NESgen Documentation

By David Pokora

Contents

[GETTING STARTED 3](#_Toc484134149)

[Purpose/About 3](#_Toc484134150)

[NESgen 4](#_Toc484134151)

[NESsys 4](#_Toc484134152)

[Sprint Folders 5](#_Toc484134153)

[Input/Controls 5](#_Toc484134154)

[REQUIREMENTS 6](#_Toc484134155)

[Windows 6](#_Toc484134156)

[Mac OS 6](#_Toc484134157)

[Linux 6](#_Toc484134158)

[UNDERSTANDING THE SOURCE TREE 7](#_Toc484134159)

[COMPILING/BUILDING 8](#_Toc484134160)

[Building automatically with bash/batch scripts 8](#_Toc484134161)

[Building manually (with CMake) 8](#_Toc484134162)

[Building manually (without CMake) 8](#_Toc484134163)

[CONCLUSION 9](#_Toc484134164)

# **GETTING STARTED**

## Purpose/About

The purpose surrounding this project is fairly simple. As a Computer Science student at University of Toronto, we were required to make some kind of compiler for one of our courses. While *most* students opted for source-to-source compilers (and had some really great concluding projects!), I had figured exploring static recompilation of NES ROMs would explore many difficult challenges regarding compilation, while allowing me to toy around with some other fields: Operating Systems, Emulation, Disassembly, etc. This was done in a relatively short period of time given that I was enrolled fulltime in other courses and chasing passion projects on the side.

It was not meant to be a fully supported recompiler, nor to be cycle-accurate (although besides some negligible cases, it does do a decent job at making sure each instruction is accurate). Rather, as mentioned, this project aimed to explore issues regarding cleanliness vs. correctness. It aims to create code that is readable while avoiding interpretation at all costs. As shown in the ProjectDemo.pptx (a PowerPoint presentation that accommodated a final demo of our projects), the end result is code that closely resembles assembly, but leaves room for higher level predictive analysis. Why does it resemble assembly? I won’t go too far into depth (although one can further research problems surrounding static recompilation themselves), but the main complications surrounding static recompilation (as opposed to interpretation or dynamic recompilation, as all sensible emulator authors would opt for) is the lack of runtime context given when statically recompiling.

Without being able to predict values at certain points in a program, one cannot know how where the CPU’s program counter will end up, and thus, what sections are code (as opposed to data). This would be easier if NES ROMs were compiled with a compiler, but unfortunately they were written by hand, and allow developers to make use of unconventional, runtime calculated jumps. Often I would hear the counter-argument that “NES seems simple enough to predict as a whole, and statically recompile successfully.” Although more work can definitely be done (such as predicting value calculation at certain points in a program, or using some heuristic for detecting virtual tables or runtime calculated jumps), there are things that would prove to be extremely difficult to predict. Polling hardware for values at certain clock cycles may return very specific values the game may come to expect, for instance. One could likely make use of symbolic execution as a means to assist against many problems such as the ones I have mentioned, but they would either be impossible to, or take an infeasible amount of time to predict every case (of user input at every clock cycle, for instance, etc.).

As a result, I opted to keep the code as close to its original form as possible, while recompiling to what would still be considered true C code, free of interpretation. I opted to not support memory mappers (swapping different banks of program code/data into a finite memory space) for the reasons mentioned above, regarding predictive analysis, as it would be *extremely* difficult to determine which bank should be loaded at a given point in program execution. It does not guarantee support for runtime calculated jumps for the same reason (although it will attempt to calculate and jump nonetheless, in hopes that the recompiler did detect the code section accordingly at compilation-time). But, it will not emulate the CPU’s program counter to advance within program execution (relying on standard C execution semantics), and thus, there is no “PC” register the CPU will track. Nonetheless, it will allow the recompiled game to jump to any position in code (if the appropriate flag was set upon compilation time) to allow compatibility for unpredicted runtime calculated jumps.   
  
As a final note, the set of NES ROMs this was originally made to support a set of test ROMs provided at the following URL: <https://www.castledragmire.com/hynes/reference/resource/index.html> . Further information regarding support can be found within the /sprint/ documents which documented the project’s progress for marking throughout the course. It will support a very small amount of “real” ROMs such as Bomberman and Defender II, although not without issues. Bomberman will load without enemies, while Defender II may have unexpected results upon player death. These issues may be related to improper instruction or hardware emulation, but are more likely the result of improper runtime calculated jumps, or reliance NES stack integrity, which no longer pushes return addresses (since we rely on C-style execution semantics, as mentioned).  
  
Nonetheless, the goal of the project was met, and the subset of ROMs meant to be supported, are fully supported. At this point, continuing this project or rebuilding it with more robust high-level predictive analysis is not in my personal interests, and as such, I will retire it in its provided state.

## NESgen

NESgen is a python program used with Python 3.5. It can compile NES .ROM files into **game.c** and **game.h** files to compile with NESsys (the provided C project) which wraps some NES system functionality

* Basic usage: **NESgen.py [-n] [-f] -i <input.NES> -c <game.c> -h <game.h>**
* Here is an example I used to compile a ROM on my machine into NESsys, after **cd**’ing into the NESgen directory:
  + *“****python3.5 NESgen.py -i ~/Desktop/testROMs/Motion.NES -c ../NESsys/src/game.c -h ../NESsys/src/game.h****”*
* This provided example put the files directly into my NESsys folder, although you can place them elsewhere if you just intend to analyze them.
* The **–n** option will compile without optimizations from Sprint3.
* The **–f** option will compile using all PRG-ROM data, instead of just predicted data sections, as outlined in optimization #0 in Sprint4.

## NESsys

NESsys is a project which provides an environment to the compiled game C files. It emulates important system tasks that these games depend on to time certain actions and such.

* Once a **game.c/game.h** is placed in the **NESsys/src/** folder, you can compile it using the **compile.sh** (UNIX) or **compile.bat** (Windows) scripts in the **NESsys/** folder, which automate the compilation for you. They rely on CMake to find an appropriate compiler on the system. For instance, on a Windows machine, it may use Visual Studio dependencies to compile, whereas on Linux it may use GCC. The compiled executable can be found in the **NESsys/bin/** afterward. For windows machines this will have an .exe extension. For UNIX it will have no extension. On Windows, you will need to copy your installed **freeglut.dll** into the same folder as the executable you compiled for it to run, otherwise it will throw an error.

## Sprint Folders

The included sprint folders were used to track progress while developing this project. The project outline and requirements provided by our instructor can be found in **Requirements & Outline.docx** found within the **/sprints/** folder. Each sprint was submitted at a different point in time within the course, to be marked by the instructor and teaching assistant.

NOTE: Some sprints included a batch compiling script, which relied on the project in its state at the time the sprint was created, so they may not produce the same results now, but copies of the compiled results are included, and well documented within the respective sprint documents.

## Input/Controls

**Universal:**

* F5 = Restart game.

**Player 1 (left side, letters):**

* WASD = Directional keys
* N = B button
* M = A button
* J = Select
* K = Start

**Player 2 (right side, directional+numpad):**

* Directional Keys = Directional Keys
* 2 = B button
* 3 = A button
* 5 = Select
* 6 = Start

**Context Menu** **(right-click with mouse):**

* CPU Resume/Pause/Restart
* PPU Greyscale option
* Emulation Speed

# **REQUIREMENTS**

Although this project is cross-platforms and should support the majority of modern machines, it is worth noting that the product has dependencies that change per-platform, and has only been officially tested on Windows 10 x64 (with Visual Studio), Mac OS Sierra (with XCode/CC) and Ubuntu 16.04 x64 (with CC). All systems were tested using Python 3.5.

Compilation scripts are included along with instructions below, which were used to set up the machines mentioned above.

## Windows

* Python 3.5 (or similar)
* CMake is required (download appropriate .msi @ <https://cmake.org/download/>)
* Windows requires an installation of freeglut (tested with 3.0.0 from <http://freeglut.sourceforge.net/index.php#download>).
* Windows users should place the freeglut.dll included with the freeglut they installed in the same folder as the compiled binary, otherwise it may error asking for the dll.
* Visual Studio is recommended, but other windows supported toolchains such as MinGW-GCC can be used. (CMake will auto-resolve it).

## Mac OS

* Python 3.5 (or similar)
* CMake is required (download appropriate .dmg @ <https://cmake.org/download/>)
* OSX should download XCode from the AppStore which is bundled with GLUT, and has been tested.
* CC/GCC or some compatible toolchain must be available to build NESsys. (CMake will auto-resolve it)

## Linux

* Python 3.5 (or similar)
* CMake is required ("sudo apt-get install cmake")
* Linux users require freeglut ("sudo apt-get install freeglut3 freeglut3-dev")
* CC/GCC or some compatible toolchain must be available to build NESsys. (CMake will auto-resolve it)

# **UNDERSTANDING THE SOURCE TREE**

**The scripts contained in NESgen are described:**

* **NESgen.py**: Main entry point
* **iNESROM.py**: Provides information about the ROM, and location/size of PRG-ROM/CHR-ROM.
* **NESMemory.py**: Memory/file address translation.
* **MOS6502Instructions.py**: MOS6502 Instruction Set
* **PRGROM.py:** Parses the NES ROM’s underlying PRG-ROM into code/data sections, storing code sections (a list of instructions) separately based off of change in control flow (jumps into a section will split it, instructions which statically jump or return mark end of code section area).
* **iNESROMDisassembler.py:** Uses **PRGROM.py** to generate ASM/C code.

**The source files contained in NESsys are described:**

* **cpu.c:** Implements CPU registers, stack (return address no longer stored on it since we don’t handle program counter), and a function to simulate x CPU cycles, which causes PPU to cycle 3\*x times, then checks for interrupt requests to handle, and finally, throttle’s execution speed.
* **game.c:** The NES ROM compiled into C source to be executed within NESsys.
* **input.c:** Handles keyboard/controller input.
* **instructions.c:** Functions which modified the CPU registers accordingly as the MOS6502 instruction normally would.
* **memory.c:** Memory mapping, reading, writing implementation for both CPU/PPU (and map’s game data sections into CPU address space as well, and game’s CHR-ROM into PPU address space).
* **NESsys.c:** Main entry point, handles actual OpenGL rendering, window, callbacks, etc.
* **platform.c:** Used to implement universal features/includes used throughout the program, including cross-platform wrappers.
* **ppu.c:** Picture Processing Unit, handles PPU’s special register operations, and rendering.
* **tests.c**: A few random tests used when implementing certain features. Not heavily maintained. Was simply to test before ROMs were working.

It is worth noting that **NESsys.h** contains many configuration variables, such as debug logging, whether to use **game.c** or the template **game\_base.c**, and testing mode which runs the tests in **tests.c** instead of the game. Also contains a multiplier for game speed.

# **COMPILING/BUILDING**

This section focuses on how a user can compile/build a converted NES ROM into a binary executable to be run on the target machine. NESsys can be thought of as a NES system wrapper (it provides functions that the NES system would), and NESgen simply takes a NES ROM, and converts it to C code which can be thrown into the NESsys project and compiled to create a runnable version of the converted NES ROM.

## Building automatically with bash/batch scripts

* **Windows**: Run compile.bat to compile.
* **OSX/Linux**: Run compile.sh to compile.

The "-c" argument can be used to clean the **build**/**cache** folder for both these scripts. This is important when adding files/changing structure of the project during development, so CMake can recreate a new project with updated references for the appropriate platform, which are then used to build an executable. Cached project files \*may\* have outdated references if moving across platforms.

The compiled binary will be output to the **/bin/** folder.

## Building manually (with CMake)

The automatic compilation scripts mentioned above simply call CMake to create the appropriate project files/makefiles to compile the code for the target platform. You can manually compile on any platform by running the following commands:

**cd** *\WhateverFolderYouWantBuildFilesToBeIn\*

**cmake** *srcFolderPath*

**cmake** **--build** *srcFolderPath*

The output project/build files will be within the folder you had executed the **cd** command on. The binaries built by the final command will be output into the **\bin\** directory.

**NOTE:** The output project files can be used to build the project manually afterward if you please. For instance, CMake building on Windows with Visual Studio 14 available will create the appropriate .vcproj project files in the \build\ directory, which can be opened in Visual Studio, and ran/debugged from there.

## Building manually (without CMake)

Although it is not recommended, you can build the project manually with your choice of compiler/IDE. You must still have all other dependencies mentioned in the Requirements section of this document (a valid compiler for the target platform and GLUT/freeglut).

NOTE: GLUT installation/linking differs per machine. The Requirements section of this document explains how to obtain GLUT appropriately for each tested operating system. Linking the libraries to the project for your given compiler will differ, so a bit of research may be required.

# **CONCLUSION**

There isn’t much to say here (as everything regarding the project has been said in the “Purpose/About” section), but I just wanted to give a special thanks to Andrew Petersen and Michael Liut (course instructor and teaching assistant, respectively) for the great job they did teaching the Compilers and Interpreters course which this project was developed and submitted for. As a result of their flexibility in allowing various different types of (re)compilers, we saw some really great submissions. Beyond that, they were great in covering many important topics in compilers and interpreters that many projects did not cover. Thank you.