The Woz Monitor

When a computer is powered up it must know what it must do. It goes without saying that a piece of software must be executed. Since the computer has just been powered up there must be some software in memory somewhere. This software is programmed in ROM memory because ROM chips are non volatile and will not forget the contents when power is removed.
Fairly obvious don't you think? Well, think again! The most popular "personal computer" back then, the Altair 8800, had no ROM at all. When you powered that machine up it did absolutely nothing. You had to flip a few hundred switches in order to enter the first program, which was then able to load the real stuff in.

The Apple 1 was far more advanced, it did contain some ROM memory. The basic version of the Apple 1 had a total of 256 bytes of ROM! Remember that memory was very expensive at the time the Apple 1 was developed. This ROM was filled with the legendary Woz Monitor. It allowed the user to examine and change memory contents and it allowed machine language programs to be started.

Using The Woz Monitor

The original Apple 1 did not come with a Reset circuit, which means that the user has to press the RESET switch in order to get the machine started. Once you do that a back slash '\' is printed on the screen and the cursor will drop down one line. The cursor position is represented by a flashing '@' symbol.

You can now type address, data and commands which will be executed as soon as you press the Return key. The input buffer can hold up to 127 character, if you type more characters before hitting the Return key the input line will be erased and will start again from scratch. This is overflow situation is indicated by a new back slash after which the cursor drops one line again.
Because of the primitive nature of the terminal there are not many line-editing features available. You can press the back arrow key to erase characters from the input buffer, but the erased characters will not be erased from the screen nor will the cursor position back-up. You'll have to keep track of the changes yourself. It's obvious that you can easily get confused when a line contains too many corrections or when an error is detected all the way at the other end of the input line. In that case it would be easiest to cancel the input and start all over again. Cancelling the input is done by pressing the ESC key.

Address inputs are truncated to the least significant 4 hexadecimal digits. Data inputs are truncated to the least significant 2 hexadecimal digits.
Thus entering 12345678 as address will result in the address 5678 to be used.
Tip: This can also be used to your advantage to correct typing errors, instead of using the back arrow key.

If an error is encountered during the parsing of the input line then the rest of the line is simply ignored, without warning! Commands executed before the error are executed normally though.

Examining memory (memory dump)

You can examine the contents of a single memory location by typing a single address followed by a Return.

```
4F

004F: 0F
```

Note: The **bold** typed characters are what the user types. All other characters are responses from the Apple 1.

Now let us examine a block of memory from the last opened location to the next specified location.

```
.5A

0050: 00 01 02 03 04 05 06 07
0058: 08 09 0A
```

Note: 004F is still considered the most recently opened location.

We can also combine the previous two examples into one commend:

```
4F.5A

0040: 0F
0050: 00 01 02 03 04 05 06 07
0058: 08 09 0A
```

Note: Only the first location of the block 4F is considered opened.

You can also examine several locations at once, with all addresses on one command line.

```
4F 52 56

004F: 0F
0052: 02
0056: 06
```

Note: 0056 is considered the most recently opened address.

Let's take this concept into the extremes and combine some block and single address examinations on one command line.

```
4F.52 56 58.5A

004F: 0F
0050: 00 01 02
0056: 06
0058: 08 09 0A
```

Note: By now you won't be surprised that 0058 is considered the most recently opened location.

Finally let's examine some successive blocks of memory. This can be handy if you want to examine a larger block of memory which will not fit on one monitor screen. Remember that there is no way to halt a large examine list other than hitting the RESET button!

```
4F.52

004F: 0F
0050: 00 01 02
.55

0053: 03 04 05
.5A

0056: 06 07
0058: 08 09 0A
```

Depositing memory (changing memory contents)

This is how to change a single memory location (provided it is RAM memory of course).

```
30:A0

0030: FF
```

Note: FF is what location 00300 used to contain before the operation, from now on it contains A0. Location 0030 is now considered the most recently opened location.

Now we're going to deposit some more bytes in successive locations, starting from the last deposited location.

```
:A1 A2 A3 A4 A5
```

Note: Location 31 now contains A1, location 32 contains A2 and so on until location 35 which now contains A5.

Combining these two techniques will give us the next example.

```
30:A0 A1 A2 A3 A4 A5

0030: FF
```

Note: Location 0030 used to contain FF in this example.

Breaking up a long entry into multiple command lines is done like this:

```
30:A1 A2
```

```
0030: FF
:A2 A3

:A4 A5
```

Note: A colon in a command means "start depositing data from the most recently deposited location, or if none, then from the most recently opened location.

Now we're going to examine a piece of memory and then deposit some new data into it:

```
30.35

0030: A0 A1 A2 A3 A4 A5
:B0 B1 B2 B3 B4 B5
```

Note: New data deposited beginning at most recently opened location, which is 0030 in this example.

Running a program

To run a program at a specified address:

```
10F0 R

10F0: A9
```

Note the cursor is left immediately to the right of the displayed data; it is not returned to the next line. It's the program's responsibility to control the rest of the output.

From now on the user program is in control of the Apple 1. If the user program does not return to the Woz monitor (by jumping to address $FF1F) you'll have to press the RESET key to stop your program and return to the Woz Monitor.

You can also enter a program and run it all from the same command line. Please note that this only works for very short programs of course.

```
40: A9 0 20 EF FF 38 69 0 4C 40 0 R

40: FF
```

Note: FF is the previous contents of location 0040.
This little program will continue printing characters to the screen. It can only be stopped by pressing the RESET key.

The Woz Monitor's RAM Use

The monitor needs some RAM memory to perform its tasks. When a user program is running all bytes used by the monitor may be recycled, the monitor doesn't care about the contents of any of the memory locations when it regains control again. Here's the complete list of all the memory the Woz Monitor requires while it is running:

Zero page	**$24 to $2B**	General purpose storage locations. None of the bytes are very important and they may all be changed by the user program.
Stack page	**$0100 to $01FF**	Although the Woz Monitor only uses 3 bytes of stack space at most there is no way of telling where the stack actually is. This is because the stack pointer is not initialized by the Woz Monitor. A user program may use the entire stack for its own purposes. However be careful when entering code on page $01 before the stack pointer is initialized, the monitor may overwrite your code again.
Input buffer	**$0200 to $027F**	This space is used as input buffer. User programs may use this area. However you can not enter code here manually because it will be overwritten by the monitor.

The next addresses are not exactly RAM locations, which doesn't make them less important though. They are the 6821 PIA control registers.

KBD	**$D010**	Keyboard input register. This register holds valid data when b7 of KBDCR is "1". Reading KBD will automatically clear b7 of KBDCR

Bit b7 of KBD is permanently tied to +5V. The monitor expects only upper case characters.

KBDCR	**$D011**	The only bit which we are interested in in this register is the read-only bit b7. It will be set by hardware whenever a key is pressed on the keyboard. It is cleared again when the KBD location is read.
DSP	**$D012**	Bits b6..b0 are the character outputs for the terminal display. Writing to this register will set b7 of DSP, which is the only input bit of this register. The terminal hardware will clear bit b7 as soon as the character is accepted. This may take up to 16.7 ms though!
DSPCR	**$D013**	This register is better left untouched, it contains no useful data for a user program. The Woz Monitor has initialized it for you. Changing the contents may kill the terminal output until you press RESET again.

Useful Routines

Apart from the monitor program itself the Woz Monitor contains only a few useful routines which can be called by user programs.

$FF1F GETLINE	This is the official monitor entry point. If your program is finished and you want to return to the monitor you can simply jump to this location. It will echo a CR and from then on you are back in the monitor.
$FFEF ECHO	This simple routine prints the character in the Accumulator to the terminal. The contents of the Accumulator are not disrupted, only the flag register will be changed. Although this is a fairly short routine it may take up to some 16.7 ms before it returns control to the user program. For more details about this behaviour please read the page about the terminal.
$FFDC PRBYTE	This routine prints the byte which is held in the Accumulator in hexadecimal format (2 digits). The contents of the Accumulator are disrupted.
$FFE5 PRHEX	Prints the least significant 4 bits of the Accumulator in hexadecimal format (1 digit). The contents of the Accumulator are disrupted.

If you want to read a single character from the keyboard from within your own machine language program you can use the following piece of code:

```
KBDIN       LDA     KBDCR       See if there is a character available
            BPL     KBDIN       Not as long as b7 remains low
            LDA     KBD         Get the character and clear the flag
```

The Woz Monitor In More Detail

As a user you do not need to know the details about the Woz Monitor. But since I had to understand how the Apple 1 worked I had to find out all the details for myself anyway. Therefore I might as well share them with you in case you're interested.

wozmon.asm program listing

Here you see the entire listing of the Woz Monitor. I did my best to describe in detail what is actually happening in the program, although most of the remarks are the original remarks made by Steve himself. The source code wozmon.asm is part of a download package which can be downloaded from the download section.

Now we're going to take a closer look under the bonnet of the Woz Monitor. After studying it a bit closer you'll soon notice that Steve had to make every effort to squeeze his monitor in the tight 256 bytes memory space of ROM he had available.
Personally I have made some modifications to the Apple]['s DOS system in the distant past. There I had to squeeze some patches in the few bytes which were still free or could be freed by altering the existing code a little. Therefore I can fully appreciate the efforts Steve has made to make it fit.

However all this squeezing came with a cost. I've listed some of the concessions Steve had to make in order to get the job done with only 256 bytes at his disposal.

- The Stack pointer is not initialized. This means that the stack can be all over the place in page $01. This is not really a problem if you use page $01 for stack purposes only. If you intend to put some code in page $01 you'd better initialize the stack pointer, otherwise the stack may overwrite your code without warning.
- The PIA is not properly initialized. It only works because the PIA is automatically set to Data Direction Mode after a Reset. Steve used this fact to set the Data Direction Mode without explicitly selecting the DDR registers. Thus initializing the PIA directly after a Reset is no problem. However you're not supposed to restart the system by calling address $FF00!
- Only upper case ASCII characters are accepted. Lower case characters are not recognized and are considered an error. Also bit b7 of every ASCII character is always set, whereas officially it should be cleared.
- Error situations simply reset the input buffer, a back slash symbol is printed to indicate that an error has occurred and the cursor is dropped one line. No error messages are output at all.
  Obviously Steve didn't have resources enough for a complete human interface.
- The program is written in a top-down manner. No problem for a processor of course, but it is harder to read and understand for us humans. But who cares, it works!
  Often results from one operation are recycled later in the program for completely unrelated purposes. This saves some bytes because certain registers already contain the proper value and don't have to initialized again. Thus altering one part of the code may cause problems somewhere else in the program.
- The GETLINE routine only partially parses the input line. Any input character below the ASCII value for '**.**' is regarded to be a blank. This means that entering `00 01 02 03` is equal to `00!01#02,03` for instance.
  Today a program like this would not be considered to be fool proof, but maybe we've got better fools today compared to '76.

Mind you, despite of all these concessions Steve managed to do it all in 254 bytes, 2 bytes of the ROM are still unused at the end.

Finally looking at the vector table at the end of the program teaches us that NMI handling should be done by code at address $0F00. And IRQ handling is done by code starting at address $0000.
Per default both interrupt lines are not connected to any source. Therefore you don't have to bother about these handling routines.