

## **KID ICARUS revealed.**

**An introduction to ROM hacking and NES architecture.**

### **PART 2: Text and VRAM V1.0**

By David Senabre Albuje. 2007

It's been more than half a year that I wrote PART 1, and I really wanted to go on with this article. Many many things remain to be untold. So, after explaining how the game works, I'll give some info about how it stores data.

## **Text on Kid Icarus**

### **Let's hunt text string.**

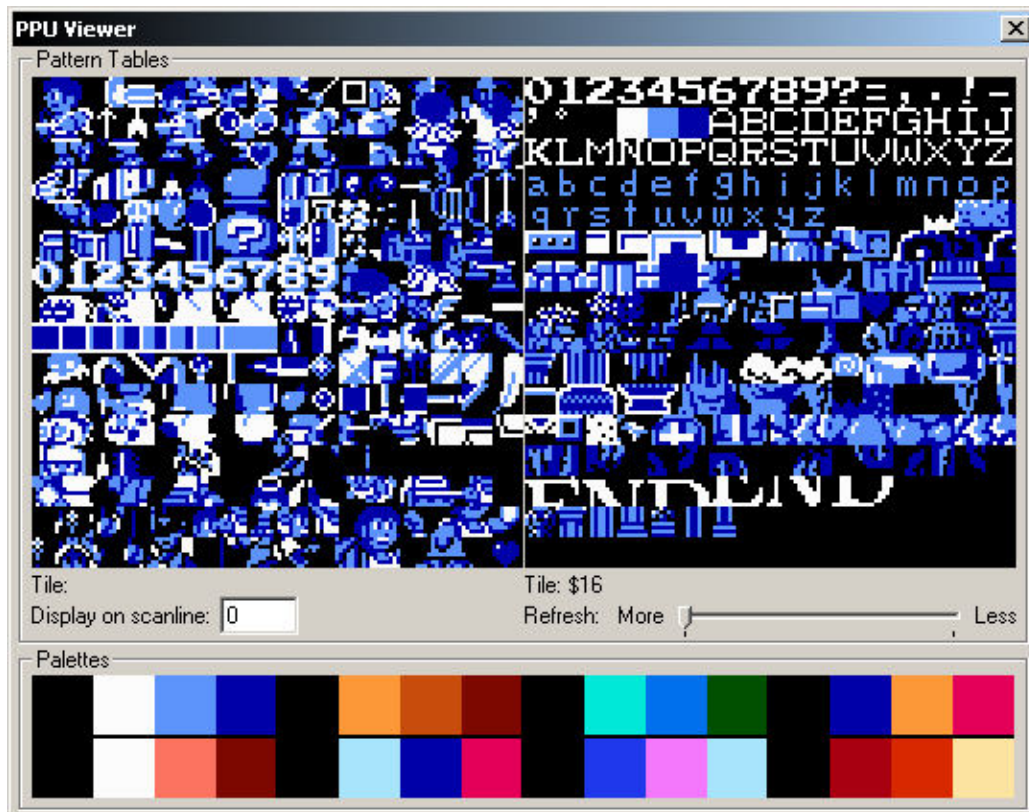
So, so easy. Anyone with some knowledge would have no problem working this out. But, as I'm going as deep as I can with this ROM, I'll also speak about it.

Finding text location is simple, as Kid Icarus ROM stores them in plain format, non compressed or encoded in any way.

You can find most of the text string by using a simple relative searcher. But I'll explain another way of doing it, for those who might not fully understand it, but also because some strings won't be found using relative search.



If you take a look at both pattern tables while in the start menu, you will necessarily see the font characters, as no tile can be displayed on screen if not loaded in the PPU memory. Using FCEUXD emulator, which is great, by the way, you can view the pattern tables seen below.



As you might know, left pattern table is used in game sprites, while right pattern table is used in backgrounds, and it's the one we are interested in, because, most times, static text which does not dynamically change or move, is drawn in the background, and not as sprites. Ok, ok, we could have worked this out by simply realizing that left pattern table does not contain font characters apart from numbers. But it is good to understand where things come from.

So, "START" and "CONTINUE" strings can only be in background at this time, because they are constituted by patterns loaded in the background pattern table. Anyway, that would be the only reasonable option, as it has no sense to create at this point 5 sprites (one for each character in START) and 8 more sprite (one for each character in CONTINUE), when none of them will move neither change.

But there is one sprite in this screen, the arrow (marked with a green square below) which points the selected option, because it must move in response of the player action.



In short, text strings will be in the background the vast majority of times.

Now, how can we find these strings in the ROM?

Go back to the pattern table and find the position of 'S' inside this table. You'll find it is 40, or \$28 (in hex). If you do the same for all the rest of characters, you get that

START is a sequence \$28 \$29 \$16 \$27 \$29  
 CONTINUE is a sequence \$18 \$24 \$23 \$29 \$1E \$23 \$2A \$1A

We can try and see if these strings are stored in the ROM using the same value for each letter that each of them will have in the pattern table.

If you search in an hex editor for \$28 \$29 \$16 \$27 \$29 you get two occurrences, in addresses 0x67E4 and 0x11122.

Let's see what we find in 0x67E4.

```

000067E0 | 04 AC 01 05 28 29 16 27 29 04 CC 01 08 18 24 23 | ..... ( ) . ' ) ..... $ #
000067F0 | 29 1E 23 2A 1A 00 00 80 A0 A0 A0 A0 20 00 00 00 | ) . # * .....
00006800 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
00006810 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
00006820 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
00006830 | 00 00 00 00 00 00 20 44 A8 20 50 A8 20 60 A8 A9 | ..... D . P . ` .
    
```

This one looks just great. A few bytes after the values we suspect could be START (in yellow), there's \$18 \$24 \$23 \$29 \$1E \$23 \$2A \$1A (in red), which corresponds to the sequence CONTINUE.

This is what we were looking for.

## Let's change some text

To change the string we just found, just change the values of each letter, for the value corresponding to the letter you want to turn it into. Example.

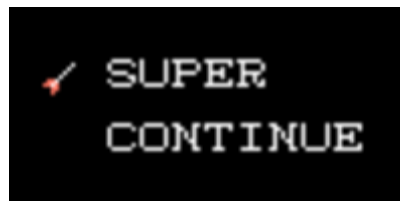
Imagine you want to change START into SUPER.

If you look up in the pattern table the values for U (\$2A), P (\$25) and E (\$1A), change,

\$28 \$29 \$16 \$27 \$29 into \$28 \$2A \$25 \$1A \$27

```
000067E0 | 04 8C 01 05 28 2A 25 1A 27 04 CC 01 08 18 24 23
000067F0 | 29 1E 23 2A 1A 00 00 80 A0 A0 A0 A0 20 00 00 00
```

And this works, as you can see.



## What we CAN NOT do.

This was an easy task. It's great that programmers store text strings in such a simple way. Keep in mind, though, that this does not happen in other ROMs, where other more sophisticated methods are used, especially when lots of text has to be stored.

Anyway, in this ROM we were lucky. But not everything is good about it. An obvious limitation is that you can't make words longer than they were, because you if you go further, you will overwrite data that was stored after the end of the word. You can make a string shorter, by turning the ending characters to value \$12, which in this case stands for an empty pattern.

Imagine you want to change START into GO PIT.

GO PIT has 6 characters (the space is pattern like any other).

But START has only 5 characters.

It's easy to see that you can't replace

\$28 \$29 \$16 \$27 \$29

with

\$1C \$24 \$12 \$25 \$1E \$29

You need one more byte.

You can't go on replacing bytes further that the end of the original string. If you do it, you will overwrite the byte after it, and could cause a disaster, depending on what it was stored there.

What would happen in this example?

Last \$29 overwrites a \$04 that was stored there (marked in red).

```
000067E0 | 04 8C 01 05 1C 24 12 25 1E 29 CC 01 08 18 24 23
000067F0 | 29 1E 23 2A 1A 00 00 80 A0 A0 A0 A0 20 00 00 00
```

And it happens that the value stored in that position was crucial for the ROM to work, and you get a nice black screen.

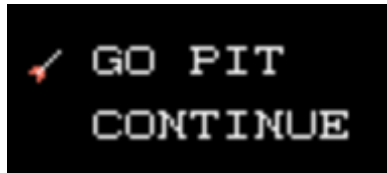
You might think on just inserting value \$29 between \$1E and \$04. You just CAN'T do this and walk again as if nothing happened, because you would shift all data after the inserted byte by one position, and thus, make the ROM unusable.

```
000067E0 | 04 8C 01 05 1C 24 12 25 1E 29 04 CC 01 08 18 24
000067F0 | 23 29 1E 23 2A 1A 00 00 80 A0 A0 A0 A0 20 00 00
```

Hope you were not thinking on this, because if you did, you might need to learn a few more things on ROM basics. This is something everyone who wants to play around with ROMs must know. A general rule is “never insert bytes”, ROMs don't like this, except if you know very well what you're doing... as I'll show next.

### How we CAN do it

Yes, it can be done.



Doing this (I'll explain now).

```
000067E0 | 04 8C 01 06 1C 24 12 25 1E 29 04 CC 01 08 18 24
000067F0 | 23 29 1E 23 2A 1A 00 80 A0 A0 A0 A0 20 00 00
```

By some reason, some bytes after CONTINUE string seem to not affect the game, and they can be overwritten (I guess what they may be, but I'll not discuss it now, to help keep things clear). So, what you need to do is insert the value \$29 (in green) between \$1E and \$04, and then, delete one byte after CONTINUE (the one after \$1A, in red), so all bytes after it go back to its original position

Step by step.

Find original START string.

```
000067E0 | 04 8C 01 05 28 29 16 27 29 04 CC 01 08 18 24 23
000067F0 | 29 1E 23 2A 1A 00 00 80 A0 A0 A0 A0 20 00 00
```

Replace START with GO PI

```
000067E0 | 04 8C 01 05 1C 24 12 25 1E 04 CC 01 08 18 24 23
000067F0 | 29 1E 23 2A 1A 00 00 80 A0 A0 A0 A0 20 00 00
```

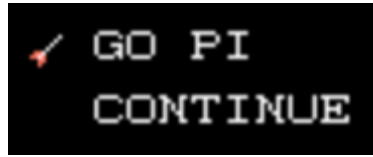
Insert \$29, that is T. Notice how all bytes after it are shifted forward one position

```
000067E0 | 04 8C 01 05 1C 24 12 25 1E 29 04 CC 01 08 18 24
000067F0 | 23 29 1E 23 2A 1A 00 00 80 A0 A0 A0 A0 20 00 00
```

Now we delete the byte after \$1A. See how all bytes after it go back to its original position.

```
000067E0 | 04 8C 01 05 1C 24 12 25 1E 29 04 CC 01 08 18 24
000067F0 | 23 29 1E 23 2A 1A 00 80 A0 A0 A0 A0 20 00 00 00
```

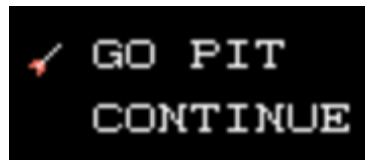
Now, even though this will work, and not crash, you will only see,



We need to make one more change. Change the byte just before the string GO PIT, from 05 to 06 (marked in red, in the image below). This value tells the game about the number of characters that the string coming next has.

```
000067E0 | 04 8C 01 06 1C 24 12 25 1E 29 04 CC 01 08 18 24
000067F0 | 23 29 1E 23 2A 1A 00 80 A0 A0 A0 A0 20 00 00 00
```

Now we've got it



You can do the same for the string CONTINUE, to turn it into a longer string. I tried to add up to 4 characters using this method, and the game seemed to work fine.

Keep in mind that this I've just done, will probably not work in other situation, because most of the times, you just can't overwrite bytes. In this particular situation, I found, by ROM corruption trial and error, that the game would still be happy if we slay a few bytes after CONTINUE string.

### List of string found using the simple method shown here.

67E4 - START (28 29 16 27 29)  
 67ED - CONTINUE (18 24 23 29 1e 23 2a 1a)

#### *Inventory*

1E706 - HEART (1d 1a 16 27 29)  
 1E70F - ENDURANCE (1a 23 19 2A 27 16 23 18 1A)  
 1E71C - STRENGTH (28 29 27 1A 23 1C 29 1D)  
 1E72F - WEAPONS (2c 1a 16 25 24 23 28)  
 1E73A - TREASURES (29 27 1a 16 28 2a 27 1a 28)  
 1E784 - M A P (22 12 16 12 25 12 12) → M A P A (22 12 16 12 25 12 16)  
 1E78C - TOTAL (28 24 29 16 21)  
 1E791 - SCORE (28 18 24 27 1A)  
 1E796 - STAGE (28 29 16 1C 1A)

#### *When you defeat Pandora*

ED26 - PIT (25 1E 29)  
 ED2A - EQUIPPED (1A 26 2A 1E 25 25 1A 19)  
 ED33 - HIMSELF (1D 1E 22 28 1A 21 1B)

ED40 - WITH THE (2C 1E 29 1D 12 29 1D 1A)  
ED48 - TREASURES! (29 27 1A 16 28 2A 27 1A 28 0E)

### *Password input screen*

6B6F - ENTER (1A 23 29 1A 27)  
6B75 - SACRED (28 16 18 27 1A 19)  
6B7C - WORDS (2C 24 27 19 28)

There's another SACRED WORDS but I don't know where it appears in the game.

1C32A- SACRED  
1C331 - WORDS

*(The following PUSH START BUTTON is not the one appearing at the beginning of the game. Don't know yet where it is displayed).*

1111a - PUSH (25 2A 28 1D)  
11122 - START (28 29 16 27 29)  
11128 - BUTTON (17 2A 29 29 24 23)

### **Title Screen is diferent**

While in the title screen, only a few letters are loaded into the pattern table, and they're scattered across it. So, their values in the ROM are expected to change. The pattern table looks like:



12, stands for 'space' again. And the numbers 0..9 are in the same position.

But now, letters have the following positions:

P = 31  
U = 5D  
H = 64  
B = 65  
E = 9F  
N = A6  
A = A7  
Y = AF  
G = B5

R = B6  
O = B7  
D = BF  
S = D0  
T = C3  
. = 0B

Let's try to find PUSH START BUTTON



According to the above table, this strings should be coded as

PUSH = 31 5D C0 64

START = C0 C3 A7 B6 C3

BUTTON = 65 5D C3 C3 B7 A6

Searching the ROM for it PUSH leads us to a unique coincidence.

```
000062F0 | 19 A5 0B 43 2B 24 A5 05 31 5D C0 64 12 C0 C3 A7
00006300 | B6 C3 12 65 5D C3 C3 B7 A6 12 00 00 00 00 80 AA
00006310 | A2 A0 AA 22 00 00 08 0A 0A 0A CE FF FC F0 55 55
00006320 | 55 00 CC FF FF FF FF FF 03 00 00 0C 0F 0F 0F
```

Very nice! We're right again! As you can see, we can be sure we found it, because just after PUSH come the other 2 strings, separated by a 12 (space), just as we see it on screen. Let's play around, turning spaces into dots (replace 12 by 0B)



Offset 0x62F8

PUSH (31 5D C0 64)

START (C0 C3 A7 B6 C3)

BUTTON (65 5D C3 C3 B7 A6)

If you want to translate this into Spanish (PULSA BOTON START), you would change above values into;

31 5D 7B C0 A7 12 65 B7 C3 B7 A6 12 C0 C3 A7 B6 C3



### What we CAN NOT do.

Again, we are not able to just make the sentence longer. You can show up to eighteen maximum characters. Of course, you can make the sentence shorter, by using 'space' (12).

Remember how the code originally was

```

000062F0 | 19 A5 0B 43 2B 24 A5 05 31 5D C0 64 12 C0 C3 A7
00006300 | B6 C3 12 65 5D C3 C3 B7 A6 12 00 00 00 00 80 AA
00006310 | A2 A0 AA 22 00 00 08 0A 0A 0A CE FF FC F0 55 55
00006320 | 55 00 CC FF FF FF FF FF 03 00 00 0C 0F 0F 0F

```

If you take space from the bytes after the last 12 (marked in blue, in the image above), the game seems to be unaffected. Even if you overwrite 80 AA after them, nothing seems to happen. But it is better to leave things as God made them. To do these changes is not the way to succeed.

## Understanding VRAM

Before I tell you how we can do some magic in the title screen, you need to understand how VRAM works, and what it is used for. I'll give a general overview, and experienced people may want to skip this section.

You already know that all graphics displayed on screen must be directly accessible by the PPU (picture processor). This is what the VRAM is used for. Everything loaded in this memory can be read or written by the PPU with no external help. VRAM it's not a different kind of memory, It's exactly the same as the main RAM. The difference between one and another is the use you make of it in the system. While main RAM is controlled by the CPU, VRAM is controlled by the PPU. If the CPU wants to access the VRAM, it has to tell the PPU to do it. This is done using what it's called ports.

There's an onboard 2Kb VRAM chip, used for Name Tables, and another chip placed in cartridges, to store Pattern Tables, which holds the graphics. If you don't know very well what these tables are, read the first part of this article, especially section "Tiles". I'll talk a bit more on Name Tables later on, for those who are not familiarized with NES architecture.

Game cartridges usually have a CHR-ROM, where all graphics used in the game are stored, and like any other ROM, cannot be written. The PPU can only access 8Kb of CHR-ROM, so it can also be mapped, as it happens with the PGR-ROM (the ROM where the game code is stored), to allow the use of much larger CHR-ROMs, and only accessing to an area of 8Kb at a time..

But Kid Icarus cartridge has an 8Kb CHR-RAM instead of a CHR-ROM. It has the same purpose, but it is writeable, and it's empty when the game starts. Both, code and graphics are stored in a single ROM. The game transfers the patterns needed from this ROM to CHR-RAM.

Like the CPU has a memory map, the PPU also has a memory map.

PPU memory map

Memory ranges	
\$3F10-\$3F1F	Sprite palette
\$3F00-\$3F0F	Background palette
\$2000-\$3EFF	Name tables
\$1000-\$1FFF	Pattern table 1
\$0000-\$0FFF	Pattern table 0

The pattern tables are 4Kb each, and are always stored in a CHR-ROM or CHR-RAM in the cartridge.

The Name Tables are stored in the system's VRAM. Really, there's only space for 2 Name Tables, which are 1Kb each, but the programmer sees 4 Name Tables. What you need to know is that only 2 of them really exist. The other ones are just mirrors. But you can choose at any time which one is mirror of which one.

This will be clear with an example.

The first and third world use horizontal mirroring. The content of the Name Tables at the beginning of the game is:



Keep in mind that only Name Tables 0 and 2 exist.

Table 1 is a mirror of 0.

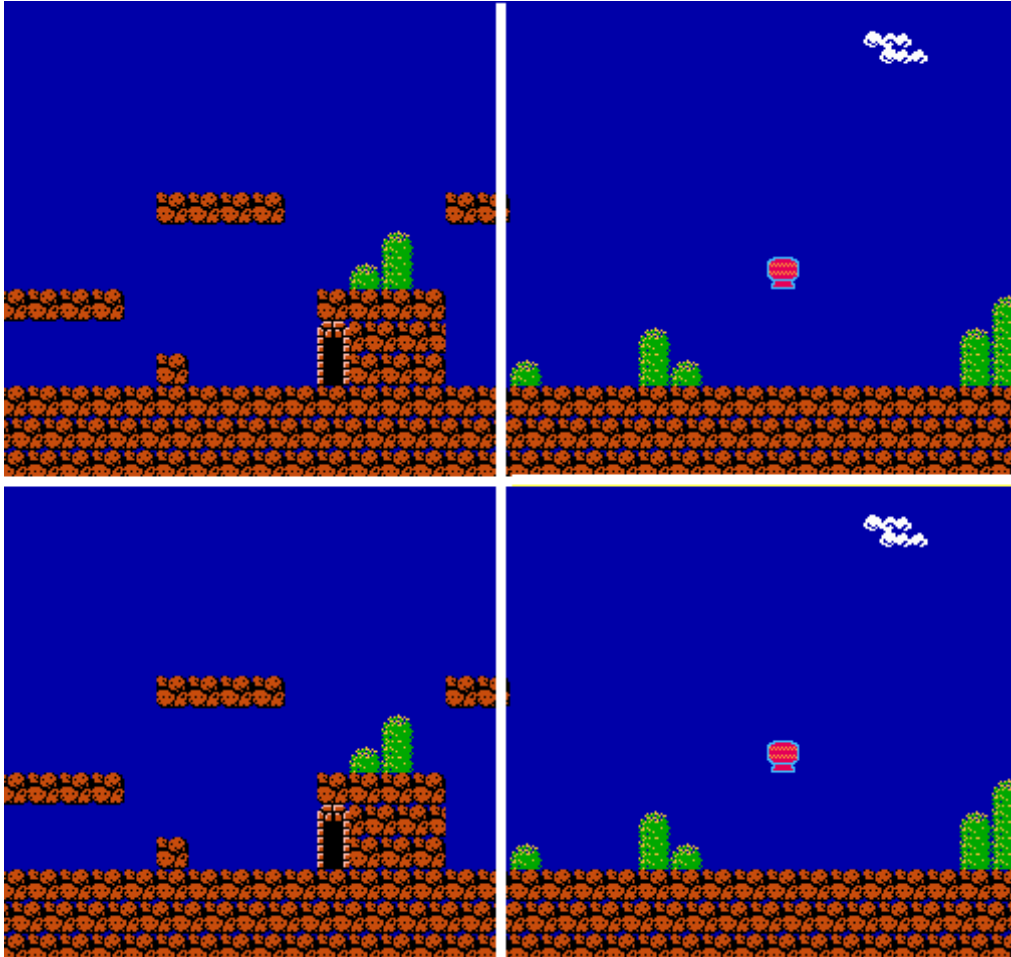
Table 3 is a mirror of 2.

This arrangement lets do a simple scroll upward. I recommend you to open up a NES emulator with a name table viewer, and play a bit , going up, and looking how name tables work.

Second world uses a vertical mirroring instead.

Now table 2 is a mirror of table 0, and table 3 a mirror of table 1.

This arrangement makes an horizontal scrolling easier.



Ok, but how does a Name Table look like in terms of bytes? How is it stores in VRAM?  
Name table 0 starts at PPU address \$2000  
Name table 1 starts at PPU address \$2400  
Name table 2 starts at PPU address \$2800  
Name table 3 starts at PPU address \$2C00

Remember that if you are using horizontal mirroring, Name Table 0 and 1 are the same, so writing to \$2000 is the same as writing to \$2400. And so on.

A Name Table is a chunk of 960 bytes, each one representing one area of 8x8 pixels of the screen. If the first byte is a \$43, then, pattern in position \$43 of background Pattern Table is drawn at the top-left 8x8 pixels area. If a \$12 comes next, then, pattern in position \$12 of patter table is drawn to the right. An so on, until the screen is filled...

This is the least you must know to understand what comes next. But I encourage you to learn more about this, to get a good knowledge.

### **How we CAN do it**

Again YES, it can be done. You ARE able to make the PUSH START BUTTON sentence longer, and also move it to a different position on screen. This required some ROM debugging and disassembling, to study how code loads and manages this data.

I'll go to the conclusions for now, and explain later all the process, because it's a little complicated as it is, for non experienced people. And even if you're a master, you might find it interesting.

Look at the following image. There's a table of data starting at ROM offset 0x628A (where the E7 marked in blue is), that specifies what will be loaded into the name tables, and thus, be shown on screen. Believe it or not, the 5 bytes marked in colour, will tell the game about PUSH START BUTTON sentence, and will let us do our magic. But does this, in a some weird way. Let's see how.

```
00006280 | 99 41 00 E8 C8 C0 05 D0 F4 60 E7 21 E8 A2 L2 C0
00006290 | 2B FA A2 40 35 28 3A A3 04 53 28 3E A3 08 72 28
000062A0 | 4B A3 0E 80 28 59 A3 20 A0 28 79 A3 20 C0 28 99
000062B0 | A3 05 E0 28 9E A3 04 00 29 A2 A3 80 80 29 22 A4
000062C0 | 80 0A 2A A2 A4 13 2A 2A B5 A4 13 58 2A C8 A4 05
000062D0 | 79 2A CD A4 03 89 2A D0 A4 0C B7 2A DC A4 03 C6
000062E0 | 2A DF A4 16 E3 2A F5 A4 19 01 2B 0E A5 0B 21 2B
000062F0 | 19 A5 0B 43 2B 24 A5 05 31 5D C0 64 12 C0 C3 A7
00006300 | B6 C3 12 65 5D C3 C3 B7 A6 12 00 00 00 00 80 AA
```

Blue bytes E7 21 will be interpreted as a VRAM memory 0x21E7, which corresponds to name table 0, and determine the screen position where it will be shown.

Green bytes E8 A2 will be interpreted as a CPU memory address 0xA2E8, which falls inside mapped ROM bank at 0x8000, which is bank 1 at the moment this code is used, as it should be obvious. Why?

We know ROM is divided in chunks of 16Kb that can be mapped for the CPU to be able to access it.

First ROM chunk will be from 0x0000 to 0x3FFF. This will be bank 0.

Second ROM chunk will be from 0x4000 to 0x7FFF. This will be bank 1.

And so on.

If you recall, these coloured bytes are stored in ROM address 0x628A. It falls in bank 1, which starts at 0x4000. And thus, 0x628A has an offset of 0x228A inside bank 1.

Now, 0xA2E8 address, specified by these green bytes, point exactly to the beginning of PUSH string (in yellow), where we were working before.

If you don't see this, read the following.

Remember how to swing from ROM addresses to REAL addresses. I already explained in the first part of this document, so now I'll only refresh the main ideas.

0xA2E8 is a ROM offset. We know ROM is divided in chunks of 16Kb that can be mapped in real address 0x8000, for the CPU to be able to access it. 0xA2E8 has a bank offset of 0x22E8. Because we know it is bank 1 which is mapped at this time, this address corresponds to an offset of 0x22E8 inside bank 1. As bank 1 starts at ROM address 0x4000, final address is 0x62E8.

Now add 0x10, because all NES ROM files stored in computer have a 0x10 added header, for emulators to know which mapper the cartridge uses, etc. This leads us to 0x62F8, where PUSH START BUTTON starts.

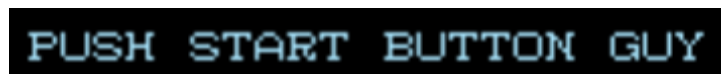
So, the green bytes tells the game where the PUSH STRART BUTTON sentence is stored inside the ROM. So it can be move elsewhere if we want to, with the only

restriction, that it must be in bank 1, that is, inside 4000-7FFF, or the CPU won't be able to access it (remember that bank 1 is mapped at this time, and banks not mapped are not accessible). Of course, bank 7 is always mapped at C000-FFFF, and will do the job too.

Red byte 12 will be interpreted as the number of bytes that will be transferred. Realize that 12 is eighteen in decimal form, which is the maximum number of characters that we were allowed to use for the PUSH START BUTTON sentence.

Great. Isn't it? We can now change the maximum number of characters, and the screen position, by changing the VRAM destination address (which requires knowledge about VRAM memory map).

To illustrate a sentence enlargement, see this.



Below you can see the changes made. Red byte is increased in 3, to accommodate 3 more characters at the end (marked in blue).

```

00006280 | 99 41 00 E8 C8 C0 05 D0 F4 60 E7 21 E8 A2 15 C0
00006290 | 2B FA A2 40 35 28 3A A3 04 53 28 3E A3 08 72 28
000062A0 | 4B A3 0E 80 28 59 A3 20 A0 28 79 A3 20 C0 28 99
000062B0 | A3 05 E0 28 9E A3 04 00 29 A2 A3 80 80 29 22 A4
000062C0 | 80 DA 2A A2 A4 13 2A 2A B5 A4 13 58 2A C8 A4 05
000062D0 | 79 2A CD A4 03 89 2A D0 A4 0C B7 2A DC A4 03 C6
000062E0 | 2A DF A4 16 E3 2A F5 A4 19 01 2B 0E A5 0B 21 2B
000062F0 | 19 A5 0B 43 2B 24 A5 05 31 5D C0 64 12 C0 C3 A7
00006300 | B6 C3 12 65 5D C3 C3 B7 A6 12 B5 5D AF 00 80 AA
  
```

Remember I said that there was a table at 0x628A, starting with this 5 bytes (first, marked in black) that told us about PUSH START BUTTON sentence?

There're much more 5-byte-groups in this table. After the red byte, each 5 bytes do the same we've just seen, but for other elements in the title screen. They tell the game to copy some data, of a certain size, to name table in VRAM. These groups of 5 bytes make up the whole table. And this table determines the arrangement of the title screen.

### Let's go for the title screen

Let's find and change the year under the game title. Ok?

1986 is coded as 01 09 08 06, because those are its pattern table positions.

We find them a little bit under the place we've just been (marked in yellow)

```

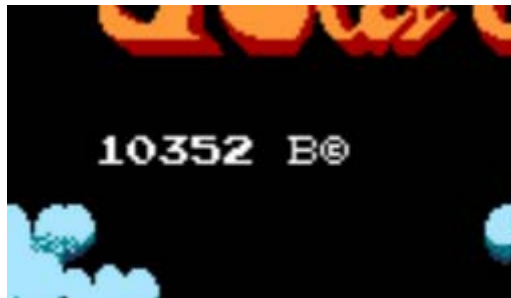
000064E0 | 0A 01 09 08 06 20 21 22 23 24 25 26 1A 1B 1C 1A
000064F0 | 1C 1A 1C 12 12 12 12 12 12 12 12 12 12 12 12 1A
  
```

Marked in green is the symbol of copyright, and in red, string Nintendo.

So, we found the title screen element at offset 0x64E0:



Imagine this game is so great, that it was programmed by gods in year 10352 B.C, while the pyramids of Gizeh were built, and kept in secret until 1986, when Gumpei Yokoi, a great master, decided to make a conversion for Famicom/NES, and show the world pure true power. Just joking. But let's see how it'd look!



Just change,

0A 01 09 08 06 20 21 22 23 24 25 26

Into,

12 01 00 03 05 02 12 65 0A 12 12 12

(I had to use © instead of C because there's no C loaded in the pattern table, as you see)

Now, we can find which of the 5-bytes-group in the table refers to this element, so we can change its position on screen, and its length. How?

We know the element is stored in 0x64E0, so we need to find a 5-bytes-group which CPU address pointer points to this address. First, we need to translate it to a real address.

0x64E0 – 0x10 (from the file header) = 0x64D0

0x64D0 is at bank 1 offset 0x24D0

As bank 1 is mapped at real address 0x8000

0x24D0 offset in real address 0x8000 leads us to a final address of 0xA4D0

Because pointers in 5-bytes-groups are inverted, we need to search for D0 A4. And we've got it, shaded in grey below.

```

00006280 | 99 41 00 E8 C8 C0 05 D0 F4 60 E7 21 E8 A2 15 C0
00006290 | 2B FA A2 40 35 28 3A A3 04 53 28 3E A3 08 72 28
000062A0 | 4B A3 0E 80 28 59 A3 20 A0 28 79 A3 20 C0 28 99
000062B0 | A3 05 E0 28 9E A3 04 00 29 A2 A3 80 80 29 22 A4
000062C0 | 80 DA 2A A2 A4 13 2A 2A B5 A4 13 58 2A C8 A4 05
000062D0 | 79 2A CD A4 03 89 2A D0 A4 0C B7 2A DC A4 03 C6
000062E0 | 2A DF A4 16 E3 2A F5 A4 19 01 2B 0E A5 0B 21 2B
000062F0 | 19 A5 0B 43 2B 24 A5 05 31 5D C0 64 12 C0 C3 A7
00006300 | B6 C3 12 65 5D C3 C3 B7 A6 12 B5 5D AF 00 80 AA
  
```

This element will be copied to VRAM memory address 0x2A89 (which falls in name table 2), took from real address 0xA4D0 (which corresponds to ROM offset 0x64E0 as we've just seen, where the element is stored), and it will \$C in size (that is, twelve, in decimal base).

VRAM memory address 0x2A89 is a "name table 2" address. As you should know, this name table starts at 0x2800 of VRAM memory map. So, the element is copied at offset \$289 of this table. \$20 bytes are required for a complete row of 32 patterns in screen. So, with basic maths, we can get that "C 1986 Nintendo" is displayed at screen position (8, 14)

Knowing this, let's move PUSH START BUTTON sentence below it.



It's enough to move the element PUSH START BUTTON from its original position, to VRAM address 0x2AA5, in its 5-bytes-group.

```
00006280 | 99 41 00 E8 C8 C0 05 D0 F4 60 A5 2A E8 A2 12 C0
```

I won't give more details, as the reader should try its own changes, if he wants to, and I gave a lot of material for him/her to try to understand this.

You can now change the whole title screen :-)

### The palettes in the title screen. The Attrib Table.

When you go and play around enough, to get some experience, you get to funny things like the following:



As you should know, the colouring is due to the attribute tables in VRAM, which select which palette will be used in each 4x4 patterns on screen.

We just found where the attrib data was stored. But, when?

0x630A.

```
000062F0 | 19 A5 0B 43 2B 24 A5 05 | 31 5D C0 64 | 12 CU C3 A7
00006300 | B6 C3 | 12 65 5D C3 C3 B7 A6 | 12 B5 5D AF | 00 80 AA
```

The green bytes are the string “PUSH START BUTTON”

The red bytes are the string “GUY” I added before, remember?

Well, those bytes, initially set to 00, did have a meaning. They are the beginning of the attrib table of the screen where the title “KID ICARUS” is shown (yes, the one you can see in the screenshot above). We used these bytes to enlarge the “PUSH START BUTTON” string, remember? The game does not crash because any value is ok as an attrib table byte.

Even though for those who know a lot about the NES architecture won't need it, let's see why the string shows that colour, and by the way, try to explain how attrib table works.... Damn table.

The attrib table is a group of 64 bytes that comes just after each Name Table in VRAM. As Name Tables are 960 bytes in size, and the Attrib Table 64 bytes, both make up 1024 bytes, or 1Kbyte.

What the Attrib Table does, is to select which palette a group of 2x2 patterns will use. This is hard to explain so let's go with an example to illustrate before a more formal description.

B5 in binary equals to 10 11 01 01

The binary 10 refers to the red 2x2 patterns square where US is.

The binary 11 refers to the red 2x2 patterns square where P is.

The rest four applies to the two squares above, which are not completely seen (because I cut the image there. It's of no interest for us now).



Now let's see the background palette (below). It is divided in 4 sub-palettes of transparent (seen in black below) + 3 colours.

Binary 10 selects the third sub-palette for all patterns drawn inside the red square where US is. You can see that it matches, since the light blue colour is taken from there.

Binary 11 selects the last sub-palette for all patterns drawn inside the red square where P is. You can again see that it matches, since the light brown colour is there.



The next byte, 5D, is treated the same way, for the 8x8 patterns next to the right.

5D in binary equals to 01 01 11 01

First two bits refers to the red 2x2 patterns square where ST is.

Next two bits refers to the red 2x2 patterns square where H is.

Binary 01 selects second sub-palette for these patterns.

In general, each byte of the attrib table, can be interpreted as it follows.

Take the byte and write it as an 8 bit number. Suppose it is abcdefgh, where each letter is a binary 0 or 1. Each two bits refer to a group of 2x2 patterns as follow

gh	ef
cd	ab

Each one of the above squares represents 2x2 patterns, or 16x16 pixels.

This table is quite difficult to understand for a newbie, and makes a bit complicated to design and work with graphics in the NES, because it limits the colours any pattern can use to just 3. And also, does not let you set different palettes to patterns that share the same 2x2 square... hope it is a little clearer now...

Easier to understand than to explain, believe me.

Another important thing to get clear is that, the bytes we overwrite are NOT the Attrib Table! They are copied to the Attrib Table when the Name Table is loaded with the information of the screen. Be careful with not getting this absolutely clear. The Attrib Table is a 64 bytes space after each Name Table in VRAM.

Going back to our trick of overwriting the first 3 bytes of the Attrib Table (yo know what I mean), If we do not place the PUSH START BUTTON string there, we would have no problem overwriting these 3 bytes, or even the fourth. So, again, it's a matter of knowing a lot about what you're doing, and how you can get the most out of something.

## Conclusion

I planned to talk about many other things in this SECOND PART of Kid Icarus reveiled... I also planned this section about text, to be much shorter, but I found many interesting things I wanted to say. This is the reason I'll leave the rest of the stuff for the third part.

Hope you enjoy reading these pages!

David Senabre. 2007-03-19