Raspberry Pi 2 Model B v1.1 65C02 Simulator

I created this software package to provide an inexpensive development system for 65C02 code running on modern hardware. The Pi has serial, SPI, I2C, Ethernet, audio, SD card, and video peripherals.

I have managed so far to gain access to the serial UART, video system, and read-only access to the SD card's root directory.

I will probably work on the I2C and SPI interfaces next.

With the addition of read access from the SD card, I am now able to provide a means for users to provide their own 65C02 OS. My SBC-2 OS will still load, but if the file is found and loaded from the SD card, it will overwrite the default SBC-2 OS.

The memory map is as follows:

$0000 - $7FFF = RAM

$8000 - $80FF = IO

$8100 - $FFFF = ROM

Further down this instruction, I'll provide details on the IO addressing.

To load you own OS into the simulator, create a binary image of your OS and name it "SBCOS.ROM". The ROM image file needs to be exactly 32,768 bytes long. The first page will not be used, but the remainder is available for your use. Be sure to load the rom on to root directory of the SD card.

The SD Card reader code was verified on 3 different SD cards. All were formatted with a 256MB FAT partition. This is how the Raspian SD is configured when that image is built. If the simulator cannot read your SD card, check the size of the 1st partition and be sure it is formatted using FAT.

The IO page is set up with the currently supported devices. I will expand it as I add more support.

READ

0x8000	read serial data register
0x8001	check status of serial input buffer: 0= no data, 0xFF=data present
0x8010	check status of serial output buffer: 0=full (wait), 0xFF=ready to accept data
0x8030	read lowest 8 bits of the 1MHz system timer
0x8040	read video color register
0x8041	read video Y register
0x8042	read LSB of video X register
0x8043	read MSB of video X register
0x8044	read Video Core status: 0x00=ready for command, 0xFF= busy
0x8048	read Video Core status: 0x00=ready for command, 0xFF= busy

WRITE

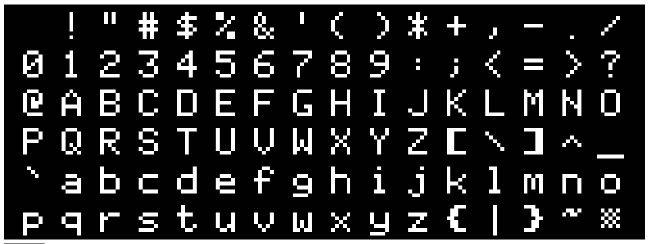
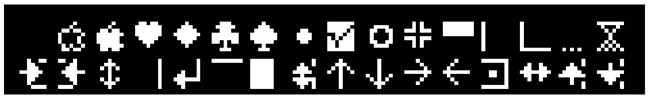
0x8010	Write data to serial port		
0x8020	Turn system LED off (any value)		
0x8021	Turn System LED on (any value)		
0x8040	Set pixel color	Color = 0x00RRGGBB	
0x8041	set pixel Y register	0-199	
0x8042	set pixel LSB X register	0-255	When MSB=1, range is 0-63
0x8043	set pixel MSB X register	0-1	
0x8044	Signal Video Core to plot the pixel (any value)		
0x8045	Clear the screen (any value)		
0x8048	Print ASCII character on the screen and advance the cursor		
0x8049	Set cursor to custom character (ASCII value)		
0x804A	Set character background color		Color = 0x00RRGGBB
0x804B	Set character foreground color		Color = 0x00RRGGBB

The video display screen is set up as a 320 x 200 pixel graphical display using 6 bits for color, 2 for each RED, GREEN, & BLUE. It uses the HDMI port. The color byte is formatted as 0x00RRGGBB. Writes outside of the 230x200 range will be ignored. You need to set up all 4 locates: color, Y coordinate, X low byte, and X high byte before you write location 0x8044. When using text, you can set the background and character colors using the same color byte scheme. In this way, each character can have its own color pattern.

The text generator uses some common ASCII control codes to generate the display. In addition, there are some commands in the upper half of the character map to allow for direct placement of the cursor and to select cursor shape and visibility. Also, using ASCII codes 0x01 and 0x02 will control what the upper half of the ASCII code generates. Setting it to upper replaces the cursor controls with more graphical symbols.

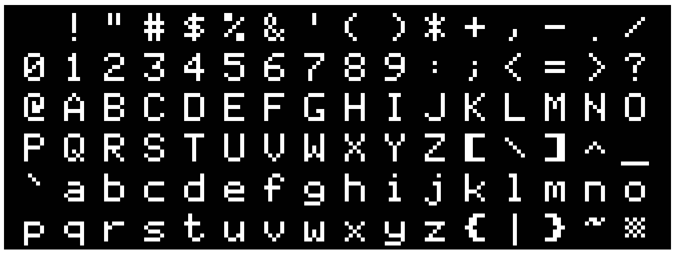
It might best be understood by playing with sending various codes to 0x8048 and observing the results.

Here is a copy of the font map, it has both Low font (0x01) and high font (0x02) shown:

	Low Font	
00	null	
01	Set Low Font	
02	Set High Font	
08	backspace	
09	tab	
0A	linefeed	
0C	formfeed	
0D	return	
20 - 7E	std ASCII	
7F	delete	
80-9F	80 90	
A0 - B8	setrow	A0 (top) to B8 (bottom)
B9	null	
BA	cursor home	
BB	cursor off	
BC	cursor block	
BD	cursor underscore	
BE	cursor on	
BF	null	
C0 - E7	Set column	C0 (left) to E7 (right)
E8	cursor up	
E9	Cursor down	
EA	Cursor left	
EB	Cursor right	
*	Any code not shown is "null", does nothing	

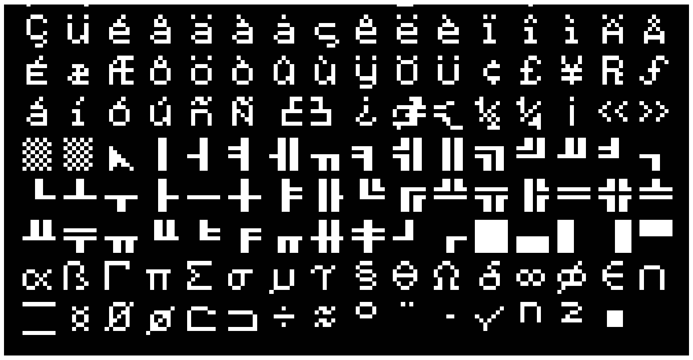
High Font

00	null
01	Set Low Font
02	Set High Font
08	backspace
09	tab
0A	Line feed
0C	Form feed
0D	return
20 - 7E	std ASCII
7F	▒



80 - FF

80
90
A0
B0
C0
D0
E0
F0



The serial port is configured at 115,200 baud, No parity, 8 data bits, & 1 Stop bit. The port pins are on the expansion port. Pin 6 is Gnd, pin 8 is TX, and pin 10 is RX. This requires a 3.3V TTL serial adapter. A 5V adapter can damage to IO ports. These are inexpensive and can be found online.

If you are not sure where to start, you can try this program with my SBCOS. There is a little more infor about it here: https://sbc.rictor.org/sbcos.html

The Zip file will contain the following files:

To be loaded to the SD card:

Start.elf – PI's core startup code

Bootcode.bin – Secondary boot file

Config.txt – used to set up turbo mode and sets the clock to 900MHz

Kernel7.img – this is the 6502 Simulator package – it has a copy of my SBCOS included.

Raspberry PI 2 Model B v1.1.pdf – This file

This program is currently ustilizing 2 of the 4 cores from the PI's processor. The first core sets up the system caches and memory manager, tries to load the SBCOS.ROM file from the SD card, and start the simulator. The seconds core initializes the display and stands by waiting for input from the simulator.

The other two cores are "parked" in a low-power state. I plan to put them to work also when I get to things like audio and Ethernet.

I estimate that the simulator is running at about 125MHz. This is not a consistent speed, as there is no cycle counting or pacing being done, and some instructions will take longer than others.

I hope you find this useful and fun!

Daryl