# “Apple 2 Next”

*Written by Paul Robson 11th July 2023*

This is a first draft specification of the “Apple II Next” the working (but unlikely to be final) name for the retro computer build on Olimex’s Neo6502 board.

The idea is conceptually the same as the Mega65 and Spectrum Next machines. The system is a more modern version of the original, but can run the software of its parent in compatibility mode.

## Processor Hardware

The processor is a 65C02 rated at up to 8Mhz, connected to 64k of RAM memory, which is stored in the Raspberry PI Pico.

As with the standard machine, there is RAM memory from 0000-BFFF, and hardware i/o is between C000 and CFFF. The ROM space D000-FFFF is also writeable (*is this needed to be write protected in compatibility mode ?)*

Compatibility mode largely is identical save for the CPU speed. It would be nice to maximise the processor speed of the 65C02 but in compatibility mode this should run at 1.023Mhz

## Keyboard

The keyboard generates ASCII keys and a strobe to $C000 which can be reset via $C010 software switch (ref Little pp 160)

## Audio

The Apple II speaker toggles between high and low whenever location $C030 is accessed, so the pitch can be calculated from the cycles between.

### *Extension*

The ability to simply specify the pitch of the sound by writing a value to <location>, which overrides the Apple behaviour. Off would be specified by value zero.

## Display

The Apple II display is somewhat interesting and is described in detail in Little. However, in practice there are 3 modes, each of which has 2 areas of memory that can store the text/graphic data

- a 40x24 text display with upper case text (6 bit ASCII) that supports inverse and flashing using the other 2 bits.
- A 40x48 low res colour graphics display – each character is divided into an upper and lower half which defines the colour of that half.
- A 280x192 6 colour high res colour graphics display where pixels are encoded in 7 bits of 8 (the MSB is a colour selector) and operate in pairs.

Additionally the lower 32 scan lines can be used as 4 rows of 40 characters, rendered in the same way as the text display.

These are controlled by soft switches at $C050 … $C057.

## *Extension*

The concept behind the extension is to give each of the 192 lines its own renderer. This specifies what is drawn on each scan line.

- Render type 0 would be the text display (with the current scan line identifying which row of the CG Rom to access for display data)
- Render type 1 would be the low res colour display.
- Render type 2 would be the high res colour display.

Each of these would take their data from the position specified by the hardware reference ; the page (400-7FF,800-BFF) or (2000-3FFF,4000-5FFF), and the position in that memory as specified by offset for types 0,1 or the interlaced offset system of the High Res mode (consecutive lines are not consecutive in memory).

For extended renderers there is a 17 bit register which sets the display memory address. This is ignored by Renders 0,1,2 which take their addresses from the hardware reference.

Each renderer advances through memory starting at that position. There is no requirement for all renderers to use the same amount of memory.

Lines can be repeated using type code $FF. This allows unused or simple areas to be effectively repeated multiple times without using system memory.

If the render code is $FF the previous line is repeated using the same data. One can simply alternate graphic render lines with $FF which would halve the resolution to 96 pixels , saving memory.

The 17th bit allows the use of extended memory (if this is available) e.g. another bank of the 264k provided by the RP2040, which could reduce screen memory usage in the main memory space to zero.

Actual renderers would be a topic for further discussion, but a blank line would be useful, as would 1 bpp, 2 bpp, 4 bpp and perhaps 8 bpp lines. Palettes could be provided for the first 2 at least, so there would be say 4 1bpp renderers each with its own reserved colour byte.

## *Example*

So as an example (with sample renderers) :

The top 16 lines contains the score, which requires a line of 280 pixels in one colour (Renderer 4, 35 bytes per line)

The next 4 lines are empty

(Renderer 5, 0 bytes per line)

The next 128 lines are graphic , 280 pixels in one of four colours, (Renderer 6, 4 pixels per byte, 140 bytes per line).

The rest are blank

The address register is at $2000

The set up would be something like the following table.

| Scanline | Renderer# | Bytes this line | Addr of Gfx Data |
|---|---|---|---|
| 0 | 4 | 35 | $2000 |
| 1 | 4 | 35 | $2023 |
| 2 | 4 | 35 | $2069 |
| …. | | | |
| 15 | 4 | 35 | $21C7 |
| 16 | 5 | 0 | $220D |
| 17 | 5 | 0 | $220D |
| 18 | 5 | 0 | $220D |
| 19 | 5 | 0 | $220D |
| 20 | 6 | 140 | $220D |
| 21 | 6 | 140 | $220D |
| 22 | 6 | 140 | $2299 |
| …. | | | |
| 147 | 6 | 140 | $6781 |
| 148 | 5 | 0 | $680D |
| …. | | | |
| 191 | 5 | 0 | $680D |

There is no requirement for the layout to vary in this fashion ; it is simply an option. One could simply put the same renderer on every line.

### ***Compatibility***

When locations $C050..$C057 are activated the line renderer table can be set accordingly, so the first 160 could be high-resolution and the lower 32 text. This would be done by the handler code in the RP2040. From the point of view of the Apple compatibility it would make no difference.

## Disk System

I don't know much about this other than it's another Wozniak special design.

@OliverSchmidt commented

*"The Apple ][+ firmware is extensible. The most important extension controls a mass storage device. Such a device offers an interface to read/write 512 byte blocks to/from memory. It's trivial to create a firmware implementing that interface that just hands the block number in question and the memory location in question to the RP2040 which performs the operation"*

Which sounds a bit like it's not difficult to interface. The slight downside with this is the Flash chip in the Neo6502 I think only erases 4k sectors.  Have to think about this one :)

## Memory Interface

Technically the memory space is all used, though looking at the extensions while the Apple ][ decodes for (say $C00X) software uses $C000 as $C001 is used in later machines.

If not problematical I would suggest claiming one of the expansion connector pages, say page $CFxx possibly as follows.

| | |
|---|---|
| $CF00-$CFEF | Write sets the renderer for the given scanline. Allows for 240 lines. |
| $CFF0-$CFF1 | Base address of graphics display |
| $CFF2 | Bit 7 use extended memory for display (1) use CPU memory (0) |
| $CFF3 | Set CPU speed  bit 7 : full speed (0) 1.023Mhz (1) |
| $CFF4 | Set Sound frequency to <constant>/byte value, 0 = off |

## Software

I wouldn't suggest we have what one would normally call a ROM memory. More that the open source firmware is loaded in at start up.

Through some method ; a BASIC command might be the simplest, providing a copy of the Applesoft ROM Image is available (which I don't think we could distribute !) the ROM image could be loaded into the upper 12k of RAM space, and the system rebooted.

References:

- Apple 2 Reference Manual
- Inside the Apple Iic by Gary B Little.