Dragon Warrior SRAM Guide

by jdratlif Updated to v1.0 on Feb 23, 2007

```
--------------------------------------------------------------------------------
| Dragon Warrior (NES) SRAM Document 1.0
| by John David Ratliff
|
| The most recent version of this guide can always be found at
| http://games.technoplaza.net/dwsrame/sram-doc.txt
|
| Copyright (C) 2007 emuWorks (http://games.technoplaza.net/)
|   Permission is granted to copy, distribute and/or modify this document
|   under the terms of the GNU Free Documentation License, Version 1.2
|   or any later version published by the Free Software Foundation;
|   with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
|   Texts.  A copy of the license can be found at
|   http://www.gnu.org/licenses/fdl.html
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
| Table of Contents
--------------------------------------------------------------------------------

  - 1.0 Introduction
  - 2.0 Copyright Notice
  - 3.0 Revision History
  - 4.0 The Dragon Warrior SRAM
    - 4.1 SRAM Basics
    - 4.2 SRAM Offsets
    - 4.3 The Sanity Algorithm
    - 4.4 Checksum Bypass Using a Game Genie
  - 5.0 dwsrame - The Dragon Warrior (NES) SRAM Editor
  - 6.0 Contact Information

--------------------------------------------------------------------------------
| 1.0 Introduction
--------------------------------------------------------------------------------

  This document is a guide to the SRAM format used by Dragon Warrior for the
  original Nintendo (NES). SRAM (short for save random access memory, and has
  many other names and acronyms) was the format for saving data in most NES
  cartridge games including Dragon Warrior.

  SRAM was an area of memory that was supplied power by an internal cartridge
  battery. It's purpose was to preserve information even when the game was not
  running. In this manner, game progress could be preserved even when the system
  was not active.

  SRAM is not the same as emulator save states. SRAM was internal to the game
  cartridge and thus the format is applicable to any NES emulator (and probably
  all NES copiers as well).

  This document aims to discuss the data format used in SRAM to store game
  progress.

  There have been many adaptations, remakes, and translations of Dragon Warrior
  over the years. To be clear, this guide covers the SRAM used by the original
```

Dragon Warrior for the NES (the original English translation of the original Dragon Quest for the Famicom) released by Enix in 1989.

2.0 Copyright Notice

3.0 Revision History

Version 1.0 - Friday, February 2, 2007
- First Public Release

4.0 Dragon Warrior SRAM

This section details the SRAM format used by Dragon Warrior (NES).

4.1 SRAM Basics

The Dragon Warrior cartridge had an SRAM of 0x2000 (8192 decimal) bytes. This was the standard for NES cartridges of the time.

There were three possible save slots available in the game. Each of these slots used 0x140 bytes of SRAM, though in reality it is actually just 0x20 bytes repeated 7 times.

Because the SRAM was battery backed, and the battery will eventually die, the game also stored certain data to act as a check on the sanity of the preserved data. Two checks were used. The first was the data replication. The 0x20 bytes of game data are replicated 7 times. Secondly, the 0x20 bytes of data also have some embedded sanity data, which I will call the checksum.

Finally, there are 4 bytes in the SRAM that determines what save slots are in use.

The three games begin at offset 0x68 from the start of the SRAM. From there, the three game's 0x140 bytes of data are stored sequentially. The 4 bytes of save slot usage start at offset 0x35 from the start of SRAM.

4.2 SRAM Offsets

The following are the known offsets within the game data. They will be presented relative to the start of a particular game slot rather than from the start of SRAM. This means an offset of 0 for the first game is actually at the SRAM position 0x68. For the duplication, this means it is also as 0x88, 0xA8, and so on for all 6 duplicate sets.

00-01: Hero's Experience
02-03: Hero's Gold

Both of these values are little-endian 16-bit unsigned values. This means they can range from 0 - 65535. It also means the bytes when viewed as hex should be reversed. ex: 34245 decimal = 85C5 hex = C585 little-endian hex.

04-07: Hero's Inventory

The hero's inventory is comprised of at most 8 items. Because of the limited number of possible items, each item can be represented in a half-byte (4 bits). This means each byte can represent two numbers. The item values are as follows:

0: No Item
1: Torch
2: Fairy Water
3: Wings
4: Dragon's Scale
5: Fairy Flute
6: Fighter's Ring
7: Erdrick's Token
8: Gwaelin's Love
9: Cursed Belt
A: Silver Harp
B: Death Necklace
C: Stones of Sunlight
D: Staff of Rain
E: Rainbow Drop
F: (unused) displays current number of Herbs, but you cannot use this

So, if 04 had the value E3, the hero will have the Rainbow Drop and a set of Wings in his inventory.

08: Hero's Keys
09: Hero's Herbs

These two bytes tell how many of these particular items the hero has. Valid values are 0 - 6. Using values outside the valid range is untested.

0A: Equipment Byte

The equipment byte detemines the hero's weaponry. To decide what the hero is equipped with, you simply need to add the possible values together.

Weapons:
- Bamboo Pole: 0x20
- Club: 0x40
- Copper Sword: 0x60
- Hand Axe: 0x80
- Broad Sword: 0xA0
- Flame Sword: 0xC0
- Erdrick's Sword: 0xE0

```
  Armor:
    Clothes:          0x04
    Leather Armor:    0x08
    Chain Mail:       0x0C
    Half Plate Armor: 0x10
    Full Plate Armor: 0x14
    Magic Armor:      0x18
    Erdrick's Armor:  0x1C

  Shields:
    Small Shield:     0x01
    Large Shield:     0x02
    Silver Shield:    0x03

  So, if you want the best equipment possible, add the value of Erdrick's
  Sword (0xE0), Erdrick's Armor (0x1C), and the Silver Shield (0x3) to get an
  equipment byte value of 0xFF.

0B: Quest byte 1
0C: Quest byte 2
0D: Quest byte 3

  Quest bytes determine whether certain things have happened in the game or
  not. Things not covered by the quest bytes are usually determined by the
  hero's inventory. For example, if you have the staff of rain, the old man
  who gives it to you will tell you to go away.

  The quest bytes use individual bits to determine various quest markers. The
  bits not documented are not known to do anything.

  Byte 1:
    bit 2 - Charlock Hidden Stairs Revealed
    bit 3 - Rainbow Bridge Built
    bit 4 - Wearing Dragon's Scale
    bit 5 - Wearing Fighter's Ring
    bit 6 - Wearing Cursed Belt
    bit 7 - Wearing Death Necklace

  Byte 2:
    bit 0 - Gwaelin Rescued (in Hero's arms)
    bit 1 - Gwealin on Throne (bits 1 and 0 should be mutually exclusive)
    bit 3 - Started Quest

  Byte 3:
    bit 1 - Golem Killed
    bit 2 - Dragonlord Killed
    bit 6 - Green Dragon Guarding Princess Gwaelin Killed

0E-15: Hero's Name

  The hero's name is stored in a very odd manner. The first four characters
  0E-11 are the first four characters of the name, but they are stored in
  reverse order. In other words, if your characters name is 'Loki', it will
  be stored ikoL.

  The second four characters 12-15 are the second four characters of the
  name, and they too are stored in reverse order.

  Here are the values for the Dragon Warrior alphabet.
```

```
    0x00 - 0x09 : The numbers 0-9
    0x0A - 0x23 : The lowercase letters 'a' - 'z'
    0x24 - 0x3D : The uppercase letters 'A' - 'Z'
    0x40        : The apostrophe ' character
    0x47        : The period . character
    0x48        : The comma , character
    0x49        : The dash - character
    0x4B        : The question mark ? character
    0x4C        : The exclamation point ! character
    0x4E        : The close paren ) character
    0x4F        : The open paren ( character
    0x60        : The space ' ' character

    These are the only valid characters accepted by the name input screen.
    Using other values may produce other symbols in the name, though this is
    untested.

  16: Message Speed

    This offset determines the message speed for the game. Valid values are 0
    for fast, 1 for normal, and 2 for slow.

  17: Hero's HP
  18: Hero's MP

    The HP and MP are the current HP and MP for the hero. The valid range is
    0 - 255.

  19:    (unused) always AB
  1A-1D: (unused) always C8

  1E-1F: Checksum

    The checksum is a two-byte value. It will be described in the next section.

  Finally, the four bytes that determine game slot usage are as follows:

  0x35, 0x36, and 0x37 - 0 if the slot is unused, C8 if it is
  0x38 - Three bits determine game usage
    bit 0 - slot 1
    bit 1 - slot 2
    bit 2 - slot 3

  So, if all three slots are in use, you will see 0xC8 0xC8 0xC8 0x07 for these
  four bytes. These offsets are relative to the start of SRAM, not the start of
  game data.

-------------------------------------------------------------------------------
| 4.3 The Sanity Algorithm
-------------------------------------------------------------------------------

  As mentioned earlier, the game needs to ensure the data stored in the SRAM is
  still valid. To do this, it uses data replication and a checksum word. This
  word must be generated by us if you expect the game to accept modified data.

  Here is the sanity algorithm, ripped from the Dragon Warrior ROM in 6502
  assembly. I have added comments at the end of each line.

  $FBEF:A0 1D     LDY #$1D        ; load counter with 0x1D
  $FBF1:84 94     STY $0094       ; init checksum low byte
```

```
$FBF3:84 95     STY $0095       ; init checksum high byte
$FBF5:B1 22     LDA ($22),Y     ; load data[counter] into a
$FBF7:85 3C     STA $003C       ; store to memory
$FBF9:20 2A FC  JSR $FC2A       ; jump to subroutine
$FBFC:88        DEY             ; y = y - 1
$FBFD:10 F6     BPL $FBF5       ; repeat 0x1D + 1 times
$FBFF:60        RTS             ; end of checksum algorithm

$FC2A:98        TYA             ; put counter into a
$FC2B:48        PHA             ; push counter to stack
$FC2C:A0 08     LDY #$08        ; load new counter with 8
$FC2E:A5 95     LDA $0095       ; load checksum high byte
$FC30:45 3C     EOR $003C       ; xor data[counter] with checksum high byte
$FC32:06 94     ASL $0094       ; shift left checksum low byte
$FC34:26 95     ROL $0095       ; rotate shifted bit onto checksum high byte
$FC36:06 3C     ASL $003C       ; shift left data[counter]
$FC38:0A        ASL             ; shift left original checksum high byte
$FC39:90 0C     BCC $FC47       ; skip if shifted bit was 0
$FC3B:A5 94     LDA $0094       ; load checksum low byte
$FC3D:49 21     EOR #$21        ; xor checksum low byte with 0x21
$FC3F:85 94     STA $0094       ; store into checksum low byte
$FC41:A5 95     LDA $0095       ; load checksum high byte
$FC43:49 10     EOR #$10        ; xor checksum high with 0x10
$FC45:85 95     STA $0095       ; store checksum high byte
$FC47:88        DEY             ; y = y - 1
$FC48:D0 E4     BNE $FC2E       ; repeat 8 times
$FC4A:68        PLA             ; pull counter from stack
$FC4B:A8        TAY             ; restore counter to y
$FC4C:60        RTS             ; return
```

It is a simple process, though fairly cumbersome for anyone to duplicate by hand. I would recommend either checksum bypass with a game genie or using dwsrame to fix the checksum for you. Here is the C++ conversion of the assembly routine used in dwsrame.

```
wxUint16 SRAMFile::checksum(int game) const {
  wxASSERT((game >= 0) && (game < 3));

  unsigned char cl = 0x1D, ch = 0x1D, carry = 0;
  unsigned char al, bl, temp;

  for (int i = 0x1D; i >= 0; --i) {
      al = sram[GAME_OFFSET + (game * GAME_SIZE) + i];

      for (int j = 8; j > 0; --j) {
          bl = al ^ ch;

          // asl cl
          carry = (cl & 0x80) ? 1 : 0;
          cl <<= 1;

          // rol ch
          temp = (ch & 0x80) ? 1 : 0;
          ch = (ch << 1) | carry;
          carry = temp;

          // asl al
          carry = (al & 0x80) ? 1 : 0;
          al <<= 1;
```

```
            // asl bl
            carry = (bl & 0x80) ? 1 : 0;
            bl <<= 1;

            if (carry) {
                cl ^= 0x21;
                ch ^= 0x10;
            }
        }
    }

    return (cl | (ch << 8));
}
```

If you know basic arithmetic and binary operations, the algorithm should look pretty straightforwawrd. I will detail the operations real quick.

We start with two counters. I will refer to these as checksum high and checksum low. They are unsigned bytes (range 0 - 255) and can be represented in 8 bits. Assign these two counters the initial value 0x1D.

Starting at the end of the game data, which is at offset 0x1D from the start of a game slot, we do the following for each game byte from end to beginning.

Grab the next byte of game data. I will refer to this as al.

Do the following 8 times (once for each bit of the game data byte).

Compute the exclusive OR of the game byte with checksum high. Shift the checksum low byte 1 bit left (this may result in a carry if the high bit was 1 - do not discard this bit). Shift the checksum high byte left 1 bit. If a carry resulted from the checksum low shift, that bit should become the new low bit of checksum high. As with the first shift, do not discard the high bit shifted off of checksum high.

Now shift left 1 bit the game data byte (al). If a carry results in this shift, it will replace any former carry.

Now shift left 1 bit the exclusive OR we calculated earlier. If a carry results, it will replace any former carry.

If we have a carry, then we XOR checksum low with 0x21 and XOR checksum high with 0x10.

Repeat as directed.

It's certainly an annoying process to do by hand, but not a complicated one. I would recommend you avoid doing this by hand though. The next section has a method for bypassing the checksum if you have a game genie or an emulator that supports game genie codes.

4.4 Checksum Bypass Using a Game Genie

If you just want to poke around in the game data and change things without worrying too much about the checksum, one method you can use is a game genie. We can use a game genie code to bypass the sanity check.

APVYNUAU will bypass the game sanity check and allow any modified SRAM data

to work. What will happen if the data is invalid is unknown.

5.0 dwsrame - The Dragon Warrior (NES) SRAM Editor

If you really want to edit the SRAM, I recommend a program called 'dwsrame', the Dragon Warrior (NES) SRAM Editor. Not surprisingly, I wrote it.

It will edit any of the offsets I have outlined in this document an will keep fix the sanity values for you so you don't need to. It's far simpler than trying to edit by hand.

If you want to try it out, head over to http://games.technoplaza.net/dwsrame/ It's free software under the GNU GPL, tested in Windows, Linux, and Mac OS X, and is likely to run on almost any unix that support GTK+.

6.0 Contact Information

The author (John Ratliff) can be contacted at
webmaster [AT] technoplaza [DOT] net. Replace as necessary.

I can also be reached via an online feedback form at
http://www.technoplaza.net/feedback.php