nesdoug

Neslib Example Code

I thought this would take me 5 minutes. Boy was I wrong. Here's some examples on neslib use for NES development. I've made some changes, that will probably annoy everyone. Sorry.

I changed the cfg, moving the symbols to crt0.s, adding ONCE segment
I changed PAD_STATE to _PAD_STATE in crt0.s (etc), so I can access it from c code as
extern unsigned char PAD_STATE;
I made slight changes to neslib.s (notably removing _ppu_wait_frame as potentially buggy with split screen effects)

The first example is a simple "Hello World". Note, I have arrays of chars called "PALETTE" and "TEXT". As stated in an earlier blog post, I have created a tile set that positions the letters and numbers exactly in their ASCII position, so I can reference them with actual letters in the code "Hello World!".

This is how to write to the screen with rendering **OFF**. Set an address, vram_adr(), then write, vram_write().

```
void main (void) {

 // rendering is disabled at the startup
 // the init code set the palette brightness to
 // pal_bright(4); // normal

 // load the palette
 pal_bg(PALETTE);

 // load the text
 // vram_adr(NTADR_A(x,y));
 vram_adr(NTADR_A(10,14)); // screen is 32 x 30 tiles
 // this sets a start position on the BG, where to draw the text, left to right

 vram_write((unsigned char*)TEXT,sizeof(TEXT));
 // this draws the array to the screen
 // this function only works with rendering off, and should come after vram_adr()

 // normally, I would reset the scroll position
 // but the next function waits till v-blank and scroll is set automatically in th
 // since the RAM was blanked to 0 in init code, scroll variables will be x = 0, y

 // turn on screen
 ppu_on_all();

 // infinite loop
 while (1){

 // game code will go here.

 }
};
```

Pretty straightforward way to write to the screen with rendering off. Here's the source code.

http://dl.dropboxusercontent.com/s/5p8o0umed5k10r5/lesson21.zip (http://dl.dropboxusercontent.com/s/5p8o0umed5k10r5/lesson21.zip)

Part 2

This "Hello World" writes 1 letter at a time, then blanks it, and starts over. This is how to write to the screen when rendering is **ON**.

In this example, I am writing to the screen in 2 different ways, with rendering on. With rendering on, you will write data to a buffer, and set a pointer to that data. The nmi code will automatically push the data to the screen during v-blank.

Both of them are examples of set_vram_update(). The first, non-sequential data. This will write the letter A and the letter B at different screen locations.

```
// example of non-sequential vram data
const unsigned char TWOLETTERS[]={
MSB(NTADR_A(10,17)),
LSB(NTADR_A(10,17)),
'A',
MSB(NTADR_A(18,5)),
LSB(NTADR_A(18,5)),
'B',
NT_UPD_EOF}; // data must end in EOF
```

The second, using the CLEAR array, is a sequential data set. It will write 12 zeros to the screen, covering over the "Hello World!" when the loop ends. NT_UPD_HORZ or NT_UPD_VERT is required to tell the vram update to go sequentially.

```
// example of sequential vram data
const unsigned char CLEAR[]={
MSB(NTADR_A(10,14))|NT_UPD_HORZ, // where to write, repeat horizontally
LSB(NTADR_A(10,14)),
12, // length of write
0,0,0,0, // what to write there
0,0,0,0, // data needs to be exactly the size of length
0,0,0,0,
NT_UPD_EOF}; // data must end in EOF
```

And, I'm actually constructing a data set on the fly, when pushing letters of "Hello World!" one at a time to the screen.

```
v_ram_buffer[0] = high;
v_ram_buffer[1] = low;
data = TEXT[text_Position]; // get 1 letter of the text
v_ram_buffer[2] = data;
v_ram_buffer[3] = NT_UPD_EOF;
```

This is also an example of delay(), which waits a certain number of frames, before moving to the next line. Here's the main code (with some comments edited out).

```
void main (void) {

 // load the palette
 pal_bg(PALETTE);

 // set some initial values
 text_Position = 0;

 // turn on screen
 ppu_on_all();

 // load some non-sequential vram data, during rendering
 memcpy(v_ram_buffer,TWOLETTERS,sizeof(TWOLETTERS)); // copy from the ROM to the R
 set_vram_update(v_ram_buffer); // this just sets a pointer to the data, and sets
 // works only when NMI is on

 // infinite loop
 while (1){

    delay(30); // wait 30 frames = 0.5 seconds

    address = NTADR_A(10,14) + text_Position; // 2 bytes wide
    high = (char)(address >> 8); // get just the upper byte
    low = (char)(address & 0xff); // get just the lower byte

    v_ram_buffer[0] = high;
    v_ram_buffer[1] = low;

    data = TEXT[text_Position]; // get 1 letter of the text
    v_ram_buffer[2] = data;

    v_ram_buffer[3] = NT_UPD_EOF;

    ++text_Position;

    if (text_Position >= sizeof(TEXT)){
      text_Position = 0;
      ppu_wait_frame();
      memcpy(v_ram_buffer,CLEAR,sizeof(CLEAR)); // if at end, clear screen
      // by overwriting zeros over the text
    }

    set_vram_update(v_ram_buffer); // set a pointer to the buffer
    // it will auto-update during v-blank

  }
};
```

So, step 1, fill the v_ram_buffer. Step 2, set_vram_update(v_ram_buffer); will set a pointer to the data and set a flag to push the data to the PPU during the next v-blank.

Note: set_vram_update(NULL); will disable v-ram updates.

And, here it the source code…

http://dl.dropboxusercontent.com/s/cupgyz9bg8ibjny/lesson22.zip (http://dl.dropboxusercontent.com/s/cupgyz9bg8ibjny/lesson22.zip)

Note: here's a quick rundown of all the files, and what they are.

Alpha.chr is the graphics. You can edit it in a tile editor like YY-CHR.

Compile.bat is for windows users. Double-click will go through all the commands needed to compile the source code.

crt0.s is the starup/init code. It also is an easy place to 'include' ASM source code (or 'incbin' binary data files) to a project. It's where the graphics and vectors are included. It's where the music data will be included, later on.

famitone.s is the music code, for when we add music.

labels.txt is auto-generated by the linker, if you put a -g in Compile.bat (or if .DEBUGINFO is added at the top of every ASM source file). It's just a listing of all symbols and their location in the CPU memory. Very useful for debugging.

lesson21.c is the main source code.

lesson21.nes is the final NES ROM.

lesson21.s is an auto-generated file by the cc65 compiler. You can see what kind of ASM code is generated by the compiler, and possibly spot a bug.

License.txt is my open source MIT license.

makefile is for Linux users, who want to use 'make' to compile the source code. You might have to edit it…it looks like I set it up for Windows users also.

MoreLib.c was supposed to be a library of functions and define statements, but I haven't added much to it yet.

neslib.h is mostly prototypes for neslib functions.

neslib.s is the actual ASM code for the neslib functions.

nrom_128_horz.cfg is the linker map. It is set up for the smallest possible NROM configuration, using only half the available ROM space. If your files get bigger, you will need to change many of the sizes here.

April 13, 2017August 6, 2017 dougfraker