NESCafe Guide

Creating Overrides for NESCafe

Author David de Niese
Release 0.701 (June 2006)
Email nescafe@davieboy.net
Web-site http://www.davieboy.net

This document covers how to create Overrides for NESCafe using the Profiler. Readers should have a good understanding of 6502 Assembler and have version 0.701 or higher of NESCafe.

Introduction

NESCafe is distributed under the GNU General Public License. A copy of this license agreement has been included with this distribution of NESCafe. If you have a question or wish to give me any feedback on the NESCafe Nintendo emulator then please do not hesitate to contact me via email (my address is above) as your comments and suggestions are always welcome. More information on NESCafe can also be found on the NESCafe website (address also shown above).

Requirements

Please ensure that you can meet the following requirements before following this guide:

- You must have a good understanding of 6502 Assembler
- You must have downloaded NESCafe 0.701 Standalone
- You must know how to use NESCafe for opening and playing games

Information on downloading and using NESCafe can be found in the NESCafe Release note, which is available from the website. The Code Profiler is only available in version 0.701 onwards of the Standalone version of NESCafe, so please check your version first.

Scope of Document

The purpose of this document is to introduce you to NESCafe Overrides and the NESCafe Code Profiler, and then to show you how to create your own NESCafe Override using the Profiler to assist you. I recommend that readers met the requirements outlined above before continuing, especially the requirement to understand 6502 assembly language.

Background: The NESCafe Override Engine

What are Overrides?

This feature is unique to NESCafe. Overrides are scriptable commands that tell NESCafe how to respond as a game runs. For example, Overrides can be used to implement very powerful game logic. In the example below for Mike Tyson's Punch-Out, the following Override tells NESCafe to not let Little Mac's (the boxer) energy go too high (limits it).

NESCafe Override Example for Mike Tyson's Punch-Out:

```
// Keep Low Energy for Little Mac

     on read 0x0393 if VALUEOF 0x0393 > 5 then VALUEOF 0x0391 = 5;
```

How do NESCafe Overrides Work?

Before continuing with the description of what this is doing, you will need to know a little about how computers operate. If you don't fully understand the following description, don't worry, you can download pre-made Override Scripts from the NESCafe website.

`on read 0x0393`	This tells NESCafe to wait for a memory read to address 0x0393, which is Little Mac's current energy level in the game. We were able to determine which memory address was used to store this by dissembling the game, but you don't need to worry if you don't know how this is done – that's why most of these NESCafe Override scripts will be pre-made!
`if VALUEOF 0x0393 > 5 then`	When the above memory read occurs, NESCafe is then told to check if the current VALUEOF the memory address is greater than 5 (which is a nominal about of energy for poor Little Mac)
`0x0391 = 5`	If the above check is satisfied then NESCafe is told to set the value of register 0x0391 to 5. This register is used to hold the target energy for Little Mac after he gets hit. The game then compares this value in 0x0391 against his current recorded energy (before the hit) in 0x0393 and if different slowly reduces the energy until 0x0393 is equal to 0x0391. Therefore, by setting 0x0391 to 5, we are ensuring that his energy can never go above 5 – in order words he is an easy knockout!

The above example could equally be achieved with a Game Genie code, but here are some additional examples that prove how powerful NESCafe Overrides can actually be. The following example will wait for Little Mac to win a fight and will then display a message on the NESCafe screen that he has won, as well as broadcast the number of the round that he won in, and the number of times Little Mac hit the mat during the fight! A similar approach to this is used on the NESCafe Online website at www.davieboy.net/play

NESCafe Override Example for Mike Tyson's Punch-Out (more advanced):

```
// Wait for Little Mac to Win and then Broadcast the Round Number

   on write 0x0170
            if VALUE != VALUEOF 0x0170 and VALUE != 0 Then
                   trigger win;

   on write 0x0171
            if VALUE != VALUEOF 0x0171 and VALUE != 0 Then
                   trigger win;


// Declare the Trigger to Broadcast the Round Number and Number of Times Down

   declare trigger
            win say "You won!" broadcast 0x0006 0x03D0;
```

NOTE: Please note that you should only use the above NESCafe Override with a Saved State file that starts at the beginning of a fight, otherwise the check that is made sometimes fires too early (during the initial menu system for Mike Tyson's Punch-Out).

```
on write 0x0170
  if VALUE != VALUEOF 0x0170
     and
     VALUE != 0 Then


     trigger win;
```

Just like with the previous example, this tells NESCafe to wait for a write instruction against memory address 0x0170, it checks if the VALUE that is being written doesn't equal what is currently stored in that address and if that the value is not equal to 0. If this is all true then it will call a Trigger called WIN.

Please don't worry about the memory addresses. You don't need to understand the significance of this instruction and the memory address that is being referenced. You should just know that when Mac wins a fight the address 0x0170 is changed to a non-zero value, which is what is being tested for here.

```
on write 0x0171
  if VALUE != VALUEOF 0x0171
     and
     VALUE != 0 Then

     trigger win;
```

This is the same instruction as above, but with a different memory address. This needs to be checked too because under certain circumstances within the game when Mac wins a match it will write to address 0x0171 instead of 0x0170 (see above). By added a clause for this address as well, we are saying we want to catch either of the write instructions.

```
declare trigger win
    say "You won!"
    broadcast 0x0006 0x03D0;
```

This introduces the topic of Triggers. Triggers are methods by which NESCafe is told to do something. In this example, when the Trigger called WIN is called (by either of the above instructions being satisfied) it will display the message “You Won!” on the screen and then broadcast the data in memory locations 0x0006 and 0x03D0 to the nescafesave.php script (using the nescafe/trigger content-type).

The memory addresses 0x0006 and 0x03D0 are used to store the Round Number and the number of times Little Mac has hit the ground during the match. This data is written to a file within the *trigger* directory on your web-server (from where your nescafesave.php is located). The first byte in the recorded trigger file will be the value from 0x0006 and the second byte will be the value from 0x03D0.

Hopefully now you will see the power that NESCafe Overrides provided. However, it will require enthusiastic 6502 developers to start getting into the games and working on interesting Overrides. As Overrides becomes more popular you will be able to download them from the NESCafe website (and hopefully other websites that decide to host them).

Here is one final example, which can be useful if you want to simply broadcast a value when the user hits the T key on their keyboard. For example, if you are using the below Override then whilst playing Mike Tyson’s Punch-Out can press the T key to broadcast your score. See if you can work out how this is achieved without a walkthrough:

NESCafe Override Example for Mike Tyson's Punch-Out (using key-triggered broadcasts):

```
   // Declare the Trigger to Broadcast the Score

      declare trigger
               win say "You won!" broadcast  0x03E8 0x03E9 0x03EA
                                             0x03EB 0x03EC 0x03ED;

   // Tell NESCafe to call Trigger WIN when user hits the T key

      on keypress trigger win;
```

How Do I Use a NESCafe Override?

If you want to use Overrides, you can specify them in both Applet and Standalone mode by placing an OVERRILEFILE tag in your NESCafe Settings file and then the actual Overrides in the file that the tag points to. For example, you could do the following:

Example Entry in the NESCafe Settings File:

```
OverRideFile=override.settings
```

Example override.settings File:

```
   // Little Mac can Keep Getting Up

        on read 0x03C1 VALUEOF 0x03C1 = 0;

   // Little Mac can not get any Star Uppercuts

        on read 0x0341 VALUEOF 0x0341 = 0;
```

How Do I Create Override Files?

There is currently no automatic tool available that can work out what these memory addresses are and create these Overrides files for you, just like there is no tool available that can automatically work out a Game Genie Code for a particular game you want to play. It is up to programmers with 6502 Assembler experience to disassemble these games and work out which memory locations provide what functionality. However, NESCafe does include a Code Profiler which will assist the decompiling of the game. This guide aims to walk you through an example Code Profiling in order to create an Override.

Background: The NESCafe Code Profiler

The NESCafe Code Profiler is available only in version 0.701 onwards of the Standalone version. Please ensure that you have the correct version before proceeding.

Background

The Code Profiler is intended for the use of assisting with the creation of NESCafe Overrides or disassembling Nintendo games. NESCafe has a Code Profiler, which is built into the Standard distribution (not available for the Applet version). Pressing F6 (when the DISABLEDEBUG tag is not set in the *nescafe.settings* file) will cause NESCafe to dump memory to disk and to start recording the disassembly of the ROM. A detailed description of how to use the Code Profiler is out of the scope of this user guide, and can be found in the Guide to Creating NESCafe Overrides document on the NESCafe website.

In summary, pressing F6 whilst in NESCafe will produce an excel spreadsheet called *profile.csv* (it can be found in the profiling sub-directory of the NESCafe application directory). The file has the following format, where the memory address is displayed against the value stored within that memory address at the time you pressed F6.

Address	P1
0x0000	0x00
0x0001	0x03
0x0002	0xD0
...	...
0x07FF	0xA5

P1 stands for Profile 1. If you now close down the spreadsheet, return to your game and press the F6 key again and then once more (so that you have pressed F6 a total of 3 times in all), the *profile.csv* file will have automatically been updated as shown below:

Address	P1	P2	P3	Reads	Writes
0x0000	0x00	0x02	0x02	25,310	20,757
0x0001	0x03	0x03	0x03	4,210	3,501
0x0002	0x01	0x01	0x01	469	4
...					
0x07FF	0xA5	0xA5	0xA5	0	0

This time, each of the three profiles that you requested are displayed side-by-side, with some additional details (such as the number of reads and writes that the game performed against that memory location between the profiles that you took. This can be useful in finding particular registers, for example, if you know that your character in the

game was hit 4 times across the profiles then you could look for registers (memory addresses) that received 4 write operations as a way of identifying potential candidates because the game would have had to re-write your health value after each of the 4 hits.

You should also notice a second file call *decompiled.txt*. This contains a 6502 disassembly of all code executed between profile snapshots. This will be useful only if you understand 6502 assembler, and therefore it's outside of the scope of this document.

Next close down the Spreadsheet again and return to your game. Press the F6 key a further 2 times (so that you have pressed F6 a total of 5 times in all). The *profile.csv* file will have been automatically updated again, as shown below and will include some additional profiling columns, which help you identify trends in the data being written to particular registers (and therefore help you determine what type of data is stored there).

Address	P1...	Writes	Increment	Decrement	Constant	Unique	All Zero	Flag	Unique Values
0x0000	0x00	20,757	0	0	0	0	0	0	2
0x0001	0x03	3,501	0	0	1	0	0	0	1
0x0002	0x01	4	0	0	1	0	0	0	1
...									
0x07FF	0xA5	0	0	0	1	0	0	0	1

The table above shows the trend columns that have been added (in green). When NESCafe has recorded enough profiles, it will run trend analysis on each memory address to help you determine what it is potentially being used for. Each of the metrics above has been described in detail below, together with what it could potentially be used for.

Trend Metric	Description	Usage
READS	The total number of read operations that the game has made against this memory address, between when you took the first and last profiles.	This is useful in detecting heavily access registers, which could be internal counters or health and status bars, which would be being read on every screen re-draw (60 times/sec).
WRITES	The total number of write operations that the game has made against this memory address, between when you took the first and last profiles.	This is useful in detecting health bars or timers. If you know that your health was adjusted in the game 4 times between the profiles (as a result of hits you took), then you may be able to identify the health bar by looking for a value of 4 here in this field.
INC	Whether the profile of this memory address showed an incrementing trend (the first profile had a smaller value than the last profile value, and all other values were incrementing within the range of the first and last).	This is useful in detecting an increasing value, for example, it could be used to detect registers that may store the level of the game (which you would expect to increment as you progress in the game).
DEC	Whether the profile of this memory address showed a decreasing trend (the first profile had a greater value than the last profile value, and all other values were consistently decreasing within the range of the first and last).	This is useful in detecting a decreasing value, for example, your health register or life-count if you know that you lost health or lives during this profiling exercise.
CONSTANT	Whether the profile of this memory address showed a constant trend (all values profiled here were the same).	This is useful in detecting a constant value throughout the profiling exercise. For example, if you ensured that you kept the same count of lives throughout the game.
UNIQUE	Whether all values profiled were unique with respect to each other during the exercise	This is useful in detecting changing values. For example, if you ensured that each of the profiles taken were when you had different numbers of lives or different amounts of health.
ALLZERO	Whether all values profiled were consistently zero during the profiling exercise.	This is useful in detecting consistently zero values.

Trend Metric	Description	Usage
FLAG	Whether all values profiled were consistently zero or one during the profiling exercise.	This is useful in detecting flag registers, such as “has sword” or “doesn’t have sword”.
UNIQUEVALS	The total number of unique values that were profiled.	This is useful for finding values that you expect to be unique on each profile. For example, if you took 50 profiles, but know that the register you are looking for can only be one of two values.

Example: Worked Example

This section of the document will walk you through how to use the Code Profiler with Legend of Zelda to produce an Override to prevent Link (the character you play within the game) from picking up a sword. You could use this Override to see if any of the visitors on your website can complete the game with this extra limitation in place.

General Approach

We want to identify the memory location that stores the status of whether or not Link has picked up the sword. We will run the game and profile the memory usage. After taking a couple of snapshots we will pick up the sword and then continue profiling for a few more snapshots. We will then analyse the results and see if NESCafe can determine which memory location stores the status of whether Link has the sword or not.

In summary:

1. Run the Profiling
2. Pick up the Sword
3. Analyse the Profiling Statistics
4. Analyse the Code
5. Write the NESCafe Override

Step 1: Run the Profiling

Start the Standalone version of NESCafe and run the Legend of Zelda game. When you first start the game, wander around the map without picking up the sword from the cave that is immediately above the start position. As you wander around, go through different areas of the map, perhaps lose and gain some health, and remember to take around 10 different profile snapshots by pressing F6.

The aim of this step is to get all the memory addresses moving, which will happen the more you wander around. Enemies will use their memory addresses, your health and status registers will change. One of the only registers that won't change is the one that says whether you have picked up the sword yet or not.

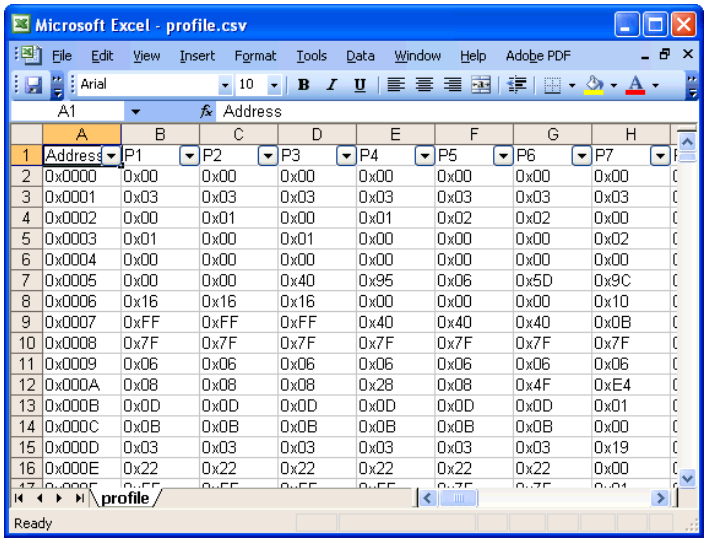
Step 2: Pick up the Sword

Next go back to the cave at the start point and pick up the sword. Continue to walk around and take around 10 more profile snapshots as you go. Most importantly, make sure that you use the Sword several times (to ensure that the code for checking for the sword executes).

Try and go back through the same rooms that you went through before, maybe losing and gaining some health at the same time. Finally, close Zelda and close down NESCafe.

Step 3: Analyse the Profiling Statistics

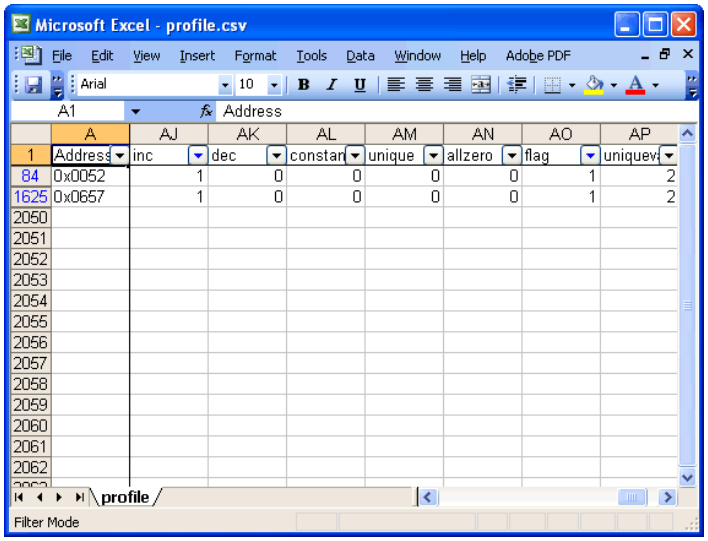
Look in the profiling subdirectory of the NESCafe application directory and open up the *profile.csv* file with Excel (or a Spreadsheet package of your choosing). Enable the Auto Filter option from the Data menu in Excel. The screen should look something like below:



	A	B	C	D	E	F	G	H
1	Address	P1	P2	P3	P4	P5	P6	P7
2	0x0000	0x00	0x00	0x00	0x00	0x00	0x00	0x00
3	0x0001	0x03	0x03	0x03	0x03	0x03	0x03	0x03
4	0x0002	0x00	0x01	0x00	0x01	0x02	0x02	0x00
5	0x0003	0x01	0x00	0x01	0x00	0x00	0x00	0x02
6	0x0004	0x00	0x00	0x00	0x00	0x00	0x00	0x00
7	0x0005	0x00	0x00	0x40	0x95	0x06	0x5D	0x9C
8	0x0006	0x16	0x16	0x16	0x00	0x00	0x00	0x10
9	0x0007	0xFF	0xFF	0xFF	0x40	0x40	0x40	0x0B
10	0x0008	0x7F	0x7F	0x7F	0x7F	0x7F	0x7F	0x7F
11	0x0009	0x06	0x06	0x06	0x06	0x06	0x06	0x06
12	0x000A	0x08	0x08	0x08	0x28	0x08	0x4F	0xE4
13	0x000B	0x0D	0x0D	0x0D	0x0D	0x0D	0x0D	0x01
14	0x000C	0x0B	0x0B	0x0B	0x0B	0x0B	0x0B	0x00
15	0x000D	0x03	0x03	0x03	0x03	0x03	0x03	0x19
16	0x000E	0x22	0x22	0x22	0x22	0x22	0x22	0x00

Each of the profiles against each of the memory locations is shown. Since we are looking for a status register, based upon ones knowledge of how these games are typically written, we will assume that the status values that are stored to memory to indicate whether Link has the sword or not will either be 0 or 1 (where 0 represents the Sword is not picked up and 1 represents that it is picked up). This is only a guess at this stage.

Go to the far right and then filter the Flag column on the value 1 (this column is set to 1 if the values written to the corresponding memory location were only 0 or 1). Next filter the Increment column on the value 1 on the assumption that Link went from not having a Sword (0) to having a sword (1). This will leave you with only two memory locations.



At this stage it is also worth doing some additional due-diligence and confirming that the value in the registers changed from 0 to 1 at the Profile where you actually picked up the sword. In my example I picked up the sword after Profile 16, so I would want to confirm that the value changes from 0 to 1 immediately after Profile 16, which it did.

We have the following candidates as potential addresses for holding the sword status:

0x0052
0x0657

Step 4: Analyse the Code

The next step is to take a look at the disassembled code. A file called *decompiled.txt* can be found in the same directory as the *profile.csv* file and contains a complete listed of all the 6502 instructions that were executed between the profiles. Once you have opened this file in a text editor, the next stage is to search for references to these addresses.

You can search for references to 0x0657 using the search string $0657, but remember that the 6502 processor has a zero-page address mode, so you need to search for 0x0052 using both $0052 and $52 (you also need to check for other addressing modes that may have been used, such as indexed and indirect based upon the X or Y register).

From analysing the code you should be able to tell that the register that stores the status of whether Link has the sword or not is actually 0x0657 (a description of how to analyse 6502 assembler is outside of the scope of this document, this knowledge is assumed). Therefore we are now ready to construct a NESCafe Override to prevent this register from ever being set to 1 (in order to prevent Link from picking up the sword).

Step 5: Write the NESCafe Override

The following Override will activate whenever the game attempts to read from the 0x0657 memory address. The result is that NESCafe will keep the value stored in that address register at 0 and therefore never let Link pick up the sword. You can test the Override with NESCafe in the usual way (as documented in the NESCafe Release Note).

NESCafe Override Example for Legend of Zelda to prevent Sword Pickup:

```
// Zelda Prevent Sword Pickup
     on read 0x0657 valueof 0x0657 = 0;
```

Please also consider sending me any Overrides that you produce (via the Contact Me section on the NESCafe website) and I will list them on my website, with a reference to yourself. See how creative you can get with your Nintendo games.