Introduction

Introduction

Basic information about FCEUX and its features.

Introduction

Overview

FCE Ultra Version History

What's Combined In FCEUX?

2008

Introduction

Welcome to the FCEUX Help menu.

The following information is about how to use FCEUX, its commands, how to use FCEUX to its fullest, and the communities for which FCEUX is designed.

Introduction

Basic information about FCEUX and its features.

<u>Overview</u>

FCE Ultra Version History

What's Combined In FCEUX?

Additional Chapters

<u>General</u>

Guides for general uses of FCEUX and the FCEUX NES menu.

Config

Commands under FCEUX Config menu.

<u>Tools</u>

Commands under FCEUX Tools menu.

Debug

Commands under FCEUX Debug menu.

FAQ / Guides

Information regarding various concepts such as TAS, ROM Hacking, RAM Mapping.

Technical Information

Technical information relating to NES hardware emulation & FCEUX file formats.

Help menu created & compiled by adelikat. Information collected and/or written/edited by adelikat. Minor edits of lua-related text by Leeland Kirwan (FatRatKnight).

2008

Overview

<u>FCEUX</u>

FCEUX is a cross platform, NTSC and PAL Famicom/NES emulator that is an evolution of the original FCE Ultra emulator. Over time FCE Ultra had separated into many separate branches.

The concept behind FCEUX is to merge elements from FCEU Ultra, FCEU rerecording, FCEUXD, FCEUXDSP, FCEUXDSP CE, and FCEUmm into a single branch of FCEU. As the X implies, it is an allencompassing version of the FCEU emulator that provides the best of all worlds for the general player, the <u>ROM-hacking community</u>, and the <u>Tool-Assisted Speedrun Community</u>.

2008

FCE Ultra Version History

History of FCEUX / FCE Ultra

FCEUX was started in 2006 by zeromus and rheiny (sp) as an attempt to merge various branches of FCE Ultra into a unified emulator. Additional authors joined the project, including mz, adelikat, nitsujrehtona, maximus, CaH4e3, qFox, punkrockguy318, and Sebastian Porst.

FCEUX contains all features and enhancements from FCE, FCE Ultra, FCEU rerecording, FCEUXD, and FCEUXDSP as well as many new mappers from FCEU-mm.

FCEUX sourceforge page

Version Releases

Look at the Side Bar navigation for changelog information on FCEUX 2.1 and newer.

FCEUX 2.0.3 - Released November 02, 2008 (see changelog)

FCEUX 2.0.2 - Released August 14, 2008 (see changelog)

FCEUX 2.0.1 - Released August 04, 2008 (see changelog)

FCEUX 2.0.0 - Released August 02, 2008 (see changelog)

FCE / FCEUltra

Bero originally wrote a Nintendo Entertainment System/Famicom emulator that was referred to as <u>FCE</u>. This name was apparently meant only to serve as a temporary name, but its usage remained. Xodnizel originally ported it to Linux SVGAlib, and made a few improvements. This code base was abandoned, and work began anew, under DOS, with the original FCE source code. At the end of November, 1998, FCE Ultra Beta 1 was released.

FCE Ultra remained DOS-only until version 0.18, when it was ported to Linux SVGAlib, and released as a statically-linked executable. The first MS Windows port was released as version 0.25.

The source code of 0.40 was released on November 12, 2000. It retained the simple license of FCE for a long time, which stated that "This software is freeware. You can use it non-commercially." Almost two years later, in June 2002, 0.80 was released, and FCE Ultra was re-licensed under the GNU GPL.

It has been tested (and runs) under DOS, Linux SVGAlib, Linux X, Mac OS X, and Windows. A native GUI is provided for the Windows port, and the other ports use a command-line interface. The SDL port should run on any modern UNIX-like operating system (such as FreeBSD, Solaris or IRIX) with no code changes. It has also been ported to the GP2X, PlayStation Portable as PSPFceUltra, the Nintendo GameCube and Pepper Pad.

FCE Ultra was created by Xodnizel. Development appeared to stop and the homepage and forums for the emulator were taken down. The last version before this was v0.98.13-pre, released in September 2004 as source-only. The last binary release was v0.98.12 in August 2004.

However, it was resurrected again in March of 2006 by Anthony Giorgio and Mark Doliner.

There is also a graphical frontend for FCE Ultra. GFCE Ultra is written in Python and uses the GTK2 user interface library. Because is it written in Python and with portability in mind, it can be run on any UNIX-like platform and any processor architecture that is supported by Python.

FCEU Rerecording

The "rerecording" version of FCE Ultra was implemented to FCE Ultra 0.98.10 with <u>movie recording</u> support. This was done by blip, and was implemented for the purpose of creating <u>Tool-Assisted Speedruns</u>.

The rerecording branch continued with 0.98.12, adding movie support features, such as "bullet proof" recording. In 2006, FCEU 0.98.16 was implemented by nitsuja and luke. Various tools such as read-only toggling, increased <u>hotkey mapping</u>, and <u>memory watch</u> were added.

In 2008, FCEU rerecording was picked up again by mz, maximus, adelikat, and nitsujrehtona with various updates named FCEU.0.98.17 - 0.98.28

FCEU.28 GoogleCode Page

FCEUD / FCEUXD / FCEUXDSP / FCEUXDSP CE

FCEUD

In 2002, Parasyte modified the then-current version (0.81.3) of FCE Ultra and added a Nesten-style debugger, along with several other features, and named it "FCEUD" (FCE Ultra Debugger).

FCEUXD

In January 2004, bbitmaster began working on more features and called it "FCEUXD" (FCE Ultra Extended Debugger).

It is a branch of FCE Ultra that contains many extended debugging features compared to the original FCE Ultra code such as a trace logger, a built-in hex editor, a name table viewer, code/data logger, inline assembler, and Game Genie decoder/encoder in addition to the debugger and PPU viewer from FCEUD. The last version made was FCEUXD 1.0a.

FCEUXDSP

FCEUXDSP stands for FCEUXD "SP" version and is a branch of FCEUXD 1.0a.

It was created in 2006 by sp. The project extends the debugging tools even further compared to FCEUXD by adding new tools, functions, and usability of debugging tools.

The last version of FCEUXDSP was 1.07 which adds a feature known as the RAM Filter. This has since been removed, due to functional redundancy.

FCEUXDSP homepage

FCEUXDSP CE

CE stands for "Champion Edition" and is a branch of XDSP that adds a text hooker tool.

FCEUXDSP CE homepage

FCEU-mm

FCEU "mappers modified" is an unofficial build of FCEU Ultra by CaH4e3, which supports a lot of new mappers including some obscure mappers such as one for unlicensed NES ROM's.

FCEUX supports mappers from older versions of FCEU-mm.

FCEU-mm SourceForge page

2008

What's New? 2.1.2 (changelog)

What's New? 2.1.4

Released 31 May 2010

The 2.1.4 release that fixes some many bugs and adds new features compared to 2.1.3. In addition it also fixes up the movie code significantly; fixing implementation problems, loading speed, adding new features, and fixing bugs.

Common

- Added microphone support option. When enabled, Port 2 Start activates the Microphone
- Prevent .zip files containing no recognized files from causing crash
- Autohold Added player 3 and 4 to autohold notification window, labeled controller input
- mapper 19 savestate fix mirroring for "Dream Master (J)" corrected to "four-screen" by CRC check
- Disable auto-savestates during turbo
- Fixed so Gotcha! auto-enables the zapper
- Autohold Added player 3 and 4 to autohold notification window, labeled controller input

Movies

- Fully implemented "bulletproof" read-only

- Movie code now fully conforms to the Savestate section of the Laws of TAS
- Fixed a potential desync that plays out an extra frame without an update to the frame count involving heavy lua use, joypad.get, and a loadstate
- Movie support for microphone
- Movies now have a "finished" mode. If a playback stops the movie isn't cleared from memory, and can be replayed or a state loaded Similar functionality as DeSmuME and GENS rerecording

- New PPU flag in movie headers (doesn't change an emulators PPU state when loading a movie)
- Much faster movie loading and movie-savestate loading
- Made gamepad 2 off by default (so less movies should have unused player 2 data)
- Implemented a "full savestate-movie load" mode similar to the implementation in VBA-rr and SNES9x-rr. In this mode loading a savestate in read+write doesn't truncate the movie to its frame count immediately. Instead it waits until input is recording into the movie (next frame). For win32 this feature is togglable in movie options and the context menu. For SDL this is off by default and a toggle will need to be added
- Movie + loadstate errors are handled more gracefully now, with more informative error messages and the movie doesn't have to stop if backups are enabled
- Fix PlayMovieFromBeginning when using a movie that starts from savestate

Lua

- fix bug that caused zapper.read() to crash when movie playback ends
- Win32 Added option for palette selection as color for LUA colors. Included is a LUA script to display all choices with the value used to pick displayed color

New Lua functions

- movie.ispoweron()
- movie.isfromsavestate()
- emu.addgamegenie()
- emu.delgamegenie()
- savestate.object() which is savestate.create() with intuitive numbering under windows
- gui.getpixel() which gets any gui.pixel() set pixel colors, and possibly other functions
- emu.getscreenpixel() which gets the RGB and Palette of any pixel on the screen
- lua function movie.getfilename() which returns the current movie filename without the path included

Input Display

- Input display updates on loadstate
- Input display overhaul that uses different colors for different input contexts
- Input display now shows both currently pressed buttons and buttons held the previous frame

Win32

- Added NTSC 2x scalar option with some CFG config options of it's own Added Ram Search hotkeys for the first 6 search types in the list
- Add Cheat buttons for Ram Search and Ram Watch
- With special scaler in window mode, it's possible to resize to anything above the minimum.
- Recording a new movie adds it to recent movies list
- Replay dialog, when selecting a movie in a relative path (.\movies for example), the recent movies list stores an absolute path instead
- Replay dialog shows PAL flag and New PPU flags
- CDLogger fixed bug preventing correct interrupt vectors from logging
- Memwatch ignore spaces at the beginnign of an address in the address boxes
- Replay dialog fix bug that was causing it to always report savestate movies as soft-reset

Debugger

- Added conditional debugging option 'K', for bank PC is on
- Fixed bug involving pausing emulation outside of the debugger, then trying to use the debugger commands, and having the CPU registers become corrupted
- Made debugger able to break on and distinguish Stack reads/writes

Hex Editor

- Added "Goto" command
- Made the Hex Editor display the Frozen, Bookmarked, etc. status of

the selected address, and made the Frozen color override the Bookmarked color.

Cheat Search

- Made enabling/disabling cheats no longer deselect the selected cheat
- Added context menu to Cheat Dialog Cheat Listbox, populated list with Toggle Cheat, Poke Cheat Value, and Goto In Hex Editor
- Enabled multi-select for Cheat menu to allow multiple toggles and deletes
- Made cheat menu's Pause When Active effect immediate

GUI

- Added Tools GUI option to partially disable visual themes, so the emulator can be made to look like it did in 2.1.1 and earlier releases. Drag & Drop - if dropping a .fcm with no ROM loaded, prompt for one (same functionality that was added to .fm2 files)
- Added single-instance mode, which makes starting a second copy of FCEUX load the file into the first, then exit.Mode off by default, togglable under Config - GUI

2008

What's New? 2.1.2 (changelog)

What's New? 2.1.3

Released April 8, 2010

The 2.1.3 release that fixes some bugs of 2.1.2, increases game compatibility, and adds usability enhancements to the windows port and adds a GUI to the SDL port.

Common

- Fixed mappers 82, 25, 21, and 18. Games such as SD Kiji Blader, Ganbare Goemon Gaiden, and Ganbare Goemon Gaiden 2, Jajamaru Gekimadden are now playable
- Fixes for mappers 253 & 226 fixes games such as Fire Emblem (J) and Fire Emblem Gaiden (J)
- Fix crashing on game loading for any battery backed ROMs with mappers from MapInitTab (fixes Esper Dream 2 Aratanaru Tatakai (J)
- FDS show name of missing bios file in error message
- NewPPU fixed sprite hit before 255 and for non transparent hits only, thanks to dwedit for providing the fix
- .fm2 file format header now has an FDS flag

SDL

- A GUI! A graphic user interface (using GTK) with many basic menu options
- ported to SDL 1.3; compatibility maintained with 1.2
- unix netplay is now functional; gtk network gui created
- now prints the name of the mapper on ROM load
- fixed dpad/joyhat support
- VS unisystem keys now configable
- changed default hotkeys and keys to match Win32
- disallow --inputcfg gamepad0 and gamepad5

Win32

- Made savestate backups optional (config enable backup savestates)
- Made savestate compression togglable (config enable compress savestates)
- Cheats dialog Pause while active checkbox
- Cheats dialog Toggling a cheat in the cheats list now updates the active cheats count
- Debugger added an auto-load feature
- Debugger Fix so it doesn't crash if unminimized with no game loaded
- Closing minimized windows no longer moves them the next time they get opened
- Lua console added a menu
- Lua console filename updates when lua scripts are dragged to emulator or recent filenames invoked
- Name Table Viewer Fix for use with New PPU
- Trace Logger Trace logger now logs the values of the stack pointer register
- If a .fm2 file is drag and dropped with no ROM load, the open ROM dialog will appear
- disable movie messages menu item
- Added more window positions bounds checks. Accounts for -32000 positions and less out-of-range too
- TASEdit Added interface functionality (save/load, running TASEdit mid-movie, etc.)

Lua

- New lua functions: gui.parsecolor(), joypad.getup(), joypad.getdown(), emu.emulating()
- Change gui.line, gui.box, joypad.get to function consistently with other lua emulators such as GENS rerecording
- fixed zapper.read() to read movie data if a movie is playing. Also changed the struct values to x,y,fire. This breaks lua scripts that used it previous, sorry
- gui.text() now has out of bounds checking

• Lua no longer unpauses the emulator when a script is loaded

2008

What's New? 2.1.2 (changelog)

What's New? 2.1.2

Released November 3, 2009

The 2.1.2 release that fixes some bugs of 2.1.0a, increases game compatibility, launches a new PPU core, and adds usability enhancements to the windows port.

Common

- New PPU is now functional! You can access it by changing the newPPU flag in the config file. Windows users can access it from Config > PPU > New PPU
- Dragon Ball Z 3 now playable again
- Fixed action 52 game that was broken in post-FCEUX 2.0.3 versions
- Mapper 253 mostly implemented
- Mapper 43 fixed bug

Win32

- Imported NSF features from FCEUXDSP-NSF. Debugging tools are now compatible with NSF files.
- Movies now record FDS disk swapping commands
- Movie play dialog displays movie time based on ~60.1 (~50.1 PAL) instead of 60 & 50
- Ram Watch and Ram Search dialogs imported from GENS rerecording
- Ram Filter dialog removed (now redundant compared to both cheat search and ram search)
- Lua script window ported from GENS
- Fix for the directory overrides bug that caused overrides to reset
- Debugger: .deb file saving/loading restored
- "Save config file" menu item
- "New PPU" menu item

Minor Bug fixes

- Minor fixes to recent menus
- Fixed a bug that prevented the Map Hotkeys dialog's X button from closing the dialog
- Restored DPCM Logging when Code/Data Logger is active
- Memory watch Save Changes Prompt clicking save will default to quicksave first and save as 2nd (instead of always defaulting to save as)
- Made Trace Logger refresh adequately when using stepping options in the debugger.

Lua

- joypad.set() fixed. True,False, and Nil now work properly for all buttons. In addition there is a new "invert" option.
- Lua5.1.dll no longer required to use lua.
- fceu.unpause()
- Added savestate.registerload(), savestate.registersave(), savestate.loadscriptdata()
- emu. library, has all the same functions as fceu. library for better compatibility between lua emulators
- Many additional function names to increase consistency with other lua emulators
- Added movie.recording() and movie.playing()
- Added memory.getregister() and memory.setregister()
- Added gui.popup and input.popup
- Added savestate.registerload(), savestate.registersave(), and savestate.loadscriptdata()

New Lua Scripts

- A multi-track movie recording tools written by FatRatKnight. Allows input for different players to be recorded separately.
- A rewinding tool written by Jonathan Blow

What's New? 2.1.1 (changelog)

What's New? 2.1.1

Released July 29, 2009

This release includes a multitude of new features, major fixes, and enhancements.

The 2.1 new release that fixes some bugs of 2.1.0a, improves the accuracy of the sound core, and adds useability enhancements to the windows port.

Common - Bug fixes

- Fixed reported issue 2746924 (md5_asciistr() doesn't produce correct string)
- Made default save slot 0 instead of 1

Improved Sound core/PPU

- Fixed the noise value, it seems that the noise logic was shifting the values to the left by 1 when reloading, but this doesn't work for PAL since one of the PAL reload value is odd, so fix the logic and used the old tables. Revert a stupid CPU ignore logic in PPU. Sorry about that.
- Updated with the correct values for the noise and DMC table,
- Fixed the CPU unofficial opcode ATX, ORing with correct constant \$FF instead of \$EE, as tested by blargg's. These fixes passes the IRQ flags test from blargg, and also one more opcode test from blargg's cpu.nes test.
- Square 1 & square 2 volume controls no longer backwards
- Length counters for APU now correct variables

NewPPU (still experimental, enabled by setting newppu 1 in the config file)

- Added experimental \$2004 reading support to play micro machines with (little) shakes, and fixed some timing in the new PPU.
- Added palette reading cases for the new PPU.

Win32

Minor Bug fixes

- Replay movie dialog Stop movie at frame x feature fixed off by 1 error on the stop frame number
- Hex Editor changed ROM values again dsiplay as red, saved in the config as RomFreezeColor
- Fixed bug in memory watch that would make the first watch value drawn in the wrong place if watch file was full
- Debugger Step type functions now update other dialogs such as ppu, nametable, code/data, trace logger, etc.
- "Disable screen saver" gui option now also diables the monitor powersave
- Recent menus no longer crash if item no longer exists, instead it ask the user if they want to remove the item from the list
- Sound Config Dialog When sound is off, all controls are grayed out
- Memory Watch fixed a regression made in 2.0.1 that broke the Save As menu item
- Memory Watch save menu item is grayed if file hasn't changed

GUI/Enhancements

- Last save slot used is stored in the config file
- Made fullscreen toggle (Alt+Enter) remappable
- Hex editor Reverted fixedFontHeight to 13 instead of 14. Gave the option of adjusting the height by modifying RowHeightBorder in the .cfg file
- Hex Editor allowed the user to customize the color scheme by use of RGB values stored in the .cfg file
- Hex editor freeze/unfreeze ram addresses now causes the colors to update immediately, but only with groups of addresses highlighted at once (single ones still don't yet update)
- Hex Editor Save Rom As... menu option enabled and implemented
- Window caption shows the name of the ROM loaded

- Recent Movie Menu added
- Load Last Movie context menu item added
- Save Movie As... context menu item (for when a movie is loaded in read+write mode)
- Drag & Drop support for all files related to FCEUX including: .fcm (autoconverts to .fm2 and begins movie playback) Savestates Palette files (.pal)
- Commandline -palette commandline option
- Memory Watch option to bind to main window, if checked it gives GENS dialog style control, where there is no extra task bar item, and it minimizes when FCEUX is minimized

SDL

- added --subtitles
- fixed Four Score movie playback
- added --ripsubs for converting fm2 movie subtitles to an srt file
- Lua is optional again, fixed the real issue
- Lua is NO longer optional, so the SConscripts have been updated to reflect that change. This fixes the mysterious non-working input issue.
- implemented saving/loading a savestate from a specific file on Alt+S/L
- implemented starting an FM2 movie on Alt+R
- added --pauseframe to pause movie playback on frame x
- dropped UTFConverter.c from SDL build
- added hotkey Q for toggling read-only/read+write movie playback

2008

What's New? 2.1 (changelog)

What's New? 2.1

Released March 29, 2009

This release includes a multitude of new features, major fixes, and enhancements.

New Features Win32

*The latest mappers and mapper fixes from FCEU-mm. Adds support for many new games such as Warioland II (Unl), Shu Qi Yu, and Street Dance

*Full screen mode fixed! Also, Alt+Enter properly toggles full screen. *Individual control for sound channels! (See <u>sound config</u> for details).

*Undo/Redo Savestate/Loadstate features installed! No more loss of data to unintentional presses. (See <u>getting started</u> for details).

*Movie subtitles can now be included in .fm2 files. See .fm2

documentation for details and <u>Movie options</u> for details on customizing. *Auto-backup for movie files. (See <u>movie options</u> for details).

*A Ram change monitor for the Memory watch dialog. (see <u>memwatch</u> for details).

*Frame counter works even without a movie loaded.

*AVI Directory Override option.

Major Bug / Crash Bug Fixes

*Fixed throttling problems that resulted on AMD Dualcore processors. (Caused FCEUX to appear to be in turbo mode).

*Fix major crash issue where NROM game (such as SMB) savestates were writing erroneous information if a non NROM game was loaded prior.

*Fixed a bug that caused a new sav file to not get created when loading a 2nd battery backed game.

*Fix Directory Overrides so to allow users to have no override. Also fixes

directory override reset bug.

Minor Bug fixes

*Hotkeys - prevent "Hotkey explosion" where some laptop keys set off all unassigned hotkeys

*Timing - "disable throttling when sound is off" now only affects FCEUX when sound is off

*Clip Left and Right sides taken into account when drawing on screen (record/play/pause, lag & frame counters, messages, etc)

*Fixed bug where having sound off and Mute turbo caused chirps when toggling

*Video settings - fixed bug when both aspect ratio correction and special scaling 3x are set, video was getting resized incorrectly

*Auto-save cleanup -prevent loading an auto-save from previous session. Added flags for enabling auto-save menu item.

*Fixed issues related to big endian compiling.

*Fix bug so that Escape can now be assigned as a hotkey

*Fixed bug in screenshot numbering that caused numbering to not reset when changing games.

GUI / Menu Enhancements

*A right-click context menu added! Includes many commonly used items for a variety of situations.

*Menu items that are hotkey mappable now show their current hotkey mapping

*Major overhaul to the Menu organization.

*All FCEUX features are now accessible in the menu

*Alt Menu Shortcuts properly configured

*Menu items are properly grayed when not useable

*All movie related menu items moved to a Movie options dialog

*Removed hard-coded Accel keys and replaced with re-mappable hotkeys (Open & Close ROM)

*Drag & Drop for .fm2 and .lua files

*Many new functions added to the context menu (See <u>context menu</u> for

details)

*New Mappable Hotkeys: Open Cheats, Open ROM, Close ROM, Undo/Redo savestate, Toggle Movie Subtitles

Lua

*Added input.get() ! Returns the mouse info and all keyboard buttons pressed by the user.

*Fixed joypad.set(). False now sets a button to off. Nil does not affect the button at all (allowing the user to still control it).

*gui.text() Increased height (to approx. 7 lines).

*speedmode("turbo") now turns on turbo (which employs frame-skipping) rather than max speed.

*memory.readbyte will recognize frozen addresses (cheats).

*movie.framecount() always return a number, even when no movie is playing (since the frame counter is implemented without a movie loaded).

*Added FCEU.poweron()

*Added FCEU.softreset()

*Added FCEU.lagged()

*Added FCEU.lagcount()

*Added FCEU.getreadonly()

*Added FCEU.setreadonly()

*Added FCEU.fceu_setrenderplanes(sprites, background)

*Added movie.active()

*Added movie.rerecordcount()

*Added movie.length()

*Added movie.getname()

*Added movie.playbeginning()

*Added -lua command line argment, loads a Lua script on startup

*Added zapper.read() - returns the zapper (mouse) data. (Currently does return zapper data in movie playback).

*Added joypad.write and joypad.get for naming consistency.

*Added rom.readbyte()

*Added rom.readbytesigned()

Sound Config

*Turning sound off disabled sound config controls *Re-enabled sound buffer time slider control

Hex Editor

*Freezing ram addresses automatically updates the Cheats dialog if it is open.

* Added prevention from freezing more than 256 addresses at once (doing so caused crash bugs).

*Dialog remembers window size.

*Dump Rom & Dump PPU to file Dialogs use ROM to build default filename

*Maximize and minimize buttons added.

*Help menu item added

Memory Watch

*Dialog now includes Ram change monitoring. (see <u>memwatch</u> for details).

*Dialog is now collapsible to 1 column.

*No longer crashes when attempting to load an invalid file from the recent file menu.

*Cancel option added to the save changes dialog.

*Memory address values that are frozen by the debugger or hex editor are displayed in blue.

*Fixed bug that caused dialog to "disappear" due to saving -32000 as its window position.

*Save as dialog uses ROM name to build a default memory watch filename if there was no last used memory watch filename

*Drag and drop for .txt (memory watch) files.

*Minor menu and hotkey fixes.

*Watch values now compatible with custom windows dialog colors.

Debugger

*Shows scanlines and PPU pixel values

*Shows scanlines even while in VBlank

*Added a Run Line button (runs 1 scanline per click)

*Run 128 Lines button (runs 128 scanlines per click)

*Number of active cheats listed.

*Cheats list automatically updated if ram addresses are frozen in the hex editor.

*Fixed bug that caused dialog to "disappear" due to saving -32000 as its window position.

*Debugger now has a minimum valid size

*Added "Restore original window size" button

PPU Viewer

*Default refresh value set to 15 *Refresh value stored in the .cfg file

Nametable Viewer

*Default refresh value set to 15 *Refresh value stored in the .cfg file

Trace Logger

*Fixed bug where user can't scroll the log window while it is autoupdating.

*Changed message about F2 pause (left over from FCEUXDSP) to display the current hotkey mapping.

Text Hooker

*Saving a .tht file no longer crashes

*Dialog updates every frame

*Initialization error checking reinstalled,

*Dialog remembers window position

*Fixed bug where canceling save as produces an error message.

*Save As produces default filename based on the current ROM

Message Log

*Remembers X,Y position *Resized width and height *Allowed more lines of text to appear on the screen at once.

Metadata

*Remembers window position

*Can be called from the context menu if a movie is loaded (see <u>context</u> <u>menu</u> for details).

TASEdit

*added help menu item

*disabled menu items that are not currently implemented.

Turbo

*Turbo now employs frame skip, greatly increasing its speed *The mute turbo option completely bypasses sound processing (another big speed boost)

*Turbo now works with the Lazy wait for VBlank sync setting

SDL

*SDL Movie subtitle support and subtitle toggle hotkey added.

*SDL Added fcm to fm2 converter tool to SDL version.

*SDL Improved the SDL sound code; drastically improves quality of sound.

*SDL Savestate slots are now mappable.

*SDL Major updates to SDL documentation

*SDL Added Shift+M for toggling automatic movie backups.

*SDL Added option to mute FCEUX for avi capturing, check the documentation for more details.

*SDL Added --noconfig command line option *SDL Frame Advance Skip Lag frames toggle implemented

2008

What's New? 2.0.3 (changelog)

What's New? 2.0.2

Released November 02, 2008

This release includes some key bug fixes and feature enhancements.

Major Bug / Crash Bug Fixes

* Reset/Power-on recording for .fm2 files!

* fix ..fcm conversion, recording, and playback of reset and power commands

* Win32 - auto-load the only useful ROM or movie from an archive, in cases where there is only one

* Win32 - permit user optionally to proceed through the movie savestate mismatch error condition, in case he knows what he is doing.

* Win32 - fix a bug in the savestate recovery code which prevent aborted savestate loads from recovering emulator state correctly.

* gracefully handle non-convertible broken UTF-8 text without crashing * Win32 - don't read every archive file when scanning for replay dialog. scan them, and only look for *.fm2

New Features Win32

- * Win32 added a toggle for binding savestates to movies
- * Win32 added -cfg (config file) command line argument

Minor Bug fixes

* Win32 - Sound config dialog will now look to see if Mute Turbo should be checked

* Win32 - Debugger - Fix Child windows inside debugging window get invalid sizes

- * Win32 bind a menu option for display frame counter
- * Win32 fix problem where replay dialog couldn't work when the process

current directory had changed to something other than emulator base directory

* Lua ignores second joypad.set()

* Load state as... does not use the savestate override dir (fixed; now, it does)

*Win32 - debugger - fix issue where keyboard keys get stuck when switching between debugger window and main window

SDL

* SDL - added support for AVI creation for SDL, see documentation/Videolog.txt for more

* SDL - --inputcfg can now be used without a filename

* SDL - should fix issues with missing author field crashing FCEUX

* SDL - toggle lag frame counter for SDL, default hotkey F8

* SDL - toggle skipping of lag frames for SDL, default hotkey F6

* SDL - user ability to toggle "bind savestates to movie" added for SDL, default hotkey F2

* SDL - Lua is now optional, thanks Shinydoofy for a patch. also fixed some build issues.

* SDL - fixed an issue where flawed movie would crash FCEUX on every startup

* SDL - fixed issue where windowed mode would always be set to 32 bpp

* SDL - fixed ppc build errors and added LSB_FIRST option to build scripts

* SDL - -- newppu option added to SDL, disabled by default

GFCEUX (SDL)

* GFCEUX - made the input config window more usable

* GFCEUX - added uninstall script for GFCEUX

2008

What's New? 2.0.2 (changelog)

What's New? 2.0.2

Released August 14, 2008

This release includes a large number of bug fixes, feature enhancements, and new features.

Fixed Crashing Bugs

* restore savestate error recovery functionality. Will prevent crashes after savestate error messages

- * Fixed Low speeds (1%) crash FCEUX
- * fixes bug where palflag 1 in .fm2 files crashes fceux
- * FCEUX no longer crashes when attempting to open a non movie file
- * Buffer overflow (change vsprintf to vsnprintf)

Minor Bug fixes

- * SRAM not wiped on power cycle (during movies)
- * Moviefilenames without extension now automatically get fm2
- * auto-fill .fcs extension in save state as dialog
- * FCM>FM2 converter releases file handle

* fix a new bug in windows build which caused fourscore emulation to fail in some cases

* Player 3 no longer inputs when not used

* prints a special message when trying to open an FCM reminding user to convert.

- * fixes bug where Avi recording with no sound messes up the format
- * Fixed bug where Convert .fcm didn't do special characters
- * fixed the (null) in the default lua directory listing
- * Ctrl+X now works in the memory watch dialog

* Dialog window positions won't "disappear" (-32000 protection on all dialogs that remember x,y)

* fixed View Slots bug - will now always show the used slots

- * added shift+L as default hotkey for reload lua script
- * added input display to the FCEUX main menu
- * change config filename from fceu98.cfg to fceux.cfg

New Features

* restore IPS patching capability which was lost when archive support was added

* restore ungzipping (and unzipping in sdl) capability which was lost when archive support was added

* re-enable an "author" text field in the record movie dialog

* re-enable support for old-format savestates. (Note: can not be loaded into a movie!)

* Added new toggle - frame adv. - lag skip (menu item + hotkey mapping + saved in config), will cause frame adv. to skip frames where input is not read

* Added support for loading movies from archives (just like ROM files). Note: Movies loaded from an archive file will be read-only.

* movie replay dialog displays fractions of a second on movie length

* Savestates now save the Lagcounter information.

* added a mute turbo option in sound config

* add an option to pick a constant color to draw in place of BG when BG rendering is disabled (look for gNoBGFillColor in config).

Mappers

* remove cnrom chr rom size limit for homebrew roms

* mmc5 - 64KB WRAM games now work correctly

* mmc5 - use of chr A regs for BG in sprite 8x8 mode is fixed

* upgrade to cah4e3's latest mapper 163&164 code to fix a crash in a game

Debugging Tools

* Debugger - restore snap functionality

* Debugger - add FORBID breakpoints - regions which block breakpoints from happening if they contain the PC

* Debugger - debugger window is now resizeable

* nametable viewer will display correct NT,CHR,ATTR data in more cases (specifically, including some exotic mmc5 cases).

Lua

* Savestates remember Lua painting

* add memory.readbyterange to emulua

SDL only

* SDL: fixed --input(1-4) options. input1 and 2 are regular inputs, input3 and 4 are famicom expansion inputs

- * SDL fix configfile woes. configfile now goes to ~/.fceux/fceux.cfg
- * SDL: fixed segfault when opening .fcm files
- * SDL: Saner sound defaults for less choppy sound
- * SDL: "--special" option fixed for special video scaling filters
- * SDL: cleaned up the SConsruct
- * SDL: fixed issue where fceu would lock up when file dialogs were opened during fullscreen
- * SDL: fixed bug where fceux would close when file dialogs were closed
- * SDL: File open dialog is now used to movie playback
- * SDL: File open wrapper now takes a titlebar argument
- * SDL: Cleanup of usage
- * SDL: rename options --no8lim -> --nospritelim and --color -> --ntsccolor
- * SDL: Screenshots now always prepend the game name.
- * SDL: Changed default A/B from numpad 2 and 3 to j and k.
- * SDL: Enable frameskip by default

* SDL: Fixed a bug that would crash fceux if the emulation speed was overincreased

* SDL: New default hotkeys to more closely match win32 defaults

* SDL: Added lua script loading hotkey (f3). Non win32 SDL requires zenity for this to function.

* SDL: Build script cleanup; also added option for DEBUG builds.

2008

What's New? 2.0.1 (changelog)

What's New? 2.0.1

Released August 04, 2008

This was a maintenance release that fixes a few oversights in the 2.0.0 release.

* reorganize display toggle options in the menu

- * autofire fix (wasn't initializing to any autofire pattern from a fresh .cfg)
- * homebrew mmc5 games now have 64KB of exwram instead of only 8KB
- * fix crash related to player2 in lua scripts
- * fixed player2 in lua scripts

2008

What's New?

<u>What's New? 2.0.0</u>

Released August 02, 2008

FCEUX has all the latest tools, enhancements, and features from FCEU 0.28 rerecording and FCEUXDSP 1.07 In addition, it has many new tools, bug fixes, and enhancements not seen in previous branches.

<u>General</u>

-A detailed Help Menu! No longer are you aimlessly searching the internet for long lost info on FCEU's options!

-Numerous Dialog box reformats.

-FCEU remembers its last screen (x,y) position.

-Increased command line options

-More options under the Directory Override Menu

-A Turbo Toggle option (turbo now can be toggled on rather than having to hold the key down)

-More hotkey assignable options in the Map Hotkeys Menu.

-A lag counter

-Autofire uses the lag counter (so it will skip over lag frames)

Movie support

Overhauls in both the movie and savestate file formats.

.fm2 File format

The .fcm file format has been overhauled into a new .fm2 format. Changes include:

-Uncompressed and text based format. Movie editing can be done

simply in a text editor.

-Recording from soft reset option removed.

-Recording from start (hard reset) no longer has an empty savestate at the beginning.

-GUID inserted into movies for better savestate/loadstate error handling. -Rather than an Author field, it has a full metadata menu where an author can put any info needed.

-A tool to <u>convert .fcm</u> files to .fm2 files.

-More specific info on .fm2 files in the <u>.fm2 documentation</u>.

Savestate/Loadstate

-New savestate file format. NOTE: Savestates from previous FCEU versions CAN NOT be used in FCEUX.

-Fully functional error handling (savestates from other movies cannot be loaded).

-Read-only toggling related bugs fixed.

-Savestate filenames include the name of the movie (if a movie was playing when made). This prevents loading wrong savestates. (This also means that savestate 0 is different when a movie is playing and when it is not).

7z Archive Support

-ROMs in any 7z compatible compressed format can be opened directly. -If more than one valid ROM exists in an archive file, then a dialog box will open with a list of available ROM choices.

TAS Editor

-A brand new powerful movie making tool that revolutionizes the way TAS movies are made. See <u>TAS edit</u>.

New Tools

TAS Edit - a revolutionary new way of making TAS movies.

Input Presets - a system for quickly toggling different input configurations.

Tool Upgrades

Numerous enhancements have been made to various Tools/Options.

Memory Watch

-Resource management optimized so that memory watch now uses a minimal amount of CPU

-FCEUX remembers memory watch's last screen position (x,y)

-Tab-able Edit boxes

-Edit boxes now can hold 64 characters

-A Menu bar for all Memory watch functions

-Both "Save as" and "Save" options

-Hotkeys for New, Open, Save, Save As and Close

-A recent files Menu

-A "load on startup" option. If checked, memory watch will open automatically when FCEUX is opened

-A "load last file" option. If checked, memory watch will load the last file used

Cheat Search

-Now has a minimize button

-Cheat Search Menu from FCEUXDSP (a major overhaul compared to other FCEU branches)

-Possibilities update while playing/frame advancing a game

-Double clicking a value in the possibilities window sends the value directly to Memory Watch

RAM Filter

-Double clicking a value in the possibilities window sends the value directly to Memory Watch

Lua Scripting

-Uses the latest features of <u>Lua Scripting</u> from FCEU 0.28 -Many enhancements and new commands including dialog creation commands! Now scripts can create their own dialog's and GUI features.

Lua Basic Bot

-Basicbot removed (from the rerecording version of FCE Ultra). In its place is <u>lua bot.</u>

AVI Recording

-"Movie playback stopped" message recorded in AVI by default -Turbo Toggle Hotkey. (Allows turbo to be left on for a faster AVI capture).

2008

General

<u>General</u>

Guides for general uses of FCEUX.

Getting Started

A guide for loading games, setting up controls, etc.

Command Line Options

FCEUX as an extensive set of options for running from command line (or .bat file). This guide explains all command options available.

Famicom Disk System

A guide for playing Famicom (.fds) games.

Movie Recording

A guide for playing and recording movie input files (.fm2).

AVI Capturing

A guide for capturing a game/movie file into an AVI file.

2008

Getting Started

Getting Started Playing Games

The most basic function of FCEUX is to play Nintendo Entertainment System (NES) and Famicom Disk System (FDS) games.

To play a game, simply open a ROM by selecting "Open" in the File Menu (or press Ctrl+O). (See <u>Game Compatibility</u> for information regarding file types that are compatible with FCEU.)

To get set up properly, you may need to configure any of the following:

- -<u>Input</u>
- -<u>Video</u>
- -<u>Sound</u>
- -<u>Timing</u>
- -GUI settings
- -Hotkeys
- -Directory Overrides

Using Savestates

In emulation, a savestate (alternatively called freeze state or game freeze) is a snapshot of all of an emulated device's state information at a given moment. This makes it possible to pause emulation, and restart it later, even in another instance of the emulator, or to test the emulated machines reaction to different series of inputs using the saved state as a common starting point.

To make a savestate press shift + F1-F10 to save to a save slot (0-9). Or select a save slot with the number keys (0-9) and select the quick save command (Default hotkey is "I")

To load a savestate press F1-F10. Or select a save slot with the number keys (0-9) and loadstate by navigating to File > Savestate > Loadstate or by pressing the loadstate hotkey (Default hotkey is "P").

To save a state to a specific file, go to "Save state as..." in the FCEUX File menu.

To load a specific savestate file, go to the "Load state from..." in the FCEUX File menu.

Undo Savestate / Loadstate

If you load a state by accident, you can right-click and select "Undo Loadstate" to restore the emulator back to the state it was in before the loadstate. Upon using undo loadstate, a redo loadstate will appear as an option.

If you make a savestate, it will overwrite the existing savestate for that slot. You have the option to undo this and restore the previous savestate file by right-clicking and selecting undo savestate. Once you undo, you will have the option to redo savestate to restore the savestate that you made. You can also map a <u>hotkey</u> to this function.

2008

File Formats/Expansion Hardware Valid Game Types

FCEUX supports the iNES, FDS(raw and with a header), UNIF, and NSF file formats. FDS ROM images in the iNES format are not supported; it would be silly to do so and storing them in that format is nonsensical.

FCEUX supports loading ROM/disk images from some types of compressed files. FCEUX can load data from both PKZIP-format files and gzip-format files. Only the "deflate" algorithm is supported, but this is the most widely used algorithm for these formats.

Playing from compressed (.zip) files

FCEUX is compatible with all compression types compatible with 7z. Compatible types include .7z, .zip, .rar, and .tar.

If an archive file is opened, it will be scanned for the followings extensions: .nes, .fds, .nsf, .unf, .nez, .unif. If more than one valid type is detected, a dialog box will open up with a list of available choices.

Automatic IPS Patching (Playing Hacked Games)

FCEUX supports automatic IPS patching.

Place the IPS file in the same directory as the file to load, and name it [filename].ips.

Examples: Boat.nes - Boat.nes.ips Boat.zip - Boat.zip.ips Boat.nes.gz - Boat.nes.gz.ips Boat - Boat.ips (Some operating systems and environments will hide file extensions. Keep this in mind if you are having trouble.)

Patching is supported for all supported formats (iNES, FDS, UNIF, and NSF), but it will probably only be useful for the iNES and FDS formats. UNIF files can't be patched well with the IPS format because they are chunk-based with no fixed offsets.

2008

Command Line Options

Command Line Options

FCEUX offers numerous command line options. All commands are case sensitive.

ROM name

Plays specified ROM (ROM name must always be put last in command line arguments)

fceux path\rom.nes (or rom.zip)

fceux smb.nes
 fceux c:\fceux\roms\smb.zip

Play Movie File

Plays a specified movie (.fm2) file. A valid ROM must be specified or movie will not be played.

fcuex -playmovie path\movie.fm2 romname

fceux -playmovie smb.fm2 smb.nes

Read-only Status

Specifies whether a movie will be in "read-only" or "read & write" mode. (Note: a specified movie is not required to be used in conjunction with this command). 1 specifies read only status, 0 specifies read & write.

fceux -readonly flag

```
fceux -readonly 1
    fceux -readonly 0 -playmovie smb.fm2 smb.nes
```

fceux -readonly 1 -playmovie
c:\fceux\movies\smb.fm2 c:\fceux\roms\smb.nes

Stop Movie at frame x

Specifies that the movie will automatically stop at the specified frame. (A movie must be specified with -playmovie for this command to work)

fceux -playmovie path\movie.fm2 -stopframe framenumber romname

fceux -playmovie smb.fm2 -stopframe 10000 smb.nes

Load State

Specifies FCEUX to automatically load a specified savestate file. (Must have a specified ROM for this to work).

fceux -loadstate path\savestatefile romname

fceux -loadstate smb0.fc0 smb.nes
 fceux -playmovie smb.fm2 -readonly 0 -loadstate
smb0.fc0 smb.nes

Pause Movie After Playback

Sets the "<u>Pause Movie After Playback</u>" switch on/off. 1 sets it to enabled, 0 sets it to disabled.

fceux -stopmovie flag

fceux -stopmovie 1
 fceux -playmovie smb.fm2 -readonly 0 stopmovie 1 smb0.nes

Frame Display Toggle

Toggles whether or not the frame count will be displayed. 1 sets it to on, 0 sets it to off.

fceux -framedisplay flag

```
fceux -framedisplay 1
    fceux -playmovie smb2.fm2 -framedisplay 1
smb0.nes
```

Input Display Toggle

Toggles whether the movie input will be displayed. 1 sets it to on, 0 sets it to off.

fceux -inputdisplay flag

```
fceux -inputdisplay 1
  fceux -playmovie smb2.fm2 -inputdisplay 1
smb0.nes
```

Allow L+R/U+D

Sets whether or not the game will allow L+R/U+D input (see Input config). 1 enables L+R, 0 disables it.

fceux -allowUDLR flag

fceux -allowUDLR 1
 fceux -allowUDLR 0 smb0.nes

Enable Background Input

Sets the "<u>Enable Background Input</u>" switch on/off. 1 sets it to enabled, 0 sets it to disabled.

fceux -bginput flag

```
fceux -bginput 0
    fceux -playmovie smb.fm2 -readonly 0 -bginput 1
smb0.nes
```

Use Game Genie

Sets the Game Genie Flag (see <u>Toggle Switches</u>). 1 sets it to enabled, 0 sets it to disabled.

fceux -gg flag

fceux -gg 1
fceux -gg 1 smb0.nes

PAL Emulation

Sets the PAL Emulation Toggle (see <u>Toggle Switches</u>). 1 sets it to enabled, 0 sets it to disabled.

(note: FCEUX will assign PAL emulation automatically if a PAL ROM is loaded)

fceux -pal flag

fceux -pal 1
fceux -pal 1 smb0.nes

Movie Status Icon Toggle

Sets the Status Icon Toggle (see <u>Toggle Switches</u>). 1 turns off the status icon, 0 turns it on.

fceux -noicon flag

fceux -noicon 1 fceux -noicon 0 smb0.nes

Clip Left and Right Sides

Sets the Clip Left and Right Sides Toggle (see <u>Video Config</u>). 1 turns on clipping, 0 turns it off.

fceux -clipsides flag

fceux -clipsides 0 smb0.nes

Allow More than 8 Sprites per Scanline

Sets the 8 Sprites per scanline flag (see <u>Video Config</u>). 1 turns on extra sprites, 0 turns it off.

fceux -no8lim flag

fceux -no8lim 0 smb0.nes

Disable Speed Throttling

Sets the Disable Speed Throttling When Sound is Disabled flag (see <u>Timing Config</u>). 1 disables throttling, 0 leaves it on.

fceux -nothrottle flag

fceux -nothrottle 0 smb0.nes

Turbo Toggle

Sets the Turbo Toggle. 1 Sets Turbo on, 0 leaves it off.

fceux -turbo flag

fceux -turbo 1 smb0.nes

Load Config File

Loads a specified config file rather than the default fceux.cfg

Warning: the config file must be in the base directory. A pathname can NOT be specified in the filename

fceux -cfg filename

```
fceux -cfg fceux-smbconfig.cfg smb.nes
```

Load Lua Script Loads a Lua script on startup.

fceux -lua filename

fceux -lua memwatch.lua

2008

Customizing through the Config File

Customizing through the Config File

There are some options that can only be done by directly editing the config (fceux.cfg) file. All of those options are documented here.

The .cfg file is a text file and can be opened by any text editor (just as wordpad).

Emulator background Color when Graphics Background is disabled

gNoBGFillColor

When you disable the backgrounds (Config > Display > Graphics: GB), the default color is black. You can change that color by modifying this value. By default it is 255 (black).

Hex Editor

HexRowHeightBorder 0

This value determines the number of pixels between each row of values in the Hex Editor. By default it is 0.

HexBackColorR 255 HexBackColorG 255 HexBackColorB 255

HexForeColorR 0 HexForeColorG 0 HexForeColorB 0

HexFreezeColorR 0

HexFreezeColorG 0 HexFreezeColorB 255

These values allows are the Hex Editor color scheme values (RGB). The background color is 255,255,255 (white) by default. The foreground color (text) is 0,0,0 (black) by default. When an address is frozen it is 0,0,255 (blue) by default.

2008

Famicom Disk Sytem

Famicom Disk System

The Family Computer Disk System (FDS) was released in 1986 by Nintendo as a peripheral for the Family Computer ("Famicom") console in Japan. It was a unit that used proprietary floppy disks for data storage. It was announced, but never released, for the North American Nintendo Entertainment System.

The device was connected to the Famicom deck by plugging a modified cartridge known as the RAM Adapter into the system's cartridge port, which attached via a supplied cable to the disk drive.

The floppy disks used were double-sided, with a capacity of 64 kilobytes per side. Many games spanned both sides of a disk, requiring the user to switch sides at some point during gameplay. A few games used two full disks (four sides). The Famicom Disk System was capable of running on six C-cell batteries or the supplied AC adapter. The battery option was included due to the likelihood of a standard set of AC sockets already being occupied by a Famicom and a television.

FDS BIOS ROM

In order to play any Famicom (.fds) game, you will need the FDS BIOS ROM image and it must be named disksys.rom.

It must be in the base FCEU directory unless you specified a path to disksys.rom in the <u>Directory Overrides List</u>. FCEUX will not load FDS games without this file.

File types

Two types of FDS disk images are supported: disk images with the FWNES-style header, and disk images with no header. The number of sides on headerless disk images is calculated by the total file size, so don't put extraneous data at the end of the file.

Writing to disk image

If a loaded disk image is written to during emulation, FCEUX will store the modified disk image in the save games directory, which is "sav" under the base directory by default (unless changed under the <u>Directory Overrides</u> <u>List</u>).

Eject/Insert Disk

Emulates the ejecting of the current disk or the inserting of a new disk. If a disk image is loaded, this command will eject it. If a disk is ejected, this will insert a new disk.

This command can be mapped to a keyboard/joypad button in the <u>Map</u> <u>Hotkeys Menu</u>.

Switch Disk Side

If/when prompted by the game, you can emulate the Switching sides of the FDS disk with the NES > Switch Disk Side command. This command can be mapped to a keyboard/joypad button in the <u>Map</u><u>Hotkeys Menu</u>.

2008

AVI Capturing

Video & Audio Capturing

FCEU allows for outputting Video/Audio into .avi files or capturing audio only into .wav files. This can be used to capture one's playing or for dumping movie files (.fm2) to .avi files.

Capturing a Movie File (.fm2) to Video/Audio (AVI)

-Pause the emulator by navigating to NES > Emulation Speed > pause or pressing the pause hotkey (the pause key by default).

-For a faster capture increase emulation speed (you can capture at any emulation speed and FCEUX will still output a 60 (or 50) fps video file). -Select "Replay Movie" from the File > Movie Menu and select the movie file

-If you intend to capture beyond the final frame of the movie file, make sure "Pause after Playback" (Config Menu) is not checked.

-Select "Record AVI" in the File > AVI/Wav menu.

-Select a file location and the video codec you wish to use.

-Unpause the emulator.

-When capturing is complete, pause the emulator and select "Sop AVI" in the File Menu.

Capture Audio only

To capture audio only, navigate to File > AVI/Wav > Record WAV. Pick a filename and destination for FCEUX to begin capturing the audio to a .wav file (raw .pcm). To stop WAV recording, select File > AVI/Wav > Stop WAV.

Movie Recording

Introduction

A movie file is a file which contains data needed to reconstruct actions in a game. In most emulators, the movie files consist of simply the buttons that were pressed during the game. Because the emulation is completely predictable (deterministic), it will always play back the same way.

Unless the movie starts from the console power-on or from reset, the movie file might also contain a savestate that loads the beginning point of the game. Movie files don't contain any sound or image data. Such data is not needed, because the emulator can reconstruct it during movie playback.

Movie files in FCEUX are .fm2 files. The file format is unique to FCEUX and not compatible with other movie recording versions of FCE Ultra. Movie files from other versions (.fcm) can be converted to .fm2 for playback with the <u>.fcm to .fm2 converter</u>.

Movie features in FCEUX are designed specifically for making Toolassisted Speedruns. For more information visit <u>TASVideos</u>.

Recording Movies

To record a movie, open a ROM. Then simply select "Record Movie" in the File > Movie Menu. You will be prompted to name the file and to select where to record from. Selecting "Start" will begin the recording from a Power-on (Hard Reset). If you select "Now", a savestate will be made at your current location in the game, and the movie will begin recording from there. If you select browse, you will be prompted to find a preexisting savestate file to begin recording from.

Savestates, Slowdown, and Frame Advance

At anytime while recording, you can make a *savestate. This is a snapshot of the game's current memory contents. Once a savestate is made, it can be loaded with the *loadstate command. This will return the movie back to the spot in the game where the savestate was made. This can be used to undo mistakes or to test different strategies for a particular segment.

(The default key for making a savestate is "I" and the default key for loading a state is "P". Both of these can be assigned under the <u>Map</u> <u>Hotkeys Menu</u>). Both can also be access through the File > Savestate Menu

Tool Assisted movies take advantage of slowing the emulator down in order to increase precision of the movie making process. Navigating to NES > Emulation Speed > Slow down or pressing the "-" key will slow down emulation. NES > Emulation Speed > Speed up or the "=" will speed it up. (These can be re-mapped in the <u>Map Hotkeys Menu</u>).

Even greater precision can be made using the frame advance key. Pressing the frame advance key will pause emulation and advance it a single frame (1/60th of a second NTSC). By holding down input and pressing the frame advance key, it will record that input for that particular frame.

For more info seeing <u>Tool Assisted Speedruns</u>.

"Bullet Proof Rerecording"

All savestates made during movie recording contain the movie information up to the frame of the savestate. When a savestate is loaded, the movie file in the savestate is also loaded. This is referred to as "Bullet Proof Rerecording" because it prevents possible desyncs and lost data from improper/out of order savestate loading.

Playing Back Movies

To play back a recorded movie, open the ROM. Then select "Replay Movie" in the File Menu. A movie dialog box will open where you can select the movie file.

You can also select whether the movie is in Read-only mode. If a movie is in read-only mode, the movie file can not be altered in any way. If you make a savestate while playing the movie and load that state, the playback will simply "rewind" to that state. If the movie is not in readonly, however, loading a state will set the movie to record mode and begin recording from that savestate.

You can also select "Pause movie at frame" x. If selected, the movie will automatically pause when reaching the frame selected (the default is the last frame of the movie).

Read only

You can select read-only when playing a movie. You can also toggle the read-only status by navigating to File > Movie > Read only. In read-only mode a movie can not be edited. Loading a savestate will take the movie to that point in the movie and stay in playback mode.

In read-write status, loading a state will change a movie from playback mode to record mode.

Resuming Recording

You can resume recording a previous movie by playing back the movie, setting the record status to read+write, and then loading a state.

Play Movie from Beginning

At any point while recording or playing back a movie, you can navigate to File > Movie > Play Movie from Beginning. This will set the movie to read only status and reset playback to frame 0.

Frame Counter

The Frame counter displays what frame the movie is currently on. If the movie is playing in read-only mode, it will also display the total number of frames in the movie. The default key for toggling the Frame Counter display is the "." (period) key. (This can be re-mapped in the Map Hotkeys Menu).

Frame Advance

The frame advance key ("backlash" key by default. Re-mappable under the <u>Map Hotkeys Menu</u>) will advance the game by a single frame and then pause the game. If the hotkey is held down, it will auto advance quickly through the game.

This is a critical tool when perfecting input in movie recording.

Metadata

When you record a new movie via the record movie dialog there is an author field. This sends the info to the .fm2 file in the form of comment Author [author name] (see .fm2).

Any line in the .fm2 that starts with "comment" is known as metadata. You can include any number of comments manually by editing the .fm2 file with any text editor.

On the replay movie dialog, clicking the metadata button will display all

metadata in a separate dialog box (If a movie is currently loaded you can also access the meta-data by right-clicking and selecting Metadata in the context menu).

Subtitles

FCEUX now supports subtitles in the .fm2 file format. Subtitles will be displayed on the screen automatically as a movie plays. You can turn on/off subtitles by navigating to Config > Movie Options > Display movie subtitles (see <u>Movie options</u>).

For adding subtitles to a movie see the <u>.fm2 documentation</u>.

2008

NES

<u>NES</u>

Explains commands in the NES menu of FCEUX. All these commands can be mapped to a keyboard/joypad button in the <u>Map Hotkeys Menu</u>.

Reset

Emulates the pressing of the Reset Button on the NES. ("Soft reset").

Power

Emulates a power cycle, which is turning the power on and off with the Power button on the NES. ("Hard Reset")

Eject/Insert Disk

Ejects or inserts a FDS disk. (See Famicom Disk System)

Switch Disk Side

Switches Sides of a FDS disk. (See Famicom Disk System)

Insert Coin

Emulates the inserting of a coin in an arcade-style game.

Emulation Speed Sub Menu

Speed Up Speeds up emulation (emulation speed ranges from 1% to 6400%)

Slow Down Slows down emulation

Slowest Speed

Sets emulation to 1% speed

Normal Speed

Sets emulation speed to 100%

Turbo

Toggles turbo mode. In turbo mode, emulation is set its fastest settings.

2008

Palette Options

Palette Options

FCEUX comes packaged with several palette files. This page describes details for each one.

To load a palette file, see <u>Palette config</u>.

FCEUX.pal

This is the default palette that FCEUX uses. It is the same palette used in FCEU.12 or earlier, and FCEUD/FCEUXD,FCEUXDSP.

FCEU-13-default_nitsuja.pal

This is the palette added to FCEU.13 rerecording by Nitsuja.

FCEU-15-nitsuja-new.pal

This is the palette added to FCEU.15 rerecording by Nitsuja. It is a slight adjustment to the FCEU.13 palette. This one most closely resembles the default palette of Nestopia.

ASQ_realityA.pal & ASQ_realityB.pal

BMF palettes had some flaws. AspiringSquire tweaked BMF's palettes and came up with this. They fix issues mostly related to brightness.

BMF_final2.pal & BMF_final3.pal

These palettes were designed by BMF. He customized these by looking at snapshots of his television screen and attempting to replicate them as close as possible.

Config

The Config Menu

These chapters explain options under FCEUX's Config menu.

Menu Items & Sub-menus

Explains the basic menu items and sub-menu items in the top of the menu.

<u>GUI</u>

Settings related to the FCEU emulator window.

Directories

Sets Directory override assignments.

<u>Input</u>

Assigns keys/joypad buttons to emulated controllers.

Network Play

Various settings related to playing over the internet.

Palette

Palette options.

<u>Sound</u>

Sets sound configuration options.

Timing

Settings related to emulation timing.

<u>Video</u>

Sets video & graphics configuration options.

Movie Options

Sets options related to playing/recording movie files

Map Hotkeys

Sets Hotkey assignments.

2008

Toggle Switches (Hide Menu, etc)

Config Toggle Switches

Explains the various toggle switch commands in the top two groups of commands under the Config Menu.

Hide Menu

Hides the Menu commands on the FCEUX main window. Pres ESC to unhide the menu.

Game Genie

Allows the use of the game genie ROM. You must have a game genie ROM named gg.rom (it is save to rename a game genie.nes file to gg.rom) and it must be in the FCEUX base directory (which is the folder fceux.exe is in unless you specified a different folder in the <u>Directory</u> <u>Override Menu</u>).

If enabled, FCEUX will open gg.rom first when you load a new game. Any codes applied in the game genie screen will be applied to the game just like on a real NES.

(Remember that enabling/disabling Game Genie emulation will not take effect until a new game is loaded)

Note: Game genie codes can also be added with the <u>Game Genie</u> <u>Encoder/Decoder</u> via the <u>Cheat Search Menu</u> (and does not require a game genie ROM).

PAL Emulation

Toggles between NTSC (60fps) and PAL (50fps) frame rates. By default,

FCEUX will detect the proper choice when loading a ROM and set the flag accordingly.

PPU (Sub-menu)

New PPU / Old PPU

As of FCEUX 2.1.2, FCEUX has a new PPU core. The new PPU has improved accuracy and greater game compatibility than the previous PPU. However, some games may not work properly and there will be slight timing differences that would hurt movie compatibility. Therefore, FCEUX will still support the old PPU. Currently, Old PPU is the default setting.

Enable (Sub-menu)

Run in Background

If enabled, FCEUX will continue to emulate when the window is not in focus. If disabled, the emulator will pause when out of focus.

Background Input

If enabled, FCEUX can continue to receive input while not in focus. (Useful for playing 2 FCEUX's simultaneously)

Auto-savestates

Enables the Auto-save feature. If enabled, FCEUX will make periodic savestates as you play (or play/record a movie). You can right-click and select the "load last auto-save" in the context menu or press the Auto-save <u>hotkey</u> to back up to the last auto-save savestate.

Frame Adv. - Skip Lag

This feature, if enabled, will cause the frame advance key (see <u>movie</u> <u>recording</u>) to skip over lag frames. It does this by reading the lag counter (see <u>display</u>) and skipping past any frames where the input is not polled.

For instance, in a 30fps game (such as double dragon), frame advance will advance 2 frames instead of 1.

Backup Savestates

Enabled by default. This option allows for savestate & loadstate Undo (& redo). (see <u>context menu</u>)

Compression Savestates

Enabled by default. This option compresses non movie savestates.

Display (Sub-Menu)

Input Display

The input display will display 1-4 pictures of a NES controller at the bottom of the screen. When playing/recording a movie, these controllers will display the input that is captured in the file.

When input comes from a movie file rather than then user, it is displayed in a different color (silver)

The input display can also be toggled by hotkey/ The default key for toggling the Input display is the "," (comma) key. (This can be re-mapped

in the Map Hotkeys Menu).

Lag Counter

The lag counter will increment every time to the game fails to poll for user input. It will display in red on any frame that is currently lagging and will increment the lag counter by 1. These situations occur when the game is lagging (too much information to process), or the game is in a screen transition state (so not polling for user input). In a 30fps game (such as Double Dragon 2), it will increment every other frame.

The lag counter value is stored in savestates.

Displaying the lag counter can be done by hotkey as well. It must be mapped to a hotkey in the <u>Map Hokeys Menu</u>. It can be reset to 0 by mapping the reset lag counter hokey in the <u>Map Hotkeys Menu</u>.

Frame Counter

Toggles the display of the frame counter. The frame counter will increment once per frame.

Display BG

Turning this off will turn off the backgrounds in the game.

Display OBJ

Turning this off will turn off the objects in the game.

Note: You can set the default color when the Backgrounds are turned off. To do so, open fceux.cfg and change the value of the entry named: gNoBGFillColor

Save Config File

Saves current settings to fceux.cfg. Normally settings are not saved until FCEUX is closed.

2008

Directories

Directory Overrides

This menu sets a default directory override for various files relating to FCEU.

Base Directory

sets the default directory FCEU will use. It will be the folder that FCEU creates all the sub folders (unless they are also overridden).

ROMS

where FCEU will look for ROMS by default. (What folder will appear when selecting the Files > Open...)

Battery Saves

where .sav files will stored and opened from. These files contain the battery backed SRAM used in some games (such as Dragon Warrior).

Save States

where .fcs (savestate) files will be stored.

FDS BIOS ROM

where FCEU can find disksys.rom. disksys.rom is a required file in order to load FDS (Famicom Disk System) games. If not specified, FCEUX will default to the base directory.

Screenshots

where screen captures (.png) files will be saved.

Save Screenshots as "<filebase>-<x>.png"

sets how the .png files will be named. Left unchecked, the file names will simply be 0.png, 1.png etc. Checked adds the ROM name into the file as well (such as Double Dragon 2 (U)-0.png)

Cheats

where .cht files will be stored. .cht files store the active cheats set up in <u>Cheat Search</u>.

Movies

where .fm2 files will be saved/loaded. These files are the input files used in movie recording.

Memory Watch

where memory watch files are saved/loaded. These are used by <u>memory</u> <u>watch</u>.

Input Presets

where input presets will be saved/loaded. These are used in the presets section on the input config window.

Lua Scripts

where Lua scripts will be saved/loaded. These are used when using the Lua Scripting tool.

AVI Output

overrides which directory FCEUX will default to when saving a .avi file.

2008

GUI

<u>GUI</u>

Various toggle boxes related to the FCEUX main window.

Load "File Open" dialog when FCEUX starts.

If enabled, FCEUX will ask for a ROM to open upon FCEUX start up.

Automatically hide menu on game load.

If enabled, The FCEU Menu will be hidden while a ROM is loaded. To unhide it, press the ESC key.

Ask confirmation on exit attempt.

If enabled, FCEUX will ask you before closing the window. (It may also say some other things...)

Disable screen saver while game is loaded.

This is enabled by default. If a game is running, the windows screen saver will not turn on.

Enable right-click context menu.

This is enabled by default. This allows you to right-click on the emulator to get context menus. The context menu gives many common options for a given situation and has a few options not available otherwise.

Input

Input Configuration

Setting up controllers

On the pull down menus, you can select the device you want to be emulated on input ports 1 and 2 (game pad, zapper, pad, paddle). If you check the box labeled "Attach four-score(implies four gamepads)", you won't be able to select any of these options, because the four-score allowed someone to use 2 extra controllers.

The device currently being emulated on each port is listed above the drop down list; loading certain games will override your settings, but only temporarily.

To bind these controls to specific keys/joystick controls use the "configure" the device listed above each drop-down list.

Zapper / Arkanoid Paddle

Most Zapper NES games expect the Zapper to be plugged into port 2. and most VS Unisystem games expect the Zapper to be plugged into port 1.

The left mouse button is the emulated trigger button for the Zapper. The right mouse button is also emulated as the trigger, but as long as you have the right mouse button held down, no color detection will take place, which is effectively like pulling the trigger while the Zapper is pointed away from the television screen. Note that you must hold the right button down for a short time to have the desired effect.

The Arkanoid Paddle emulates the same way the zapper.

Power Pad A / B

Emulates the NES Power pad. The 12 pad buttons can be routed via the configure button. FCEUX allows up to 2 Power Pads to be emulated at once (Power Pad A and B).

Famicom Controllers

You can also select the input device to be emulated on the Famicom Expansion port. If you select a device for the Famicom Expansion Port, you should probably have emulated game pads on the emulated NES-style input ports.

In addition to the traditional famicom controller, FCEUX can emulate the Famicom version of the Arkanoid controller, the "Space Shadow" gun, the Famicom 4-player adapter, the Family Keyboard, the HyperShot controller, the Mahjong controller, the Oeka Kids tablet, the Quiz King buzzers, the Family Trainer, and the Barcode World barcode reader.

Replace Port 2 Start With Microphone

Checking this box will replace the Start button used by controller 2 with the microphone option found on the famicom. Pressing the Microphone button is like blowing or yelling into it on the console equipment. The Port 2 controller used for the Famicom included a microphone and a volume control in place of the Start and Select buttons. This option isn't automatically detected, so it has to be manually enabled by the user. Movie files may also enable and use this feature. Both Famicom Cartridges and Famicom Disks have made use of this feature, such as both the cartridge and disk version of Zelda 1, Hikari Shinwa, and Takeshi no Chosenjo. Games other than those listed here use this feature.

Input Presets

This feature allow you to set the current input configuration to one of three presets. This gives you the option to quickly change from one input configuration to another (such as toggling between 1 or 2 controllers

and/or toggling from controller 2 being bound to controller 1 or having its own controls).

To assign the current input configuration to a preset press the down arrow next to one of the presets. To assign the preset as the current input configuration press the up arrow or use the hotkey assigned to that specific preset. Preset hotkeys can be assigned in the <u>Map Hotkeys</u> menu.

Disable left+right/up+down

By default FCEUX allows you to press both the left and right controls at the same time (or up and down). To disable this feature uncheck the checkbox on the left.

Auto-Hold

Clicking the auto hold button will allow you to assign a hotkey to the autohold feature.

Clicking the clear button will allow you to assign a hotkey to the clear auto-holds feature.

To use this feature, close the input config window and return to the FCEUX main window. Hold down the auto-hold hotkey and press one of your controller inputs. This will add it as one of the auto-hold assignments. The game will keep auto-hold assigned buttons held be default. Pressing one of these keys will release the button for the duration that it is held.

To turn off all auto-hold assignments press the clear auto-holds hotkey.

2008

Network Play

Network Play

Allows you to play against a human opponent over the internet. Requires the use of FCEU server.

Currently however, FCEU Server runs very poorly and is hardly useable. This issue will be resolved in a future release.

2008

Palette

<u>Palette</u>

Settings related to the emulator's color palette choices.

See

NES Palette

Load Palette

Allows you to load a custom color palette (.pal) file to use for the current game loaded.

A note on on the format of external palettes; Palette files are expected to contain 64 8-bit RGB triplets(each in that order; red comes first in the triplet in the file, then green, then blue). Each 8-bit value represents brightness for that particular color. 0 is minimum, 255 is maximum.

Palettes can be set on a per-game basis. To do this, put a palette file in the same directory the game is in, and add the extension "pal". Examples:

| File name: | Palette file name: |
|-------------------|--------------------|
| BigBad.nes | BigBad.pal |
| BigBad.zip | BigBad.pal |
| BigBad.Better.nes | BigBad.Better.pal |

With so many ways to choose a palette, figuring out which one will be active may be difficult. Here's a list of what palettes will be used, in order from highest priority to least priority(if a condition doesn't exist for a higher priority palette, the emulator will continue down its list of palettes).

* NSF Palette(for NSFs only)

* Palette loaded from the "gameinfo" directory.

* NTSC Color Emulation(only for NTSC NES games).

* VS Unisystem palette(if the game is a VS Unisystem game and a palette is available).

* Custom global palette.

* Default NES palette.

Restore to Default Palette

Restores the color palette to its default settings.

NTSC Color Emulation

If enabled, FCEUX will simulate actual NTSC signal processing. The result should be the actual colors you would see if outputting to an actual NTSC television.

The Tint and Hue knobs can be used to make adjustments to the resulting color change.

2008

Sound

Sound Configuration

where you can configure sound

Input/Output format

The sound enabled/disabled checkbox will turn on/off FCEUX's sound.

The force 8-bit sound checkbox will override the current sound configuration and use 8-bit sound instead.

You can select the sound quality in the sound quality pull down menu.

Rate sets the audio sample rate.

Mute frame advance

If checked, no sound will be produce when frame advance is pressed.

Mute Turbo

If checked, the sound processing will be bypassed when emulation is in turbo mode

Buffering

On older machines, increased buffering may be necessary. If the sound is glitchy or crackling, increasing the buffing time may resolve the issue. Lower buffering settings can reduce sound latency.

Volume Control

Sets the sound volume of the master sound or individual sound channels.

Master

Sets the Master volume level. You can also set volume levels using the sound volume up, volume down, mute, and volume normal hotkeys under map <u>hotkeys menu</u>.

Triangle/Square 1/Square 2/Noise/PCM

Sets the volume to each individual sound channel.

Note: When using low quality sound, the amount of channel control is greatly limited. Some sound channels are disabled.

Restore Defaults

Restores the master and individual sound channel volumes to their default location.

2008

Timing

<u>Timings</u>

Settings related to emulation timing.

Disable Speed Throttling Used When Sound is Disabled

If checked, speed throttling will not be used while sound is disabled. (Speed throttling gives a performance boost while sound is off).

Set High Priority Thread

Sets processing priority. Enabling can help slower computers keep a steady 60fps (or 50fps) framerate.

2008

Video

Video Configuration

This window sets various graphics emulation options.

Full Screen Settings

Full Screen If checked, FCEUX will enter full screen mode when it is loaded.

Enter Full Screen Mode after file is loaded If checked, FCEUX will only enter full screen mode when a game is loaded.

Video Mode:

Sets the image size while in full mode. If custom is selected, FCEUX will use the settings under "Custom Video Mode".

Sync Method:

If the emulator is running poorly, trying out these sync options can help make it run smoother.

Disable Hardware Acceleration If full screen is causing problems, checking this might fix it.

Custom Video Mode

Mode:

Sets the image size during full screen mode, if custom is selected under Full Screen Settings.

Image Size Transform

Special Scaler

Within this box is four options: Hq2x, Scale2x, Hq3x, and Scale3x. - Scale2x/3x just attempts to render out the corners of the pixels to make them look a bit rounder. "2x" means two times bigger than 1x1 and "3x" means three times bigger than 1x1.

- Hq2x/3x does a much better job than scale2x/3x by smearing the pixels together with a slight blur. However, Hq2x/3x requires a faster computer for decent speed (at least 1 GHz and above). "2x" means two times bigger than 1x1 and "3x" means three times bigger than 1x1.

Scale Dimensions by:

Takes the image size and multiplies the x and y by a specific amount.

Stretch to Fill Screen

Stretches the image to fill the screen during full screen mode. (may not be the best choice depending on your screen size).

Windowed Settings

Size Multiplier

Takes the image size and multiples the x and y by a specific amount. You can also set these by clicking and dragging the FCEUX window.

Force Integral Scaling Factors

If checked, FCEUX can only be stretched by even amounts (1x, 2x, 3x, etc.). If unchecked, it can be stretched by any amount.

Force Aspect Ratio Correction

If Integral Scaling Factors is unchecked, checking this will only allow the correct aspect ratio while stretching the window.

Current Pixel Aspect Ratio Displays the current Aspect Ratio

Sync Method

If the emulator is running poorly, trying out these sync options can help make it run smoother.

Disable Hardware Acceleration

This is checked by default. Slower computers my experience slow emulation and should uncheck this option. The video will be blurred somewhat with this option checked.

Drawing Area

First Line

Sets the first scan line for NTSC and PAL Modes. This should be left on the default of 8.

Last Line

Sets the last scan line for NTSC and PAL Modes. This should be left on the default of 231.

Clip Left and Right Sides (8 columns each) If enabled, 8 pixels from each side of the windows will be removed. Who knows why you would want to!

Emulation

Allow More than 8 sprites per scanline.

On real NES hardware, more than 8 sprites on the screen causes flickering. Enabling this can cause less flickering by allowing more sprites to be visible at once. If you prefer to say "true" to NES hardware, this should not be checked.

2008

Movie Options

Movie Options

The movie option dialog has various settings related to movie making.

Pause After Movie Playback

If checked, FCEUX will automatically pause emulation when reaching the last frame of a movie file. If disabled, the movie will end, and emulation will continue without a movie loaded.

Show Movie Status Icon

The status Icon shows the "play" "record" and "pause" icons in the lower right corner. The Show Status Icon checkbox toggles these on/off.

Bind savestates to movies

Affects the savestate naming system when a movie is loaded. If checked, the movie name will be appended to a savestate filename.

Display movie subtitles

Toggles whether or not movie subtitles (imbedded into the .fm2 file, see .<u>fm2</u> documentation) will be displayed on screen.

Put movie subtitles in AVI

Toggles whether or not movie subtitles will be recorded into a .avi file.

Automatically backup movies

If checked, the auto-movie backup is toggled on. Whenever a movie is loaded then set into record mode (by loading a savestate while in read-write mode), a backup copy of the .fm2 is saved before changing the file.

Movie backups will be created only once each time a movie is loaded into FCEUX. Movie backups are appended with a backup number and the .bak file extension.

2008

Map Hotkeys

Map Hotkeys

The map hotkeys dialog allows you to assign hotkeys to various FCEUX commands.

To assign or remove a hotkey assignment, double click on the name of the hotkey in the list box. Then press the key combination you wish to assign it. To clear the assignment, press the clear button.

The filter pull down menu allows you to only see hotkey listings in various categories (the list shows all hotkey assignments by default).

The Restore defaults button will change all hotkeys to their default values.

2008

Context Menu Items

Context Menu

FCEUX includes a context menu that allows commonly used menu functions for various situations. There are some functions that appear only here.

This page describes all the possible menu items in each possible context situation.

No game loaded.

Appears when no game is loaded.

Open ROM Same as the File > Open ROM option

Last ROM used Opens the most recently used file from the Recent Files Menu

Help Brings up the Getting Started chapter in the help document.

Game Loaded

Appears when a game is loaded, but not a movie (.fm2).

Play Movie... Same as the File > Movie > Play Movie menu item.

Record Movie... Same as the File > Movie > Record Movie menu item.

Undo savestate

If this option is enabled it means the last savestate saved over-wrote a previous savestate file. This option restores the previous savestate file.

Redo savestate

If this option is in the menu, it means that Undo savestate was recently used to restore a previous savestate. This reverts that change.

Rewind to last auto-save

Auto-save must be enabled for this menu item to be accessible. Same as the Load last auto-save Hotkey Item. It loads the last auto-savestate. Auto-savestates are created once about every 4 seconds, so this typically has the effect of rewinding emulation.

Screenshot Same as File > Screenshot.

Close ROM Same as File > Close

Movie loaded - Read-only

Appears when a movie is loaded and Read-only mode is set.

Toggle to read+write Sets Read status to Read+Write.

Play Movie from Beginning Same as File > Movie > Play from Beginning. Turns Read status to Read-Only and plays the movie from frame 1.

Stop Movie Replay Same as File > Movie > Stop Movie.

View comments and subtitles

Opens up the Metadata dialog. Same as the Metadata button on the Play movie dialog.

Undo savestate

If this option is enabled it means the last savestate saved over-wrote a previous savestate file. This option restores the previous savestate file.

Redo savestate

If this option is in the menu, it means that Undo savestate was recently used to restore a previous savestate. This reverts that change.

Rewind to last auto-save

Auto-save must be enabled for this menu item to be accessible. Same as the Load last auto-save Hotkey Item. It loads the last auto-savestate. Auto-savestates are created once about every 4 seconds, so this typically has the effect of rewinding emulation.

Help

Opens the Movie recording chapter of the help document.

Movie loaded - Read + Write

Toggle to Read-only Sets Read status to Read-Only.

Play Movie From Beginning Same as File > Movie > Play from Beginning. Turns Read status to Read-Only and plays the movie from frame 1.

Stop Movie Recording Same as File > Movie > Stop Movie.

View comments and subtitles Opens up the Metadata dialog. Same as the Metadata button on the Play movie dialog.

Make backup

Generates a backup .fm2. Uses the same file naming system as the auto-movie backup. (See <u>movie options</u> for details).

Undo savestate

If this option is enabled it means the last savestate saved over-wrote a previous savestate file. This option restores the previous savestate file.

Redo savestate

If this option is in the menu, it means that Undo savestate was recently used to restore a previous savestate. This reverts that change.

Undo loadstate

If this option is enabled it was because the Loadstate function was called sometime while the game was loaded. This function restores the game state to where it was before loadstate was called.

Redo loadstate

If Undo loadstate was called, this option is available. It reverts the change and restores the game back to the point it was when loadstate was called.

Help

Opens the Movie recording chapter of the help document.

Additional items may also appear related to these situations:

Lua

Load last Lua

If there is at least 1 filename in the Recent Lua Files menu this calls the most recently used Lua script file. Has the same effect as the File > Lua > Reload Lua Script menu item.

Stop Lua

If a Lua script is currently loaded this option is available. Same as File > Lua > Stop Lua Script.

Hide Menu

Unhide menu If the main FCEUX menu is hidden this option is available. Restores the main menu.

Subtitles

If a movie is loaded and has subtitles, a toggle subtitles option will be in the menu a Dump to SRT file option will be available. This dumps the subtitles to a standard subtitle file compatible with A/V containers such as .mkv

2008

Tools

<u>Tools</u>

Guides for the specific tools and settings under FCEUX's Tools menu.

Cheat Search

A guide to using the cheat search tool.

Memory Watch

A guide to using the Memory Watch tool.

RAM Filter

A guide to using the RAM filter tool.

TAS Edit

A new tool designed for making TAS movies.

Convert fcm

A tool that will convert .fcm movie files to the .fm2 file format.

Auto Fire settings

A guide for setting auto-fire, auto-fire offset, and alternate A and B options.

Use External Input

Explanation of Use External Input

Text Hooker

A guide for using the text hooking tool.

2008

FCE Ultra Cheat Guide Introduction

FCE Ultra allows cheating by the periodic "patching" of arbitrary addresses in the 6502's memory space with arbitrary values, as well as read substitution. "Read substitution" is the method that would be used on a real NES/Famicom, such as done by the Game Genie and Pro Action Replay. It is required to support GG and PAR codes, but since it is relatively slow when done in emulation, it is not the preferred method when a RAM patch will suffice. Also, in FCE Ultra, read substitution will not work properly with zero-page addressing modes(instructions that operate on RAM at \$0000 through \$00FF).

The RAM patches are all applied a short time before the emulated vertical blanking period. This detail shouldn't concern most people, though. However, this does mean that cheating with games that use bank-switched RAM may be problematic. Fortunately, such games are not very common (in relation to the total number of NES and Famicom games).

The cheat search comes with its own set of tools for finding addresses in memory to use for making cheats (or for monitoring the addresses in the <u>memory watch window</u>)

Cheat Files

By default cheat files (.cht) are stored in the "cheats" subdirectory under the base FCEUX. The files are in a simple plain-text format. Each line represents a one-byte memory patch. The format is as follows(text in brackets [] represents optional parameters):

[S][C][:]Address(hex):Value(hex):[Compare value:]Description

Example:

040e:05:Infinite super power.

A colon(:) near the beginning of the line is used to disable the cheat. "S" denotes a cheat that is a read-substitute-style cheat(such as with Game Genie cheats), and a "C" denotes that the cheat has a compare value.

Note: When a game is loaded, FCEUX will load any accompanying saved .cht file automatically.

The default .cht file folder can be changed with the <u>Directory Override</u> <u>menu</u>.

The Cheat Search Menu

The cheat search interface consists of several components: a list of addresses and associated data for a search, several command buttons, and the search parameters.

All addresses listed in the cheat search windows are in unsigned 16-bit hexadecimal format and all values in these windows are in an unsigned 8-bit decimal format(the range for values is 0 through 255).

Active Cheats

The Active cheats window on the left contains the list of cheats for the currently loaded game. Existing cheats can be selected, edited, and updated using the "Update" button.

Each entry in the list is in the format of: * Address = Value

The address is the location in the 6502's address space. The * denotes that the current cheat is active (double clicking will toggle on/off). Value is the value (in hex) that the addresses has been locked into.

You can add, delete, and update cheats in the active cheats window with the boxes below.

To find an address use the cheat search portion of the window.

There is also a right-click menu with the options Toggle Cheat, Poke Cheat Value, and Goto In Hex Editor.

Toggle Cheat is like Double Clicking. It enables or disables the cheat code.

Poke Cheat Value is like turning the cheat on, but in this case there's no off switch. If the code is on when you use this, then when the code is turned off, it will revert to the value last used. Good for one time life refills, if you want that sort of thing.

Goto in Hex Editor opens the Hex Editor window, and puts the cursor on the address shown. It's somewhat similar to how Bookmarks work in the Hex Editor.

Cheat Search

The cheat search is used to find a specific value in the games RAM by process of elimination.

The possibilities window is in the format of Address:Original Value:Current Value

The address is the location in the 6502's address space, the original value is the value that was stored at this address when the search was reset, and the current value is the value that is currently stored at that address. Selecting an item in this list will automatically cause the "Address" field in the cheat information box on the right side of the window to be updated with the selected address.

The "Reset" button resets the search process; all valid addresses are displayed in the possibilities window and the data values at those addresses noted in both the left and right columns. The number of possibilities is displayed at the top. Resetting will set it to 2048 or 10240 depending on if the game uses "On cartridge ram" (\$6000-\$7FFF). (See NES RAM)

The left column is the "previous value" and the right column is the "current value"

The "Known Value", "Equal", "Not Equal", Greater than" and Less than" buttons perform a search based on the search parameter and removes any non-matching addresses from the address list. It then sets the "previous value" column to the contents of the "current value"

"Known Value" will search for all addresses that match the value in the box to the right (the value is in hex).

"Equal" will search for all addresses that have the same value now as the last search (or since reset was pressed, if there has not yet been a search).

"Not equal" will search for all addresses that have changed sine the last search (or since reset was pressed, if there has not yet been a search). If the checkbox next to it is checked it will looks for values that have changed by the value in the number box to the right. For instance, if it is checked and the number is 5, it will search for all values that are +- 5 from the previous value.

"Greater than" functions like "Not equal" except it only searches for values that have increased since the last search.

"Less than" functions like "Not equal" except it only searches for values that have decreased since the last search.

Using the Results

Any value in the possibilities list can be sent to memory watch by double clicking it.

Highlighting it and hitting the "Add" button under the Active cheats window will automatically activate it as a cheat with the value set to its current value.

Example

Here is an example of cheat search in action.

Let's say I am playing Mega man 3 and I want to find Mega man's energy level in the game's ram. I will start by opening the ROM and selecting a level. At this point, I know Mega man's energy address is active. So I will pause the game and open the cheat search and hit the reset button. The game uses SRAM so the possibilities window will say 10240 "possibilities".

Next I will frame advance (or briefly unpause) the game. At this point I know Mega man's energy level is still the same as it was. So I click the "equal" button. Next I want to take damage. I know for sure now that the energy level has decreased so after the "ouch" animation, I click the "Less than button". This will cut the possibilities down significantly. Next I will advance some more and click the "Equal" button since I know the value is still the previous value. I will repeat this cycle until I am down to 1 or just a few values. From there I can double click the values to send them to memory watch to monitor them more closely to weed them out. (Note: Mega man's energy is located in \$00A2).

Context Menu

Right-clicking in the active cheats list brings up the context menu.

Toggle Cheat - does the same thing as double clicking

Poke cheat value - has a different affect that normal freezing, this makes a one time write of that value as opposed to freezing it temporarily to that value and having it restored later. It has the same affect as typing in values in the <u>Hex Editor</u>.

Goto In Hex Editor - Opens the <u>Hex editor</u> dialog to the position of the selected RAM value.

RAM Search

Ram Search

Ram Search is a tool originally written for <u>GENS rerecording</u>. It was ported to FCEUX in version 2.1.2. This dialog has also been ported to <u>SNES9x-rr</u>, <u>Desmume</u>, <u>PCEjin</u>, <u>VBA-rr</u>, <u>PCSX-rr</u>, <u>Yabause</u>, <u>VBjin</u>, and <u>FBA-rr</u>.

It is designed to filter ram values just like in the <u>Cheat Search</u> dialog. However, it features many options that are lacking in the Cheat Search dialog. Among these are search undo, search preview, a modulus filter, a data size option, signed/unsigned/hex options, autosearch, and several more compare by options.

Documentation on this dialog can be found on TASVideos here.

Hotkeys

<u>Hotkeys</u> can be assigned to common search commands so they can be easily selected while in the main window.

2008

RAM Watch

Ram Watch

Ram Watch is a tool originally written for <u>GENS rerecording</u>. It was ported to FCEUX in version 2.1.2. This dialog has also been ported to <u>SNES9x-rr</u>, <u>Desmume</u>, <u>PCEjin</u>, <u>VBA-rr</u>, <u>PCSX-rr</u>, <u>Yabause</u>, and <u>FBA-rr</u>.

It is designed to filter ram values just like in the <u>Cheat Search</u> dialog. However, it features many options that are lacking in the Cheat Search dialog. Among these are search undo, search preview, a modulus filter, a data size option, signed/unsigned/hex options, autosearch, and several more compare by options.

Documentation on this dialog can be found on TASVideos here.

2008

Memory Watch

Memory Watch Overview

Memory watch is a tool designed to values of specific known memory values in the game's <u>RAM</u>. Memory watch does not find values. To find useful values to monitor, see <u>Cheats</u>, <u>Ram filter</u>, <u>Hex Editor</u>, and <u>Debugger</u>.

Inserting Values

To display a ram value, simply type its address into one of the address fields. The name field allows you to put a brief description of the value.

Prefixes

You must put in the hexi-decimal value of the address, but the value will be displayed will be decimal by default.

To display the value in hex, use a prefix of "x" (such as x00FD).

Use the prefix "!" to display a 2 byte value.

Use a prefix of "X" to watch a 2 byte value in hex.

Saving/Loading Watch files

You can save your addresses into watch files, as well as loading previous files using the standard save,load,new options in the File menu.

FCEUX uses the /memw folder by default but you can specify a new default folder in the <u>Directory Override</u> menu.

Options Menu

If you select **Load on Start up**, Memory watch will load up automatically when FCEU is started.

If you select **Load Last File on Start up**, the most recent file in the Recent folder will be loaded when memory watch is loaded.

If you select **Collapse to 1 Column** (or press the right arrow button on the bottom left of the dialog), the memory watch dialog is reduced to just 1 column.

Frozen Memory Addresses

If one of the watched addresses is frozen by the <u>cheats dialog</u> or the <u>hex</u> <u>editor</u>, it will display blue in the memory watch dialog.

Memory Change Monitor

The bottom of the memory watch dialog displays a memory change monitoring section. This monitors the 1st two values of each memory watch column. Rather than monitoring the value itself, this monitors the value's behavior.

The address being monitored is under the address column.

The Formula drop down box shows which criteria the change monitoring is using.

The count value displays how many times the value has changed based

on the criteria.

Reset will reset the count to 0.

Usage Example:

As an example of the memory change monitoring, Let's say we are recording a movie of the game Super C and want to keep track of when the game lags.

The ram address 001C functions as a "lag flag". It will remain 0, then change to a positive value on a frame that the game lags.

We could put 001C in one of the 1st two memory watch edit boxes. Then set the corresponding formula in the memory change monitoring to "> then" (greater than). Now the count will show us how many lag frames occur in the movie.

2008



TAS Edit

TAS Edit is an overhaul in very logic of creating TAS movies (see <u>Tool</u> <u>Assisted Speedruns</u>). It is a powerful new design that takes movie making from less of a "recording" concept to a "creating an input file" way of thinking. The traditional savestates are not used and is replaced with a list of frames and input. Clicking on a frame sends the emulator to that frame.

As of the 2.0 release this is an unfinished tool and therefore much of the features are not implemented.

2008

Covert fcm

Converting .fcm to .fm2 files

FCEUX uses a new movie file format (.fm2). In order to use movie files frame previous FCE Ultra versions (.fcm) you will need to convert to .fm2 first.

Using .fcm Convert

To use it simply highlight it. Then select the .fcm you wish to convert (or shift+click to select multiple .fcm files). Then click Open to have the select files converted. All files selected will have a matching .fm2 file copied into the same folder.

2008

Auto Fire Configurations

Auto Fire Settings Autofire Pattern

The default configuration for an auto fire key is the alteration of on/off/on/off every frame. For most games this works nicely, but there are situations where this doesn't work properly. For example, Double Dragon 2 and Teenage Mutant Ninja Turtles run at 30fps (screen updates every other frame). To use autofire in these types of games, you would want to set the autofire pattern to 2 on / 2 off. In a situation where a players weapon on fires every 4th frame, you can set the autofire pattern to 1 on / 3 off.

Autofire Offset

The default is for certain frames to have the on setting and others to have the off setting. For instance, "on" might be lined up with a movie file's even numbers. But a situation may need the autofire pattern to have "on" on the odd numbers instead. In this case the autofire offset should be set to 1. This will delay the normal "on" fire by 1 frame. If an autofire pattern is set to 2 on / 2 off, an autofire offset of 2 might be necessary.

Alternate A and B

Alternate A and B is for a specific case where both the A and B autofire buttons are pressed simultaneously. With alternate A and B, the fire pattern will be A,B,A,B rather than A+B, off, A+B, off.

Note: All autofire patterns read the Lag Counter (see <u>display</u>) and skip over any frames where input is not polled. This means that in a laggy area, the autofire pattern will not be affected.

External Input

Use External Input

Use External Input release control of FCEU so that an external program (such as a TAS bot) can save/load states, frame advance, and deliver input. This feature was original implemented for older version of **Basic** bot.

2008

Text Hooker

Text Hooker

(written by Ugly Joe, author of the Text Hooker tool)

What is Text Hooker?

Here's a premise for you. Suppose you've pirated a bunch of Japanese NES roms and you load one of them up at random. Cool music. Cool title screen. You go to start a game, put in ???? at the name entry screen, and get to the actual game. Well, big surprise here, it's an RPG. You soon realize that you have no idea what people are saying, what shops are selling, or what your battle options are. It can be fun to trial-and-error for a while, but you're ultimately stuck in the first town. Time to load up a new ROM.

Well, being the aspiring Japanophile that I am, I have all kinds of translation tools and websites at my disposal. It's not impossible for me to figure out the kana for an item name, put it into a website somewhere, and figure out what it is. It's a slow process, but I can figure out short, simple strings of Japanese text. Sometimes, this is all I need to know to get by.

This is why I made the Text Hooker. What it allows you to do is highlight text boxes in the game and copy the kana right to the clipboard. I no longer have to look up stuff, I can just copy from the emulator, paste into the website, and go from there. While developing it, I took it a bit further by adding a (shoddy) translator right into the app, and added features such as word substitutions (so you only have to look up the word once and then the app will know what it is as soon as you copy it). What you end up with is kind of like a translator's notebook. It keeps commonly used words in a dictionary and helps you get through a Japanese game without having too much knowledge of the Japanese language.

What do I need to use to use it?

Some knowledge of the Japanese language I really can't say how much you need to know, but I suppose the more you know the better. I could be wrong, but I think you need to know at least something about the language before you can start copy/pasting translations.

Know how to make a Japanese table file I'm not going to explain how to do this since there are adequate tutorials already out there. You'll need to be able to do this per game in order for the Text Hooker to work.

Japanese font support

Okay, I have tested this thing on a Win98 installation with no Japanese font. It still works. However, I didn't test it for very long and I'm not sure how well translation websites are going to work without it. So, it might work without Japanese font support, but I'm not officially saying it does.

A Japanese ROM Duh, you'll need a game to play. Find it yourself.

How do I use the Text Hooker?

First of all, you need to make your table file. The text hooker doesn't use Thingy tables, but uses a modified Thingy table instead. So, make your standard Thingy table file, but save it with a .tht extension (instead of .tbl). What you need to add to the table are the dakuten and handakuten marks (tenten and maru). The byte for the dakuten mark needs to be set to tenten and the byte for the handakuten mark needs to be set to tenten. Like:

DC=tenten DD=maru

If you don't do this, the Text Hooker will fail miserabley when copying the text over from the game.

Once you have your table file ready, open up your rom in FCEUXDSP CE and open the text hooker window (Tools -> Text Hooker). Click on the "Load Table" button and open up your .tht file. Now you can really get ready to work.

Basic Usage

A warning

All information is saved in the table file. You have to save your table manually using the Save Table button. If you close the Text Hooker window or load a different table, your changes since the last save will be lost. You will not be prompted to save changes. Please remember to save!

Making Selections

The Selection Window is where you select the text in the game. It is basically the same view as the actual emulator window, but it updates less often and does not show sprites (text is not drawn with sprites, so they are not needed). To make a selection, click on a deselected tile and drag your mouse. To remove a selection, click on a selected tile and drag your mouse. It works a lot like a pen tool and an eraser tool in standard paint programs.

Once you have made a selection, you can save it for later use. This comes in handy since most RPGs will display their text boxes and battle menus in the same place throughout the entire game. To save a selection, type a name for the selection into the New Selection Name field and press the Save Selection button. Note that this selection will not be saved to your table file until you press the Save Table button.

You can also use the Clear Selection button to deselect all of the tiles in the selection window.

Please note that when you select text, you should not select the mostly blank rows that contains the dakuten and handakuten marks. You're essentially selecting every other row. Please see the UI image above for an example.

Translating Text

Once you've made a selection, press the big Snap button to copy the text into the Hooked Text window. Only the tiles that are defined in your table file will be copied over. All other tiles will be ignored. Once you have some Japanese text in your Hooked Text window, you have a few options. You can press the Excite.co.jp button to receive a really bad translation (better than Babelfish, but still bad) in the Translated Text window, or you can select all or part of the text in the Hooked Text window and copy/paste it into another translation tool or website. If you're translating a block of text (as opposed to item names or menus), you should probably use the Trim button to clean up the excess whitespace.

Please bear in mind that, due to the limitations of the NES, Japenese games use very little kanji. This means you'll have to look up the kana representation of what would normally be a kanji. Most translation tools will give you a hard time about this.

The word substitution feature can be used to process the selected text before it is sent to the Hooked Text window. By entering in Japanese-to-English definitions, you build up your word subs dictionary. If word subs are enabled and you press the snap button, the selected text is checked against your dictionary and any words that it finds are replaced by their definition.

This is useful for a few reason. One, many words written in katakana don't translate too well. You can use this to stop the translators from mangling them. Two, character names are often the same thing as words. For example, if your character's name is ??? (Sakura), the translator will likely translate it to "cherry blossom". If you define ??? as Sakura, then you won't have to worry about that. Three, you only really need to translate menus and items once. Once you have them figured out, add them to your dictionary. This way, you can just select your menu (perhaps from a saved selection?) and press Snap -- instant menu translation! Four, I'm not positive about this, but if you know that a string of kana is going to always mean a particular kanji, you could put the kana in the Japanese side and the kanji in the English side. This would aid translators since it wouldn't have to try and figure it out itself. Note that I haven't tested that last one since I don't know enough kanji to

put it to the test.

Again, please remember that your dictionary will not be saved unless you use the Save Table button.

Tweaking Here are some other helpful features.

Pause Button: this is used to pause and unpause the emulator.

Scanline: this is used to determine on what scanline the Selection Window will be updated. Some games will switch their font tiles in and out of the PPU. If this happens, you may need to change the scanline to a bigger number in order to see the tiles you're looking for. For example, this happens a lot in the game Metal Slader Glory.

Update every x frames: this is used to determine how often the Selection Window is updated. The smaller the number, the slower the emulator will go.

Selection Window checkbox: this is used to determine whether or not the selection window should be updated. If you're not going to be needing the Text Hooker for a while, you should probably uncheck this box while you play.

Word Substitution checkbox: this is used to determine whether or not word substitution will be used.

(han)dakuten mark position checkbox: this is used to tell the text hooker where the dakuten and handakuten marks are located in relation to the kana. Most games will use Above, but some games that try to squeeze in as much text into a small area as possible will use Right.

Features > Text Hooker > Reference

Features > Text Hooker > Reference > Text Hooker Table file reference

I suppose this is the kind of thing that should be documented, so here it is. When I started to make this thing, I was just using Thingy tables. When I started to add other features, I knew I needed to save them somewhere. It seemed kind of dumb to me to store this information in separate files, so I decided I would append the other sections to the end of the table files. In the far off chance that there becomes some kind of archive for Text Hooker table files, I decided to use a different extension.

A .tht file is comprised of three parts (and possibly more in the future). The first part resembles a Thingy table, since it's more or less that same thing. You have a hex byte value, and equals sign, and the corresponding character after the equals sign. The biggest difference from Thingy tables is that the tenten and maru marks must be defined using the words tenten and maru.

The next section is the Selections storage. This section begins with a

[selections]

declaration. What follows are hashes for saved selections (name of selection, equals sign, hash). The hashes should be safe for viewing and saving in any text editor that is capable of viewing and saving Japanese text. These hashes are, admittedly, under tested. If anyone can find a situation in which the selection hashes are corrupted but the rest of the table file is not, please let me know. Up next is the Word Substitution Dictionary. This section begins with a

[words]

declaration. These lines are formatted in a Japanese=English manner. You should be able to have Japanese or English on either or both sides. It's nothing more than a list of values used during a search and replace function.

2008

Debug

<u>Debug</u>

Guides for the specific tools and settings under FCEUX's Debug menu.

<u>Debugger</u>

A tool for looking at game instructions in assembly language. With experience,

one can use it to fix game patching errors, or find RAM and Game Genie codes.

PPU Viewer

A tool that displays the current PPU contents and related information. The PPU

viewer allows you to view the graphic squares that make up what's displayed.

Name Table Viewer

A tool for displaying the current Name Table contents. Helps to isolate PPU

and tile information, which allows the debugger to be used to check PPU coding.

Hex Editor

A tool for displaying a game's RAM contents and for memory poking. Also allows for reading in the raw PPU data, copy/paste-ing RAM, and visually debugging RAM.

Trace Logger

Captures assembly code instructions and outputs them to a file or the window. Very

useful for modifying code, finding crash addresses, fixing transferred routines, and

for comparing routine function between a game and a persistently buggy NSF.

Code/Data Logger

Allows you to extract the data used by a game. Make patch demos, find data

loaded by a game around a certain point, or just map out a single routine run.

Game Genie Encoder/Decoder

Allows you to add Game Genie codes to the Cheats menu, decode existing

ones to their component information, and (re)create a code with desired values.

2008

Debugger

<u>Debugger</u>

Taken from the FCEUXDSP 1.07 documentation.

Introduction

The debugger is a powerful tool that reads, displays, and manipulates assembly language instructions in a game.

Debugger Features

-When you hold the mouse over the left pane in the debugger, you can now see the ROM file address of the data loaded there.

-Right-click in that pane, it will bring up the Hex Editor at that address so you can immediately begin editing.

-Left-clicking in that pane brings up the inline assembler.

-"Break on bad opcode" feature; this can help you figure out where your game is crashing. Middle-clicking on a byte will bring up the <u>Game Genie</u> <u>Encoder</u> at that address, so you can easily make Game Genie codes.

-Debugging data like breakpoints or bookmarks are automatically saved and restored when games are closed / opened.

-Ability to give breakpoints a brief description/name.

-All debugging information for addresses < \$8000 into the name list file romname.nes.ram.nl.

-Added a feature to NL files to support arrays.

-Range options for freezing / unfreezing addresses

-Dump RAM to file option

-Dump PPU memory to file option

-Automatically generated .deb debug files created if the debugger is used.

Using Debugger

Symbolic Debugging

The most important feature (at least for me) that was introduced in FCEUXD SP is symbolic debugging. With this new feature it's possible to rename addresses in the disassembly window (like \$C022) to easily understandable names (like AddHealthpoints). It's also possible to add comments to lines in the disassembly window.

To be able to use this feature it's necessary to create so called name list files (*.(bank).nl/*.ram.nl, Ex: NES Test Cart (PD).nes.0.nl, NES Test Cart (PD).nes.ram.nl) which contain all names and comments you wish to display in the disassembly window. These files are plain ASCII files of the following format (example follows):

\$C000#NewName1#Comment1 \$C002##Comment2 \$C004#NewName2# \$C006#NewName3#MultilineComment-Part1 \MultilineComment-Part2 \MultilineComment-Part3 \$C008/10#NewName4#

Every line contains two *#* characters which separate the three parts of one line: The first part (starting with a \$ character) is the address to be renamed. Optionally you can add a "/number" part which marks the offsets as a beginning of an array of the given size (the size must be specified in hex form). The second (optional) part is the new name of that address. Whenever the line of that address is shown in the disassembly window an extra line saying "Name: NewName" is shown above it. Instructions referencing this address, for example JSR \$C000 are also changed to JSR NewName1 (in that example). The third (optional) part is the comment that's also added above the disassembly line the comment refers to. It works exactly like the additional name line, only the prefix of that line is different. Comment lines start with "Comment: " rather than with "Name: ". Multi-lines comments are possible. Lines starting with a \ character are just appended to the comment of the preceding line. Multi-line comments are also shown in multiple lines in the disassembly window.

Let's get back to the example.

The first line contains all three parts. Using this name list file all references to the address

\$C000 are replaced with NewName1 and whenever line \$C000 is shown in the disassembly window an

additional comment is also visible right above the actual disassembled line.

The second line defines only a comment while the third line defines only a name. Following that

there's a multi-line comment definition for address \$C006.

The last line defines an array called NewName4 of size \$10 (= 16) bytes starting at offset \$C008.

Now you know the format of the nl files but you do not yet know the naming convention for the

file names. Due to the bank-swapping nature of the NES it's getting a little bit difficult here.

Each bank needs it's own nl file. The naming convention goes like this: Take the name of the ROM

file and just add ".X.nl" to it where the X is the hexadecimal representation of the number of the

ROM bank. Suppose you have the ROM file "Faxanadu (U).nes" and you want to create a nl file for

ROM bank 15. As 15 is 0x0F in hex the name of the nl file would be "Faxanadu (U).nes.F.nl". All

nl files go into the same directory as the ROM file itself.

There is also the *.ram.nl file specification, which allows you to substitute RAM addresses for

execution addresses, and have those named as well. In this case, you could use lines of this type:

\$00A5#Mic Test OK#00=Not Passed, 01=Passed

You can enable and disable symbolic debugging by clicking the checkbox

"Symbolic Debugging" in the debugger window. To forcibly reload the nl files of the currently active ROM file press the button with the text "Reload Symbols".

Arrays

The array feature is an easy way to group names and comments for sequential offsets.

\$C000/5#NewName1#Comment1

is equivalent to

\$C000#NewName1#Comment1

Inline Assembler

The debugger an Inline Assembler designed by Parasyte. To activate it, left-click in the left pane of the debugger, beside the assembly display. To use it, type in some code and press Enter to add it to the patch list. If you make a mistake, press "Undo". Once the patch is set up the way you want it, press "Apply". Be aware that this cannot be undone unless you reload the ROM. Parasyte implemented this feature before I had the Hex Editor working, otherwise I would have implemented a way to undo it from there. Press "Save" to write to the ROM file on disk; note that this will also save any changes you may have done in the Hex Editor.

2008

PPU Viewer

PPU Viewer

Introduction

The NES architecture includes a 6502 CPU as well as a custom video controller known as a PPU (Picture Processing Unit). The PPU's video memory is separated from the main CPU memory and can be read/written via special ports (see <u>PPU Memory</u>).

The PPU viewer will display the contents of the current PPU memory. (It does not alter game data in anyway).

Using PPU Viewer

Show on Scanline

This options makes it show what the PPU looks like when the screen is drawing that scanline. It is useful for games like SMB, that swap pattern tables mid-frame (status bar stuff).

Right clicking on one of the PPU panels cycles the palettes it is shown with, though pattern palettes and then sprite ones Putting the mouse cursor over a tile will display the tile address. Moving cursor over palette color will give palette address.

2008

Name Table Viewer

Name Table Viewer

This displays the name tables as they exist in PPU memory. Furthermore, it shows you the game's current mirroring, and the current state of the PPU's scroll registers (if the option for this is set). It also lets you change the mirroring on the fly (which will break most games).

Using the Name Table Viewer

Note that the Name Table Viewer will display the name tables using whatever CHR is present at the time the "Display on Scanline" scanline is reached. So for example if it does not correctly display a game's status bar, try setting it to update on a scanline in which the status bar is displayed.

The same applies to the Scroll Lines: they display the state of the PPU scroll registers when the "Display on Scanline" scanline is reached. So for example if said scanline is within the game's status bar, it will not display level scrolling because the horizontal scroll is always zero at the time that scanline is drawn. To display the level scrolling, set it to update on a scanline in which the level is displayed.

Display on scanline

This will show what it looks like when the NES has finished drawing that many scanlines to screen including any PPU data scroll line movement

Getting Tile Addresses

Placing the mouse cursor over the name table image will display the tile address of a given tile.

2008

Hex Editor

Hex Editor

Introduction

The Hex editor is a very powerful memory viewing/editing tool and obsoletes the Memory Viewer tool from the FCE Ultra and FCEU Rerecording branches.

It can do a wide range of things. It allows you to view the entire RAM & ROM contents in an expandable dialog Window. It makes it easy to edit the game's RAM, PPU memory, and even its currently-loaded ROM data by simply typing in values in the editor. You can also "freeze" parts of RAM (to prevent the game from modifying the data there), search for data, and even copy and paste data to/from the clipboard. Furthermore, table files are supported, so you can edit a game's text in real-time and see the result immediately.

Basically, it lets you tinker with any part of a game's RAM or ROM while it is running.

Using the Hex Editor

The Hex Editor lets you edit three major areas:

1. NES MEMORY

This allows you to directly edit all of the NES address space (\$0000-\$FFFF). While you can easily modify RAM, or write directly to registers by typing in data, you cannot modify ROM data (\$8000-\$FFFF) itself. This is because most mappers have registers which are located in this space; so writing there can trigger mapper operations that may cause the game to crash or glitch if you don't know what you're doing. If you want to edit the ROM itself, right-click on the offset and select "Go here in ROM file"; that will take you directly to where you need to be so you can start editing. You can also freeze RAM by clicking on it with the middle mouse button, or by using the right-click menu. This works by adding it directly to the Cheat List, which you can see from the Cheat Console. Finally, the right-click menu can be used to quickly add a read or write breakpoint to the debugger.

2. PPU MEMORY

This allows you to directly view and write to PPU memory (VRAM).

3. THE ROM FILE

This is possibly the coolest part of FCEUXD: It allows you to edit the ROM file in real-time, i.e. while the game is running. If you make a mistake, press Ctrl+Z or Edit->Undo to undo your change (then load a save-state if the game crashed). If you have the Code/Data Logger running, then bytes that were logged as code will be colored yellow, while bytes logged as data will be colored blue. Bytes that have been logged as code *and* data will be colored green.

The Hex Editor also has support for table files (*.tbl) to map bytes to text. Each line consists of four characters of the form "xx=y", where "xx" is the hex value, and "y" is the character that that value represents. I have also added an extension to represent the Return key: xx=ret whereby pressing the Return key will enter that value into the ROM. You can copy/paste data or text by selecting it and using Ctrl+Z (to copy) and Ctrl+V (to paste). Plus, there is an Edit->Find feature that you can use to search for data. This feature should be fairly intuitive, so I won't bother to explain it.

When you're done editing, remember to save the ROM file (File->Save) or your changes will be lost when you close the ROM.

Why can't I edit NES memory beyond \$8000?

NES memory from \$8000-\$FFFF is where the game's PRG-ROM code is mapped. Whenever you type in a value in the NES memory editor, it effectively writes that value to that address. Many games use mappers,

which are usually accessed by writing to \$8000-\$FFFF (which is readonly)... and if *you* were to do so, it may trigger a bankswitch, which could easily make the game crash. In any event, doing so will not modify the ROM itself. What you *can* do, though, is edit the PRG-ROM itself by right-clicking on the offset you wish to edit, and selecting "Go here in the ROM file", which should take you to that spot in the ROM instead, where you can change the data at instead.

2008

Trace Logger

Trace Logger

The Trace Logger logs every executed instruction and every byte of ROM accessed to the window, or a file if you prefer. Logging to a file is useful if you just want to dump everything that was executed and then search through it later. Logging to the window is useful when you wish to see the instructions that were executed prior to a breakpoint being hit. Both options produce the same data, but the desire to keep that data for a short amount of time or a long amount of time will determine which is best for you.

Using the Trace Logger

The Trace Logger is a very nice feature which logs each instruction as it is being executed; this log can be output either to a file or the window. If you choose to log to the window, you can set how many lines it will retain before discarding old lines. The higher this setting, the more RAM it will consume, but the more lines you'll have available to work with. Normally, when logging to a window, the only time it shows the log is if you snap the <u>debugger</u> or pause execution. The option to automatically update the log window will keep the window updating while the game runs; this is normally useless unless it is working with the Code/Data Logger to only show newly-executed instructions.

The Trace Logger has extra options which work with the Code/Data Logger so that it only shows instructions executed for the first time, or those which access data for the first time. This can be quite useful for finding certain key routines or finding otherwise impossible-to-find data in almost any game. The best way to use this feature is in conjunction with the option to automatically update the window while logging. Then, as you play the game, you can watch new results appear at once. If you're searching for something specific, try to get everything-- EXCEPT what you're looking for-- to execute, then watch closely as what you're looking for executes for the first time.

There are two ways to filter what the Code/Data Logger shows. The first filter lets you log only newly-executed code (so that an instruction is not logged again if it has already been logged). The second logs only instructions when they access data which hadn't been accessed before. Note that both filters can be used at once (which shows bytes that pass either filter).

2008

Code/Data Logger

Code/Data Logger

(Taken from the FCEUXD readme.txt)

Introduction

The Code/Data Logger makes it *much* easier to reverse-engineer NES ROMs. The idea behind it is that a normal NES disassembler cannot distinguish between code (which is executed) and data (which is read). The Code/Data logger keeps track of what is executed and what is read while the game is played, and stores this information in a .cdl file, which is essentially a mask that tells which bytes in the ROM are code and which are data. The file can then be used in conjunction with a suitable disassembler (which I plan to make) to disassemble only the actual game code, resulting in a much cleaner source code, with code and data properly separated.

Using the Code/Data Logger

The Code/Data Logger keeps track of every byte in the ROM and records whether it's code (is executed) or data (is read). In the future, I hope to combine this with a suitable disassembler that will disassemble only bytes marked as "code", in order to generate near-perfect source code (provided you play through the game several times, very thoroughly, to ensure everything gets logged). In order to get that feature to work, I need to get the Address Label Logger working, in order to create all the labels.

But right now, it is very useful for finding specific code and data by using it with the Trace Logger (see above for instructions on doing this). Furthermore, while it is running, the Hex Editor will color-code bytes depending on whether they were logged as code or data. And it can also be used to create a stripped NES ROM (see below). Some notes: when you open another .cdl file, it does not clear the current log; instead, it combines it with the information in the file. This can be useful if you're trying to obtain a complete log, as multiple people can play through the game and keep code/data logs, and then the results can be combined. But if you would like to actually clear the code/data log, press the "Reset Log" button.

A "stripped" NES ROM is one in which everything that was not logged by the code/data logger is removed. This may be just a novelty feature, however it can be useful because you can view the ROM in a Hex Editor, and you'll see only the parts that were used while playing. Furthermore, you could use it to create a demo ROM by only playing through the parts you would like others to see.

To do so, follow these steps:

1. Open the Code/Data Logger, and press Start to begin logging.

2. Perform a soft and a hard reset while logging, in order to capture the ROM's startup sequence. If you don't do so, you can distribute a savestate file so they will start from within the game.

3. If the game has Save-RAM (e.g. Zelda), you will need to capture the game's Save-RAM initialization routines; you can try to do so by deleting the game's *.sav file and then perform a soft and hard reset (F10, F11) again while logging.

4. Play through whatever levels you want present in the demo. Be sure to perform every move, get every item, etc., so that the code and data necessary for those things are logged. If, for example, you fail to perform some special move, then if someone plays the stripped ROM and attempts to perform that move, the game may very well crash or glitch up.

5. Save the stripped NES ROM.

6. Rejoice, for it is done.

CDL log files are just a mask of the PRG-ROM; that is, they are the same size as the PRG-ROM, and each byte represents the corresponding byte of the PRG-ROM. The format of each byte is like so (in binary):

xPdcAADC

C = Whether it was accessed as code.

- D = Whether it was accessed as data.
- AA = Into which ROM bank it was mapped when last accessed:

00 = \$8000-\$9FFF 01 = \$A000-\$BFFF

- 10 = \$C000-\$DFFF 11 = \$E000-\$FFFF
- c = Whether indirectly accessed as code.

(e.g. as the destination of a JMP (\$nnn) instruction)

d = Whether indirectly accessed as data.

(e.g. as the destination of an LDA (\$nn), Y instruction)

P = If logged as PCM audio data.

x = unused.

CDL files make possible a number of things never before done. First, a PCM data ripper could be created that scans for data that has the 'P' bit set, in order to find/rip/play every PCM sample in a ROM. Also, it is possible for someone to make a more intelligent ROM corruptor that only corrupts data (by checking the 'D' bit). In any case, the Code/Data Logger opens many new possibilities for discovering useful things in games. Another interesting possibility (which is now partially supported) would be to use the Code/Data Logger on an NSF file to create a stripped NSF. Such an NSF would contain nothing but the relevant subroutines and data required by each tune played; this would be helpful to NSF rippers by removing irrelevant information. Thus, an NSF ripper could create a stripped NSF by listening to each track while the Code/Data Logger operates on it, and then saving the stripped NSF. It should be noted that this capability, though tested and working on private builds, is detrimental to the process of fixing broken NSF files. For this reason, data logging is allowed for NSF files, but stripping NSF files of unused data is disabled.

2008

Game Genie Encoder/Decoder

Game Genie Decoder/Encoder

This will take an NES address space PRG address (\$8000-\$FFFF), a comparison value (for 8-letter GG codes; refer to a Game Genie code FAQ for an explanation of what this does), and a Value that replaces the addressed byte.

Filling in the Address and Value fields will produce a 6-letter code; if you also fill out the Compare field, it will produce an 8-letter code. The code so produced will appear in the Game Genie Code box immediately; you can then click "Add to Cheat List" to activate it.

To decrypt a Game Genie code, enter it into the Game Genie Code box, and the Address and Value fields will be automatically filled in, as will the Compare field if it was an 8-letter code.

Adding Game Genie codes

In the Game Genie Code Decoder/Encoder window, type the code into the Game Genie Code box and click "Add to Cheat List", which will add it to the <u>Cheat Search</u> cheat list. You can then enable/disable them by double-clicking the code in the box (a * means the code is active).

Making Game Genie codes permanent

Using the Game Genie Code Decoder/Encoder, enter in your code in the "Game Genie Code" box, and under "Possible Affected ROM File Addresses", a list of possible matches (usually from 1 to 5) is displayed. Using the built-in Hex Editor, go to the first listed address in the ROM, and change its value to the value given in the "Value" box (of the GG code Decoder/Encoder window). If the desired effect isn't achieved, undo the change (Ctrl+Z) and try the next address. Repeat until the desired effect is achieved, and then save the ROM.

How do I make my own Game Genie codes?

First of all, you must:

* have a decent amount of ASM knowledge;

* know how to use the debugger;

* understand NES PRG-ROM bank switching.

Once you've found a part of PRG-ROM you want to change to create a code effect, snap the Debugger (if it's not so already) and find the code's location in the PRG-ROM's address space (\$8000-\$FFFF) (you'll want the debugger snapped so the game won't swap banks out from under you). Then, using the built-in <u>Hex Editor</u>, view the NES memory and go to the PRG-ROM address you wish to modify, then right-click the byte and choose "Create Game Genie Code at this Address". The Game Genie Code Decoder/Encoder will appear, with the Address and Compare boxes filled in (the Compare box represents the address's original value). Enter the new value into the "Value" box.

An alternative way to enter the code is to locate the desired address in the debugger, and then middle-click on it, which will summon the GG Code Decoder/Encoder. Then enter the code as described above

2008

Lua Scripting

Lua Scripting

Lua is a scripting language similar to Perl or Python. It allows for logical evaluation equivalent to languages like C but in a much more dynamic way that eliminates much of the need to compile programs and worry about low level resource management like deleting objects. In the context of FCEUX, Lua allows for direct control of the emulator through this logical construct.

What this means to the non-"programmer" is that it you can essentially automate certain tasks in FCEUX, such as holding controller inputs, displaying additional graphical information and saving/loading savestates.

A bit of previous programming knowledge will be useful in taking advantage of this feature, but it is certainly not a requirement. Lua is specifically written with the intention of being easier than most languages for anyone to understand and use.

Getting Started

The basics of Lua scripting, its implementation into FCEUX, and how to get started using Lua.

Using Lua

How to use Lua and basic syntax/commands that are useable under FCEUX.

Lua Functions List

A list of Lua functions available in FCEUX and a brief description of each.

Lua Bot

How to use Luau's version of Basic bot.

2008

This help file has been generated by the freeware version of <u>HelpNDoc</u>

Getting Started

Using Lua scripting

Lua is built into FCEUX as of 2.1.2, and luapack DLL files are no longer needed in this and later versions.

To run lua scripts in older versions of FCEUX, you will need the lua pack which can be found <u>here</u>. The .dll files must be unzipped in the same folder as fceux.exe.

Core Lua Documentation

If you have never programmed, you will probably want to start by learning the basic of Lua, which is too broad for the scope of this help file. Try searching on the Internet for "Lua tutorial". As of this writing, it's official homepage is <u>http://www.lua.org/</u>

If you are familiar with any programming language you will probably not have too much difficulty adjusting to the syntax and structure of Lua. You will probably also find useful information on the Internet.

GUI Frontend

To use a Lua script, you need to create one in a text editor. The name of the file created should end in .lua to indicate that it is a Lua script.

To run a Lua script, choose "Run Lua Script" ***from where*** In the dialog that pops up, click "Browse" and find the file you wish to run. This will insert the path of this file into the dialog. You can then click on "Run" to run the script or "Cancel" to return to FCEUX without running the script.

To end a Lua script, choose "Stop Lua Script" *** from where ***.

FCEUX Lua Basics

Your script will be constructed according to the rules of Lua, but you will use FCEUX-specific functions to interact with the emulator. For example, one of the most often-used functions is emu.frameadvance() which will tell the emulator to advance exactly one frame, which is the basic unit of time on an NES.

In general, your script will probably want to be run until you tell it to stop, so it will look something like this:

emu.speedmode("normal") -- Set the speed of the emulator

```
-- Declare and set variables or functions if needed
```

while true do

```
-- Execute instructions for FCEUX
```

```
emu.frameadvance() -- This essentially tells FCEUX to keep running end
```

The way instructions are sent to FCEUX is through a set of specially defined functions (and variables) which are called an API, the specification of which follows.

2008

This help file has been generated by the freeware version of HelpNDoc

Lua Functions List

Lua Functions

The following functions are available in FCEUX, in addition to standard LUA capabilities:

Emu library

emu.poweron()

Executes a power cycle.

emu.softreset()

Executes a (soft) reset.

emu.speedmode(string mode)

Set the emulator to given speed. The mode argument can be one of these:

- "normal"
- "nothrottle" (same as turbo on fceux)
- "turbo"
- "maximum"

emu.frameadvance()

Advance the emulator by one frame. It's like pressing the frame advance button once.

Most scripts use this function in their main game loop to advance frames. Note that you can also register functions by various methods that run "dead", returning control to the emulator and letting the emulator advance the frame. For most people, using frame advance in an endless while loop is easier to comprehend so I suggest starting with that. This makes more sense when creating bots. Once you move to creating auxillary libraries, try the register() methods.

```
emu.pause()
```

Pauses the emulator. FCEUX will not unpause until you manually unpause it.

```
emu.unpause()
```

Unpauses the emulator.

```
emu.exec_count(int count)
```

emu.exec_time(int count)

emu.setrenderplanes(bool sprites, bool
background)

Toggles the drawing of the sprites and background planes. Set to false or nil to disable a pane, anything else will draw them.

emu.message(string message)

Displays given message on screen in the standard messages position. Use gui.text() when you need to position text.

int emu.framecount()

Returns the framecount value. The frame counter runs without a movie running so this always returns a value.

int emu.lagcount()

Returns the number of lag frames encountered. Lag frames are frames where the game did not poll for input because it missed the vblank. This happens when it has to compute too much within the frame boundary. This returns the number indicated on the lag counter.

```
bool emu.lagged()
```

Returns true if currently in a lagframe, false otherwise.

bool emu.emulating()

Returns true if emulation has started, or false otherwise. Certain operations such as using savestates are invalid to attempt before emulation has started. You probably won't need to use this function unless you want to make your script extra-robust to being started too early.

bool emu.readonly()
Alias: movie.readonly

Returns whether the emulator is in read-only state.

While this variable only applies to movies, it is stored as a global variable and can be modified even without a movie loaded. Hence, it is in the emu library rather than the movie library.

emu.setreadonly(bool state)

Alias: movie.setreadonly

Sets the read-only status to read-only if argument is true and read+write if false. Note: This might result in an error if the medium of the movie file is not writeable (such as in an archive file).

While this variable only applies to movies, it is stored as a global variable and can be modified even without a movie loaded. Hence, it is in the emu library rather than the movie library.

emu.registerbefore(function func)

Registers a callback function to run immediately before each frame gets emulated. This runs after the next frame's input is known but before it's used, so this is your only chance to set the next frame's input using the next frame's would-be input. For example, if you want to make a script that filters or modifies ongoing user input, such as making the game think "left" is pressed whenever you press "right", you can do it easily with this.

Note that this is not quite the same as code that's placed before a call to emu.frameadvance. This callback runs a little later than that. Also, you cannot safely assume that this will only be called once per frame. Depending on the emulator's options, every frame may be simulated multiple times and your callback will be called once per simulation. If for some reason you need to use this callback to keep track of a stateful linear progression of things across frames then you may need to key your calculations to the results of emu.framecount.

Like other callback-registering functions provided by

FCEUX, there is only one registered callback at a time per registering function per script. If you register two callbacks, the second one will replace the first, and the call to emu.registerbefore will return the old callback. You may register nil instead of a function to clear a previously-registered callback. If a script returns while it still has registered callbacks, FCEUX will keep it alive to call those callbacks when appropriate, until either the script is stopped by the user or all of the callbacks are de-registered.

emu.registerafter(function func)

Registers a callback function to run immediately after each frame gets emulated. It runs at a similar time as (and slightly before) gui.register callbacks, except unlike with gui.register it doesn't also get called again whenever the screen gets redrawn. Similar caveats as those mentioned in emu.registerbefore apply.

emu.registerexit(function func)

Registers a callback function that runs when the script stops. Whether the script stops on its own or the user tells it to stop, or even if the script crashes or the user tries to close the emulator, FCEUX will try to run whatever Lua code you put in here first. So if you want to make sure some code runs that cleans up some external resources or saves your progress to a file or just says some last words, you could put it here. (Of course, a forceful termination of the application or a crash from inside the registered exit function will still prevent the code from running.)

Suppose you write a script that registers an exit function and then enters an infinite loop. If the user

clicks "Stop" your script will be forcefully stopped, but then it will start running its exit function. If your exit function enters an infinite loop too, then the user will have to click "Stop" a second time to really stop your script. That would be annoying. So try to avoid doing too much inside the exit function.

Note that restarting a script counts as stopping it and then starting it again, so doing so (either by clicking "Restart" or by editing the script while it is running) will trigger the callback. Note also that returning from a script generally does NOT count as stopping (because your script is still running or waiting to run its callback functions and thus does not stop... see here for more information), even if the exit callback is the only one you have registered.

bool emu.addgamegenie(string str)

Adds a Game Genie code to the Cheats menu. Returns false and an error message if the code can't be decoded. Returns false if the code couldn't be added. Returns true if the code already existed, or if it was added.

Usage: emu.addgamegenie("NUTANT")

Note that the Cheats Dialog Box won't show the code unless you close and reopen it.

bool emu.delgamegenie(string str)

Removes a Game Genie code from the Cheats menu. Returns false and an error message if the code can't be decoded. Returns false if the code couldn't be deleted. Returns true if the code didn't exist, or if it was deleted. Usage: emu.delgamegenie("NUTANT")

Note that the Cheats Dialog Box won't show the code unless you close and reopen it.

emu.print(string str)

Puts a message into the Output Console area of the Lua Script control window. Useful for displaying usage instructions to the user when a script gets run.

```
emu.getscreenpixel(int x, int y, bool
getemuscreen)
```

Returns the separate RGB components of the given screen pixel, and the palette. Can be 0-255 by 0-239, but NTSC only displays 0-255 x 8-231 of it. If getemuscreen is false, this gets background colors from either the screen pixel or the LUA pixels set, but LUA data may not match the information used to put the data to the screen. If getemuscreen is true, this gets background colors from anything behind an LUA screen element.

Usage is local r,g,b,palette = emu.getscreenpixel(5, 5, false) to retrieve the current red/green/blue colors and palette value of the pixel at 5x5.

Palette value can be 0-63, or 254 if there was an error.

You can avoid getting LUA data by putting the data into a function, and feeding the function name to emu.registerbefore.

FCEU library

The FCEU library is the same as the emu library. It is

left in for backwards compatibility. However, the emu library is preferred.

ROM Library

rom.readbyte(int address)
rom.readbyteunsigned(int address)

Get an unsigned byte from the actual ROM file at the given address.

This includes the header! It's the same as opening the file in a hex-editor.

rom.readbytesigned(int address)

Get a signed byte from the actual ROM file at the given address. Returns a byte that is signed.

This includes the header! It's the same as opening the file in a hex-editor.

Memory Library

memory.readbyte(int address)
memory.readbyteunsigned(int address)

Get an unsigned byte from the RAM at the given address. Returns a byte regardless of emulator. The byte will always be positive.

memory.readbyterange(int address, int length)

Get a length bytes starting at the given address and return it as a string. Convert to table to access the individual bytes.

memory.readbytesigned(int address)

Get a signed byte from the RAM at the given address. Returns a byte regardless of emulator. The most significant bit will serve as the sign.

memory.writebyte(int address, int value)

Write the value to the RAM at the given address. The value is modded with 256 before writing (so writing 257 will actually write 1). Negative values allowed.

int memory.getregister(cpuregistername)

Returns the current value of the given hardware register. For example, memory.getregister("pc") will return the main CPU's current Program Counter.

Valid registers are: "a", "x", "y", "s", "p", and "pc".

memory.setregister(string cpuregistername, int
value)

Sets the current value of the given hardware register. For example, memory.setregister("pc",0x200) will change the main CPU's current Program Counter to 0x200.

Valid registers are: "a", "x", "y", "s", "p", and "pc".

You had better know exactly what you're doing or you're probably just going to crash the game if you try to use this function. That applies to the other memory.write functions as well, but to a lesser extent. memory.register(int address, [int size,]
function func)
memory.registerwrite(int address, [int size,]
function func)

Registers a function to be called immediately whenever the given memory address range is written to.

size is the number of bytes to "watch". For example, if size is 100 and address is 0x0200, then you will register the function across all 100 bytes from 0x0200 to 0x0263. A write to any of those bytes will trigger the function. Having callbacks on a large range of memory addresses can be expensive, so try to use the smallest range that's necessary for whatever it is you're trying to do. If you don't specify any size then it defaults to 1.

The callback function will receive two arguments, (address, size) indicating what write operation triggered the callback. If you don't care about that extra information then you can ignore it and define your callback function to not take any arguments. The value that was written is NOT passed into the callback function, but you can easily use any of the memory.read functions to retrieve it.

You may use a memory.write function from inside the callback to change the value that just got written. However, keep in mind that doing so will trigger your callback again, so you must have a "base case" such as checking to make sure that the value is not already what you want it to be before writing it. Another, more drastic option is to de-register the current callback before performing the write.

If func is nil that means to de-register any memory

write callbacks that the current script has already registered on the given range of bytes.

memory.registerexec(int address, [int size,]
function func)
memory.registerrun(int address, [int size,]
function func)
memory.registerexecute(int address, [int size,]
function func)

Registers a function to be called immediately whenever the emulated system runs code located in the given memory address range.

Besides that, most of the information about memory.register applies to this function as well.

Joypad Library

table joypad.get(int player)
table joypad.read(int player)

Returns a table of every game button, where each entry is true if that button is currently held (as of the last time the emulation checked), or false if it is not held. This takes keyboard inputs, not Lua. The table keys look like this (case sensitive):

up, down, left, right, A, B, start, select

Where a Lua truthvalue true means that the button is set, false means the button is unset. Note that only "false" and "nil" are considered a false value by Lua. Anything else is true, even the number 0.

joypad.read left in for backwards compatibility with older versions of FCEU/FCEUX.

table joypad.getdown(int player)
table joypad.readdown(int player)

Returns a table of only the game buttons that are currently held. Each entry is true if that button is currently held (as of the last time the emulation checked), or nil if it is not held.

table joypad.getup(int player)
table joypad.readup(int player)

Returns a table of only the game buttons that are not currently held. Each entry is nil if that button is currently held (as of the last time the emulation checked), or false if it is not held.

joypad.set(int player, table input) joypad.write(int player, table input)

Set the inputs for the given player. Table keys look like this (case sensitive):

up, down, left, right, A, B, start, select

There are 4 possible values: true, false, nil, and "invert". true - Forces the button on false - Forces the button off nil - User's button press goes through unchanged "invert"- Reverses the user's button press

Any string works in place of "invert". It is suggested as a convention to use "invert" for readability, but strings like "inv", "Weird switchy mechanism", "", or "true or false" works as well as "invert". nil and "invert" exists so the script can control individual buttons of the controller without entirely blocking the user from having any control. Perhaps there is a process which can be automated by the script, like an optimal firing pattern, but the user still needs some manual control, such as moving the character around.

joypad.write left in for backwards compatibility with older versions of FCEU/FCEUX.

Zapper Library

```
table zapper.read()
```

Returns the zapper data When no movie is loaded this input is the same as the internal mouse input (which is used to generate zapper input, as well as the arkanoid paddle).

When a movie is playing, it returns the zapper data in the movie code.

The return table consists of 3 values: x, y, and fire. x and y are the x,y coordinates of the zapper target in terms of pixels. fire represents the zapper firing. 0 = not firing, 1 = firing

Note: The zapper is always controller 2 on the NES so there is no player argument to this function.

Input Library

table input.get()
table input.read()

Reads input from keyboard and mouse. Returns pressed keys and the position of mouse in pixels on game The function returns a table with at least screen. two properties; table.xmouse and table.ymouse. Additionally any of these keys will be set to true if they were held at the time of executing this function: leftclick, rightclick, middleclick, capslock, numlock, scrolllock, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12, F13, F14, F15, F16, F17, F18, F19, F20, F21, F22, F23, F24, backspace, tab, enter, shift, control, alt, pause, escape, space, pageup, pagedown, end, home, left, up, right, down, numpad0, numpad1, numpad2, numpad3, numpad4, numpad5, numpad6, numpad7, numpad8, numpad9, numpad*, insert, delete, numpad+, numpad-, numpad., numpad/, semicolon, plus, minus, comma, period, slash, backslash, tilde, quote, leftbracket, rightbracket.

string input.popup

Alias: gui.popup

Requests input from the user using a multiple-option message box. See gui.popup for complete usage and returns.

Savestate Library

object savestate.object(int slot = nil)

Create a new savestate object. Optionally you can save the current state to one of the predefined slots(1-10) using the range 1-9 for slots 1-9, and 10 for 0, QWERTY style. Using no number will create an "anonymous" savestate. Note that this does not actually save the current state! You need to create this value and pass it on to the load and save functions in order to save it.

Anonymous savestates are temporary, memory only states. You can make them persistent by calling memory.persistent(state). Persistent anonymous states are deleted from disk once the script exits.

object savestate.create(int slot = nil)

savestate.create is identical to savestate.object, except for the numbering for predefined slots(1-10, 1 refers to slot 0, 2-10 refer to 1-9). It's being left in for compatibility with older scripts, and potentially for platforms with different internal predefined slot numbering.

savestate.save(object savestate)

Save the current state object to the given savestate. The argument is the result of savestate.create(). You can load this state back up by calling savestate.load(savestate) on the same object.

savestate.load(object savestate)

Load the the given state. The argument is the result of savestate.create() and has been passed to savestate.save() at least once.

If this savestate is not persistent and not one of the predefined states, the state will be deleted after loading.

savestate.persist(object savestate)

Set the given savestate to be persistent. It will not be deleted when you load this state but at the exit of this script instead, unless it's one of the predefined states. If it is one of the predefined savestates it will be saved as a file on disk.

savestate.registersave(function func)

Registers a callback function that runs whenever the user saves a state. This won't actually be called when the script itself makes a savestate, so none of those endless loops due to a misplaced savestate.save.

As with other callback-registering functions provided by FCEUX, there is only one registered callback at a time per registering function per script. Upon registering a second callback, the first is kicked out to make room for the second. In this case, it will return the first function instead of nil, letting you know what was kicked out. Registering nil will clear the previously-registered callback.

savestate.registerload(function func)

Registers a callback function that runs whenever the user loads a previously saved state. It's not called when the script itself loads a previous state, so don't worry about your script interrupting itself just because it's loading something.

The state's data is loaded before this function runs, so you can read the RAM immediately after the user loads a state, or check the new framecount. Particularly useful if you want to update lua's display right away instead of showing junk from before the loadstate.

savestate.loadscriptdata(int location)

Accuracy not yet confirmed.

```
Intended Function, according to snes9x LUA
documentation:
Returns the data associated with the given savestate
(data that was earlier returned by a registered save
callback) without actually loading the rest of that
savestate or calling any callbacks. location should be
a save slot number.
```

Movie Library

```
bool movie.active()
```

Returns true if a movie is currently loaded and false otherwise. (This should be used to guard against Lua errors when attempting to retrieve movie information).

int movie.framecount()

Returns the current frame count. (Has the same affect as emu.framecount)

string movie.mode()

Returns the current state of movie playback. Returns one of the following:

- "record"
- "playback"
- "finished"
- nil

movie.rerecordcounting(bool counting)

Turn the rerecord counter on or off. Allows you to do some brute forcing without inflating the rerecord count.

```
movie.stop()
movie.close()
```

Stops movie playback. If no movie is loaded, it throws a Lua error.

int movie.length()

Returns the total number of frames of the current movie. Throws a Lua error if no movie is loaded.

```
string movie.name()
string movie.getname()
```

Returns the filename of the current movie with path. Throws a Lua error if no movie is loaded.

```
movie.getfilename()
```

Returns the filename of the current movie with no path. Throws a Lua error if no movie is loaded.

```
movie.rerecordcount()
```

Returns the rerecord count of the current movie. Throws a Lua error if no movie is loaded.

```
movie.replay()
movie.playbeginning()
```

Performs the Play from Beginning function. Movie mode is switched to read-only and the movie loaded will begin playback from frame 1.

If no movie is loaded, no error is thrown and no message appears on screen.

bool movie.readonly()
bool movie.getreadonly()
Alias: emu.getreadonly

FCEUX keeps the read-only status even without a movie loaded.

Returns whether the emulator is in read-only state.

While this variable only applies to movies, it is stored as a global variable and can be modified even without a movie loaded. Hence, it is in the emu library rather than the movie library.

movie.setreadonly(bool state)

Alias: emu.setreadonly

FCEUX keeps the read-only status even without a movie loaded.

Sets the read-only status to read-only if argument is true and read+write if false. Note: This might result in an error if the medium of the movie file is not writeable (such as in an archive file).

While this variable only applies to movies, it is stored as a global variable and can be modified even without a movie loaded. Hence, it is in the emu library rather than the movie library.

bool movie.recording()

Returns true if there is a movie loaded and in record mode.

bool movie.playing()

Returns true if there is a movie loaded and in play mode.

bool movie.ispoweron()

Returns true if the movie recording or loaded started from 'Start'. Returns false if the movie uses a save state. Opposite of movie.isfromsavestate()

```
bool movie.isfromsavestate()
```

Returns true if the movie recording or loaded started from 'Now'. Returns false if the movie was recorded from a reset. Opposite of movie.ispoweron()

```
string movie.name()
```

If a movie is loaded it returns the name of the movie, else it throws an error.

```
bool movie.readonly()
```

Returns the state of read-only. True if in playback mode, false if in record mode.

GUI Library

gui.pixel(int x, int y, type color)
gui.drawpixel(int x, int y, type color)
gui.setpixel(int x, int y, type color)
gui.writepixel(int x, int y, type color)

Draw one pixel of a given color at the given position on the screen. See drawing notes and color notes at the bottom of the page.

gui.getpixel(int x, int y)

Returns the separate RGBA components of the given pixel set by gui.pixel. This only gets LUA pixels set, not background colors.

Usage is local r,g,b,a = gui.getpixel(5, 5) to retrieve the current red/green/blue/alpha values of the LUA pixel at 5x5.

See emu.getscreenpixel() for an emulator screen variant.

gui.line(int x1, int y1, int x2, int y2 [, color [, skipfirst]]) gui.drawline(int x1, int y1, int x2, int y2 [, color [, skipfirst]])

Draws a line between the two points. The x1,y1 coordinate specifies one end of the line segment, and the x2,y2 coordinate specifies the other end. If skipfirst is true then this function will not draw anything at the pixel x1,y1, otherwise it will. skipfirst is optional and defaults to false. The default color for the line is solid white, but you may optionally override that using a color of your choice. See also drawing notes and color notes at the bottom of the page.

```
gui.box(int x1, int y1, int x2, int y2 [,
fillcolor [, outlinecolor]]))
gui.drawbox(int x1, int y1, int x2, int y2 [,
fillcolor [, outlinecolor]]))
gui.rect(int x1, int y1, int x2, int y2 [,
fillcolor [, outlinecolor]]))
```

gui.drawrect(int x1, int y1, int x2, int y2 [, fillcolor [, outlinecolor]]))

Draws a rectangle between the given coordinates of the emulator screen for one frame. The x1,y1 coordinate specifies any corner of the rectangle (preferably the top-left corner), and the x2,y2 coordinate specifies the opposite corner.

The default color for the box is transparent white with a solid white outline, but you may optionally override those using colors of your choice. Also see drawing notes and color notes.

gui.text(int x, int y, string str [, textcolor
[, backcolor]])
gui.drawtext(int x, int y, string str [,
textcolor [, backcolor]])

Draws a given string at the given position. textcolor and backcolor are optional. See 'on colors' at the end of this page for information. Using nil as the input or not including an optional field will make it use the default.

gui.parsecolor(color)

```
Returns the separate RGBA components of the given
color.
For example, you can say local r,g,b,a =
gui.parsecolor('orange') to retrieve the
red/green/blue values of the preset color orange. (You
could also omit the a in cases like this.) This uses
the same conversion method that FCEUX uses internally
to support the different representations of colors
that the GUI library uses. Overriding this function
will not change how FCEUX interprets color values,
```

however.

gui.savescreenshot()

Makes a screenshot of the FCEUX emulated screen, and saves it to the appropriate folder. Performs identically to pressing the Screenshot hotkey.

string gui.gdscreenshot()

Takes a screen shot of the image and returns it in the form of a string which can be imported by the gd library using the gd.createFromGdStr() function.

This function is provided so as to allow FCEUX to not carry a copy of the gd library itself. If you want raw RGB32 access, skip the first 11 bytes (header) and then read pixels as Alpha (always 0), Red, Green, Blue, left to right then top to bottom, range is 0-255 for all colors.

Warning: Storing screen shots in memory is not recommended. Memory usage will blow up pretty quick. One screen shot string eats around 230 KB of RAM.

gui.gdoverlay([int dx=0, int dy=0,] string str [, sx=0, sy=0, sw, sh] [, float alphamul=1.0]) gui.image([int dx=0, int dy=0,] string str [, sx=0, sy=0, sw, sh] [, float alphamul=1.0]) gui.drawimage([int dx=0, int dy=0,] string str [, sx=0, sy=0, sw, sh] [, float alphamul=1.0])

Draws an image on the screen. gdimage must be in truecolor gd string format.

Transparency is fully supported. Also, if alphamul is specified then it will modulate the transparency of the image even if it's originally fully opaque. (alphamul=1.0 is normal, alphamul=0.5 is doubly transparent, alphamul=3.0 is triply opaque, etc.)

dx,dy determines the top-left corner of where the image should draw. If they are omitted, the image will draw starting at the top-left corner of the screen.

gui.gdoverlay is an actual drawing function (like gui.box and friends) and thus must be called every frame, preferably inside a gui.register'd function, if you want it to appear as a persistent image onscreen.

Here is an example that loads a PNG from file, converts it to gd string format, and draws it once on the screen: local gdstr = gd.createFromPng("myimage.png"):gdStr() gui.gdoverlay(gdstr)

gui.opacity(int alpha)

Scales the transparency of subsequent draw calls. An alpha of 0.0 means completely transparent, and an alpha of 1.0 means completely unchanged (opaque). Noninteger values are supported and meaningful, as are values greater than 1.0. It is not necessary to use this function (or the less-recommended gui.transparency) to perform drawing with transparency, because you can provide an alpha value in the color argument of each draw call. However, it can sometimes be convenient to be able to globally modify the drawing transparency.

gui.transparency(int trans)

Scales the transparency of subsequent draw calls. Exactly the same as gui.opacity, except the range is different: A trans of 4.0 means completely transparent, and a trans of 0.0 means completely unchanged (opaque).

function gui.register(function func)

Register a function to be called between a frame being prepared for displaying on your screen and it actually happening. Used when that 1 frame delay for rendering is not acceptable.

```
string gui.popup(string message [, string type
= "ok" [, string icon = "message"]])
string input.popup(string message [, string
type = "yesno" [, string icon = "question"]])
```

Brings up a modal popup dialog box (everything stops until the user dismisses it). The box displays the message tostring(msg). This function returns the name of the button the user clicked on (as a string).

type determines which buttons are on the dialog box, and it can be one of the following: 'ok', 'yesno', 'yesnocancel', 'okcancel', 'abortretryignore'. type defaults to 'ok' for gui.popup, or to 'yesno' for input.popup.

icon indicates the purpose of the dialog box (or more specifically it dictates which title and icon is displayed in the box), and it can be one of the following: 'message', 'question', 'warning', 'error'. icon defaults to 'message' for gui.popup, or to 'question' for input.popup.

Try to avoid using this function much if at all, because modal dialog boxes can be irritating.

Linux users might want to install xmessage to perform the work. Otherwise the dialog will appear on the shell and that's less noticeable.

Bitwise Operations

All the following functions are left for backward compatibility, since <u>LuaBitOp</u> is embedded in FCEUX.

int AND(int n1, int n2, ..., int nn)

Binary logical AND of all the given integers. This function compensates for Lua's lack of it.

int OR(int n1, int n2, ..., int nn)

Binary logical OR of all the given integers. This function compensates for Lua's lack of it.

int XOR(int n1, int n2, ..., int nn)

Binary logical XOR of all the given integers. This function compensates for Lua's lack of it.

int BIT(int n1, int n2, ..., int nn)

Returns an integer with the given bits turned on. Parameters should be smaller than 31.

Appendix

On drawing

A general warning about drawing is that it is always one frame behind unless you use gui.register. This is because you tell the emulator to paint something but it will actually paint it when generating the image for the next frame. So you see your painting, except it will be on the image of the next frame. You can prevent this with gui.register because it gives you a quick chance to paint before blitting.

```
Dimensions & color depths you can paint in:
--320x239, 8bit color (confirm?)
256x224, 8bit color (confirm?)
```

On colors

Colors can be of a few types. Int: use the a formula to compose the color as a number (depends on color depth) String: Can either be a HTML colors, simple colors, or internal palette colors. HTML string: "#rrggbb" ("#228844") or #rrggbbaa if alpha is supported. Simple colors: "clear", "red", "green", "blue", "white", "black", "gray", "grey", "orange", "yellow", "green", "teal", "cyan", "purple", "magenta". Array: Example: {255,112,48,96} means {red=255, green=112, blue=48, alpha=96} Table: Example: {r=255,g=112,b=48,a=96} means {red=255, green=112, blue=48, alpha=96} Palette: Example: "P00" for Palette 00. "P3F" for palette 3F. P40-P7F are for LUA.

For transparancy use "clear".

2008

This help file has been generated by the freeware version of HelpNDoc

Commands

(written by qFox) Introduction

Lua is a scripting language. It is used in games like Farcry and World of Warcraft (and many other games and applications!). Even though you can find all kinds of tutorials online, let me help you with the basics.

I will assume you are at least somewhat familiar with the basics of programming. So basic stuff like arrays, variables, strings, loops and if-then-else and branching are not explained here.

A hello world EmuLua program looks like this:

```
while (true) do
gui.text(50,50,"Hello world!");
emu.frameadvance();
```

end;

When you load the script, the emulator will sort of go into pause mode and hand controls over to Lua (you!). Hence you are responsible for frameadvancing the emulator.

IF YOU DO NOT CALL emu.frameadvance AT THE CYCLE OF THE MAIN LOOP YOU WILL FREEZE THE EMULATOR! There. You have been warned. Don't worry though, you'll make this mistake at least once. Just force-quit the application and try again :)

Syntax

Now then. Just like any other language, Lua has a few quirks you should be aware of.

First of all, if's require a then and end. After a couple of days intensive Lua coding, I still make this mistake myself, but the Lua interpreter will prompt you of such errors on load, so don't worry too much about it. So: if (something) then dostuff end:

Lua uses nil instead of null.

There are only two values that evaluate to "false", these are "nil" and "false". ANYTHING else will evaluate to true, even 0 or the empty string.

Comments are denoted by two consecutive dashes; --. Anything after it on the same line is a comment and ignored by Lua. There is no /* */ type of commenting in Lua.

Variables have a local and global scope. You explicitly make a variable local by declaring it with the "local" keyword.

somethingglobal; -- accessible by any function or flow local something; -- only known to the same or deeper scope as where it was declared

Note that variables declared in for loops (see below) are always considered local.

Arrays are called tables in Lua. To be more precise, Lua uses associative arrays.

Do not rely on the table.length() when your table can contain nil values, this function stops when it encounters a nil value, thus possibly cutting your table short.

One experienced programmers will have to get used to is the table offset; tables start at index 1, not 0. That's just the way it is, deal with it.

There are a few ways to create a table:

local tbl1 = {}; -- empty table

local tbl2 = {"a","b","c","d"}; -- table with 5 strings local tbl3 = {a=1,b=2,c=3}; -- associative table with 3 numbers local tbl4 = {"a",b=2,c="x","d"=5}; -- associative table with mixed content

Note that you can mix up the data in one table, as shown by tbl4.

You can refer to table values in a few equivalent manners, using the examples above:

tbl1[1] -- = nil because tbl1 is empty tbl2[2] -- = "b" tbl3["a"] -- = 1 tbl4.b -- = 2 tbl2.3 -- = "c"

When the argument of a function is just a table, the parantheses "()" are optional. So for instance:

processTable({a=2,b=3});

Is equivalent to

processTable{a=2,b=3};

Another notation that's equivalent is

filehandle.read(filehandle, 5);
filehandle:read(5);

When using the colon notation ":" Lua will call the function adding the self-reference to the front of the parameterstack.

Functions behave like objects and are declared in the follow manner:

function doSomething(somevalue, anothervalue) dostuffhere end;

```
So no curly braces "{}" !
```

Some flow control:

for i=0,15 do -- do stuff here, i runs from 0 to 15 (inclusive!) end;

for key,value in pairs(table) do -- do stuff here. pairs will iterate through the table, splitting the keys and values end;

while (somethingistrue) do

end;

if (somethingistrue) then

end;

if (somethingistrue) then

else

end;

if (somethingistrue) then

elseif (somethingelseistrue) then

end;

For comparison, you only have to remember that the exclamationmark is not used. Not equal "!=" is written like tilde-equals "~=" and if (!something) then ... is written with "not " in front of it; if (not something)

then...

For easy reference to the standard libraries look on the bottom half of this page: http://www.lua.org/manual/5.1/

Lua in FCEUX

Now then, let's get to the emulator specifics!

To load a Lua script in FCEU first load a rom (Lua can only do things after each frame cycle so load a rom first). Go to file, at the bottom choose Run Lua Script and select and load the file.

When Lua starts, the emulator pauses and hands control over to Lua. Lua (that's you!) decides when the next frame is processed. That's why it's very common to write an endless while loop, exiting the main loop of a script will exit the script and hand control back to the emulator. This also happens when a script unexpectingly crashes.

A bare script looks like this:

```
while (true) do
  emu.frameadvance();
end;
```

And is about equal to not running Lua at all. The frameadvance function is the same called internally, so no loss of speed there!

Bitwise operators:

Lua does not have bitwise operators, so we supply some for you. These are common bitwise operators, nothing fancy.

```
AND(a,b);
OR(a,b);
XOR(a,b);
BIT(n); -- returns a number with only bit n set (1)
```

The emulator specific Lua is equal to the one of snes9x, with some platform specific changes (few buttons, for instance).

You can find the reference here: http://dehacked.2y.net/snes9x-lua.html The following is a quick reference, you can go to the snes9x reference for more details.

To paint stuff on screen, use the gui table. This contains a few predefined functions to manipulate the main window. For any coordinate, 0,0 is the top-left pixel of the window. You have to prevent out-of-bound errors yourself for now. If a color can be passed on, it is a string. HTML-syntax is supported ("#34053D"), as well as a FEW colors ("red", "green", "blue" ...).

gui.text(x, y, str); -- Print a line to the window, you can use \n for a return but it will only work once

gui.pixel(x, y, color); -- plot a pixel at the given coordinate gui.line(x1, y1, x2, y2, color); -- plot a line from x1,y1 to x2,y2 gui.box(x1, y1, x2, y2, color); -- draw a square from x1,y1 to x2,y2 gui.popup(str); -- pops up a messagebox informing the user of something. Real handy when debugging! gui.getpixel(x,y); -- return the values of the pixel at given position. Returns three numbers of the emulator image before paiting is applied. gui.gdscreenshot(); -- Takes a screen shot of the image and returns it in the form of a string which can be imported by the gd library using the gd.createFromGdStr() function

(for more gd functions see DeHackED's reference:

http://dehacked.2y.net/snes9x-lua.html)

PAINTING IS ALWAYS ONE FRAME BEHIND! This is because the painting is done at the creation of the next frame, not while Lua is running.

Emulator control:

emu.frameadvance(); -- advances emulation ONE frame emu.pause(); -- same as pressing the pause button emu.speedmode(strMode); -- Supported are "normal","turbo","nothrottle","maximum". But know that except for "normal", all other modes will run as "turbo" for now. emu.wait(); -- skips the emulation of the next frame, in case your script needs to wait for something

Memory control:

memory.readbyte(adr); -- read one byte from given address and return it. Besides decimal values Lua also allows the hex notation 0x00FA. In FCEUX reading is done BEFORE the cheats are applied! memory.writebyte(adr, value); -- write one byte to the RAM of the NES. writing is done AFTER the hexeditor receives its values, so if you are freezing an address by Lua, it will not show in the hex editor (but it will in the game :)

memory.readbytesigned(adr); -- same as readbyte, except this returns a signed value, rather then an unsigned value.

memory.register(adr, function); -- binds a function to an address. The function will be called when an address changes. NOTE THAT THIS IS EXPENSIVE (eg.: slow)! Only one function allowed per address.

Input control:

You can read and write input by using the joypad table. A input table has the following (case sensitive) keys, where nil denotes they are not to be pressed: up down left right start select A B

joypad.read(playern); -- get the input table for the player who's input you want to read (a number!)

joypad.write(playern, inputtable); -- set the input for player n. Note that this will overwrite any input from the user, and only when this is used.

Savestates:

You can load and save to the predefined savestates 1 ... 9 or create new "anonymous" savestates. You must first create a savestate object, which is your handle to a savestate. Then you can pass this handle on to

savestate.load or save to do so.

savestate.create(n); -- n is optional. When supplied, it will create a savestate for slot n, otherwise a new (anonymous) savestate object is created. Note that this does not yet save or load anything! savestate.load(state); -- load the given savestate savestate.save(state); -- save the given savestate

For an up-to-date list of functions, see the Lua Functions List.

2008

This help file has been generated by the freeware version of <u>HelpNDoc</u>

Lua Bot

LuaBot

LuaBot employs a new concept in FCEUX Tool creation. It is an external lua script that creates the Basic bot GUI. The GUI then uses lua scripting to perform botting tasks.

To run it you must have lua scripting enabled (see <u>Getting Started</u>). LuaBot is included in the lua pack under /luaScripts. to get started run luabot_framework.lua.

What is Lua Bot?

LuaBot is...well, a bot. It uses a combination of probability, scripting and RAM monitoring to play games. Specifically basic bot is used to create portions of <u>Tool Assisted Speedrun</u>. It is most powerful for finding solutions in highly random situations, or highly improbably events (such as manipulating a critical hit in an RPG). Basic bot comes with a rather powerful scripting language in order to be "programmed" to handle these specific situations. LuaBot in its most extreme application can even be "taught" to play video games!

How to Use Lua Bot

LuaBot is a trial and error script that exhausts the input-search-space by simply trying to push buttons.

You can program it to limit this searchspace, as it can become exponentially large. You can press eight possible buttons at any frame, each on or off. That's 2 raised to the 8, or 256 possible combinations in that one frame. There are 60 frames in one second, so you have 256 raised to the power of 60. Write a three. Now start writing 144 zeroes after it. It's not a small number.

Anyways, the bot has two parts. The frontend, which we'll call BeeBee,

and the Lua part, which we call LuaBot.

You start the bot by opening the LuaBot_front.lua script file. Make sure the LuaBot_backend.lua file is in the same directory.

BeeBee

BeeBee (who received it's name from BasicBot, its predecessor) just writes it's contents into the LuaBot framework and produces a big Lua script for you.

All you need to do is enter Lua code for the specific functions and the code will generate the script.

You can also save and load the contents of the front-end. That way you can easily manage your bot scripts, without actually having to look into the LuaBot code.

BeeBee is only a pasting mechanism. It does not compile Lua or warn for errors.

LuaBot

LuaBot is a generic trial-and-error script that serves as a bot framework. It will set inputs as you program them for a number of frames (called an attempt). When the isAttemptEnd() says the attempt ends, a new attempt is started. All the attempts fall under one segment. At the end of a segment (denoted by the isSegmentEnd() function), the best attempt is kept (judged by the score and tie functions) and the next segment is started. The bot is capable of rolling back if a segment runs into a dead end. This allows you to backtrack and restart a previous segment.

The bot evaluates a true or false by checking to see whether the return value of a function is bigger then a certain value. It does this for EVERY function that returns something and every function that returns something must return a number (or Lua _will_ complain). For absolute true or false you can return "yes" and "no", "maxvalue" and "minvalue" or "pressed" and "released". Read variable info for more information.

The script takes a number of variables and functions into account. Some variables become important to prevent desyncing over segments.

- maxvalue

The maximum value (exclusive) of the random evaluation. If a value is higher than rand(minvalue, maxvalue), it evaluates as true, else false. By default this is set to 100.

- minvalue

The lowest value (inclusive) of the random evaluation. If a value is lower than rand(minvalue, maxvalue), it evaluates to false, else true. By default this is set to 0.

- yes / no

- pressed / released

These map to the minvalue/maxvalue.

- loopcounter

The number of times a frameadvance has been called by the main botloop.

- key1 key2 key3 key4

The input table of players 1-4. The keys are: A B up down left right select start. Set any to 1 if you want them to be set and to nil if you don't want them set.

Note that these get cleared right before onInputStart is called. This variable is saved in a pseudo-movie table if the attempt is better then the previous one and used for playback when moving to the next segment.

- lastkey1 lastkey2 lastkey3 lastkey4

The inputs that were given to FCEU on the PREVIOUS frame. This holds for segments as well (at the beginning of a new segment, the lastkeys of the previous segment are set). This also goes for the start. If you use key1-4 in onStart, the first segment will have those keys as lastkey.

- frame

The number of frames of the current attempt. Starts at 1.

- attempt

The number of attempts in the current segment. Starts at 1.

- segment

The segment the bot is currently running. Note that rolledback segments are deducted from this number.

- okattempts

The number of attempts that have been deemed ok. This is a statistical variable. It might tell you how well your bot is doing (combined with the number of failed attempts).

- failattempts

The number of attempts in the current segment that have been deemed bad. This is a statistical variable. It might tell you how well your bot is doing (combined with the number of approved attempts).

- segments

This is the big table that holds everything together. Don't mess with it.

- maxframes

You can set maxframes and check it in the isAttemptEnd function to simply limit a attempt by this many frames. You can also just ignore this and do something else instead.

- maxattempts

Same as maxframes, except for attempts in a segment.

- maxsegments

Same as maxframes, except for segments in a run.

- playingbest

Will be set to true when the bot is playing back it's best attempt to advance to the next segment. Not really used by other functions.

- keyrecording1-4

A simple table with the pressed keys for playback.

-XYZPQ

Some "static" variables. These allow you to easily set them onStart and use them in various functions to return the same number. Like a global variable. The P and Q numbers used to denote a random number between 0 and P or Q, but they don't right now.

- vars

This is your variable table. It's contents is saved at the end of an attempt and will be loaded at the beginning of a segment. On rollback, this table is also kept. Put any variable you want to keep across segments in this table.

Ok. That's it for the variables. Now for functions. There are basically three types of functions. The functions that determine whether a button is pressed (8 for each player), to determine whether an attempt/segment/run has ended or was ok and functions for certain events. This number is not evaluated by the random-eval function.

- getScore

This returns how "well" the current attempt is. At the end of a segment, the best scoring good attempt will be used to continue to the next segment. In case of a tie, see the tie functions. This number is not evaluated by the random-eval function!

- getTie1-4

If the score ends in a tie, that is, two attempts score equally well (have an equal number of points for instance), you can use these functions to break that tie. Like, which attempt has the most health or is the fastest or whatever. This number is not evaluated by the random-eval function!

- isRunEnd

Return whether the bot should stop running. If the returned number is bigger then the random number rand(minvalue-maxvalue), the bot will

stop.

- mustRollBack

Returns whether the bot should rollback the current attempt. In such case, the previous segment is loaded and the current segment is completely discarded. If the returned number is bigger then the random number rand(minvalue-maxvalue), the segment will rollback one segment.

- isSegmentEnd

If the returned number is bigger then the random number rand(minvaluemaxvalue), the bot will stop the current segment, play back the best recorded attempt and start a new segment. Mostly done when a certain number of attempts is reached, but possibly you know when have the best possible attempt and can move on.

- isAttemptEnd

If the returned number is bigger then the random number rand(minvaluemaxvalue), the attempt will stop and a new attempt will be started. Some examples when this function should return yes is when you reached a certain goal, a number of frames or when you died (in which case the bot should try again :).

- isAttemptOk

If the returned number is bigger then the random number rand(minvaluemaxvalue), the current attempt (which has just ended) is deemed ok. Only attempts that are deemed ok are up for being saved. For instance, when the player died in the current attempt, you should return no.

- pressKeyX (pressKeyA1, pressKeyStart4, etc...)

These functions determine whether a button should be pressed in the next frame. If the returned number is bigger then the random number rand(minvalue-maxvalue), the button is pressed, otherwise it is not. To absolutely press a button, simply return yes or no. To use some odds, return a number between minvalue and maxvalue. For instance, using the default settings, if you return 50, there is a 50% chance the button will be pressed.

- onStart

Maybe a little misleading, but the onStart function is called BEFORE the main botloop starts. You can do some non-generic startup stuff here like press start at the title screen and get the game started. Returns nothing.

- onFinish

The opposite to onStart, this function is called when the main botloop exits. You can cleanup, or write stuff or whatever.

- onSegmentStart

When a new segment is started, this is called. After initializing variables and such, but before onAttemptStart is called. Returns nothing.

- onSegmentEnd

When isSegmentEnd evaluates to true, this function is called. Returns nothing.

- onAttemptStart

Called at the start of a new attempt, after onSegmentStart (in case of a new segment) but before onInputStart. Returns nothing.

onAttemptEnd(wasOk)

Called at the end of an attempt. The only function to have a parameter (note: case sensitive). The parameter wasOk will return (boolean) whether isAttemptOk evaluated to true or false. Returns nothing.

- onInputStart

In a frame, this is the first place where the key1-4 variables are cleared. This is called before all the input (pressKeyX) functions are called. Returns nothing.

- onInputEnd

This is called immediately after the input (pressKeyX) functions have been called. Returns nothing.

This help file has been generated by the freeware version of <u>HelpNDoc</u>

FAQ / Guides

FAQ / Guides

Information regarding various concepts such as TAS, ROM Hacking, RAM Mapping.

Troubleshooting FAQ

A guide to common problems people experience, and what to do about them.

Tool Assisted Speedruns (TAS)

Information regarding Tool Assisted Speedruns and the TAS community.

ROM Hacking

Information regarding making ROM Hacks and the ROM Hacking community.

NES RAM Mapping

A guide to the layout of NES RAM, and how to interpret its contents.

Debugger Usage Guide (Intermediate)

This is a guide that explains some of the debugging features in terms that someone with previous experience with assembly can understand, and delves into the most basic understanding to a degree as well. Likely won't help a beginner too much.

2008

This help file has been generated by the freeware version of <u>HelpNDoc</u>

Troubleshooting

Troubleshooting

This section describes potential problems/question that could arise when using FCEUX.

Slow emulation / Sound crackle

FCEUX may not run well on slower CPU's. You can improve performance by setting sound to low quality in the <u>Sound Dialog</u>. In addition, for windowed mode try enabling hardware acceleration in the <u>Video config</u> dialog may also help.

Slow savestates when recording movies

On slower computers, savestates can be slow with long movies. A small speedup can be done by disabling config > enable > Backup savestates.

"The colors in game X do not look right"

FCEUX uses the color palette of the old FCEU / FCEUXD branches. However, FCEUX comes pre-packaged with several additional color palettes. For more information see <u>Palette config</u> and <u>Palette options</u>.

"FCEUX won't open my ROM/File. It let's me select it in the Open File dialog then ignores my selection"

It maybe be that the filename or pathname includes one or more on ASCII characters. Currently FCEUX does not support Unicode characters.

"I converted a .fcm file to .fm2 but the .fm2 desyncs"

Depending on what version of FCEU / Game your .fcm was made, there maybe a number of sync issues. These are listed in detail at http://tasvideos.org/ConvertFCMtoFM2.html.

In addition, the .fm2 conversion tool has had some issues on certain operating systems including Vista and Mac.

"When I attempt to load a Lua script I get an error / crash"

Double check you have the Lua .dll files that came packaged with FCEUX 2.1. They must be in the /dll folder from the root directory (where fceux.exe is stored).

"Directdraw: Error creating secondary surface"

Currently this error will happen when attempting to do Full screen mode on Windows Vista. In addition, there has been one reported case of this happening on the 32-bit version of Windows XP. This is a known issue with FCEUX that has not yet been resolved.

"Can't find FDS Bios image when I attempt to load a .fds game!"

FCEUX requires the FDS Bios to be named disksys.rom. It must be located in the root directory (where fceux.exe is stored) or in the folder of the FDS Directory override (see <u>Directory overrides</u>).

In addition, there are some bad versions of disksys.rom. The one FDS requires is 8192 bytes.

"How can I use Netplay / Where can I get FCEU Server?"

Currently FCEUX is not compatible with the FCEU-server code. This is a

known issue and will be resolved in a future release.

"I have a Game Genie rom, how can I use it with FCEUX?"

While FCEUX has a <u>Game Genie code converter</u>, you can also use game genie codes with the game genie rom. It must be named gg.rom and be placed in the root directory (where fceux.exe is stored). You must also check NES > Game Genie in the main menu.

2008

This help file has been generated by the freeware version of <u>HelpNDoc</u>

Tool Assisted Speedruns What is Tool Assisted Speedrunning?

A tool-assisted speedrun (commonly abbreviated TAS) is a speedrun movie or performance produced with the use of tools such as slow motion and re-recording. The basic premise of these runs is that a "tool" (such as an emulator that provides the author with features that are unavailable in regular playing) is used in order to overcome human limitations such as skill and reflex.

Creating a tool-assisted speed run is the process of finding the ideal set of inputs to complete a given criterion - usually completing a game as fast as possible. No limits are imposed on the tools used for this search, but the result has to be a set of timed key-presses that, when played back on the actual console, achieves the target criterion. Traditionally, the only available tool for this was an emulator with re-recording - the ability to use savestate while recording key-presses. However, due to advances in the field, it is now often expected that frame-advance, stepping through emulation one frame at a time, is used. A tool-assisted speed run done without this technique may be criticised as "sloppy play". Before frameadvance became common, playing in slow motion was a common technique, but frame-advance has displaced this.

In essence, Tool Assistance allows one to overcome human limitations of skill and reflex in order push a game to its limits. One important thing to remember is that TAS movies are not competing in terms of playing skill, nor do they claim to.

For more info on Tool Assisted Speedruns: http://tasvideos.org/ http://tasvideos.org/WhyAndHow.html http://en.wikipedia.org/wiki/Tool-assisted_speedrun

FCEUX TAS features

FCEUX provides a wealth of tools and resources for creating TAS Movies for NES & FDS games. It features the most current and cutting edge tools for optimizing movies and making the process of movie making quicker an easier.

Basic Recording features: <u>Frame advance, Slow-downs, "bullet-proof" rerecording, TAS edit</u>

Advanced Recording features Input presets, Auto Hold & Auto-Fire

Automated Movie Making Processes <u>Macros & Multi-tracking</u>, <u>Lua scripting</u>, <u>Basic Bot</u>,

Finding Ram values: Cheat Search, RAM Filter, Hex Editor, Debugger, NES RAM Guide

RAM Monitoring: <u>Memory Watch</u>, <u>Hex Editor</u>

Movie Splicing Editing Text based file format

Movie making

To get started making a Tool Assisted Movie, simply begin recording a movie (see <u>Movie Recording</u>). The basic premise of TASing, however, is to use re-records to optimize the execution of a decided upon goal (usually to complete the game as fast as possible). Re-recording is the act of replacing an already recorded part (of a movie) with something else; also called undo.

In the making of emulator movies, re-recording is done by loading a

savestate of earlier event in the movie and continuing playing from that point. The emulator will update the movie file to undo everything that was cancelled by the savestate loading, and continue recording from that point. The makers of tool-assisted speedruns use re-recording very extensively to reach perfection and to avoid mistakes.

* In single-segment non-assisted speedruns, re-recording is starting over from beginning. The recording of the failed playing is usually not preserved.

* In multi-segment non-assisted speedruns, re-recording is starting over from the beginning of current segment. The recording of the failed segment is not preserved.

* In tool-assisted speedruns, re-recording only undoes a small part of playing. The undone part will not be seen in the resulting movie. A tool-assisted movie may have been re-recorded anything between 50 and 200000 times, depending on the precision of the movie and the difficulty of the game. Often, the same small passage of the game (could be as small as fractions of second long) is attempted tens of times before continuing.

For more info on making TAS movies: http://tasvideos.org/CommonTricks.html http://tasvideos.org/GenericTips.html

2008

This help file has been generated by the freeware version of HelpNDoc

ROM Hacking

ROM Hacking What is ROM Hacking?

ROM Hacking is the process of modifying a video game ROM image to alter the game's graphics, dialogue, levels, gameplay, or other gameplay elements. This is usually done by technically-inclined video game fans to breathe new life into a cherished old game, as a creative outlet, or to make essentially new unofficial games using an old game as a foundation.

ROM hacking is generally accomplished through use of a hex editor (a program for editing non-textual data) and various specialized tools such as tile editors, and game-specific tools which are generally used for editing levels, items, and the like, although more advanced tools such as assemblers and debuggers are occasionally used. Once ready, they are usually distributed on the Internet for others to play on an emulator.

For more information on ROM Hacking: <u>"The ROM Hackers Bible"</u> <u>Parodius - ROM Hacking Community</u> <u>ROM-Hacking.net Archive</u>

FCEUX ROM-Hacking Features

FCEUX provides a wealth of tools and resources to aid in hacking NES & FDS games. It features the most current and cutting edge tools debugging and hacking games as well as making the process quicker an easier.

Debugging / Reverse engineering: Debugger, Trace Logger, Code/Data Logger, Cheat Search, RAM Filter, Movie Making tools/Frame Advance

Memory & PPU Viewing:

Debugger, PPU Viewer, Hex Editor, Trace Logger, Code/Data Logger

ROM Editing / Game Genie Codes

Hex Editor, Cheat Search, Game Genie Decoder/Encoder

2008

This help file has been generated by the freeware version of <u>HelpNDoc</u>

NES RAM (Mapping/Finding Values)

NES Mapping

This guide gives a map of the addresses in the NES cpu and explains each portion in detail.

It also provides information for the basic layout of ram values in typical NES games. This info can be used to quickly map and find useful values in the game's ram.

Contents

Memory Map

Gives a diagram of the 2A03 CPU memory map .

2C02 PPU memory map

Gives more detailed info about each section of the Memory map diagram

Game Ram Details

On board RAM Map (\$000-\$07FF) Map (gives specific info on the how NES games typically layout their ram values)

Online Resources

<u>NES Programming - Wikipedia</u> <u>NES Memory Map</u>

Memory Map (NES RAM/ROM)

2A03 CPU memory map

2A03 CPU is a 6502-compatible CPU without the decimal mode (CLD and SED do nothing). It has an on-die sound generator, very limited DMA

capability, and an input device controller that can be accessed through the 2A03 registers.

| 6502 CPU Mer | mory | |
|---|---|---|
| Map Address Range <u>(Hexadecimal)</u> | Size in bytes | Notes (Page size = 256bytes) |
| \$0000 - \$07FF | 2048 | Game Ram |
| (\$0000 - \$00FF) addressing modes g (\$0100 - \$01FF) (\$0200 - \$07FF) | 256 ive faster memory 256 1536 | Zero Page - Special Zero Page read/write access Stack memory RAM |
| \$0800 - \$0FFF | 2048 | Mirror of \$0000-\$07FF |
| (\$0800 - \$08FF) (\$0900 - \$09FF) (\$0A00 - \$0FFF) | 256 256 Stac 1024 | Zero Page ck Ram |
| \$1000 - \$17FF | 2048 bytes | Mirror of \$0000-\$07FF |
| (\$1000 - \$10FF) \$1100 - \$11FF \$1200 - \$17FF | 256 256 5 1024 | Zero Page Stack RAM |
| \$1800 - \$1FFF | 2048 bytes | Mirror of \$0000-\$07FF |
| (\$1800 - \$18FF) (\$1900 - \$19FF) | 256 256 | Zero Page Stack |

| (\$1A00 - \$1FFF) | 1024 | RAM | | | | |
|--|----------------------|---|--|--|--|--|
| \$2000 - \$2007 \$2008 - \$3FFF times) | 8 bytes 8184 byte | Input / Output registers Mirror of \$2000-\$2007 (mulitple | | | | |
| \$4000 - \$401F32 bytesInput / Output registers\$4020 - \$5FFF8160 bytesExpansion ROM - Used withNintendo's MMC5 to expand the capabilities of VRAM. | | | | | | |
| \$6000 - \$7FFF save data between g | | es SRAM - Save Ram used to | | | | |
| \$8000 - \$BFFF executable code \$C000 - \$FFFF executable code | 16384 by 16384 by | | | | | |
| | 2 bytes e | Address of Non Maskable Interrupt | | | | |
| \$FFFC - \$FFFD routine | 2 | Address of Power on reset handler | | | | |
| \$FFFE - \$FFFF instruction) handler ı | , | Address of Break (BRK | | | | |

2C02 PPU memory map

2C02 PPU is a character generator with sprites, designed by Nintendo specifically for the NES.

| I |
|---|
| 1 |
| I |
| |
| |
| |
| |

The NES PPU has enough RAM for two nametables (0 and 3); it brings some PPU nametable address lines to the cart edge so that the cart can decide whether to map 0 onto 2 and 1 onto 3 (vertical mirroring as in Super Mario Brothers and Contra) or 0 onto 1 and 2 onto 3 (horizontal mirroring as in Kid Icarus and Ikari), all screens to either 0 or 3 (as in many Rare games such as Battletoads and Jeopardy!), or all screens to RAM on the cartridge (as in Gauntlet). Split-screen games that scroll in all four directions (such as Super Mario Brothers 3 and Kirby's Adventure) often use vertical or one-screen mirroring (with a small amount of screen corruption at the sides due to tiles wrapping around the sides) and stick the status bar in some random unused area of the screen.

Game RAM Details

Mapping RAM/Finding Ram Written by: adelikat This guide is written specifically for finding useful values for TAS <u>movie</u> <u>making</u>.

It does not tell you how to use specific tools to find values. For that refer to <u>Hex editor</u>, <u>Cheat Search</u>, and <u>RAM filter</u>.

Most games use the basic on board ram. The address range of this ram is \$0000-\$07FF. This translates to 2048 possible ram values.

Pages

This ram is broken down into 8 pages. A "page" is a block of 256 ram values.

I will refer to these values as such:

| Block 0 | \$00xx | (\$0000-\$00FF) |
|---------|--------|-----------------|
| Block 1 | \$01xx | (\$0100-\$01FF) |
| Block 2 | \$02xx | (\$0200-\$02FF) |
| Block 3 | \$03xx | (\$0300-\$03FF) |
| Block 4 | \$04xx | (\$0400-\$04FF) |
| Block 5 | \$05xx | (\$0500-\$05FF) |
| Block 6 | \$06xx | (\$0600-\$06FF) |
| Block 7 | \$07xx | (\$0700-\$07FF) |

Each block will be organized will similar data. For instance, all sprite data will be in the same block. Enemy/Player statistics (energy, coordinates, speed, etc.) will be in another. For instance, if you find the main character's HP and it is located in block 3, you know that the remaining stats for the character are also in that block. This can significantly cut down time when trying to find related values.

There are always the following blocks:

Sprite Data Block 2

I've yet to see map a game that does not use this block solely for sprite data. It will contain the "ID" numbers for all the items currently on the

screen. Simply put, this data is precisely the data you see on the screen. For making TAS movies this is not useful data. If you are using cheat search and have narrowed it down your search to a few values, you can immediately discard any \$02xx values.

In games with a lot of sprite data, I've seen blocks 1 & 3 also reserved for sprite data.

Music & Sound FX Block 1 or 7, generally

This one has more deviation, but almost all games reserve an entire block for memory allocated to the game's Music and Sound FX. Again, for TAS purposes these values are not *useful. By finding even 1 of these values, you can eliminate that block from your search possibilities. Finding which block is reserved for music is often quite simple with the <u>Hex editor</u>. Watching the ram values with the game playing, you can see which addresses "move to the beat".

*Actually they can come in handy for "dancing to the beat"

Player & Enemy Stats Blocks 1,3,4,5 generally (any or all of these)

This is your "sweet spot" for movie making, as often you will be wanting to track the players speed or coordinates, enemy energy, or enemy coordinates.

These values rarely (if at all) reside outside blocks 1, 3, 4, or 5. This knowledge already reduces your search possibilities in half!

Rows

Each block is broken down into 16 "rows" of addresses. For example, in block 3, the first row is \$030x (\$0300-\$030F).

Each row of 16* will contain similar data. For instance all x coordinates will generally be in the same row. So xxx0 might be the main characters x position. xxxx1 would be "enemy 1" (1st enemy loaded onto the

screen), and so on.

The y coordinates would be in another row, x subpixel values in yet another row, etc.

*Super Mario Bros. 2 (U) is a rare example that uses rows of 10

Columns

A column would be all the values of a block that share the same last digit. So a column would be 16 addresses such as \$0300, \$0310, \$0320, etc.

For enemy/player stats, columns usually refer to the same player or enemy.

So for example, if a player's energy was stored in \$0300. The remaining row will be other player/enemy's energy.

If the next row (\$031x) is x positions. \$0310 would be the player's x position. The remaining positions of that row would correspond to the other player/enemy x positions in line with the hp values of the previous row.

Example

These distinctions are easier to see in a visual example. This is the enemy/player stats as they are mapped in the game *Teenage Mutant Ninja Turtles*.

Block 4

| | Р | W1 W2 W3 | <u>E1 E2 E3 E4 E5</u> |
|--|-------------|----------|-----------------------|
| <u>E6 E7 E8 X X X X</u> | <u>X_</u> | | |
| Sprite ID: 0 | 040x: 09 00 | 00 00 00 | 9E 9E 9E 9E 00 |
| 00 00 00 00 00 00 | | | |
| ID counter: 0 | 041x: 00 00 | 00 00 00 | 00 00 00 00 00 |
| $\begin{array}{cccccccccccccccccccccccccccccccccccc$ | | | |

P = current turtle (player)W = weapon (up to 3 on the screen at one time)E = enemy (up to 8 on the screen at one time)X = no use

E1 = "Enemy slot 1" which will be the first enemy on the screen loaded into memory. The 2nd will be placed in "Enemy slot 2". When enemy 1 is removed from memory (killed or goes off screen), the next enemy will be loaded into that slot. Enemy's always take the lowest available slot when loaded. Note: usually enemy slots are in reverse order. So the first addresses is usually the last enemy slot loaded into memory. TMNT is an exception.

All object (player, weapon, enemy) characteristics reside in block 4. Each row is a different characteristic of each object on the screen (040x refers to a sprite ID of an object)

Each column corresponds to a specific object on the screen. (All 04x0 's refer to the player).

See also, <u>Memory Watch</u>, <u>Hex Editor</u>, <u>Cheat Search</u>, <u>Ram Filter</u>, <u>Movie</u> <u>Making</u>, <u>Tool Assisted Speedruns</u>

2008

This help file has been generated by the freeware version of <u>HelpNDoc</u>

Debugger Usage (Intermediate)

Debugger Usage (Intermediate) Intent

This guide assumes that you have a working knowledge of how to interpret Assembly instructions, and are willing to look up the table of 6502 instructions on your own. This document describes the debugger, and to a much lesser degree, how to tell it what you want to do in terms of the assembly.

Main Window

In the main window, you'll see lines like this on the left: 1E:80E5:66 FF ROR \$00FF = #\$9A

This is the disassembly area.

On the right are Breakpoints, the stack address and partial contents, status flags, PC and registers, and several buttons and text boxes. Not all buttons and features will be described, simply because one can't write about what one hasn't used.

Of the buttons, the one's you'll likely use the most are Run, Step Into, and Step Out.

Run simply makes the program continue operating until it hits a condition that causes it to stop running.

Step Into allows you to move to the next line, and if you Step Into a JSR, you'll be taken to the address listed for it to continue running.

Step Out attempts to run until the debugger is outside of the function you Stepped Into. This doesn't always work, and may appear to make the emulator 'Run' instead. The next time you use Step Out, it will tell you it will erase the old Step Out watch to do the new one. Press OK to do so.

Other buttons include Step Over, which basically tries to run to the PC address after the JSR, Run Line, which runs a single VBlank line, 128 lines, which is the last button's action 128 times, "Seek To:" and a text box, which lets you move around the disassembly window quickly, and Seek PC, which orients you back to the code to be run without doing anything else.

Interpreting the line below is as follows: 1E:80E5:66 FF ROR \$00FF = #\$9A

(1E:) is the bank used. A bank is 0x4000 in size for a .NES file(and 0x1000 for a .NSF file, but don't worry about this for now). 80E5, masked with 3FFF, equals 00E5. 0x1E multiplied by 4000 is 0x78000. Together, this gives 0x78000 + 0x00E5, which equals the address 0x780E5. Add 0x10 to this to account for the 0x10 byte .NES header, and you get 0x780F5, which is what the address is within the .NES file. That is all that this piece of information means.

1E:(80E5) is the address that the data that follows is being read from. The banking stated above allows the same address to show different data, but (1E:80E5) will always be the same address on the same page of data.

1E:80E5:(66 FF ROR \$00FF) is the byte code for ROR, followed by the parameter that tells it to use \$00FF. After that is the interpreted assembly code. Note that just because code can be interpreted doesn't mean that it can be run. Even data that's for graphics will inevitable look like code to a certain degree to the disassembly window.

1E:80E5:66 FF ROR \$00FF(= #\$9A) tells you what the value of \$00FF is. \$00FF refers to 00FF in RAM, and the value is 0x9A. If the program points to an address in ROM instead of RAM, it still shows the value for you.

Breakpoints

The breakpoint buttons are Add, Delete, and Edit. These should be selfexplanatory. Delete removes the selected code. Edit will allow you to change an existing entry with the same window as you use to Add. Add will open an empty window, which allows you to fill in several details. Double-clicking a breakpoint will enable and disable it, so you don't have to delete items to stop them from causing the debugger to halt the game(the short term for this is called 'breaking' the debugger, referred to in the same way as one would refer to a broken vase)

In the Add Breakpoint window, you have 2 Address boxes [] - []. You must put in at least 1 value, preferable in the left box. If you want to use a range of values, you can fill in the first box as the start, and the second box as the end. The addresses can be either RAM (0000-07FF), ROM (8000-FFFF), or the special addresses(6000-7FFF, which can be extra RAM, ROM, or unused, depending on game)

The Read, Write, and Execute checkboxes allow you to specify whether to break the debugger when that address is Read From, Written To, or Executed by the CPU. All of them can be checked at the same time if you don't know how the data is used, but are sure it's used somehow. Note that if it's the value to an assembly instruction, it won't cause a break, because the byte itself isn't the start of an instruction, and isn't read by the CPU so much as used for an instruction.

The Option boxes CPU Mem, PPU Mem, and Sprite Mem tell the debugger where to look. The Hex Viewer will show you CPU Mem by default, and PPU Mem if you tell it to. PPU Mem is where data is written to display the actual images on the screen, rather than to decide what it is that should be put there. My own experience is that this won't have to be touched unless you accidentally click off of CPU Mem. PPU debugging is more of an advanced skill, primarily because when something does go wrong, it's usually because an address outside of the normal PPU address is getting written to and causing very weird graphics errors.

Name is simply a name that can be seen to the right, if there's space for it.

The forbid option allows you to exclude a range of RAM or ROM from the breakpoint list. These can be enabled and disabled as of this writing.

Condition is part of the Conditional Debugging system. This system is slightly complicated, and extremely powerful when used correctly, and can make extremely hard problems to debug turn into lazy clicking when you understand how to use the feature.

Conditional Debugging

Conditions only apply to the breakpoint they are applied to. You can breakpoint the same address multiple times with different conditions applied to each one.

Registers: A, X, Y, (P = PC) (A==#12, to break if A = 12) (P!=#804C to NOT break when the PC is on 804C) Flags: N, C, Z, I, B, V, U, D (N==#0 for off, N==#1 for on) Bank: K (K!=#1E to NOT break when the bank is 1E)

Addresses: (\$0000!=#00 makes it test RAM address 0000, and if it's NOT 00, it can break) Numbers: (#0123 is interpreted as 123. It can be used in tests for known values to exclude or include)

Numerical Comparisons: == = Equal != = Not Equal <= = Less Than or Equal >= = Greater Than or Equal < Less Than > Greater Than Math Operators:

+, -, *, / (which are add, subtract, multiply, divide) If you want to do complex math that involves 5 memory addresses, 2 ROM addresses, and a smattering of defined values, you can use these to do it. I've never had to, but if you want to, this is how.

Multiple Tests: (N==#0||A!=#0 makes it break if EITHER N is off, or A isn't 00) (A!=#0&&X==#a5 makes it break is A isn't 00, AND X is A5)

Brackets:

[#2CC + X] == #34: The value of the byte at address 2CC + X is 0x34

Parentheses:

```
(#1 + #2) = = #3 (This evaluates to true, so the breakpoint can break. If you need a different order of operation for math operators, you can use parentheses)
```

This is what's currently found in the source file for Conditional Debugging:

2008

This help file has been generated by the freeware version of HelpNDoc

Technical Information

Technical Information

These chapters deal with documentation of specific hardware configurations of the NES and/or how the FCEU core emulates these aspects.

More documentation about NES and Famicom hardware specifications can be found at: <u>http://nesdev.parodius.com/</u>

2008

This help file has been generated by the freeware version of <u>HelpNDoc</u>

Movie formats

Movie and Savestate File Formats

The Following documentation deals with the specific technical information regarding the format of movie & savestate files.

<u>.fm2</u> - FCEUX Movie file format

<u>.fcm</u> - Movie file format from previous FCEU versions (compatible with FCEUX via <u>Convert FCM</u>)

<u>.fcs</u> - Savestate file format

2008

This help file has been generated by the freeware version of <u>HelpNDoc</u>

.fm2

FCEUX Movie File format

designed by: Zeromus FCEUX uses a new movie file format - .fm2.

This differs from the previous FCE Ultra movie format (.fcm) in the following ways:

-It is completely text based; allowing easy movie editing/splicing -An imbedded GUID so FCEUX can tell if a savestate belongs to a movie file

-Movies recorded from Start (Power-on) no longer have a redundant savestate

-Can no longer record from soft reset

-Contains mouse input for recording the Zapper & Arkanoid Paddle

Format

FM2 is ASCII plain text. It consists of several key-value pairs followed by an input log section.

The input log section can be identified by its starting with a | (pipe). The input log section terminates at eof. Newlines may be \r\n or \n

Key-value pairs consist of a key identifier, followed by a space separator, followed by the value text.

Value text is always terminated by a newline, which the value text will not include.

The value text is parsed differently depending on the type of the key. The key-value pairs may be in any order, except that the first key must be version.

Integer keys (also used for booleans, with a 1 or 0) will have a value that is a simple integer not to exceed 32bits

- version (required) - the version of the movie file format; for now it is always 3

- emuVersion (required) - the version of the emulator used to produce the movie

- rerecordCount (optional) - the rerecord count

- palFlag (bool) (optional) - true if the movie uses pal timing

- fourscore (bool) (*note C) - true if a fourscore was used

- FDS (bool) (optional) - true if movie was recorded on a Famicom Disk System (FDS) game

- port0, port1 (*note C) - indicates the types of input devices. Supported values are:

SI_GAMEPAD = 1,

 $SI_ZAPPER = 2$

- port2 (required) - indicates the type of the FCExp port device which was attached. Supported values are:

 $SIFC_NONE = 0$

Header

String keys have values that consist of the remainder of the key-value pair line. As a consequence, string values cannot contain newlines. - romFilename (required) - the name of the file used to record the movie

Hex string keys (used for binary blobs) will have a value that is like 0x0123456789ABCDEF...

- romChecksum (required) - the MD5 hash of the ROM which was used to record the movie

- savestate (optional) - a fcs savestate blob, in case a movie was recorded from savestate

GUID keys have a value which is in the standard guide format: 452DE2C3-EF43-2FA9-77AC-0677FC51543B

- guide (required) a unique identifier for a movie, generated when the movie is created, which is used when loading a savestate to make sure it belongs to the current movie.

Comments

- comment (optional) - simply a memo.

by convention, the first token in the comment value is the subject of the comment.

by convention, subsequent comments with the same subject will have their ordering preserved and may be used to approximate multi-line comments.

by convention, the author of the movie should be stored in comment(s) with a subject of: author

example: comment author adelikat

Subtitles

- subtitle (optional) - a message that will be displayed on screen when movie is played back (unless Subtitles are turned off, see <u>Movie options</u>).

by convention, subtitles will begin with the word subtitle

by convention, an integer value will proceed which will indicate the frame that the subtitle will be displayed

by convention, a string of text will follow which will be the string displayed

example: subtitle 1000 Level Two

At frame 1000 the words "Level Two" will be displayed on the screen

Input log

The input log section consists of lines beginning and ending with a | (pipe).

The fields are as follows, except as noted in note C. |c|port0|port1|port2| field c is a variable length decimal integer which is a bit field corresponding to miscellaneous input states which are valid at the start of the frame.

Current values for this are MOVIECMD_RESET = 1

the format of port0, port1, port2 depends on which types of devices were attached.

SI_GAMEPAD:

the field consists of eight characters which constitute a bit field. any character other than ' ' or '.' means that the button was pressed. by convention, the following mnemonics will be used in a column to remind us of which button corresponds to which column:

RLDUTSBA (Right,Left,Down,Up,sTart,Select,B,A)

SI_ZAPPER:

XXX YYY B Q Z

XXX: %03d, the x position of the mouse

YYY: %03d, the y position of the mouse

B: %1d, 1 if the mouse button is pressed; 0 if not

Q: %1d, an internal value used by the emulator's zapper code (this is most unfortunate..)

Z: %d, a variable-length decimal integer; an internal value used by the emulator's zapper code (this is even more unfortunate..)

SIFC_NONE:

this field must always be empty.

* Notes *

A. There is no key-value pair that indicates the length of the movie. This must be read by scanning the input log and counting the number of lines.

B. All movies start from power-on, unless a savestate key-value is present.

C.

If a fourscore is used, then port0 and port1 are irrelevant and ignored. The input types must all be gamepads, and the input log will be in the following format:

{player1 player2 player3 player4} |c|ABSTUDLR|ABSTUDLR|ABSTUDLR|port2| If a fourscore is not used, then port0 and port1 are required.

- D. The emulator uses these framerate constants
 - NTSC: 1008307711 /256/65536 = 60.099822938442230224609375
 - PAL : 838977920 /256/65536 = 50.00698089599609375

E. The author of this format is curious about what people think of it. Please let him know!

2008

This help file has been generated by the freeware version of <u>HelpNDoc</u>

.fcm

FCE Ultra Movie File Format

- Updated March 22, 2004

The FCM file format is a somewhat "joined" file format. The first part of a FCM $% \left({{\rm{FCM}}} \right)$

file will contain an FCS-format state save. After this data, the FCM-specific data

begins, which is being referred to from this point.

Currently, the only supported input scheme for a FCM is four joysticks.

The FCM data consists of a stream of joystick commands:

dLLjjbbb

d = Dummy update, if set. Used to reset frame timestamp.

LL = timestamp length, in bytes(maximum of 3 bytes).

jj = Which joystick(0-3).

bbb = Which button(0-7).

If the dummy update bit is set, a command can also have occurred. Look at the

lower 5 bits:

0 = Just a dummy update. 1 = Reset 2 = Power

The timestamp is stored after the joystick command, in LSB-first format. It is

the number of frames since the last event. A timestamp length of "0" is valid, to

be used when several different buttons need to change state at the same time(same frame,

at least).

This help file has been generated by the freeware version of <u>HelpNDoc</u>

Savestate (.fcs)

FCE Ultra Save State Format Updated: Mar 9, 2003

FCE Ultra's save state format is now designed to be as forward and backwards

compatible as possible. This is achieved through the (over)use of chunks.

All multiple-byte variables are stored LSB(least significant byte)-first. Data types:

(u)int8 - (un)signed 8 bit variable(also referred to as "byte")
(u)int16 - (un)signed 16 bit variable
(u)int32 - (un)signed 32 bit variable

-- Main File Header:

The main file header is 16-bytes in length. The first three bytes contain the string "FCS". The next byte contains the version of FCE Ultra that saved

this save state. This document only applies to version "53"(.53) and higher.

After the version byte, the size of the entire file in bytes(minus the 16 byte main file header) is stored. If oldversion is set to 255, the 32-bit version field will be used. In this field, a version such as 0.98.10 is stored as "9810"(decimal).

The rest of the header is currently unused and should be nulled out. Example of relevant parts:

FCS <uint8 oldversion> <uint32 totalsize> <uint32 version>

-- Section Chunks:

Sections chunk headers are 5-bytes in length. The first byte defines what section it is, the next four bytes define the total size of the section (including the section chunk header).

<uint8 section> <uint32 size>

Section definitions:

| 1 | - | "CPU" |
|----|---|---------|
| 2 | - | "CPUC" |
| 3 | - | "PPU" |
| 4 | - | "CTLR" |
| 5 | - | "SND" |
| 16 | - | "EXTRA" |

-- Subsection Chunks

Subsection chunks are stored within section chunks. They contain the actual

state data. Each subsection chunk is composed of an 8-byte header and the data.

The header contains a description(a name) and the size of the data contained

in the chunk:

```
<uint8 description[4]> <uint32 size>
```

The name is a four-byte string. It does not need to be null-terminated. If the string is less than four bytes in length, the remaining unused bytes must be null.

-- Subsection Chunk Description Definitions

Note that not all subsection chunk description definitions listed below are guaranteed to be in the section chunk. It's just a list of what CAN be in a section chunk. This especially applies to the "EXTRA" subsection.

---- Section "CPU"

Name: Type: Description:

| | P X Y S | uint8 P uint8 X uint8 Y uint8 S | Program Counter Accumulator Processor status register Cregister Cregister Stack pointer |
|---------|-----------------------------|--|--|
| | RAM | | 2KB work RAM |
| | Section | "CPUC" (emula | itor specific) |
| | Name: | Туре: | Description: |
| 000 | JAMM IRQL Istantly | uint8 uint8 | Non-zero value if CPU in a "jammed" state Non-zero value if IRQs are to be generated |
| | ICoa ICou | int32 int32 | Temporary cycle counter Cycle counter |
| | Section | "PPU" | |
| | Name: | Туре: | Description: |
| | NTAR PRAM SPRA PPU | uint8[0x100] | 32 bytes of palette index RAM 256 bytes of sprite RAM |
| PP | | unito[4] L | ast values written to \$2000 and \$2001, the |
| | 0 | status ro \$2003. | egister, and the last value written to |
| | XOFF | uint8 | Tile X-offset. |
| | VTOG | uint8 | Toggle used by \$2005 and \$2006. |
| 1 - I - | RADD | uint16 | PPU Address Register(address written |
| to/r | ead fron | | 2002 is accessed) |
| | TADD | uint16 | 2007 is accessed). PPU Address Register |
| | VBUF | uint8 | VRAM Read Buffer |

PGENuint8PPU "general" latch. See Ki's document.---- Section "CTLR" (somewhat emulator specific)Name:Type:Description:J1RBuint8Bit to be returned when first joystick isread.J2RBuint8Bit to be returned when second joystick is

read.

---- Section "SND" (somewhat emulator specific)

| NREG | uint16 | Noise LFSR. |
|------|--------|--|
| P17 | uint8 | Last byte written to \$4017. |
| PBIN | uint8 | DMC bit index. |
| PAIN | uint32 | DMC address index(from \$8000). |
| PSIN | uint32 | DMC length counter(how many bytes left |
| | to fet | ch). |

<to be finished>

---- Section "EXTRA" (varying emulator specificness)

For iNES-format games(incomplete, and doesn't apply to every game):

| Туре: | Description: |
|---------------|---|
| | |
| uint8[0x2000] | 8KB of WRAM at \$6000-\$7fff |
| uint8[0x8000] | (very emulator specific) |
| uint8[0x2000] | 8KB of CHR RAM at \$0000-\$1fff(in |
| space). | |
| uint8[0x800] | Extra 2KB of name/attribute table RAM. |
| uint8[32] (\ | /ery emulator specific) |
| uint8 C | Current mirroring: |
| 0 = "H | orizontal" |
| | uint8[0x2000] uint8[0x8000] uint8[0x2000] space). uint8[0x800] uint8[32] (vuint8 |

| | | . = "Vertical" 10 = Mirror from \$2000 |
|--------------|---------------|---|
| | | 511 = Mirror from \$2400 |
| IRQC | uint32 | Generic IRQ counter |
| - | uint32 | Generic IRQ latch |
| • | uint32 | Generic IRQ latch |
| IRQA | uint8 | Generic IRQ on/off register. |
| PBL | | List of 4 8KB ROM banks paged in at |
| \$8000-\$FFF | | |
| | [-] | List of 8 1KB VROM banks page in at |
| \$0000-\$1FF | F(PPU). | |
| For FDS | games(inco | mplete): |
| Name: | Type: | Description: |
| DDT <x></x> | • uint8[65500 |)] Disk data for side x(0-3). |
| FDSR | - | 000] 32 KB of work RAM |
| CHRR | uint8[0x20 | 000] 8 KB of CHR RAM |
| - | | IRQ counter |
| - | | IRQ latch |
| IRQA | uint8 | IRQ on/off. |
| WAVE | [-] | |
| MWAV | [-] | |
| AMPL | uint8[2] | Amplitude data. |

2008

This help file has been generated by the freeware version of <u>HelpNDoc</u>

Sound

<u>Sound</u>

Includes specifications for the NSF Format & NES Sound core

NSF Format

NES Sound

2008

This help file has been generated by the freeware version of <u>HelpNDoc</u>

NSF Format

NES Music Format Spec

By: Kevin Horton khorton@iquest.net

NOTE:

Remember that I am very willing to add stuff and update this spec. If you find a new sound chip or other change let me know and I will get back

with you. E-mail to the above address.

V1.61 - 06/27/2000 Updated spec a bit
V1.60 - 06/01/2000 Updated Sunsoft, MMC5, and Namco chip information
V1.50 - 05/28/2000 Updated FDS, added Sunsoft and Namco chips
V1.32 - 11/27/1999 Added MMC5 register locations
V1.30 - 11/14/1999 Added MMC5 audio bit, added some register info
V1.20 - 09/12/1999 VRC and FDS prelim sound info added
V1.00 - 05/11/1999 First official NSF specification file

This file encompasses a way to transfer NES music data in a small, easy to

use format.

The basic idea is one rips the music/sound code from an NES game and prepends

a small header to the data.

A program of some form (6502/sound emulator) then takes the data and loads

it into the proper place into the 6502's address space, then inits and plays

the tune.

Here's an overview of the header:

offset # of bytes Function

```
0000 5 STRING "NESM",01Ah ; denotes an NES sound format file
0005
      1 BYTE Version number (currently 01h)
                 Total songs (1=1 song, 2=2 songs, etc)
0006
      1 BYTE
0007
      1 BYTE
                 Starting song (1= 1st song, 2=2nd song, etc)
8000
      2 WORD (lo/hi) load address of data (8000-FFFF)
000a
      2 WORD
                   (lo/hi) init address of data (8000-FFFF)
000c
       2 WORD
                   (lo/hi) play address of data (8000-FFFF)
000e
      32 STRING The name of the song, null terminated
002e
      32 STRING The artist, if known, null terminated
004e
       32 STRING The Copyright holder, null terminated
                   (lo/hi) speed, in 1/1000000th sec ticks, NTSC (see
006e
       2 WORD
text)
0070
                  Bankswitch Init Values (see text, and FDS section)
       8 BYTE
0078
       2 WORD
                   (lo/hi) speed, in 1/1000000th sec ticks, PAL (see
text)
007a
                  PAL/NTSC bits:
       1 BYTE
          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. they *must* be 0
       1 BYTE Extra Sound Chip Support
007b
          bit 0: if set, this song uses VRCVI
          bit 1: if set, this song uses VRCVII
          bit 2: if set, this song uses FDS Sound
          bit 3: if set, this song uses MMC5 audio
```

bit 4: if set, this song uses Namco 106 bit 5: if set, this song uses Sunsoft FME-07 bits 6,7: future expansion: they *must* be 0 007c 4 ---- 4 extra bytes for expansion (must be 00h) 0080 nnn ---- The music program/data follows

This may look somewhat familiar; if so that's because this is somewhat sorta of based on the PSID file format for C64 music/sound.

Loading a tune into RAM

If offsets 0070h to 0077h have 00h in them, then bankswitching is *not* used. If one or more bytes are something other than 00h then bankswitching

is used. If bankswitching is used then the load address is still used, but you now use (ADDRESS AND 0FFFh) to determine where on the first bank

to load the data.

Each bank is 4K in size, and that means there are 8 of them for the entire 08000h-0ffffh range in the 6502's address space. You determine where

in memory the data goes by setting bytes 070h thru 077h in the file. These determine the inital bank values that will be used, and hence where

the data will be loaded into the address space.

Here's an example:

METROID.NSF will be used for the following explaination.

The file is set up like so: (starting at 070h in the file)

0070: 05 05 05 05 05 05 05 05 - 00 00 00 00 00 00 00 00 00 0080: ... music data goes here...

Since 0070h-0077h are something other than 00h, then we know that this tune uses bankswitching. The load address for the data is specified as 08000h. We take this AND 0fffh and get 0000h, so we will load data in at byte 0 of bank 0, since data is loaded into the banks sequentially starting from bank 0 up until the music data is fully loaded.

Metroid has 6 4K banks in it, numbered 0 through 5. The 6502's address space has 8 4K bankswitchable blocks on it, starting at 08000h-08fffh, 09000h-09fffh, 0a000h-0afffh ... 0f000h-0ffffh. Each one of these is 4K in size, and the current bank is controlled by writes to 05ff8h thru 05fffh, one byte per bank. So, 05ff8h controls the 08000h-08fffh range, 05ff9h controls the 09000h-09fffh range, etc. up to 05fffh which controls the 0f000h-0ffffh range. When the song is loaded into RAM, it is loaded into the banks and not the 6502's address space. Once this is done, then the bank control registers are written to set up the inital bank values. To do this, the value at 0070h in the file is written to 05ff8h, 0071h is written to 05ff9h, etc. all the way to 0077h is written to 05fffh. This is should be done before every call to the init routine.

If the tune was not bankswitched, then it is simply loaded in at the specified load address, until EOF

Initalizing a tune

This is pretty simple. Load the desired song # into the accumulator, minus 1 and set the X register to specify PAL (X=1) or NTSC (X=0). If this is a single standard tune (i.e. PAL *or* NTSC but not both) then the X register contents should not matter. Once the song # and optional PAL/NTSC standard are loaded, simply call the INIT address. Once init is done, it should perform an RTS.

Once the tune has been initalized, it can now be played. To do this, simply call the play address several times a second. How many times per second is determined by offsets 006eh and 006fh in the file. These bytes denote the speed of playback in 1/1000000ths of a second. For the "usual" 60Hz playback rate, set this to 411ah.

To generate a differing playback rate, use this formula:

1000000 PBRATE= -----speed

Where PBRATE is the value you stick into 006e/006fh in the file, and speed is the desired speed in hertz.

"Proper" way to load the tune

- 1) If the tune is bankswitched, go to #3.
- 2) Load the data into the 6502's address space starting at the specified load address. Go to #4.
- 3) Load the data into a RAM area, starting at (start_address AND 0fffh).

4) Tune load is done.

"Proper" way to init a tune

1) Clear all RAM at 0000h-07ffh.

- 2) Clear all RAM at 6000h-7fffh.
- 3) Init the sound registers by writing 00h to 04000-0400Fh, 10h to 4010h, and 00h to 4011h-4013h.
- 4) Set volume register 04015h to 00fh.
- 5) If this is a banked tune, load the bank values from the header into 5ff8-5fffh.
- 6) Set the accumulator and X registers for the desired song.

7) Call the music init routine.

"Proper" way to play a tune

1) Call the play address of the music at periodic intervals determined by the speed words. Which word to use is determined by which mode you are in- PAL or NTSC.

Sound Chip Support

Byte 007bh of the file stores the sound chip flags. If a particular flag is set, those sound registers should be enabled. If the flag is clear, then those registers should be disabled.

* VRCVI Uses registers 9000-9002, A000-A002, and B000-B002, write only.

Caveats: 1) The above registers are *write only* and must not disrupt music

code that happens to be stored there.

2) Major caveat: The A0 and A1 lines are flipped on a few games!! If you rip the music and it sounds all funny, flip around the xxx1 and xxx2 register pairs. (i.e. 9001 and 9002) 9000 and 9003 can be left untouched. I decided to do this since it would make things easier all around, and this means you only will have to change the music code in a very few places (6). Esper2 and Madara will need this change, while Castlevania 3j will not for instance.

3) See my VRCVI.TXT doc for a complete register description.

* VRCVII Uses registers 9010 and 9030, write only.

Caveats: 1) Same caveat as #1, above.

2) See my VRCVII.TXT doc for a complete register description.

* FDS Sound uses registers from 4040 through 4092.

Caveats: 1) 6000-DFFF is assumed to be RAM, since 6000-DFFF is RAM on the

FDS. E000-FFFF is usually not included in FDS games because it is the BIOS ROM. However, it can be used on FDS rips to help the ripper (for modified play/init addresses).

2) Bankswitching operates slightly different on FDS tunes. 5FF6 and 5FF7 control the banks 6000-6FFF and 7000-7FFF respectively. NSF header offsets 76h and 77h correspond to *both* 6000-7FFF *AND* E000-FFFF. Keep this in mind!

* MMC5 Sound Uses registers 5000-5015, write only as well as 5205 and 5206,

and 5C00-5FF5

Caveats: 1) Generating a proper doc file. Be patient.

2) 5205 and 5206 are a hardware 8*8 multiplier. The idea being you write your two bytes to be multiplied into 5205 and 5206 and after doing so, you read the result back out. Still working on what exactly triggers it (I think a write to either 5205 or 5206 triggers the multiply).

3) 5C00-5FF5 should be RAM to emulate EXRAM while in MMC5 mode.

Note: Thanks to Mamiya for the EXRAM info.

* Namco 106 Sound Uses registers 4800 and F800.

This works similar to VRC7. 4800 is the "data" port which is readable and writable, while F800 is the "address" port and is writable only.

The address is 7 bits plus a "mode" bit. Bit 7 controls address auto-incrementing. If bit 7 is set, the address will auto-increment after a byte of data is read or written from/to 4800.

```
$40 ffffffff f:frequency L
$42 ffffffff f:frequency M
$44 ---sssff f:frequency H s:tone length (8-s)*4 in 4bit-samples
$46 ttttttt t:tone address(4bit-address,$41 means high-4bits of
```

\$20)

\$47 -cccvvvv v:linear volume 1+c:number of channels in use(\$7F

only)

\$40-47:ch1 \$48-4F:ch2 ... \$78-7F:ch8 ch2-ch8 same to ch1

\$00-3F(8ch)...77(1ch) hhhhllll tone data h:odd address data(signed 4bit) l:even address data(signed 4bit) real frequency = (f * NES_BASECYCLES) / (40000h * (c+1) * (8s)*4 * 45) NES BASECYCLES 21477270(Hz)

Note: Very Special thanks to Mamiya for this information!

* Sunsoft FME-07 Sound uses registers C000 and E000

This is similar to the common AY 3-8910 sound chip that is used on tons of arcade machines, and in the Intellivision.

C000 is the address port E000 is the data port

Both are write-only, and behave like the AY 3-8910.

Note: Special thanks to Mamiya for this information as well

Caveats

1) The starting song number and maximum song numbers start counting at

1, while the init address of the tune starts counting at 0. To "fix", simply pass the desired song number minus 1 to the init routine.

2) The NTSC speed word is used *only* for NTSC tunes, or dual PAL/NTSC tunes.

The PAL speed word is used *only* for PAL tunes, or dual PAL/NTSC tunes.

3) The length of the text in the name, artist, and copyright fields must be 31 characters or less! There has to be at least a single NULL byte (00h) after the text, between fields. 4) If a field is not known (name, artist, copyright) then the field must contain the string "<?>" (without quotes).

5) There should be 8K of RAM present at 6000-7FFFh. MMC5 tunes need RAM at

5C00-5FF7 to emulate its EXRAM. 8000-FFFF Should be read-only (not

writable) after a tune has loaded. The only time this area should be writable is if an FDS tune is being played.

6) Do not assume the state of *anything* on entry to the init routine except A and X. Y can be anything, as can the flags.

7) Do not assume the state of *anything* on entry to the play routine either.

Flags, X, A, and Y could be at any state. I've fixed about 10 tunes because of this problem and the problem, above.

8) The stack sits at 1FFh and grows down. Make sure the tune does not attempt to use 1F0h-1FFh for variables. (Armed Dragon Villigust did and

I had to relocate its RAM usage to 2xx)

9) Variables should sit in the 0000h-07FFh area *only*. If the tune writes outside this range, say 1400h this is bad and should be relocated. (Terminator 3 did this and I relocated it to 04xx).

That's it!

2008

This help file has been generated by the freeware version of <u>HelpNDoc</u>

NES Sound

2A03 sound channel hardware documentation

Brad Taylor (big_time_software@hotmail.com) 4th release: February 19th, 2K3

All results were obtained by studying prior information available (from nestech 1.00, and postings on NESDev from miscellanious people), and through a series of experiments conducted by me. Results acquired by individuals prior to my reverse-engineering have been double checked, and final results have been confirmed. Credit is due to those individual(s) who contributed miscellanious information in regards to NES sound channel hardware. Such individuals are:

Goroh Memblers FluBBa Izumi Chibi-Tech Quietust SnowBro

Kentaro Ishihara (Ki) is responsible for posting (on the NESdev mailing list) differrences in the 2 square wave channels, including the operation of 2A03 hardware publically undocumented (until now) such as the frame IRQ counter, and it's ties with sound hardware. Goroh had originally discovered some of this information, and Ki confirmed it.

A special thanks goes out to Matthew Conte, for his expertise on pseudorandom number generation (amoung other things), which allowed for the full reverse engineering of the NES's noise channel to take place. Without his help, I would still be trying to find a needle in a haystack, as far as the noise's method of pseudo-random number generation goes. Additionally, his previous findings / reverse engineering work on the NES's sound hardware really got the ball of NES sound emulation rolling. If it weren't for Matt's original work, this document wouldn't exist.

```
* Introduction *
```

The 2A03 (NES's integrated CPU) has 4 internal channels to it that have the ability to generate semi-analog sound, for musical playback purposes. These channels are 2 square wave channels, one triangle wave channel, and a noise generation channel. This document will go into full detail on every aspect of the operation and timing of the mentioned sound channels.

```
* Channel details *
```

Each channel has different characteristics to it that make up it's operation.

The square channel(s) have the ability to generate a square wave frequency in the range of 54.6 Hz to 12.4 KHz. It's key features are frequency sweep abilities, and output duty cycle adjustment.

The triangle wave channel has the ability to generate an output triangle wave with a resolution of 4-bits (16 steps), in the range of 27.3 Hz to 55.9 KHz. The key features this channel has is it's analog triangle wave output, and it's linear counter, which can be set to automatically disable the channel's sound after a certain period of time has gone by.

The noise channel is used for producing random frequencys, which results in a "noisey" sounding output. Output frequencys can range anywhere from 29.3 Hz to 447 KHz. It's key feature is it's pseudo-random number generator, which generates the random output frequencys heard by the channel.

```
*****
```

```
* Frame counter *
```

The 2A03 has an internal frame counter. The purpose of it is to generate the various low frequency signals (60, 120, 240 Hz, and 48, 96, 192 Hz) required to clock several of the sound hardware's counters. It also has the ability to generate IRQ's.

The smallest unit of timing the frame counter operates around is 240Hz; all other frequencies are generated by multiples of this base frequency. A clock divider of 14915 (clocked at twice the CPU speed) is used to get 240Hz (this was the actual measured ratio).

+----+

|\$4017 operation|

+----+

Writes to register \$4017 control operation of both the clock divider, and the frame counter.

- Any write to \$4017 resets both the frame counter, and the clock divider. Sometimes, games will write to this register in order to synchronize the sound hardware's internal timing, to the sound routine's timing (usually tied into the NMI code). The frame IRQ is slightly longer than the PPU's, so you can see why games would desire this syncronization.

- bit 7 of \$4017 controls the frame counter's divide rate. Every time the counter cycles (reaches terminal count (0)), a frame IRQ will be generated, if enabled by clearing bit 6 of \$4017. \$4015.6 holds the status of the frame counter IRQ; it will be set if the frame counter is responsible for the interrupt.

\$4017.7 divider frame IRQ freq.

----- -----

0 4 60 1 5 48 On 2A03 reset, both bits of \$4017 (6 & 7) will be cleared, enabling frame IRQ's off the hop. The reason why the existence of frame IRQ's are generally unknown is because the 6502's maskable interrupt is disabled on reset, and this blocks out the frame IRQ's. Most games don't use any IRQ-generating hardware in general, therefore they don't bother enabling maskable interrupts.

Note that the IRQ line will be held down by the frame counter until it is acknowledged (by reading \$4015). Before this, the 6502 will generate an IRQ *every* time interrupts are enabled (either by CLI or RTI), since the IRQ design on the 6502 is level-triggered, and not edge. If you've written a program that does not read \$4015 in the IRQ handler, and you execute CLI, the processor will immediately go into a infinite IRQ call-return loop.

+-----+

|Frame counter operation|

+-----+

Depending on the status of \$4017.7, the frame counter will follow 2 different count sequences. These sequences determine when sound hardware counters will be clocked. The sequences are initialized immediately following any write to \$4017.

\$4017.7 sequence

0 4, 0,1,2,3, 0,1,2,3,..., etc.

1 0,1,2,3,4, 0,1,2,3,4,..., etc.

During count sequences 0..3, the linear (triangle) and envelope decay (square & noise) counters recieve a clock for each count. This means that both these counters are clocked once immediately after \$4017.7 is written with a value of 1.

Count sequences 1 & 3 clock (update) the frequency sweep (square), and length (all channels) counters. Even though the length counter's smallest unit of time counting is a frame, it seems that it is actually being clocked twice per frame. That said, you can consider the length counters to contain an extra stage to divide this clock signal by 2.

No aforementioned sound hardware counters are clocked on count sequence #4. You should now see how this causes the 96, and 192 Hz signals to be generated when \$4017.7=1.

The rest of the document will describe the operation of the sound channels using the \$4017.7=0 frequencies (60, 120, and 240 Hz). For \$4017.7=1 operation, replace those frequencies with 48, 96, and 192 Hz (respectively).


```
* Sound hardware delay *
```

After resetting the 2A03, the first time any sound channel(s) length counter contains a non-zero value (channel is enabled), there will be a 2048 CPU clock cycle delay before any of the sound hardware is clocked. After the 2K clock cycles go by, the NES sound hardware will be clocked normally. This phenomenon only occurs prior to a system reset, and only occurs during the first 2048 CPU clocks after the activation of any of the 4 basic sound channels.

The information in regards to this delay is only provided to keep this document accurate with all information that is currently known about the 2A03's sound hardware. I haven't done much tests on the behaviour of this delay (mainly because I don't care, as I view it as a inconvenience anyway), so this information should be taken with a grain of salt.

* Register Assignments *

The sound hardware internal to the 2A03 has been designated these special memory addresses in the CPU's memory map.

\$4000-\$4003Square wave 1\$4004-\$4007Square wave 2 (identical to the first, except for upwardfrequency sweeps (see "sweep unit" section))\$4008-\$400BTriangle\$400C-\$400FNoise\$4015Channel enable / length/frame counter status\$4017frame counter control

Note that \$4015 (and \$4017, but is unrelated to sound hardware) are the only R/W registers. All others are write only (attempt to read them will most likely return the last byte on the bus (usually 040H), due to heavy capacitance on the NES's data bus). Reading a "write only" register, will have no effect on the specific register, or channel.

Every sound channel has 4 registers affiliated with it. The description of the register sets are as follows:

+-----+ | Register set 1 | +-----+

\$4000(sq1)/\$4004(sq2)/\$400C(noise) bits

- 0-3 volume / envelope decay rate
- 4 envelope decay disable
- 5 length counter clock disable / envelope decay looping enable
- 6-7 duty cycle type (unused on noise channel)

\$4008(tri) bits

- 0-6 linear counter load register
- 7 length counter clock disable / linear counter start

+----+ | Register set 2 | +----+

\$4001(sq1)/\$4005(sq2) bits

- 0-2 right shift amount
- 3 decrease / increase (1/0) wavelength
- 4-6 sweep update rate
- 7 sweep enable

\$4009(tri)/\$400D(noise) bits

0-7 unused

+-----+

| Register set 3 |

+----+

\$4002(sq1)/\$4006(sq2)/\$400A(Tri) bits

0-7 8 LSB of wavelength

\$400E(noise) bits

- 0-3 playback sample rate
- 4-6 unused
- 7 random number type generation

+----+

| Register set 4 |

+----+

\$4003(sq1)/\$4007(sq2)/\$400B(tri)/\$400F(noise) bits

- 0-2 3 MS bits of wavelength (unused on noise channel)
- 3-7 length counter load register

+-----+ | length counter status register | +-----+

\$4015(read)

- 0 square wave channel 1
- 1 square wave channel 2
- 2 triangle wave channel
- 3 noise channel
- 4 DMC (see "DMC.TXT" for details)
- 5-6 unused
- 7 IRQ status of DMC (see "DMC.TXT" for details)

+-----+

| channel enable register |

+-----+

\$4015(write)

·----

- 0 square wave channel 1
- 1 square wave channel 2
- 2 triangle wave channel
- 3 noise channel
- 4 DMC channel (see "DMC.TXT" for details)
- 5-7 unused

* Channel architecture *

This section will describe the internal components making up each individual channel. Each component will then be described in full detail.

Device

Triangle Noise

Square _ _ _ _ _ triangle step generator linear counter programmable timer Х Х length counter Х Х 4-bit DAC Х Х Х volume/envelope decay unit Х Х sweep unit Х duty cycle generator Х wavelength converter random number generator Х

Х

Х

Х

Х

Х

+-----+

| Triangle step generator |

+-----+

This is a 5-bit, single direction counter, and it is only used in the triangle channel. Each of the 4 LSB outputs of the counter lead to one input on a corresponding mutually exclusive XNOR gate. The 4 XNOR gates have been strobed together, which results in the inverted representation of the 4 LSB of the counter appearing on the outputs of the gates when the strobe is 0, and a non-inverting action taking place when the strobe is 1. The strobe is naturally connected to the MSB of the counter, which effectively produces on the output of the XNOR gates a count sequence which reflects the scenario of a near- ideal triangle step generator (D,E,F,F,E,D,...,2,1,0,0,1,2,...). At this point, the outputs of the XNOR gates will be fed into the input of a 4-bit DAC.

This 5-bit counter will be halted whenever the Triangle channel's length

or linear counter contains a count of 0. This results in a "latching" behaviour; the counter will NOT be reset to any definite state.

On system reset, this counter is loaded with 0.

The counter's clock input is connected directly to the terminal count output pin of the 11-bit programmable timer in the triangle channel. As a result of the 5-bit triangle step generator, the output triangle wave frequency will be 32 times less than the frequency of the triangle channel's programmable timer is set to generate.

+----+

| Linear counter |

+----+

The linear counter is only found in the triangle channel. It is a 7-bit presettable down counter, with a decoded output condition of 0 available (not exactly the same as terminal count). Here's the bit assignments:

\$4008 bits

------0-6 bits 0-6 of the linear counter load register (NOT the linear counter itself)

7 linear counter start

The counter is clocked at 240 Hz (1/4 framerate), and the calculated length in frames is 0.25*N, where N is the 7-bit loaded value. The counter is always being clocked, except when 0 appears on the output of the counter. At this point, the linear counter & triangle step counter clocks signals are disabled, which results in both counters latching their current state (the linear counter will stay at 0, and the triangle step counter will stop, and the channel will be silenced due to this).

The linear counter has 2 modes: load, and count. When the linear counter is in load mode, it essentially becomes transparent (i.e. whatever value is currently in, or being written to \$4008, will appear on the output of the counter). Because of this, no count action can occur in load mode.

When the mode changes from load to count, the counter will now latch the value currently in it, and start counting down from there. In the count mode, the current value of \$4008 is ignored by the counter (but still retained in \$4008). Described below is how the mode of the linear counter is set:

Writes to \$400B

cur mode

- ---- ----
- 1 load
- 0 load (on next linear counter clock), count

Cur is the current state of the MSB of \$4008.

Writes to \$4008

| old | new | / mode | |
|-----|-----|---|--|
| | | | |
| 0 | Х | count | |
| 1 | 0 | no change (during the CPU write cycle), count | |
| 1 | 1 | no change | |

Old and new represent the state(s) of the MSB of \$4008. Old is the value being replaced in the MSB of \$4008 on the write, and new is the value replacing the old one.

"no change" indicates that the mode of the linear counter will not change from the last.

Note that writes to \$400B when \$4008.7=0 only loads the linear counter with the value in \$4008 on the next *linear* counter clock (and NOT at the end of the CPU write cycle). This is a correction from older versions of this doc.

+----+

| Programmable timer |

+-----+

The programmable timer is a 11-bit presettable down counter, and is found in the square, triangle, and noise channel(s). The bit assignments are as follows:

\$4002(sq1)/\$4006(sq2)/\$400A(Tri) bits

0-7 represent bits 0-7 of the 11-bit wavelength

\$4003(sq1)/\$4007(sq2)/\$400B(Tri) bits

0-2 represent bits 8-A of the 11-bit wavelength

Note that on the noise channel, the 11 bits are not available directly. See the wavelength converter section, for more details.

The counter has automatic syncronous reloading upon terminal count (count=0), therefore the counter will count for N+1 (N is the 11-bit loaded value) clock cycles before arriving at terminal count, and reloading. This counter will typically be clocked at the 2A03's internal 6502 speed (1.79 MHz), and produces an output frequency of 1.79 MHz/(N+1). The terminal count's output spike length is typically no longer than half a CPU clock. The TC signal will then be fed to the appropriate device for the particular sound channel (for square, this terminal count spike will lead to the duty cycle generator. For the triangle, the spike will be fed to the triangle step generator. For noise, this signal will go to the random number generator unit).

+----+

```
| Length counter |
```

+----+

The length counter is found in all sound channels. It is essentially a 7-bit down counter, and is conditionally clocked at a frequency of 60 Hz.

When the length counter arrives at a count of 0, the counter will be stopped (stay on 0), and the appropriate channel will be silenced.

The length counter clock disable bit, found in all the channels, can also be used to halt the count sequence of the length counter for the appropriate channel, by writing a 1 out to it. A 0 condition will permit counting (unless of course, the counter's current count = 0). Location(s) of the length counter clock disable bit:

\$4000(sq1)/\$4004(sq2)/\$400C(noise) bits

5 length counter clock disable

\$4008(tri) bits

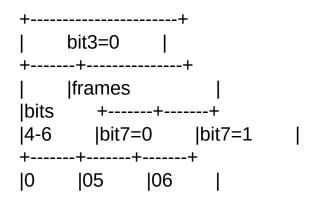
7 length counter clock disable

To load the length counter with a specified count, a write must be made out to the length register. Location(s) of the length register:

\$4003(sq1)/\$4007(sq2)/\$400B(tri)/\$400F(noise) bits

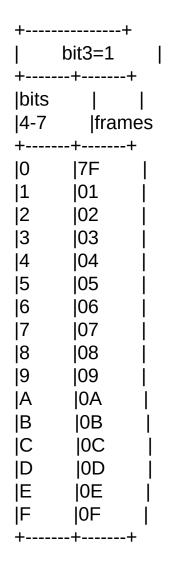
3-7 length

The 5-bit length value written, determines what 7-bit value the length counter will start counting from. A conversion table here will show how the values are translated.



| 1 | 0A | 0C | |
|---|----|-----|--|
| 2 | 14 | 18 | |
| 3 | 28 | 30 | |
| 4 | 50 | 60 | |
| 5 | 1E | 24 | |
| 6 | 07 | 08 | |
| 7 | 0D | 10 | |
| + | + | -++ | |

I



The length counter's real-time status for each channel can be attained. A 0 is returned for a zero count status in the length counter (channel's sound is disabled), and 1 for a non-zero status. Here's the bit description of the length counter status register:

\$4015(read)

0 length counter status of square wave channel 1

- 1 length counter status of square wave channel 2
- 2 length counter status of triangle wave channel
- 3 length counter status of noise channel
- 4 length counter status of DMC (see "DMC.TXT" for details)
- 5 unknown
- 6 frame IRQ status
- 7 IRQ status of DMC (see "DMC.TXT" for details)

Writing a 0 to the channel enable register will force the length counters to always contain a count equal to 0, which renders that specific channel disabled (as if it doesn't exist). Writing a 1 to the channel enable register disables the forced length counter value of 0, but will not change the count itself (it will still be whatever it was prior to the writing of 1).

Bit description of the channel enable register:

\$4015(write)

·----

- 0 enable square wave channel 1
- 1 enable square wave channel 2
- 2 enable triangle wave channel
- 3 enable noise channel
- 4 enable DMC channel (see "DMC.TXT" for details)
- 5-7 unknown

Note that all 5 used bits in this register will be set to 0 upon system reset.

+----+

4-bit DAC

+----+

This is just a standard 4-bit DAC with 16 steps of output voltage resolution, and is used by all 4 sound channels. On the 2A03, square

wave 1 & 2 are mixed together, and are available via pin 1. Triangle & noise are available on pin 2.

These analog outputs require a negative current source, to attain linear symmetry on the various output voltage levels generated by the channel(s) (moreover, to get the sound to be audible). Instead of current sources, the NES uses external 100 ohm pull-down resistors. This results in the output waveforms having some linear asymmetry (i.e., as the desired output voltage increases on a linear scale, the actual outputted voltage increases less and less each step).

The side effect of this is that the DMC's 7-bit DAC port (\$4011) is able to indirectly control the volume (somewhat) of both triangle & noise channels. While I have not measured the voltage asymmetery, others on the NESdev messageboards have posted their findings. The conclusion is that when \$4011 is 0, triangle & noise volume outputs are at maximum. When \$4011 = 7F, the triangle & noise channel outputs operate at only 57% total volume.

The odd thing is that a few games actually take advantage of this "volume" feature, and write values to \$4011 in order to regulate the amplitude of the triangle wave channel's output.

+-----+

| Volume / envelope decay unit |

+----+

The volume / envelope decay hardware is found only in the square wave and noise channels.

\$4000(sq1)/\$4004(sq2)/\$400C(noise)

- 0-3 volume / envelope decay rate
- 4 envelope decay disable
- 5 envelope decay looping enable

When the envelope decay disable bit (bit 4) is set (1), the current volume

value (bits 0-3) is sent directly to the channel's DAC. However, depending on certain conditions, this 4-bit volume value will be ignored, and a value of 0 will be sent to the DAC instead. This means that while the channel is enabled (producing sound), the output of the channel (what you'll hear from the DAC) will either be the 4-bit volume value, or 0. This also means that a 4-bit volume value of 0 will result in no audible sound. These conditions are as follows:

- When hardware in the channel wants to disable it's sound output (like the length counter, or sweep unit (square channels only)).

- On the negative portion of the output frequency signal coming from the duty cycle / random number generator hardware (square wave channel / noise channel).

When the envelope decay disable bit is cleared, bits 0-3 now control the envelope decay rate, and an internal 4-bit down counter (hereon the envelope decay counter) now controls the channel's volume level. "Envelope decay" is used to describe the action of the channel's audio output volume starting from a certain value, and decreasing by 1 at a fixed (linear) rate (which produces a "fade-out" sounding effect). This fixed decrement rate is controlled by the envelope decay rate (bits 0-3). The calculated decrement rate is 240Hz/(N+1), where N is any value between \$0-\$F.

When the channel's envelope decay counter reaches a value of 0, depending on the status of the envelope decay looping enable bit (bit 5, which is shared with the length counter's clock disable bit), 2 different things will happen:

bit 5 action

0 The envelope decay count will stay at 0 (channel silenced).
1 The envelope decay count will wrap-around to \$F (upon the next clock cycle). The envelope decay counter will then continue to count down normally.

Only a write out to \$4003/\$4007/\$400F will reset the current envelope decay counter to a known state (to \$F, the maximum volume level) for the appropriate channel's envelope decay hardware. Otherwise, the envelope decay counter is always counting down (by 1) at the frequency currently contained in the volume / envelope decay rate bits (even when envelope decays are disabled (setting bit 4)), except when the envelope decay counter contains a value of 0, and envelope decay looping (bit 5) is disabled (0).

+----+ | Sweep unit |

+----+

The sweep unit is only found in the square wave channels. The controls for the sweep unit have been mapped in at \$4001 for square 1, and \$4005 for square 2.

The controls

Bit 7 when this bit is set (1), sweeping is active. This results in realtime increasing or decreasing of the the current wavelength value (the audible frequency will decrease or increase, respectively). The wavelength value in \$4002/3 (\$4006/7) is constantly read & updated by the sweep. Modifying the contents of \$4002/3 will be immediately audible, and will result in the sweep now starting from this new wavelength value.

Bits 6-4 These 3 bits represent the sweep refresh rate, or the frequency at which \$4002/3 is updated with the new calculated wavelength. The refresh rate frequency is 120Hz/(N+1), where N is the value written, between 0 and 7.

Bit 3 This bit controls the sweep mode. When this bit is set (1), sweeps will decrease the current wavelength value, as a 0 will increase the current wavelength.

Bits 2-0 These bits control the right shift amount of the new

calculated sweep update wavelength. Code that shows how the sweep unit calculates a new sweep wavelength is as follows: bit 3

```
0 New = Wavelength + (Wavelength >> N)
```

```
1 New = Wavelength - (Wavelength >> N) (minus an additional 1, if using square wave channel 1)
```

where N is the the shift right value, between 0-7.

Note that in decrease mode, for subtracting the 2 values: 1's compliment (NOT) is being used for square wave channel 1 2's compliment (NEG) is being used for square wave channel 2

This information is currently the only known difference between the 2 square wave channels.

On each sweep refresh clock, the Wavelength register will be updated with the New value, but only if all 3 of these conditions are met:

- bit 7 is set (sweeping enabled)

- the shift value (which is N in the formula) does not equal to 0
- the channel's length counter contains a non-zero value

Notes

There are certain conditions that will cause the sweep unit to silence the channel, and halt the sweep refresh clock (which effectively stops sweep action, if any). Note that these conditions pertain regardless of any sweep refresh rate values, or if sweeping is enabled/disabled (via bit 7).

an 11-bit wavelength value less than \$008 will cause this condition
 if the sweep unit is currently set to increase mode, the New calculated wavelength value will always be tested to see if a carry (bit \$B) was generated or not (if sweeping is enabled, this carry will be examined before the Wavelength register is updated) from the shift addition calculation. If carry equals 1, the channel is silenced, and sweep action is

halted.

+-----+

| Duty cycle generator |

+-----+

The duty cycle generator takes the fequency produced from the 11-bit programmable timer, and uses a 4 bit counter to produce 4 types of duty cycles. The output frequency is then 1/16 that of the programmable timer. The duty cycle hardware is only found in the square wave channels. The bit assignments are as follows:

\$4000(sq1)/\$4004(sq2)

6-7 Duty cycle type

duty (positive/negative)

- val in clock cycles
- --- -----
- 00 2/14
- 01 4/12
- 10 8/8
- 11 12/4

Where val represents bits 6-7 of \$4000/\$4004.

This counter is reset when the length counter of the same channel is written to (via \$4003/\$4007).

The output frequency at this point will now be fed to the volume/envelope decay hardware.

+-----+

| Wavelength converter |

+-----+

The wavelength converter is only used in the noise channel. It is used to

convert a given 4-bit value to an 11-bit wavelength, which then is sent to the noise's own programmable timer. Here is the bit descriptions:

\$400E bits

0-3 The 4-bit value to be converted

Below is a conversion chart that shows what 4-bit value will represent the 11-bit wavelength to be fed to the channel's programmable timer:

| value octave | | scale | CPU clock cycles (11-bit wavelength+1) | |
|--------------|---------|-------|--|--|
| 0 | - 15 | A | 002 | |
| 1 | 14 | А | 004 | |
| 2 | 13 | А | 008 | |
| 3 | 12 | А | 010 | |
| 4 | 11 | А | 020 | |
| 5 | 11 | D | 030 | |
| 6 | 10 | А | 040 | |
| 7 | 10 | F | 050 | |
| 8 | 10 | С | 065 | |
| 9 | 9 | А | 07F | |
| А | 9 | D | 0BE | |
| В | 8 | А | 0FE | |
| С | 8 | D | 17D | |
| D | 7 | А | 1FC | |
| Е | 6 | А | 3F9 | |
| F | 5 | А | 7F2 | |

Octave and scale information is provided for the music enthusiast programmer who is more familiar with notes than clock cycles.

+-----+

| Random number generator |

+-----+

The noise channel has a 1-bit pseudo-random number generator. It's

based on a 15-bit shift register, and an exclusive or gate. The generator can produce two types of random number sequences: long, and short. The long sequence generates 32,767-bit long number patterns. The short sequence generates 93-bit long number patterns. The 93-bit mode will generally produce higher sounding playback frequencys on the channel. Here is the bit that controls the mode:

\$400E bits

7 mode

If mode=0, then 32,767-bit long number sequences will be produced (32K mode), otherwise 93-bit long number sequences will be produced (93-bit mode).

The following diagram shows where the XOR taps are taken off the shift register to produce the 1-bit pseudo-random number sequences for each mode.

mode <--------- EDCBA9876543210 32K ** 93-bit * *

The current result of the XOR will be transferred into bit position 0 of the SR, upon the next shift cycle. The 1-bit random number output is taken from pin E, is inverted, then is sent to the volume/envelope decay hardware for the noise channel. The shift register is shifted upon recieving 2 clock pulses from the programmable timer (the shift frequency will be half that of the frequency from the programmable timer (one octave lower)).

On system reset, this shift register is loaded with a value of 1.

RP2A03E quirk

I have been informed that revisions of the 2A03 before "F" actually lacked support for the 93-bit looped noise playback mode. While the Famicom's 2A03 went through 4 revisions (E..H), I think that only one was ever used for the front loading NES: "G". Other differences between 2A03 revisions are unknown.

EOF

2008

This help file has been generated by the freeware version of <u>HelpNDoc</u>

NES Processor

NES Processing

Includes Technical Specifications for the emulation of the NES CPU (6502) and the NES PPU (2C02).

<u>CPU</u>

<u>PPU</u>

NES Scrolling part 1

NES Scrolling part 2

2008

This help file has been generated by the freeware version of HelpNDoc

6502 CPU

```
#
# $Id: 6502_cpu.txt,v 1.1.1.1 2004/08/29 01:29:35
bryan Exp $
#
# This file is part of Commodore 64 emulator
       and Program Development System.
#
#
# See README for copyright notice
#
# This file contains documentation for
6502/6510/8500/8502 instruction set.
#
#
# Written by
                    (john@ucc.gu.uwa.edu.au)
#
    John West
    Marko MŠkelŠ
                    (msmakela@kruuna.helsinki.fi)
#
#
#
# $Log: 6502_cpu.txt,v $
# Revision 1.1.1.1 2004/08/29 01:29:35
                                          brvan
# no message
#
# Revision 1.1
                2002/05/21 00:42:27
                                      xodnizel
# updates
#
# Revision 1.8
                1994/06/03
                            19:50:04
                                       jopi
# Patchlevel 2
#
# Revision 1.7
                1994/04/15
                            13:07:04
                                       jopi
# 65xx Register descriptions added
#
# Revision 1.6
                1994/02/18
                            16:09:36
                                       jopi
#
# Revision 1.5
                1994/01/26
                            16:08:37
                                       jopi
# X64 version 0.2 PL 1
#
```

```
# Revision 1.4 1993/11/10 01:55:34
                                          jopi
#
# Revision 1.3 93/06/21
                                      jopi
                            13:37:18
# X64 version 0.2 PL 0
#
# Revision 1.2 93/06/21 13:07:15
                                      jopi
# *** empty log message ***
#
#
Note: To extract the uuencoded ML programs in this article most
   easily you may use e.g. "uud" by Edwin Kremer,
   which extracts them all at once.
```

Documentation for the NMOS 65xx/85xx Instruction Set

6510 Instructions by Addressing Modes 6502 Registers 6510/8502 Undocumented Commands Register selection for load and store Decimal mode in NMOS 6500 series 6510 features Different CPU types 6510 Instruction Timing How Real Programmers Acknowledge Interrupts Memory Management Autostart Code Notes References

6510 Instructions by Addressing Modes

| off- | +++++++ | +++ Posi | tive | +++++++++ | | |
|-------|---------|----------|------|-----------|----|----|
| Negat | tive | | | | | |
| set | 00 | 20 | 40 | 60 | 80 | a0 |
| C0 | e0 | mode | ; | | | |

| | | | RTI | RTS | NOP* | LDY |
|------------|--------------------|-------------|------------------|--------|-------|-------|
| +01 | CPX ORA | AND | EOR | ADC | STA | LDA |
| +02 | SBC t t NOP* | t t | t t /immod | t | NOP*t | LDX |
| +03 | SLO* ISB* | RLA* | SRE* | RRA* | SAX* | LAX* |
| +04 | NOP* CPX | BIT | NOP* | NOP* | STY | LDY |
| +05 | ORA SBC | AND | EOR | ADC | STA | LDA |
| +06 | ASL INC | ROL | LSR | ROR | STX | LDX |
| +07 | | RLA* | SRE* | RRA* | SAX* | LAX* |
| | | | PHA | PLA | DEY | TAY |
| +09 | INX ORA | AND | EOR | ADC | NOP* | LDA |
| +0a | | ROL | LSR | | | ТАХ |
| | | | /impl ASR** | ARR** | ANE** | LXA** |
| +0c | | BIT | JMP | JMP () | STY | LDY |
| +0d | CPX ORA SBC | AND | EOR | ADC | STA | LDA |
| +0e | ASL INC | ROL | LSR | ROR | STX | LDX |
| +0f | SLO* ISB* | RLA* | SRE* | RRA* | SAX* | LAX* |
| | BPL | | | BVS | BCC | BCS |
| BNE +11 | ORA | Rela AND | EOR | ADC | STA | LDA |
| | SBC t | • | ir),y t | t | t | t |

| t | t | ? | | | | |
|-------------|------|--------------|--------|-------------|--|---------|
| +13 | SL0* | RLA* | SRE* | RRA* | SHA** | LAX* |
| DCP* | ISB* | (ind | ir),y | | | |
| +14 | | NOP* | | NOP* | STY | LDY |
| NOP* | | Zero | | | | |
| +15 | ORA | AND | EOR | ADC | STA | LDA |
| CMP | | Zero | | | | |
| +16 | ASL | ROL | | ROR | STX y) | LDX y) |
| DEC | INC | | page,x | | | |
| +17 | | | | RRA* | SAX* y) | LAX* y) |
| DCP* | ISB* | Zero | page,x | | | |
| 110 | | 050 | | <u>с</u> гт | T \/ A | |
| +18 | CLC | SEC | | SEI | TYA | CLV |
| | SED | - | | | CT A | |
| +19 CMD | | AND | | ADC | STA | LDA |
| | NOP* | Abso NOP* | | | TVC | TSX |
| +1a NOP* | | | | NOP* | TXS | 137 |
| +1b | SLO* | Impl RLA* | SRE* | RRA* | SHS** | LAS** |
| DCP* | | Abso | | KKA | 303 | LAS |
| +1C | NOP* | NOP* | | NOP* | SHY** | LDY |
| NOP* | - | | lute,x | NOT | 5111 | LDT |
| +1d | ORA | AND | • | ADC | STA | LDA |
| CMP | | Abso | | 100 | OTA | LDA |
| +1e | ASL | ROL | | ROR | SHX**v) | LDX y) |
| DEC | | Abso | | | ., ,, | |
| +1f | | RLA* | • | RRA* | SHA**v) | LAX* y) |
| DCP* | | Abso | | | ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,, | J J |
| | | | , | | | |

ROR intruction is available on MC650x microprocessors after June, 1976.

Legend:

- t Jams the machine
- *t Jams very rarely
- * Undocumented command
- ** Unusual operation

- y) indexed using Y instead of X
- () indirect instead of absolute

Note that the NOP instructions do have other addressing modes than the implied addressing. The NOP instruction is just like any other load instruction, except it does not store the result anywhere nor affects the flags.

6502 Registers

The NMOS 65xx processors are not ruined with too many registers. In addition

to that, the registers are mostly 8-bit. Here is a brief description of each register:

PC Program Counter

This register points the address from which the next instruction byte (opcode or parameter) will be fetched. Unlike other registers, this one is 16 bits in length. The low and high 8-bit halves of the register are called PCL and PCH, respectively. The Program Counter may be read by pushing its value on the stack. This can be done either by jumping to a subroutine or by causing an interrupt.

S Stack pointer

The NMOS 65xx processors have 256 bytes of stack memory, ranging

from \$0100 to \$01FF. The S register is a 8-bit offset to the stack page. In other words, whenever anything is being pushed on the stack, it will be stored to the address \$0100+S.

The Stack pointer can be read and written by transfering its value to or from the index register X (see below) with the TSX and TXS instructions.

P Processor status

This 8-bit register stores the state of the processor. The bits in this register are called flags. Most of the flags have something to do with arithmetic operations. The P register can be read by pushing it on the stack (with PHP or by causing an interrupt). If you only need to read one flag, you can use the branch instructions. Setting the flags is possible by pulling the P register from stack or by using the flag set or clear instructions.

Following is a list of the flags, starting from the 8th bit of the P register (bit 7, value \$80):

N Negative flag

This flag will be set after any arithmetic operations (when any of the registers A, X or Y is being loaded with a value). Generally, the N flag will be copied from the topmost bit of the register being loaded.

Note that TXS (Transfer X to S) is not an arithmetic operation. Also note that the BIT instruction affects the Negative flag just like arithmetic operations. Finally, the Negative flag behaves differently in Decimal operations (see description below).

V oVerflow flag

Like the Negative flag, this flag is intended to be used with 8-bit signed integer numbers. The flag will be affected by addition and subtraction, the instructions PLP, CLV and BIT, and the hardware signal -SO. Note that there is no SEV instruction, even though the MOS engineers loved to use East European abbreviations, like DDR (Deutsche Demokratische Republik vs. Data Direction Register). (The Russian abbreviation for their former trade association COMECON is SEV.) The -SO (Set Overflow) signal is available on some processors, at least the 6502, to set the V flag. This enables response to an I/O activity in equal or less than three clock cycles when using a BVC instruction branching to itself (\$50 \$FE).

The CLV instruction clears the V flag, and the PLP and

BIT instructions copy the flag value from the bit 6 of the topmost stack entry or from memory.

After a binary addition or subtraction, the V flag will be set on a sign overflow, cleared otherwise. What is a sign overflow? For instance, if you are trying to add 123 and 45 together, the result (168) does not fit in a 8-bit signed integer (upper limit 127 and lower limit -128). Similarly, adding -123 to -45 causes the overflow, just like subtracting -45 from 123 or 123 from -45 would do.

Like the N flag, the V flag will not be set as expected in the Decimal mode. Later in this document is a precise operation description.

A common misbelief is that the V flag could only be set by arithmetic operations, not cleared.

1 unused flag

To the current knowledge, this flag is always 1.

B Break flag

This flag is used to distinguish software (BRK) interrupts from hardware interrupts (IRQ or NMI). The B flag is always set except when the P register is being pushed on stack when jumping to an interrupt routine to process only a hardware interrupt.

The official NMOS 65xx documentation claims that the BRK instruction could only cause a jump to the IRQ vector (\$FFFE). However, if an NMI interrupt occurs while executing a BRK instruction, the processor will jump to the NMI vector (\$FFFA), and the P register will be pushed on the stack with the B flag set.

D Decimal mode flag

This flag is used to select the (Binary Coded) Decimal mode for addition and subtraction. In most applications, the flag is zero.

The Decimal mode has many oddities, and it operates differently on CMOS processors. See the description of the ADC, SBC and ARR instructions below.

I Interrupt disable flag

This flag can be used to prevent the processor from jumping to the IRQ handler vector (\$FFFE) whenever the hardware line -IRQ is active. The flag will be automatically set after taking an interrupt, so that the processor would not keep jumping to the interrupt routine if the -IRQ signal remains low for several clock cycles.

Z Zero flag

The Zero flag will be affected in the same cases than the Negative flag. Generally, it will be set if an arithmetic register is being loaded with the value zero, and cleared otherwise. The flag will behave differently in Decimal operations.

C Carry flag

This flag is used in additions, subtractions, comparisons and bit rotations. In additions and subtractions, it acts as a 9th bit and lets you to chain operations to calculate with bigger than 8-bit numbers. When subtracting, the Carry flag is the negative of Borrow: if an overflow occurs, the flag will be clear, otherwise set. Comparisons are a special case of subtraction: they assume Carry flag set and Decimal flag clear, and do not store the result of the subtraction anywhere.

There are four kinds of bit rotations. All of them store the bit that is being rotated off to the Carry flag. The left shifting instructions are ROL and ASL. ROL copies the initial Carry flag to the lowmost bit of the byte; ASL always clears it. Similarly, the ROR and LSR instructions shift to the right.

A Accumulator

The accumulator is the main register for arithmetic and logic operations. Unlike the index registers X and Y, it has a direct connection to the Arithmetic and Logic Unit (ALU). This is why many operations are only available for the accumulator, not the index registers.

X Index register X

This is the main register for addressing data with indices. It has a special addressing mode, indexed indirect, which lets you to have a vector table on the zero page.

Y Index register Y

The Y register has the least operations available. On the other hand, only it has the indirect indexed addressing mode that enables access to any memory place without having to use self-modifying code.

6510/8502 Undocumented Commands

-- A brief explanation about what may happen while using don't care states.

| ANE \$8B | A = (A #\$EE) & X & #byte | | | |
|----------|----------------------------------|--|--|--|
| | same as | | | |
| | A = ((A & #\$11 & X) (#\$EE & | | | |

X)) & #byte

In real 6510/8502 the internal parameter #\$11 may occasionally be #\$10, #\$01 or even #\$00. This occurs when the video chip starts DMA between the opcode fetch and the parameter fetch of the instruction. The value probably depends on the data that was left on the bus by the VIC-II.

LXA AB C=Lehti: A = X = ANE Alternate: A = X = (A & #byte)

TXA and TAX have to be responsible for these.

```
      SHA $93,$9F
      Store (A & X & (ADDR_HI + 1))

      SHX $9E
      Store (X & (ADDR_HI + 1))

      SHY $9C
      Store (Y & (ADDR_HI + 1))

      SHS $9B
      SHA and TXS, where X is

      replaced by (A & X).
      Store (X & X & (ADDR_HI + 1))
```

Note: The value to be stored is copied also to ADDR_HI if page boundary is crossed.

- SBX \$CB Carry and Decimal flags are ignored but the Carry flag will be set in substraction. This is due to the CMP command, which is executed instead of the real SBC.
- ARR \$6B This instruction first performs an AND between the accumulator and the immediate parameter, then it shifts the accumulator to the right. However, this is not the whole truth. See the description below.

Many undocumented commands do not use AND between registers, the CPU

just throws the bytes to a bus simultaneously and lets the open-collector drivers perform the AND. I.e. the command called 'SAX', which is in the STORE section (opcodes \$A0...\$BF), stores the result of (A & X) by this way.

More fortunate is its opposite, 'LAX' which just loads a byte simultaneously into both A and X.

\$6B ARR

This instruction seems to be a harmless combination of AND and ROR at first sight, but it turns out that it affects the V flag and also has a special kind of decimal mode. This is because the instruction has inherited some properties of the ADC instruction (\$69) in addition to the ROR (\$6A).

In Binary mode (D flag clear), the instruction effectively does an AND between the accumulator and the immediate parameter, and then shifts the accumulator to the right, copying the C flag to the 8th bit. It sets the Negative and Zero flags just like the ROR would. The ADC code shows up in the Carry and oVerflow flags. The C flag will be copied from the bit 6 of the result (which doesn't seem too logical), and the V flag is the result of an Exclusive OR operation between the bit 6 and the bit 5 of the result. This makes sense, since the V flag will be normally set by an Exclusive OR, too.

In Decimal mode (D flag set), the ARR instruction first performs the AND and ROR, just like in Binary mode. The N flag will be copied from the initial C flag, and the Z flag will be set according to the ROR result, as expected. The V flag will be set if the bit 6 of the accumulator changed its state between the AND and the ROR, cleared otherwise.

Now comes the funny part. If the low nybble of the AND result, incremented by its lowmost bit, is greater than 5, the low nybble in the ROR result will be incremented by 6. The low nybble may overflow as a consequence of this BCD fixup, but the high nybble won't be adjusted. The high nybble will be BCD fixed in a similar way. If the high nybble of the AND result, incremented by its lowmost bit, is greater than 5, the high nybble in the ROR result will be incremented by 6, and the Carry flag will be set. Otherwise the C flag will be cleared.

To help you understand this description, here is a C routine that illustrates the ARR operation in Decimal mode:

unsigned A, /* Accumulator */ AL, /* low nybble of accumulator */ AH, /* high nybble of accumulator */ C, /* Carry flag */

Z, /* Zero flag */ V, /* oVerflow flag */ N, /* Negative flag */ t, /* temporary value */ s; /* value to be ARRed with Accumulator */ /* Perform the t = A & s;AND. */ /* Separate AH = t >> 4;the high */ AL = t & 15;/* and low nybbles. */ N = C;/* Set the N and */ Z = !(A = (t >> 1) | (C << 7)); /* Z flagstraditionally */ $V = (t ^ A) \& 64;$ /* and V flag in a weird way. */ if (AL + (AL & 1) > 5) /* BCD "fixup" for low nybble. */ $A = (A \& 0 \times F0) | ((A + 6) \& 0 \times F);$ if (C = AH + (AH & 1) > 5) /* Set the Carry flag. */ A = (A + 0x60) & 0xFF; /* BCD "fixup" for high nybble. */ SBX X <- (A & X) - Immediate \$CB

The 'SBX' (\$CB) may seem to be very complex operation, even though it is a combination of the subtraction of accumulator and parameter, as in the 'CMP' instruction, and the command 'DEX'. As a result, both A and X are connected to ALU but only the subtraction takes place. Since the comparison logic was used, the result of subtraction should be normally ignored, but the 'DEX' now happily stores to X the value of (A & X) - Immediate. That is why this instruction does not have any decimal mode, and it does not affect the V flag. Also Carry flag will be ignored in the subtraction but set according to the result.

Proof:

```
begin 644 vsbx
M`0@9$,D'GL(H-#,IJC(U-JS"*#0T*:HR-
@```*D`H#V1*Z`_D2N@09$KJ0>%
M^QBE^VEZJ+$KH#F1*ZD`2"BI`*(`RP`(:-
B@.5$K*4#P`E@`H#VQ*SAI`)$K
JD-Z@/[$K:0"1*Y#4J2X@TO\XH$&Q*VD`D2N0Q,;
[$+188/_^]_:_OK>V
```

end

and

```
begin 644 sbx
M`0@9$,D'GL(H-#,IJC(U-JS"*#0T*:HR-@```'BI`*!-
D2N@3Y$KH%&1*ZD#
MA?L8I?M*2)`#J1@LJ3B@29$K:$J0`ZGX+*G8R)$K&/BXJ?
2B8\L)AOP(:(7]
MV#B@3;$KH$\Q*Z!1\2L(1?SP`0!H1?
TIM]#XH$VQ*SAI`)$KD,N@3[$K:0"1
9*Y#!J2X@T0\XH%&Q*VD`D2N0L<;[$))88-#X</pre>
```

end

These test programs show if your machine is compatible with ours regarding the opcode \$CB. The first test, vsbx, proves that SBX does not affect the V flag. The latter one, sbx, proves the rest of our theory. The vsbx test tests 33554432 SBX combinations (16777216 different A, X and Immediate combinations, and two different V flag states), and the sbx test doubles that amount (16777216*4 D and C flag combinations). Both tests have run successfully on a C64 and a Vic20.

They ought to run on C16, +4 and the PET series as well. The tests stop with BRK, if the opcode \$CB does not work as expected. Successful operation ends in RTS. As the tests are very slow, they print dots on the screen while running so that you know that the machine has not jammed. On computers running at 1 MHz, the first test prints approximately one dot every four seconds and a total of 2048 dots, whereas the second one prints half that amount, one dot every seven seconds.

If the tests fail on your machine, please let us know your processor's part number and revision. If possible, save the executable (after it has stopped with BRK) under another name and send it to us so that we know at which stage the program stopped.

The following program is a Commodore 64 executable that Marko M"akel"a

developed when trying to find out how the V flag is affected by SBX. (It was believed that the SBX affects the flag in a weird way, and this program shows how SBX sets the flag differently from SBC.) You may find the subroutine at \$C150 useful when researching other undocumented instructions' flags. Run the program in a machine language monitor, as it makes use of the BRK instruction. The result tables will be written on pages \$C2 and \$C3.

begin 644 sbx-c100
M`,%XH`",#L&,\$,&,\$L&XJ8*B@LL7AOL(:
(7\N#BM#L\$M\$,'M\$L\$(Q?OP`B@`
M:\$7\\`,@4,'N#L'0U.X0P=#/SB#0[A+!T,
<````)BJ\!>M#L\$M
L\$,'=_\'0":T2P=W_PM`!8,K0Z:T.P2T0P9D`PID`!*T2P9D`PYD`!

Other undocumented instructions usually cause two preceding opcodes being executed. However 'NOP' seems to completely disappear from 'SBC'

code \$EB.

The most difficult to comprehend are the rest of the instructions

located on the '\$0B' line.

All the instructions located at the positive (left) side of this line should rotate either memory or the accumulator, but the addressing mode turns out to be immediate! No problem. Just read the operand, let it be ANDed with the accumulator and finally use accumulator addressing mode for the instructions above them.

RELIGION_MODE_ON

/* This part of the document is not accurate. You can read it as a fairy tale, but do not count on it when performing your own measurements. */

The rest two instructions on the same line, called 'ANE' and 'LXA' (\$8B and \$AB respectively) often give quite unpredictable results. However, the most usual operation is to store ((A | #\$ee) & X & #\$nn) to accumulator. Note that this does not work reliably in a real 64! In the Commodore 128 the opcode \$8B uses values 8C, CC, EE, and occasionally 0C and 8E for the OR instead of EE,EF,FE and FF used in the C64. With a C128 running at 2 MHz #\$EE is always used. Opcode \$AB

does not cause this OR taking place on 8502 while 6510 always performs it. Note that this behaviour depends on processor and/or video chip revision.

Let's take a closer look at \$8B (6510).

A <- X & D & (A | VAL)

where VAL comes from this table:

X high D high D low VAL \$EE (1) even even --odd \$EE even --odd \$EE even --odd \$EE odd 0 odd odd not 0 \$FE (2) (1) If the bottom 2 bits of A are both 1, then the LSB of the result may be 0. The values of X and D are different every time I run the test. This appears to be very rare.

(2) VAL is \$FE most of the time. Sometimes it is \$EE - it seems to be random,

not related to any of the data. This is much more common than (1).

In decimal mode, VAL is usually \$FE.

Two different functions have been discovered for LAX, opcode \$AB. One is A = X = ANE (see above) and the other, encountered with 6510 and 8502, is less complicated A = X = (A & #byte). However, according to what is reported, the version altering only the lowest bits of each nybble seems to be more common.

What happens, is that \$AB loads a value into both A and X, ANDing the low bit of each nybble with the corresponding bit of the old A. However, there are exceptions. Sometimes the low bit is cleared even when A contains a '1', and sometimes other bits are cleared. The exceptions seem random (they change every time I run the test). Oops - that was in decimal mode. Much the same with D=0.

What causes the randomness? Probably it is that it is marginal logic levels - when too much wired-anding goes on, some of the signals get very close to the threshold. Perhaps we're seeing some of them step over it. The low bit of each nybble is special, since it has to cope with carry differently (remember decimal mode). We never see a '0' turn into a '1'.

Since these instructions are unpredictable, they should not be used.

There is still very strange instruction left, the one named SHA/X/Y, which is the only one with only indexed addressing modes. Actually, the commands 'SHA', 'SHX' and 'SHY' are generated by the indexing algorithm.

While using indexed addressing, effective address for page boundary crossing is calculated as soon as possible so it does not slow down operation. As a result, in the case of SHA/X/Y, the address and data are processed at the same time making AND between them to take place.

Thus, the value to be stored by SAX, for example, is in fact (A & X & (ADDR_HI + 1)). On page boundary crossing the same value is copied also to high byte of the effective address.

RELIGION_MODE_OFF

Register selection for load and store

| bit1 | bit0 | А | Х | Y |
|------|------|---|---|---|
| Θ | Θ | | | Х |
| Θ | 1 | | Х | |
| 1 | Θ | Х | | |
| 1 | 1 | Х | Х | |

So, A and X are selected by bits 1 and 0 respectively, while \sim (bit1|bit0) enables Y.

Indexing is determined by bit4, even in relative addressing mode, which is one kind of indexing.

Lines containing opcodes xxx000x1 (01 and 03) are treated as absolute after the effective address has been loaded into CPU.

Zeropage, y and Absolute, y (codes 10x1 x11x) are distinguished by bit5.

Decimal mode in NMOS 6500 series

Most sources claim that the NMOS 6500 series sets the N, V and Z flags unpredictably when performing addition or subtraction in decimal mode. Of course, this is not true. While testing how the flags are

set, I also wanted to see what happens if you use illegal BCD values.

ADC works in Decimal mode in a quite complicated way. It is amazing how it can do that all in a single cycle. Here's a C code version of the instruction:

unsigned A, /* Accumulator */ AL, /* low nybble of accumulator */ AH, /* high nybble of accumulator */ C, /* Carry flag */ Z, /* Zero flag */ V, /* oVerflow flag */ N, /* Negative flag */ s; /* value to be added to Accumulator */ AL = (A & 15) + (S & 15) + C;/* Calculate the lower nybble. */ AH = (A >> 4) + (S >> 4) + (AL > 15); /*Calculate the upper nybble. */ if (AL > 9) AL += 6;/* BCD fixup for lower nybble. */ Z = ((A + s + C) & 255 != 0);/* Zero flag is set just like in Binary mode. */ /* Negative and Overflow flags are set with the same logic than in Binary mode, but after fixing the lower nybble. */ N = (AH & 8 != 0);

 $V = ((AH << 4) \land A) \& 128 \&\& !((A \land s) \& 128);$

/* Carry is the only flag set after fixing the result. */

C = (AH > 15); A = ((AH << 4) | (AL & 15)) & 255;

The C flag is set as the quiche eaters expect, but the N and V flags are set after fixing the lower nybble but before fixing the upper one. They use the same logic than binary mode ADC. The Z flag is set before any BCD fixup, so the D flag does not have any influence on it.

Proof: The following test program tests all 131072 ADC combinations in Decimal mode, and aborts with BRK if anything breaks this theory. If everything goes well, it ends in RTS.

```
begin 600 dadc
M 0@9",D'GL(H-#,IJC(U-JS"*#0T*:HR-@ 'BI&*
A/N$_$B@+)$KH(V1
M*Q@(I?PI#X7]I?LI#V7]R0J0 FD%J"D/A?VE^RGP9?PI\ C $)
":0^JL @H
ML ?)H) &""@X:5\X!?V%_0AH*3W@ ! ""8"HBD7[$ JE^T7\,
28"4"H**7[
M9?S0!)@) J@8N/BE^V7\V A%_= G:(3]1?W0(.;[T(?F_-"#:$D8\)
)88*D=
0&&4KA?NI &4LA?RI.&S[ A%
```

end

All programs in this chapter have been successfully tested on a Vic20 and a Commodore 64 and a Commodore 128D in C64 mode. They should run on

C16, +4 and on the PET series as well. If not, please report the problem to Marko M"akel"a. Each test in this chapter should run in less than a

minute at 1 MHz.

SBC is much easier. Just like CMP, its flags are not affected by the D flag.

Proof:

```
begin 600 dsbc-cmp-flags
M 0@9",D'GL(H-#,IJC(U-JS"*#0T*:HR-@ 'B@ (3[A/RB
XH8:66HL2N@
M09$KH$R1*XII::BQ*Z!%D2N@4)$K^#BXI?OE_-@(:
(7].+BE^^7\"&A%_? !
5 .;[T./F_-#?RA"_8!@X&#CEY<7%</pre>
```

end

The only difference in SBC's operation in decimal mode from binary mode

is the result-fixup:

unsigned A, /* Accumulator */ AL, /* low nybble of accumulator */ AH, /* high nybble of accumulator */ C, /* Carry flag */ Z, /* Zero flag */ V, /* oVerflow flag */ N, /* Negative flag */ s; /* value to be added to Accumulator */ AL = (A & 15) - (s & 15) - !C; /* Calculate the lower nybble. */ if (AL & 16) AL -= 6; /* BCD fixup for lower nybble. */ AH = (A >> 4) - (s >> 4) - (AL & 16); /*Calculate the upper nybble. */

/* The flags are set just like in Binary mode.
*/

C = (A - s - !C) & 256 != 0; Z = (A - s - !C) & 255 != 0; $V = ((A - s - !C) \land s) \& 128 \&\& (A \land s) \& 128;$ N = (A - s - !C) & 128 != 0;A = ((AH << 4) | (AL & 15)) & 255;

Again Z flag is set before any BCD fixup. The N and V flags are set at any time before fixing the high nybble. The C flag may be set in any phase.

Decimal subtraction is easier than decimal addition, as you have to make the BCD fixup only when a nybble overflows. In decimal addition, you had to verify if the nybble was greater than 9. The processor has an internal "half carry" flag for the lower nybble, used to trigger the BCD fixup. When calculating with legal BCD values, the lower nybble cannot overflow again when fixing it.

So, the processor does not handle overflows while performing the fixup. Similarly, the BCD fixup occurs in the high nybble only if the value overflows, i.e. when the C flag will be cleared.

Because SBC's flags are not affected by the Decimal mode flag, you could guess that CMP uses the SBC logic, only setting the C flag first. But the SBX instruction shows that CMP also temporarily clears the D flag, although it is totally unnecessary.

The following program, which tests SBC's result and flags, contains the 6502 version of the pseudo code example above.

```
begin 600 dsbc
M 0@9",D'GL(H-#,IJC(U-JS"*#0T*:HR-@ 'BI&*
A/N$_$B@+)$KH':1
M*S@(I?PI#X7]I?LI#^7]L /I!1@I#ZBE_"GPA?VE^RGP"#CE_2GPL
KI7RBP
M#ND/.+ )*+ &Z0^P NE?A/T%_87]*+BE^^7\"&BH.+CXI?OE_-
@(1?W0FVB$
8_47]T)3F^]">YOS0FFA)&- $J3C0B%A@
```

```
end
```

Obviously the undocumented instructions RRA (ROR+ADC) and ISB (INC+SBC) have inherited also the decimal operation from the official instructions ADC and SBC. The program droradc proves this statement for ROR, and the dincsbc test proves this for ISB. Finally, dincsbc-deccmp proves that ISB's and DCP's (DEC+CMP) flags are not affected by the D flag.

```
begin 644 droradc
M`0@9",D'GL(H-#,IJC(U-JS"*#0T*:HR-
@```'BI&*``A/N$_$B@+)$KH(V1
M*S@(I?PI#X7]I?LI#V7]R0J0`FD%J"D/A?VE^RGP9?
PI\`C`$)`":0^JL`@H
ML`?)H)`&""@X:5\X!?
V%_0AH*3W@`!`""8"HBD7[$`JE^T7\,`28"4"H**7[
M9?
S0!)@)`J@XN/BE^R;\9_S8"$7]T"=HA/U%_=`@Y000A>;\T(%H21CP
2J1T892N%^ZD`92R%_*DX;/L`
```

```
end
```

```
begin 644 dincsbc
M`0@9",D'GL(H-#,IJC(U-JS"*#0T*:HR-
@```'BI&*``A/N$_$B@+)$KH':1
M*S@(I?PI#X7]I?LI#^7]L`/I!1@I#ZBE_"GPA?
VE^RGP"#CE_2GPL`KI7RBP
M#ND/.+`)*+`&Z0^P`NE?A/T%_87]*+BE^^7\"&BH.+CXI?0&_.?
\V`A%_="9
```

::(3]1?WODN;[T)SF_-"8:\$D8T`2I.-"&6&#\

end

```
begin 644 dincsbc-deccmp
M`0@9",D'GL(H-#,IJC(U-JS"*#0T*:HR-
@```'B@`(3[A/RB`XH8:7>HL2N@
M3Y$KH%R1*XII>ZBQ*Z!3D2N@8)$KBFE_J+$KH%61*Z!BD2OX.+BE^
L"&B%_3BXI?OF_,?\"&A%_?`!`.;[T-_F_-
#;RA"M8!@X&#CFYL;&Q\?GYP#8
``
```

end

6510 features

 O PHP always pushes the Break (B) flag as a `1' to the stack. Jukka Tapanim"aki claimed in C=lehti issue 3/89, on page 27 that the processor makes a logical OR between the status register's bit 4 and the bit 8 of the stack pointer register (which is always 1). He did not give any reasons for this argument, and has refused to clarify

it afterwards. Well, this was not the only error in his article...

o Indirect addressing modes do not handle page boundary crossing at all.

When the parameter's low byte is \$FF, the effective address wraps around and the CPU fetches high byte from \$xx00 instead of \$xx00+\$0100.

E.g. JMP (\$01FF) fetches PCL from \$01FF and PCH from \$0100, and LDA (\$FF),Y fetches the base address from \$FF and \$00.

o Indexed zero page addressing modes never fix the page address on crossing the zero page boundary.

E.g. LDX #\$01 : LDA (\$FF,X) loads the effective address from \$00 and \$01.

o The processor always fetches the byte following a relative branch

instruction. If the branch is taken, the processor reads then the opcode from the destination address. If page boundary is crossed, it first reads a byte from the old page from a location that is bigger or smaller than the correct address by one page.

- o If you cross a page boundary in any other indexed mode, the processor reads an incorrect location first, a location that is smaller by one page.
- Read-Modify-Write instructions write unmodified data, then modified (so INC effectively does LDX loc;STX loc;INX;STX loc)
- o -RDY is ignored during writes
 (This is why you must wait 3 cycles before doing any DMA -the maximum number of consecutive writes is 3, which occurs during interrupts except -RESET.)

o Some undefined opcodes may give really unpredictable results.

o All registers except the Program Counter remain unmodified after - RESET.

(This is why you must preset D and I flags in the RESET handler.)

Different CPU types

The Rockwell data booklet 29651N52 (technical information about R65C00

microprocessors, dated October 1984), lists the following differences between

NMOS R6502 microprocessor and CMOS R65C00 family:

 Indexed addressing across page boundary. NMOS: Extra read of invalid address. CMOS: Extra read of last instruction byte. 2. Execution of invalid op codes.

NMOS: Some terminate only by reset. Results are undefined. CMOS: All are NOPs (reserved for future use).

- Jump indirect, operand = XXFF.
 NMOS: Page address does not increment.
 CMOS: Page address increments and adds one additional cycle.
- Read/modify/write instructions at effective address. NMOS: One read and two write cycles. CMOS: Two read and one write cycle.
- 5. Decimal flag.

NMOS: Indeterminate after reset. CMOS: Initialized to binary mode (D=0) after reset and interrupts.

- Flags after decimal operation.
 NMOS: Invalid N, V and Z flags.
 CMOS: Valid flag adds one additional cycle.
- Interrupt after fetch of BRK instruction.
 NMOS: Interrupt vector is loaded, BRK vector is ignored.
 CMOS: BRK is executed, then interrupt is executed.

6510 Instruction Timing

The NMOS 6500 series processors always perform at least two reads for each instruction. In addition to the operation code (opcode), they fetch the next byte. This is quite efficient, as most instructions are two or three bytes long. The processors also use a sort of pipelining. If an instruction does not store data in memory on its last cycle, the processor can fetch the opcode of the next instruction while executing the last cycle. For instance, the instruction EOR #\$FF truly takes three cycles. On the first cycle, the opcode \$49 will be fetched. During the second cycle the processor decodes the opcode and fetches the parameter #\$FF. On the third cycle, the processor will perform the operation and store the result to accumulator, but simultaneously it fetches the opcode for the next instruction. This is why the instruction effectively takes only two cycles.

The following tables show what happens on the bus while executing different kinds of instructions.

Interrupts

NMI and IRQ both take 7 cycles. Their timing diagram is much like BRK's (see below). IRQ will be executed only when the I flag is clear. IRQ and BRK both set the I flag, whereas the NMI does not affect its state.

The processor will usually wait for the current instruction to complete before executing the interrupt sequence. To process the interrupt before the next instruction, the interrupt must occur before the last cycle of the current instruction.

There is one exception to this rule: the BRK instruction. If a hardware interrupt (NMI or IRQ) occurs before the fourth (flags saving) cycle of BRK, the BRK instruction will be skipped, and the processor will jump to the hardware interrupt vector. This sequence will always take 7 cycles.

You do not completely lose the BRK interrupt, the B flag will be set in the pushed status register if a BRK instruction gets interrupted. When BRK and IRQ occur at the same time, this does not cause any problems, as your program will consider it as a BRK, and the IRQ would occur again after the processor returned from your BRK routine, unless you cleared the interrupt source in your BRK handler. But the simultaneous occurrence of NMI and BRK is far more fatal. If you do not check the B flag in the NMI routine and subtract two from the return address when needed, the BRK instruction will be skipped.

If the NMI and IRQ interrupts overlap each other (one interrupt occurs before fetching the interrupt vector for the other interrupt), the processor will most probably jump to the NMI vector in every case, and then jump to the IRQ vector after processing the first instruction of the NMI handler. This has not been measured yet, but the IRQ is very similar to BRK, and many sources state that the NMI has higher priority than IRQ. However, it might be that the processor takes the interrupt that comes later, i.e. you could lose an NMI interrupt if an IRQ occurred in four cycles after it.

After finishing the interrupt sequence, the processor will start to execute the first instruction of the interrupt routine. This proves that the processor uses a sort of pipelining: it finishes the current instruction (or interrupt sequence) while reading the opcode of the next instruction.

RESET does not push program counter on stack, and it lasts probably 6 cycles after deactivating the signal. Like NMI, RESET preserves all registers except PC.

Instructions accessing the stack

BRK

| # | address | R/W | description |
|---|---------|-----|---------------------------------|
| | | | |
| 1 | PC | R | fetch opcode, increment PC |
| 2 | PC | R | read next instruction byte (and |

throw it away), increment PC 3 \$0100,S W push PCH on stack (with B flag set), decrement S \$0100,S push PCL on stack, decrement S 4 W push P on stack, decrement S \$0100,S 5 W \$FFFE R fetch PCL 6 7 \$FFFF R fetch PCH RTI # address R/W description - - -. fetch opcode, increment PC PC 1 R read next instruction byte (and 2 PC R throw it away) \$0100,S 3 R increment S pull P from stack, increment S 4 \$0100,S R pull PCL from stack, increment 5 \$0100,S R S pull PCH from stack \$0100,S R 6 RTS # address R/W description fetch opcode, increment PC PC R 1 R read next instruction byte (and 2 PC throw it away) R increment S 3 \$0100,S \$0100,S R pull PCL from stack, increment 4 S \$0100,S R pull PCH from stack 5 6 PC R increment PC

PHA, PHP

address R/W description _ _ _ _ _ _ PC R fetch opcode, increment PC PC R read next instruction byte 1 R read next instruction byte (and 2 PC throw it away) \$0100,S W push register on stack, 3 decrement S PLA, PLP # address R/W description - - - - - - - - - - - - - - - -PC R fetch opcode, increment PC 1 R read next instruction byte (and 2 PC throw it away) 3 \$0100,S R increment S \$0100,S R pull register from stack 4 JSR # address R/W description --- ----- --- --------1 PC R fetch opcode, increment PC 2 PC R fetch low address byte, increment PC R internal operation \$0100,S 3 (predecrement S?) W push PCH on stack, decrement S \$0100,S 4 push PCL on stack, decrement S \$0100,S 5 W copy low address byte to PCL, PC 6 R fetch high address byte to PCH

Accumulator or implied addressing

address R/W description

1 PC R fetch opcode, increment PC PC R read next instruction byte (and 2 throw it away) Immediate addressing # address R/W description -----PC R fetch opcode, increment PC 1 R fetch value, increment PC 2 PC Absolute addressing JMP # address R/W description --- -----PC R fetch opcode, increment PC 1 PC R fetch low address byte, 2 increment PC R copy low address byte to PCL, 3 PC fetch high address byte to PCH Read instructions (LDA, LDX, LDY, EOR, AND, ORA, ADC, SBC, CMP, BIT, LAX, NOP) # address R/W description fetch opcode, increment PC PC 1 R R fetch low byte of address, 2 PC increment PC PC R fetch high byte of address, 3

increment PC 4 address R read from effective address Read-Modify-Write instructions (ASL, LSR, ROL, ROR, INC, DEC, SLO, SRE, RLA, RRA, ISB, DCP) # address R/W description -----R fetch opcode, increment PC 1 PC PC R fetch low byte of address, 2 increment PC R fetch high byte of address, PC 3 increment PC 4 address R read from effective address 5 address W write the value back to effective address, and do the operation on it W write the new value to 6 address effective address Write instructions (STA, STX, STY, SAX) # address R/W description PC R fetch opcode, increment PC 1 R fetch low byte of address, 2 PC increment PC R fetch high byte of address, 3 PC increment PC 4 address W write register to effective address Zero page addressing

Read instructions (LDA, LDX, LDY, EOR, AND, ORA,

ADC, SBC, CMP, BIT,

LAX, NOP)

address R/W description R fetch opcode, increment PC 1 PC R fetch address, increment PC PC 2 3 address R read from effective address Read-Modify-Write instructions (ASL, LSR, ROL, ROR, INC, DEC, SLO, SRE, RLA, RRA, ISB, DCP) # address R/W description PC R fetch opcode, increment PC 1 R fetch address, increment PC PC 2 3 address R read from effective address W write the value back to address 4 effective address, and do the operation on it W write the new value to 5 address effective address Write instructions (STA, STX, STY, SAX) address R/W description # --- ----- --- ------- - - - - - - - - -PC R fetch opcode, increment PC 1 PC R fetch address, increment PC 2 3 address W write register to effective address

Zero page indexed addressing

Read instructions (LDA, LDX, LDY, EOR, AND, ORA, ADC, SBC, CMP, BIT, LAX, NOP) address R/W description # _ _ _ _ _ _ _ _ _ _ PC R fetch opcode, increment PC 1 PC R fetch address, increment PC address R read from address, add index 2 3 register to it 4 address+I* R read from effective address Notes: I denotes either index register (X or Y). * The high byte of the effective address is always zero, i.e. page boundary crossings are not handled. Read-Modify-Write instructions (ASL, LSR, ROL, ROR, INC, DEC, SLO, SRE, RLA, RRA, ISB, DCP) # address R/W description -----R fetch opcode, increment PC 1 PC PC R fetch address, increment PC 2 3 address R read from address, add index register X to it 4 address+X* R read from effective address 5 address+X* W write the value back to effective address, and do the operation on it address+X* W write the new value to 6 effective address

Note: * The high byte of the effective address is always zero, i.e. page boundary crossings are not handled.

Write instructions (STA, STX, STY, SAX)

address R/W description
1 PC R fetch opcode, increment PC
2 PC R fetch address, increment PC
3 address R read from address, add index
register to it
4 address+I* W write to effective address

Notes: I denotes either index register (X or Y).

Absolute indexed addressing

Read instructions (LDA, LDX, LDY, EOR, AND, ORA, ADC, SBC, CMP, BIT,

LAX, LAE, SHS, NOP)

address R/W description
1 PC R fetch opcode, increment PC
2 PC R fetch low byte of address,
increment PC
3 PC R fetch high byte of address,
add index register to low

address byte, increment PC 4 address+I* R read from effective address, fix the high byte of effective address 5+ address+I R re-read from effective address Notes: I denotes either index register (X or Y). * The high byte of the effective address may be invalid at this time, i.e. it may be smaller by \$100. + This cycle will be executed only if the effective address was invalid during cycle #4, i.e. page boundary was crossed. Read-Modify-Write instructions (ASL, LSR, ROL, ROR, INC, DEC, SLO, SRE, RLA, RRA, ISB, DCP) # address R/W description -------. R fetch opcode, increment PC 1 PC PC R fetch low byte of address, 2 increment PC fetch high byte of address, PC 3 R add index register X to low address byte, increment PC address+X* R read from effective address, 4 fix the high byte of effective address

address+X R re-read from effective 5 address address+X W write the value back to 6 effective address, and do the operation on it write the new value to address+X W 7 effective address Notes: * The high byte of the effective address may be invalid at this time, i.e. it may be smaller by \$100. Write instructions (STA, STX, STY, SHA, SHX, SHY) address R/W description # fetch opcode, increment PC PC 1 R fetch low byte of address, 2 PC R increment PC fetch high byte of address, PC 3 R add index register to low address byte, increment PC address+I* R read from effective address, 4 fix the high byte of effective address address+I W write to effective address 5 Notes: I denotes either index register (X or Y). * The high byte of the effective address may be invalid at this time, i.e. it may be smaller by \$100. Because the processor cannot undo a write to an invalid

address, it always reads from the address first.

Relative addressing (BCC, BCS, BNE, BEQ, BPL, BMI, BVC, BVS)

address R/W description # --- ----. R fetch opcode, increment PC 1 PC R fetch operand, increment PC 2 PC R Fetch opcode of next 3 PC instruction, If branch is taken, add operand to PCL. Otherwise increment PC. PC* Fetch opcode of next 4+ R instruction. Fix PCH. If it did not change, increment PC. R Fetch opcode of next PC 5! instruction, increment PC.

Notes: The opcode fetch of the next instruction is included to this diagram for illustration purposes. When determining real execution times, remember to subtract the last cycle.

* The high byte of Program Counter (PCH) may be invalid at this time, i.e. it may be smaller or bigger by \$100.

+ If branch is taken, this cycle will be executed.

! If branch occurs to different page, this cycle will be executed.

Indexed indirect addressing

Read instructions (LDA, ORA, EOR, AND, ADC, CMP, SBC, LAX)

address R/W description . R fetch opcode, increment PC PC 1 PC R fetch pointer address, 2 increment PC pointer R read from the address, add 3 X to it 4 pointer+X R fetch effective address low R fetch effective address 5 pointer+X+1 high R read from effective address address 6

Note: The effective address is always fetched from zero page, i.e. the zero page boundary crossing is not handled.

Read-Modify-Write instructions (SLO, SRE, RLA, RRA, ISB, DCP)

address R/W description
1 PC R fetch opcode, increment PC
2 PC R fetch pointer address,
increment PC
3 pointer R read from the address, add
X to it

pointer+X R fetch effective address low 4 R fetch effective address pointer+X+1 5 high address R read from effective address 6 7 address W write the value back to effective address, and do the operation on it W write the new value to address 8 effective address Note: The effective address is always fetched from zero page, i.e. the zero page boundary crossing is not handled. Write instructions (STA, SAX) address R/W description # R fetch opcode, increment PC 1 PC PC R fetch pointer address, 2 increment PC pointer R read from the address, add 3 X to it 4 pointer+X R fetch effective address low pointer+X+1 R fetch effective address 5 high address W write to effective address 6 Note: The effective address is always fetched from zero page, i.e. the zero page boundary crossing is

Indirect indexed addressing

not handled.

Read instructions (LDA, EOR, AND, ORA, ADC, SBC, CMP)

address R/W description -- --------- - - - - - - - - - - - - - - -R fetch opcode, increment PC PC 1 R fetch pointer address, PC 2 increment PC R fetch effective address low pointer 3 pointer+1 R fetch effective address 4 high, add Y to low byte of effective address 5 address+Y* R read from effective address, fix high byte of effective address R read from effective address 6+ address+Y Notes: The effective address is always fetched from zero page, i.e. the zero page boundary crossing is not handled. * The high byte of the effective address may be invalid at this time, i.e. it may be smaller by \$100. + This cycle will be executed only if the effective address was invalid during cycle #5, i.e. page boundary was crossed. Read-Modify-Write instructions (SLO, SRE, RLA, RRA, ISB, DCP) # address R/W description

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

R fetch opcode, increment PC PC 1 2 R fetch pointer address, PC increment PC pointer R fetch effective address low 3 pointer+1 R fetch effective address 4 high, add Y to low byte of effective address address+Y* R read from effective 5 address, fix high byte of effective address address+Y R read from effective address 6 address+Y W write the value back to 7 effective address, and do the operation on it W write the new value to address+Y 8 effective address Notes: The effective address is always fetched from zero page, i.e. the zero page boundary crossing is not handled. * The high byte of the effective address may be invalid at this time, i.e. it may be smaller by \$100. Write instructions (STA, SHA) address R/W description # _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ R fetch opcode, increment PC 1 PC

2 PC R fetch pointer address, increment PC 3 pointer R fetch effective address low 4 pointer+1 R fetch effective address high,

add Y to low byte of effective address address+Y* R read from effective 5 address, fix high byte of effective address W write to effective address 6 address+Y Notes: The effective address is always fetched from zero page, i.e. the zero page boundary crossing is not handled. * The high byte of the effective address may be invalid at this time, i.e. it may be smaller

by \$100.

Absolute indirect addressing (JMP)

address R/W description #

-----_ _ _ _ _ _ _ _ _ _ _ R fetch opcode, increment PC 1 PC R fetch pointer address low, PC 2 increment PC R fetch pointer address high, PC 3 increment PC pointer R fetch low address to latch 4 pointer+1* R fetch PCH, copy latch to PCL 5

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _

Note: * The PCH will always be fetched from the same page than PCL, i.e. page boundary crossing is not handled.

How Real Programmers Acknowledge Interrupts

With RMW instructions:

; beginning of combined raster/timer interrupt routine LSR \$D019 ; clear VIC interrupts, read raster interrupt flag to C BCS raster ; jump if VIC caused an interrupt ; timer interrupt routine

Operational diagram of LSR \$D019:

data address R/W # ----_ _ _ _ _ _ _ _ - - -. PC fetch opcode 4E 1 R R fetch address low
R fetch address high 2 19 PC+1 PC+2 3 D0 R read memory W write the va \$D019 4 XX write the value back, 5 \$D019 ХХ rotate right xx/2 \$D019 W write the new value 6 back

The 5th cycle acknowledges the interrupt by writing the same value back. If only raster interrupts are used, the 6th cycle has no effect on the VIC. (It might acknowledge also some other interrupts.)

With indexed addressing:

; acknowledge interrupts to both CIAs LDX #\$10 LDA \$DCFD,X

Operational diagram of LDA \$DCFD,X: description data address R/W # - - - -- - -- - -- - - - - - -. R BD PC fetch opcode 1 R fetch address low 2 FD PC+1 R fetch address high, 3 DC PC+2 add X to address low read from address, 4 ΧХ \$DC0D R fix high byte of address R read from right \$DD0D 5 УΥ address ; acknowledge interrupts to CIA 2 LDX #\$10 STA \$DDFD,X Operational diagram of STA \$DDFD,X: description address # data R/W - - - -_ _ _ _ _ _ _ _ _ _ _ fetch opcode PC 1 9D R R R fetch address low 2 FD PC+1 fetch address high, 3 DC PC+2 add X to address low read from address, 4 XX \$DD0D R fix high byte of address write to right \$DE0D 5 ac W address With branch instructions: ; acknowledge interrupts to CIA 2 LDA #\$00 ; clear N flag JMP \$DD0A BPL \$DC9D ; branch DD0A

; return

DC9D

BRK

You need the following preparations to initialize the CIA registers:

LDA #\$91 ; argument of BPL STA \$DD0B LDA #\$10 ; BPL STA \$DD0A STA \$DD08 ; load the ToD values from the latches LDA \$DD0B ; freeze the ToD display LDA #\$7F STA \$DC0D ; assure that \$DC0D is \$00

Operational diagram of BPL \$DC9D:

| | # | data | address | R/W | description | | | | |
|--|------|--|---------|-----|-----------------------|--|--|--|--|
| | | | | | | | | | |
| | 1 | 10 | \$DD0A | R | fetch opcode | | | | |
| | 2 | 91 | \$DD0B | R | fetch argument | | | | |
| | 3 | XX | | | fetch opcode, add | | | | |
| argument to PCL | | | | | | | | | |
| | 4 | уу | \$DD9D | | fetch opcode, fix PCH | | | | |
| (| (5 | 00 | \$DC9D | R | fetch opcode) | | | | |
| ; acknowledge interrupts to CIA 1 LSR ; clear N flag JMP \$DCFA DCFA BPL \$DD0D DD0D BRK | | | | | | | | | |
| ; CIA 1 and | • | Again you need to set the ToD registers of | | | | | | | |
| | | Interrupt Control Register of CIA 2 first. | | | | | | | |
| C | Oper | perational diagram of BPL \$DD0D: | | | | | | | |
| | # | data | address | R/W | description | | | | |

. fetch opcode 1 10 \$DCFA R 2 11 \$DCFB fetch argument R fetch opcode, add 3 **\$DCFC** ΧХ R argument to PCL fetch opcode, fix PCH 4 \$DC0D R уу fetch opcode) (5 00 \$DD0D R ; acknowledge interrupts to CIA 2 automagically ; preparations LDA #\$7F STA \$DD0D ; disable all interrupt sources of CIA2 LDA \$DD0E ; ensure that \$DDOC AND #\$BE remains constant ; and stop the timer STA \$DD0E LDA #\$FD STA \$DD0C ; parameter of BPL LDA #\$10 ; BPL STA \$DD0B LDA #\$40 ; RTI/parameter of LSR STA \$DD0A LDA #\$46 ; LSR STA \$DD09 ; load the ToD values STA \$DD08 from the latches ; freeze the ToD LDA \$DD0B display LDA #\$09 STA \$0318 LDA #\$DD ; change NMI vector to STA \$0319 \$DD09 ; Try changing this LDA #\$FF instruction's operand STA \$DD05 ; (see comment below).

LDA #\$FF STA \$DD04 ; set interrupt frequency to 1/65536 cycles LDA \$DD0E AND #\$80 ORA #\$11 LDX #\$81 STX \$DD0D ; enable timer interrupt STA \$DD0E ; start timer LDA #\$00 ; To see that the interrupts really occur, STA \$D011 ; use something like this and see how DEC \$D020 LOOP ; changing the byte loaded to \$DD05 from BNE LOOP ; #\$FF to #\$0F changes the image. When an NMI occurs, the processor jumps to Kernal code, which jumps to (\$0318), which points to the following routine: DD09 ; clear N flag LSR \$40 ; Note: \$DD0A contains BPL \$DD0A RTI. Operational diagram of BPL \$DD0A: # data address R/W description ----- - - - - - -- - -. fetch opcode 1 10 \$DD0B R R fetch argument 11 \$DD0C 2

3 xx \$DD0D R fetch opcode, add argument to PCL

4 40 \$DD0A R fetch opcode, (fix

PCH)

With RTI:

; the fastest possible interrupt handler in the 6500 family ; preparations SEI ; disable ROM and LDA \$01 enable I/O AND #\$FD ORA #\$05 STA \$01 LDA #\$7F ; disable CIA 2's all STA \$DD0D interrupt sources LDA \$DD0E AND #\$BE ; ensure that \$DD0C remains constant ; and stop the timer STA \$DD0E LDA #\$40 ; store RTI to \$DD0C STA \$DD0C LDA #\$0C STA \$FFFA LDA #\$DD STA \$FFFB ; change NMI vector to \$DD0C ; Try changing this LDA #\$FF instruction's operand STA \$DD05 ; (see comment below). LDA #\$FF STA \$DD04 ; set interrupt frequency to 1/65536 cycles LDA \$DD0E AND #\$80 ORA #\$11 LDX #\$81 ; enable timer STX \$DD0D interrupt

STA \$DD0E ; start timer LDA #\$00 ; To see that the interrupts really occur, STA \$D011 ; use something like this and see how LOOP DEC \$D020 ; changing the byte loaded to \$DD05 from BNE LOOP ; #\$FF to #\$0F changes the image.

When an NMI occurs, the processor jumps to Kernal code, which jumps to (\$0318), which points to the following routine:

DD0C RTI

How on earth can this clear the interrupts? Remember, the processor always fetches two successive bytes for each instruction.

A little more practical version of this is redirecting the NMI (or IRQ) to your own routine, whose last instruction is JMP \$DD0C or JMP \$DC0C. If you want to confuse more, change the 0 in the address to a hexadecimal digit different from the one you used when writing the RTI.

Or you can combine the latter two methods:

DD09 LSR \$xx ; xx is any appropriate BCD value 00-59. BPL \$DCFC DCFC RTI

This example acknowledges interrupts to both CIAs.

If you want to confuse the examiners of your code, you can use any of these techniques. Although these examples use no undefined opcodes, they do not necessarily run correctly on CMOS processors. However, the RTI example should run on 65C02 and 65C816, and the latter branch instruction example might work as well.

The RMW instruction method has been used in some demos, others were

developed by Marko M"akel"a. His favourite is the automagical RTI method, although it does not have any practical applications, except for some time dependent data decryption routines for very complicated copy protections.

2008

This help file has been generated by the freeware version of <u>HelpNDoc</u>

PPU

NTSC 2C02 technical operation

Brad Taylor (big_time_software@hotmail.com) 1st release: Sept 25th, Y2K 2nd release: Jan 27th, 2K3 3rd release: Feb 4th, 2K3 4th release: Feb 19th, 2K3

This document describes the low-level operation and technical details of the 2C02, the NES's PPU. In general, it contains important information in regards to PPU timing, which no NES coder/emulator author should be without. This document assumes that you already understand the basics of how the PPU works, like how the playfield/object images are generated, and the behaviour of scroll/address counters during playfield rendering.

Alot of the concepts behind how the PPU works described here have been extracted from Nintendo's patent documentation (U.S.#4,824,106). With block diagrams of the PPU's architecture (and even some schematics), these papers will definetely aid in the comprehension of this complex device.

Since the first release, this document has been given a major overhaul. Most sections of the document have been reworked, and new information has been added just about everywhere. If you've read the old version of this document before, I recommend that you read this new one in it's entirity; there's new information even in sections which may look like they haven't changed much.

Topics discussed hereon are as follows.

- Video signal generation

- PPU base timing

- Miscellanious PPU info
- PPU memory access cycles
- Frame rendering details
- Scanline rendering details
- In-range object evaluation
- Details of playfield render pipeline
- Details of object pattern fetch & render
- Extra cycle frames
- The MMC3's scanline counter
- PPU pixel priority quirk
- Graphical enhancements

+----+

|History|

+----+

On the weekend of Sept. 25th, Y2K, I setup an experiment with my NTSC NES MB & my PC so's I could RE the PPU's timing. What I did was (using a PC interface) analyse the changes that occur on the PPU's address and data pins on every rising & falling edge of the PPU's clock. I was not planning on removing the PPU from the motherboard (yet), so basically I just kept everything intact (minus the stuff I added onto the MB so I could monitor the PPU's signals), and popped in a game, so that it would initialize the PPU for me (I used DK classics, since it was only taking somthing like 4 frames before it was turning on the background/sprites).

The only change I made was taking out the 21 MHz clock generator circuitry. To replace the clock signal, I connected a port controlled latch to the NES's main clock line instead. Now, by writing a 0 or a 1 out to an PC ISA port of my choice (I was using \$104), I was able to control the 21 MHz clockline of the NES. After I would create a rise or a fall on the NES's clock line, I would then read in the data that appeared on the PPU's address and data pins, which included monitoring what PPU registers the game read/wrote to (& the data that was read/written).

+----+

|Video signal generation|

+----+

A 21.48 MHz clock signal is fed into the PPU. This is the NES's main clock line, which is shared by the CPU.

Inside the PPU, the 21.48 MHz signal is used to clock a three-stage Johnson counter. The complimentery outputs of both master and slave portions of each stage are used to form 12 mutually exclusive output phases- all 3.58 MHz each (the NTSC colorburst). These 12 different phases form the basis of all color generation for the PPU's composite video output.

Naturally, when the user programs the lower 4-bits of a palette register, they are essentially selecting any 1 of 12 phases to be routed to the PPU's video out pin (this corresponds to chrominance (tint/hue) video information) when the appropriate pixel indexes it. Other chrominance combinations (0 & 13) are simply hardwired to a 1 or 0 to generate grayscale pixels.

Bits 4 & 5 of a palette entry selects 1 of 4 linear DC voltage offsets to apply to the selected chrominance signal (this corresponds to luminance (brightness) video information) for a pixel.

Chrominance values 14 & 15 yield a black pixel color, regardless of any luminance value setting.

Luminance value 0, mixed with chrominance value 13 yield a "blacker than black" pixel color. This super black pixel has an output voltage level close to the vertical/horizontal syncronization pulses. Because of this, some video monitors will display warped/distorted screens for games which use this color for black (Game Genie is the best example of this). Essentially what is happening is the video monitor's horizontal timing is compromised by what it thinks are extra syncronization pulses in the scanline. This is not damaging to the monitors which are effected by it, but use of the super black color should be avoided, due to the graphical distortion it causes. The amplitude of the selected chrominance signal (via the 4 lower bits of a palette register) remain constant regardless of bits 4 or 5. Thus it is not possible to adjust the saturation level of a particular color.

+----+

|PPU base timing|

+----+

Other than the 3-stage Johnson counter, the 21.48 MHz signal is not used directly by any other PPU hardware. Instead, the signal is divided by 4 to get 5.37 MHz, and is used as the smallest unit of timing in the PPU. All following references to PPU clock cycle (abbr. "cc") timing in this document will be in respect to this timing base, unless otherwise indicated.

- Pixels are rendered at the same rate as the base PPU clock. In other words, 1 clock cycle= 1 pixel.

- 341 PPU cc's make up the time of a typical scanline (or 341/3 CPU cc's).

- One frame consists of 262 scanlines. This equals 341*262 PPU cc's per frame (divide by 3 for # of CPU cc's).

+-----+

|PPU memory access cycles|

+-----+

All PPU memory access cycles are 2 clocks long, and can be made back-to-back (typically done during rendering). Here's how the access breaks down:

At the beginning of the access cycle, PPU address lines 8..13 are updated with the target address. This data remains here until the next time an access cycle occurs. The lower 8-bits of the PPU address lines are multiplexed with the data bus, to reduce the PPU's pin count. On the first clock cycle of the access, A0..A7 are put on the PPU's data bus, and the ALE (address latch enable) line is activated for the first half of the cycle. This loads the lower 8-bit address into an external 8-bit transparent latch strobed by ALE (74LS373 is used).

On the second clock cycle, the /RD (or /WR) line is activated, and stays active for the entire cycle. Appropriate data is driven onto the bus during this time.

+-----+

|Miscellanious PPU info|

+-----+

- Sprite DMA is 1536 clock cycles long (512 CPU cc's). 256 individual transfers are made from CPU memory to a temp register inside the CPU, then from the CPU's temp reg, to \$2004.

- The PPU makes NO external access to the PPU bus, unless the playfield or objects are enabled during a scanline outside vblank. This means that the PPU's address and data busses are dead while in this state.

- palette RAM is accessed internally during playfield rendering (i.e., the palette address/data is never put on the PPU bus during this time). Additionally, when the programmer accesses palette RAM via \$2006/7, the palette address accessed actually does show up on the PPU address bus, but the PPU's /RD & /WR flags are not activated. This is required; to prevent writing over name table data falling under the approprite mirrored area (since the name table RAM's address decoder simply consists of an inverter connected to the A13 line- effectively decoding all addresses in \$2000-\$3FFF).

- the VINT impulse (NMI) and bit \$2002.7 are set simultaniously. Reading \$2002 will reset bit 7, but it seems that the VINT flag goes down on it's own. Because of this, when the PPU generates a VINT, it doesn't require

any acknowledgement whatsoever; it will continue firing off VINTs, regardless of inservice to \$2002. The only way to stop VINTs is to clear \$2000.7.

- Because the PPU cannot make a read from PPU memory immediately upon request (via \$2007), there is an internal buffer, which acts as a 1stage data pipeline. As a read is requested, the contents of the read buffer are returned to the NES's CPU. After this, at the PPU's earliest convience (according to PPU read cycle timings), the PPU will fetch the requested data from the PPU memory, and throw it in the read buffer. Writes to PPU mem via \$2007 are pipelined as well, but it is unknown to me if the PPU uses this same buffer (this could be easily tested by writing somthing to \$2007, and seeing if the same value is returned immediately after reading).

+----+ |Frame rendering details|

+----+

The following describes the PPU's status during all 262 scanlines of a frame. Any scanlines where work is done (like image rendering), consists of the steps which will be described in the next section.

0..19: Starting at the instant the VINT flag is pulled down (when a NMI is generated), 20 scanlines make up the period of time on the PPU which I like to call the VINT period. During this time, the PPU makes no access to it's external memory (i.e. name / pattern tables, etc.).

20: After 20 scanlines worth of time go by (since the VINT flag was set), the PPU starts to render scanlines. This first scanline is a dummy one; although it will access it's external memory in the same sequence it would for drawing a valid scanline, no on-screen pixels are rendered during this time, making the fetched background data immaterial. Both horizontal *and* vertical scroll counters are updated (presumably) at cc offset 256 in this scanline. Other than that, the operation of this scanline is identical to any other. The primary reason this scanline exists is to start the object render pipeline, since it takes 256 cc's worth of time to

determine which objects are in range or not for any particular scanline.

21..260: after rendering 1 dummy scanline, the PPU starts to render the actual data to be displayed on the screen. This is done for 240 scanlines, of course.

261: after the very last rendered scanline finishes, the PPU does nothing for 1 scanline (i.e. the programmer gets screwed out of perfectly good VINT time). When this scanline finishes, the VINT flag is set, and the process of drawing lines starts all over again.

+-----+

|Scanline rendering details|

+----+

Naturally, the PPU will fetch data from name, attribute, and pattern tables during a scanline to produce an image on the screen. This section details the PPU's doings during this time.

As explained before, external PPU memory can be accessed every 2 cc's. With 341 cc's per scanline, this gives the PPU enough time to make 170 memory accesses per scanline (and it uses all of them!). After the 170th fetch, the PPU does nothing for 1 clock cycle. Remember that a single pixel is rendered every clock cycle.

Memory fetch phase 1 thru 128

- 1. Name table byte
- 2. Attribute table byte
- 3. Pattern table bitmap #0
- 4. Pattern table bitmap #1

This process is repeated 32 times (32 tiles in a scanline).

This is when the PPU retrieves the appropriate data from PPU memory

for rendering the playfield. The first playfield tile fetched here is actually the 3rd to be drawn on the screen (the playfield data for the first 2 tiles to be rendered on this scanline are fetched at the end of the scanline prior to this one).

All valid on-screen pixel data arrives at the PPU's video out pin during this time (256 clocks). For determining the precise delay between when a tile's bitmap fetch phase starts (the whole 4 memory fetches), and when the first pixel of that tile's bitmap data hits the video out pin, the formula is (16-n) clock cycles, where n is the fine horizontal scroll offset (0..7 pixels). This information is relivant for understanding the exact timing operation of the "object 0 collision" flag.

Note that the PPU fetches an attribute table byte for every 8 sequential horizontal pixels it draws. This essentially limits the PPU's color area (the area of pixels which are forced to use the same 3-color palette) to only 8 horizontally sequential pixels.

It is also during this time that the PPU evaluates the "Y coordinate" entries of all 64 objects in object attribute RAM (OAM), to see if the objects are within range (to be drawn on the screen) for the *next* scanline (this is why Y-coordinate entries in the OAM must be programmed to a value 1 less than the scanline the object is to appear on). Each evaluation (presumably) takes 4 clock cycles, for a total of 256 (which is why it's done during on-screen pixel rendering).

In-range object evaluation

An 8-bit comparator is used to calculate the 9-bit difference between the current scanline (minus 21), and each Y-coordinate (plus 1) of every object entry in the OAM. Objects are considered in range if the comparator produces a difference in the range of 0..7 (if \$2000.5 currently = 0), or 0..15 (if \$2000.5 currently = 1).

(Note that a 9-bit comparison result is generated. This means that setting object scanline coordinates for ranges -1..-15 are actually interpreted as

ranges 241..255. For this reason, objects with these ranges will never be considered to be part of any on-screen scanline range, and will not allow smooth object scrolling off the top of the screen.)

Tile index (8 bits), X-coordinate (8 bits), & attribute information (4 bits; vertical inversion is excluded) from the in-range OAM element, plus the associated 4-bit result of the range comparison accumulate in a part of the PPU called the "sprite temporary memory". Logical inversion is applied to the loaded 4-bit range comparison result, if the object's vertical inversion attribute bit is set.

Since object range evaluations occur sequentially through the OAM (starting from entry 0 to 63), the sprite temporary memory always fills in order from the highest priority in-range object, to lower ones. A 4-bit "in-range" counter is used to determine the number of found objects on the scanline (from 0 up to 8), and serves as an index pointer for placement of found object data into the 8-element sprite temporary memory. The counter is reset at the beginning of the object evaluation phase, and is post-incremented everytime an object is found in-range. This occurs until the counter equals 8, when found object data after this is discarded, and a flag (bit 5 of \$2002) is raised, indicating that it is going to be dropping objects for the next scanline.

An additional memory bit associated with the sprite temporary memory is used to indicate that the primary object (#0) was found to be in range. This will be used later on to detect primary object-to-playfield pixel collisions.

Playfield render pipeline details

As pattern table & palette select data is fetched, it is loaded into internal latches (the palette select data is selected from the fetched byte via a 2-bit 1-of-4 selector).

At the start of a new tile fetch phase (every 8 cc's), both latched pattern table bitmaps are loaded into the upper 8-bits of 2- 16-bit shift registers

(which both shift right every clock cycle). The palette select data is also transfered into another latch during this time (which feeds the serial inputs of 2 8-bit right shift registers shifted every clock). The pixel data is fed into these extra shift registers in order to implement fine horizontal scrolling, since the periods when the PPU fetch tile data is fixed.

A single bit from each shift register is selected, to form the valid 4-bit playfield pixel for the current clock cycle. The bit selection offset is based on the fine horizontal scroll value (this selects bit positions 0..7 for all 4 shift registers). The selected 4-bit pixel data will then be fed into the multiplexer (described later) to be mixed with object data.

Memory fetch phase 129 thru 160

- 1. Garbage name table byte
- 2. Garbage name table byte
- 3. Pattern table bitmap #0 for applicable object (for next scanline)
- 4. Pattern table bitmap #1 for applicable object (for next scanline)

This process is repeated 8 times.

This is the period of time when the PPU retrieves the appropriate pattern table data for the objects to be drawn on the *next* scanline. When less than 8 objects exist on the next scanline (as the in-range object evaluation counter indicates), dummy pattern table fetches take place for the remaining fetches. Internally, the fetched dummy-data is discarded, and replaced with completely transparent bitmap patterns).

Although the fetched name table data is thrown away, and the name table address is somewhat unpredictable, the address does seem to relate to the first name table tile to be fetched for the next scanline. This would seem to imply that PPU cc #256 is when the PPU's scroll/address counters have their horizontal scroll values automatically updated.

It should also be noted that because this fetch is required for objects on

the next scanline, it is neccessary for a garbage scanline to exist prior to the very first scanline to be actually rendered, so that object attribute RAM entries can be evaluated, and the appropriate bitmap data retrieved.

As far as the wasted fetch phases here, well, what can I say. Either Nintendo's engineers were VERY lazy, and didn't want to add the small amount of extra circuitry to the PPU so that 16 object fetches could take place per scanline, or Nintendo couldn't spot the extra memory required to implement 16 object scanlines. Thing is though- between the object attribute mem, sprite temporary & buffer mem, and palette mem, that's already 2406 bits of RAM; I don't think it would've killed them to just add the 408 bits it would've took for an extra 8 objects, which would've made games with horrible OAM cycling (Double Dragon 2 w/ 2 players) look half-decent (hell, with 16 object scanlines, games would hardly even need OAM cycling).

Details of object pattern fetch & render

Where the PPU fetches pattern table data for an individual object is conditioned on the contents of the sprite temporary memory element, and 2000.5. If 2000.5 = 0, the tile index data is used as usual, and 2000.3selects the pattern table to use. If 2000.5 = 1, the MSB of the range result value become the LSB of the indexed tile, and the LSB of the tile index value determines pattern table selection. The lower 3 bits of the range result value are always used as the fine vertical offset into the selected pattern.

Horizontal inversion (bit order reversing) is applied to fetched bitmaps, if indicated in the sprite temporary memory element.

The fetched pattern table data (which is 2 bytes), plus the associated 3 attribute bits (palette select & priority), and the x coordinate byte in sprite temporary memory are then loaded into a part of the PPU called the "sprite buffer memory" (the primary object present bit is also copied). This memory area again, is large enough to hold the contents for 8 sprites.

The composition of one sprite buffer element here is: 2 8-bit shift registers (the fetched pattern table data is loaded in here, where it will be serialized at the appropriate time), a 3-bit latch (which holds the color & priority data for an object), and an 8-bit down counter (this is where the x coordinate is loaded).

The counter is decremented every time the PPU renders a pixel (the first 256 cc's of a scanline; see "Memory fetch phase 1 thru 128" above). When the counter equals 0, the pattern table data in the shift registers will start to serialize (1 shift per clock). Before this time, or 8 clocks after, consider the outputs of the serializers for each stage to be 0 (transparency).

The streams of all 8 object serializers are prioritized, and ultimately only one stream (with palette select & priority information) is selected for output to the multiplexer (where object & playfield pixels are prioritized).

The data for the first sprite buffer entry (including the primary object present flag) has the first chance to enter the multiplexer, if it's output pixel is non-transparent (non-zero). Otherwise, priority is passed to the next serializer in the sprite buffer memory, and the test for non-transparency is made again (the primary object present status will always be passed to the multiplexer as false in this case). This is done until the last (8th) stage is reached, when the object data is passed through unconditionally. Keep in mind that this whole process occurs every clock cycle (hardware is used to determine priority instantly).

The multiplexer does 2 things: determines primary object collisions, and decides which pixel data to pass through to index the palette RAM- either the playfield's or the object's.

Primary object collisions occur when a non-transparent playfield pixel coincides with a non-transparent object pixel, while the primary object present status entering the multiplexer for the current clock cycle is true. This causes a flip-flop (\$2002.6) to be set, and remains set (presumably) some time after the VINT occurence (prehaps up until scanline 20?).

The decision for selecting the data to pass through to the palette index is made rather easily. The condition to use object (opposed to playfield) data is:

(OBJpri=foreground OR PFpixel=xparent) AND OBJpixel<>xparent

Since the PPU has 2 palettes; one for objects, and one for playfield, the appropriate palette will be selected depending on which pixel data is passed through.

After the palette look-up, the operation of events follows the aforementioned steps in the "video signal generation" section.

Memory fetch phase 161 thru 168

- 1. Name table byte
- 2. Attribute table byte
- 3. Pattern table bitmap #0 (for next scanline)
- 4. Pattern table bitmap #1 (for next scanline)

This process is repeated 2 times.

It is during this time that the PPU fetches the appliciable playfield data for the first and second tiles to be rendered on the screen for the *next* scanline. These fetches initialize the internal playfield pixel pipelines (2-16-bit shift registers) with valid bitmap data. The rest of tiles (3..32) are fetched at the beginning of the following scanline.

Memory fetch phase 169 thru 170

1. Name table byte

2. Name table byte

I'm unclear of the reason why this particular access to memory is made. The name table address that is accessed 2 times in a row here, is also the same nametable address that points to the 3rd tile to be rendered on the screen (or basically, the first name table address that will be accessed when the PPU is fetching playfield data on the next scanline).

After memory access 170

The PPU simply rests for 1 cycle here (or the equivelant of half a memory access cycle) before repeating the whole pixel/scanline rendering process.

+----+

|Extra cycle frames|

+----+

Scanline 20 is the only scanline that has variable length. On every odd frame, this scanline is only 340 cycles (the dead cycle at the end is removed). This is done to cause a shift in the NTSC colorburst phase.

You see, a 3.58 MHz signal, the NTSC colorburst, is required to be modulated into a luminance carrying signal in order for color to be generated on an NTSC monitor. Since the PPU's video out consists of basically square waves (as opposed to sine waves, which would be preferred), it takes an entire colorburst cycle (1/3.58 MHz) for an NTSC monitor to identify the color of a PPU pixel accurately.

But now you remember that the PPU renders pixels at 5.37 MHz- 1.5x the rate of the colorburst. This means that if a single pixel resides on a scanline with a color different to those surrounding it, the pixel will probably be misrepresented on the screen, sometimes appearing faintly.

Well, to somewhat fix this problem, they added this extra pixel into every odd frame (shifting the colorburst phase over a bit), and changing the way the monitor interprets isolated colored pixels each frame. This is why when you play games with detailed background graphics, the background seems to flicker a bit. Once you start scrolling the screen however, it seems as if some pixels become invisible; this is how stationary PPU images would look without this cycle removed from odd frames.

Certain scroll rates expose this NTSC PPU color caveat regardless of the toggling phase shift. Some of Zelda 2's dungeon backgrounds are a good place to see this effect.

+----+

|The MMC3's scanline counter|

+-----+

As most people know, the MMC3 bases it's scanline counter on PPU address line A13 (which is why IRQ's can be fired off manually by toggling A13 a bunch of times via \$2006). What's not common knowledge is the number of times A13 is expected to toggle in a scanline (although if you've been paying close attention to the doc here, you should already know ;)

A13 was probably used for the IRQ counter (as opposed to using the PPU's /READ line) because this address line already needed to be connected to the MMC for bankswitching purposes (so in other words, to reduce the MMC3's pin count by 1). They also probably used this method of counting (as opposed to a CPU cycle counter) since A13 cycles (0 -> 1) exactly 42 times per scanline, whereas the CPU count of cycles per scanline is not an exact integer (113.67). Having said that, I guess Nintendo wanted to provide an "easy-to-use" method of generating special image effects, without making programmers have to figure out how many clock cycles to program an IRQ counter with (a pretty lame excuse for not providing an IRQ counter with CPU clock cycle precision (which would have been more useful and versatile)).

Regardless of any values PPU registers are programmed with, A13 will operate in a predictable fashion during image rendering (and if you understand how PPU addressing works, you should understand that A13 is the *only* address line with fixed behaviour during image rendering). +-----+ |PPU pixel priority quirk| +-----+

Object data is prioritized between itself, then prioritized between the playfield. There are some odd side effects to this scheme of rendering, however. For instance, imagine a low priority object pixel with foreground priority, a high priority object pixel with background priority, and a playfield pixel all coinciding (all non-transparent).

Ideally, the playfield is considered to be the middle layer between background and foreground priority objects. This means that the playfield pixel should hide the background priority object pixel (regardless of object priority), and the foreground priority object should appear atop the PF pixel.

However, because of the way the PPU renders (as just described), OBJ priority is evaluated first, and therefore the background object pixel wins, which means that you'll only be seeing the PF pixel after this mess.

A good game to demonstrate this behaviour is Megaman 2. Go into airman's stage. First, jump into the energy bar, just to confirm that megaman's sprite is of a higher priority than the energy bar's. Now, get to the second half of the stage, where the clouds cover the energy bar. The energy bar will be ontop of the clouds, but megaman will be behind them. Now, look what happens when you jump into the energy bar here... you see the clouds where megaman underlaps the energy bar.

+-----+

|Graphical enhancements|

+-----+

Since an NES cartridge has access to the PPU bus, any number of oncart hardware schemes can be used to enhance the graphic capabilities of the NES. After all, the PPU's playfield pipeline is very simple: it fetches 272 playfield pixels per scanline (as 34*2 byte fetches, in real-time), and outputs 256 of them to the screen (with the 0..7 pixel offset determined by the fine X scroll register), along with object data combined with it.

Essentially, you can bypass the PPU's simple scrolling system, implement a custom one on your cart (fetching bitmap data in your own fashion), and feed the PPU bitmap data in your own order.

The possibilities of this are endless (like sporting multiple playfields, or even playfield rotation/scaling), but of course what it comes down to is the amount of cartridge hardware required.

Generally, playfield rotation/scaling can be done quite easily- it only requires a few sets of 16-bit registers and adders (the 16 bits are broken up into 8.8 fixed point values). But this kind of implementation is more suited for an integrated circuit, since this would require dozens of discrete logic chips.

Multiple playfields are another thing which could be easily done. The caveat here is that pixel pipelines (i.e., shift registers) and a multiplexer would have to be implemented on the cart (not to mention exclusive name table RAM) in order to process the playfield bitmaps from multiple sources. The access to the CHR-ROM/RAM would also have to increased- but as it stands, the CHR-ROM/RAM bandwidth is 1.34 MHz, a rather low frequency. With a memory device capable of a 10.74 MHz bandwith, you could have 8 playfields to work with. Generally, this would be very useful for displaying multiple huge objects on the screen- without ever having to worry about annoying flicker.

The only restriction to doing any of this is that:

- every 8 sequential horizontal pixels sent to the PPU must share the same palette select value. Because of this, hardware would have to be implemented to decide which palette select value to feed the PPU between 8 horizontally sequential pixels, if they do not all share the same palette select value. The on-screen results of this may not be too flattering sometimes, but this is a small price to pay to do some neat graphical tricks on the NES.

-only the playfield palette can be used. As usual, this pretty much limits your randomly accessable colors to about 12+1.

It's a damn shame that Nintendo never created a MMC which would enhance graphics on the NES in useful ways as mentioned above. The MMC5 was the only device that came close, and it's only selling features were the single-tile color area, and the vertical split screen mode (which I don't think any game ever used). Considering the amount of pins (100) the MMC5 had, and number of gates they put in it just for the EXRAM (which was 1K bytes), they could've put some really useful graphics hardware inside there instead.

Prehaps the infamous Color Dreams "Hellraiser" cart was the closest the NES ever came to seeing such sophisticated graphics. The cart was never released, but from what I've read, it was going to use some sort of frame buffer, and a Z80 CPU to do the graphical rendering. It had been rumored that the game had 3D graphics (or at least 2.5D) in it. If so (and the game was actually good), prehaps it would have raised a few eyebrows in the industry, and inspired Nintendo to develop a new MMC chip with similar capabilities, in order to keep the NES in it's profit margin for another few years (and allow it to compete somewhat with the more advanced systems of the time).

EOF

2008

This help file has been generated by the freeware version of HelpNDoc

NES Scrolling 1

Subject: [nesdev] the skinny on nes scrolling Date: Tue, 13 Apr 1999 16:42:00 -0600 From: loopy <zxcvzxcv@netzero.net> Reply-To: nesdev@onelist.com To: nesdev@onelist.com From: loopy <zxcvzxcv@netzero.net>

the current information on background scrolling is sufficient for most games;

however, there are a few that require a more complete understanding.

here are the related registers:

(v) vram address, a.k.a. 2006 which we all know and love. (16 bits)

(t) another temp vram address (16 bits)

(you can really call them 15 bits, the last isn't used)

(x) tile X offset (3 bits)

the ppu uses the vram address for both reading/writing to vram thru 2007,

and for fetching nametable data to draw the background. as it's drawing the

background, it updates the address to point to the nametable data currently

being drawn. bits 0-11 hold the nametable address (-\$2000). bits 12-14 are

the tile Y offset.

stuff that affects register contents:

(sorry for the shorthand logic but i think it's easier to see this way)

2000 write:

t:000011000000000=d:00000011 2005 first write: t:000000000011111=d:11111000

x=d:00000111

2005 second write:

t:0000001111100000=d:11111000

t:011100000000000=d:00000111

2006 first write:

t:001111110000000=d:00111111

2006 second write:

t:000000011111111=d:1111111

v=t

scanline start (if background and sprites are enabled):

```
v:0000010000011111=t:0000010000011111
```

frame start (line 0) (if background and sprites are enabled):

v=t

note! 2005 and 2006 share the toggle that selects between first/second writes. reading 2002 will clear it.

note! all of this info agrees with the tests i've run on a real nes. BUT if there's something you don't agree with, please let me know so i can verify

it.

2008

This help file has been generated by the freeware version of <u>HelpNDoc</u>

NES Scrolling 2

nes scrolling Date: Tue, 13 Apr 1999 17:48:54 -0600 From: loopy <zxcvzxcv@netzero.net> Reply-To: nesdev@onelist.com To: nesdev@onelist.com

From: loopy <zxcvzxcv@netzero.net>

(more notes on ppu logic)

you can think of bits 0,1,2,3,4 of the vram address as the "x scroll"(*8) that the ppu increments as it draws. as it wraps from 31 to 0, bit 10 is switched. you should see how this causes horizontal wrapping between name

tables (0,1) and (2,3).

you can think of bits 5,6,7,8,9 as the "y scroll"(*8). this functions slightly different from the X. it wraps to 0 and bit 11 is switched when it's incremented from _29_ instead of 31. there are some odd side effects

from this.. if you manually set the value above 29 (from either 2005 or 2006), the wrapping from 29 obviously won't happen, and attrib data will be

used as name table data. the "y scroll" still wraps to 0 from 31, but without switching bit 11. this explains why writing 240+ to 'Y' in 2005 appeared as a negative scroll value.

2008

This help file has been generated by the freeware version of <u>HelpNDoc</u>