figures out if we have passed the number 5 or not.

To list a program, you type `0`.

To run a program, you type `#=1`. Remember the earlier discussion--`#=0` does nothing and `#=1` will take us to line number 1 or the next highest line. (We could also run the program with using `#=10`, but consistently using `#=1` isn't a bad habit to build.)

After running the program (or if you mistyped something and end up in a loop), you can use the PAL-1's RS (Reset) button on the main board. Press enter a time or two to get into the monitor and then you can do a warm start of VTL-02 by jumping to $D715.

After a warm start, you can list and run the program you were working on.

The VTL manuals online have some sample programs that will let you further explore VTL-02.

4. TROUBLESHOOTING

If the ROM Card does not function, the first thing to check is the soldering to make sure there are no solder bridges, unsoldered pins, or other problems of that sort.

With the ROM Card installed, if you use the monitor to examine $A000, you should see $4C as the contents. If you hit `<enter>` three times to look at $A003, you should see another $4C. (This is because both ROM images start with two jump instructions which start with the instruction byte of $4C.)

If you don't see the expected values and if the values don't seem to change as you go from $A000 to $A003, it could be several things. If the values don't seem to change at all, the ROM may not be getting selected and enabled when it should. If the values look like they might be 6502 machine code but don't work like they should, two address lines might be shorted or an address line might be floating.

The vast majority of the time, a careful visual inspection will find the problem.

5. SERIAL COMMUNICATIONS LIMITATIONS

The KIM-1 was originally built as a low-cost single-board computer to show how much could be done with so few components. Many people used the built in hexidecimal keypad and LED display while a lucky few might have had teletype machine or even a dumb terminal to connect to it.

The serial connection is primitive by today's standards. Today, we would politely call it a "software UART", but in the old days it was often called a "bit banger".

For output, there is basically a line that we can toggle high or low. By doing this eight times (a start bit and seven data bits for the KIM-1) in a carefully timed loop, we can send a single ASCII character.

Conversely, for input we have to watch a line, wait for it to toggle states, and then check again at carefully timed intervals to determine a sequence of bits has been sent to us over that single input line.

Because we have to watch for the line to go low, this style of input is sometimes referred to as "blocking" (since we block execution of any other code as we watch and wait).

Today, most of us are used to the behavior of a real UART. UARTs can handle these tasks that require precise timing without bogging down the microprocessor. Because they have built in buffers, a UART can capture a character and wait for us to ask if one is available or accept a character to be sent while hiding the details of sending it from the user. (This is called "non-blocking" since we can just pop over and check whether there is a character waiting for us or not whenever we want to.)

The KIM-1 also operated with a 1 MHz clock speed. Normally that is "fast enough", but handling input and output (I/O) can push the limits. The KIM-1 serial port was designed with a teletype operating at 110 baud in mind. A lucky few with the early hobby-grade dumb terminals might have been able to connect at speeds of 1200 baud.

The PAL-1 can handle serial speeds of 9600 baud so long as you are careful. But if you type too fast or type while the system is printing something, things may get garbled. As long as you have reasonable expectations about the level of performance that is realistic, things should be okay. But there are things like checking for Control-C that can't be done in what is considered "the normal way".

From experience, the details of the "best" serial setting (baud rate, pause between characters, pause at end of line, etc.) vary with different serial adapters and with different terminal emulator programs.

Most terminal emulator programs allow you to specify "pauses" between characters or at the end of a line. For eWoz, I get reliable transfers of hex files with 7 bits, no parity, and 1 stop bit

along with a 5 millisecond delay between characters and a 500 millisecond delay after the line. For PBUG, the setting are much different (because of the way input is handled) and I get reliable transfers of hex files with 7 bits, no parity, and 1 stop bit along with 25 milliseconds between characters and 500 milliseconds at the end of line.

BASIC interpreters are another application where the end of line delay can be important. When you type in the line and hit enter, the computer has to tokenize the line, potentially move the contents of memory around to make room for it (which means moving existing content around and rebuilding the "codes" that help the interpreter parse through tokenized programs), and then insert it in the proper location.

With BASIC, the delay can vary based on program size and where the line we just typed needs to go. (If you have a 4K program and type a new line that goes at the beginning, BASIC has to move the entire 4K to make room for your new line of code.)

I have noticed that if I try to get settings as low as possible (so transfers are as fast as possible), there seems to be some dependence on the USB to serial adapter I am using. I have also found that things seem to be able to support higher speeds more reliably when I turn off the hardware echo that the KIM-1 hardware normally enables by default.

6. BUILDING YOUR OWN ROM IMAGES

All of the code needed to create your own ROMs is available on the W4JBM GitHub site:

```
https://github.com/w4jbm
```

But just because the code is available doesn't mean the process is trivial to someone just beginning.

For example, consider PBUG…

I built and debugged it in RAM with an address of $2000. When it was working, I did the final testing by building it for $B400. (That is the address it ended up at somewhat naturally, but also it is easy to remember because it is like the word "before".)

That particular piece of code is built with the CC65 assembler, CA65, which can create a binary image using tools developed by Jeff Tranter. I then used a command line tool called srec_cat to move the binary into an Intel Hex file where the address of $B400 was carried over.

I combined that with the files for Apple 1 Integer BASIC, eWoz, some patch code that makes Integer BASIC work with the PAL-1's I/O routines, and the code for VTL-02.

At the end, that gave me an Intel Hex file that had pieces of object code scattered between $A000 and $DFFF.

Since we are using 16K of a 32K ROM and this is what goes into the upper half, we need to correct the addresses of the hex records without changing the object code itself. Some EPROM burners include software to handle things like this, but I use a simple command line burner that I would rather feed with a properly structured Intel Hex file.

So, to do this, I used srec_cat again and used an offset of -$6000 to bring the addresses for the object code records down from $A000-$DFFF to $4000-$7FFF. (If this had been for the "bottom half" of the EPROM, I would have used an offset of -$A000 instead.)

There is nothing overly complicated about all of that, but you will likely need to keep notes and double check your math if you decide to create your own multi-bank ROM. (I know I did--and I still messed it up a few times in the process.)

7. REPORTING BUGS

The best way to report any bugs is to report them through the W4JBM GitHub Repositories.

```
https://github.com/w4jbm
```

Direct e-mail is likely to get caught up in spam filters although I (and lots of other helpful people) are on various lists and in various groups related to vintage computing and the 6502 family of processors.

8. THANKS, RECOGNITION, AND LICENSE STATEMENT

Any original material, modified material, or newly developed material necessary to create the ROMs along with the object code burned into the EPROMs are all freely available on the W4JBM GitHub. Material is intended for personal use only.

Where applicable, original and modified source code is made available under the appropriate licenses. For details on licensing, see the various portions of code on GitHub.

This material was created for educational and research purposes as well as to preserve parts of what you could call microcomputing's digital heritage. (Okay, I'll admit it… That and so we could all have some fun with it.)

Any copyrighted material without explicit licenses for use is included under the fair use provision of copyright laws. Any material will be removed at the request of a copyright holder or those holding other rights to it.

This effort would not have been possible without the previous work of so many people.

Thanks to Liu Ganning for creating the PAL-1. I started rekindling my interest in retro computing about five or so years ago and the PAL-1 has, by far, been the most enjoyable portion of that journey. Liu's enthusiasm for the PAL-1 is contagious and it has been great to have the opportunity to get to know him by e-mail.

From a hardware perspective, much has been built on the work of Vince Briel, Ruud Baltissen, and, of course, Chuck Peddle and the team from MOS Technology, Inc..

For general KIM-1 documentation, Hans Otten's site has been invaluable. His trick to prevent the auto-echo on the serial port is in almost everything I do with the PAL-1! Also the information on Tiny BASIC from Bob Applegate at Corsham Technologies along with other material he provides has helped things along.

And I cannot begin to count the number of times I have wanted to start something, did some searching, and ended up at Jeff Tranter's GitHub pages as the starting point. The breadth and depth of Jeff's knowledge and work is incredible. (And he started out with an Ohio Scientific system that he purchased around the same time I also purchased my own OSI as my first computer.) PBUG's start from JMON may be one of the most noticeable things as you read through the documentation, but if you dig into the source code you will find incredible amounts of material based on work by Jeff (including the code I started with in porting A1B over).

Some other by the particular application (but not in any particular order):

Microsoft BASIC: Thanks to Bill Gates, Paul Allen, Ric Weiland, and the others who wrote the early versions of BASIC and created an industry. Thanks again to Hans for keeping the object code and documentation available.

eWoz Monitor: Thanks to Steve Wozniack for the original Woz Monitor and to fsafstrom for the enhanced version. Also thanks to TangentDelta and Glitchworks for the implementation on the R6511 that both hooked me on eWoz and helped me migrate it to the PAL-1.

FIG FORTH: Thanks to the FORTH INTEREST GROUP (FIG) and to W.F. Ragsdale's widely distributed version for the Rockwell AIM. I also struggled to get things working until I happened across a version modified by W.K. Rodinger and Mike Fortuna.

Memory Test: Thanks to Jim Butterfield and The First Book of KIM for this.

Apple 1 Integer BASIC: Of course, thanks to Steve Wozniack (yet again) for a piece of code that is simply incredible. It has some rough spots, but it also embodies so much of the "get it done" attitude of the early days of hobby computing. Also thanks to Jeff Tranter (yet again) for the re-created source code that helped get it running. And thanks to Eric Smith and to Paul Santa-Maria for their disassembly/reassembly work.

PBUG Monitor: Thanks to Jeff Tranter for making JMON available! The features it offers make it a great tool for anyone wanting to learn more about the 6502. I hope my port to the PAL is up to the high standard it sets.

VTL-02: Thanks to Frank McCoy and Gary Shannon for the original. And thanks to Mike Barry for the port to the 6502.

On a personal note, my journey with the 6502 started as a 16 year old in 1980. I wanted a computer and I know my parents weren't totally convinced of its necessity, but that first Ohio Scientific C1P has led to me being better prepared for so many aspects of my career from the technical bits to the interpersonal bits. Thanks to the late Fred Gauger who hired me at Gauger and Associates to help service and sell OSI computers in Tulsa. The 18 months or so after getting that first computer, working for the dealer, and starting college was the beginning of an incredible journey.

Finally, thanks to my wife who understands the hours in the basement tinkering with my computers.

I apologize in advance to anyone I may have forgotten. There is a league of 6502 hobbyists who support the PAL-1 and other projects by sharing their knowledge and experience so freely.

Jim McClanahan, W4JBM
May 2021