NSF ripper Guide Intro

About this PDF

Author of this guide is Gil Galad, and you can find it on http://gilgalad.arc-nova.org/intro.html, but to make sure, that this guide suddenly won't disappear from the internet, I had copy each lesson to this PDF file. So maybe, one day, it will come handy.

Author: Gil Galad // http://gilgalad.arc-nova.org/intro.html
Twierdza RPG Makera (RPG Maker Stronghold) // www.rpgmaker.pl

NSF ripper Guide Intro

This is a guide to ripping NSF's in a detailed manner.

A NSF is a sound format file that has been ripped from a NES game system ROM. The
process of ripping a NSF from the ROM image requires careful study and detailed observation.
The sound is often independent from the other code in the game and can be removed for play in a
NSF file.

As many know the entire program is encoded on the game cartridge ROM. You have your PRG (program
ROM) that is encoded on a chip along with your sound driver that is embedded in the PRG in most cases.
The sound driver is then extracted in all it's 5 channels and can be heard with a NSF player.

Kevin Horton and Chris Covell are the ones that started the NSF format and we are very grateful to them
for the creation of this format. Upon hearing NSF's for the first time you will appreciate the emulated sound
Coming from the player. And thanks for that.

It is well worth ripping NSF's when you understand how great they sound.

In order to extract the data you must analyze the data. And this is where this NSF ripping guide
will come in handy.

This guide assumes that you know nothing about NSF ripping and even a beginner can rip NSF's eventually
after reading and using this guide.

After reading this guide you should be able to rip NSF's and you will become the "ripper"

And now we will start to view NES ripping on an intellectual level. Starting with the very basics and working our way to more detailed matters. Starting with the 3 main address calls in a NSF.

- init: This is where your tune number is set and this routine also changes your tune. Also initializes memory addresses.
- play: This is your basic sound driver and is usually found in your NMI interrupt.
- load: This is the start of your program or bank and is often $8000.

With these 3 areas being described you can analyze the data more closely in these areas
Also you should study Family Computer references as needed when analyzing the data

And as far as the contents are concerned, you will find some errors perhaps.

All the information you can get will help you.

NOTE:

This guide is an adaptation of the original NSF Ripping guide. The original was written by Izumi and is in Japanese. I have translated quite abit of the guide and also rewrote some of the guide and changed some of the data so that it's more accurate. Also this guide was written quite some time ago and now I have figured out new methods and also there is better tools that you can use and this doc will reflect this.

So you have 20 levels of NSF ripping where the lower levels are easy to rip and the higher levels get really hard. For each level I will show you how to rip them and in some cases I will let you do the work instead of giving you the answer.

We will start you out at level 0. Level 0 is preparation. In this level you will begin and gather tools and docs. You will also learn about the format and other things. Yes, you gotta start somewhere and so this ends the introduction.

INDEX

NSF Reference Data

Family Computer NSF Spec Mapper Registers 6502 Guide Opcodes
APU Reference NES Audio Ripping

LEVEL0 Preparation

Start gathering the tools!!

- Hex Editor.....I recommend Hex Workshop
- Split/join file utility.....Being able to join and split 4KB - 32K and more is a must
- 6502 Disassembler.....A disassembler that is specifically for the NES is a must.
- NSF Player.....There is several NSF players out there. Each are alittle different.

- NES2NSF v0.9.....NSF ROM ripping tool (partial translated version for download)
- DCC6502.....A 6502/NES specialized disassembler that is helpful in NSF ripping.

The first thing you want to do is get a NSF player and a couple of NSF's. Go to Zophar's Domain to get all the NSF's and NSF players you want. You can also get NSF's here on my page Gil Galad.

As far as the players are concerned you could chose several. NotsoFatso is supposedly one of the best Winamp plugins. I have no idea since it's way too slow to run on my pc. If you have a slow pc you might want to get Nosefart. Nosefart is not an accurate player at all, the player was designed to run on low end systems like mine and it sacrifices a lot of accuracy to do so. You also have stand alone NSF players like GNSF. GNSF is a good player and has some tools that you can use. You have a menu with a sound channel meter and you turn the channels off and on if you wish. You can also tell what channel is missing in your rip. SlickNSF is also a good stand alone player because you can dump a trace log which helps so much in debugging the NSF's code.

You also have emulators that support NSF playing. FCEUD,Nesten,VirtuaNES,Nintendulator has built in NSF players. Most of them has some type of 6502 debugging tool. FCEUD is about the best NES debugger you can get. Nesten also has a debugger and can dump ram which is useful in trapping the right banks with music code/data. Nintendulator has a trace logger and can dump the trace. Also if you have problems with your NSF then Nintendulator will not play the NSF. What will happen is that you will get an error message

saying the you have a NSF that's corrupted or has raw PCM. In most cases you have a routine in the code that is timing out and doesn't return in so many frames. This is useful because after you fix the problem the NSF will run on just about any player.

Make sure that when you chose a player that has support for the expansion sound chips. Here is the expansion sound chip format and the only ones you will find left to rip are FDS disks and some FDS games dont have FDS expansion sound so be aware of that. Very few players support all sound chips so look around.

- VRC VI - Konami
- VRC VII - Konami
- N106 - Namco
- FME-07 - Sunsoft
- MMC5 - Nintendo - Various companies
- FDS - Various companies

Pre-preparation Analysis

Prepare Header

The 128 byte header is used to prepend to the data that you have ripped.

Remove the NES header

The 16 byte header is what emulators use to run the ROM. However this header is not needed nor used in the actual cartridge. If the header is not removed then the banks will be off by 16 bytes.

When **NES2NSF** is used. It's as simple as it gets. Also with this tool you don't need to remove the 16 byte NES header. A 128 byte NSF header is added to every 16K bank ripped from the ROM. You cannot use UNIF format NES roms with this tool.

The NSF Header

You must use this header for NSF compiliation. Provided is the NESM header format and in detail.

```
offset # of bytes   Function
----------------------------

0000  5   STRING  "NESM",01Ah  ; NSF standard indication.
0005  1   BYTE    Current NSF version number. Presently 1
0006  1   BYTE    Number of tracks
0007  1   BYTE    Starting track
0008  2   WORD   load(lo/hi) Load address of data            memory range:
(8000-FFFF)
000A  2   WORD   init(lo/hi) Start of initialization data    memory range:
(8000-FFFF)
000C  2   WORD   play(lo/hi) Start of driver data           memory range:
(8000-FFFF)
000E  32  STRING  Name of game or song
002E  32  STRING  Name of artist or songwriter
004E  32  STRING  Name of the copywrite holder
006E  2   WORD    (lo/hi) This is the NTSC playback speed. 411A is usually
just fine.
```

```
0070  8   BYTE    Bankswitching Init 8 bit value. Explained in better
detail later.
0078  2   WORD    (lo/hi) This is the PAL playback speed.
007A  1   BYTE    PAL/NTSC The following bits are set for either a PAL or
NTSC game.
                bit 0: if clear, this is an NTSC tune
                bit 0: if set, this is a PAL tune
                bit 1: if set, this is a dual PAL/NTSC tune
                bits 2-7: Not used and must be 0's
007B  1   BYTE   Special sound chip support
                bit 0: 1 = Konami   VRC6
                bit 1: 1 = Konami   VRC7
                bit 2: 1 = Nintendo FDS
                bit 3: 1 = Nintendo MMC5
                bit 4: 1 = Namcot   106
                bit 5: 1 = Sunsoft  FME7
                bits 6-7:  Currently unused. For future expansion. More
sound chips out there possibly.
007C    4   ----    4 extra bytes for expansion (must be 00h)
0080  nnn ----    The ripped data goes here.
-----------------------
```

- (lo/hi)Data in the header is used in this fashion. The playback speed is one of them. 411A is the speed, however you
  would switch the bytes to 1A41
- These 6 bytes of the load/play/init are treated as (lo/hi) as above.
- For those 6 important bytes it's easy to make a mistake. So make sure you get them in the proper (lo/hi) order.

LEVEL1 CAPCOM(1943)

When you read the **NES Music Ripping Guide** you will understand that many games have the load/play/init addresses set at 8000/8003/8000.1943 is one of these games and is simplicity in itself. What your going to do is take a 16KB bank from the rom. This bank should be 16,384 bytes. Bank 005 should be the right bank with the music data. Attach the header to the 16KB bank that you ripped and then set the 8000(load)/8003(init)/8000(play) addresses in the header respectively. 01 is the starting track where you hear music.

And now this makes you a Level 1 NSF ripper. With this level out of the way be prepared to advance in difficulty and a sense of acomplishment.

Let's discuss bank sizes for a moment. This is important to learn as you will learn that NES games load different size banks into memory. What is loaded is the PRG(Program ROM) banks into memory. Depending on the mapper the banksize select can be different. Mapper 0 is not actually a mapper, this is the standard 32K PRG that is loaded into memory at addresses $8000 - $FFFF, there is also a mapper 0 16K version as well $C000 - $FFFF. You do have some mappers that are 32KB bank size select and one mapper is mapper 7. The most common bank sized select is 16KB and usually that's mapper 1 or MMC1. And you have 8KB banksized select which is many mappers,the most common mapper is mapper 4 or MMC3. And last but not least is the NSF mapper. The banksize select of a NSF is 4KB.

Here is a table for the banksizes.

- 4KB - NSF bank size select - 1000
- 8KB - MMC3 and others - 2000
- 16KB - Mapper 0,most games like MMC1 - 4000
- 32KB - Mapper 0,Mapper 7,Mapper 3 and others - 8000

Now is the time to talk about using nes2nsf.The app is a program written by a Japanese guy and I've taken the time to partially translate it. This utility is a very useful tool and is used to rip banks from the NES ROM. You can divide the entire rom up in banks that you select in the program or you can choose an option that allows you to rip certain banks from the rom. The split banks that are output from nes2nsf are numbered from 0 - nn(however many banks is in the rom depending on the banksize). You can set an option in the program that will only rip certain banks based on the sound enable register 4015. If you have a write to that sound register then you will get only those banks. Now for some specifics about the program. You

have a .ini file which is a config file for nes2nsf and I will show you how to set the bits of the program on this .ini file.

1000100000000000

This is the default setting of the program and as you can see there is 16 bits here and I give you a brief explanation of what you need to use right now.

- bit 0 - If this is a 1 then the program prepends a NSF header to the rip.
- bit 1 - This is where you set the banksize for the output of the program.
- bit 3 - must always be a 0
- bit 4 - if 8D1540 is found then a file is output ; STA $4015 ; 0 = OFF 1 = ON
- bit 5 - if 8E1540 is found then a file is output ; STX $4015 ; 0 = OFF 1 = ON
- bit 6 - if 8C1540 is found then a file is output ; STY $4015 ; 0 = OFF 1 = ON
- bit 7 - if 1540 is found then a file is output ; 0 = OFF 1 = ON
- bit 08-0D are always 0

So you use nes2nsf for the game 1943 with the default setting enabled in the .ini file and nes2nsf will output bank # 05 which is the right bank with the music data. The header will also be prepended if you left the settings the same. So you can load the NSF in your player and WOW it works and plays. The beginning track is #1 and you need to count the tracks in the NSF and set the bit in the header to reflect this. You use the hexadecimal number system so you set the byte in the header with a hex editor. Look back at Level 0 and you will see the header and so you look at the data there and figure out what offset to change the ending tune at and I will tell you the first time. 06h and the "h" stands for offset. Dont even try to submit this NSF, It's already been ripped. :)

LEVEL 2 DATA EAST Glory of Hercules 2 (Herakles no Eikou 2)

Now we are going to get down and dirty with some hex hacking and a lot of these techniques you will use over and over again. First you will use nes2nsf to extract the bank and then load up the NSF in a hex editor. You will learn to make a boostrap code and arrange the tunes. Also you you learn learn about some of the guidelines of NSF ripping.

What are you waiting for go ahead and rip the bank from the rom with nes2nsf. Sometimes more then one bank will be outputted, however you will only get one bank with the default settings and that's the right bank. The bank 013 and this bank will be 16KB and already has a header prepended so you're ready to rock.

First try to play the NSF and you will notice that you don't get any sound or with other players you may get some bleeps and that's it. So load the NSF up in a hex editor, we need to fix this problem. You will see tons of numbers and is rather confusing if you don't normally do hex editing. However we will make some sense of this today. If you remember the NSF header is 128 bytes and the range of the header viewable in a hex editor is 0h - 7Fh , the actual PRG area that you extracted will be 80h - 4080h. So you are going to look right after the header and you will see a series of 9 bytes. These bytes are code and mean something. So you see these bytes 4C 11 80 4C 1C 81 4C B4 81 are the first bytes you see after the header and like I said they do mean something. This is code and so 4C means JMP, JMP is a opcode that jumps to a certain address. In this case you have the bytes 4C 11 80 , So you have JMP $8011, jump to address $8011. And yes with addresses you have to switch the bytes like I just did. This is a format used by many CPU's. What I'm going to do is translate the data for you and assign addresses for each operation so it's easy to understand.

```
Address     Data        Opcode  Operand
$8000     4C 11 80       JMP $8011
$8003     4C 1C 81       JMP $811C
$8006     4C B4 81       JMP $81B4
```

What you want to do now is change the addresses in the header because the NSF doesn't play yet. The load address is at offset 08h - 09h and is 2 bytes, remember always that a PRG address is always 2 bytes. The Load address in the header is the origin of the program and that tells the NSF player where to start loading the data. In this case the load will be at $8000 in CPU memory and starts at offset 80h in the NSF. The 2 bytes in the header at 0Ch - 0Dh are the play address. The play address is what updates the audio hardware and is called thousands of times. The 2 bytes at 0Ah - 0Bh is the init address. The init address is the code that changes the tune and

initializes memory addresses and sets up data blocks sometimes. This address is called once before the tune plays.

The current addresses in the header are 8000(load)/8003(play)/8000(init). However the NSF doesn't play so you have to change them. Looking at the code I translated for you, you can change the addresses based on the code I have there. Leave the load address the same and don't change it. Just switch the play and init addresses. If you change the init to $8003 and the play to $8006 then you will get something and these are just sound effects. You can switch the tunes in your NSF player and you will never run into music, just sound effects. So I will show you what to do about that. You have to write some code and it's easy once you get used to it. I will tell you what the bytes are and then after this you will need to look the opcodes up in the reference data that I have provided in Level 0.

```
48       PHA ; Push Accumulator to the stack (you save A so it doesn't get
changed)
20 00 80 JSR $8000 ; jump save return to bootstrap code
68       PLA ; Pull Accumulator from stack (you bring back the number you
saved)
4C 03 80 JMP $8003 ; jump to init
```

Next you want to know where to place the code. Since you have a 16K bank extracted and not the entire 32K you can place the code at the end of the bank. Note you cannot exceed this area if you have a 32K bank unless you have bankswitching. If you have a 32K bank then you must find free space somewhere in the NSF.

We can place the code at offset 4080h in the NSF. This is at the end of the bank. If you have Hex Workshop you have to be in insert mode to add bytes to the end of the file. Type in the following bytes at that offset 48 20 00 80 68 4C 03 80 and you are about ready to try the NSF. However you must change the address in the header because you had to use code before going to the real init address.

I have a simple formula for figuring out the address of the code based on what offset you placed the code. Offset -80(header) + load address. You placed the code at 4080h in the NSF. So you subtract 80 and so you get 4000. Next you add 8000 to this because that's your load address. The final address is $C000. So put 00C0 in the header at the init address place. So now you fire up the NSF, and what the heck the sound effects are still there, keep switching the tunes and you will run into the music tunes. Listen to them abit if you want, however you can't leave the tunes at the back of the NSF. You have to arrange the tunes and I will show you how to do this as well.

First of all start playing the NSF and go to the first music tune and write down the number of each tune that you play on notepad or something like it. You need to do this because there is some sound effects and silence spaces in the NSF and so you have to get those out of there. So you put the tunes in the front and sound effects to

the back of the NSF, this is a guideline of the format and should be followed. I eliminate the sound effects myself. However I shall show you how to add them as well.

The order of the tunes is as follows: 28-39,41-43,45-46. The order of the sound effects is as follows: 6-25,40,44,47. What you do is open a calculator and make sure it's in decimal mode and convert the number to hex and subtract one from it because the tunes actually start at 00 that is loaded into the accumulator. And now you have to write some code and you place this code once again at $C000 (4080h in the NSF). It's ok you can overwrite the code I gave you before because I'm going to integrate the bootstrap into the tune arrangement code.

```
TAX
LDA $C00C,X ; load tune index that you arranged, point this address to the
tune index.
PHA
JSR $8000
PLA
JMP $8003
.db 1B,1C,1D,1E,1F,20,21,22,23,24,25,26,28,29,2C,2D,05,06,ETC.
```

You see this code here. I will explain it some so that it doesn't completely confuse you. You have the tune arrangement code and then the bootstrap right after this. You also have a line here that starts with .db and has bytes listed after it. These are the tune numbers that you arranged. Make sure that you don't type .db in the hex editor lol. So you would type in the following bytes at the CPU address $C000 in the NSF or 4080h offset. AA,BD,0C,C0,48,20,00,80,68,4C,03,80,1B,1C,1D,1E,1F, 20,ETC.

Now play the NSF and you will notice this is some really nice music and soothing. And the tunes are in order and the sound effects are to the back. However you need to do one more thing and that's to fill out the header information. The Title of the NSF(game name),Artist or composer,and Company name that programmed the game. You need to get a relative search program or get a hex editor with table file support to search the game for text. I will not get into how to make a table file or using the programs as that's covered really well elsewhere. Bring the header specs up that I showed you in Level 0 and find the offsets for the title,composer,company and fill in the data. You can only type in so much and you cant type so far that you mess up the next entry in the header so be careful.

Most Data East games have the music driver stored like this and usually that bootstrap code that I showed you is actually called from the reset interrupt code, more about that later.

This is a level that was written completely by me Gil Galad. I ripped this NSF awhile back and you can't submit it yourself. All rips in this guide have already been submitted.

LEVEL 3 Sun Electronics Corp. Shanghai II (DPCM)

In Level 3 you're going to learn how to use a disassembler, learn about the NES sound registers and rehashing some methods used in the last couple of levels.

Go find the disassembler DCC6502. This disassembler has many usefull options for NES and other 6502 CPU's. You can also use this for FDS disassembling. All you have to do is set the origin and that helps in being able to use any memory architecture of any 6502 CPU.

First of all you might want to create a folder for NSF compilation. I usually place this folder on the desktop. Within this directory I place 3 folders and all the utilities I need. These 3 folders should be complete,docs, and work. Any rips that you are frustrated with you can place in the work folder and come back to them later. All the complete NSF's go in the complete folder. All your docs you have as reference can go there.

So you get DCC6502 and place this disassembler in that folder. Also you need edit.com. You can find this program somewhere in windows, just search for it and place it in this folder. You can also get NSFTool at Zophar's Domain in the Utility/general/Nitendo section. I usually prefer to edit the basics of the header in a hex editor myself. Make sure that you have nes2nsf in the folder as well. What I usually do is include some emulators in the folder as well. What I do is make a folder for each one and place the contents in their respective folders. I do this to keep the files down in the main folder.

Now you get to do some .bat file programming. Don't worry this is really easy and the very basics. Make 2 new text files in the main folder and rename the first one asm.bat and the second one edit.bat. After you have done this right click on asm.bat and click edit. So you have a blank note pad. What you type here is DCC6502 -h -n -o8000 filename.DMP >> filename.asm . This is your bat file to run the disassembler DCC6502 based on what you typed in the .bat file. Next you right click on edit.bat and type in EDIT filename.asm and you're done. You have to change the file name all the time depending on what game you're messing with. I usually rename them in DOS format with 8 characters or less to make it easy.

The options was set on the .bat for DCC6502 are as follows. -h is hex output and you need this there because it tells you the bytes of the opcode and operation. -n is Nintendo register commenting and is useful in knowing what areas of code do what. If you see a write to $2006 then more then likely this is not a sound driver area. And the next one -0xxxx is what you set the origin of the disassembly output.

If the start of the code is CPU address $8000 then you set it at -o8000. With FDS rips you can set the origin at $6000 since that is where PRG RAM starts. After all this is the file you want to disassemble and last but not least is >> and after that is the disassembly and you should name the extension .asm.

Now comes the time to learn some NES sound registers. $4015 is the sound enable register and turns on and off the sound channels. There is 5 channels in the NES. You have to write a byte to $4015 to enable or disable certain channels. If you are familiar with the bits of a hex byte then this should make sense.

First of all when you designate a value to the A register you are turning ON/OFF the sound register. A91F8D1540 The following routine.

```
A9 1F      LDA #$1F  When you have 1F in the A register and then convert
this value to binary you get 00011111
8D 15 40   STA $4015 Store your value in A that is 1F to address 4015.
4015       When you set bits---abcde(00011111)You're actually turning all
the 5 channels ON.
```

By the way here is the 5 channels.

Channels

1(e) Square #1

2(d) Square #2

3(c) Triangle

4(b) Noise

5(a) DPCM (Delta Modulation)

Now I will list the sound registers and what channel they are associated with.

Square 1/Square 2

$4000/4 ddle nnnn duty, loop env/disable length, env disable, vol/env

period

$4001/5 eppp nsss enable sweep, period, negative, shift

$4002/6 pppp pppp period low

$4003/7 llll lppp length index, period high

Triangle

$4008 clll llll control, linear counter load

$400A pppp pppp period low

$400B llll lppp length index, period high

Noise

$400C --le nnnn loop env/disable length, env disable, vol/env period

$400E s--- pppp short mode, period index

$400F llll l--- length index

DMC

$4010 il-- ffff IRQ enable, loop, frequency index

$4011 -ddd dddd DAC

$4012 aaaa aaaa sample address

$4013 llll llll sample length

Common

$4015 ---d nt21 length ctr enable: DMC, noise, triangle, pulse 2, 1

$4017 fd-- ---- 5-frame cycle, disable frame interrupt

Status (read)

```
$4015   if-d nt21   DMC IRQ, frame IRQ, length counter statuses
```

As you can see these are the sound channels. Don't panic you don't have to absorb all of this right away. Just be kinda familiar with them.

Now we are going to do the rip after all that. First of all look at the .ini file for nes2nsf. Set bit 0F to 1 and run the program. Load up Shanghai 2. You will see a display come up and it will tell you it's a write to 4015, this is a part of the program I couldn't translate because I don't have the source yet. Check up above for a very vague description of 4015. Everytime nes2nsf stops you will see a menu and it will tell you the CPU address location of the write each time nes2nsf runs into one in the PRG.

Now go ahead and use nes2nsf to rip a NSF from Shanghai 2. You will notice that the menu comes up with option 0F set. You will notice that a write to $4015 is found at the following CPU addresses $800C,$8030,$8796,$879B. If you want to find these in the NSF use the following formula. -8000 +80 = NSF Offset in a hex editor. You will see the bytes 8D1540.

You will notice that the bank 002 is ripped with the file name Shanghai 2(J)_002.nsf. Try to play the NSF and it won't play at all. So we should disassemble the bank and see what the problem is. First you have to remove the NSF header in order to disassemble the bank. The header is in the range of 0h - 7Fh, Don't remove 80h and past that offset. Do this with a hex editor. Rename the file without the header shang.DMP . Change the asm.bat file contents to DCC6502 -h -n -o8000 >>shang.asm. And change edit.bat contents to Edit shang.asm and click on the asm.bat, if you don't have any errors then you did it. Wait for a second or 2 then click on edit.bat and you should see a DOS text window come up with the banks disassembly.

You should see the following code if you did the disassembly right.

```
8000 : 02  db $02        ; invalid op

8001 : 4C 6E 81  JMP $816E  ;

8004 : 4C A6 80  JMP $80A6  ;

8007 : 4C 0A 80  JMP $800A  ;

800A : A9 1F  LDA #$1F  ;

800C : 8D 15 40  STA $4015  ; Sound control
```

```
800F : A9 00  LDA #$00  ;

8011 : 8D 10 40  STA $4010  ; PCM Control reg.
```

Remember the Level 1 with the load/init/play addresses are 8000/8003/8000 and about the same as level 2 with the JMP's as well except you don't have to do the bootstrap code. However the .db(data byte) at CPU address $8000 is throwing the order of the JMP's off. That byte is actually the bank number, however moving on. So you have JMP's at $8001,$8004,$8007. Change the load/init/play addresses in the header to 8000/8004/8001 and load the NSF up and wow it plays. However you don't hear the DPCM and I will get to that problem next. Also remember to look at the addresses that nes2nsf gave you for the writes to $4015 and look at some of the other sound writes and look at the description above and you can even check out the APU reference. This will help you in the long run so you should do it.

So you have a rip that works and is missing the DPCM. This rip would be labeled as Incomplete in the composer info in the header if you could not fix it. However I do have a solution. First rename the NSF rip to shang.nsf to keep it from being overwritten because you're going to use nes2nsf again.

You might be somewhat familiar with mappers and this game is mapper 2. Mapper 2 swaps banks at $8000 - $BFFF and the bank at $C000 - $FFFF is hardwired and never changes. The hardwired bank comes from the last PRG bank in the rom right before the CHR(Character ROM) if the rom has any. This last bank is important for DPCM because DPCM samples can only be placed in CPU memory $C000 - $FFF9. So you have to find this bank and append it to the NSF for it to pick up the DPCM samples.

I have alittle trick to use here. What you're going to do is use nes2nsf again and change the .inf settings. Change bit 0 to 0 because you don't need the header because you are adding a bank on to the NSF at the end. And change bit 4 to 0 because you want nes2nsf to split the entire rom into banks. Now run the program. Make sure you change them back to the defaults when you're done so that you don't forget.

I use a hex editor to join files so you should probably do what I do and use Hex Workshop. If not you will have to use another hex editor that has that option or a file split/join utility.

Take a look at what nes2nsf outputted and see that you have 8 files that are numbered 000-007. Take file 007 because this is the last bank in the ROM, this rom has no CHR. Load up the Shanghai NSF that you ripped earlier and load up bank 007 and append bank 007 to the end of the NSF. Load the NSF in a player and presto you have DPCM and it works and sounds good. Sunsoft has real good tunes.

Now you have a couple things left to do. You must edit the header. Fill in the Name of the game,Composer(if you can't find his/her name or fill in a ? mark). Also you have to arrange the tunes. Find some free space in the rom. Free space is usually a real large chunk of 00's or FF's. I have a recommended spot in the NSF to place the tune arrangement code. That's 3530h Offset in the NSF. You want to figure out what the CPU address is for that offset. Once again I have a formula. Offset -80 +8000 = CPU address. Place this address in the header in the init address. Now you do the tune arrangement code. This time it's much simpler and no bootstrap. Place this code at offset 3530h in the NSF.

The CPU address answer is $B4BO and put that in the header in the init address section. Remember to flip the address bytes.

```
$B4B0 AA        TAX
$B4B1 BD B7 B3  LDA $B4B7,X ; point to the tune index
$B4B4 4C 04 80  JMP $8004   ; jump to init
.db tune numbers here ; arrange them yourself :)
```

I shall show you how to optimize this rip soon. This bank joining will be real usefull one day.

LEVEL 4 Technos Japan Double Dragon 2 The Revenge

You might have noticed before that some companies have the same type of music driver. Sunsoft,Capcom,and Data East has about the same type of music driver. You can rip these the same way everytime just about. There is another company that has the same type of drivers and that's Technos Japan. However this rip will require you to work harder to make it play. I will explain Double Dragon 2 in this level and River City Ransom in level 5.

Use nes2nsf to extract the NSF from the game. You will run into 2 banks 005 and 007. 005 is the bank you want, The reason is because it has several writes to $4015. If you want you can verify it and disassemble each bank and look at them. You will see a lot of sound code in bank 005 so that's what you go for.

So do what I explained in level 3 and disassemble bank 005 and remove the NSF header before you do so. Look around the code for alittle bit and get familiar with it. I shall paste some code here and let you look at what we need to mess with in order to get this rip to work.

```
$8000> 4C 2680: JMP $8026 ;
$8003> 4C 9681: JMP $8196 ;
$8006> 00: BRK  ;
```

```
$8007> 00: BRK  ;
$8008> 00: BRK  ;
$8009> 00: BRK  ;
```

By using the header defaults and even switching the init and play addresses around to 8000/8003 will not get this rip to work. This is an important part of the code and remember it. $8003 is indeed the play address and $8000 is the init address. However you have to initialize some addresses and run some code to get this rip to work. Don't worry right now how this was figured out.

So you will edit the code alittle bit starting out. Change this code JMP $8026 to JMP $8196. And after doing that change the play address in the header to $8000. You switched the JMP's because you need to make it nice and neat for the following code.

```
8000 : 4C 46 81  JMP $8146  ;
8003 : 48  PHA   ;
8004 : A9 00  LDA #$00  ;
8006 : 8D FF 07  STA $07FF  ;
8009 : 20 20 80  JSR $8020  ;
800C : 68  PLA   ;
800D : 8D FF 07  STA $07FF  ;
8010 : EE FF 07  INC $07FF  ;
8013 : A9 0F  LDA #$0F  ;
8015 : 8D 15 40  STA $4015  ; Sound Enable
8018 : 4C 20 80  JMP $8020
```

This is a complex bootstrap code and nicely made as well. You will notice that address $07FF is initialized and is the address that is used to change the tune. So type in the code in a hex editor and change the init and play addresses in the header to 8003/8000 and fire up the NSF and it plays.

Now what you do is arrange the tunes. Find some free space in the NSF and if you can't, you can tack on the code at the end of the NSF since this one is a 16KB bank. Remember that the end of the bank in this case is $C000. After you have arranged the tunes you will do something new and that's count the tunes and convert number to hex and type in the number in the header at offset 06h. This is the entry in the header that tells you how many tunes you have. Don't forget to fill in the name,composer, and company info in the header. Arranging the tunes is more then just moving sound effects. You can open the game up and order the tunes based on what order they are played in the game.

This Level here you might have learned a few things. I'm giving you a break so that you don't have to learn too many things here. You have a lot of catching up to do. The levels are going to get really hard soon slowly but surely, however you can handle it. The next level will feature another Technos Japan game and is much like this one. So company formats are a lot alike since most seem to use the same development kit in making the games and sound engines. Have fun listening to the rip.

LEVEL 5 Technos Japan River City Ransom

We could use nes2nsf and be able to do the rip and use some of the same techniques that was used in level 4. However I have a new technique to show you and that's using an emulator debugger with a dump function - Nesten.

Open Nesten and load the rom River City Ransom and while the music is playing open the debugger. Go to view and activate the debugger there. Set a write breakpoint for $4000 - $4009. So type in $4000 in the start location and $4009 in the end location. Next you click memory write. After you do this the debugger snaps on the code. And then you dump ram. This is a range of the sound registers. You do this so that you trap the right banks and then you click the Dump $8000 - $FFFF which is the program rom section.

Also note the stack contents on the left. I write these bytes down. These are addresses, these are all the routines that you have executed but have not returned to yet. So everything that was done before is logged so to speak. Write these locations down and subtract -2 from everyone, also flip the bytes around. D784 = $84D5, 7481 = $8172, 5FFA = $FA5D. These are the CPU addresses that have been executed in the sound driver. $FA5D is the entry point of the play address and is more then likely in the NMI code. Also note where the write was executed. At $07F8 in this case which is in WRAM and not PRG as you might expect.

The dump file was extracted to the saves folder in the Nesten folder. Go to that folder and copy/paste the dump file to the main directory where all your tools are. Next you disassemble the file and then load the file in edit from the edit.bat and then you're ready just about. Also load the file in a hex editor. Also load another NSF in the hex editor so that you can get the header from it and transfer it to the dump. This is simple. I highlight 0h - 7Fh in the hex editor and copy it. Then I go to the dump file and paste the header at the beginning of the file. So you are ready to view the disassembly and to edit the NSF at the same time.

Before going farther I want to talk about interrupts for a moment. In the NES you have 3 interrupts and these are located in a vectored table at $FFFA - $FFFF in every NES rom. These are what you could call pointers and do indeed point to the code address locations for the interrupt function. They are as follows.

- NMI $FFFA - $FFFB - Non-Maskable Interrupt. Updates screen on vblank. Also updates sound hardware most of the time.
- Reset $FFFC - $FFFD - Triggered when machine is powered on. Sometimes has sound code here.

- BRK/IRQ $FFFE - $FFFF - Very rarely used. Can have sound code as well.

As it says in the NES Music Ripping Guide by Chris Covell, that the entry to the play code is usually in the NMI code and this is almost true everytime. Sometimes you will not find the play code here. This is why it is a good idea to write down the contents of the stack to notepad or something.

First of all look at the disassembly of River City Ransom and look to see where the music driver code is. The code is in the $8000 region of the PRG. Next you want to look in the NMI code but first you have to look in the bank to get the location of the NMI. Remember the NMI vectored location is $FFFA - $FFFB. Use this formula to find the NMI pointer in the NSF. FFFA - 8000 + 80 = NSF Offset. The answer is 807Ah Offset. So you look at these 2 bytes there at that location and flip them around. This is your NMI location and you should look in the disassembly at that location which is $FF4F in this case.

Look at $FF4F in the disassembly. This code is rather tricky, however I'm not going to explain it much. I will show where the code leads to.

```
$FF4F> 2C 0001: BIT $0100 ; test bits, A & M,N=M7,V=M6
$FF52> 10 08: BPL $FF5C ; will not branch
$FF54> 50 03: BVC $FF59 ; will branch if updating audio hardware
$FF56> 4C 68F1: JMP $F168 ; jump to the main NMI code.
$FF59> 4C 2BF1: JMP $F12B ; go to call to play routine. updates audio
hardware.



$F15A> 20 54FA: JSR $FA54 ; leads you here.
```

$FA5D> 20 0380: JSR $8003 ; leads you here. This is the entry to the play address here. This address is in the stack contents when you copied it down earlier. $8003 is the play address. Put this address in the header. This is how you find the play address from a disassembly and emulator debugger. Tracing the NMI code can be a real pain sometimes so do it more then once if you have to because the code can lead you in tons of different address directions.

Now that you know what the address is for the play code you can get to work on the init code. We are going to do the same thing we did in level 4. Switch the JMP's and write some code. Here is the code in the NSF bank before you change it.

```
$8000> 4C 2080: JMP $8020 ;
$8003> 4C 4681: JMP $8146 ;
$8006> 4C 00B8: JMP $B800 ;
$8009> 4C 33B8: JMP $B833 ;
$800C> 4C AFBA: JMP $BAAF ;
$800F> 4C 8DB8: JMP $B88D ;
$8012> 4C D2B8: JMP $B8D2 ;
```

```
$8015> 00: BRK  ;
```

Change the above code with the following below.

```
8000 : 4C 46 81  JMP $8146  ;

8003 : 48  PHA   ;

8004 : A9 00  LDA #$00  ;

8006 : 8D FF 07  STA $07FF  ;

8009 : 20 20 80  JSR $8020  ;

800C : 68  PLA   ;

800D : 8D FF 07  STA $07FF  ;

8010 : EE FF 07  INC $07FF  ;

8013 : A9 0F  LDA #$0F  ;

8015 : 8D 15 40  STA $4015  ; Sound Enable

8018 : 4C 20 80  JMP $8020
```

So now the NSF plays and sounds real good. Technos Japan has a cool sounding driver. A lot of the games sound the same and could possibly be ripped in the same way. I've seen several of them that I could have ripped easily. Now arrange the tunes. You will notice that the first tune is silence. I have a trick for skipping those silence tunes if there is no silence tunes anywhere in the rip. The silence tunes have to be in order at the front of the NSF in order for this code to work.

```
TAY
INY
TYA
JMP $init ; this is for skipping 1 silence tune
```

```
CLC

ADC #$nn ; nn is number of tunes you want to skip

JMP $init
```

One last thing you can do. You can optimize this rip by removing a section of the NSF. Remove $C000 - $FFFF, that's 4080h - 8080h Offset in the NSF. So that

leaves a 16KB rip after you remove the excess data because you don't need it. You can arrange the tunes at the end or somewhere where there is free space.

So this wraps this level up quite nicely. Don't forget to fill out the other header information.

LEVEL 6 Hudson Hector 87

Use nes2nsf to extract the bank from hector 87 and the bank extracted will be 005. Go ahead and try to play the rip and you notice that it will not play. A NSF at these harder levels will never play with the NSF address calls set at 8003/8000 for init and play respectively. So your first task is to find the play address that I showed you in level 5. Before you do this remove the header and disassemble the game.

This time you don't have to use Nesten to debug because you have already extracted the right bank. Use FCEUD and bring up the debugger while the music is playing. Set a write break point for $4000 - $4009 and wait for the snap. Write the stack contents down. Remember you have to flip the bytes and subtract 2. Next you will want to find the NMI interrupt routine. With FCEUD it's easy. First disable the sound write break points and then type in NMI and check the execute box and click run and let it snap. The debugger snaps on $DB5E at the start of the NMI. Now if you was paying attention you will have noted where the music code is and the music driver is located in the $8xxx area of PRG. You want to look for a JSR or JMP that goes to this area. So first look at Address $DB8E, this was one of the addresses in the contents of the stack. Check the code there. $DB8E - JMP $8009. So $8009 is the play address. Place this address in the header in the play area.

The play address call is usually the easiest to find. The init address call is another story. The init initializes memory addresses,sets up data blocks,changes the tune. However you can sometimes find the init address by setting a write break point to 4015 in an emulator debugger before the music starts. So you write down the stacks contents like you did for the play and look at your disassembly and try out the addresses in the header and see if they work. Usually this doesn't work. What you might find most of the time is a sub routine that is a bootstrap code that initializes the sound registers and that's about it, this will not make the tunes play unless you add it to the init routine that you found. There is a few different types of init routines and the one we are going to focus on is the type that uses a memory address to switch the tunes. First I need to explain memory addresses to you.

Memory addresses are used to store numbers and use them again and save them for later whenever the code needs them. So usually the accumulator contents is transferred to these addresses and the accumulator loads from these addresses as

well. The data in addresses is also manipulated in other ways as well by the code. I'm going to show you the memory map of the WRAM(Work RAM) page by page.

- Zero Page - $0 - $FF
- Page 1 - $0100 - $01FF ; used for the stack mostly
- Page 2 - $0200 - $02FF
- Page 3 - $0300 - $03FF
- Page 4 - $0400 - $04FF
- Page 5 - $0500 - $05FF
- Page 6 - $0600 - $06FF
- Page 7 - $0700 - $07FF

So the range of WRAM is $0 - $07FF. The entire game uses these addresses and you have to figure out which is used to switch the tunes. I have a couple of tricks to use to discover this address. Using a memory viewer helps with this and FCEU(not the D version) and VirtuaNES has a memory viewer. Nesten and FCEUD has a cheat console. And you can view the disassembly to help you.

First check this disassembly around the play address. $8009 leads to $8585. So you want to check this area in the code $8585 - $86AA. Write down all the Zero Page addresses that are used in the code. Yes, commonly $0 - $FF is used to switch the tunes, however this isn't always so, be on the lookout. Write down these addresses.

- $FA ; used several times
- $FB
- $F2,X ; $F2 + contents of the X register

So you narrowed down the list some. Run the game and open the memory viewer. Note the contents of the first address $FA. So the title screen is playing, now go start the game and get on a level or something and see if the tunes changes. Usually this number is going to be low, right in the range of 00 - 30 in hex about. This looks like it's the right address. The next thing you have to do is find the right code and this isn't as hard as you think. Open FCEUD and set a read or write breakpoint to $FA, try them both individually and note the code addresses it snaps on and write them down. This does take some time however to deduce the right init address even with the right memory address.

Eventually you will see the following code and this is the right address.

```
$85F2> A5 FA: LDA $FA  ; load tune to A, could be stuck on 0
$85F4> 09 80: ORA #$80 ; OR A
$85F6> 85 FA: STA $FA  ; store results to tune address
$85F8> 0A: ASL A  ;
$85F9> 0A: ASL A  ;
$85FA> 0A: ASL A  ;
$85FB> A8: TAY  ;
$85FC> A2 07: LDX #$07 ;
$85FE> B9 D1A2: LDA $A2D1,Y ;
```

```
$8601> 95 F2: STA $F2,X  ;
$8603> 8D 7901: STA $0179 ;
$8606> 88: DEY  ;
$8607> CA: DEX  ;
$8608> 10 F4: BPL $85FE
```

Use 85F2 for the init address and you find out it still doesn't play. Well, you have to initialize this address first. The simple solution is to build a tune arrangement code and store to $FA and then jump to $85F2. Do it like this.

```
$C000   AA       TAX
$C001   BD 00C0  LDA $C009,X ; point to tune index
$C004   85 FA    STA $FA     ; store to tune address
$C006   4C F285  JMP $85F2   ; go to init
.db tune index
```

Place the code wherever you want but you have to point the LDA $xxxx,X right on the tune index. Count the bytes from the start of the code that you wrote and add that number to the address of the start of the code. Look and see the above code to see how I did it.

So you learned one way of how to find the entry point to the init code and learned about memory addresses. And now that you did all that work you should listen to the tune for a few moments.

LEVEL 7 Nintendo Wrecking Crew

You can use 2 methods in ripping this game if you want. nes2nsf or using Nesten to dump the 32KB bank. If you use nes2nsf to rip the bank then you have to remove the header in order to disassemble, and you have to change the origin to $C000 in the asm.bat file.

Next you want to find the entry to the play address. This is the easy part of the rip and you should find it quick. I found the entry to the play by using the stack. Here is the code to the play routine.

```
$F2E6> A9 C0: LDA #$c0 ;
$F2E8> 8D 1740: STA $4017 ; [NES] Joypad & I/O port for port #2
$F2EB> A5 F0: LDA $F0  ;
$F2ED> 4A: LSR A  ;
$F2EE> B0 F3: BCS $F2E3  ;
$F2F0> A5 F1: LDA $F1  ;
$F2F2> 4A: LSR A  ;
$F2F3> B0 CA: BCS $F2BF  ;
$F2F5> AD FD07: LDA $07FD ;
$F2F8> D0 18: BNE $F312  ;
$F2FA> 20 DCF3: JSR $F3DC ;
$F2FD> 20 00F6: JSR $F600 ;
$F300> 20 21F7: JSR $F721 ;
$F303> 20 7EF8: JSR $F87E ;
$F306> 20 3EF2: JSR $F23E ;
$F309> A9 00: LDA #$00 ;
```

```
$F30B> 85 F0: STA $F0  ;
$F30D> 85 F1: STA $F1  ;
$F30F> 85 F2: STA $F2  ;
$F311> 60: RTS  ;
```

This is the entire play code nice and neat. You see it starts out with the following code.

```
$F2E6> A9 C0: LDA #$c0 ;

$F2E8> 8D 1740: STA $4017
```

This is what most old Nintendo games do, except that most of the others also write to $4015. You can sometimes find the play code and not debug for it at all.

The real problem with this game is that the driver doesn't have an init code and you have to design one. So here is how it works. You go to a level and it loads a tune number into a certain address, once the play address is called then the code checks the address to see if the number has changed and if it has then the play address updates the hardware and the tune.

So you want to figure out how to design an init routine. This is not one of the more easy things to do. First of all you need to find out what memory addresses are used for the tune. You won't find these using a memory editor because the addresses are updated all the time like crazy, everytime the NMI is triggered. So look at the bottom of the play routine and notice this code here.

```
$F309> A9 00: LDA #$00 ;
$F30B> 85 F0: STA $F0  ;
$F30D> 85 F1: STA $F1  ;
$F30F> 85 F2: STA $F2  ;
$F311> 60: RTS
```

These are your memory addresses you are looking for. Yes, all the old Nintendo games are the same way and you can find these addresses right at the end of the play code. Simple as that. What you do is make a tune arrangement code routine and start with one address at a time and find out what the tunes are. You will find out that one of them is for looping tunes, the next is for non looping tunes and the last is for sound effects. Write down the number for each for tune and convert to hex. You should make the following code to figure out what the tune numbers are and change the memory address write to the next one when done.

```
85 F1:  STA $F0
60:     RTS
```

So you find out F0 is for sound effects, F1 is for most non looping tunes and a couple looping and some sound effects. F2 is all sound effects and these are better then F0.

What you do next is look for some code in the game before making an init routine. Using read and write break points in a emulator debugger you eventually end up with this code. Ignore any reads and writes in the game play code. Only within the sound driver range.

```
$F87E> AD DB07: LDA $07DB ;
$F881> 4A: LSR A  ;
$F882> B0 30: BCS $F8B4  ;
$F884> 4A: LSR A  ;
$F885> B0 22: BCS $F8A9  ;
$F887> A5 F1: LDA $F1  ;
$F889> 4A: LSR A  ;
$F88A> 4A: LSR A  ;
$F88B> B0 EA: BCS $F877  ;
$F88D> 4A: LSR A  ;
$F88E> B0 12: BCS $F8A2  ;
$F890> 4A: LSR A  ;
$F891> B0 5A: BCS $F8ED  ;
$F893> 4A: LSR A  ;
$F894> B0 3C: BCS $F8D2  ;
$F896> 4A: LSR A  ;
$F897> B0 27: BCS $F8C0  ;
$F899> AD F307: LDA $07F3 ;
$F89C> D0 01: BNE $F89F  ;
$F89E> 60: RTS  ;
```

This is part of the play code and is rather complicated. This determines when the tune address is changed for F1. Also you note the address $07DB which you must also use in your init code because you must initialize this address to work with this code here.

Now is the time to design some code for the tunes and all the addresses must be in the same code routine. If you use one address to play a tune you must silence the others. You do this by arranging the data. So here is the code you can use that works for multiple memory addresses.

```
ASL
ASL
TAX
LDA $FFA0,X
STA $F0
LDA $FFA1,X
STA $F1
LDA $FFA2,X
STA $F2
LDA $FFA3,X
STA $07DB
RTS
.db see rip for data ordering
```

This code is set up like the actual rip with the addresses pointing to the tune index. Remember that when you have a setup like this you would for example load a byte into $F0 and silence the rest out to play a tune. Then you would play the next tune and it would silence $F0 and load a tune byte into $F1 and so on. In order to use the code this way you would have to arrange the data and this takes some practice to get it right. For instance 00,08,00,00,00 this silences F0,and F2 and loads F1 with a tune. This is also based on what the contents of the X register is as well. Look at the actual rip and check out the code and data ordering and play with them some. The rip out there is not ordered right anyway, you could fix that up if you wanted to.

This is the end of the level and you learned how to deal with a complex situation where you use more then one address for the tunes. Also that this is the way older Nintendo games store their music driver. You could do many Nintendo games this way.

```
TEST:
------
```

Rip Urban Champion. This rip will require a bootstrap code along with using 3 memory addresses, you must silence one of them. I will only tell you this. The tunes are 10,20,08,04,02,40,80 and the sound effects you have to find. This test requires methods from a couple of different levels so have fun. You must use this level's method in order to get the sound effects with the tunes. Some sound effects are in the tune address. This NSF hasn't been ripped yet.

LEVEL 8 Chunsoft Tetris 2 + Bombliss

Wow the music in the game is super cool and that's because the music was composed by Koichi Sugiyama, the same guy that made Dragon Warrior/Dragon Quest music. This game is mapper 1 and so it's like mapper 2 in a lot of ways that we already worked with. $8000 - $BFFF is where the 16K banks are switched in and out. $C000 - $FFFF is the hardwired section of the game. The upper 16K which is the hardwired bank is what you could call the framework of the rom and calls the banks for different levels,music driver,takes care of the interrupts and anything else. As you read here in this doc and The NES Music Ripping Guide that the entry point to the play will be found in the NMI and so this is right and is found in the hardwired bank and so is the entry point to the init but you can't find it because it's using an indirect routine which is a real pain to find even if you're experienced. You can use nes2nsf and get the right bank which is 005 in the rom. However your best bet is to use an emulator and dump $8000 - $FFFF because you need to disassemble the entire space to find the entry to the init code.

Indirect jump code is difficult to find like I say. However you need to understand it first to be able to find it. JMP ($xxxx) is Jump Indirect. You don't jump to the

address in the parentheses you read the bytes at that address and you flip the bytes around for your address to jump to. Usually these bytes are in PRG somewhere, you have to initialize these addresses before you can make the indirect jump so you load 2 data pointers into the address stated in the indirect jump and then you run the operation. A lot of games use this method. This game uses a slightly different method and is the same in principle.

You have a code routine that is used many times for many different things. What you do is allocate code in zero page area of memory and you JSR to the WRAM area and then JMP to another routine. It's different in that you don't use a JMP indirect operation. I will paste the following code for you and you can check it out. This is the indirect routine.

```
$CEAF:85 20     STA $20 = #$B8
$CEB1:86 21     STX $21 = #$01
$CEB3:AD 04 03  LDA $0304 = #$05
$CEB6:48        PHA
$CEB7:08        PHP
$CEB8:AD 04 03  LDA $0304 = #$05
$CEBB:8D 15 03  STA $0315 = #$04
$CEBE:20 DE CE  JSR $CEDE         ; go to initializaton
$CEC1:A9 4C     LDA #$4C          ; initialize JMP op
$CEC3:85 23     STA $23 = #$4C    ;   "         "  "
$CEC5:A6 21     LDX $21 = #$01
$CEC7:A5 20     LDA $20 = #$B8
$CEC9:28        PLP
$CECA:20 23 00  JSR $0023         ; indirect



$0023:4C 42 89  JMP $8942         ; init code
```

This code might be confusing at first until you understand it. Addresses $23,$24,$25 are initialized somewhere, $23 is initialized in this routine. You JSR(Jump Save Return) to address $0023 and execute the code that was initialized there in WRAM. In this case the code leads you right to the init routine.

You might want to know how I found it. It was all guess work for the most part. I always check all the interrupts. In this case I checked the BRK/IRQ interrupt and did an address break for $CF2C. I checked zero page for code and sure enough I ran into some code in this area. I kept debugging this code at different times to see if I would run into the init code and sure enough I did.

You know the drill, as always arrange your tunes. Count your tunes and fill in the composer and company names in the header. After all this have fun listening to the rip. The addresses should be 8000/8942/8000 load/init/play respectively.

LEVEL 9 Cony World Heroes 2

World Heroes 2 is a pirate game and uses a method that requires you to build an init routine for the NSF. You can chose 2 methods to build the routine. However I've chosen an indirect jump routine to make things really simple. Go ahead and dump ram and then disassemble the bank. Also debug the code for the play address. The play address is $80F5 and right above this code is $80CB. This is a bootstrap code and is part of the init. You can try it in the header and it won't play. You will notice that this bootstrap will be called everytime a tune plays so you will have to search for it. You can do it the long way or the short way. The long way is playing the entire game and debug the code, however you might not get all the tunes this way. I have a method that's quick and easy. Load the disassembly in notepad or whatever text program you use with a search function and search for $80CB. You will find many matches in the $Cxxx area of PRG and you will writes these down as you go. This first match you get is at $C14D, however you want to load the X and Y registers as well so you write down $C149 as your first address.

```
$C149> A2 DB: LDX #$db ;
$C14B> A0 8C: LDY #$8c ;
$C14D> 20 CB80: JSR $80CB ;
$C150> 60: RTS



$C241> A2 94: LDX #$94 ;

$C243> A0 87: LDY #$87 ;

$C245> 20 CB80: JSR $80CB ;

$C248> 60: RTS



$C4F7> A2 C3: LDX #$c3 ;

$C4F9> A0 8D: LDY #$8d ;

$C4FB> 20 CB80: JSR $80CB ;

$C4FE> 60: RTS
```

These are just 3 matches you get when you search in the disassembly and you have a lot more to find so go ahead and look for them. Once you're done doing this then you have to make an init routine. This is going to be your indirect routine.

```
C000   0A        ASL   A
C001   AA        TAX
C002   BD 10C0   LDA   C010,X
C005   85        STA   00
C007   E8        INX
C008   BD 10C0   LDA   C010,X
C00B   85        STA   01
C00D   6C 0000   JMP   (0000)
```

```
.db 49,C1,41,C2,F7,C4,ETC
```

You have to figure out 2 memory address that you are going to use for the JMP indirect and the ones that I used are $00 and $01. I used a memory viewer to make sure the addresses was not used by the sound driver. It's ok if you use addresses used by anything else as they won't be used in the NSF. Remember those addresses you wrote down. Well, these are going to be your data that is pointed to by the above code. You flip the address bytes and you add them at the end of the code just like you would for arranging the tunes except that you have 2 bytes instead of one. If you understand level 8 then this should hit you like a tun of bricks.

LEVEL 10 Human Creative Egypt

Egypt is a puzzel game that gets rather hard at the end where it kicks you totally. This game also has Egypt sounding music and is rather cool sounding. This NSF is rather hard to rip, took me quite awhile to figure it out and the solution wasn't as hard as I thought. When you rip the NSF and find the right init and play addresses it crashes in some players and won't work in others. When you load the game up in a emulator and set a write for $4000 - $4009 and you end up seeing the writes to the sound registers in the WRAM area. The first thing you should check is if these memory addresses are being initialized in the init code and at the right time. So I shall show you the code here that is in the rom.

```
$01D8:8D 00 40  STA $4000 = #$FF
$01DB:60        RTS
```

This code is used over and over again and the sound register address is changed a lot. 8D 00 40 the 00 byte is changed all the time and is updated in the play code thousands of times. You probally know by now that the init code initializes memory addresses and you need to trace the init code once you find it. I would check the sound registers first when you have them in the WRAM area and then go from there. In this case the sound registers are not being initialized and so you have to do them manually. So you need to make a loop to initialize these addresses. A countdown decrement loop might be best in this case. You should use the Y register and not the X register because the X register is used most often. Also you need to check the code to see that this routine doesnt mess anything else up by using the Y register. So here is the code to use and some data to go along with it.

```
FF70   48        PHA
FF71   A0 04     LDY   #04
FF73   B9 8CAD   LDA   AD8C,Y
FF76   99 D701   STA   01D7,Y
FF79   88        DEY
FF7A   D0 F7     BNE   $FF73
FF7C   68        PLA
FF7D   4C 6FB7   JMP   B76F
.db 60,40,00,8D
```

You can place the code wherever you want. However this is where I placed the code. And so this initializes the addresses and you can do this with other games as well that has register writes in WRAM. You can also initialize any other code that ends up here in WRAM in the same way. This is a simple level and so ends here.

LEVEL 11 Carry Lab Mystery Quest

This game is a pain to rip if you don't know how. You can use nes2nsf and you will get a 16K bank output but you won't know where the play address is and you won't be able to find the code with sound register writes. So you need to use a emulator and you make a breakpoint write to 4000 - 4009 and you never get a snap. This is because the sound registers are written to indirectly. Any emulator with a debugger will ignore indirect writes to any register because it could mess up emulation. So you gotta figure out where the play address is. So what you can do is either keep tracing the NMI code which could take hours sometimes or you can use Nintendulator with the trace log option in the debugger. So go ahead and use Nintendulator and while the music is playing log a trace for a moment or 2, don't trace for too long or you will get a gigantic log that's too big to open. Open this log and use the search function and search for 4000 or any of the other sound registers. If you get a match then you see more writes in the same place then you found part of the play address. If you don't find a match then you need to do the log over again. Write this down and then close everything. Now open FCEUD or Nesten and set a address breakpoint for the address where the sound register is being written to. Once you get a snap then you can dump the bank and look at the stack to get an idea where the play entry is and now you should be able to find where in the NMI code the entry to the play code begins. Now here is some part of the play code.

```
$F775:A9 00     LDA #$00
$F777:A0 02     LDY #$02
$F779:91 09     STA ($09),Y @ $400A = #$FF
$F77B:C8        INY
$F77C:91 09     STA ($09),Y @ $400A = #$FF
$F77E:E6 04     INC $04 = #$19
```

As you can see the sound registers are being written to indirectly and isn't much of a problem to find once you know how. There is lots of games out there like this. Init and play is F5D1/F664 respectively. I will explain optimizing in the next level and you could try to optimize it for practice.

LEVEL 12 Optimizing

Optimizing is a must if you want a good clean rip and a small file size that contains the music driver and no game data. After you have made the rip and make sure

every sound channel works and the header updated with name of game, composer, and company name you are ready to optimize.

If you use NES2NSF then you have partially optimized the rip because you have a 16KB file section and usually this contains all the data you need. Still you may need to trim the file more. Take the game 1943 in Level 1 and you can optimize this game. Look at the end of the NSF and trim off all the extra FF bytes in a hex editor. This only trims the file alittle bit. If you used this space for your tune arrangement code then don't trim this space.

Next thing you can do is disassemble the code if you haven't already. You notice the start of the music code is at $8000. You can look through the disassembly till you run into some code that is not part of the music driver. You will run into some code at $A902 so then you eliminate everything after. So how do I figure out how to find that address in the NSF. Simple, all you do is use this formula, address +80(header) - 8000 = NSF Offset. Eliminate everything after 2982h. If you arranged the tunes then you need to rewrite the code at the end of the music driver. Now load the NSF and you find out it plays. The NSF is nice and trim now that it's optimized.

Now you need to learn how to optimize a rip from an emulator dump if the music driver is past $8000. Let's use Wrecking Crew for an example. The play is at $F2E6 and the init is wherever you place some code. This time it's highly recommended that you place the code at $FF80 and eliminate the rest of the data after this code. The rest of the data should be all FF's. FF's or 00's are usually padded data at an end of a rom to make a bank add up to the size it's supposed to be. However you can get rid of this data. Now move on to the next step.

Now look at your disassembly and next you want to figure out where to eliminate the data before the sound driver. First you gotta figure out the start of the sound driver. Look at the play code and track every JSR or JMP that goes before the address $F2E6 and next you want to check any LDA $xxxx,X or Y that reads in the sound driver area. After checking all this you finally figure out that the start of the sound driver is about $F100 and so you eliminate everything before this address. So how do you find this address again in the NSF. This time you do it alittle differently. Take the header off first and then do this formula. address - 8000 = offset. The offset would be 7100h in the headerless NSF you have. Now eliminate everything before that. Do not remove the byte at 7100h. Now put the header back on the NSF and restore the init and play addresses. Next you will change the load address, yes that address that you have been wondering what to use for. Well, the load address is the start of the program and you're changing the start of the program to $F100 so that's your load address in the header. Remember to flip the bytes before entering into the header.

If you have any code that is writing to any other registers like $2006/$2007 for instance there is a slight possibility that the game is picking up data in the CHR(Character ROM) for the music driver but this doesn't happen very often. So you can zero out any data that is not sound code. Track the routine in the sound driver area and debug in an emulator if you have to so that you can track down the data as well and other routines in the sound driver. Make sure that you also zero out any bankswitching code. If the music driver uses this bankswitching code and the rip is not a bankswitching rip then you can either make it return via a RTS or NOP out the code. Any writes to anything other then ram or expansion rom or saveram or mirrored areas is illegal and must be changed. You can only write to sound registers, expansion sound chip registers, and NSF bankswitching registers, and RAM areas.

This is the end of basic optimization of a NSF. Next you have Paging Optimization and this is alittle more difficult but you can manage.

Paging Optimization

Paging optimization is a method to optimize 2 banks and shuffle one at the end and one at the beginning of the NSF. This makes for a much smaller NSF rip then most simple optimizations like stated above. You don't have to design any code for this as long as the code is linking the banks. All you do is switch the banks around and adjust the load address and the bankswitching bytes in the header and you're done. This is a prelude to bankswitching and in the long run will help you understand bankswitching.

You need to learn about the banks in a NSF first. NSF's have 4KB banks and there is 8 of them in the $8000 - $FFFF NES address space. Each bankswitching byte in the header at 70h - 77h controls a 4KB bank and tells the player what bank to load at what address location. Look to the following table.

- 70h - $8000 - $8FFF
- 71h - $9000 - $9FFF
- 72h - $A000 - $AFFF
- 73h - $B000 - $BFFF
- 74h - $C000 - $CFFF
- 75h - $D000 - $DFFF
- 76h - $E000 - $EFFF
- 77h - $F000 - $FFFF

next you can adjust the load address and you do this according to the spec, load address AND 0FFFh. This is to determine where on the fist bank to load the data.

Now we can get down and dirty with an example. Remember Level 3, we will use the game and it's Shanghai 2 for the example NSF to optimize. This first thing you want to do is get the rip that is 32KB in size and eliminate the banks that are not used in the game. We do this by figuring out what banks are used by debugging the code. You open the disassembly and look at all the code in the music driver and note the locations. All of the code is in bank 0 or $8000 - $8FFF area. Next you want to debug the code to find out where the data is. Start with the data that is not DPCM. You have the following code.

Find a particular address with an indirect read operation and debug for the code location like so. This would be for the first tune.

```
$8757:B1 81     LDA ($81),Y @ $933B = #$50
$8759:C8        INY
$875A:9D E2 06  STA $06E2,X @ $06E2 = #$00
$875D:98        TYA
```

Indirect Indexed operations are useful for transferring and loading large blocks of data and in this case is loading part of the sound driver data for the first tune. As you can see that some of the data is being loaded at $933B which is in page 1 of the NSF, $9000 - $9FFF. Next you can debug for some of the other tunes. In this case you can also check the init code.

```
$80AD:0A        ASL
$80AE:AA        TAX
$80AF:BD 9D 88  LDA $889D,X @ $889D = #$66
$80B2:85 81     STA $81 = #$00
$80B4:BD 9E 88  LDA $889E,X @ $889E = #$90
$80B7:85 82     STA $82 = #$00
$80B9:A0 00     LDY #$00
$80BB:B1 81     LDA ($81),Y @ $0000 = #$00
$80BD:30 E6     BMI $80A5
```

This code here loads data in page 1 and for the other tunes the data goes deeper into ROM. For tune 5 you load data at $AAA8 and is in page 2. Eventually you will run into data that uses page 3 as well. So now you know that banks 00-03 are used. Next you need to locate the DPCM data.

Finding the DPCM data is not hard in this game and as a matter of fact you can look at the NES Audio Ripping Doc to help you with that. Except that you don't have to move the samples, all you want to do is locate the bank the samples are in. Also look at the APU reference to get the registers for the DPCM and read up on them as well. Now locate the code.

```
$8774:0A        ASL
$8775:A8        TAY
$8776:B9 3D B4  LDA $B43D,Y @ $B43F = #$A0
$8779:8D 10 40  STA $4010 = #$FF
$877C:C8        INY
$877D:B9 3D B4  LDA $B43D,Y @ $B43F = #$A0
$8780:8D 11 40  STA $4011 = #$FF
```

```
$8783:C8        INY
$8784:B9 3D B4  LDA $B43D,Y @ $B43F = #$A0
$8787:8D 12 40  STA $4012 = #$FF
$878A:C8        INY
$878B:B9 3D B4  LDA $B43D,Y @ $B43F = #$A0
$878E:8D 13 40  STA $4013 = #$FF
```

This is the DPCM register write code and what you're looking for is the write to $4012 which will tell you where the samples are located at. $4012 is loaded with the byte A0. So you do the following formula. Contents of $4012 * 40 + $C000. So you do this A0 * 40 + $C000 = $E800. This is tune 1 and you can do it for the others as well. $E800 is located in page 06 and you will find out page 07 is used as well. So you use pages 00 - 03 and 06 - 07. Next you piece them together.

Each 4KB bank is 1000h in size and so banks 00 - 03 is in the range of 0h - 3FFFh. Banks 06 and 07 are 6000h - 6FFFh and 7000h - 7FFFh respectively. So divide the file in half and take the upper 4000h and divide that in half and take the upper 2000h and append that to the bottom 16KB at the end of the file. So now you have banks 00 - 03 and 06 - 07 in one file and you eliminated banks 04 and 05 from the NSF. Now you have one step to go. You have to set the bankswitching bytes in the header. Yes it's nice that you can use the bankswitching bytes for something other then bankswitching and paging optimizations is it. Since you relocated the banks, pages 06 and 07 become 04 and 05 in the NSF. You use the bankswitching bytes in the header to determine what pages are loaded into what 4KB address space. 00 - 03 is $8000 - $BFFF and 06 - 07 is $E000 - $FFFF. So you fill in the bytes at 70h - 77h in the header. 00,01,02,03,00,00,04,05 . You may notice that $C000 - $DFFF is not used, that's because we eliminated the banks that didn't have music data. You leave the load address the same for now because we have not changed the start of the program just yet. Next we are going to figure out the best banks to trim.

LEVEL 13 Free Lance Ripping

So far you've learned how to rip various NSF's based on company and many other basic methods that you can take with you no matter what NSF that you attempt to rip. You will not always be successful at ripping an NSF and maybe you can rip the NSF and it has emulation issues such as working in one emulator or player and not in another. Or maybe it sounds decent on one player and not the other, etc. It will be your job to try and make sure that a NSF is compatible with all players and emulators. Chances are that if your NSF plays in all NSF players and emulators, the rip will have a better chance of playing on the real hardware. Yes, there is a NSF hardware player designed by Kevin Horton, that's usable with CopyNES.

So now it comes to the point where you would like to rip NSF's from many different games. You can attempt to rip any NSF at will, given some determination and effort is given. Keep in mind the many different company formats, most of the time you can rip an NSF instantly when you run into such a game. Capcom, Data

East, Tecmo, Nintendo, Sunsoft, etc - these are the companies that use the same music driver format for nearly all of their games. There are exceptions and don't expect a game to use the same format. Debug and have fun with it.

Now it's time to mention the basics again, I will cover a few of them below.

Load address is the origin, start of the program code, beginnings of the NSF. The load address is commonly and most likely at $8000, which is the starting address of active ROM in memory for the NES gaming system. If you have trimmed data from the beginning of the bank, the load address must be adjusted in the NSF header. Also, according to the NSF spec in bank switching which is the second step in initializing the tunes which would use load address AND 0FFF.

Play address. Usually this is quite simple, the entry point is almost always found in the NMI interrupt routine, using an emulator like FCEUXDSP can help by setting break points to sound registers and in tracing the NMI code itself. When the entry point is not found in the NMI interrupt, you can trace back in the code from the sound register break point. Using the stack viewer in FCEUXDSP can help trace back in the code.

Init address. The elusive init address call is the life and death of a successful NSF rip in most cases. If you cannot find or design a proper init address call, the NSF will not play or work properly. An init routine is code that initializes RAM addresses, sets up the data blocks and calls the tune based on the number of the tune, commonly from an index or LUT (Look Up Table). A game can also have more than one init address, I've seen this in games and in some that use expansion sound like FDS rips for example.

Also a game can have what is called a bootstrap routine. This "bootstrap" code is usually accessed from the Reset interrupt or just before the main init called is called somewhere in the game code. There are also games that do not have an init routine to speak of. These type of games sometimes require more effort in discovering what addresses change the tune and you may have to initialize other addresses to make sure the NSF will play. Other than the RAM address that changes the tune, I've seen some addresses that needed to be initialized with 00 or even FF before the NSF would play.

You also have games that initialize the index registers, X or Y and then intializing some addresses that don't appear to be related to the sound driver somewhere deep in the game code. However, if you're familiar with the sound driver, you can possibly track these routines down from looking at a disassembly and viewing debugger code from FCEUXDSP. You can set up a table and load theses values using custom made code. You can also use a Indirect Jump routine to jump to these addresses in the same bank, again using custom code.

Those three address calls are most important to the NSF, so I mention them again in recap and some explanation even a couple years later after the first writing of this document as I've gained a better understanding of NSF ripping. I continue to learn more and rip NSF's that can be very difficult or as easy as many stated in this document. Now we can go over a brief analysis of the levels to understand what we have learned.

Level 0 - Gather tools, documents, emulators, NSF players, etc. I would also like to add to the list the following tools - FCEU MOD, Unofficial Nintendulator, FDS Explorer, NSF Optimizer, FCEUXDSP (NSF version updated by UGETAB) You can rip NSFs the old school way by using a hex editor, Nesticle and a disassembler (the way I used to do it). However, with the new tools available will help you to make better and more accurate rips, as well as the process being speeded up greatly.

Level 1 - This level shows you NES bank sizes and how to rip a NSF from a rom using NES2NSF, or using a hex editor to extract certain banks that contain the music driver. Then you prepend an NSF header to the bank and assign the Load, Init and Play address respectively to produce a playing NSF file. Also shows you to count the tunes and adjust in the NSF header.

Level 2 - This level shows you how to design a bootstrap code that inits a tune as well as games that apparently only have sound effects (there are other reasons too). Also shows you how to arrange tunes and sound effects. Be aware that many companies use the same driver format, like Data East for example. Filling out the header, with the name of game, artist/composer, company, etc.

Level 3 - This level discusses setting up a folder for your NSF ripping, as well as setting up DCC6502 to use in dissassembling the banks that you have selected to examine. A brief analysis of the sound registers which was taken from the APU Reference doc, nearly verbatim. This level shows you how to place your own custom init code in a certain place in the NSF. Also shows you bank joining, and in this case makes sure that the DPCM is present in the rip.

Level 4 - A little more of Level 3 ripping with a bit more hex editing and code that is more complex. Technos is the theme in this level, as that's the type of driver you're dealing with there.

Level 5 - This level once again deals with a Technos driver. You'll learn how to use a debugger and dump the contents of active memory that can isolate most if not all of the sound driver. Again in this level you'll be using some previous level techniques to rip NSF's.

Level 6 - This level teaches a few more debugging techniques using FCEUD, to deduce the play address more effectively. Discussion of memory addresses and

how to design a basic init routine where only one address need be initialized for the tune switch, and how to deduce the location of the main init routine.

Level 7 - How to deal with old Nintendo music drivers and how to deal with a game that uses several different addresses that switch the tunes. A way to design an init routine to make sure that you can include the music and sound effects in your rip.

Level 8 - This level shows you how to deal with indirect routines that have code stored in WRAM and how to trace and deduce address calls.

Level 9 - This level deals with a pirate game and one that uses tons of init entry points for the tunes and how to consolidate them using an indirect jump routine.

Level 10 - This level deals with initializing the sound registers in WRAM so that the NSF will play. Sometimes you will need to do this, other times you will not. Keep a close eye on the games that write to the sound registers in WRAM.

Level 11 - This level states a method where you can isolate and dump the correct banks that contain the sound driver when you're dealing with a game that writes indirectly to the sound registers. Back in the day, Nesten would not snap on a indirect write.

Level 12 - This is the optimization level. I have covered some methods briefly, including paging optimization which is a lot more work but trims the NSF as much as you can by trimming the first and the last banks based on how you're loading the banks into NES memory, respectively. A warning to all; you don't want to optimize too much as this can potentially comprimise some part of the rip. I've done this a few times myself and figured it out sometime later.

Using all of the previous levels, you can potentially rip nearly all non-bank switching games there are in existance. There are exceptions to this rule and of course you'll need to be inventive in dealing with some games. Also, I might add that if you're using this document to learn how to rip NSF's, if you've gotten to this point you could consider yourself a Level 13 NSF ripper :), assuming that you can use the methods described in this level.

Tricks of the Trade

We will examine the game **Final Fantasy 1**. We are going to use Unofficial Nintendulator in this example. Unofficial Nintendulator has a very useful debugger with some debug functions that are not available in other emulators. That is one reason why I use many emulators to rip NSFs.

First set a write breakpoint to the sound registers when you're on the first screen with music which happens to be the prelude I believe, I usually set a range $4000 - $4009 to be on the safe side. The following code will be shown in the debugger.

```
B0C1    8D 01 40        STA $4001 = #FF
B0C4    A5 BE           LDA $BE = #CF
B0C6    8D 02 40        STA $4002 = #FF
B0C9    A5 BF           LDA $BF = #02
B0CB    8D 03 40        STA $4003 = #FF
B0CE    A9 80           LDA #$80
B0D0    85 BF           STA $BF = #02
B0D2    4C E1 B0        JMP $B0E1
```

You can trace this all the way back to find the play entry address which is address $B000. Now the init entry point is what you want and here is what we will do. While the debugger is still frozen and emulation stopped, uncheck the breakpoint on the sound register writes. Now what you'll do is set a read break point for addresses $8000 - $BFFF. I have chosen this range because this is where the code is located at and this mapper is Mapper 1 which is a 16K bank sized select game. Sometimes data will be located in the $C000 - $FFF9 area, but not this game. You will get a few different snaps, one of them is for some misc sound data (that you can continue to debug if you wish) and other snaps are for the sound data being read that you're looking for, you'll have to click run a few times to see what I'm talking about. Now for the code.

```
B1A3    A0 00           LDY #$00
B1A5    B1 18           LDA ($18),Y @ $80E1 = #07
B1A7    C9 C0           CMP #$C0
B1A9    B0 11           BCS $B1BC
B1AB    20 50 B2        JSR $B250
B1AE    B5 00           LDA $00,X @ $00B0 = #E1
B1B0    18              CLC
B1B1    69 01           ADC #$01
B1B3    95 00           STA $00,X @ $00B0 = #E1
B1B5    B5 01           LDA $01,X @ $00B1 = #80
B1B7    69 00           ADC #$00
B1B9    95 01           STA $01,X @ $00B1 = #80
```

Let's see, by looking at the code, the data for this tune is at $80E1. If you continue to debug and track the data, you'll eventually find out that the data has a pointer table for each sound channel per tune.

Now for the good part of Unofficial Nintendulator. We have confirmed that the data is being loaded from the $8xxx range and the code is being executed at the $Bxxx range. We look at the PRG area on the debugger and note that you have 8 boxes and that means that each box is a 4K bank, perfect for NSF rippers. Each box will tell you the Offset of the bank loaded, where the bank came from in the ROM. $8000, $9000, $A000, $B000, $C000, $D000, $E000, $F000 - These are the boxes that are viewable and what do you know, What's in the $8000 box is Offset 34010 (yes, the iNES header is accounted for, don't add it), $B000 is 37010. After quite a

bit more debugging, I deduced that $8000 - $BFFF is used for this music driver, the end of the bank could be trimmed afterwards.

The point of using this feature is that you could select Offset 34010 - 3800F from a hex editor and extract this bank from the ROM, driver would be extracted and isolated. Also useful in picking out banks to use for bankswitching NSFs, as you will see later on in this document.

Out of curiosity and the fact that you could eventually debug to this point in the ROM, let's go to Offset 34010 in the ROM and take a look in a hex editor. What do you know, any person that does a lot of hex editing will immediately recognize the data as pointers. Ok, let's show ya a few bytes.

C080BD81BC820000C1820B835783

Let's break this data string down a bit. CO80, could that be $80C0? Sure enough, it is and it's a pointer to a string of data. Evenually you'll find out that a group of three pointers are for one tune and one pointer represents one sound channel. This game use Square 1, Square 2, Triangle channels. So we have $80C0, $81BD, $82BC. Let's set a read breakpoint to address $80C0 and what do you know, we have a snap when you reset the game. Here is the code.

```
B1A5    B1 18           LDA ($18),Y @ $80C0 = #FD ; This is what we are
looking for.
B1A7    C9 C0           CMP #$C0
B1A9    B0 11           BCS $B1BC
B1AB    20 50 B2        JSR $B250
```

So we found out where the data point is being loaded at and it's using address $18 in WRAM as an indirect read. The code that we are looking at is located at $B1A5, so then let's look up in the disassembly or the debugger that you're using and see what we can find. Ok, here we go.

```
$B050:B1 10     LDA ($10),Y @ $0018 = #$00
$B052:95 00     STA $00,X @ $0000 = #$00
$B054:C8        INY
$B055:B1 10     LDA ($10),Y @ $0018 = #$00
$B057:95 01     STA $01,X @ $0001 = #$00
$B059:20 89 B1  JSR $B189
```

Ok, we are hot on the tracks of the entry point to the init code, let's keep looking up and here is what we shall see.

```
$B007:29 3F     AND #$3F
$B009:85 4B     STA $004B = #$02
$B00B:A9 00     LDA #$00
$B00D:85 4C     STA $004C = #$00
$B00F:A9 00     LDA #$00
$B011:85 7E     STA $007E = #$00
$B013:8D 02 40  STA $4002 = #$FF
$B016:8D 03 40  STA $4003 = #$FF
```

```
$B019:8D 06 40  STA $4006 = #$FF
$B01C:8D 07 40  STA $4007 = #$FF
$B01F:8D 0A 40  STA $400A = #$FF
$B022:8D 0B 40  STA $400B = #$FF
$B025:8D 0E 40  STA $400E = #$FF
$B028:A9 30     LDA #$30
$B02A:8D 00 40  STA $4000 = #$FF
$B02D:8D 04 40  STA $4004 = #$FF
$B030:8D 08 40  STA $4008 = #$FF
$B033:8D 0C 40  STA $400C = #$FF
$B036:A5 4B     LDA $004B = #$02
```

This code is right after the BCC opcode, let's try address $B007 as our init entry point and what do you know, music comes out so nice and sweet, success is at hand in your NSF ripping. Have fun listening to the rip and don't even try to submit this rip. It was ripped by Necrosaro in 1999. This is the address chosen for the rip and we could shift the address up a bit to $B003 and use some additional code to conform to the original logic structure, but just the same the rip works just fine as it was ripped.

In summary this is a method to use in tracking down init addresses and/or to know where the data is located at in the ROM. Knowing where the data is located at can be useful in tracking down the banks for bankswitching NSFs. This method is very useful for NES music hackers as well.

Another trick of the trade is that when you have an NSF that just won't play no matter what address you chose for the init entry point, or what addresses you are poking. Well, you can open two instances of FCEUXDSP, one with the NSF and one with the ROM open and playing music. Open the hex editor from tools in the FCEUXDSP that is running the ROM and select a page from WRAM, copy and paste into the FCEUXDSP version that is running the NSF, chances are that if you have the play address correctly, the NSF will begin to play, but not always. You can tinker with this page and change values, you can then deduce what you need to look for in the disassembly, or in writing your own code to boot up the NSF. Make sure that you carefully choose which page you are pasting into WRAM, you should know which one to paste after looking at the disassembly.

If a given ROM is not supported in FCEUXDSP, then you can use FCEU MOD to dump any page of WRAM that you wish. Find a select area in the NSF and paste it there, make sure that you do not go out of bounds of the NES address range. You may have to delete an equal section in order to accomodate this new piece of data. In your custom init code, you will have to write some code in order to load the data back into the page from which you dumped it from. For example, this code will load the entire page back into Zero Page WRAM.

```
PHA
LDA #$00
```

```
TAX
LDA $8000,X
STA $00,X
INX
CPX #$FF
BCC lda_8000
PLA
RTS
```

There are many more tricks to figure out how to get an NSF to play and now I leave this up to you to figure these things out, hopefully you will be able to use what I have shown you to make some killer NSF rips. One more thing, make sure that you are very careful in your choice as to which game that you wish to rip, many have already been ripped and/or are imcomplete. UGETAB has made a list of NSF rips from various sites and collection files. I often refer to this list.

LEVEL 14 NSF Repair/Clean-up

A good deal of Level 14 can be used at any stage of the NSF ripper's current skill level. However, some things that I will show you in this level will require a bit of effort to make an NSF rip clean and as compatible with many different players and emulators as possible. You can use this level with your own rips and/or to repair other older rips or ones that are not quite complete or compatible with other players.

Making sure that a NSF header is clean and of precise size can indeed make the difference for a properly ripped NSF. As far as the size of the header is concerned, it's 128 bytes or 7Fh in hexidecimal, so the range of a NSF header is 0h - 7F in a hex editor. The data of the NSF starts at 80h, and if the header is not of proper size, the address space is thrown off by how many bytes the header is off or over sized at.

The next thing to check is that the NESM, 1A is in the right place in the NSF header, use the specs for reference. Also, the NSF version number must be correct, the NSF I'm looking at now states it's 01. The starting tune must always start at 1, the number of total tunes should also be correct and at the right Offset in the header. The next six bytes are the Load, Init and Play address calls respectively, you should check to make sure these are correct, they make the difference between a playing and non-playing NSF commonly.

Sometimes you'll notice that a rip plays a bit too fast. In this case it's a wise idea to check the header at location 7Ah. This is the byte that determines if the rip is NTSC or PAL, or even dual NTSC/PAL. 00 is NTSC, 01 is PAL, 02 is dual NTSC/PAL. It is important to make sure that 7B is set appropiately depending on what type of rip this is. Games that are designated as (E) are almost always PAL. Some Chinese pirate games are PAL or SECAM, SECAM from what I understand is a format

designed by the French or in use by the French that is near NTSC speeds, so SECAM can use NTSC setting, the speed can also be adjusted.

Depending on what type of rip the game is in regards to NTSC or PAL, you can adjust the PB (Play Back) speed in the header. By aware that some players ignore this setting. The setting should be 411A for 60HZ PB, for NTSC tunes, use the formula in the NSF spec to calculate a different HZ playback speeds.

Now it's time for Extra Sound Chip Support. If you have a normal NSF that does not use expansion sound, 7Bh in the header should always be 00. According to the NSF spec, depending on which bit is set in the byte that is at 7Bh, that determines what expansion sound chip is in use for the rip. To my knowledge, there are no commercial games that have used more than one expansion chip, that being said, only one sound chip will be used per rip and the header should reflect this. There are some Homebrew NSFs out there that use two or more expansion sound chips, the header is set appropriately.

Now we will deal with 70h - 77h, which are the bankswitching bytes in the header. The bytes in this range should always be all 00 if we are dealing with a non-bankswitched tune or a rip that is not using paging optimization. In a continuous NSF rip, starting at the load address, the 4K banks shall be loaded sequencially and therefore would require no bankswitching. If the rip uses bankswitching, then you would have to debug the rip and determine what banks are used and how they are loaded. You use the bankswitching bytes to set up the address space and then you write code that switches the banks based on the tune that you're playing. Currently this is out of the scope of this level but is mentioned so that 70h - 77h should be clean and set properly.

Last but not least is the name of the game, composer, company credit fields. These must be included in the rip and when one of these fields is not known, < ? > must be filled in this field. Also, I might mention that you have a limit as to how many characters that you can use for each of the three information fields, the last byte must always be null terminate or 00, cannot be over-written.

As I had stated earlier, having a clean and properly set header can make the difference in a properly ripped NSF.

The data section of the NSF should be clean and isolated from the rest of the game play code and data. Given that statement, the following should be known for having a clean rip. A NSF does not use the standard vectored interrupt table of the NES/6502 CPU, RTI should be changed to RTS should most of the init and/or play code be contained within any of the interrupts, or rewritten to properly isolate the sound code.

An NSF has the following illegal address ranges that must be changed and/or removed in order to conform to the spec and for proper playback. Commonly these illegal address ranges are write only, in some cases even reading can be considered illegal. For example, reading any of the graphical display registers is considered illegal.

- $0800 - $5FFF (Exluding NSF bankswitch registers)
- $8000 - $FFFF (Exception: FDS)
- $E000 - $FFFF Illegal area for FDS writes
- $2000 - $3FFF PPU/Sprite - graphical display/scroll, etc
- $4014 DMA Transfer
- $4016 Joypad controller 1
- Any cart/device specific register that does not have anything to do with sound and/or the NSF spec.

All data and code that is associated with the illegal address ranges should be changed or isolated for a proper rip, that is the main point of this section. Another example, if you have data in the sound driver that is being written and/or read in the $0800 - $5FFF range, these addresses must be changed to somewhere within the legal WRAM area, maybe to an addresses unmirrored counterpart, perhaps?

One big problem with many NSFs that don't play in all players and emulators, is timing out. When an NSF times out, this means that somewhere the NSF is stuck in a loop or in a bunch of data and does not return, nor is the operation halted as it's supposed to, for example in the init code.

I have listed a few reasons why an NSF could time out. Not to worry, usually there is a solution if you debug the rip. Here are the possible causes of timing out and not returning. Stack underflow or overflow, maybe the possibility of missing a PHA or a PLA, these two opcodes work in pairs commonly. If you are missing one of these opcodes in the code somewhere, add one where you think one is missing and see if this solves the problem. When you return from a routine, then you wind up in an address area that you had not intended which strands you out in the middle of no where in the middle of some data that continues on for seemingly forever. That is another possible cause of missing one of the PHA/PLA pair.

I have also ran into a few games where the game was not coded correctly, you would have for example JSR $xxxx, random data after the JSR. After the call was executed, of course you could possibly return and when you do, you run into a big mess. This is solved by remapping the call to some freespace and placing an RTS at the end in most cases. If the data is irrelevant to the sound driver, then you could place a RTS right after the JSR $xxxx operation.

Last but not least of this timing out problem is the infinite loop. You can change the code and NOP the branch out or add an RTS, this solves the problem but not always. Another solution is to limit the number of times that a game would loop. I sometimes do it by using a compare value, the compare value will be how many times the loop will be executed. The loop using the compare value will set a flag once the value is equal or greater. A branch afterwards will not branch and so you keep the code from going into an infinite loop. Try the NSF again to see if it times out.

```
CMP #$10
BCC cmp_10
```

This is the basics of the code and perhaps you may have to design it to be a little more complex to mesh with the code. There are other ways of dealing with infinite loop problems, it will be your job to figure this out. Also, you can use SlickNSF to log an NSF. If the log is too large to open in NotePad, I recommend PFE (Programmer's File Editor).

When ripping an NSF, I often check "Don't wait for Play Return" in Config 3 in NotSoFatSo. This way you have a better chance of finding the init/play address, you can clean up the routine after you find it. Make sure that you don't forget to uncheck that box to find out if the NSF will play, when it does play you'll know that your NSF plays without timing out.

You have other boxes and they are the following.

- Ignore BRK
- Ignore Illegal Opcodes
- Don't wait for Play Return
- Reset 6502 regs
- Ignore NSF Version Number
- Force $4017 write

The original purpose of this config was to provide some compatibility with many NSF rips, mostly the older rips. However, I use it to trouble shoot NSFs. If I need to check any of those boxes to make an NSF work, then I need to debug the rip and find out what is wrong. Sometimes a re-rip is in order in more extreme cases if an NSF does not work properly.

There is the possibility that an NSF is missing some songs. Sometimes correcting the LUT (Look Up Table) will help add these missing tunes. Sometimes a bank is missing that contains the data needed to play the tunes missing. It is your job to

figure out why the tunes are missing. If the NSF is missing some tunes, indicate the NSF as incomplete, later on the possibility of someone adding the tunes may decide to repair the NSF.

Last but not least is the frame sequencer. The frame sequencer often makes a big difference when activated by writing to $4017. Sometimes you'll notice a big difference when you force a write in NotSoFatSo's Config #3, or when writing some additional code in your init code. Many times $4017 is written to in a game to time the music in the game to a certain speed. You can read up on this register in a few documents about sound and NES architecture. In games where the sound driver does not have a write to $4017, I include in my init code. You can write 00h or 80h.

```
LDA #$80
STA $4017
```

LEVEL 15 Nintendo - Backgammon FDS

Now comes the time to learn how to rip NSF's from FDS disks. FDS (Famicom Disk System) is a NES add-on that uses disks. Lucky for us emulation people, many disks have been dumped so that we may play them. The FDS uses the 6502 instruction set, all of the NES specific address registers, in addition you have the registers that are concerned with disk loading and the operation of the unit itself. The great thing about the FDS system is that in addition to the NES APU, you have one FDS expansion sound channel. Many FDS games use this expansion sound channel, but not all will use this channel, be aware of this and browse the code to know for a fact.

The first thing that I am going to do is paste verbatim from a document by Disch, a list of sound registers for the FDS sound expansion channel. You can read the document by Disch and other FDS documents as reference if you do not know much about the FDS. The register list is as follows.

4040h..407Fh - Wave RAM - 64 x 6bit sample data (Read/Write)
Writes to these registers are ignored unless Write Mode is turned on (see register 4089h).

4089h - Wave RAM Control (Write Only) Bit7 Wave Write Mode (1=Stop Sound output & Allow to write to Wave RAM)
  Bit6-2 Not used
  Bit1-0 Master Volume (0-3 = 100%,66%,50%,40% = 30/30,20/30,15/30,12/30)

4082h - Wave RAM Sample Rate LSB (Write Only) Bit7-0 Lower 8 bits of the main unit's frequency (upper 4 bits in 4083h)

4083h - Wave RAM Sample Rate MSB and Control (Write Only) Bit7   Main Unit disable (0=Enable, 1=Disable Sound Output)
  Bit6   Envelope disable   (0=Normal, 1=Disable Volume/Sweep Envelopes)
  Bit5-4 Not used
  Bit3-0 Upper 4 bits of the main unit's frequency

Main Unit / Sample Rate: (per entry of the 64-entry wave ram)   F = 1.79MHz * (Freq + Mod) / 65536
   Mod = Frequency change based on the Modulation unit

If the 12bit frequency is zero, the Main unit is disabled (channel silent).

408Ah - Envelope Base Frequency (Write Only) Bit7-0 Envelope Base Frequency, Fbase=1.79MHz/8/N

Fbase used by 4080h and 4084h. Volume/Sweep Envelope are disabled if N=0.

4080h - Volume Envelope (Write Only) Bit7   Volume Envelope Mode (0=Volume Envelope, 1=Fixed Volume)
  Bit6   Volume Envelope Direction (When enabled / at specified rate)
           0=Decrease Volume by 1 (only if Volume>00h)
           1=Increase Volume by 1 (only if Volume<20h)
  Bit5-0 When Bit7=1: Volume Level (0-20h=Muted-Loudest, 21h-3Fh=Same as 20h)
  Bit5-0 When Bit7=0: Volume Envelope Rate, F=Fbase/(N+1)

The volume level can be set to 00h-3Fh by write with Bit7=1, this level is also used as initial volume when switching to envelope mode by setting Bit7=0.

In decrease mode, initial values 21h-3Fh are resulting delayed decrease; volume stays at maximum level until the value gets smaller than 20h.

4084h - Sweep Envelope (Write Only) Bit7   Sweep Envelope Disable (1=Disable)
  Bit6   Sweep Envelope Mode   (0=Decrease, 1=Increase sweep gain)
  Bit5-0 When Bit7=1: Sweep Gain
  Bit5-0 When Bit7=0: Sweep Envelope Rate, F=Fbase/(N+1)

4085h - Sweep Bias (Write Only) Bit7   Not used
  Bit6-0 Sweep Bias (signed 7bit; -40h..+3Fh)

Sweep Bias is a used by the Modulation unit in calculating frequency bend. Sweep Bias negative: Modulation unit will be bending frequency down. Sweep Bias positive: Modulation unit will be bending frequency up. Any write to Sweep Bias register resets Modulation Unit's address to zero. This address is used by the Modulation Unit when looking up entries written to the Modulation table (via $4088).

4086h - Modulation Frequency LSB (Write Only) Bit7-0 Lower 8bit of 12bit Modulation frequency

4087h - Modulation Frequency MSB (Write Only) Bit7   Modulation Enable/Disable (0=Enable, 1=Disable)
  Bit6-4 Not used
  Bit3-0 Upper 4bit of 12bit Modulation frequency

Modulation Unit: Modulation Rate (per entry of the 64-entry modulation table)   F = 1.79MHz * ModFreq / 65536

```
If the 12bit frequency is zero, the Modulation unit is disabled.

4088h - Modulation Table (Write Only)  Bit7-3  Not used
  Bit2-0  Modulation input
```
Writing to this register puts the value written at the END of the modulation table **twice**, and shifts each entry already in the table 2 places to the front. The first 2 entries of the Modulation table are shifted out and lost. old, old <-- ModTable_0 <-- ModTable_1 <-- ... <-- ModTable_63 <-- new, new
```
4090h - Current Volume Gain Level (6bit) (Read Only)

4092h - Current Sweep Gain Level (6bit) (Read Only)

4023h - 2C33 I/O Control Port  Bit0  Disk I/O   (0=Disable, 1=Enable)
  Bit1  Sound      (0=Disable, 1=Enable)


FDS Sound by Disch, Release 1, 07/14/2004, based on info from Nori.
```

Read this list and become familiar with the FDS sound registers, when you see any of these registers in a disassembly of one of the files in a disk, chances are that this game could be using FDS sound.

Now is the time to learn how to rip a NSF from a FDS game. We can use two methods to rip this NSF, one of them is dumping the active contents of memory in Nesten or FECU, and debug until you find the Play and Init address calls. However, we are not going to extract the NSF from the game in this way. The reason is because the skills you learn here will help you become accustomed to ripping NSFs from a disk that uses both sides of the disk and/or a game that uses two disks that use both sides of each disk. For now, you'll learn how to rip from a FDS game that uses only one side of one disk. A little history about this NSF rip that I done back in 08-03-2004. I had dumped this game using an emulator and the debugger, and now I will show you the new method of ripping this game. This method will also produce a much smaller and compact file for play.

The first thing that we are going to do is use FDS Explorer. FDS Explorer is a utility that enables you to browse the contents of a disk, on both sides and all of the disks of a given game. You can view the hex and disassembly of each file that you wish to look at.

File #	ID	File Name	Address	Size	Type
0	0	KYODAKU-	$2800	224	NT
1	0	SOUND	$C300	7417	PRG
2	0	BG-MAIN	$6000	18816	PRG
3	1	CP	$A980	4736	PRG
4	0	VECTOR	$DFFA	6	PRG

5	16	BG-DEMO	$A980	6528	PRG
6	32	BG-FINAL	$A980	1664	PRG
7	1	BG-CHR10	$BE00	1280	PRG
8	0	BG-CHR11	$0200	512	PRG
9	0	BG-CHR12	$0600	512	PRG
10	1	BG-CHR00	$0000	8192	CHR
11	17	BG-CHR20	$1B00	1280	CHR
12	33	BG-CHR30	$1400	1536	CHR

Using FDS Explorer, you can see that the FDS game Backgammon has many files on the A side of the disk. Your job is to figure out which file(s) contain the sound driver and then extract them. Usually this task can be a frustrating and tedious job to do, especially when you have to piece together the files and integrate them into one NSF file using code that you have to write. The task for making an NSF from Backgammon is about the easiest I've ever seen in my life.

The boot ID is 15 for this game. Any file that has a boot ID of equal to or less than the game will be loaded at bootup, always keep that in mind in every FDS game that you attempt to rip the NSF from. The file called SOUND has a boot ID of 0, is a PRG (Program RAM) section of the disk and has the starting address at $C300, file size 7417. Click on this file and view the hex and the disassembly from this program, carefully look at the code to determine if this is in fact the file that contains your sound driver. After examining the file, it is in fact the entire sound driver. Extract the file called SOUND into your NSF project folder as a .bin and then slap a NSF header on it. Set a load address of $C300, since that's the start of the music driver (and the file) and then you'll need to debug for the play and init addresses respectively, is an easy task for this game. Make sure that you set the sound expansion chip for FDS in the NSF header, the setting will be 04 in hex if you're wondering. What you have now is a compact sized file, that plays excellent FDS sound.

LEVEL 16 Tokuma Shoten - Clox - Famimaga Disk Vol. 4 FDS

Now is the time to take the next step in FDS NSF ripping. We are going to use the same method to extract an NSF from the disk as we had in level 15, even though you could rip the NSF from an emulator dump. We are dealing with a one sided disk that contains all of the data of the game, including the sound driver. Before we get started, let's examine how interrupts are handled in a FDS game. Interrupts are still pointed to in the same way, with the exception that they are dealt with

indirectly. FDS Interrupts, as far as I know are always located at the same address. Here they are as follows.

- IRQ: $E1C7
- NMI: $E18B
- RESET: $EE24

While you're at it, open a debugger and set a NMI execute or PC same breakpoint for the address $E18B, in the game Clox - Famimaga Disk Vol. 4. Here is the code that you'll get.

```
$E18B:2C 00 01  BIT $0100 = #$C0
$E18E:10 08     BPL $E198
$E190:50 03     BVC $E195
$E192:6C FA DF  JMP ($DFFA) = $D8A8
$E195:6C F8 DF  JMP ($DFF8) = $0000
$E198:50 03     BVC $E19D
$E19A:6C F6 DF  JMP ($DFF6) = $0000
```

More than likely the code will jump to $D8A8, in fact it does when I stepped the code. This NMI routine will lead you to the play code entry point somewhere. You could have easily also backtracked the stack in order to find the play address as well.

Now it's time to get started in the ripping process. As we had done in level 15, we will use FDS Explorer to examine the disk and to extract the file(s) that contain the sound driver. Here is the list of files to browse.

File #	ID	File Name	Address	Size	Type
0	0	KYODAKU-	$2800	224	NT
1	0	LOGODATA	$7020	11488	PRG
2	0	LOGOPROG	$D800	2048	PRG
3	0	LOGOCHAR	$0000	8192	CHR
4	16	CL100PRG	$8D00	21248	PRG
5	17	CL100CHR	$0000	8192	CHR
6	32	SAVEDATA	$8CC0	64	PRG

This time we don't have a file that is named sound, so it's not going to be so easy of a job this time, still not too bad though. Like level 15, this game has a Boot ID of 15, so examine all of the files with an ID 15 or less that are labled as PRG. If you had done any debugging prior to opening up this file in FDS Explorer and I suggest that you always do this, you would know that most if not all of the sound driver is located in the $7xxx range. You can also examine the disassembly of each file in

FDS Explorer. Look at the list and determine which file has the $7xxx range included. File #1, ID #0, LOGODATA, starting at $7020, size 11488 is the file that we are looking for. Extract this file as a bin into your NSF project folder.

First thing, before we slap a header on the file is that we will padd the beginning of the bank, get used to it, you may have to padd banks in other rips. This file's address starts at $7020, what we want to do is for this file to start at $7000, so you need to insert 20h at the beginning of the file. When you're done, if the first byte of the unmodified file still lands on address $7020, then you done the job right. There are other rips that you may have to padd the file at the beginning or end. The reason why you may have to do this is because an NSF is like a ROM file, it uses banks and not files like an FDS disk does. You may want to know what padding is, well padding is a bunch of bytes to fill in space that you won't be using and to fill up the rest of the bank. Commonly, 00 or FF is used for bank padding. 00 is often used in FDS games when it's an area that will be modified by a PRG RAM file that is loaded from the disk. I might mention that the FDS since it uses RAM instead of ROM for it's program, this means that the code and data can be self-modifying, anything in the $6000 - $DFFF range can be changed at anytime.

Slap a NSF header on your file, if you had done some debugging you would deduce that your load/init/play addresses would be the following, $7000/$7447/$747B, in addition to the init address, you have a bootstrap at $7400, $E0 is used to switch the tune. However, you can't use any of these addresses because you want to conform to the NSF spec and to make the NSF compatible to as many players as possible. Especially the load address has to be $8000 and/or greater. All the main address calls have to be equal to or higher than $8000, in the range of $8000 - $FFFF. We can do this by using the bankswitching bytes in the header and writing some simple code in the init.

As I had listed in level 12, Optimization, I will list again the bankswitch bytes and what they correspond to, with only one difference for FDS NSFs. We will use the registers as though we were optimizing, it's a trick.

- 70h - $8000 - $8FFF
- 71h - $9000 - $9FFF
- 72h - $A000 - $AFFF
- 73h - $B000 - $BFFF
- 74h - $C000 - $CFFF
- 75h - $D000 - $DFFF
- 76h - $E000 - $EFFF (FDS: $6000 - $6FFF)
- 77h - $F000 - $FFFF (FDS: $7000 - $7FFF)

76h and 77h can be used for both addresses, normal bankswitching and FDS bankswitching. What are going to do now is determine what banks are being used in this NSF. The following are used - $7000, $8000, $9000 currently before

optimization. The three banks are also in order starting with the $7000 bank. We set the load address at $8000, and then we set the bankswitch bytes in the header as follows: 01,02,00,00,00,00,00,00. This allows you to load below $8000, with minimal bank shuffling in the ripping process.

Now it is time to write the init code, you must find some space and I suggest that you do so higher up in the $8000 bank or somewhere in the $9000 bank. Here is how I suggest that you design your init code for this game, since you must also load the FDS $7000 bank into memory. Before we go any farther, a little about the bankswitching registers for FDS NSF, keep in mind that the bankswitching registers are a little different than normal NSF bankswitching registers, although you may still use the normal mode if you do not need to load anything below $8000.

- $5FF6 - $6000 - $6FFF (FDS Only)
- $5FF7 - $7000 - $7FFF (FDS Only)
- $5FF8 - $8000 - $8FFF
- $5FF9 - $9000 - $9FFF
- $5FFA - $A000 - $AFFF
- $5FFB - $B000 - $BFFF
- $5FFC - $C000 - $CFFF
- $5FFD - $D000 - $DFFF
- $5FFE - $E000 - $EFFF
- $5FFF - $F000 - $FFFF

Now that you see all of the bankswitching registers, only the FDS registers will be used and not for bankswitching but for loading in banks that are below $8000 in this level. Now I will show you some basic code that I will use for this game.

```
PHA
LDA #$00
STA $5FF7 ; load first bank in the file to $7000
JSR $7400
PLA
TAX
LDA $tune_index,X
STA $E0
JSR $7447
RTS
```

One more thing before we end this level. This is one method of loading banks below $8000, there are other methods depending on the game that you rip. You will notice that this game does not have FDS expansion sound, but the FDS byte is set in the header. The FDS header byte setting serves a two-fold purpose, one of them is obvious and the other is to use FDS bankswitching. So a rip may have the FDS set in the header but not have any expansion sound, chances are that this tune loads banks below $8000 and/or is in fact bankswitching. It is possible that a game has

expansion sound, loads below $8000 and is also bankswitching, so be aware of this fact.

LEVEL 17 Takara - Transformers The Head Masters FDS

Now we step up in difficulty, again we are going to rip a NSF from a FDS game called Transformers - The Head Masters. Here is where my insistance about ripping from the disk becoming useful, because you'll more than likely never be able to rip the NSF from this game without browsing the contents of the disk. What I am going to do is use emulator dumping in combination with disk browsing to rip this game. Load up Nesten or FCEU MOD with the game Transformers - The Head Masters and debug for the sound registers in the usual way and then dump $8000 - $FFFF, or $C000 - $FFFF in FCEU MOD. After you have dumped the contents of memory, disassemble the bank and debug for the play and init address calls respectively. You'll find them at C000(load)/C061(init)/C057(play) respectively, in addition a bootstrap at $C034, tune switching address $28. Write some init code and you'll find that you have a bunch of sound effects and one tune on side A of the disk. If you do the same for side B, you'll get about the same thing, a different tune and some sound effects. So it's time to browse both sides of the disk.

SIDE A

File #	ID	File Name	Address	Size	Type
0	0	KYODAKU-	$2800	224	NT
1	6	SPRITE	$0000	2144	CHR
2	7	*FIX*	$1000	4080	CHR
3	8	PROG.001	$6200	16384	PRG
4	9	PROG.002	$A200	10496	PRG
5	11	*TITLE*	$D280	2480	PRG
6	12	JMP-TBL	$DFE0	32	PRG
7	38	*LASTMP*	$C580	6720	PRG
8	38	PLLASTCG	$0980	1152	CHR
9	38	*LASTCG*	$1700	1408	CHR
10	48	*END-PRO	$D280	2560	PRG
11	48	*ENDSCG	$0980	320	CHR
12	48	*ENDFCG	$1700	960	CHR

Wow, we have a lot of files here, and no file that specifically says sound. The more you look at this pile the worse it looks and you'll find out soon enough and then the work load gets worse on SIDE B of the disk. This disk has a Boot ID of 15 like the others. Files 0 - 6 are loaded up at bootup, that is the first clue to locating the files that you need, the next one is that if you debugged and dumped active memory from one of the emulators then you'll know the address range of the sound driver code is located from $C034 - $C455. Check each file, the one that has sound driver code is File # 4. PROG.002, address $A200, size 10496, PRG RAM. You'll get no sound from this file, I debugged the file and come to find out, it's missing the sequence data. So I decided to debug the disk again to locate the missing sequence data. Set a write breakpoint range $4000 - $4004 in FCEUD. Click run a few times if you have to, to figure out where the sequence data is located at. It is known already that the sequence data is not located in the $A000 - $CAFF range.

```
$C18B:0A        ASL
$C18C:AA        TAX
$C18D:B1 32     LDA ($32),Y @ $D724 = #$B4
$C18F:9D 00 40  STA $4000,X @ $4000 = #$FF
$C192:E0 07     CPX #$07
$C194:B0 06     BCS $C19C
$C196:C8        INY
$C197:B1 32     LDA ($32),Y @ $D724 = #$B4
$C199:9D 01 40  STA $4001,X @ $4001 = #$FF
$C19C:68        PLA
$C19D:AA        TAX
$C19E:C8        INY
$C19F:60        RTS
```

I have snapped the debugger a few times and wound up in the range of $D5xx - $D8xx, I believe this is where the sequence data is located at. Let's take a look at the FDS file list again and note all of the files that could contain this address range and possibly the sequence data. They are as follows.

- *TITLE*
- *LASTMP*
- *END-PRO

One way to find out without playing through the entire game is to replace the known sequence data with one of the files on the disk. I have a feeling that we are already playing TITLE, on the title screen of the game, so we can try the file *END-PRO, address $D280 which seems more likely to contain the sequence data than the other file *LASTMP*, later on we will find out this file contains music as well.

Take the emulator dump and trim off off $8000 - $BFFF if you haven't done so already, or dumped it from $C000 - $FFFF. Next you'll extract the file *END-PRO from the disk into your NSF project folder. The starting address of this file is $D200, so in order to test this file, trim off $D200 - $FFFF from your emulator dump and then append this file, you'll find out that a different tune plays, so this

file is good, save it. You can then trim that file off that you appended to your emulator dump and replace it with the file *TITLE*, sure enough you hear the familiar title screen music once again. In the file *LASTMP*, go to address $D200 and copy and paste that address and the remainder of the file and append to your once again trimmed emulator dump, again another tune, so that's three tunes on this side of the disk. We need to tally up the results of SIDE B of the disk before we decide upon a strategy of ripping this game. Dealing with the fact that all three tunes' sequence data is located at the same address location, this could be a problem. Onward we go.

SIDE B

File #	ID	File Name	Address	Size	Type
0	16	PL_CGSET	$C680	1408	PRG
1	32	*RESMAP*	$CB00	5312	PRG
2	32	ENEMYCGE	$0980	1664	CHR
3	32	**RESCG*	$1700	1632	CHR
4	33	*BONUS *	$C580	896	PRG
5	33	*REMMAP*	$D280	3392	PRG
6	33	**REMCG*	$1700	944	CHR
7	34	*RSSMAP*	$CB00	5312	PRG
8	34	ENEMYCGS	$0980	1664	CHR
9	34	**RSSCG*	$1700	1120	CHR
10	35	*BONUS *	$C580	896	PRG
11	35	*RSMMAP*	$D280	3392	PRG
12	35	**RSMCG*	$1700	656	CHR
13	36	*RJSMAP*	$CB00	5312	PRG
14	36	ENEMYCGJ	$0980	1664	CHR
15	36	**RJSCG*	$1700	1440	CHR
16	37	*BONUS *	$C580	896	PRG
17	37	*RJMMAP*	$D280	896	PRG
18	37	**RJMCG*	$1700	944	CHR
19	64	NAMEPRO	$D600	1792	PRG
20	65	NAMEDATA	$0700	128	PRG

We have a total of 21 files on SIDE B of the disk, nearly twice as much as SIDE A, but not quite. Even though we have more files to deal with, the structure of the data is not too much different, nor are the file names. It's easier to deduce which files are the sequence data, because any file that contains the address range $D280 will be

the files that we extract, you can test those files out, as we had for SIDE A. Here are the files to extract from this side of the disk.

- 1 - *RESMAP*
- 5 - *REMMAP*
- 7 - *RSSMAP*
- 11 - *RSMMAP*
- 13 - *RJSMAP*
- 17 - *RJMMAP*

That's 6 music files on SIDE B, adding the 3 from SIDE A, that makes 9 tunes in total. I'm currious as to the files that are at $CB00, what is the data used for, before the verified sequence data, let's check it out. I have just verified that the data is not part of the sound driver, this fact may help us complete the NSF a little easier. You can also set a read breakpoint for the address range $CB00 - $D27F and check it out yourself, do this once you reach the level on SIDE B.

First of all, let's trim those files that start at $CB00, then we will calculate how much space these files will take in total. We have 9 files that are betweeen 2.5K and 3.31K, 3 of those files are duplicate songs, so we'll not use three files. Now we are going to set up the banks on the NSF, using the emulator dump, padd the last bank until it's reached it's last byte, $DFFF. The next thing that you want to do is add a bank that will be $E000 - $EFFF, insert 1000h in padding, 00's or FF's. You're going to use this bank for your code, the rest of the NSF does not have enough space for code and data indexes. Now you will set your NSF bankswitch header bytes up as follows, 00,00,00,00,00,01,02,00. 00,01,02 is what you're concerned with since you're using banks $C000, $D000 and $E000. Now you're ready to padd the 6 files and convert them into banks. Again in this level we are using a NSF paging optimization trick. In the next level is how we will deal with straight forward bankswitching according to the spec.

Since you trimmed your files earlier on, they should be clean as possible and ready to convert into banks. You have 6 banks, some of them are of the same size. For example, *TITLE * has a starting address of $D280, you want to make this file start at $D000. Simply add 280h to the beginning of the file with a hex editor of your choice, at the end of the file you'll have to use a calculator on some of them, this one I believe you only add 40h in bytes. Do this with each file and append them on to the end, one at a time. When you're done, you should have a file size of about 32896 with the NSF header prepended, that's assuming that you added 6 - 4K banks.

Now it's time to write some init code in the $E000 bank, my code is right at the beginning. What makes this rip kind of tricky is that sequence data for each of the 6 tunes starts at $D280, and they are of the same tune number which is 10h. Here is how I'm going to handle it.

```
PHA
LDA #$80
STA $4017
JSR $C034                   ; Bootstrap code
PLA
LDA $bankswitch_index,X
STA $5FFD                   ; Switch the bank in the $D000 - $DFFF range
LDA $tune_index,X
STA $28
JMP $C057                   ; Jump to main init

bankswitch_index: .db
0307060401050101010101010101010101010101010101010101010101010101010101010
10101

tune_index: .db 10101010101001020304050607080 90A0B0C0D0E
```

The code looks pretty simple, however there is more here than what meets the eye. As I said before, all of the tunes and not the sound effects have an index number of 10h, as you can see with six 10h's in a row, normally you would never see this in a normal NSF rip. The bankswitching corresponds to the tune index, if you change the first six bankswitch index bytes, that is how you order the tunes and how the banks are loaded at the same time, remember one tune per file that we converted to a bank. The rest of the 01's are to make sure the $D000 - $DFFF bank stays loaded while the sound effects play. Make sure that you set the header for FDS sound, indeed this rip has expansion sound. Enjoy listening to the music after doing this tedious job.

LEVEL 18 Bankswitching

As many know, the address space for active loaded ROM memory is $8000 - $FFFF. When you have more banks and data than what can fit into this space is when you need to use bankswitching. Also there is a situation where you have a few banks that need to be loaded into a certain memory range, like $8000 $8FFF for example. Many games have done this, such as Metroid, Zelda, and many many more.

However, all we need to do is use bankswitching for NSF files that have eliminated everything but the driver or music engine. Before we get into the process, I am going to describe bank sizes for you.

Bank Size Select	Hex Size	Address Range	Mapper
4KB	1000h	Variable	NSF
8KB	2000h	Variable	MMC5
16KB	4000h	$8000 - $BFFF/$C000 - $FFFF	MMC1

32KB	8000h	$8000 - $FFFF	AOROM

This chart will generically handle nearly any mapper. For example, MMC3 or mapper 4 is either 8KB or 16KB bankswize select. AOROM, which is listed is 32KB bankseized select, or what most people are familiar with, that's Mapper 7. There are other mappers that will not be handled well by the chart that I showed you. MMC2 or Mapper 9 is a good exmple. Mapper 9 has a 24KB fixed bank at $A000 - $FFFF, the 8KB bank is switched in and out at $8000 - $9FFF. Another example would be Mapper 42, this is a FDS port that has a fixed 32K bank at $8000 - $FFFF, a 8KB bank is switched in and out in address range $6000 - $7FFF. How you deal with these odd mappers depends on the sound driver and in some cases, takes some strategy in order to rip. MMC2 no longer has any games that need to be ripped. However, one may pop up in the future, so I mention it to be complete. For FDS port games, I suggest to set the FDS bit in the header and rip it as though it was a FDS game (if the sound data or code is $6000 - $7FFF).

Now it's time to start digging into a game to locate the banks. First off, I suggest to make every available effort to prevent a NSF from bankswitching, if it's possible. If you only have a string of code, then it should be moved near the sound engine core, located in some freespace somewhere. For example, I have ripped a game called Golden KTV. The game is standard ripping fair and quite easy until you find out that only 3 tunes plays along with some sound effects. Since this is a large scale game with a lot of tune selections in the game, it's logical to assume this game will be bankswitching. Assuming that you made every available effort to locate the tunes. Now, here is how I done it and that's by a indirect jump routine so that I could simply plug in the banks and use the same index number for all the different tunes.

Basic Initialization Start

```
00:F450:48        PHA
00:F451:A9 40     LDA #$40
00:F453:8D 17 40  STA $4017 = #$FF
00:F456:A9 00     LDA #$00
00:F458:8D 10 40  STA $4010 = #$FF
00:F45B:A9 1F     LDA #$1F
00:F45D:8D 15 40  STA $4015 = #$FF  ; turn all channels on
00:F460:68        PLA
00:F461:0A        ASL
00:F462:AA        TAX
00:F463:BD 70 F4  LDA $F470,X @ $F470 = #$F0
00:F466:85 00     STA $0000 = #$00            ; store low jump byte
00:F468:BD 71 F4  LDA $F471,X @ $F471 = #$F4
00:F46B:85 01     STA $0001 = #$00            ; store high jump byte
00:F46D:6C 00 00  JMP ($0000) = $0000
```

Intialize 1

```
00:F4F0:A9 04     LDA #$04
```

```
00:F4F2:85 02     STA $0002 = #$00  ; bankswitch variable 1
00:F4F4:20 20 F4  JSR $F420         ; bankswitch sub 1
00:F4F7:A9 09     LDA #$09          ; tune #
00:F4F9:4C 42 F4  JMP $F442
```

Bankswitch Sub-routine 1

```
00:F420:A5 02     LDA $0002 = #$00  ; bankswitch variable 1
00:F422:8D F8 5F  STA $5FF8 = #$FF  ; NSF reigster to load $8000 - $8FFF
00:F425:60        RTS
```

Intialize 2

```
00:F442:8D A6 07  STA $07A6 = #$00  ; tune variable
00:F445:20 9B CA  JSR $CA9B         ; jump to main sound driver
initialization
00:F448:60        RTS
```

This is how I set the NSF up after I realized that only the sequence data is being loaded at $8000 - $9FFF, the core of the sound driver is at $C000 - $DFFF, also has DPCM. I gave you a quick run down of the code first and then you start plugging in the banks. So, the game is UNROM or Mapper 2, which means 16KB banksize select, there are 64 - 16KB sized banks. So, safe to say, I used a file splitter and divided the ROM into 16KB sections and gradually worked on them to remove anything that wasn't needed. I also saved $F000 - $FFFF as my "hardwired" bank to write the code. Most of the tunes did not go over 4KB, those that don't use address range $8000 - $8FFF, a few others use $8000 - $9FFF. I loaded them using the following NSF bank registers, using them in the initialization code.

You may want to know how to load the banks, it's simple, you use the NSF bankswitching registers to load the banks at certain times. I will now show you the registers. No matter what strategy you use, these registers will switch the banks.

Register	Address Range
$5FF6	$6000 - $6FFF
$5FF7	$7000 - $7FFF
$5FF8	$8000 - $8FFF
$5FF9	$9000 - $9FFF
$5FFA	$A000 - $BFFF
$5FFB	$B000 - $BFFF
$5FFC	$C000 - $CFFF
$5FFD	$D000 - $DFFF
$5FFE	$E000 - $EFFF
$5FFF	$F000 - $FFFF

Next is the bankswitch bytes in the header, you can use these to preload certain banks at reset of the NSF. You can learn more about them in Level 12. However, you can use them here too and that's to help you get rid of any banks that you don't need. If you use all of the core banks, you can sequencially number them 00h - 07h. Now, you may wonder how I tracked down these banks. That is also simple for this rip, is why I chose this game for the tutorial. Mapper 2 or UNROM switches banks by the following register, $8000 - $FFFF, you write to any address in this range to switch the banks, depending on the number written. So, set a write break point to $8000 - $FFFF, here is the code I got from it.

```
00:CFD4:A9 0F     LDA #$0F
00:CFD6:8D 15 40  STA $4015 = #$FF
00:CFD9:A9 C0     LDA #$C0
00:CFDB:8D 17 40  STA $4017 = #$FF
00:CFDE:A9 00     LDA #$00
00:CFE0:8D 00 C0  STA $C000 = #$4C
```

As you can see, the sound is turned on and the frame sequencer is written to, as bank 00 is loaded by writing to $C000. We are on the right track, so we keep surfing the code as we track down banks one at a time and plug them in as I mentioned above. That's about it for this rip here.

The next rip and that's for the game GLK Dance By FengLi, which was somewhat of a difficult rip since the 4 sound driver cores were located in different address ranges, and is also bankswitching. What I'll do is show you the code that I designed to handle the rip.

Initialize 1

```
00:B000:48        PHA
00:B001:A9 40     LDA #$40
00:B003:8D 17 40  STA $4017 = #$FF
00:B006:A9 1F     LDA #$1F
00:B008:8D 15 40  STA $4015 = #$FF
00:B00B:68        PLA
00:B00C:AA        TAX
00:B00D:BD 90 B0  LDA $B090,X @ $B090 = #$00
00:B010:85 00     STA $0000 = #$00          ; sound core variable
00:B012:8D 0E 01  STA $010E = #$00          ; Tune variable
00:B015:C9 09     CMP #$09
00:B017:B0 4D     BCS $B066                 ; Tune if greater than 09
00:B019:C9 06     CMP #$06
00:B01B:B0 29     BCS $B046
00:B01D:C9 03     CMP #$03
00:B01F:B0 03     BCS $B024
00:B021:4C 0C B9  JMP $B90C
00:B024:A9 05     LDA #$05                  ; tunes 4-6
00:B026:8D FC 5F  STA $5FFC = #$FF
00:B029:A9 06     LDA #$06
00:B02B:8D FD 5F  STA $5FFD = #$FF
00:B02E:A9 07     LDA #$07
00:B030:8D FE 5F  STA $5FFE = #$FF
00:B033:A9 08     LDA #$08
00:B035:8D FF 5F  STA $5FFF = #$FF
00:B038:AD 0E 01  LDA $010E = #$00
00:B03B:18        CLC
```

```
00:B03C:E9 02     SBC #$02
00:B03E:8D 0E 01  STA $010E = #$00
00:B041:4C 14 C0  JMP $C014
```

Initialize 2

```
00:B046:A9 09     LDA #$09                  ; tunes 7-9
00:B048:8D FC 5F  STA $5FFC = #$FF
00:B04B:A9 0A     LDA #$0A
00:B04D:8D FD 5F  STA $5FFD = #$FF
00:B050:A9 0B     LDA #$0B
00:B052:8D FE 5F  STA $5FFE = #$FF
00:B055:A9 0C     LDA #$0C
00:B057:8D FF 5F  STA $5FFF = #$FF
00:B05A:AD 0E 01  LDA $010E = #$00
00:B05D:18        CLC
00:B05E:E9 05     SBC #$05
00:B060:8D 0E 01  STA $010E = #$00
00:B063:4C CF C1  JMP $C1CF
```

Play Driver Select

```
00:B0D0:A5 00     LDA $0000 = #$00
00:B0D2:C9 09     CMP #$09
00:B0D4:B0 12     BCS $B0E8
00:B0D6:C9 06     CMP #$06
00:B0D8:B0 0B     BCS $B0E5
00:B0DA:C9 03     CMP #$03
00:B0DC:B0 04     BCS $B0E2
00:B0DE:4C 2D B9  JMP $B92D         ; sound core 1
00:B0E1:00        BRK
00:B0E2:4C 35 C0  JMP $C035         ; sound core 2
00:B0E5:4C F0 C1  JMP $C1F0         ; sound core 3
00:B0E8:4C 2D C0  JMP $C02D         ; sound core 4
```

As you can see, that's the strategy I used to rip GLK Dance by Feng Li. It was a fun game to hack. What I have explained here is loaded banks by the init code, however, there are games that need to have banks loaded in during the play code and is timing sensitive and beyond the scope of this level, currently. Once you learn basic bankswitching by init, then you can move on to the harder bankswitching rips.