

SoniFight User Guide



Alastair Lansley / Federation University Australia

a.lansley@federation.edu.au

Contents

Introduction.....	3
Demonstration / Quick Start	4
Download, Installation and System Requirements	5
File Structure	6
User Interface Elements – Main Tab	7
User Interface Elements – Edit Tab	8
Edit Tab – Tree View	8
Edit Tab – Details Panel	9
Creating a New Config	10
Creating Watches	10
Creating Triggers.....	12
Finding and Using Watches and Triggers.....	16
Finding Pointer Trails in Cheat Engine via Pointer Scans.....	17
Part 1 – Finding an Initial Value	17
Part 2 – Scanning for Pointers to a Value	18
The Clock Watch and Triggers	21
Normal Triggers	21
Continuous Triggers.....	22
Modifier Triggers	22

Pointer Trail Tester	24
Frequently Asked Questions.....	25
Help! My Trigger Doesn't Make a Sound!	25
Does SoniFight support game X? / Could you write a config for game X?	25
Does SoniFight use a lot of CPU or RAM? / Will it have a detrimental effect on game performance?	25
Is SoniFight cheating? If I use it online will it get me banned from services like Steam?	26
I've made a config! Can you ship it with the next release?	26
Both my friend and I are partially or non-sighted, can we play against each other properly?	26
I want to add additional triggers, how easy it is to do?	26
I only want some of the triggers to play / random non-sensical menu triggers sometimes play, can I disable them?	27
How are configs shipped?.....	27
What platforms does SoniFight run on?	27
Can I have access to and modify the SoniFight source code? Can I sell it?	27
I have an issue with the software or a question that's not covered here.....	28

Introduction

SoniFight is utility software to provide additional sonification cues to video games, especially fighting games, for visually impaired players.

SoniFight is written in C# and licensed under the MIT software license. The source code is freely available for use and modification at: <https://github.com/FedUni/SoniFight>. Please see LICENSE.txt for further details, including separate licensing details for the embedded irrKlang audio library.

To run SoniFight either download a precompiled binary release or build the Visual Studio 2015 solution yourself, then launch the SoniFight executable, choose a game config for the game you want to play, click the "Run Selected Config" button and then launch the game that the selecting game config targets.

SoniFight presently ships with game configs to add sonification to Ultra Street Fighter IV Arcade Edition and Mortal Kombat 9 (aka Mortal Kombat Komplete Edition) and a basic config for Doom (2016).

Once running, SoniFight will provide additional sonification cues such as clock, health and meter-bar status updates for both players. In the Street Fighter game config, there are also triggers for a large number of menu options so that there is less need to memorise sequences of menu selections. In the Doom game config there are low health and ammunition alerts.

SoniFight also provides a user interface where you can create your own game configs for games of your choice, although the process to find pointer trails requires additional free software and can be a little bit tricky and time consuming.

To learn more about creating your own game configs as well as how the software operates through 'watches' and 'triggers' please see relevant sections of this user documentation.

Demonstration / Quick Start

If you want to quickly get an idea of what the SoniFight software can do then a demonstration video is available at the following location:

TO DO - LINK TO DEMONSTRATION VIDEO HERE

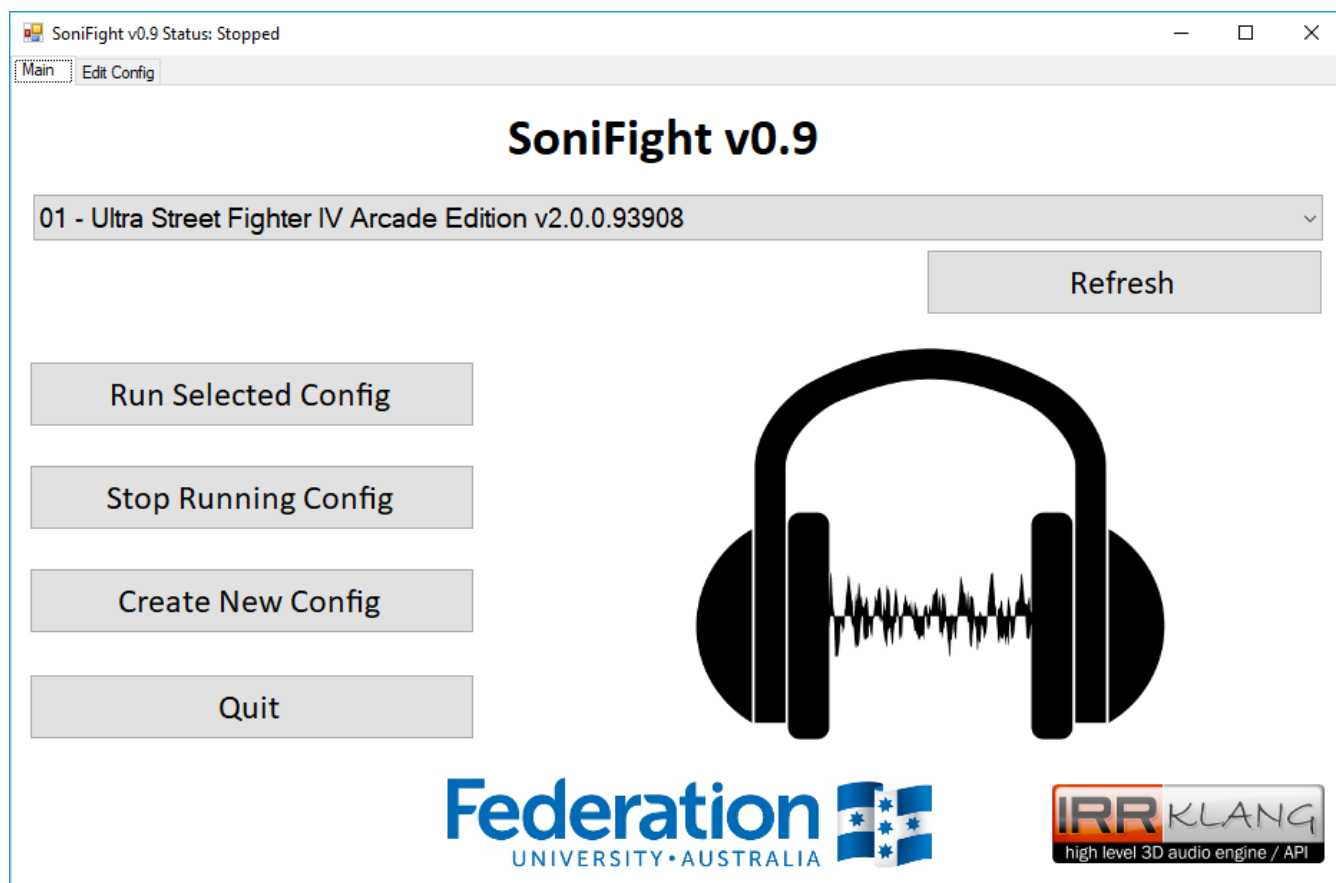


Figure 1 – The main tab of the SoniFight user interface.

To run the SoniFight and PointerTrailTester applications, the .NET framework version 4.7 or later must be installed on your computer. If you do not have this installed it is freely available from Microsoft at the following URL:

<http://go.microsoft.com/fwlink/?LinkId=825299>

Download, Installation and System Requirements

If you just want to use the software then you can download a precompiled binary release from:

<https://github.com/FedUni/SoniFight/releases>

Once downloaded, extract the zip file wherever you'd like and run the SoniFight executable within.

With SoniFight running, select a game config from the dropdown menu, click the **[Run Selected Config]** button and then launch the game related to the game config you've chosen to provide sonification.

If you want to build the software from source then you can either download a zip of the latest files from:

<https://github.com/FedUni/SoniFight>

Or, if you have git source control tools installed, such as those from <https://git-for-windows.github.io/>, then you can type the following into the command prompt to clone the current master branch of the repository:

```
git clone https://github.com/FedUni/SoniFight
```

Once downloaded you can open the SoniFight solution in Visual Studio 2015 to build it for yourself. Unless you plan on debugging the app you should build your version using the **Release** configuration. If you do not have Visual Studio 2015, then the Community Edition may be freely downloaded from Microsoft at:

<https://www.visualstudio.com/downloads/>

To successfully run the pre-compiled software you will need the .NET framework version 4.7 runtime installed on your computer, which can be freely obtained from:

<http://go.microsoft.com/fwlink/?LinkId=825299>

However, if you plan to build the software yourself, you will need the .NET framework version 4.7 **developer pack** installed on your computer, which can be freely obtained from:

<http://go.microsoft.com/fwlink/?LinkId=825319>

In terms of operating system requirements, SoniFight requires Windows 7 SP1 or higher to work (as that's the lowest available operating system version for the .NET framework v4.7).

In terms of memory usage, the app uses approximately 30MB to edit a complex config with dozens of watches and hundreds of triggers such as the included Street Fighter IV game config. When the app is started to provide sonification, samples are loaded into memory for instant playback – so the memory usage will vary depending on the number and size of the samples used. In the complex Street Fighter IV config provided memory, usage is approximately 70MB.

File Structure

Releases are provided as a zip archive containing pre-compiled versions of the SoniFight executable including provided game configs and the pointer trail tester utility in the following structure:

```
C:\SoniFight
|
|  build.txt
|  ikpFlac.dll
|  ikpMP3.dll
|  irrKlang.NET4.dll
|  msvcr100.dll
|  PointerTrailTester.exe
|  SoniFight.exe
|  SoniFight-User-Guide.pdf
|
└─ Configs
    └─ 01 - Ultra Street Fighter IV Arcade Edition v2.0.0.93908
        └─ 02 - Mortal Kombat Komplete Kollektion (IN-GAME ONLY)
```

Figure 2 - The SoniFight release directory structure, as of SoniFight v0.9.

The **build.txt** file identifies the overall version of the SoniFight software. This version number may not necessarily match the individual versions of the SoniFight and pointer trail tester components which may change independently, however any increment of either component's version number will result in an increment of this overall build version number.

The **Configs** folder contains game configs and the related audio samples for various games, where each config has its own subdirectory, and the samples within that directory are used by that game config.

It is advisable to keep all the game config file (**config.xml**) and all the samples it uses in the same directory so that game configs can be transferred or shared without any additional dependencies. Although this may mean that multiple game configs contain some of the same audio samples, the sample file sizes themselves are typically very small, so it's a small price to pay to keep the game config independent of all others.

User Interface Elements – Main Tab



Figure 3 - SoniFight Main tab user interface elements

The numbered elements in the above figure are as follows:

1. **Title Bar** – Displays the status of whether SoniFight is stopped or running a given game config.
2. **Main tab** – The tab which provides functionality to select a game config and start or stop it.
3. **Edit Config tab** – The tab which provides functionality to modify a game config.
4. **Config dropdown menu** – The dropdown menu which selects which config to use when the **Run Selected Config** button is clicked.
5. **Refresh button** – If you copy a new folder into the Configs directory you can click this button to refresh the config dropdown menu so that it contains the new folder as an available option instead of needing to restart the SoniFight software for it to be picked up.
6. **Run Selected Config button** – Runs the SoniFight software to enable additional sonification as specified in the selected game config. You do not need to launch the game process before clicking this as it will happily wait while attempting to connect to the specified game config process without issue.
7. **Stop Running Config button** – Stops the currently running game config and unloads all samples.
8. **Create New Config** – Creates a new, blank game config and switches to the **Edit tab**. A folder with the name of the config will be created, within which the **config.xml** file will exist, when the config is saved.
9. **Quit button** – Exits the SoniFight software.

User Interface Elements – Edit Tab

The **edit tab** is used to modify game configs and is broken up into two main sections – the Tree View and buttons on the left third of the screen and the details panel on the right two-thirds of the screen.

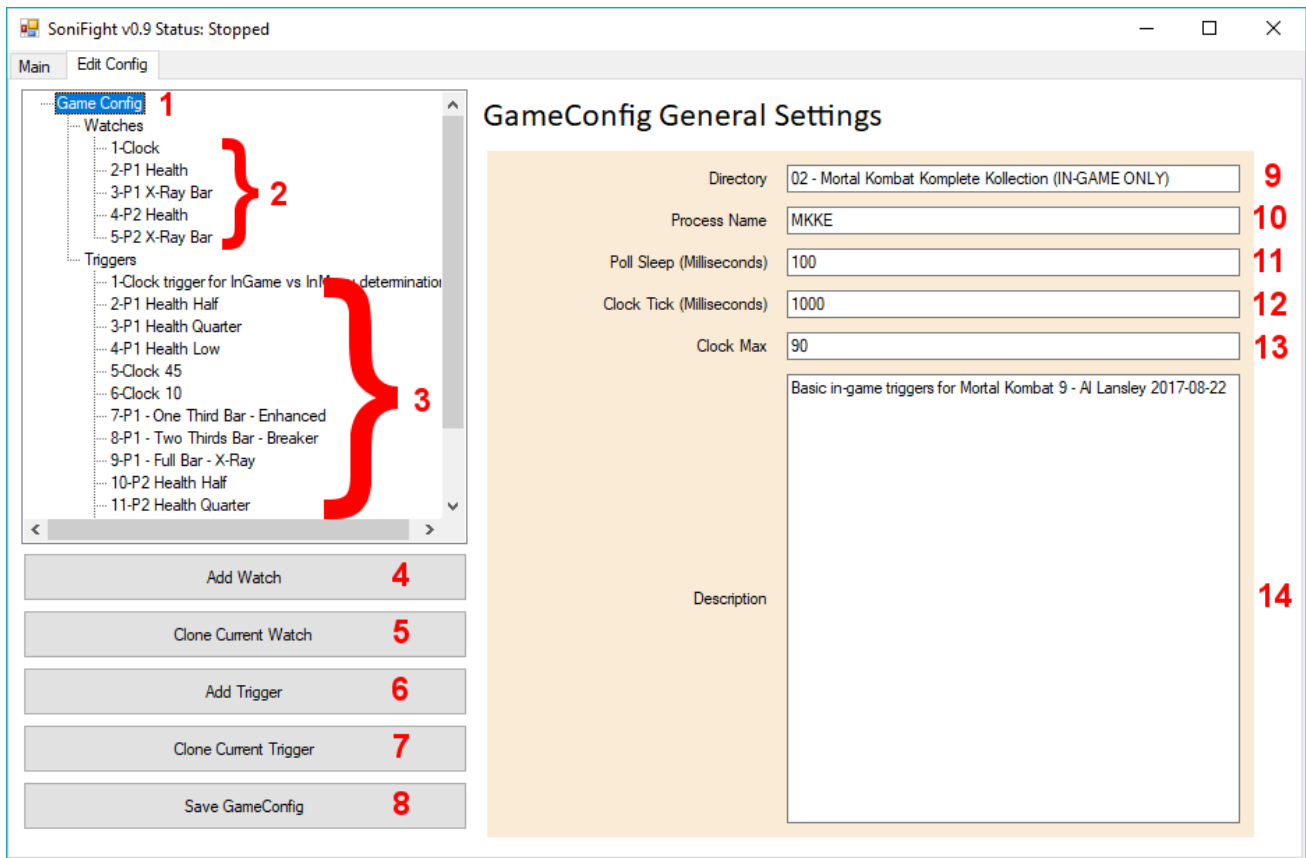


Figure 4 - The Edit Config tab with the main Game Config tree node selected.

Edit Tab – Tree View

The tree view is where you can choose to modify the game config settings including its main options such as the process to use, poll sleep delay and its watches and its triggers. You can, of course, modify the config.xml file for a game config manually using a text editor if you so choose, but great caution is advised as a single bad character in the wrong place will cause the config to become ‘corrupted’ and unable to be loaded. Typically, leave the config modification to to the SoniFight user interface unless you’re confident you know precisely what you’re doing.

Clicking on the main **Watches** or **Triggers** nodes displays a brief description what they are.

The numbered elements in the above figure are as follows:

1. **Game Config** – The main config settings node holds details of the directory the config lives in, the process name to query the base address of, the poll sleep delay, clock tick delay and the gameconfig’s overall description.
2. **Watches** – These tree nodes specify the pointer trail to a memory location and the type of data to read from that location. In essence, they ‘watch’ a memory location and read a value from it every poll sleep milliseconds.

3. **Triggers** – These tree nodes specify which watch to read data from, and the conditions under which they should ‘trigger’ a sonification event.
4. **Add Watch** – Adds a new, blank watch with a unique ID.
5. **Clone Current Watch** – Creates a new watch with a unique ID value which is populated with the details of the currently selected watch. The text “-CLONE” is appended to the name of the new watch to easily distinguish it from the original.
6. **Add Trigger** – Adds a new, blank trigger with a unique ID.
7. **Clone Current Trigger** – Creates a new trigger with a unique ID value which is populated with the details of the currently selected trigger. The text “-CLONE” is appended to the name of the new trigger to easily distinguish it from the original.
8. **Save GameConfig** – Saves the current configuration settings to the *config.xml* file within the directory specified for the game config.

Edit Tab – Details Panel

The details panel is where you can edit and view configuration settings of a game config, its watches and its triggers. The available options will vary depending on which of these items is selected in the left-side Tree View. For the main Game Config tree node, the available options are:

9. **Directory** – The directory to save this game config to. If you change this value you will need to manually copy any required audio samples into the new directory for the config to work.
10. **Process Name** – The name of the process to connect to. You can find the name of a running process by browsing the list of running processes in Windows Task Manager (shortcut: **Ctrl+Shift+Esc**). The process name should not contain the .EXE suffix of the executable.
11. **Poll Sleep (Milliseconds)** – The length of time in milliseconds (i.e. one thousandths of a second) to wait before polling all watches and triggers e.g. if this value is 100 milliseconds then SoniFight will poll for changes 10 times per second. The value must be between 1 and 200 milliseconds, where smaller values will poll more often and use higher CPU. Typically, a value of 100 milliseconds is sufficient to provide a sonification cue in a tenth of a second while using very little CPU resources.
12. **Clock Tick (Milliseconds)** – The estimated length of time in milliseconds that one ‘tick’ of the game clock (aka round timer) takes. This is used to help keep track of the game state so SoniFight knows we’re either in a live game or in the menus by enforcing that at least two ‘ticks’ have passed before allowing any triggers marked as “In-Menu” to provide sonification events.
13. **Clock Max** – The maximum value for the clock in a round. This value is used to prevent SoniFight from thinking we’re back ‘*In-Game*’ when the clock gets reset between rounds or matches.
14. **Description** – This is simply an optional multi-line text box where you can write some details regarding the game config should you wish.

Creating a New Config

To create a new, blank game config you can click the **[Create New Config]** button from the main tab.

To save the current game config in a different directory, change the directory name of an existing config and then click the **[Save GameConfig]** button. Please note that you will need to manually copy all the audio samples from the original to the new config directory for them to be used – that is, samples are not shared between directories and all samples used by a game config should exist in the same directory as the game config's *config.xml* file itself.

Creating Watches

To create a new, blank watch click the **[Add New Watch]** button from the edit tab.

To clone an existing watch, select a watch from the left-hand watch sub-tree and then click the **[Clone Current Watch]** button. After which a new watch will exist with the details copied from the original watch and the word "CLONE" appended to the name.

A **watch** itself is simply a few pieces of data that help to locate a memory address and the type of data that should be read from that memory address. However, the watch address isn't a single value – as due to technologies such as Address Space Layout Randomization (ASLR) and the current state of memory usage on the host machine, a single stored memory address would not be sufficient to consistently reproduce the location of a given game value such as the clock or a player's health across reboots of the host machine.

Instead, a watch must use a kind of **relative address** in the form of a **pointer trail**. This is a series of 'jumps', starting at the beginning of where the game process is loaded into memory that will always lead us to the memory address of a value of interest such as the clock etc.

A watch has the following user interface elements:

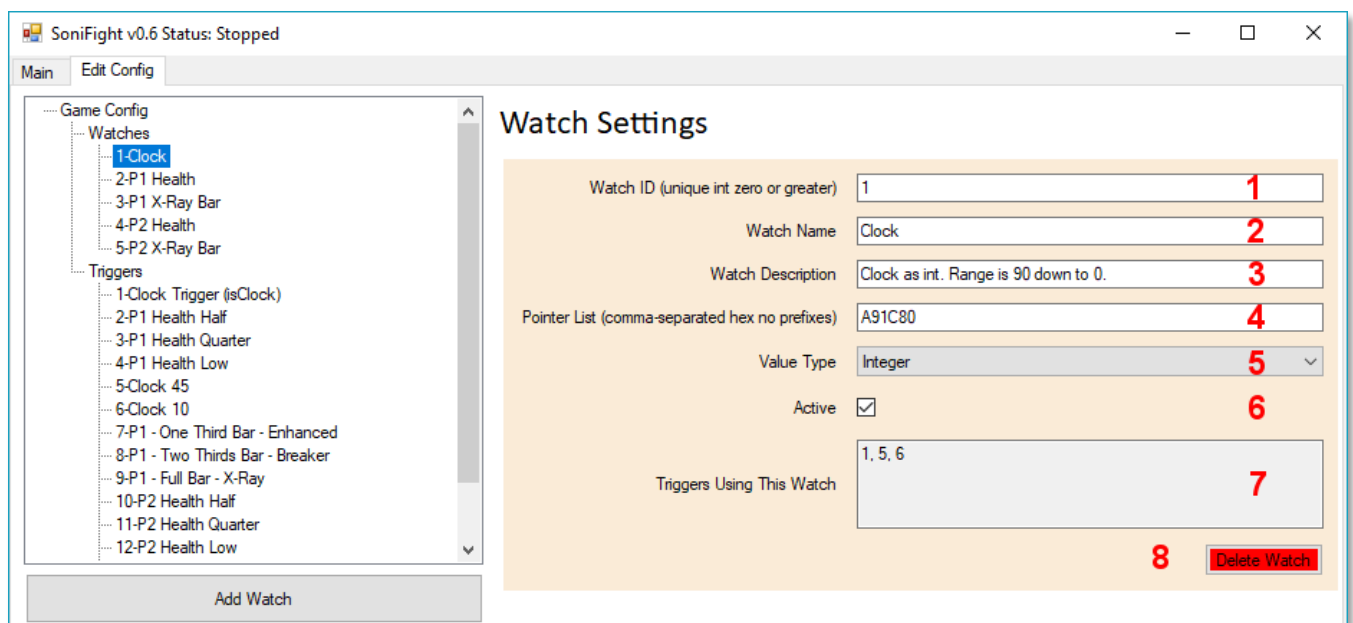


Figure 5 – Watch configuration details.

The watch details in the above figure are as follows:

1. **Watch ID** – This is a positive integer value that uniquely identifies this watch for use by triggers. Each new watch will be assigned a new unique value, or you may assign your own. Once set, it's not recommended to change the watch ID because any triggers which use a watch will not have the associated watch ID automatically updated if the watch ID is changed.
2. **Watch Name** – A name for the watch, optional but useful.
3. **Watch Description** – A description for the watch, again optional but useful.
4. **Pointer List** – This is a series of one or more comma-separated hexadecimal values used to offset from the game process' base address to find a useful value such as the clock or player health. Do not include any 0x prefixes or such to indicate that offsets are in hexadecimal format.

For example, the following pointer trail will point at the 'clock' (i.e. round timer) in the game Street Fighter IV:

6A7DD8, 18, 90, 110, 38

Further details on how pointer lists work internally can be found in the ***Finding and Using Watches and Triggers*** section of this document below.

5. **Value Type** – The above ***Pointer List*** provides enough information to locate a given memory address of interest, but once there we need to know what ***type*** of data we should read from that memory address. The value type dropdown provides the following options for data types to read from the address:
 - **Integer** (4 bytes),
 - **Short** (2 bytes),
 - **Long** (8 bytes),
 - **Float** (4 bytes),
 - **Double** (8 bytes),
 - **Boolean** (1 byte),
 - **String (UTF-8)** – 1 byte per character, read up until the null terminator or 33 chars and trimmed to remove whitespace,
 - **String (UTF-16)** – 2 bytes per character, read up until the null terminator or 33 chars and trimmed to remove whitespace.
6. **Active** - This checkbox is used to toggle whether this watch is in use or not. The default is checked (active). If the checkbox is unchecked then this watch will not be polled and as such cannot activate any triggers that might depend upon it. Sometimes you may wish to deactivate a given watch in a config to temporarily disable it without losing the watch data when should you wish to reactive it – this is the mechanism to do so.
7. **Triggers Using This Watch** – This read-only textbox simply displays the ID values of triggers which depend upon this watch so that you can easily tell if it's important or not. If no triggers depend on this watch then the value displayed will be **None**. Note that the active status of any given watch or trigger does not affect whether a given trigger ID may be displayed here – if the trigger depends on the watch, then it will show up regardless of whether either is marked as active or not.

- **Delete Watch Button** – Deletes the currently selected watch. There is no undo option or confirmation dialogue, but the change is not permanently applied until the **[Save GameConfig]** button is clicked, so if you clicked the button by accident and wanted the watch back then you could switch back to the **Main** tab and then back to the **Edit Config** tab to force a reload of the game config from its last saved state.

Creating Triggers

A **trigger** is a condition that we check against to determine whether we should play a sample (i.e. provide a sonification event) or not. Each trigger has its own unique ID, but will also depend on at least one watch (as specified by the **Watch 1 ID** field) along with some **comparison criteria** such as equals, more than, less than etc. and a **value** that must match that criteria for the sonification event to occur.

To create a new, blank trigger click the **[Add New Trigger]** button from the **Edit Config** tab.

To clone an existing trigger, select a trigger from the left-hand trigger sub-tree and then click the **[Clone Current Trigger]** button. After which a new trigger will exist with the details copied from the original trigger and the word “CLONE” appended to the trigger name.

A trigger has the following user interface elements:

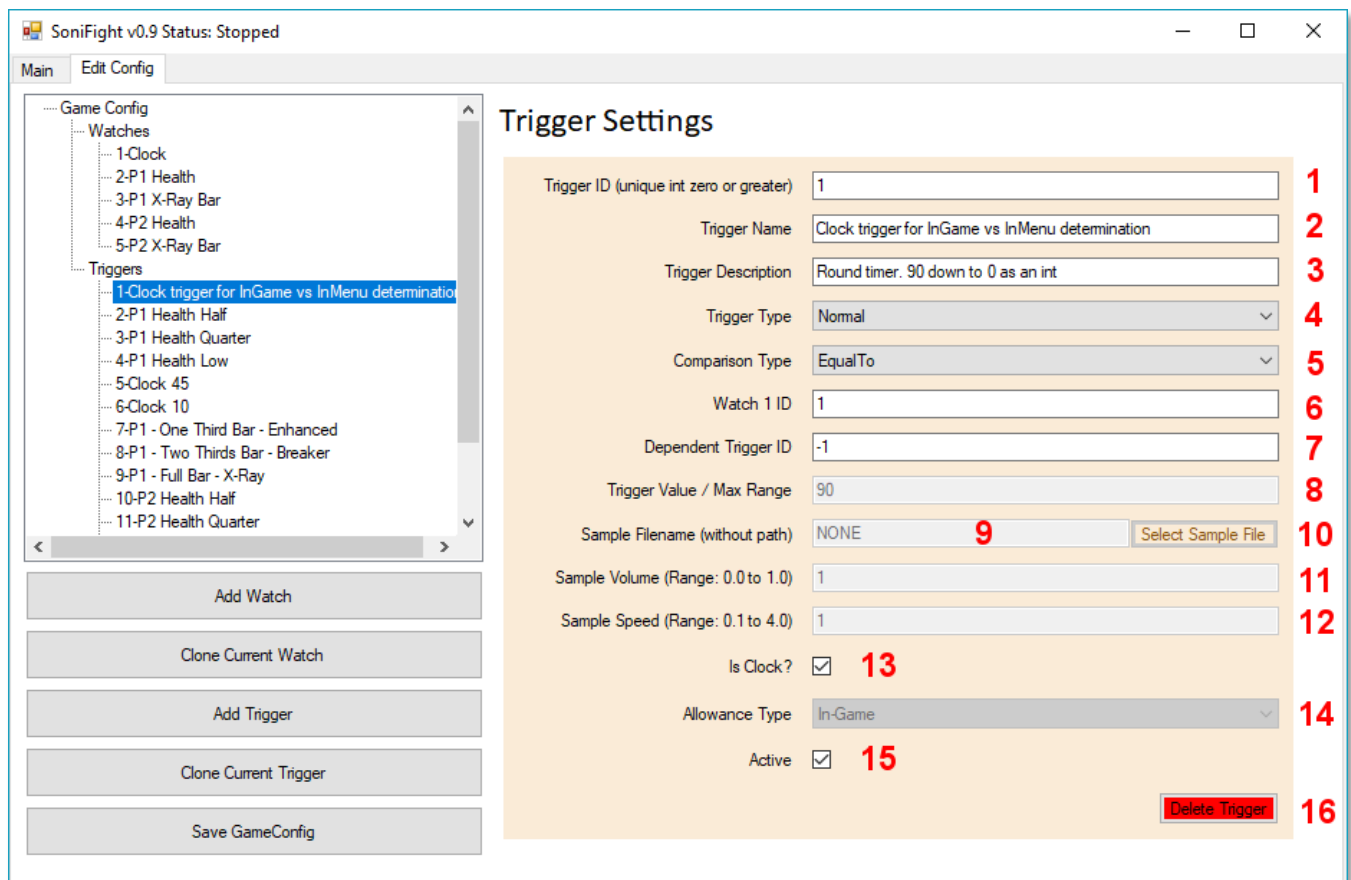


Figure 6 – An example Trigger and its configuration details

The trigger details in the above figure are as follows:

1. **Trigger ID** – This is a positive integer value that uniquely identifies this trigger. Each new trigger will be assigned a new unique value, or you may assign your own unique value. As triggers depend on watches but not vice versa you are free to change the trigger ID as you please, so long as the trigger ID remains a unique integer greater than or equal to zero.
2. **Trigger Name** – A name for the trigger, optional but useful.
3. **Trigger Description** – A description for the trigger, yet again optional but useful.
4. **Comparison Type** – The comparison type is how we compare the trigger value (described below) against its previous value. The comparison must pass through this threshold to trigger a sonification event. Available comparison types are as follows:
 - **Equal To** (threshold: *Not Equal To*),
 - **Less Than** (threshold: *Greater Than Or Equal To*),
 - **Less Than Or Equal To** (threshold: *Greater Than*),
 - **Greater Than** (threshold: *Less Than Or Equal To*),
 - **Greater Than Or Equal To** (threshold: *Less Than*),
 - **Not Equal To** (threshold: *Equal To*),
 - **Changed** (threshold: *Not Equal To Previous Value*), or
 - **Distance – Volume Descending** (no threshold – only valid for triggers with a Trigger Type of **Continuous**),
 - **Distance – Volume Ascending** (no threshold – only valid for triggers with a Trigger Type of **Continuous**),
 - **Distance – Pitch Descending** (no threshold – only valid for triggers with a Trigger Type of **Continuous**),
 - **Distance – Pitch Ascending** (no threshold – only valid for triggers with a Trigger Type of **Continuous**),

The final four distance types allow for sonification of a continuously looping sample where the volume or pitch either increase or decrease with distance. The comparison types are designed to be used with a trigger of type **Continuous** where the watch IDs provided are those of the player 1 and player 2 X (i.e. horizontal) locations.

5. **Watch 1 ID** – The ID of the watch associated with this trigger. The watch contains the pointer trail to the piece of memory we're interested in.
6. **Dependent Trigger ID / Watch 2 ID / Continuous Trigger ID** – This field is used for one of three different purposes based on whether this trigger is a **Normal**, **Continuous** or **Modifier** trigger - as outlined below:
 - **Dependent Trigger ID** – If this trigger is a **Normal** trigger, then this optional value may specify the ID of a secondary 'dependent' trigger which must also meet its own trigger condition for *this* trigger to be allowed to play a sonification event (i.e. sound sample). If the value specified here is **-1** then it means "this trigger does not depend on any further triggers". The reason for the ability to specify a dependent trigger is merely to help suppress triggers which may play at inopportune / non-sensical times such as between rounds or matches. Dependent triggers may

be a maximum of five triggers 'deep' – that is you can't have a trigger depend on more than five additional triggers. This is to minimise CPU usage and prevent cyclic dependencies which would cause the software to fail if **trigger A** had a dependency on **trigger B**, which itself had a dependency on **trigger A** and so on.

- **Watch 2 ID** – If this trigger is a **Continuous** trigger, then this value is the mandatory second watch related to the trigger. For example, a continuous trigger may be set up with the **Watch 1 ID** as the player 1 horizontal location, and the **Watch 2 ID** as the player 2 horizontal location so that the continuous trigger's volume or pitch may be modified as the distance between players changes. In this situation, the **Value** field of this continuous trigger (discussed below) is used as the maximum distance between players.
- **Continuous Trigger ID** – If this trigger is a **Modifier** trigger, then this field is used to store the ID of the continuous trigger that this trigger modifies. In this situation, the **Sample Volume** and/or **Sample Speed** fields of this modifier trigger control what modification is made to the specified continuous trigger.

The text on the label to the left of this textbox will change based on the trigger type selected.

7. Trigger Type – The type of the trigger. There are three possible trigger types available, which are:

- **Normal** – A standard trigger which may play once or many times per round while **In-Game**, or **In-Menu**, depending on whether it passes the threshold of its comparison type (that is, whether it passes the requirements to play the sample, such as "Is the clock less than 10?" or "Is the opponents health less than 250?" etc).
- **Continuous** – A trigger which plays a looped sample, typically while playing the game only (i.e. not in the menus). This trigger's sound may change based on the distance between the two watches and the **Value** field (used as the range between values) specified to control it.
- **Modifier** – A trigger which modifies a continuous trigger, for example by changing the continuous trigger's volume or pitch based on the conditions of this this trigger. A modifier trigger may be used to change the volume or pitch of a continuous trigger based on whether the second player is crouching (and hence susceptible to overhead attacks) etc. If a modifier trigger has a volume of 0.5 when it means it will multiply the current continuous trigger's volume by that value when the modifier condition is met, essentially halving the volume, and then divide by the volume to restore the standard continuous trigger volume when the trigger condition is no longer met.

A walkthrough / discussion of the process of finding pointer trails for watches and creating triggers can be found below in the **Finding and Using Watches and Triggers** section of this document.

8. Trigger Value / Max Range – For a trigger with a type of **Normal** or **Modifier** this is the value that the comparison type must meet to activate a sonification event. For a normal trigger the value must pass through a threshold (listed above under **comparison types**), while for a modifier trigger no threshold is necessary. For a **Continuous** trigger this value means **Max Range** which specifies the maximum difference between the values of the watches for the continuous trigger.

9. **Sample Filename** – The filename of the sample to be played. It's best if this filename is simply the name of the file within the config directory rather than a relative path to the sample so that game configs can be copied / moved / distributed in a single operation by simply copying the config folder. There is no minimum or maximum length of sample that can be used, as shorter samples will be loaded into memory and longer samples will be automatically streamed – but keeping the sonification samples 'short-and-sweet' where possible will stop multiple samples from overlapping as they play.
10. **Select Sample File Button** – Rather than directly entering the sample name, this button can be used to open a file dialogue from which a sample can be chosen.
11. **Sample Volume** – The volume to play the sample when it's activated. The range is between 0.0 (completely silent – which isn't very useful) to 1.0 as maximum volume.
12. **Sample Speed** – The speed with which to playback the sample. The range is between 0.1 (which would be one tenth of normal speed) and 4.0 (which would be four times normal speed). Default is 1.0.
13. **Is Clock** – This checkbox indicates whether this trigger is the clock (or 'round-timer' if you prefer). There should only be a single trigger marked as the clock per game config, and this trigger does not play a sample / sonification event. Instead, the watch of this trigger is polled to see if the value is changing or not. If the value is periodically changing then SoniFight can know with a high degree of confidence that we are "In-Game", and as such that only Triggers marked as "In-Game" or "Any" should be allowed to play. If the clock value is not moving, then after two 'ticks' of the clock (as specified in the game config's **Clock Tick (Milliseconds)** property in the main game config settings) we assume that the game is either paused or we are in the main menus, which allows any triggers marked as "In-Menu" to play.
14. **Allowance Type** – This dropdown menu specifies whether this trigger is allowed to activate based on whether the clock / round-timer is changing or not. The available options are:
 - Only when the clock is changing (**In-Game**),
 - Only when the clock is not changing (**In-Menu**), or
 - Regardless of whether the clock is changing or not (**Any**).
15. **Active** – This checkbox controls whether this trigger will be used in the game config. Only triggers which are active will activate and produce a sonification event. Any watches this trigger depends upon must also be active for the trigger to activate. By turning triggers on and off, you get to enable or disable them as you choose without deleting them entirely – so you can configure the sonification to your preference without losing the trigger details should you decide that you do want a given trigger to be enabled after all.
16. **Delete Trigger** – This button will delete the current trigger. There is no undo facility or confirmation dialogue, but if you accidentally delete a trigger you can go back to the **Main** tab then back to the **Edit Config** tab to force a reload of the game config from its last saved state.

Finding and Using Watches and Triggers

As we've discussed, we need both a **Watch** (i.e. a memory address specified as a pointer trail along with the type of value to read from that address) as well as a **Trigger** (i.e. a condition that must be matched in order to play a sample) for SoniFight to do anything useful.

While there are no-doubt a number of different pieces of software which could help you locate a pointer trail to a value of interest, the software that I've been using and that I'll demonstrate is called **Cheat Engine**. This software is a freely available from the following URL: <http://www.cheatengine.org/>

Unfortunately, the actual process of finding pointer trails to specific values involves a series of steps that will likely be difficult for non-sighted people to perform. My hope is that with a little bit of sighted assistance, or even perhaps hiring someone through Amazon's Mechanical Turk (<https://www.mturk.com/mturk/welcome>) that the gaming community can help create some great configs which can then be freely shared to anyone who may benefit from them.

If you don't want to read through the below written description of finding pointer trails I've put together a video which demonstrates the entire process at the following URL:

TO DO - LINK TO POINTER FINDING DEMONSTRATION VIDEO HERE

Once you understand the pattern of what's happening it's not incredibly complex – but it can be rather repetitive. The upside is that once you've positively identified a pointer trail to a value of interest then it's yours forever and barring changes to the app which modify memory locations, it'll keep on working just fine as long as you need it!

Finding Pointer Trails in Cheat Engine via Pointer Scans

Part 1 – Finding an Initial Value

The basic process for finding a pointer trail to a value of interest using pointer scans is as follows. Let's say we're looking for a pointer to the clock in a fighting game, we can start the game and start a match then immediately pause the game and we might see that the clock's initial value is 99. So somewhere in memory the game is keeping track of the clock – now we just need to find where!

The first thing we might do is launch Cheat Engine, connect to the game process, and then do a memory search for any 4 byte value (i.e. an integer or a float) which has the value 99. When initially searching, it's best to change the **Writable** checkbox to the **indeterminate state** to indicate that we don't care if the memory is writable or not (its initial setting is to only find memory addresses which can be written to).

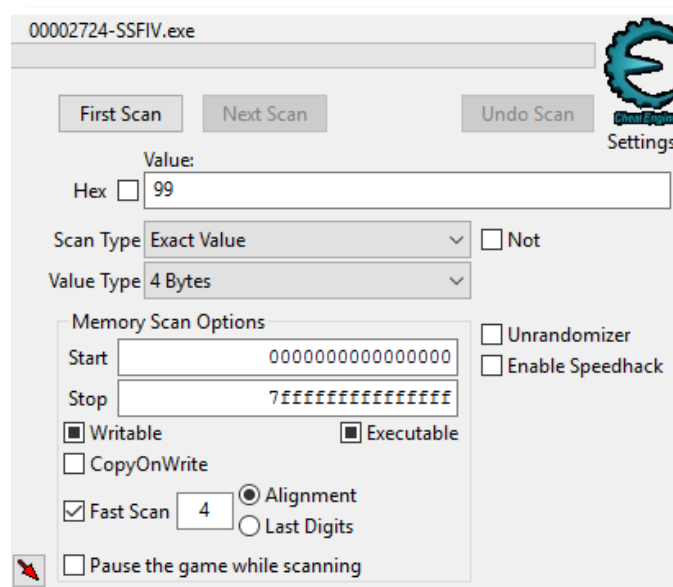


Figure 7 - Our initial scan for the value 99 as a 4 byte value where results may be writable or read-only.

When we perform this initial search we will likely find that there are dozens, if not hundreds or thousands of memory values which are currently 99 – and that's okay, but only **one** of the values will actually be the clock value we're interested in - so we need a way to narrow down the results further.

To do this, we might **un-pause** the game and allow the clock value to tick down a few seconds. Let's say it ticks down from 99 to 93. What this means is that somewhere in our result list of values which were 99, one of those values should now be 93. So what we could do is perform a "Next Scan" either looking for values in our list which are now equal to 93, or alternatively we might like to scan for values in our list which have decreased by 6 (because 99 – 93 equals 6). Either will be fine, and what we're hoping to find is that our large list of results has now been significantly cut down to maybe just a handful, or if we're lucky just a single result.

If the number of results remaining is still large, we might un-pause the game for a few seconds and then pause it again, then re-filter our result list for only those results which have the current value of the clock. This process can be repeated until we're down to just one or two results.

If we want to be absolutely sure that a found value is the one we want, we can double-click on the result to add it to Cheat Engine's details panel, and then click the checkbox on the left of the panel to freeze this value. If we

then un-pause the game and the clock no longer ticks down, or if we were finding an ammunition count in a game and we fired a gun but the number of bullets remaining did not decrease, then we know we've found the correct memory address. Alternatively, rather than freezing the value we could double-click on it and change it and then go back to the game and check if the expected change has occurred in the game state.

Part 2 – Scanning for Pointers to a Value

Once a value has been located we are only part of the way done, as if we restart the game the previous memory address will very likely not contain the same attribute such as the clock or whatever it is we're search for. As such, our next step is to generate a pointer map for the memory address we have identified.

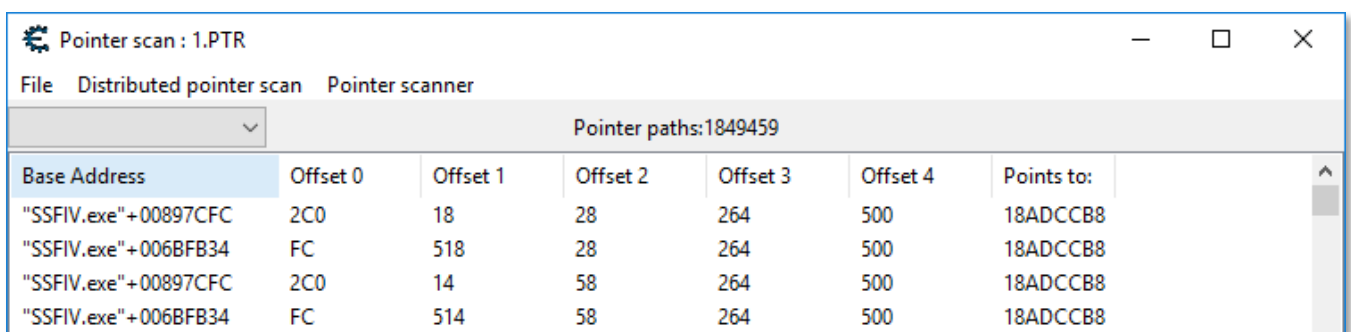
To do so, we can right-click and select **Pointer scan for this address** from the pop-up menu. This opens a **Pointerscanner scanoptions** window from which we can specify some scanning settings such as:

- **Maximum offset value**
 - This is how far apart pointers can be in memory and has a default value of 2047 bytes.
- **Max level**
 - This is the maximum number of *hops* to take to reach the memory address and has a default value of 5 hops.

For now let's leave these defaults in place and click the **[OK]** button to start the scan – it will warn us that there will be a lot of useless results, but we can accept that for now because this is our first scan. When we click **[YES]** to agree and start the scan we are prompted to save the results of the pointer scan. It's recommended to pick local storage location (i.e. not a network drive) that has lots of free space as sometimes these pointer scans can return multiple gigabytes of data. I typically create a folder called **ce-pointer-scans** and save the first scan to that directory with the name **1.PTR**.

When re-finding the Street Fighter IV clock when writing this documentation the pointer scan was only a few seconds and returned 1.8 million results which takes up only 20MB of file space. As mentioned, some scans can take a lot longer, and return significantly more results resulting in gigabytes of potential pointer trails based on your scan settings.

The results of the pointer scan will turn up in a new window, and take the form of the process name followed by an initial hexadecimal offset, followed by up to four additional hexadecimal 'hop' values (because we previously specified our maximum depth as 5) – and each of these pointer trails points to the memory address we identified, which in this example in the Street Fighter IV clock, which while writing this documentation I happened to find at the address **18ADCCB8**.



Base Address	Offset 0	Offset 1	Offset 2	Offset 3	Offset 4	Points to:
"SSFIV.exe"+00897CFC	2C0	18	28	264	500	18ADCCB8
"SSFIV.exe"+006BFB34	FC	518	28	264	500	18ADCCB8
"SSFIV.exe"+00897CFC	2C0	14	58	264	500	18ADCCB8
"SSFIV.exe"+006BFB34	FC	514	58	264	500	18ADCCB8

Figure 8 - An initial scan for pointers to the Street Fighter IV clock returns a touch over 1.8 million pointer trails.

Now that we have our pointer trails, we're making good progress but we're not there yet, because we don't know which of these pointer trails to use, so our next step will be to narrow down this list of potential trails.

The basic idea of how we narrow down the potential pointer trails is this:

1. We close down our game application,
2. We restart the game, re-connect to the game process in Cheat Engine and perform the steps required to find the memory address of our value of interest (so in this example we'd re-find the clock value),
3. Then, when we have the memory address for the clock again (which will very likely be a different memory address than the one we previously found) we then ***filter our previous pointer scan results*** and only keep the pointer trails which now point to the new memory address we found!

So if I do this and re-find the street fighter clock, in this particular run it now turns up at the memory address **197B7238** – which is a completely different location to where the clock was stored in memory on the previous run.

Now, double-clicking on the address and copying-and-pasting it, we can go back into our saved **Pointer scan : 1.PTR** window with our 1.8 million results and select **Pointer Scanner** and then **Rescan memory – Removes pointers not pointing to the right address** from the menu, and then paste in our new clock address and click the **[OK]** button and save the results as **2.PTR** in our **ce-pointer-scans** directory.

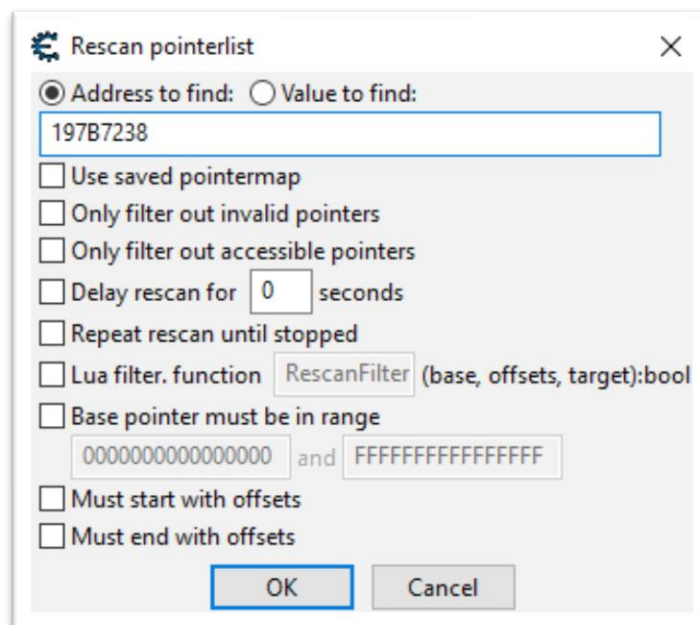
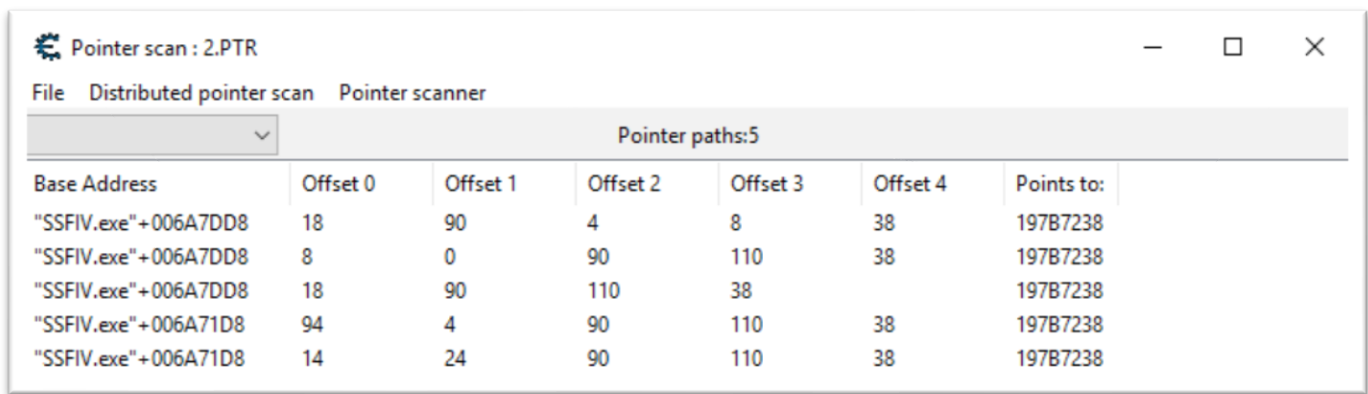


Figure 9 - Filtering our pointer scan results for the new feature address.

After performing this filtering, in this example, we've cut down the pointer trail from 1.8 million results to a mere **five** results! This means that it's very likely that any one of these pointer trails will be able to consistently find the Street Fighter IV clock across reboots.



Base Address	Offset 0	Offset 1	Offset 2	Offset 3	Offset 4	Points to:
"SSFIV.exe"+006A7DD8	18	90	4	8	38	197B7238
"SSFIV.exe"+006A7DD8	8	0	90	110	38	197B7238
"SSFIV.exe"+006A7DD8	18	90	110	38		197B7238
"SSFIV.exe"+006A71D8	94	4	90	110	38	197B7238
"SSFIV.exe"+006A71D8	14	24	90	110	38	197B7238

Figure 10 - A manageable number of results after filtering our initial pointer scan of 1.8 million potential pointer trails.

Each result is of the form: **Process-name+initial offset, offset 0, offset 1, offset 2, offset 3, offset 4**

So to use, say, the top result as a pointer trail to the Street Fighter IV clock in SoniFight, we just take those offsets and put commas between them, which would make our pointer trail:

6A7DD8, 18, 90, 4, 8, 38

We can omit the initial **00** before the first offset because those zeros don't change the value of the number.

If we wanted to, then we could close the game application then re-launch it and start a match, then pause the match and launch the **Pointer Trail Tester** provided with SoniFight and enter the process name and pointer trail to ensure that the trail is valid and that it's tracking the clock value successfully, as shown in the figure below. In this case the pointer trail is correct and successfully locates the clock value across restarts of the game, which is an excellent indication that it will work across reboots of the system, and as such on anyone's PC.

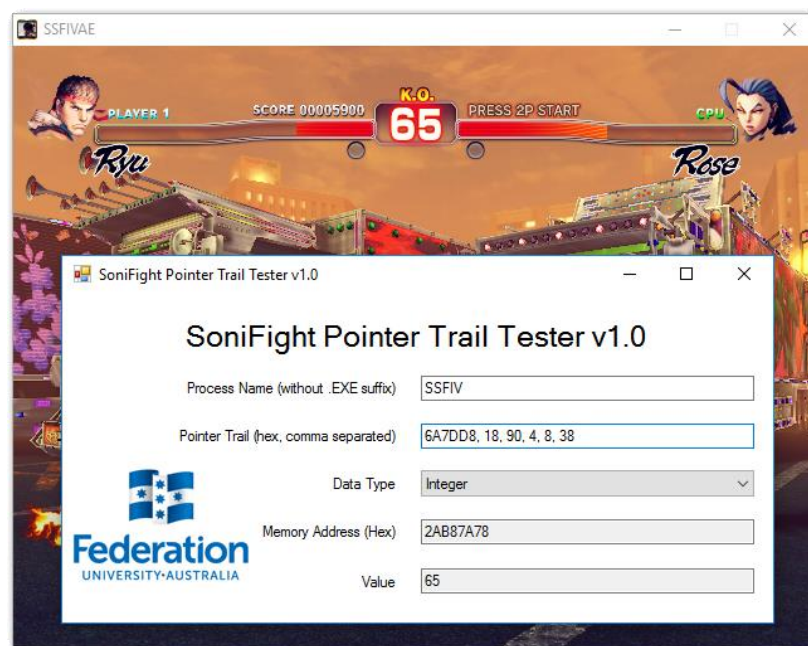


Figure 11 - The found pointer successfully locates the Street Fighter IV clock where both show a value of 65.

At this point we can delete the results of any pointer scans because we no longer need them.

The Clock Watch and Triggers

The most important watch and trigger that we find, and the first one we should always look for when creating a new game config for a fighting game, is the **clock** or **round-timer**. That is, the time value that counts down in a fighting game, and if it hits zero before either player has lost all their health then the person with the most health remaining wins the round.

By looking at this value and seeing whether it's counting down or not, SoniFight can know whether we're in a round or match (i.e. are we currently in-game and fighting someone?) or if whether we're in the menus (whether that's just because we've paused the game, or we're setting up a match in the main menus or such).

Once we have a watch that points to the clock value, and a trigger marked as the clock so we know to keep track of it and determine the in-game or in-menu state, then besides from SoniFight using it internally - we may decide to add triggers when we're half way through the round, or when the clock is low.

For example, if **watch 0** is the clock watch that points to the round timer, and **trigger 0** is the trigger marked as the clock – then we may decide to add another trigger that says when the value of the clock is 50 (so that we know when we're half-way through a round – which is typically 99 seconds in Street Fighter). Or we may decide to add a trigger when the clock hits 10 so that we know we don't have much time left in the round and may need to go on the offensive if our health is lower than the opponent's health.

Triggers to notify the user about 'mid-round' and low clock value exist in the shipped Street Fighter IV and Mortal Kombat 9 game configs.

Let's say that, just as a test, you wanted a sample to play when the value of the clock was 85. To do so, you could simply:

1. Find an existing clock trigger (triggers 2, 3 and 4 in the Street Fighter IV config or 5 and 6 in the Mortal Kombat 9 config),
2. Select your trigger of choice and click the **[Clone Current Trigger]** button,
3. Change the value of the trigger to **85**, and
4. Select a sample to play when that value is met. The sample can be anything, but must be in .ogg, .mp3 or .flac formats.

That's it! Save the game config then run it, and when the clock hits 85 your trigger will activate and play the sample! If for some reason the trigger doesn't activate, check out the **Help! My Trigger Doesn't Make a Sound!** section in the FAQ at the end of this document.

Normal Triggers

Normal triggers may be set to activate only during active rounds/matches ('In-Game'), or only in the menus (when the clock value is not changing) or in either.

You'd typically use a normal trigger to warn about things like:

- The clock being halfway or low,
- You or your opponent's health being low,
- You or your opponent gaining a set amount of meter / 'bar' (i.e. super bar for EX or 'super' moves, ultra bar for ultra-combos / 'criticals' etc).

I've provided computer generated voices in English for most of the triggers in the configs that ship with SoniFight, but you may like to use special effects (chimes, beeps, explosions etc.) if you'd prefer. Really, it's whatever you think is best, but short-and-sweet would be preferred over playing a 30 second song sample when a condition is met, otherwise other triggers might be hard to make out over the already playing sample.

Continuous Triggers

Continuous triggers are typically used in-game only and are intended to provide continuous feedback about the game state via a continuously playing (i.e. looping) audio sample that has a characteristic such as volume or pitch modified based on what's happening in the game.

The initial thought behind continuous triggers was that, depending on the game, it may be difficult for a visually impaired player to know the distance between their character and the opponent. As such, a continuous trigger could be used that varies either the volume or pitch of the looping sample dynamically based on the distance between the players (that is, the distance between the player 1 and player 2 horizontal locations) – and that additional audio information can help inform the player about what may be good or poor actions to take in the game.

For example, in the provided Street Fighter IV game config, there are continuous triggers which can play in-game that can play a 'rushing-wind' or 'electrical-arc' type sound. These continuous triggers are tied into the player 1 and 2 horizontal location watches, and then based on the comparison type (volume or pitch ascending /descending) then volume or pitch of the sample is modified. Note that by ascending or descending I mean that ascending would play increase the sample volume or pitch as the distance between players decreases, while descending would be the reverse.

In a continuous trigger the **Value** field is used as **Max Range**, which means the maximum distance that the players may be apart. This can be determined by examining the values of both players horizontal locations (as reported by their watches – you may like to use the bundled **PointerTrailTester** software for this) when they are moved to opposite sides of the screen.

Only volume *or* pitch can vary on a continuous trigger, but this can be augmented so that both can be changed using a **modifier trigger** as discussed below.

Modifier Triggers

Modifier triggers are designed to modify continuous triggers based on other triggers defined in the game config. Although it's possible to use them on a normal trigger, because a normal trigger doesn't play continuously, then result of modifying the volume or playback speed of a sample which isn't playing will obviously be that nothing changes.

The idea behind modifier triggers came from one of the key tools in fighting games being overhead attacks which must be 'blocked high' (that is, you must be standing and blocking rather than crouching and blocking) to successfully block the attack). Overhead attacks are commonly airborne attacks such as jumping kicks or punches, but may also be character specific moves which are coded into the game to act as 'overheads', such as Ryu's Collarbone Breaker (input: towards and medium-punch) in Street Fighter IV.

While most fighting games will provide an audio cue when a player jumps, such as a grunt of effort as the player launches themselves into the air, there is typically no such audio cue when a player crouches. As such, a visually impaired player is disadvantaged as they have no feedback indicating that an overhead attack would be

successful or at least a viable option in this situation. As such, a modifier trigger may be used to dramatically change the volume or pitch of a currently playing continuous sample based on a game condition.

For a modifier trigger to work then the **Watch 1 ID** field must point to the watch that controls the modification, and the **Continuous Trigger ID** must point to the specific continuous trigger to modify. In the provided Street Fighter IV game config, there is a watch that keeps track of the opponent (i.e. player 2) player state which indicates if they are currently standing, jumping or crouching. When the opponent crouch state is detected then based on the sample volume and/or sample speed fields of the modifier trigger the continuous trigger's volume or pitch is modified.

Please note that if the continuous trigger changes the volume with distance then it would likely be more useful for the modifier trigger that alters it to change that continuous trigger's pitch, and vice versa. Also, as overhead attacks are typically close range, the continuous trigger would likely be more useful to increase in volume or pitch as distance between players decreases, otherwise when the players are close together the volume or pitch is low (and hence hard to hear) and any 'modified' change to them would be equally hard to hear, if the modification could be heard at all.

Pointer Trail Tester

Along with the SoniFight executable there's another executable called **PointerTrailTester.exe**.

This is a very simple app that's bundled with any SoniFight release, and its purpose is to display the value of the memory at a specific pointer trail when interpreted as a specified data type.

In essence, you simply provide three pieces of data:

1. The process name (for example, **SSFIV**),
2. The pointer trail (comma separated hex values, for example **6A7DD8, 18, 90, 110, 38** for the Street Fighter IV clock), and
3. The type of data to read (for example, **Integer** to read the Street Fighter IV clock as whole numbers).

After that it'll show you the value at that memory address which is updated 10 times per second. This is an easy way to determine that a pointer trail in a watch is working, and to examine what possible values the watch may have while you do various things in the game.

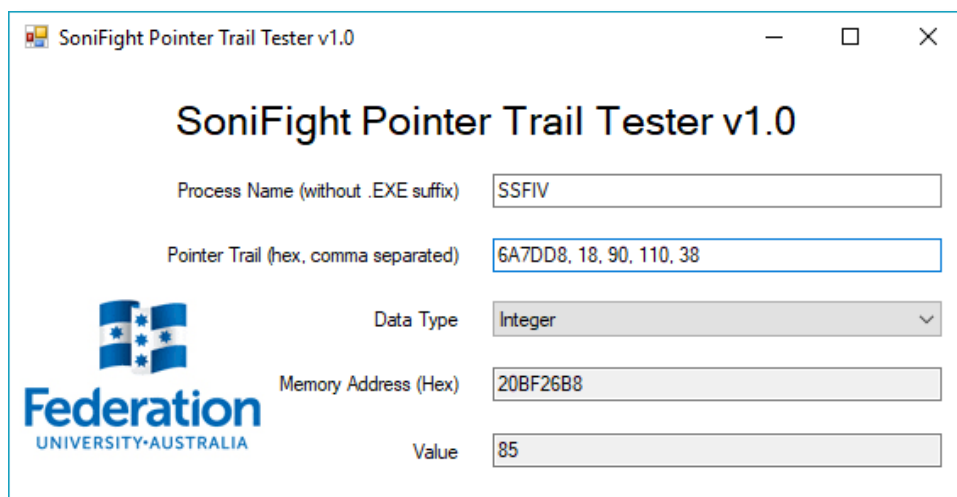


Figure 12 - Pointer Trail Tester showing the current Street Fighter IV clock as 85.

Frequently Asked Questions

Help! My Trigger Doesn't Make a Sound!

A precise series of conditions must be met for a trigger to activate, including:

- The watch the trigger depends on must:
 - o Have a 'good' (i.e. working) pointer trail,
 - o Have the correct data type set so it reads the correct amount of data in the correct format, and
 - o Be marked as active.
- The trigger using the watch must:
 - o Use the correct watch ID for the data you're watching,
 - o Have a value and comparison type which is actually met by the watch,
 - o Have a valid sample in the current config directory,
 - o Have sane volume and playback speeds (1.0 for each would be fine),
 - o Be marked as In-Game or In-Menu as appropriate for what you want to happen, and
 - o Be marked as active.

If you're not sure about whether the watch value is actually hitting the specific value used to trigger a sample you might want to put the watch's pointer trail and data type into the provided **Pointer Trail Tester** app and double check that your trigger condition is being met.

Also, while SoniFight is a windows forms application, it also writes some debug output to the console – so if you launch SoniFight via the Windows Command Prompt then additional information regarding what triggers have matched their conditions is available. If you wanted to write this debug output to file then you can simply launch SoniFight and pipe the output to a file using a command like this: **SoniFight.exe > log.txt**

Does SoniFight support game X? / Could you write a config for game X?

At present SoniFight only ships with two configs that support Ultra Street Fighter IV Arcade Edition and Mortal Kombat 9 (aka Mortal Kombat Komplete Edition) as proof of concept. However, SoniFight was built to run configurations for various games with the idea being that users can create a config for any game you want to add additional sonification cues to.

In terms of writing configs for requested games, the problem is that I'm only one man and as much as I'd love to I simply don't have the time to create additional configs for various games because as soon as this project ships I have to move on to the next one in an effort to gain my PhD in the short time I have remaining to do so.

However, while I might not have the time to create new configs - perhaps you do? There's comprehensive documentation in this user guide on how to use Cheat Engine to find pointer trails to values for use in new game configs for whatever fighting game you're interested in. Unfortunately, the process to find these pointer trails is difficult for a non-sighted person to perform, but I would hope that with some determination and/or sighted assistance configs could be made for a variety of different fighting games. And remember - once a config is made, it'll work forever (for that particular version of that particular game) - or even if one pointer trail is found, then it's found and there's no going back, so potentially making a solid game config could be a distributed 'many-hands-make-light-work' process, or at least that's my hope.

Does SoniFight use a lot of CPU or RAM? / Will it have a detrimental effect on game performance?

SoniFight will quite happily run using less than 1% CPU when using a game config with over 30 watches and 300 triggers and polling every tenth of a second, so it shouldn't affect the game's performance in any meaningful fashion. In terms of RAM usage it's directly dependent on the number and size of the samples associated with the game config (which all get loaded into memory). Before loading any samples the app will take up around 30MB of RAM, but even with the aforementioned game config loaded (which uses around 300 individual samples) we're still only up to around 60-70MB RAM usage.

Is SoniFight cheating? If I use it online will it get me banned from services like Steam?

SoniFight only aims to provide the same audio cues a sighted fighting game player has natively available, but through audio for those who may be partially or non-sighted. A sighted player will gain no real benefit from using this software because the information is already there visually - so I don't consider this cheating at all.

Whether using this software will get you banned from something like Steam is a harder question to unequivocally answer. I've been developing the software using Street Fighter IV running through Steam for over a year, including occasionally playing online matches, without any issues or problems. SoniFight only ever reads memory locations and provides sonification cues from the changes in values it encounters. It never writes to memory, and it does not attach a debugger to the host process. Please be aware that while I seriously doubt that you'd be banned from a gaming service for using this software, I cannot be held responsible should it occur and as the software license in LICENSE.txt states - you use this software entirely at your own risk.

I've made a config! Can you ship it with the next release?

Quite possibly! As long as your config works and does not use copyrighted audio materials I can incorporate it into the next release of the software so that more games are supported 'out-of-the-box' as it were. Please be aware that I can't ship copyrighted audio because I don't own the rights to do so, and unfortunately that includes ripping audio from the existing game (for example, the announcer saying the character names). While it would definitely make the audio more cohesive, as mentioned I don't have the right to distribute copyrighted audio.

Both my friend and I are partially or non-sighted, can we play against each other properly?

Yup! The configs that ship with this release provide sonification for both player 1 and player 2 using different voices so that they can be easily told apart. If you don't find that you can easily differentiate between the voices you may like to speed up or slow down the playback by modifying the trigger(s) associated with given in-game event(s) via the edit tab. You can also activate or deactivate triggers based on your preferences (i.e. you might decide you don't want any continuous sonification for distance and just disable any triggers that provide it).

I want to add additional triggers, how easy it is to do?

That depends on whether the watch associated with a trigger already exists, or if it has to be found. For example, if a watch exists for the player 1 health bar that triggers when they hit 500, 250 and 100 - and let's say you wanted to add a trigger for when player 1's health hits 750 - the easiest way would be to just clone one of the clock sonification triggers, say the 500 health trigger one, and change the matching value of the clone to 750 and give it a different sample to play (and rename the cloned trigger - it'll have the word CLONE appended to the name) and you're golden. That's the simple scenario.

If there isn't a watch for the specific value you want, then a pointer trail to that memory location must be found so that we can repeatedly find the value across game launches and reboots (i.e. it should work every time on everyone's PC, not just this one time on your PC). Further details on the process of finding pointer trails are

provided above in the ***Finding and Using Watches and Triggers*** section of this user documentation. Once a pointer trail to the value of interest is found and a watch has been created to monitor that memory location, then one or more triggers can be created which use that watch and respond to changes in value.

I only want some of the triggers to play / random non-sensical menu triggers sometimes play, can I disable them?

Absolutely. Every trigger has an active flag associated with it - just select the trigger(s) you want to turn off and uncheck the "Active" checkbox for that trigger in the edit tab. Alternatively, you can delete the offending trigger(s) entirely if you prefer. Watches may also be disabled by unchecking their active flag, but check that the watch isn't being used by any active triggers first or they'll stop working. Details of which active triggers use any given watch are shown in the details pane of the edit tab.

Finally, for ***Normal*** triggers there is the option to add a dependent trigger ID which can stop the trigger from activating if the dependent trigger condition is not met. For example, if a trigger saying the game resolution – let's say "640x480" keeps triggering between rounds, then you may be able to add a dependent trigger that checks that we're in the graphics options submenu. If you can find a condition that can determine that – and we're not, then the resolution-saying trigger won't activate between rounds because the dependent trigger won't be met. You can add up to a maximum of 5 dependent triggers for any given trigger that you might want to only activate under only very specific conditions.

How are configs shipped?

Each config is simply a subfolder that lives inside SoniFight's **Configs** folder. It contains the file ***config.xml*** (which stores all the GameConfig details for that particular game) along with a number of audio samples which are played when trigger conditions are met. If you've created a config and want to share it with someone, you can simply zip up the config folder, send it to someone and tell them to extract it inside their **Configs** folder to be added to the list of available configs. Sharing is caring!

What platforms does SoniFight run on?

SoniFight runs on Windows 7 SP1 and above only. The SoniFight application itself is 32-bit and can only connect to 32-bit processes, but it will happily run on a 64-bit system just like any other 32-bit process. Should there be sufficient demand I could also provide a native 64-bit version of SoniFight, but this would then only be able to connect to 64-bit processes.

Can I have access to and modify the SoniFight source code? Can I sell it?

Yes and no. SoniFight is released under a M.I.T. license, which broadly means that you may have the source code for no charge and that you can do with it as you please - including modifying it to your heart's content. If you're technically minded and provide a worthwhile pull request to the Github codebase then I'll happily merge it in and credit you.

However, SoniFight uses the irrKlang library for audio playback, and while free for non-commercial use, you cannot sell the irrKlang component of the SoniFight software without purchasing an irrKlang Pro (i.e. commercial) license to do so. For further details of irrKlang licensing, please see:

http://www.ambiera.com/irrclang/irrclang_pro.html

I have an issue with the software or a question that's not covered here.

While I've tested the software extensively during its creation and tried hard to put in some decent error-handling code, there are a lot of 'moving parts' in this application and it's quite possible there'll be some corner-cases I've missed. Also, if you've manually edited the ***config.xml*** file and displeased the XML gods then it may fail to deserialize (in which case it should tell you that the GameConfig object was **null** after loading).

Other than that, please feel free to raise any bug reports or issues on GitHub at the following URL – and if it's something I can fix then I'll endeavour to do so: <https://github.com/FedUni/SoniFight/issues>
