**- This guide -**  
This is a tutorial made to provide some information on the SPC700 and S-DSP sound chip used by the SNES.  
  
  
**- For who is this guide intended? -**  
This guide is aimed at users with intermediate to advanced ASM knowledge, but little to no knowledge of how the SPC, DSP, or smw music data works. If you don't know what a pointer is or why they exist, or don't have a firm understanding of how the 65816 processor status flags work, you need more experience before digging through the SPC engine routines. You should get some experience fooling around with blocks, levelasm, sprites, etc, first. If you've already written hacks for the SPC, then that's about the extent of this guide, and you won't find much here you didn't already know.  
  
You will also need to love the sound of the SNES or this will all seem very pointless and small.  
  
  
**- For what will this guide be completely useless? -**  
Basic music porting. Nothing here will help with music porting, ripping SPCS, converting midis, or "getting addmusic to work". We're concerned with the actual data upload code to the SPC and the inner workings of the sound engine used by SMW.  
  
  
**1) What is the SPC700?**  
Instead of copying the definition out of some document, I'll give my personal description of it. The SPC700 is a lot like the SNES CPU. It's a processor. It does the same type of stuff the 65816 does. It moves things around in memory, copies data, performs mathematical operations, does comparisons, and branches. The SPC700 doesn't even create any sound output. It is simply put, a processor.  
  
The SPC700 differs from the SNES CPU in that it comes with 3 hardware timers, which it controls via its memory mapped registers at $00fa~$00ff. The SPC700 is also the only means by which anything can communicate with the S-DSP.  
  
  
**1b) What is the S-DSP?**  
The S-DSP is the sound generating work horse. Here's the definition of DSP:

Quote

A digital signal processor (DSP) is a specialized microprocessor with an optimized architecture for the fast operational needs of digital signal processing.

Compared to a processor like the SPC700, the S-DSP is completely different. It doesn't perform operations like a processor. It's not concerned with moving stuff around in memory, comparing values, or branching. It has to output sound in real time, so it works in tandem with the SPC700, which organizes and sends data to it. The DSP could be likened to a musical instrument, and the SPC to the musician who plays it.  
  
Where exactly the SPC700 ends and the S-DSP begins is a bit of a fuzzy subject, but it doesn't matter either.  
  
  
**1c) Why this setup?**  
The SPC700 is a pain in the butt. Why does it have to get in the way of giving the 65816 direct access to the sound output of the S-DSP?  
  
One possible reason would be speed. Music should ideally be timed. Imagine if your game code had to stop every few dozen cycles to process an interrupt to check a hardware timer, and possibly having to deal with processing a bunch of music data. The strain and subsequent game slowdown would be immense. This is the perfect job for a coprocessor like the SPC700.  
  
  
**1d) What is the SMW SPC engine?**  
When you first turn on your SNES, there isn't actually anything in the SPC, other than the boot ROM. The boot ROM is just a basic program in the SPC that you can communicate with and upload your own code through. It basically just sits there and waits for you to send it things, and places the stuff wherever you say. There is no default music handling code in the SPC. Each game developer had to write their own music engine (or reuse an old engine from another game).  
  
N-SPC, or whatever you want to call it, is the main program that many Nintendo games use to play music and sound data. This music handling program is sent to the SPC, generally when the game boots up, using the boot ROM upload routine. This program handles everything we're concerned with.  
  
The only way to get meaningful results out of music hacking is to make changes to the spc engine. The barriers you face in this task are logistical.   
  
1. The SPC700 doesn't use the same opcodes or instruction set you're used to with the 65816, although they are somewhat similar. You still need to learn a new programming language.  
2. Since the engine itself is a program that gets uploaded to the SPC, you will have to learn the art of wrestling with the SPC upload routine to get your custom code into audio ram  
3. There isn't much information available on the smw sound engine.  
4. SPC hacking is about data. You not only have to create the routines that handle the data, but you have to have compatible data to send it. In other words, you still have to actually make the music or sprites or levels that make use of whatever features you were to add to it.  
  
If you've used any of those MORE.bins floating around out there, those are essentially edits to the spc engine. They add to or change the capabilities of the code that gets uploaded to the SPC, allowing for the music data to be interpreted in new ways.  
  
  
**2) Basic SPC**  
  
That was all background information that won't help you do anything tangible. Let's explore what can be done by communicating with the SPC700, starting with the most basic.  
  
  
**2a) Storing to the audio ports**

Code

LDA #$03

STA $2142

You may have tried this at some point in blocks, sprites, or level asm. This code causes song $03 to play obviously, but let's talk about why it does that.   
  
$2142 is an SNES hardware register and one of the SPCs audio ports. The other audio ports are $2140, $2141, and $2143. These are the only means by which you can actually communicate with the SPC at all. Unfortunately, the cpu/you don't have direct access to the SPC audio ram. Everything you do goes through these four ports.   
  
There is nothing special about $2142 that makes it a "music selection" register or anything. It's just the way the smw SPC engine was designed by the original sound programmer. In other games, the ports at $2140 will be used, but they may be used in completely different ways. Storing $03 to $2142 in another game may cause it to go into upload mode, or make the music fade out, or play sound effects, or just crash because you did something wrong. The point I'm trying to drive home here is that these ports are just gateways used for communicating with the code you uploaded into the SPC. When the SMW sound engine in the SPC checks to see what is in its input port at $2142 and finds $03 there, it uses the number 3 to find the data for the song data we know as the underwater music, and basically, begins playing the song.  
  
If you were to use the excellent APU debugger in bsnes and watch the input ports, you would see all this happening piece by piece.  
  
  
**2b) Calling addmusic**  
  
There isn't a whole lot you can do by just throwing bytes at the SPC data ports. Let's try to actually do something useful and solve a problem.  
  
An often experienced dilemma is the one of having vanilla and "custom" music in the same level. If you've ever tried to play both in the same level, you would know that the sound just crashes. This is because there are no vanilla songs loaded into audio ram when you have custom music, and vice versa. Vanilla SMW music is extremely well made and compact. Most of the soundtrack can fit in ARAM at one time. Most custom music is big and bulky, and only a couple of songs can fit into ARAM at one time (many games only have 1 song in ARAM at a time). By telling the SPC to play a song that isn't there, it usually ends up taking data from some invalid location and crashing itself.   
  
Since the song you want to play doesn't exist, you're going to have to make it exist by sending it to ARAM. This is kind of a complicated process, but luckily, your custom music hack can handle it for you.  
  
Of course, your custom music hack (I used Romi's when writing this, but it's starting to show its age. AddmusicM is the future) has been uploading music whenever you entered a level or the overworld. Why can't you just tell it to upload music whenever you want? Well, you can. If you know exactly where the hack was placed in the rom, you can simply JSL to it with your song number in A (sometimes you also need to put your song number in $1DFB). Some of the addmusics expect certain conditions, such as level mode ($0100) to be $11, and brightness to be at 0. You'll have to satisfy these conditions, or addmusic will just return without loading anything.  
  
If you don't know or don't want to keep looking up the location of addmusic in your rom everytime it decides to move, you can do what I do, which is use its hijack point as a pointer to it.  
  
(Romi's) addmusic hijacks at $00:9740 in the smw rom. The code at that location looks something like this.

Code

JSL $xxyyzz

We can use that $xxyyzz to figure out exactly where the addmusic hack is located. Simply use it as a pointer and JML to it, but note that you'll need to push your own return address.  
  
Note that music uploads only occur normally during level loading and such, so NMI is always off. For doing this in the middle of a level, NMI will be ON and will freeze the game as soon as it fires. In other words, turn NMI off until you're done uploading.  
  
This code is functional. Put your song number in A.

Code

TAX ; hold song # in X

STA $0DDA ;\ this stuff is required by romi's addmusic

STA $1DFB ;|

LDA #$11 ;|

STA $0100 ;/

STZ $4200 ; disable NMI

LDA $13BF ;\ stuff required by romi's addmusic

PHA ;|

STZ $13BF ;|

LDA $0DAE ;|

PHA ;|

LDA $0DAF ;|

PHA ;/

PHK ; We push our return address onto the stack

PER RETURNADDR-1

TDC ;\

PHA ;| set bank 0

PLB ;/

TXA ; get song # back in A

JML [$9741] ; We use the addmusic hijack as

; a pointer to jump to the hack

RETURNADDR:

; it loads the song and returns here to the

; return address we pushed

PLA ; now just pull all this stuff back

STA $0DAF

PLA

STA $0DAE

LDA #$14

STA $0100

PLA

STA $13BF

LDA #$81

STA $4200 ; reenable NMI

After a short pause, your song will begin playing. If it was a vanilla song, the vanilla bank will be loaded, otherwise, the song group containing your song will have been loaded.  
  
  
**2c) Calling Sampletool**  
  
It's possible to do the same thing we just did with addmusic, with sample tool. Sample tool operates on level numbers, so you feed it the level number that has the sample bank you want, and it loads it. Sampletool expects the level number to be at RAM $0E, so put it there.  
  
Sampletool's hijack doesn't reside in bank 0, so it's a bit trickier to get to. If you can find the sample tool hack in your rom, great. It never moves around, so you can just JSL to it. If you want to use the pointer method, you'll use $05D8E7, which is where sampletool hijacks from.

Code

PHK

PLB

REP #$30

LDA #$001F ;\ Let's say we want to load the sample bank of Level 1F

STA $0E ;/ dump $001F at $0E

ASL ; sampletool expects Y to contain the level number \* 2

TAY ;

SEP #$30

STZ $4200 ; disable NMI

PHP

PHK ; push the return address

PER SAMPLERETURN-1

SEP #$10

LDA $05D8E7 ;\ read the bytes directly from the JSL in the ROM

STA $00 ;| and dump them at $0000

LDA $05D8E8 ;|

STA $01 ;|

LDA $05D8E9 ;|

STA $02 ;/

TYA ; get level number back in A

REP #$30

JML [$0000] ; use the bytes dumped at $0000 as a pointer

SAMPLERETURN:

; it loads the sample bank set for level 1F and returns here

PLP

SEP #$10

That's all for that. Note that you'll have to restart your music.  
  
  
**3) Intermediate SPC**  
  
**3a) SPC upload routine**  
  
If you want to upload your own data to the SPC, you'll need to learn the upload routine. It's a tedious, 1 byte at a time upload to the SPC, full of annoying handshakes. I've also got to warn that this won't be of any use for normal hacking. You could get by fine just by calling addmusic and sampletool and letting them handle the data uploads.   
  
If you've ever looked at the routine in all.log, it probably looked like the dog's dinner. It's sandwiched between the rest of the boot sequence and is in fact fully optimized spaghetti code. There are a couple of different entry points to the routine. One loads the pointers for the level music, one the pointers for the overworld, and one containing the ending music. At boot, it uses a different entry point which first uploads the NSPC engine itself. There's also "upload misc data", which is the most interesting one for us since we'll use it to upload whatever arbitrary stuff we want to.  
  
So let's look at the standard misc spc upload piece by piece.

Code

StrtSPCMscUpld:

LDA #$FF ; Get SPC ready for upload

STA $2141

JSR UploadDataToSPC

RTS

Sending $FF to $2141 instructs the NSPC engine to drop what it's doing and get ready to receive data.

Code

UploadDataToSPC:

SEI ; Disable interrupts

STZ $4200 ; Disable NMI \*NOT part of original code\*

JSR SPC700UploadLoop

LDA #$80

STA $4200 ; Reenable NMI \*NOT part of original code\*

CLI ; Reenable interrupts

RTS

The main CPU and the SPC basically communicate by poking data at each other very slowly and tediously, while counting and handshaking constantly. Since the interrupt routine of SMW fools around with the audio ports, interrupts going off during this process will crash your game 100% of the time, thus they are disabled by this wrapper code.  
  
Since the game only ever loaded banks during level load while NMI was off, NMI isn't natively shut off. We add that part ourselves so we can load stuff in the middle of a level when NMI is turned ON.

Code

SPC700UploadLoop:

PHP

REP #$30 ; 16-bit A & X/Y

LDY #$0000 ; set count to 0

LDA #$BBAA ; 0xBBAA is what $2140-41 will contain when

WaitForSPCEcho1: ; the SPC is ready

CMP $2140 ; Keep looping until it is ready

BNE WaitForSPCEcho1 ;

SEP #$20 ; 8-bit A

LDA #$CC ; 0xCC is what we send to the SPC to tell it we're ready

BRA GetLengthADDR

Pretty self explanatory. Whatever the SPC was doing before needs to get finished before we can start sending data. The SPC is slower than the 5A22 and usually has a whole lot of miscellaneous work to finish before it ever reads from $2140.

Code

GetLengthADDR:

PHA

REP #$20 ; 16-bit A

LDA [$00],y ; Get data block length.

INY

INY ; Y+2, point to next word in table

TAX ; LENGTH goes into X

LDA [$00],y ; Get target address.

INY ; Y+2, point to next word in table

INY

STA $2142 ; Send address to APU port 2&3

SEP #$20

CPX #$0001 ; Roll the carry flag from the operation CPX #$0001

LDA #$00 ; into bit 0 of A. in other words, if X is nonzero,

ROL ; load A with 0x01. otherwise A=0x00

STA $2141 ; Send it to APU port 1

ADC #$7F ; If A = 1, overflow flag will be set here

PLA ;

STA $2140 ; Store either 0xCC or our count to APU port 0 signaling that we are ready to transfer

WaitForSPCEcho0:

CMP $2140 ; Wait for the SPC to echo the value we've sent

BNE WaitForSPCEcho0

BVS GetDataBytes ; If the overflow flag was set earlier, branch, there

; is more data to send.

STZ $2140 ; Otherwise, finish up SPC transfer

STZ $2141

STZ $2142

STZ $2143

PLP

Return:

RTS

The first thing the upload routine does is read the first two bytes of the music data chunk. WHICH music data depends on the entry position we used to get here, or, whatever value we put in the scratch RAM $00 beforehand. The first two bytes of the music data need to be the length of the upload. Then it reads the next two bytes, which are the address/destination we are uploading to within the SPC. We send the destination to the SPC so it knows exactly where to put the data, and we keep the length in X so we know when we're hit the end of the data to send. Every upload needs to be formatted this way.  
  
There's a bit of tricky optimization going on with the overflow flag in there. All I can advise is read the comments and try to see how this saves the routine from using a free byte of RAM somewhere.  
  
Anyway, since there is data to send and the overflow flag is set, we go to GetDataBytes

Code

GetDataBytes:

LDA [00],y ; Put our first data byte in the high byte of the

INY ; accumulator, and 0x00 (count) in the low byte

XBA

LDA #$00

BRA StoreByte

Now we're reading the actual data we want to send to the SPC, as opposed to length/address data. This is where the main loop and bulk of the process starts.  
  
The routine will start fetching one byte of music data in A and XBAing it into the high byte of A. The low byte is set to $00 because it's going to be used as a counter from now on. The code will constantly switch these back and forth between counting in A and sending a byte of data from A to the SPC.

Code

StoreByte:

REP #$20 ; Accum (16 bit)

STA $2140

SEP #$20 ; Accum (8 bit)

DEX ; decrement LENGTH

BNE GetDataBytes2

Pretty self explanatory. Since it's a 16 bit store operation $2140 gets the byte of music data and $2141 gets the count byte. Decrement X (length) and get the next byte IF we haven't hit 0 length.

Code

GetDataBytes2:

XBA ; move count to high byte

LDA [$00],y ; get new data byte in low byte

INY

XBA ; move count back into low byte

WaitForSPCEcho2:

CMP $2140 ; wait for SPC to echo the count

BNE WaitForSPCEcho2

INC A ; increment the count

StoreByte:

` ...

This is a lot like GetDataBytes except we don't set the count to 00, we just... count.  
  
It loops back around to StoreByte until length = 0. Then comes actually the final bit of the routine.

Code

WaitForSPCEcho3:

CMP $2140 ; wait for SPC to echo count as usual

BNE WaitForSPCEcho3

Add3:

ADC #$03 ; prevent our index count from being 0

BEQ Add3

GetLengthADDR:

...

An obscure bug occurs if the count should becomes $00 at this point, which is why $03 is always added onto it. There's no need to worry about it.  
  
At this point, the code rolls back into GetLengthADDR. Let's take one last look at that part.

Code

GetLengthADDR:

PHA

REP #$20 ; 16-bit A

LDA MY\_DATA,y ; Get data block length.

INY

INY ; Y+2, point to next word in table

TAX ; LENGTH goes into X

LDA MY\_DATA,y ; Get target address.

INY ; Y+2, point to next word in table

INY

STA $2142 ; Send address to APU port 2&3

SEP #$20

CPX #$0001 ; Roll the carry flag from the operation CPX #$0001

LDA #$00 ; into bit 0 of A. in other words, if X is nonzero,

ROL ; load A with 0x01. otherwise A=0x00

STA $2141 ; Send it to APU port 1

ADC #$7F ; If A = 1, overflow flag will be set here

PLA ;

STA $2140 ; Store either 0xCC or our count to APU port 0 signaling that we are ready to transfer

WaitForSPCEcho0:

CMP $2140 ; Wait for the SPC to echo the value we've sent

BNE WaitForSPCEcho0

BVS GetDataBytes ; If the overflow flag was set earlier, branch, there

; is more data to send.

STZ $2140 ; Otherwise, finish up SPC transfer

STZ $2141

STZ $2142

STZ $2143

PLP

Return:

RTS

Since it's going to read 4 more bytes of data from the music data pointer, your music data should always terminate with 4 zero bytes. These will get sent to the SPC instructing it that there is no more data to send and the routine will exit.  
  
The audio ports are cleaned up with STZ and the processor flags are finally pulled back, and the routine returns to the wrappers.

Code

UploadDataToSPC:

SEI ; Disable interrupts

STZ $4200 ; Disable NMI \*NOT part of original code\*

JSR SPC700UploadLoop

LDA #$80

STA $4200 ; Reenable NMI \*NOT part of original code\*

CLI ; Reenable interrupts

RTS

Code

StrtSPCMscUpld:

LDA #$FF ; Get SPC ready for upload

STA $2141

JSR UploadDataToSPC

RTS

Interrupts are reenabled and control returns from the routine simply.  
  
  
  
**3b) The full routine**  
  
It's not as long as it first seemed.

Code

; before calling this routine,

; at $000000, put a 24-bit pointer to

; the data you want to upload

StrtSPCMscUpld:

LDA #$FF ; Get SPC ready for upload

STA $2141

JSR UploadDataToSPC

RTS

UploadDataToSPC:

SEI ; Disable interrupts

STZ $4200 ; Disable NMI \*NOT part of original code\*

JSR SPC700UploadLoop

LDA #$80

STA $4200 ; Reenable NMI \*NOT part of original code\*

CLI ; Reenable interrupts

RTS

SPC700UploadLoop:

PHP

REP #$30 ; 16-bit A & X/Y

LDY #$0000

LDA #$BBAA ; 0xBBAA is what $2140-41 will contain when

WaitForSPCEcho1: ; the SPC is ready

CMP $2140 ; Keep looping until it is ready

BNE WaitForSPCEcho1 ;

SEP #$20 ; 8-bit A

LDA #$CC ; 0xCC is what we send to the SPC to tell it we're ready

BRA GetLengthADDR

GetDataBytes:

LDA [$00],y ; Put our first data byte in the high byte of the

INY ; accumulator, and 0x00 in the low byte

XBA

LDA #$00

BRA StoreByte

GetDataBytes2:

XBA

LDA [$00],y

INY

XBA

WaitForSPCEcho2:

CMP $2140

BNE WaitForSPCEcho2

INC A

StoreByte:

REP #$20 ; Accum (16 bit)

STA $2140

SEP #$20 ; Accum (8 bit)

DEX ; decrement LENGTH

BNE GetDataBytes2

WaitForSPCEcho3:

CMP $2140

BNE WaitForSPCEcho3

Add3:

ADC #$03 ; prevent our index count from being 0

BEQ Add3

GetLengthADDR:

PHA

REP #$20 ; 16-bit A

LDA [$00],y ; Get data block length.

INY

INY ; Y+2, point to next word in table

TAX ; LENGTH goes into X

LDA [$00],y ; Get target address.

INY ; Y+2, point to next word in table

INY

STA $2142 ; Send address to APU port 2&3

SEP #$20

CPX #$0001 ; Roll the carry flag from the operation CPX #$0001

LDA #$00 ; into bit 0 of A. in other words, if X is nonzero,

ROL ; load A with 0x01. otherwise A=0x00

STA $2141 ; Send it to APU port 1

ADC #$7F ; If A = 1, overflow flag will be set here

PLA ;

STA $2140 ; Store either 0xCC to APU port 0, signaling that we are ready to transfer, or our nonzero index count

WaitForSPCEcho0:

CMP $2140 ; Wait for the SPC to echo the value we've sent

BNE WaitForSPCEcho0

BVS GetDataBytes ; If the overflow flag was set earlier, branch, there

; is more data to send.

STZ $2140 ; Otherwise, finish up SPC transfer

STZ $2141

STZ $2142

STZ $2143

PLP

Return:

RTS

This is the exact code sitting in every SMW ROM in the world. Unfortunately it returns with RTS and is in bank 0, so you can't call it whenever you want unless you get a bit fancy. What you could do is change the first block to contain an RTL

Code

StrtSPCMscUpld:

LDA #$FF ; Get SPC ready for upload

STA $2141

JSR UploadDataToSPC

RTL

and include a copy of the full routine somewhere in your rom/sprites/levelasm/blocks. Then you can use it wherever you want.  
  
  
**3c) NSPC Audio RAM**  
  
ARAM is pretty much the same as the SNES WRAM that you are used to by now. Things like the frame counter $13, scratch addresses $00~0F, game mode $100, in main RAM are fixed locations for values the game code needs to remember. ARAM is similar in that it has fixed locations containing whatever stuff the sound code needs to remember. Here's a really barebones and incomplete ARAM map I assembled while poking around.

Code

$0000~000D RAM - Seem to be multiple mirrors of the input ports f4 to f7. Sometimes scratch ram?

$000E~F RAM - Something involving vibrato

$0014~0015 RAM - during upload, points to destination of current upload

$001D RAM - Bitflags. Seem to "pause" channels when set. 76543210

$0020~002F RAM - Probably something to do with noise generation? (dsp register 3d)

$0030~003F RAM - 2bytes per chan. points to next (current?) music data byte for each channel

$0040~0041 RAM - Points to music data. Roughly corresponds to "current measure"

$0042 RAM - Dec by 1 each time song loops. If FF, never decreases. Music stops at 0.

$0043 RAM - Global transposition value. Negative values work.

$0044 RAM - main timer, increases after every tick of spc register $fd

$0045 RAM - high byte of main timer $0044

$0046 RAM - used as index of (current channel being processed) x 2?

$0047 RAM - ?? holds 1 when there is a key on event

$0048 RAM - seems to be a bitwise indicator of current channel being processed?

$0049 RAM - Unknown. Is added to the tempo to determine timing somehow.

$004A~4F ??? - Never seems to be used.

$0050 RAM - forced to 0 (high byte of tempo, ignored)

$0051 RAM - "tempo"

$0052 RAM - Decrementing timer. When it reaches 0, set tempo to value in $53.

$0053 RAM - Tempo to change to when $52 reaches 0.

$0054 RAM - High byte of $53. Does nothing. See $50.

$0055 RAM - Some weird tempo slide value used with tempo change $52.

$0056 RAM - Related to fade volume?

$0057 RAM - Global volume.

$0058 RAM - Decrementing timer for volume fade.

$0059 RAM - Volume to fade to when $58 reaches 0.

$005A~5B RAM - Related to fading. Does something strange like slide the volume in and out.

$005C RAM - Related to $48?

$005D~5F ??? - Never seems to be used.

$0060 RAM - loaded with d4 (Set patch command)

$0061 RAM - forced to 0

$0062 RAM - engine sends this to $2c rw EVOLL - Left channel echo volume

$0063 RAM - forced to 0

$0064 RAM - engine sends this to $3c rw EVOLR - Right channel echo volume

$0065 RAM - Related to echo fade somehow

$00F4 RAM - Input port from SNES. $2140 / $1DF9

$00F5 RAM - Input port from SNES. $2141 / $1DFA

$00F6 RAM - Input port from SNES. $2142 / $1DFB

$00F7 RAM - Input port from SNES. $2143 / $1DFC

$01CF RAM - stack

$0240~024F RAM - low bytes = channel 0-7 volumes. high bytes = ??

$0280~028F RAM - low bytes = channel 0-7 panning. high bytes = ??

$02B0~02BF RAM - low bytes = channel 0-7 current note value

high bytes = pitch modifier/tuning value. FF = (almost) 1 semitone higher

$02D0~02DF RAM - low bytes = channel 0-7 pitch modifier/tuning value. high bytes = ??

$03A0~ RAM - possibly free

$400~4FF RAM - free ram. used by carol's more.bin

$0500~ CODE - start of spc engine program

$0697~ FUNC - "send to dsp" function. call with A containing value and Y with reg to send to

$0EDC~ FUNC - table of functions?

$1295~ DATA - table of initial dsp settings (tied to $12A1, contains the values sent to dsp reg adresses)

$12A1~ DATA - table of initial dsp settings (contains the dsp register addresses themselves)

~???~ ; lot of sound data resides around here somewhere

$5619~ POINT - pointers to sound effect data (port $f7)

$5681~ POINT - pointers to sound effect data (port $f4)

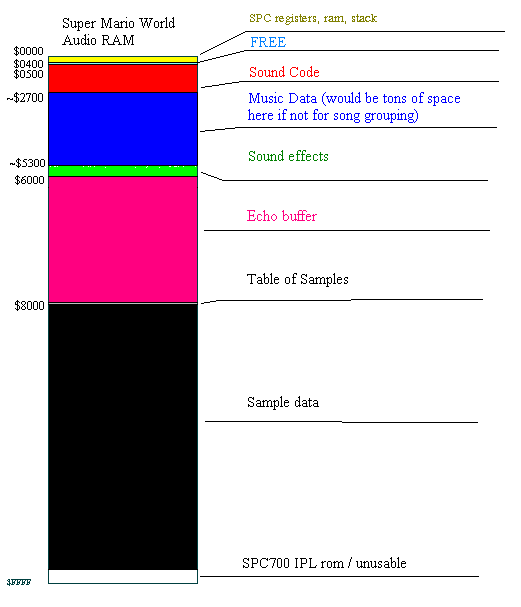
$58D6~ SFX - overworld ping sfx data (OW bank)

$5CEF~ SFX - head bump sfx data (Level bank, $f4, #$01)

$5EB6~ SFX - coin get sfx data (Level bank, $f7, #$01)

$6000~$7FFF RAM - Echo ring buffer address (see DSP register 6D)

$8000~ RAM - table of samples (see DSP register 5D) followed by the samples themselves

There's nothing quite as illuminating as being able to manually change the values in an array of RAM addresses at a whim. Want to know what something does? Change the value. Did the playback speed just change? I guess this address is where the code holds the playback speed. Why does this address decrement to 0 causing a new note to play on the first channel? These are probably individual channel timers. Once you have a theory, you can then check it by stepping through in the debugger to see if it is indeed correct.  
  
The only sane tool to use for this is bsnes debugger. It does support manual memory editing also. I suggest checking it out if you want to explore ARAM more.  
  
  
**3d) Rough visual representation of audio ram**  
  
  
  
Note that the bottom half is a bit lopsided, and the sound code actually ends much earlier, around $1300.  
  
  
**4)Advanced SPC**  
  
**4a) Programming custom commands**  
  
Wouldn't it be nice if there were a fade-in command, or a mute command that muted everything/unmuted everything that you could trigger in your levelasm/sprites/blocks? What if you could trigger tempo or volume changes in parts of the song through sprites, blocks, or levelasm? The only limits to what can be done are the limits delineated in this document: http://www.romhacking.net/docs/%5B191%5Dapudsp.txt  
  
Unless you're interested in the details of the S-DSP hardware, you should skip down to the section entitled S-DSP REGISTERS. These registers reflect the full extent of sound manipulation possible using the SPC700. Everything you've ever heard come out of an SNES came from some combination of these registers and the data fed into them. Really, they aren't even very complicated. There's a few copy patsed commands for every voice channel, some echo settings, some pitch slides, some panning, some noise/control registers, etc. The main problem are the layers between you and these DSP registers.  
  
You => your SMW rom => the SPC => the DSP registers  
  
If you've made it this far, the only level left is => the DSP registers.   
  
Starting with "You". "You" is the idea is to create something that doesn't exist in the sound code, the mute function. Taking a look at the DSP Registers doc, there are a number of ways you could go about it. You could set the left and right master volumes to 0. That's a good solution, but you would still hear the echo. You could force key off for all voices. This is a bad solution. You could use the thing called "Reset, Mute, Echo-Write flags and Noise Clock" to force a mute. This is the best solution.  
  
This register is register 6c, the "Reset, Mute, Echo-Write flags and Noise Clock" flags. As you can see, the mute control is labeled under m, bit 6 (rmennnnn). To turn this on, you simply activate bit 6. In hex, $40 is the value that contains bit 6 ON. Thus, we simply need to send the value $40 to DSP register $6c. If only we could LDA #$40 STA $xx6c, there'd be no need for all of this nonsense.  
  
The SPC sends values to the DSP through two ports in audio ram, $f2 and $f3. $f2 specifies which DSP register you want to modify and $f3 is the value you want to modify it with.  
  
So, we need to instruct the SPC to feed $40 to DSP $6c whenever we want the mute to occur. From your SMW rom => the SPC, there is no sound effect/command that does this, and there is no code in the SPC for it either, so we have to add a hack onto the spc engine that does it. The first thing we need is some free ram to place the hack. Luckily there are about 255 bytes at $0400 in ARAM. This is enough for a small hack, but space is really limited in the SPC ARAM, and it's not like a ROM where you can just expand it to some absurd size either.  
  
Now the fun part: SPC coding.   
  
  
**4b) SPC700 assembler**  
  
The SPC uses a somewhat similar but different ASM language than the 65816 family of processors. If you want to learn more about it, I suggest you step though some SPC code in the bsnes debugger.  
  
Here are some of the bigger differences:  
  
No LDA STA commands. You use "mov" to load values and store them.

Code

mov a, #$40 ; move the value #$40 into a

mov $50, y ; move the value from y into RAM at $50

mov x, a ; move the value from a into x

Be careful of the order. When you read the commands try to think of them this way: "Move, into x, a". "Move, into $50, y"  
  
  
  
Instead of push and pull PHA/PLA, you use push and pop.

Code

push a ; push a

push y ; push y

pop y ; pull y

pop a ; pull a

The SPC instruction set is a lot like the x86 processors' instruction sets.  
  
  
Since the SPC only has 1 bank of addressable memory from $0000 to $FFFF, there is no JSL or JSR, only "call".

Code

call $0697 ; call/JSR to subroutine at $0697

Subrtouines return with "ret" instead of JSR/JSL.

Code

ret ; return from subroutine

Those are the major differences. You can write some pretty substantial code with only that knowledge and your basic 65816 ASM knowledge.  
  
  
Since xkas won't compile SPC700 code, for compiling the code, you use spcas which you can download off of romhacking.net. Let's talk about assemblers and spcas for a second.  
  
  
**- spcas -**  
  
Assemblers essentially exist so programmers don't have to tediously write out machine code. We could write our hack out manually using a lookup table of SPC700 opcodes and translating them very carefully into a hex editor and saving it as a plain old binary file. For small hacks or hex edits this might even be reasonable. I started out this way before I found out there was already a great spc assembler.  
  
If you remember the previous section, you should remember the format of things sent to the SPC. The first two bytes of any upload needs to contain the length of the data we want to upload, followed by 2 bytes containing the destination to upload it to. spcas can make this easier as well. We don't have to count the bytes in our data, just instruct spcas to calculate it for you and place the length as the first two bytes in the file.

Code

dw END1-BLOCK1 ; have the compiler calculate the length

db $00,$04 ; destination = $0400 (remember the order of bytes is reversed

BLOCK1:

\*our code here\*

END1:

spcas will translate "dw END1-BLOCK1" to however long the stuff we put between those two labels is. So the first four bytes in our file will look something like 2D 00 00 04. Exactly as they should be for our length and destination headers.  
  
  
  
Anyway, here's some example code for the mute command. For the example's sake, let's say we want to mute the sound when we send $77 to audio port 4, and unmute when we send $78.

Code

dw END1-BLOCK1 ; have the compiler calculate the length

db $00,$04 ; destination = $0400 (remember the order of bytes is reversed

BLOCK1:

push a ; save a and y

push y ;

mov a,$f7 ; $f7 is input audio port 4, also known as $1DFC/$2143

cmp a,#$77 ; if it contains $77, go to muteON

beq MuteON

cmp a,#$78 ; if it contains $78, go to muteOFF

beq MuteOFF

bra Return ; otherwise do nothing

MuteON: ; -muteON

mov a,#$40 ; put our value in a

mov y,#$6c ; put our destination in y

Set:

call $0697 ; call subroutine $0697 which sends a and y to the DSP

; we could just move them to $f2 and $f3 manually with mov, but

; this takes up less space

bra Return

MuteOFF: ; -muteOFF

mov a,#$00 ; put $00 in a (this will disable mute)

mov y,#$6c ; put our destination register in y

call $0697

Return:

pop y ; get y and a back

pop a

...

END1:

That's our mute code. It will check input port $2143, in ARAM known as $f7, in SMW mirrored as $1DFC, and wait for it to either contain $77 or $78, then it will send various values to the DSP mute register.  
  
There's one more bit to assemble after that. Now that we have the code to send to the SPC, we need to add a hijack that jumps to it or the SPC will never run it. You have to have some knowledge of how the N-SPC code normally functions to know where to hijack from, but suffice it to say, $054f is a pretty good spot for a main hijack. $0500 is where the main N-SPC code begins, and $054f is just a bit after the main "wait for timer to tick" loop. So, the end of our code file should look like this

Code

...

Return:

pop y ; get y and a back

pop a

mov a,#$38 ; (restoring hijacked code)

jmp $0552 ; jump back to hijacked code

; -END OF FIRST BLOCK OF DATA-

dw END2-BLOCK2 ; new length and dest bytes for new block of data

db $4f,$05 ; dest = $054f

BLOCK2:

jmp $400 ; this is our hijack from the main code. jump to our hack.

END2:

db $00,$00,$00,$00,$00,$00 ; trailing zeroes to indicate end of upload

So, what spcas does is assemble that code into a .bin binary file. We dump that bin file somewhere into our rom and then we upload it with the upload routine we discussed in the previous section. At that point, we can send $77 or $78 to $1DFC at any time through levelasm/sprites/blocks to control the mute setting. It's basically a custom sound effect. Make sure you don't use values that are used for real sound effects (77 and 78 aren't)  
  
  
**4c) Solving mysteries & Using the debugger**  
  
You've seen how you can mod the sound engine, but how can you find out about this stuff yourself and make your own vision a reality? The most important thing is your SPC debugger: bsnes.  
  
Let's take this example. Someone wants to know more about the sfx in $1DFA/$2141/$f5. We need to find out how the sound code handles input from this port. When mario jumps, $01 is received at $f5 and the jumping sound plays. By what process exactly?  
  
There's two ways I can think of doing this. The easy way would be with breakpoints. Breakpoints allow you to pinpoint the code you're interested in. Basically, when the SPC reads $01 from $f5, the program will stop and let us look around. Go to bsnes debugger, under debugger > tools > breakpoint editor  
  
The fields in the window are thus

Code

(on/off) (address) \*optional\*(value sought) (condition) (bus select)

(on/off) - Check to activate this breakpoint. When on, this breakpoint will be active and will stop the program when the conditions are met.  
  
(address) - The address we're waiting for a read/write/execute on. Leaving any preceding digits blank will default to \*any\* address, so entering f5 would breakpoint on reads from $00f5, $01f5, $02f5, etc  
  
(value sought) - Will only break when the value specified here is read/written. Leave blank for any read/write.  
  
(condition) - When set to read, will only break when a value is read from (address). When set to write, only when written. When set to execute, will only break when the main CPU or SPC executes the instruction at the address.  
  
(bus select) - Which work ram space/cpu will be doing the reading/writing/executing. Set to S-SMP for audio ram.  
  
So, let's set a breakpoint at $f5, and set the value sought to $01 which is the mario jump. Set the condition to read and the bus select to S-SMP. Finally, check the box for "on". Nothing will happen until mario jumps. At that point the SPC code will be stopped at the code that read $f5. Check "Step S-SMP" and hit the step button to step through the code.

Code

..05ac mov a,$00f4+x A:00 X:01 Y:01 SP:01cc YA:0100 nvpbHiZc

..05af cmp a,$00f4+x A:01 X:01 Y:01 SP:01cc YA:0101 nvpbHizc

..05b2 bne $05ac A:01 X:01 Y:01 SP:01cc YA:0101 nvpbHiZC

..05b4 mov y,a A:01 X:01 Y:01 SP:01cc YA:0101 nvpbHiZC

..05b5 mov a,$008+x A:01 X:01 Y:01 SP:01cc YA:0101 nvpbHizC

..05b7 mov $008+x,y A:00 X:01 Y:01 SP:01cc YA:0100 nvpbHiZC

..05b9 cbne $008+x,$05c1 A:00 X:01 Y:01 SP:01cc YA:0100 nvpbHiZC

..05c1 mov $000+x,y A:00 X:01 Y:01 SP:01cc YA:0100 nvpbHiZC

..05c3 mov a,y A:00 X:01 Y:01 SP:01cc YA:0100 nvpbHiZC

..05c4 ret A:01 X:01 Y:01 SP:01cc YA:0101 nvpbHizC

..056b call $0816 A:01 X:01 Y:01 SP:01ce YA:0101 nvpbHizC

..0816 cmp $007,#$24 A:01 X:01 Y:01 SP:01cc YA:0101 nvpbHizC

..0819 beq $082e A:01 X:01 Y:01 SP:01cc YA:0101 NvpbHizc

..081b cmp $003,#$24 A:01 X:01 Y:01 SP:01cc YA:0101 NvpbHizc

..081e beq $082a A:01 X:01

...

The first thing you need to say to yourself is "What the hell am I looking at right now". Let's analyze the code.

Code

..05ac mov a,$00f4+x A:00 X:01 Y:01 SP:01cc YA:0100 nvpbHiZc

..05af cmp a,$00f4+x A:01 X:01 Y:01 SP:01cc YA:0101 nvpbHizc

..05b2 bne $05ac A:01 X:01 Y:01 SP:01cc YA:0101 nvpbHiZC

..05b4 mov y,a A:01 X:01 Y:01 SP:01cc YA:0101 nvpbHiZC

..05b5 mov a,$008+x A:01 X:01 Y:01 SP:01cc YA:0101 nvpbHizC

..05b7 mov $008+x,y A:00 X:01 Y:01 SP:01cc YA:0100 nvpbHiZC

..05b9 cbne $008+x,$05c1 A:00 X:01 Y:01 SP:01cc YA:0100 nvpbHiZC

..05c1 mov $000+x,y A:00 X:01 Y:01 SP:01cc YA:0100 nvpbHiZC

..05c3 mov a,y A:00 X:01 Y:01 SP:01cc YA:0100 nvpbHiZC

..05c4 ret A:01 X:01 Y:01 SP:01cc YA:0101 nvpbHizC

As you know, $f4 to $f7 are the input ports. It's taking the values from the input ports and comparing them. Since this uses +x addressing to index the input ports, this code is probably reused for every port, and the value in x specifies which port it reads from. In other words, it's probably a general routine that operates on the input ports. As you can see, it has our sfx number, #$01. It's also taking it and putting it at $008+x ($009 here). Since this also uses the +x addressing mode, you can safely assume that the ram around $008 are mirrors for the input ports. It actually does the exact same thing to $000+x as you can see. There are a number of copies of input bytes saved around there in aram, probably used when determining whether or not a sound is already playing or some other type of logic.

Code

..056b call $0816 A:01 X:01 Y:01 SP:01ce YA:0101 nvpbHizC

..0816 cmp $007,#$24 A:01 X:01 Y:01 SP:01cc YA:0101 nvpbHizC

..0819 beq $082e A:01 X:01 Y:01 SP:01cc YA:0101 NvpbHizc

..081b cmp $003,#$24 A:01 X:01 Y:01 SP:01cc YA:0101 NvpbHizc

""What am I looking at?" Well, it returned, and now it seems to be doing other things. The trail has gone cold. We need to keep following our sfx byte. Copies were sent, so let's investiagate those ram addresses. I set a breakpoint on $001 for the sound effect, which triggers instantly.

Code

..09e5 mov a,$001 A:00 X:00 Y:00 SP:01cc YA:0000 nvpbHiZC

..09e7 cmp a,#$ff A:01 X:00 Y:00 SP:01cc YA:0001 nvpbHizC

..09e9 beq $099c A:01 X:00 Y:00 SP:01cc YA:0001 nvpbHizc

..09eb cmp a,#$02 A:01 X:00 Y:00 SP:01cc YA:0001 nvpbHizc

..09ed beq $097d A:01 X:00 Y:00 SP:01cc YA:0001 NvpbHizc

..09ef cmp a,#$03 A:01 X:00 Y:00 SP:01cc YA:0001 NvpbHizc

..09f1 beq $0995 A:01 X:00 Y:00 SP:01cc YA:0001 NvpbHizc

..09f3 cmp a,#$01 A:01 X:00 Y:00 SP:01cc YA:0001 NvpbHizc

..09f5 beq $0a14 A:01 X:00 Y:00 SP:01cc YA:0001 nvpbHiZC

"What am I looking at?" Sfx byte #$01 gets read from $001 and compared to a bunch of values. It matches one and it branches. This basically means that input to this port just calls hardcoded routines.

Code

..0a14 mov $005,a A:01 X:00 Y:00 SP:01cc YA:0001 nvpbHiZC

..0a16 mov a,#$04 A:01 X:00 Y:00 SP:01cc YA:0001 nvpbHiZC

..0a18 mov $0383,a A:04 X:00 Y:00 SP:01cc YA:0004 nvpbHizC

..0a1b mov a,#$80 A:04 X:00 Y:00 SP:01cc YA:0004 nvpbHizC

..0a1d mov y,#$5c A:80 X:00 Y:00 SP:01cc YA:0080 NvpbHizC

..0a1f call $0697 A:80 X:00 Y:5c SP:01cc YA:5c80 nvpbHizC

..0697 mov $00f2,y A:80 X:00 Y:5c SP:01ca YA:5c80 nvpbHizC

..069a mov $00f3,a A:80 X:00 Y:5c SP:01ca YA:5c80 nvpbHizC

..069d ret A:80 X:00 Y:5c SP:01ca YA:5c80 nvpbHizC

A few things happen here. A copy of #$01 is sent to $005. Notice how it's not $004+x this time. This routine only operates on $005 which really drives home the point that this is a hardcoded routine for input port $f5 / $1DFA only.  
  
#$04 gets sent to $383. Dunno what all that's about. Maybe you'd like to check it?  
  
Next, a value gets sent to dsp register #$5c. This is an extremely common occurrence if you get into debugging the sound effects. 5c is key off. It's setting key off for this channel, presumably because it's about to generate the sound of mario jumping. Regardless, the code again goes off to do other random things we aren't interested in yet.  
  
  
We've found how bytes from $1DFA / $f5 are decoded.

Code

..09e5 mov a,$001 A:00 X:00 Y:00 SP:01cc YA:0000 nvpbHiZC

..09e7 cmp a,#$ff A:01 X:00 Y:00 SP:01cc YA:0001 nvpbHizC

..09e9 beq $099c A:01 X:00 Y:00 SP:01cc YA:0001 nvpbHizc

..09eb cmp a,#$02 A:01 X:00 Y:00 SP:01cc YA:0001 nvpbHizc

..09ed beq $097d A:01 X:00 Y:00 SP:01cc YA:0001 NvpbHizc

..09ef cmp a,#$03 A:01 X:00 Y:00 SP:01cc YA:0001 NvpbHizc

..09f1 beq $0995 A:01 X:00 Y:00 SP:01cc YA:0001 NvpbHizc

..09f3 cmp a,#$01 A:01 X:00 Y:00 SP:01cc YA:0001 NvpbHizc

..09f5 beq $0a14 A:01 X:00 Y:00 SP:01cc YA:0001 nvpbHiZC

The input byte literally just gets run through a list of compares and then branches to hardcoded sound routines. This is sort of unfortunate since it makes it hard to expand upon or change the sound effects from this port.