

Kid Icarus Revealed: An introduction to ROM hacking and NES architecture.

PART 3: Passwords.

By David Senabre Albuje, May 2010.

Introduction

After dropping “Kid Icarus: Erico” development, and being away from Kid Icarus projects for quite long time (that is, my C++ engine, and ROM hacking in general), I recently felt like facing a whole new area in ROM disassembly; the password system. I really had very little information about it, so I start searching for what someone could already have done. Even though I did find some interesting things, I just found like really cracking the whole thing, diving into the algorithm, and writing some kind of software to interpret and generate passwords based on a certain starting condition. A piece of software one could use for things like “give me the password to start from level 2-2, with 500 hearts, 25 feathers, sacred fire arrow, and so on,...”, or “tell me if this password is valid, and what it stands for, and if it isn't valid, make it be”. That would be cool, wouldn't it? For me, it definitely was, so I immediately started working on it.

A word of caution

Following my own style, I'll again try to reach to people who really are not experts. My main concern is to help people who want to learn, but might find some or all of the topic quite confusing. But, at the same time, I'll try to present as much as information as I can, for those who do know as much or more than me myself. This kind of writing is sometime tricky, because you have trouble making both happy. In this particular article, some processes “hard to explain without being inelegant or pedant” appear, and so I'll try to do my best to minimize it. But as I have not much time either, there probably will be many things that could be improved, so please, send me any suggestion, complaint,...

First glance...

The password is composed of a set of possible characters (see image 1):

- . 10 numbers.
- . 26 upper-case letters.
- . 26 lower-case letters.
- . 2 special characters (? and !).



Image 1

There're no hardwired password that I know, and all of them are just passed through the same decrypting process. The password are almost blindly interpreted, as long as a simple condition is met. This condition is some kind of checksum, which I'll explain later on. What this basically means, is that many "valid" password, which pass the checksum check (not terribly hard), just lead to an illegal starting condition, which makes the game engine crash. It seems that the password system has not been programmed to be "playing around with"-safe.

The password decrypting code, and some mind refreshing.

First, I needed to fully understand how the ROM interprets passwords, and how they are validated. Then I could mimic the process in my own software. It was not a difficult task, but as always happens, required quite some time, and a lot of concentration.

The place where decryption starts is at ROM address \$6BD6, which is mapped at NES address \$ABC6.

<<-----

Note: you can skip the following five paragraphs if you understand it, or don't mind about it.

Even though this is explained in the first part of these articles, I'll do a brief description on how this works, for those who don't know how, but still are interested in the password system alone.

The ROM address is one thing, and the NES CPU address is a totally different one. If you open a ROM file, the address of a certain byte is just the position of that byte relative to the beginning of the ROM. That is the ROM address. But the NES CPU doesn't "see" those bytes in a linear fashion.

To convert from ROM address to NES CPU address, first you have to subtract 0x10 bytes from ROM address, belonging to the iNES header, added after dumping, which does not exist in the cartridge ROM. This header contains information such as the mapper used by the game, which is needed by emulators. After subtracting the iNES header size, we need to find the bank that address belongs to.

Kid Icarus uses 16 KBytes banks, and it's composed by 8 banks, and thus, its total size is 128 Kbytes (16KBytes x 8). ROM address \$6BD6 is on ROM bank 1 (the second one), which goes from address \$4000 to \$7FFF (the first one goes from \$0000 to \$3FFF, and contain just graphical data). The NES memory map has only two bank slots. This means that the CPU can only "see" two ROM banks at any given time. One of them is "seen" at address \$8000, and the other one at \$C000. To be able to access any part of the ROM, a bank switch method is used. In other words, a bank replaces another. As Kid Icarus permanently maps ROM bank 7 to the second bank slot (\$C000), only the first slot (\$8000) is available for mapping the remaining seven banks.

Even though bank 7 starts at ROM address \$1C000, it appears at NES CPU address \$C000. And, when bank 1, containing the password decrypting algorithms, is mapped, it's starting address, \$4000, appears at NES CPU address \$8000.

Hence,

$$\$6BD6 - 0x10 = \$6BC6 - \$4000 = \$2BC6 + \$8000 = \$ABC6.$$

ROM address \$6BD6 is at NES CPU address \$ABC6.

Or more generally,

$$\text{ROM address} - \text{iNES header size} = \text{Raw ROM address} - (\text{bank} * \$4000) = \text{Bank offset address} + \text{ROM Slot base address} = \text{NES CPU address}.$$

----->>

When the execution reaches \$ABC6, three subroutines, which are the core of the password decrypting, are called one after the other, before the game is loaded with certain parameters (strength, hearts, score, stage level,...), given that the user entered a valid password, of course.

```
-----  
00:ABC6:20 D0 AB JSR $ABD0  
00:ABC9:20 93 AC JSR $AC93  
00:ABCC:20 36 AC JSR $AC36  
00:ABCF:60      RTS  
-----
```

Code snap 1

The three main subroutines which I mentioned just a moment ago, are located at addresses \$ABD0, \$AC93 and \$AC36. I'll talk about what they do in a moment.

When the user enters a password, it is stored in 24 bytes starting from address \$6031 (in other words, in range \$6031...\$6048, inclusive). One byte is dedicated for each of the 24 characters, and the value used to store them is a simple incremental pattern. The numbers match the internal value, that is, character '3' is assigned to value 3. 'A' is value 10, 'B' is value 11,... and then 'a' is value 36. Finally, the last two symbols are value 62 ('?') and 63 ('!'). As there are 64 possible characters, all values from 0 to 63 are used. This number is, of course, not a coincidence. With 64 possible value, 6 bits can be used to represent any of these characters. In other words, a 6 bit value can go from 0 to 63 (supposing unsigned arithmetic).

There are another two memory ranges used for password decrypting. Both of them are initialized to zero before starting to read the password from RAM.

They are:

\$600D...\$6030

\$6061...\$6072

Interesting to note, all these addresses don't belong to the internal console's RAM (which occupies addresses \$0000-\$07FF), but to an expansion RAM located inside the cartridge. This chip is absolutely necessary, and it is used continuously throughout the game, for many purposes.

Subroutine \$ABD0 does some initialization, and “compresses” password from the 24 bytes at \$6031, to 18 bytes at \$6061. This is possible because those 24 bytes really contained a 6 bit value, and so, the uppermost 2 bits were unused. The password's real size in bits is $24 \times 6 = 144$ bits. If you use the whole 8 bits of a byte, instead of just 6 bits, this can be stored in $144 / 8 = 18$ bytes. And this is exactly what is done here. For this purpose, a series of rotation are employed. The exact process is not totally clear at a first sight, when plainly looking at the disassembled code, but can be understand easily with some observation, or aided by a debugging-capable emulator.

At the end of the execution, range \$6061...\$6071 is filled with the “compressed” version of the password.

Subroutine \$AC93 fills range \$600D..\$602A, from the “compressed” password stored in \$6061. Here is where the game really gives meaning to each segment of the password.

Subroutine \$AC36 validates the password. Well, actually it calculates its checksum, and checks it against the valid one. So, before it returns, it makes this comparison, and after returning, a conditional branch is what actually validates it. This happens at NES address \$A19B.

```

-----
00:A198:20 C6 AB JSR $ABC6
00:A19B:D0 03 BNE $A1A0
00:A19D:4C 09 A1 JMP $A109
00:A1A0:20 4D AC JSR $AC4D
-----

```

Code snap 2

If checksum was wrong, the program branches to \$AC4D, and the password is rejected. If checksum matches, the “BNE” branch isn't taken, and “JMP” to \$A109 is taken. This is the place where game data is initialized, setting it ready to start from a certain initial condition.

But more on that later on. So far, let's stay a little while in subroutine \$AC93.

But before going on, summarizing, the three address ranges used for password decrypting are:
 \$6031...\$6048 - “character” password range.
 \$6061...\$6072 - “compressed” password range.
 \$600D...\$6030 - “stats” data range.

Example for newbies.

You might skip this also if you have some knowledge on computer science or digital electronic, but I'd like to throw some light on those who just feel curiosity, and have no background.

For example, the value written to \$601E is taken directly from address \$606B. At this point this might seem meaningless. But wait. Later on, \$601E will be copied to internal RAM address \$14F. This byte holds the number of mallets that Pit carries. This can be easily confirmed by using the cheat capability of an emulator (just write any value lower than 100, and check the number of mallets in the pause screen). In short, the eleventh position in the “compressed” password determines the starting number of mallets with which Pit will start the game. Due to the bit “compression”, is not straightforward to see which of the password character is responsible for this. Thinking a little bit, you get the answer. Two characters contribute to this value, fourteenth and fifteenth; in other words, the second and third character in the third group of characters.

But not any change in these chars will lead to a different value of mallets, because, again, of the compression applied to it. The 8 bits that represent the mallets starting value are taken from the upper 4 bits of the fourteenth char, and the 4 lower bits of the fifteenth char (see the *table 1* below, that may help). Remember that password char use only 6 bits. What this means is that a fourteenth char 'a', 'b', 'c' or 'd' led to the same mallets, as they share the same upper 4 bits, and change only in the lower 2 (see the *table 2* below, that may help). But a fourteenth char 'e', 'f', 'g' or 'h' will led to one more mallet. Following the same line, fifteenth char '1' instead of '0' will add 16 mallets.

B10 - b7	B10 - b6	B10 - b5	B10 - b4	B10 - b3	B10 - b2	B10 - b1	B10 - b0
C14 - b3	C14 - b2	C14 - b1	C14 - b0	C13 - b5	C13 - b4	C13 - b3	C13 - b2

Table 1.

Mallets starting value source.
 Bxx – compressed value byte xx
 Cxx – password character xx
 bx – bit of character Cxx.

Char	'a'	'b'	'c'	'd'	'e'	'f'	'g'	'h'	'0'	'1'
Value (decimal)	36	37	38	39	40	41	42	43	0	1
Value (binary)	100100	100101	100110	100111	101000	101001	101010	101011	000000	000001

Table 2.

Just to make sure everybody gets the point, before going on, I'll detail this example down to the bits.



Image 2

The following password is a valid one, and grants no mallets (see image 3).

CW01!! !!!!! 10m!!v E0!3e!



Image 3

The following password is a valid one, and grants 9 mallets.

See that characters 14th and 15th changed. Cool, eh? Forget about the last two characters for now.

CW01!! !!!!! 1a0!!v E0!381

Let's go with this last one. Take the 14th and 15th characters. 'a' and '0'.

Binary value for 'a' is 100100.

Binary value for '0' is 000000.

According to *table 1*, the 8 bit value for mallets is built from 14th char ('a') bits 2 to 5, that is, the four upper most bits ('1001'), and then, from 15th char ('0') bits 0 to 3, that is, the lower four bits ('0000'). The result is '0000 1001'. This is value 9.

If you enter 'b' instead of 'a', mallets don't change. The four upper bits remain the same.
 But if you enter 'e' instead of 'a', you get 10 mallets.
 CW01!! !!!!! 1e0!!v E0!3O1
 The result is '0000 1010'. This is value 10.

What happens with the 15th char is that it weights 16 times more. An increase of 1 in value, translates into an increase of 16 mallets. Let's (try to) clarify this.
 If you enter '1' instead of '0', you get 25 mallets (16 mallets more than before).
 Binary value for 'a' is 100100.
 Binary value for '1' is 000001.
 CW01!! !!!!! 1a1!!v E0!385
 The result is '0001 1001'. This is value 25 (see image 4).



Image 4

You may have noticed that the last two characters also changed in every one of the passwords. They do change because they hold the checksum, which validates the password. If these two chars are not those exact ones, the game will identify the password as “invalid”, and will refuse to load it.

More on initial stats matching

Not all assignments in subroutine \$AC93 are as simple as for mallets. Mallets, whose value is loaded from \$601E, is taken from one of the bytes of the “compressed” password, \$606B to be exact.

But what happens for the score, the hearts or the credit card debt ? These values can be greater than 255, and thus more than 8 bits are required for storing them. The Kid Icarus password system determines some absolute maximum values, that is 4095 for hearts and credit card debt, and 16.777.215 for the score. In other words, it uses 12 bits for hearts and debt, and 24 bits for the score.

The credit card has only two possible values. Or you have it, or you don't have it. This value is loaded from \$601D (just one byte before the mallets in the “compressed” password), which is taken from the less significant bit of \$606A.
 For someone with some programming knowledge, the operation would be:

```
(char*) (0x601D) = (char*) (0x606A) & 0x01;
credit_card = (char*) (0x601D);
```

which I will simplify as:

credit_card = 0x601D = 0x606A & 0x01

Now, the full list is:

```
shield modifier           = $600f = ($6063 & 0xF3)
3rd sacred weapon        = $6012 = ($6063 & 0x0C) >> 2
bow modifier             = $600e = ($6062 & 0xF3)
2nd sacred weapon        = $6011 = ($6062 & 0x0C) >> 2
fire arrow modifier      = $600d = ($6061 & 0xF3)
1st sacred weapon        = $6010 = ($6061 & 0x0C) >> 2
Score (byte 0)           = $6013 = $6064
Score (byte 1)           = $6014 = $6065
Score (byte 2)           = $6015 = $6066
Hearts (byte 0)          = $6019 = $6067
Hearts (byte 1)          = $601a = $6068 & 0x0F
Debt (byte 1)            = $601c = $6068 & 0xF0 >> 4
Debt (byte 0)            = $601b = $6069
Credit card              = $601d = $606a & 0x01
Mallets                  = $601e = $606b
Feathers                  = $601f = $606c
Barrel/Bottles           = $6020 = $606d
Pit strength             = $6021 = $606e & 0xF0 >> 4
Health bars - 1          = $6026 = $606e & 0x0F
1st chest                 = $6022 = $606f & 0x03
2nd chest                 = $6023 = $606f & 0x0C >> 2
3rd chest                 = $6024 = $606f & 0x30 >> 4
Centurions               = $6025 = $6070
Starting scene            = $6028 = $6071 & 0xF0 >> 4
Start level - 1          = $6029 = $6071 & 0x07
                          = $602a = $6071 & 0x08 >> 3
```

After loading range \$600d..\$602a, these values are finally copied to the internal RAM addresses which really hold the real values during the game. This last transformation is:

```
$6029 - $130 level number - 1          (0-7)
$6028 - $a0 game scene                  (0-15)
$6026 - $aa and $170 (pit's health bars - 1) (0-15)
$6025 - $156 centurions                  (0-255)
$6024 - $155 3rd chest                    (0-3)
$6023 - $154 2nd chest                    (0-3)
$6022 - $153 1st chest                    (0-3)
$6021 - $152 pit strength                  (0-15)
$6020 - $151 barrel/Bottles                (0-255)
$601f - $150 feathers                      (0-255)
$601e - $14f mallets                      (0-255)
$601d - $14e credit card                  (0-1)
$601c - $14d debt                          (0-15)
$601b - $14c debt                          (0-255)
$601a - $14b hearts                        (0-15)
$6019 - $14a hearts                        (0-255)
$6015 - $146 total score                    (0-255)
$6014 - $145 total score                    (0-255)
$6013 - $144 total score                    (0-255)
$6012 - $143 3rd sacred weapon              (0-3)
$6011 - $142 2nd sacred weapon              (0-3)
$6010 - $141 1st sacred weapon              (0-3)
$600f - $140 sacred shield modifier        (0-243, bit 2 and 3 always 0)
$600e - $13f sacred bow modifier            (0-243, bit 2 and 3 always 0)
$600d - $13e sacred fire arrow modifier    (0-243, bit 2 and 3 always 0)
```

Notes:

More on “game scene” concept later on (“checksum checking” section).
More on “sacred weapon modifiers” later on.

In this subroutine, a loop copies 8 bytes from \$6005..\$600c to \$120..\$127, which purpose at the present moment is still unknown to me.

Another loop really initializes almost all of the stats, except starting level and pit level. It copies from \$6025 to \$156 and \$16f, and then descending to \$600d, which is copied to \$13e and \$157 (“state tables”, more on this right now).

Keep in mind that level number cannot be greater than 3 to be valid for the game (0..2 for stages, and 3 for fortress).

Health bars value cannot be greater than 4. No problem if it is, but no effect. Maximum health bars value seems to be 5.

It seems that \$6027 is not used at all.

Bytes \$6016 and \$6018 are copied to the “state table” (\$13e), but they seem to have no meaning, because no value is given to them.

State Table

Address \$13e is quite important in the game. So is \$157. In fact, there are lots of references throughout the code to these addresses together. But, what is really happening here?

Starting from \$13e, there's a 24 byte table, holding most of the game stats. I call it “State Table”. If you played the game, you'll know that whenever Pit is killed, a password is generated, and game starts again from the beginning of that level, but with the items and stats you started with, the first time you reached there. If you use 2 feathers and 1 bottle, and later on you die, you start the level as if those were never been used. Although I did not rigorously checked it, address \$157 holds a backup of the State Table. When Pit dies, its content is copied back to State Table at \$13e, which is, so call it, the “active” table. The back up State Table at \$157 is updated when you first get to a new level.

Example password processing

Let's see how the famous password “8uuuuu uuuuuu uuuuuu uuuuuu” would be processed.

The password characters are stored in \$6031.
Remember, value for '8' is 8, and value for 'u' is 56 (or 0x38 in hex).
You should be able to read the password below now :-)

```
$6031..$6048  
08 38 38 38 38 38 38 38 38 38 38 38 38  
38 38 38 38 38 38 38 38 38 38 38 38 38
```

The 24 byte password is “compressed” into 18 bytes to address \$6061:

```
$6061..$6072  
08 8E E3 38 8E E3 38 8E E3 38 8E E3 38 8E E3 38 8E E3
```

The initial stats is decrypted from the “compressed” password. The result is:

```
$600d..$602a  
00 82 E3 02 03 00 38 8E E3 00 00 00 38 0E E3  
08 00 8E E3 38 08 03 00 02 38 0E 00 08 06 01
```

Only checksum check is left. Of course, as this password is valid, it passes the check. We'll see that in the next section. As an advance for impatient people, the checksum value is 0xE3.

Checksum checking

The password's last two characters build up the checksum value, and the start “game scene”, that is how I call a value which takes the following values (*table 3*)

Value	Corresponding level
0	Initial screen
1	Main menu
2	First world stage
3	First fortress
4	Second world stage
5	Second fortress
6	Third world stage
7	Third fortress
8	Last stage
9	Ending
> 9	Crash :-)

Table 3.
“Game scenes”.

So, how is this checksum value used to validate the password? Well, in nowadays it is such a simple and plain process. Basically you just start up adding up the value of every “compressed” password byte, with the exception of the last one, which is the checksum value (built from the last two password characters). The carry has to be taken into account, and all bits above the eighth are discarded. You end up with an 8 bit value, which must be equal to the last one, not included in the addition. If they match, valid password. If they don't, no luck, try again.

Basically the last byte, or checksum value, is telling us what the addition of the rest of the bytes must be. It is meant to be a proof of intentionality, but of course you can just try and try until you get a valid one. Not very hard.

The problem (not really a problem, though) with this, is that many possible values can lead to a game crash, if “game scene” doesn't have a valid value, that is, refers to a non existing level in the game.

Now, the last character is part of the checksum, but only 2 bits from the penultimate character contribute to it. To be exact, its 2 uppermost bits. This means that all penultimate byte values from 0x0 to 0xF will have the same checksum, as so will values from 0x10 to 0x1F. In fact, there are only 4 possible checksum variations due to this penultimate byte.

Values:

0 - F = 0
 10 - 1F = 1
 20 - 2F = 2
 30 - 3F = 3

Which correspond to characters:

'0' - 'F' = 0
 'G' - 'V' = 1
 'W' - 'l' = 2
 'm' - '!' = 3

The rest of the 6 bits are taken directly from the last character, to build up the checksum.

B17 - b7	B17 - b6	B17 - b5	B17 - b4	B17 - b3	B17 - b2	B17 - b1	B17 - b0
C23 - b3	C23 - b2	C23 - b1	C23 - b0	C23 - b5	C23 - b4	C22 - b5	C22 - b4

Table 4.

Checksum value

Bxx – compressed value byte xx

Cxx – password character xx

bx – bit of character Cxx.

Sacred weapons are a bit more complicated.

Sacred weapon modifiers are an important part of the game-play which takes account of some particularities about them. For example, Pit can't use some weapons until its health gets higher than certain point. These weapons are inactive, and under normal conditions, appear grayed in the inventory (see image 5).



Image 5

While not in much detail, and probably being not accurate, this is what I have clear up to now.

An active sacred weapon is represented by value 0x01 (fire arrow), 0x02 (bow) or 0x03 (shield) in the corresponding position slot, whose addresses are \$141 (slot 1), \$142 (slot 2) and \$143 (slot 3). A 0x00 means no sacred weapon in that slot. Have a look at image 6 to see what I mean by “slot” if you're not sure. A slot can have none (be empty), or any of the three sacred weapons.

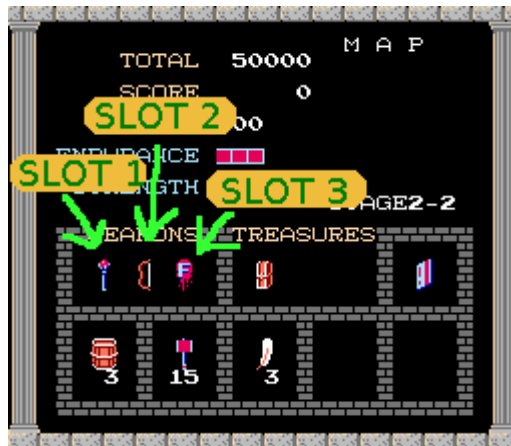


Image 6

During normal game-play, the game always grants sacred weapons in order, that is, when you get the first weapon, it goes into first slot, when you get a second weapon, it goes to the next slot, and so on. But, when you generate passwords, you can get any combination possible, even having the same sacred weapon in all the three slots!

Now, each weapon has its on modifier. When the modifier is zero, the affected weapon is disabled by writing value 0x90 (fire arrow), 0xA0 (bow) or 0xB0 (shield) to its corresponding address, depending on the position it is placed in (\$141..\$143). Disabled weapons appear to be missing or grayed in the inventory.

The code that restores weapons to 0x90, 0xA0 o 0xB0 when their modifier is zero, is located near address \$E48F, and this is checked continuously. The following code shows how the modifier is loaded into register A, and if this value is zero, branches to \$E4A1 with a “BEQ” instruction. Anyhow, I'll omit details, to not confuse too much.

```
-----
00:E48F:BD 3E 01 LDA $013E,X
00:E492:F0 0D BEQ $E4A1
00:E494:10 2E BPL $E4C4
00:E496:30 09 BMI $E4A1
-----
```

Code snap 3

If the modifier has a positive value (0x00 to 0x7F), and Pit does have an active sacred weapon, then it seems nothing is done to them.

```

-----
00:E4C4:E8          INX
00:E4C5:86 01      STX $0001
00:E4C7:CA          DEX
00:E4C8:BD 65 E5   LDA $E565,X
00:E4CB:85 02      STA $0002
00:E4CD:A0 00      LDY #$00
00:E4CF:B9 41 01   LDA $0141,Y
00:E4D2:C5 01      CMP $0001
00:E4D4:F0 25      BEQ $E4FB
00:E4D6:29 F0      AND #$F0
00:E4D8:C5 02      CMP $0002
00:E4DA:F0 12      BEQ $E4EE
00:E4DC:C8          INY
00:E4DD:C0 03      CPY #$03
00:E4DF:90 EE      BCC $E4CF
[... ]
00:E4EE:B9 41 01   LDA $0141,Y @ $0142 = #$02
00:E4F1:29 0F      AND #$0F
00:E4F3:9D 3E 01   STA $013E,X @ $0140 = #$02
00:E4F6:A5 01      LDA $0001 = #$02
00:E4F8:99 41 01   STA $0141,Y @ $0142 = #$02
00:E4FB:60          RTS
-----

```

Code snap 4

As you can see by the “BMI” instruction at \$E496 The modifiers can have a negative value too (x080 to 0xFF), and that triggers another set of actions. But I can't go into much detail now. And also I think this has come dense enough.

But there is still one more thing I'd like to note. And this is quite relevant. Not all positive modifier values are the same. Bit 1 has a great significance. \$A5 holds which weapons are really active and fully working. From LSB, one bit is dedicated to fire arrow, bow and shield. If '0', weapon can't be used (disabled, non active). If '1', weapon is being used (enabled, active). If bit 1 of sacred weapon modifier is set, then weapon is activated in \$A5. If bit 1 is cleared, then weapon is disabled in \$A5. The code:

```

-----
00:C46D:B9 3E 01   LDA $013E,Y
00:C470:30 0C      BMI $C47E
00:C472:29 02      AND #$02
00:C474:F0 08      BEQ $C47E
00:C476:A5 A5      LDA $00A5
00:C478:19 C4 C4   ORA $C4C4,Y
00:C47B:85 A5      STA $00A5
00:C47D:60          RTS
00:C47E:A5 A5      LDA $00A5
00:C480:39 C7 C4   AND $C4C7,Y
00:C483:85 A5      STA $00A5
00:C485:60          RTS
-----

```

\$C4C4 holds a 3 byte table: 0x01, 0x02 and 0x04.
 \$C4C7 holds a 3 byte table: 0xFE, 0xFD and 0xFB.

Code snap 5

Another interesting code is checking weather Pit has enough health to activate all its weapons. The code is shown next, and it is called three times with Y=0, Y=1 and Y=2. For the first weapon to be activated, you need two full health bars, or 15 health points, or \$A6=0x0F. If you meet this condition, branch to \$C4A3 is taken, and \$A5 bit 1 is set (see address \$C4AF), given that you actually have any weapon in that slot (checked at address \$C4A6).

```
00:C486:A5 A6      LDA $00A6
00:C488:D9 CA C4   CMP $C4CA,Y
00:C48B:B0 16      BCS $C4A3
[... ]
00:C4A3:20 B5 C4   JSR $C4B5
00:C4A6:F0 0C      BEQ $C4B4
00:C4A8:AA         TAX
00:C4A9:CA         DEX
00:C4AA:BD 3E 01   LDA $013E,X
00:C4AD:29 80      AND #$80
00:C4AF:09 02      ORA #$02
00:C4B1:9D 3E 01   STA $013E,X
00:C4B4:60         RTS
00:C4B5:B9 41 01   LDA $0141,Y
00:C4B8:10 07      BPL $C4C1
00:C4BA:29 30      AND #$30
00:C4BC:4A         LSR
00:C4BD:4A         LSR
00:C4BE:4A         LSR
00:C4BF:4A         LSR
00:C4C0:60         RTS
00:C4C1:29 03      AND #$03
00:C4C3:60         RTS
```

\$C4C1 holds a 3 byte table: 0x0F, 0x17 and 0x1F.

Code snap 6

Basically, if you generate a password, you want all your sacred weapons to have a modifier of, let's say, 0x02, so you can use the weapons right away.

If you set modifier to 0x01, then there may be some weapons you won't be able to use until you get full 3 and 4 health bars.

Grayed. Before activation condition.

```
$13E: 0x00, 0x00, 0x02
$141: 0x03, 0x02, 0x01
$A5: 0x04
```

When full 4 health bars, bow and shield are both active (see image 7):

```
$13E: 0x02, 0x02, 0x02
$141: 0x03, 0x02, 0x01
$A5: 0x07
```

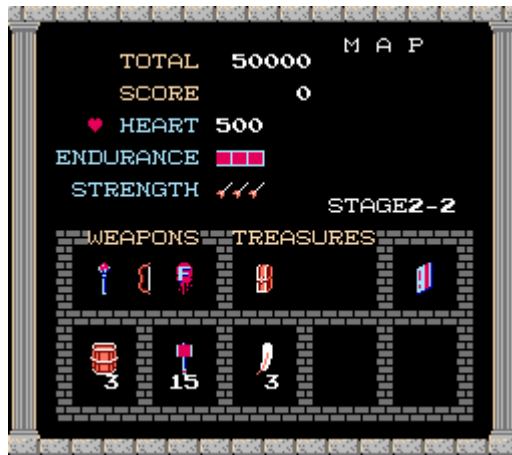


Image 7

Additionally, if you set modifier to 0x00, inactive weapons won't show up grayed in the inventory, but missing (see image 8). Nonetheless, when you reach to full 3 or 4 health bars (activation condition), weapons will come up as if you had initially set modifier to 0x01, because information about owned weapon will be stored in the modifier value, in the upper nibble.



Image 8

Missing. Before activation condition.

\$13E: 0x00, 0x00, 0x02

\$141: 0x03, 0xA0, 0x90

\$A5: 0x04

When full 3 health bars, bow is activated:

\$13E: 0x00, 0x02, 0x02

\$141: 0x03, 0x02, 0x90

\$A5: 0x06

When full 4 health bars, shield is activated:

\$13E: 0x02, 0x02, 0x02

\$141: 0x03, 0x02, 0x01

\$A5: 0x07

Centurions don't work!

Even though there is a field for centurions, and passwords can hold certain values for them, you'll always start with none. This is not a bug of my password generator tool (given that you are using it). In fact, the centurions RAM address is correctly loaded with the specified value during the password decrypting, but later on, a nice instruction clears it to zero. For all of the three fortresses, this is done inside subroutine \$92BD (while bank 5 (sixth) is mapped), called from \$8137.

At \$92E5 there's a "STA \$156" which is guilty of this behavior. For curiosity's sake, the same is done with address \$A5, which disables all sacred weapons, and increments address \$3A from 0x10 to 0x11. This last address is always higher than 0x10 while inside a fortress, a 0x09 inside a shop, 0x00 in a regular stage... but I'm not sure what its exact purpose is.

Of course a simple ROM patch could solve the "problem" with centurions. Just replace instruction at \$92E5 by NOPs, and then you can start with many centurions using a cool password (generated by my utility, for example).

Of course this doesn't make much sense, as Pit is supposed to free centurions from statues, that reside inside fortresses. If you enter the fortress having already 50 centurions, it isn't very real... but... does it need to be real anyway? :-)

Password generation when Pit dies

When Pit dies, the same addresses that were used to decrypt the password are reused to generate the password that is going to be given to the player (see image 9).

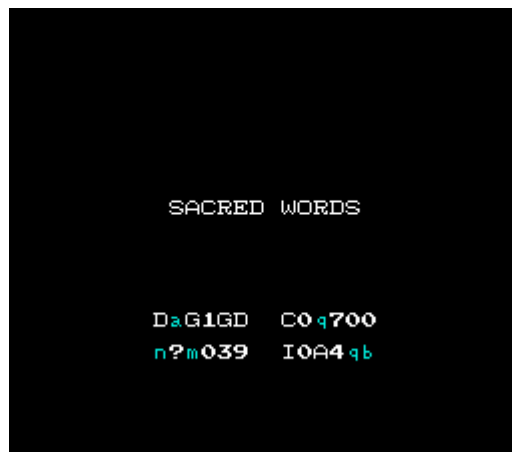


Image 9

When you are killed, range \$6031-\$6048 is cleared to zero by instructions at address \$C8B5. The program flow gets here from address \$C8A1, by means of a "JSR \$C8B1" instruction. At the same time, the program flow reached here from \$C2B1. Well... enough backtracing...

Afterward, code at \$C98A generates the sacred weapon's bytes (0x6061, 0x6062 y 0x6063), and then all the rest of stats.

Later, code at \$C975 generates the checksum.

Then, rotations are performed to "uncompress" the password into "characters" passwords, ready to be presented in the screen to the player. This task is done by code near address \$C902.

Conclusion

There're still some mysteries, but ROM hacking almost always will have some kind of mystery or unknown. It is quite a huge amount of work, and not easy to get to a global overview of the whole code. I'm yet far away from fully understanding everything.

The third part was supposed to be about enemies, but that will take on the fourth, which I hope it gets out somewhere this year. Suggestions are welcome. *David Senabre Albuje, 2010*