nesdoug

24. MMC3, Bank-switching, IRQs

(Thanks to thefox for pointing out an error in my example code, see note at the very bottom of the page)

So far, all I've been using is small NROM sized .nes files. I'm going to show how to set up a much larger .nes file using the MMC3 mapper. I don't know every mapper, but I know the MMC3, so I'll use that for my example. NROM was the first cartridge design, but they soon used up the 32k bytes of PRG ROM (code), and especially the 8k of CHR ROM (graphics). A mapper is a way to trick the NES into accessing more ROM, by swapping (mapping) banks in and out of the CPU memory.

I'm going to pretend that I'm designing for a real actual NES cartridge, and use the actual size of an actual MMC3 board. According to this website…

http://kevtris.org/mappers/mmc3/ (http://kevtris.org/mappers/mmc3/)

The choices we have, are…

Max. 64K PRG, 64K CHR
Max. 512K PRG, 64K CHR
Max. 512K PRG, VRAM
Max. 512K PRG, 256K CHR
Max. 128K PRG, 64K CHR, 8K CHR RAM

(and some others that I omitted). I'm going to use the smallest example, 64k and 64k. First, we have to set this up correctly in the header. The header is a few bytes of metadata that is used by emulators, so it knows which mapper we're using, and how many banks, etc. Here's how it should look for our 64/64 MMC3…

```
.byte $4e,$45,$53,$1a
.byte $04   ; = 4 x $4000 bytes of PRG ROM
.byte $08  ; = 8 x $2000 bytes of CHR ROM
.byte $40  ; = mapper # 4 = MMC3
```

Now, we need to set up the .cfg file for multiple banks…

```
#ROM Addresses:
#they are all at $8000, because I will be swapping them into that bank
PRG0: start = $8000, size = $2000, file = %O ,fill = yes, define = yes;
PRG1: start = $8000, size = $2000, file = %O ,fill = yes, define = yes;
PRG2: start = $8000, size = $2000, file = %O ,fill = yes, define = yes;
PRG3: start = $8000, size = $2000, file = %O ,fill = yes, define = yes;
PRG4: start = $8000, size = $2000, file = %O ,fill = yes, define = yes;
PRG5: start = $a000, size = $2000, file = %O ,fill = yes, define = yes;
PRG6: start = $c000, size = $2000, file = %O ,fill = yes, define = yes;
PRG7: start = $e000, size = $1ffa, file = %O ,fill = yes, define = yes;
```

```
# Hardware Vectors at end of the ROM
VECTORS: start = $fffa, size = $6, file = %O, fill = yes;
```

Actually, what the $8000 does here, is every label inside that bank will be given an address indexing from the start of that bank + $8000. I actually only have code in the last bank, so I could put the start address anywhere, and it would work the same, but you will probably have code in some of these banks, and so you need to give the code the correct addresses for where it will actually be going when it's swapped in.

And, defined these segments in the .cfg file…

```
SEGMENTS {
HEADER:   load = HEADER,         type = ro;
CODE0:    load = PRG0,           type = ro,  define = yes;
CODE1:    load = PRG1,           type = ro,  define = yes;
CODE2:    load = PRG2,           type = ro,  define = yes;
CODE3:    load = PRG3,           type = ro,  define = yes;
CODE4:    load = PRG4,           type = ro,  define = yes;
CODE5:    load = PRG5,           type = ro,  define = yes;
CODE6:    load = PRG6,           type = ro,  define = yes;
STARTUP:  load = PRG7,           type = ro,  define = yes;
CODE:     load = PRG7,           type = ro,  define = yes;
VECTORS:  load = VECTORS,        type = ro;
CHARS:    load = CHR,            type = rw;

BSS:      load = RAM,            type = bss, define = yes;
HEAP:     load = RAM,            type = bss, optional = yes;
ZEROPAGE: load = ZP,             type = zp;
#OAM:   load = OAM1,    type = bss, define = yes;
}
```

(the OAM segment is not used in this example).

Ok, now I'm going to write something in each bank, so we can see how it loads into the ROM file. I'm writting the words "Bank0", "Bank1", etc. in every bank. And, I'm going to load those words onto the screen, so we can see the switch visually. (I set it to be triggered by pressing 'Start').

I had to write a bunch of PRAGMAs so that each bank will be compiled into the correct bank. Like this…

```
#pragma rodata-name ("CODE0")
#pragma code-name ("CODE0")
const unsigned char TEXT1[]={
"Bank0"};

#pragma rodata-name ("CODE1")
#pragma code-name ("CODE1")
const unsigned char TEXT2[]={
"Bank1"};
```

```
#pragma rodata-name ("CODE2")
#pragma code-name ("CODE2")
const unsigned char TEXT3[]={
"Bank2"};
```

etc. And when START is pressed, it will switch banks into CPU addresses $8000-9fff, and then load the first 5 bytes of that bank and write it to the screen, with this code…

```
void Draw_Bank_Num(void){ //this draws some text to the screen
PPU_ADDRESS = 0x20;
PPU_ADDRESS = 0xa6;
for (index = 0;index < 5;++index){
PPU_DATA = TEXT1[index];
}
PPU_ADDRESS = 0;
PPU_ADDRESS = 0;
}
```

When this compiles, it will write the address of TEXT1 into the code. It's the only thing in the first bank (bank #0), and in the .cfg file, I defined that bank to start at $8000. So, it will be fetching the first 5 bytes from $8000-8004. That is the bank that I keep switching, so every time it goes here, it will be pulling those 5 bytes from whatever bank is mapped to $8000. Here's the code that switches which bank will be mapped to addresses $8000-9fff…

```
if (((joypad1old & START) == 0)&&((joypad1 & START) != 0)){
++PRGbank;
if (PRGbank > 7) PRGbank = 0;
*((unsigned char*)0x8000) = 6; //bankswitch a PRG bank into $8000
*((unsigned char*)0x8001) = PRGbank;
Draw_Bank_Num(); //re-writes the text on the screen
```

I know, it looks like we're storing 6 at address $8000. But, you can't do that, because that's a ROM address. What this does is send a signal to the MMC mapper that we want to switch a PRG bank into $8000. The next line defines which bank will be swapped in. You can get some more detailed info on the wiki…

http://wiki.nesdev.com/w/index.php/MMC3 (http://wiki.nesdev.com/w/index.php/MMC3)

I feel like this may be confusing. It's an unfortunate coincidence, that the bank we're swapping and the MMC3 register are both called $8000. If you wanted to instead swap a bank into the CPU address $a000-bfff, you would do this…

```
*((unsigned char*)0x8000) = 7; //bankswitch a PRG bank into $a000
*((unsigned char*)0x8001) = which_PRG_bank;
```

Is that clearer? Bank swapping is done with a $8000 / $8001 write combination.

I also added a few lines at the start of the main() function, which sets the initial state how everything should be mapped at the start. I don't know for sure, but I think that the only bank that is certain at RESET is…that last PRG bank will definitely be at $e000-ffff. All our startup code should (ie. MUST) be located in that bank.

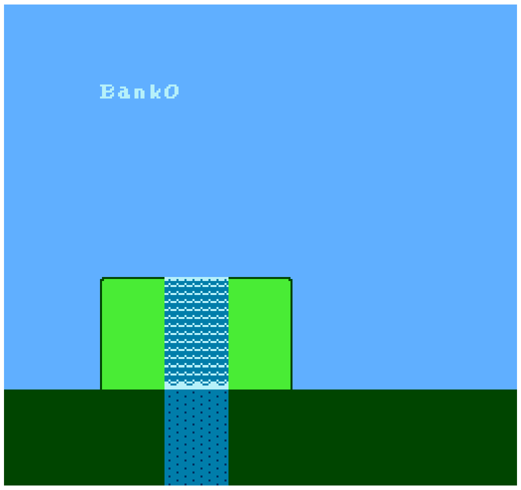
Now, that's cool and all, but you will not be using bank swapping like this (having every bank have things located at fixed positions, like $8000). In reality, you will probably have an array of addresses at the start of every bank, which point to the location of data within the bank (that way, the data can be anywhere within the bank). If you have code in that bank, you will perhaps put a 'JUMP TABLE' at the start of the bank…that's an array of addresses of the start of every function inside the bank. Essentially, the code in a fixed bank will read the address, and then jump (indirect) to that address. Or maybe, use the 'push address to the stack and RTS' Trick…

http://wiki.nesdev.com/w/index.php/RTS_Trick (http://wiki.nesdev.com/w/index.php/RTS_Trick)

That's kind of complicated ASM stuff, but it might be worthwhile learning it, if you're going to make the most of using multiple banks.

Anyway, I wanted to add a cool 'moving background' effect. This effect can be done several ways, but I think bank swapping CHR ROM is the easiest. This code waits 4 frames, and then switches which CHR banks will be mapped to $0000-$03ff of the PPU. When the PPU goes to draw tile #23 to the screen, the mapper will direct it to tile #23 of a specific CHR bank in the ROM.

MMC3 actually breaks the CHR ROM into $400 byte chunks (64 tiles), so bank 0 = the first $400 bytes, 1 = the next $400 bytes, etc. It takes multiple MMC map writes to fill the PPU full of new tiles. I'm just changing that first $400 bytes (PPU addresses 0-$3ff). I made a little waterfall. There are 4 nearly identical CHR ROM banks, with the water tiles shifted downward 1 pixel each bank down.



Here's the link to the source code. Press 'START' to see the PRG ROM bank switch.

http://dl.dropboxusercontent.com/s/gl73mp2nr1rpdzl/lesson19.zip (http://dl.dropboxusercontent.com/s/gl73mp2nr1rpdzl/lesson19.zip)

Now, that's not all MMC3 can do…it can also count scanlines. Normally, you would need to set up a Sprite Zero hit to time something to happen midframe, and you can only do that once. With MMC3, you can time multiple things to happen midframe, like changing the Scroll, or swapping CHR ROM, or other cool tricks. I'm going to change the scroll, about every 20 lines. You don't have to sit and wait for those 20 lines, the MMC3 mapper will count for you, and you can go on to do other things (game logic). It will generate an IRQ, and jump to the IRQ code. I have the IRQ code changing the Horizontal Scroll (several times a frame).

Things we need to do…The first line of every NES game (startup code) is to turn off interrupts. But, that's what an IRQ is, so in our main() function, I turned interrupts back on.

asm ("cli"); //turns ON IRQ interrupts

Also, I have to make sure that the vectors at the end of the reset.s code is pointing to the address of the IRQ code.

Now, during V-blank (you can turn this on any time, I just wanted to start it at the top of the screen) I set the MMC3 to start counting scanlines with this code…

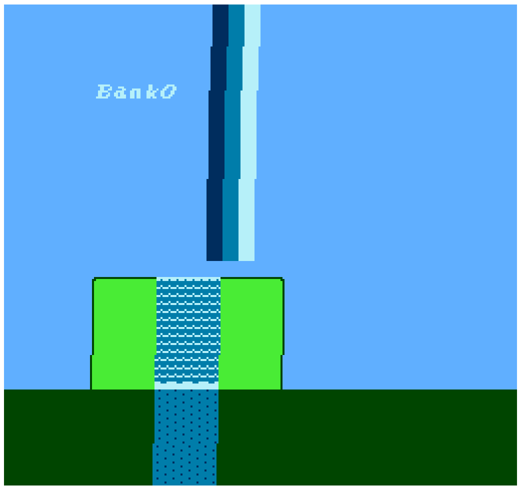
((unsigned char)0xe000) = 1; //turn off MMC3 IRQ
((unsigned char)0xc000) = 20; //count 20 scanlines, then IRQ
((unsigned char)0xc001) = 20;
((unsigned char)0xe001) = 1; //turn on MMC3 IRQ

Note: I think it skips the first scanline, so it actually doesn't generate an IRQ until at the end of scanline 21. Now, I want to change the horizontal scroll, which isn't a problem, but if I immediately tried to change the scroll, there would be a slight glitch (misalignment) of the screen at that point. I'm always amazed at how many professional games have these kind of glitches. Anyway, to avoid the glitch, you have to change the scroll during the very very very short H-blank period. What's an H-blank? When the PPU is drawing the picture to your TV, it goes left to right, then it jumps from right to left (not drawing) very quickly. That's the H-blank period.

Well, the MMC3 IRQ triggers right at the H-blank period, but by the time it jumps to the IRQ code, and you load a number to the scroll, it's already drawing the next scanline. So, in order to get it to change the scroll during H-blank, we have to wait for the next one. I wrote a tiny little loop, to wait about 100 CPU cycles, and then switch the scroll position. I think I'm timing it just right, but each emulator seems to be just a tiny bit different (ie. inaccurate), so I can't be sure.

After the H-scroll is changed, I set up another 'wait 20 scanlines and IRQ' bit. It's a bit tricky to get it to split exactly where you want. I've noticed that actual games don't do the wait loop like I'm doing here. What they do is have nothing but a flat single color at the scroll split point across the entire scanline, so the glitch isn't visible. Or, they just have a big glitch and don't worry about it.

If you want to see the glitch (so you know what I'm talking about), edit that tiny wait loop (in the IRQ code in reset.s) to be just 1 number bigger, or 1 number smaller. Recompile it. Glitches at every split! The H-blank is really that small.

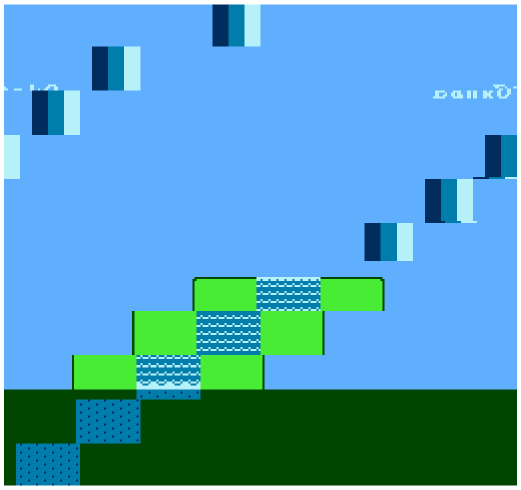


I still have 'START' change the bank number on screen…if you can read it. Here's the link to the source code…

http://dl.dropboxusercontent.com/s/1435iwsn62kixvg/lesson20.zip (http://dl.dropboxusercontent.com/s/1435iwsn62kixvg/lesson20.zip)

EDIT:

It occurred to me that, maybe people won't want to recompile this, but still want to see the glitch thing I've been yapping about. Here's an animated gif, with the wait loop off by just 1 loop. Look at the right side of the screen, the last scanline of each segment is misaligned, and that misalignment changes each frame, so it kind of dances around oddly. That's just being a few pixels off on the scroll split…imagine if the Scroll split was done halfway into the scanline. You'd have one scanline of each segment be as much as 80-100 pixels out of alignment. That would look terrible, and be distracting.



You might wonder why we would split the screen like this anyway? For parallax scrolling. Go to YouTube, and search for NES and Parallax Scrolling. You'll notice also, what I mentioned earlier, about most games do their scroll splits at a flat mono-color portion of the screen so glitches aren't noticable. That would have been better (and easier) than trying to carefully time an H-blank split.

NOTE about the error I made (and fixed). I named the first bank "CODE" in the .cfg file and defined it to be at $8000, and it was a swappable bank. Apparently "CODE" is the default name that the C compiler uses to put all the C library functions. We can't have that in a swappable bank, because if you go to call one of those C library functions (really, most code in C uses them) the bank that contains them might not be in place, and the game would crash. So, I called the last bank "CODE". That's the fixed bank. It will always be in place. Now, our C library functions will always be in the right position.

The other error that I fixed, was the IRQ code in "lesson20/reset.s". If the C code was doing anything that required the C stack or the C variables when the IRQ was called…doing any more C functions inside the IRQ handler will screw up the stack/variables, and when the IRQ is finished, and it jumps back to the MAIN() function, the game will promptly crash. So, I rewrote the entire IRQ code in ASM, to make sure none of the C stack / C variables are affected. Here's a link about that…

http://www.cc65.org/faq.php#IntHandlers (http://www.cc65.org/faq.php#IntHandlers)

January 15, 2016April 15, 2017 dougfraker

One thought on "24. MMC3, Bank-switching, IRQs"

1.
 w00tguy says:
 March 3, 2017 at 3:45 am Edit
 Wasn't working for me until I figured out that you can't just drop the PRGX definitions anywhere. In the default nes.cfg, the new definitions need to go right after ROM0, and then ROM0 needs to be removed.

Reply