Secret of Evermore SRAM Guide

by jdratlif   Updated to v1.0 on Sep 9, 2006

```
-------------------------------------------------------------------------------
| Secret of Evermore SRAM Document 1.0
| by John David Ratliff
|
| The most recent version of this guide can always be found at
| http://games.technoplaza.net/soesrame/sram-doc.txt
|
```

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

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

This document is a guide to the Secret of Evermore SRAM, the section of memory in the Secret of Evermore cartridge used to save your progress.

SNES emulators create a special file to store this memory. These are known as SRAM files and generally have a .srm extention (ZSNES and Snes9x both use the .srm extention).

This guide will describe the SRAM, what it stores and where, and how to edit it to change your save games. For a program that edits the SRAM, see section 5.0.

I consider SRAM editing superior to save state editing for two reasons. First, generally only one emulator can read any particular save state file, so save state hacking only works if you use that particular emulator. Second, save state formats can change between emulator revisions, which renders state editing documents useless. There are several old DOS save state editors and documents for Nesticle save states. As nearly no one still uses nesticle, all those programs and documents are useless. There are almost as many documents and programs for ZSNES save states, but the ZSNES save state format has changed so often, most of these programs and documents are also useless.

Save state editors/documents are more popular than SRAM editors/documents because they are easier to change. Because the SRAM employs a battery to keep the data, and the battery eventually diea, SRAM files need to be sanity checked to see if the battery is still functioning. This leads to extra information stored in the SRAM that is used to test the validity of the data. When you edit SRAM files, you must also correct this sanity data, or the game will think the SRAM is not valid, and you won't be able to play your edited games.

```
-------------------------------------------------------------------------------
| 2.0 Copyright Notice
-------------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------------
| 3.0 Revision History
-------------------------------------------------------------------------------
```

Version 1.0 - Sunday, September 10, 2006
  - First Public Release

```
-------------------------------------------------------------------------------
| 4.0 The SRAM
-------------------------------------------------------------------------------
```

This section details the SRAM format used by Secret of Evermore.

```
-------------------------------------------------------------------------------
| 4.1 SRAM Basics
-------------------------------------------------------------------------------
```

Secret of Evermore, like most SNES games, used an SRAM of 0x2000 (8 KB) bytes to store its information. Up to four games could be save. Each game used 0x331 bytes of SRAM. This means 0xCC4 bytes of SRAM are reserved for the game data, with 0x133C bytes used for other things (some unused). In this document, we are only interested in the 0xCC4 bytes of save game data.

Not all of the information contained in the save game data is known. Most of what is undocumented is related to quest progress and opened or already found treasure. The only major items of interest that I've yet to find are the save location, and the rare quest items.

Here is a short listing of what is known in no particular order. The sanity (aka checksum) data, the known alchemys and their levels, the boy's and dog's stats (name, experience, level, hp, and mp), the money (talons, jewels, gold coins, and credits), and the inventory (alchemy ingredients, items, armors, weapons, helmets, gauntlets, collars, trade goods, and charms).

The first game starts at 0x2 in the SRAM, and after its 0x331 bytes, the second game starts and so on for the four games.

The next section will present the SRAM offsets and their meanings.

```
-------------------------------------------------------------------------------
| 4.2 SRAM Offsets
-------------------------------------------------------------------------------
```

This information is presented in no particular order. Similar information will be grouped together.

When an offset is listed, this offset is relative to the start of the game data. For example, if I said the boy's name is at offset 0x26, this would be 0x28 (0x2 + 0x26) from the start of SRAM for game 1, and 0x359 (0x2 + 0x331 + 0x26), and so on.

Data in the SRAM is stored in little endian format, the format used by the SNES processor. This means words are stored with their least significant byte first. For instance, 400 is 0x190 in hex, which is two bytes 0x01 and 0x90. In little endian, this would be 0x9001 [90 01]. Mostly, this just means reverse the bytes. There are plenty of endian guides on the web if you need a more in-depth explanation.

```
-----------------------------------------------------
Sanity Data - the sanity/checksum word
  0x0 (2 bytes)
-----------------------------------------------------

-----------------------------------------------------
Boy's Stats
  0x26 (name)
  0x9D (level)
  0x6E (current HP, 2 bytes)
  0x8E (max HP, 2 bytes)
  0x9A (experience, 3 bytes)

Dog's Stats
  0x4A (name)
  0xDE (level)
  0xAF (current HP, 2 bytes)
  0xCF (max HP, 2 bytes)
-----------------------------------------------------
```

The boy and dog's names have storage space for 34 characters (null terminated). However the game limits you to 15 character names. Using longer names is untested.

The name is ASCII encoded, and names can consist of the entire alphabet (upper and lowercase), the numbers 0-9, and the characters , (comma), . (period), ! (exclamation point), ' (apostrophe), \ (backslash), - (hyphen), & (ampersand), # (pound), and   (space). This is the alphabet the game limits you to using. Using characters outside this alphabet is untested.

The byte immediately after the character's name must be the null terminator (00). If there is no terminator, the game will crash on load.

The valid range for levels are 1-99. The valid range for hp is 0-999, though the max should probably be at least 1.

The valid range for experience is 0-7562471. The high value is the dog's level 99 experience. Anything more than that is pointless.

The next offsets are for the money types in the game. Their valid range is 0-16,777,215. The high value is the max value for an unsigned 24-bit number.

```
----------------------------------------------------
0xFC  (Talons, 3 bytes)
0xFF  (Jewels, 3 bytes)
0x102 (Gold Coins, 3 bytes)
0x105 (Credits, 3 bytes)
----------------------------------------------------
```

The next offsets are for the weapon inventory. There is a single bit for each weapon. The first value is the offset in the SRAM, and the second is the bit that controls it. A set bit (1) means you have the weapon, and a clear bit means you do not have it.

Bit number start at 0 and go through 7 and run right to left. For example, in the byte with bit pattern 10010010, bit 0 is clear and bit 7 is set.

```
----------------------------------------------------
0x279 bit 1 (Bone Crusher)
0x279 bit 2 (Gladiator Sword)
0x279 bit 3 (Crusader Sword)
0x279 bit 4 (Neutron Blade)

0x279 bit 5 (Spider's Claw)
0x279 bit 6 (Bronze Axe)
0x279 bit 7 (Knight Basher)
0x27A bit 0 (Atom Smasher)

0x27A bit 1 (Horn Spear)
0x27A bit 2 (Bronze Spear)
0x27A bit 3 (Lance)
0x27A bit 4 (Laser Lance)

0x27A bit 5 (Bazooka)
----------------------------------------------------
```

The next set of offsets are for the weapon levels. The first offset is for the major level (1-3), and the second is for the minor (power up progression).

The minor level ranges from 0-255. The status screen only shows values from 0-99. To convert between the actual value and the shown value, just divide the minor level by 2.56 and ignore the decimal places.

```
----------------------------------------------------
0x116 0x115 (Bone Crusher)
0x118 0x117 (Gladiator Sword)
0x11A 0x119 (Crusader Sword)
0x11C 0x11B (Neutron Sword)

0x11E 0x11D (Spider's Claw)
0x120 0x11F (Bronze Axe)
0x122 0x121 (Knight Basher)
```

```
0x124 0x123 (Atom Smasher)

0x126 0x125 (Horn Spear)
0x128 0x127 (Bronze Spear)
0x12A 0x129 (Lance)
0x12B 0x12A (Laser Lance)

0x13E 0x13D (Dog's Attack Level)
-----------------------------------------------------
```

The next set of offsets are for the alchemy ingredients. Their valid range is from 0-99.

```
-----------------------------------------------------
0x289 (Wax)
0x28A (Water)
0x28B (Vinegar)
0x28C (Root)
0x28D (Oil)
0x28E (Mushroom)
0x28F (Mud Pepper)
0x290 (Meteorite)
0x291 (Limestone)
0x292 (Iron)
0x293 (Gunpowder)
0x294 (Grease)
0x295 (Feather)
0x296 (Ethanol)
0x297 (Dry Ice)
0x298 (Crystal)
0x299 (Clay)
0x29A (Brimstone)
0x29B (Bone)
0x29C (Atlas Medallion)
0x29D (Ash)
0x29E (Acorn)
-----------------------------------------------------
```

The next offsets are for the alchemy inventory. Just like the weapon inventory, the first value is the offset and the second is the bit that controls it. A set bit means you know the alchemy, and a clear bit means you don't.

```
-----------------------------------------------------
0x1F7 bit 0 (Acid Rain)
0x1F7 bit 1 (Atlas)
0x1F7 bit 2 (Barrier)
0x1F7 bit 3 (Call up)
0x1F7 bit 4 (Corrosion)
0x1F7 bit 5 (Crush)
0x1F7 bit 6 (Cure)
0x1F7 bit 7 (Defend)

0x1F8 bit 0 (Double Drain)
0x1F8 bit 1 (Drain)
0x1F8 bit 2 (Energize)
0x1F8 bit 3 (Escape)
0x1F8 bit 4 (Explosion)
0x1F8 bit 5 (Fireball)
0x1F8 bit 6 (Fire Power)
```

```
0x1F8 bit 7 (Flash)

0x1F9 bit 0 (Force Field)
0x1F9 bit 1 (Hard Ball)
0x1F9 bit 2 (Heal)
0x1F9 bit 3 (Lance)
0x1F9 bit 4 (Laser - dummied alchemy)
0x1F9 bit 5 (Levitate)
0x1F9 bit 6 (Lightning Storm)
0x1F9 bit 7 (Miracle Cure)

0x1FA bit 0 (Nitro)
0x1FA bit 1 (One Up)
0x1FA bit 2 (Reflect)
0x1FA bit 3 (Regrowth)
0x1FA bit 4 (Revealer)
0x1FA bit 5 (Revive)
0x1FA bit 6 (Slow Burn)
0x1FA bit 7 (Speed)

0x1FB bit 0 (Sting)
0x1FB bit 1 (Stop)
0x1FB bit 2 (Super Heal)
----------------------------------------------------
```

The next set of offsets are for the alchemy levels. The first offset is their major level (0-9), and the second is their minor level (power-up progress). The minor level can range from 0-99. For some reason, both of these offsets are 2 bytes, but since the values never exceed 255, you can just ignore their second byte and treat them like a single byte value.

```
----------------------------------------------------
0x19B 0x155 (Acid Rain)
0x19D 0x157 (Atlas)
0x19F 0x159 (Barrier)
0x1A1 0x15B (Call Up)
0x1A3 0x15D (Corrosion)
0x1A5 0x15F (Crush)
0x1A7 0x161 (Cure)
0x1A9 0x163 (Defend)
0x1AB 0x165 (Double Drain)
0x1AD 0x167 (Drain)
0x1AF 0x169 (Energize)
0x1B1 0x16B (Escape)
0x1B3 0x16D (Explosion)
0x1B5 0x16F (Fireball)
0x1B7 0x171 (Fire Power)
0x1B9 0x173 (Flash)
0x1BB 0x175 (Force Field)
0x1BD 0x177 (Hard Ball)
0x1BF 0x179 (Heal)
0x1C1 0x17B (Lance)
0x1C3 0x17D (Laser - dummied alchemy)
0x1C5 0x17F (Leviate)
0x1C7 0x181 (Lightning Storm)
0x1C9 0x183 (Miracle Cure)
0x1CB 0x185 (Nitro)
0x1CD 0x187 (One Up)
0x1CF 0x189 (Reflect)
0x1D1 0x18B (Regrowth)
```

```
0x1D3 0x18D (Revealer)
0x1D5 0x18F (Revive)
0x1D7 0x191 (Slow Burn)
0x1D9 0x193 (Speed)
0x1DB 0x195 (Sting)
0x1DD 0x197 (Stop)
0x1DF 0x199 (Super Heal)
----------------------------------------------------

The next offsets are for the inventory items, which include the common
items, equipment, and tradge goods. In other words, anything that can be
bought, sold, or traded.

Most items have a valid range of 0-6, but a few can be up to 99. These
include call beads, bazooka ammunition, and all the trade goods.

----------------------------------------------------
0x29F (Petal)
0x2A0 (Nectar)
0x2A1 (Honey)
0x2A2 (Dog Biscuit)
0x2A3 (Wings)
0x2A4 (Essence)
0x2A5 (Pixie Dust)
0x2A6 (Call Bead)

0x2A7 (Grass Vest)
0x2A8 (Shell Plate)
0x2A9 (Dino Skin)
0x2AA (Bronze Armor)
0x2AB (Stone Vest)
0x2AC (Centurion Cape)
0x2AD (Silver Mail)
0x2AE (Gold Plated Vest)
0x2AF (Shining Armor)
0x2B0 (Magna Mail)
0x2B1 (Titanium Vest)
0x2B2 (Virtual Vest)

0x2B3 (Grass Hat)
0x2B4 (Shell Hat)
0x2B5 (Dino Helm)
0x2B6 (Bronze Helmet)
0x2B7 (Obsidian Helm)
0x2B8 (Centurion Helm)
0x2B9 (Titan's Crown)
0x2BA (Dragon Helm)
0x2BB (Knight's Helm)
0x2BC (Lightning Helm)
0x2BD (Old Reliable)
0x2BE (Brainstorm)

0x2BF (Vine Bracelet)
0x2C0 (Mammoth Guard)
0x2C1 (Claw Guard)
0x2C2 (Serpent Bracer)
0x2C3 (Bronze Gauntlet)
0x2C4 (Gloves of Ra)
0x2C5 (Iron Bracer)
0x2C6 (Magician's Ring)
```

```
  0x2C7 (Dragon's Claw)
  0x2C8 (Cyberglove)
  0x2C9 (Protector Ring)
  0x2CA (Virtual Glove)

  0x2CB (Leather Collar)
  0x2CC (Spiky Collar)
  0x2CD (Defender Collar)
  0x2CE (Spot's Collar)

  0x2CF (Thunderball)
  0x2D0 (Particle Bomb)
  0x2D1 (Cryo-Blast)

  0x315 (Annihilation Amulet)
  0x316 (Beads)
  0x317 (Ceramic Pot)
  0x318 (Chicken)
  0x319 (Golden Jackal)
  0x31A (Jeweled Scarab)
  0x31B (Limestone Tablet)
  0x31C (Perfume)
  0x31D (Rice)
  0x31E (Spice)
  0x31F (Souvenir Spoon)
  0x320 (Tapestry)
  0x321 (Ticket for Exhibition)
  ----------------------------------------------------
```

The final offsets are for the charms. They are just like the weapon and alchemy inventories. The first value is the offset, and the second is the controlling bit.

```
  ----------------------------------------------------
  0x200 bit 5 (Armor Polish)
  0x200 bit 6 (Chocobo Egg)
  0x200 bit 7 (Insect Incense)

  0x201 bit 0 (Jade Disk)
  0x201 bit 1 (Jaguar Ring)
  0x201 bit 2 (Magic Gourd)
  0x201 bit 3 (Moxa Stick)
  0x201 bit 4 (Oracle Bone)
  0x201 bit 5 (Ruby Heart)
  0x201 bit 6 (Silver Sheath)
  0x201 bit 7 (Staff of Life)

  0x202 bit 0 (Sun Stone)
  0x202 bit 1 (Thug's Cloak)
  0x202 bit 2 (Wizard's Coin)
  ----------------------------------------------------
```

The remaining values in the SRAM are unknown. Please contact me if there you think I have missed something important.

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

As described earlier, Secret of Evermore uses two bytes of sanity data to

```
ensure the SRAM is still being saved by the battery. This section describes
that algorithm.

The following is the actual algorithm, obtained using Geiger's Snes9x
debugger. I have commented each line.

---------------------------------------------------------------------
$8D/B469 A0 2F 03    LDY #$032F             ; load 0x32F into counter
$8D/B46C 22 78 B4 8D JSL $8DB478[$8D:B478]  ; jump to subroutine

$8D/B478 A9 3F 04    LDA #$043F             ; a = 0x43F
$8D/B47B E2 20       SEP #$20               ; use 8-bit accumulator
$8D/B47D 18          CLC                    ; clear carry
$8D/B47E 7D 00 00    ADC $0000,x[$30:6666]  ; add data[offset]

$8D/B481 88          DEY                    ; y = y - 1
$8D/B482 F0 0B       BEQ $0B    [$B48F]     ; branch if y = 0
$8D/B484 E8          INX                    ; x = x + 1
$8D/B485 C2 20       REP #$20               ; use 16-bit accumulator
$8D/B487 0A          ASL A                  ; shift left
$8D/B488 E2 20       SEP #$20               ; use 8-bit accumulator
$8D/B48A 7D 00 00    ADC $0000,x[$30:6667]  ; add data[offset]
$8D/B48D 80 F2       BRA $F2    [$B481]     ; branch always

$8D/B48F C2 20       REP #$20               ; use 16-bit accumulator
$8D/B491 6B          RTL                    ; return from subroutine
---------------------------------------------------------------------

The SNES used a custom WDC 65c816 processor, so if you wanted to learn more
about this code, that's the assembly language it's in.

The algorithm is fairly straightforward.

First, we take an accumulator and initialize it with 0x043F. The first byte
of game data is added to the lower byte of our accumulator. If a carry
results, we ignore it.

Next we start a loop. We shift the accumulator left by 1. The shifted bit is
placed in the carry. Then we add the next byte of data to the lower byte of
our accumulator. The carry is also added here and reset to 0. If our addition
results in a carry, it will be placed in the carry. This process is repeated
for every byte of game data. Note that the carry from our addition will be
ignored due to the shift that always occurs following the next iteration of
the loop.

Here is the C++ routine from soesrame I wrote (based on C source from
phonymike).

--------------------------------------------------------------------------
quint16 SRAMFile::checksum(int game) const {
  quint32 checksum = 0x43F;
  unsigned char temp = checksum +
                       sram[SRAM_GAME_OFFSET + 2 + game * SRAM_GAME_SIZE];

  for (int i = 3; i < SRAM_GAME_SIZE; ++i) {
      checksum &= 0xFF00;
      checksum |= temp;
      checksum <<= 1;

      if (checksum > 0xFFFF) {
          checksum -= 0xFFFF;
      }

      temp = checksum + sram[SRAM_GAME_OFFSET + i + game * SRAM_GAME_SIZE];
  }

  return static_cast<quint16>(checksum);
}
---------------------------------------------------------------------------
```

To clear up any confusion, a quint16 is a 16-bit unsigned integer and a quint32 is an unsigned 32-bit integer. These are types defined by the Qt library which is used by soesrame.

When you edit the SRAM by hand, the checksum will be wrong. To get around this, you can either fix the checksum, or have the game skip the sanity check.

The latter can be accomplished by using a game gene code. DDA4-17E1 will do this.

If you don't have a game genie (either because you're using an emulator that doesn't support game genie codes, or because you're using a game copier on a real SNES), then you will need to compute the checksum and fix the SRAM sanity data for the game.

The following is a listing for a C program which will fix the sanity data for an SRAM file.

```
-------------------------------- soe-sanity.c --------------------------------
/*
 * Secret of Evermore Sanity Fixer
 * Copyright (C) 2006 emuWorks
 * http://games.technoplaza.net/
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
 */

#include <stdio.h>

// define this at compile time on big endian systems (i.e. PPC Mac's)
#ifdef BYTE_ORDER_BIGENDIAN
    #define SWAP_ON_BIGENDIAN(w) ((w << 8) | (w >> 8))
#else
    #define SWAP_ON_BIGENDIAN(w) (w)
#endif

static void fix_game(unsigned char *sram, int game) {
    unsigned short *offset = (unsigned short *)(sram + 2 + (game * 0x331));
    unsigned long checksum = 0x43F;
    unsigned char temp = checksum + sram[4 + (game * 0x331)];
    int i;

    for (i = 3; i < 0x331; ++i) {
        checksum &= 0xFF00;
        checksum |= temp;
        checksum <<= 1;

        if (checksum > 0xFFFF)
            checksum -= 0xFFFF;

        temp = checksum + sram[2 + i + (game * 0x331)];
    }

    *offset = SWAP_ON_BIGENDIAN(checksum);

    printf("fixed checksum for game %d\n", (game + 1));
}

int main(int argc, char **argv) {
    FILE *f;
    int fix_mask = 0xF, i;
    char sram[0x2000];

    if ((argc != 2) && (argc != 3)) {
        fprintf(stderr, "syntax: soe-sanity sram-file.srm [ game (1 - 4) ]\n");
        return -1;
    }

    if (argc == 3) {
        switch (*argv[2]) {
            case '1': fix_mask = 0x1; break;
            case '2': fix_mask = 0x2; break;
            case '3': fix_mask = 0x4; break;
            case '4': fix_mask = 0x8; break;
            default: break; // ignore third argument if invalid
        }
    }

    if ((f = fopen(argv[1], "rb")) == NULL) {
        fprintf(stderr, "error: unable to open SRAM file '%s'\n", argv[1]);
        return -1;
    }

    fseek(f, 0, SEEK_END);

    if (ftell(f) != 0x2000) {
        fprintf(stderr, "error: invalid SRAM file size.\n");
        return -1;
    }

    fseek(f, 0, SEEK_SET);

    if (fread(sram, 0x2000, 1, f) != 1) {
        fprintf(stderr, "error: unable to read SRAM file.\n");
        return -1;
    }
    fclose(f);

    for (i = 0; i < 4; ++i) {
        if (fix_mask & (1 << i)) {
            fix_game(sram, i);
        }
    }

    if ((f = fopen(argv[1], "wb")) == NULL) {
        fprintf(stderr, "error: unable to open SRAM file '%s'\n", argv[1]);
        return -1;
    }

    if (fwrite(sram, 0x2000, 1, f) != 1) {
        fprintf(stderr, "error: unable to write SRAM data to file.\n");
        return -1;
    }

    fclose(f);

    return 0;
}
--------------------------------------------------------------------------------
```

There is a windows binary available at
http://games.technoplaza.net/soesrame/history/soe-sanity.zip. For other
platforms, you will need to compile it yourself. It is written in standard
ANSI C, so it should run anywhere.

If you are compiling on a big endian platform, you will need to define
BYTE_ORDER_BIGENDIAN at compile time. PowerPC and Sparc machines are probably
the most common big endian platforms. Intel machines (x86) are little endian.

```
gcc -o soe-sanity soe-sanity.c
```

Will build the program using gcc on a little endian platform.

```
gcc -DBYTE_ORDER_BIGENDIAN -o soe-sanity soe-sanity.c
```

Will buile the program using gcc on a big endian machine. I don't have much
experience with compilers other than gcc, so you'll have to figure it out
yourself if you're using something else.

Using the program should be simple. Just pass it the name of an SoE SRAM file
and it will fix the checksums for all the games automatically. If you only
want to fix one game's checksum, just pass the game number (1-4) as the
second argument. This is useful when you don't have four save games, as the
game might crash if the data in the other games isn't valid.

Examples:
```
  soe-sanity "Secret of Evermore (U).srm"
  soe-sanity soe.srm 1
```

The first one fixes all the checksums in Secret of Evermore (U).srm. The
second one fixes just the first game in soe.srm.

| 5.0 soesrame - Secret of Evermore SRAM Editor

For those that don't want to play around with the SRAM directly, I have written a program that edits all the values documented here. It's also nice in that it keeps the values within their valid ranges, and handles the sanity data for you.

You can find it at http://games.technoplaza.net/soesrame/. There are binaries for Windows, Mac OS X, and Linux. It should work on any platform that supports Qt, so it should work on any unice with X Windows. The full source is available as well.

```
-------------------------------------------------------------------------------
| 6.0 Credits & Acknowledgements
-------------------------------------------------------------------------------
```

I want to thank phonymike for discovering and documenting the checksum algorithm used by Secret of Evermore. He was also very helpful in answering questions I had about his code.

Thanks to Geiger for his debugging Snes9x version and the entire Snes9x team for Snes9x. Without it, it would have been very difficult to find the SRAM offsets.

```
-------------------------------------------------------------------------------
| 7.0 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