SoniFight User Guide

Alastair Lansley / Federation University Australia

a.lansley@federation.edu.au

Contents

1 Introduction

SoniFight is a windows application designed to provide additional sonification cues to video games, especially fighting games, for blind or visually impaired players.

The software 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 and tolk screen reader abstraction library.

To run SoniFight you can either download a precompiled binary release or build the Visual Studio 2017 solution yourself. Then launch a SoniFight executable (32 or 64 bit depending on the game you're connecting to), choose a game config for the game you want to play, click the "Run Selected Config" button and launch the game that your selected game config targets.

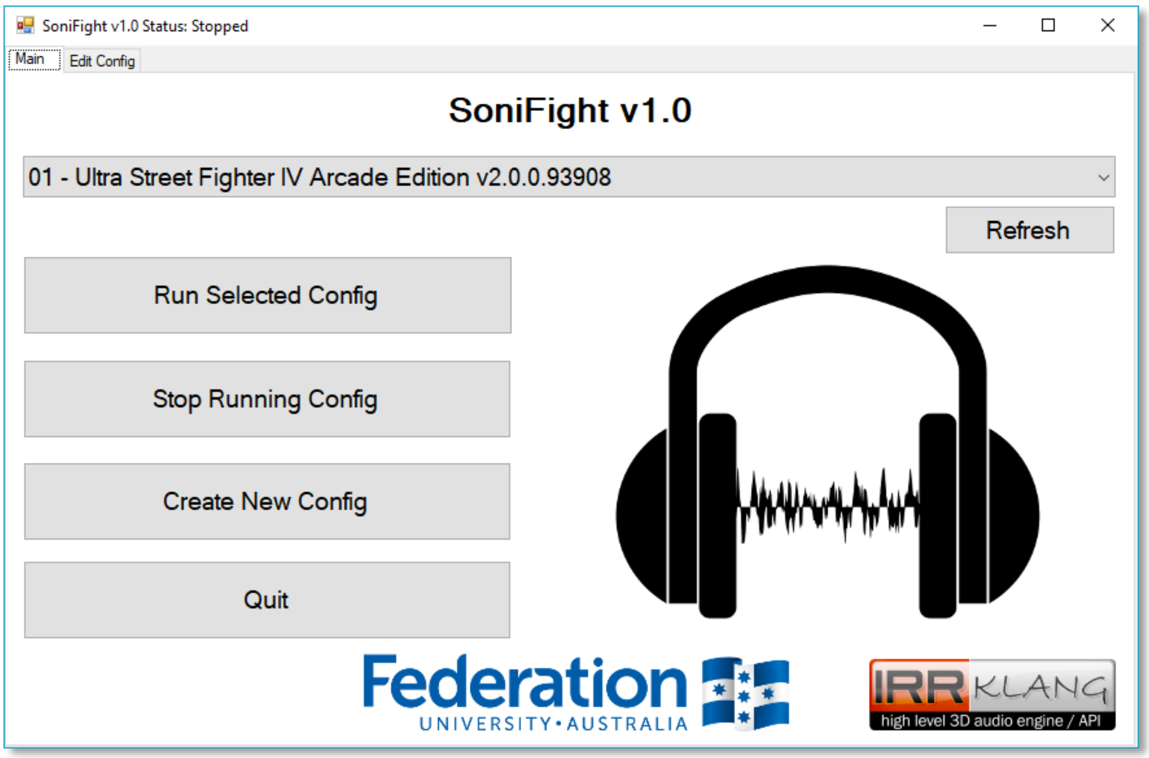
SoniFight presently ships with game configs to add sonification to Ultra Street Fighter 4 Arcade Edition (Steam version), Mortal Kombat 9 (i.e. Mortal Kombat Komplete Edition, Steam version) and a basic config for Killer Instinct (Windows Store edition, 64-bit).

Once running, SoniFight will provide a variety of additional sonification cues such as clock, health and meter-bar status updates for both players including details of many menu options as they are selected so that there is less need to memorise sequences of menu options.

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 chains requires additional free software such as Cheat Engine (http://cheatengine.org) 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.

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


SoniFight v1.0 Status: Stopped
Main
Edit Config
SoniFight v1.0
01 - Ultra Street Fighter IV Arcade Edition v2.0.0.93908
Refresh
Run Selected Config
Stop Running Config
Create New Config
Quit
Federation UNIVERSITY·AUSTRALIA
IRRKLANG high level 3D audio engine / API

2 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: https://www.youtube.com/watch?v=qHvcVv_BdmE

Figure 2- A screen capture of the SoniFight demonstration video.

SoniFight

Software to provide additional sonification cues to video games for visually impaired players.

Alastair Lansley, Peter Vamplew, Cameron Foale and Philip Smith

To run the **SoniFight** and **PointerChainTester** 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

3 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 you can extract the zip file wherever you'd like and run either of the 32 or 64-bit versions from the provided batch files, or launch the respective executables directly from their subfolders.

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 2017 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 2017, then the Community Edition may be freely downloaded from Microsoft at:

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

As mentioned previously, 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 4 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 4 config provided, memory usage is approximately 70MB.

4 File Structure

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

Figure 3 - The SoniFight release directory structure, as of SoniFight v1.0.

```
C:\SoniFight
│   build.txt
│   License.txt
│   SoniFight_x86.bat
│   SoniFight_x64.bat
└───bin/x86
│   ├───dolapi32.dll
│   ├───ikpFlac_32.dll
│   ├───ikpMP3_32.dll
│   ├───irrKlang.NET4.dll
│   ├───nvdaControllerClient32.dll
│   ├───SAAPI32.dll
│   ├───PointerChainTester_x86.exe
│   ├───SoniFight_x86.exe
│   ├───Tolk.dll
│   ├───TolkDotNet.dll
│   └───Configs
│   │       └───Various...
│   └───fr
│            └───SoniFight.resources.dll
│            └───PointerChainTester.resources.dll
└───bin/x64
│   ├───dolapi64.dll
│   ├───ikpFlac_64.dll
│   ├───ikpMP3_64.dll
│   ├───irrKlang.NET4.dll
│   ├───nvdaControllerClient64.dll
│   ├───SAAPI64.dll
│   ├───PointerChainTester_x64.exe
│   ├───SoniFight_x64.exe
│   ├───Tolk.dll
│   ├───TolkDotNet.dll
│   └───Configs
│   │       └───Various...
│   └───fr
│            └───SoniFight.resources.dll
│            └───PointerChainTester.resources.dll
└───Documentation
│   ├───SoniFight_User_Guide.pdf
│   └───SoniFight_User_Guide.html
│          └───SoniFight_User_Guide_files
```

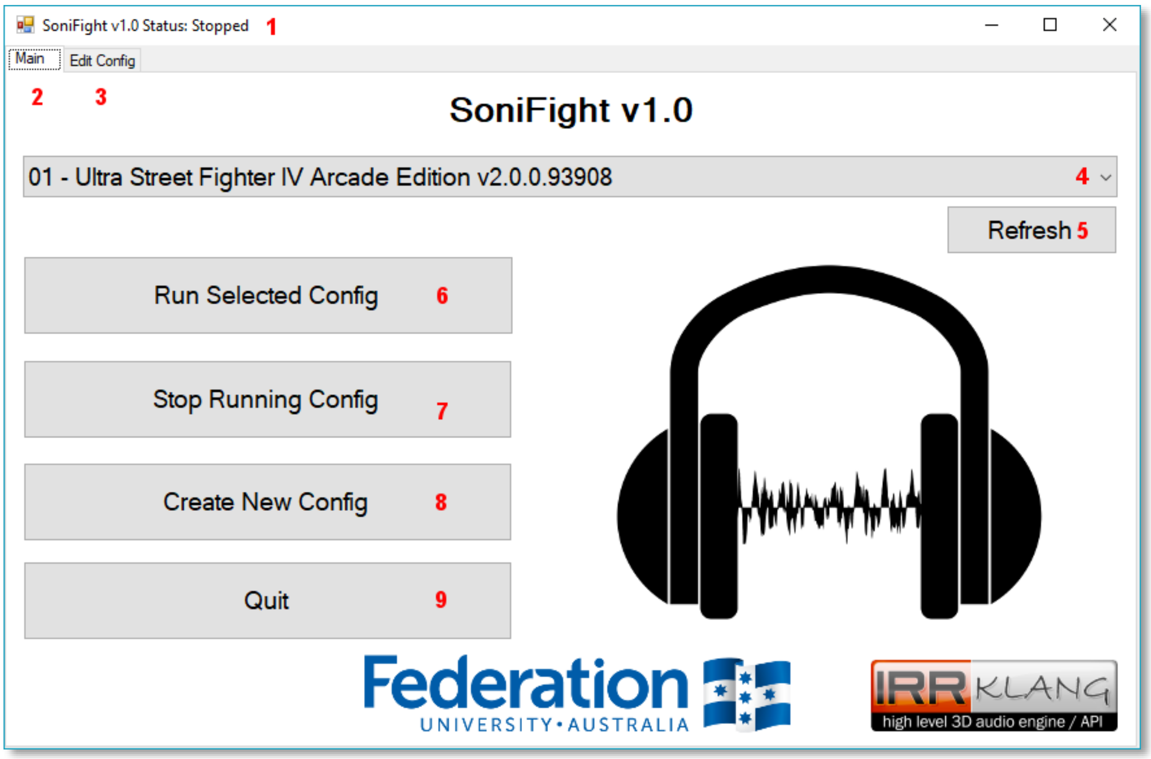
The **build.txt** file identifies the overall version of the SoniFight software. This version number may not necessarily match the individual versions of the SoniFight and pointer chain tester components which may change independently, however any increment of either component's version number will result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

The **bin/x86** and **bin/x64** contain 32-bit and 64-bit versions of SoniFight and the Pointer Chain Tester applications, respectively. The **Configs** folder contains subfolders for each separate game config which contain the **config.xml** for that game along with any related audio samples.

Please note that while SoniFight does not explicitly disable the use of relative paths, it is advisable to keep individual game config folders (i.e. the ***config.xml*** plus any samples) in the same directory so that game configs can be transferred and 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 each config independent of all others.

5 User Interface Elements – Main Tab

Figure 4 - SoniFight Main tab user interface elements.


SoniFight v1.0 Status: Stopped
1
Main
Edit Config
2
3
SoniFight v1.0
01 - Ultra Street Fighter IV Arcade Edition v2.0.0.93908
4
Refresh
5
Run Selected Config
6
Stop Running Config
7
Create New Config
8
Quit
9
Federation UNIVERSITY•AUSTRALIA
IRRKLANG high level 3D audio engine / API

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 and save 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 the new addition 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 process without issue. If you intend to or the config uses a screen reader for tolk-based sonification then you should have that up and running before

electing to run the config so that that it can be found and used.

7. **Stop Running Config button** – Stops providing sonification and unloads all samples.

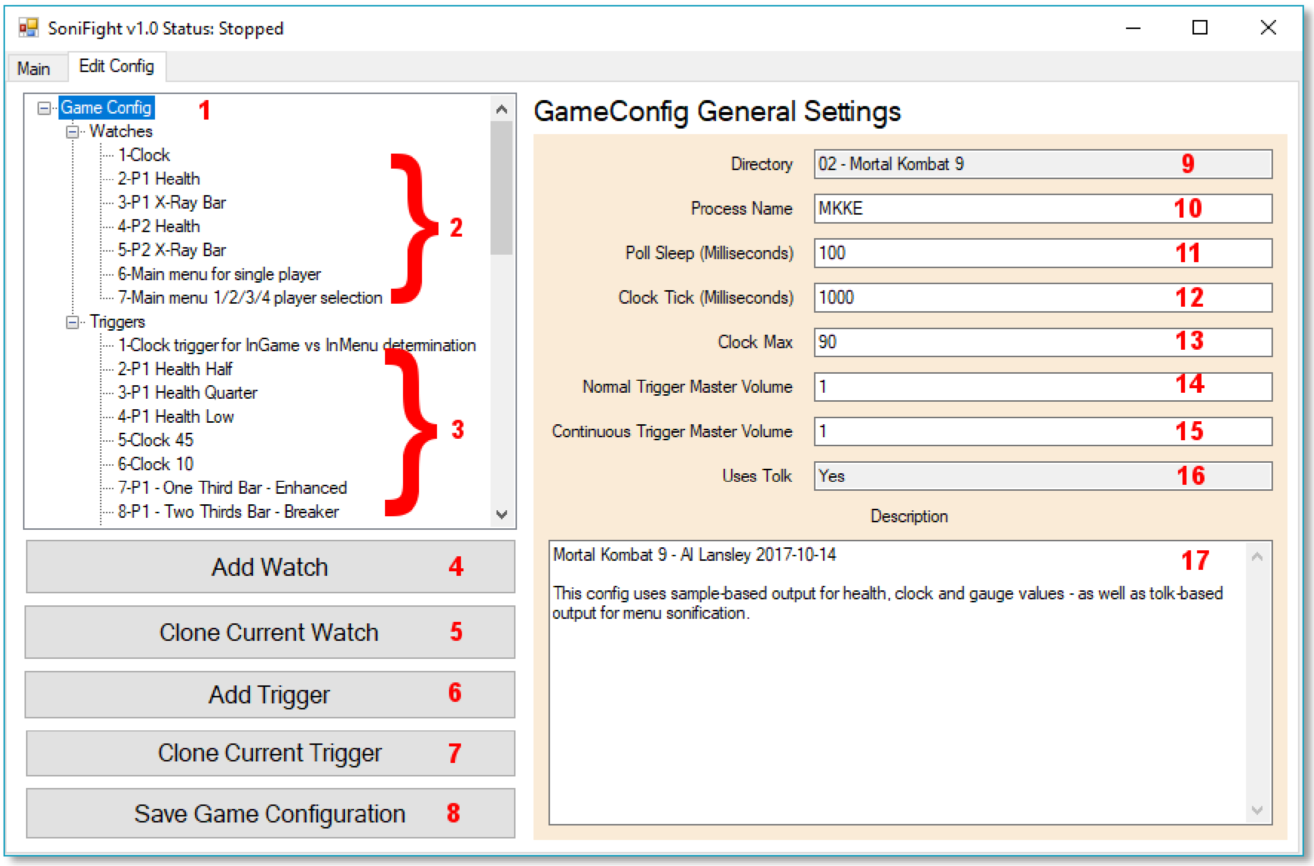
8. **Create New Config** – Creates a new, blank game config and switches to the **Edit tab**. On save, a folder with the name of the config as specified by the config directory will be created, within which the **config.xml** file will exist.

9. **Quit button** – Exits the SoniFight software.

6 User Interface Elements – Edit Config Tab

The **Edit Config** 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 5 - The Edit Config tab with the main Game Config tree node selected.


SoniFight v1.0 Status: Stopped
Main
Edit Config
Game Config
1
Watches
1-Clock
2-P1 Health
3-P1 X-Ray Bar
4-P2 Health
5-P2 X-Ray Bar
6-Main menu for single player
7-Main menu 1/2/3/4 player selection
2
Triggers
1-Clock trigger for InGame vs InMenu determination
2-P1 Health Half
3-P1 Health Quarter
4-P1 Health Low
5-Clock 45
6-Clock 10
7-P1 - One Third Bar - Enhanced
8-P1 - Two Thirds Bar - Breaker
3
Add Watch 4
Clone Current Watch 5
Add Trigger 6
Clone Current Trigger 7
Save Game Configuration 8
GameConfig General Settings

Directory	02 - Mortal Kombat 9	9
Process Name	MKKE	10
Poll Sleep (Milliseconds)	100	11
Clock Tick (Milliseconds)	1000	12
Clock Max	90	13
Normal Trigger Master Volume	1	14
Continuous Trigger Master Volume	1	15
Uses Tolk	Yes	16

Description 17

Mortal Kombat 9 - Al Lansley 2017-10-14

This config uses sample-based output for health, clock and gauge values - as well as tolk-based output for menu sonification.

6.1 Edit Tab – Tree View

The tree view is where you can choose which elements of the game config to modify – whether that's the main config options, or individual watches or triggers. You can, of course, modify the **config.xml** file manually using a text editor, 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 de-serialised. Unless absolutely necessary, it's advised to leave the config modification to the SoniFight software 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, master volumes for normal and continuous triggers, along with a field indicating whether this config uses talk-based output and the game config's overall description.

2. **Watches** – These tree nodes specify the pointer chain 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 multiple times per second.

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 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 distinguish it from the original.

8. **Save Game Configuration** – Saves the current configuration settings to the ***config.xml*** file within the directory specified for the game config. If saving the configuration file failed you'll receive a notification.

6.2 Edit Tab – Details Panel

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

1. **Directory** – The directory to save this game config to. This field is read-only unless you're creating a new game config.

2. **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.

3. **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.

4. **Clock Tick (Milliseconds) –** The estimated length of time in milliseconds that one ‘tick’ of the game clock (aka round timer) takes. This isn’t particularly important and the default of 1000 can be left alone, but it’s used to count game ‘ticks’ so SoniFight knows we’re either in a live game or in the menus by enforcing that at least a single ‘tick’ has passed before changing game state from “InMenu” to “InGame” or vice-versa.

5. **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.

6. **Normal Trigger Master Volume –** A master volume used to multiply all normal trigger volumes so they can be made quieter or louder in bulk. Range is 0.0 to 1.0.

7. **Continuous Trigger Master Volume –** A master volume used to multiply all continuous trigger volumes so they can be made quieter or louder in bulk. Range is 0.0 to 1.0.

8. **Uses Tolk** – A simple yes/no notification about whether this game config has any active triggers which use tolk-based output. This field is read-only.

9. **Description** – An optional multi-line text box where you can write some details regarding the game config as you wish. There is no particular limit on the amount of text you can enter here, and vertical scroll bars will appear when there is more text than will fit in the current textbox.

6.3 Creating a New Config

To create a new, blank game config you can click the **[Create New Config]** button from the main tab. Once done you’ll have a new blank config and the directory field will be editable. Once you save the config the directory name will be created, a **config.xml** file placed within, and the directory field will become read-only.

If you want to experiment with modifying a config but don’t want to risk breaking the original then you can simply duplicate the config folder (i.e. copy & paste it in the same location), and then click the **[Refresh]** button in the Main tab for the new config to be available for selection and modification in the main tab’s dropdown menu.

6.4 Creating Watches

To create a new, blank watch click the **[Add New Watch]** button from the edit tab. It will be given a unique ID but all other values will be placeholders.

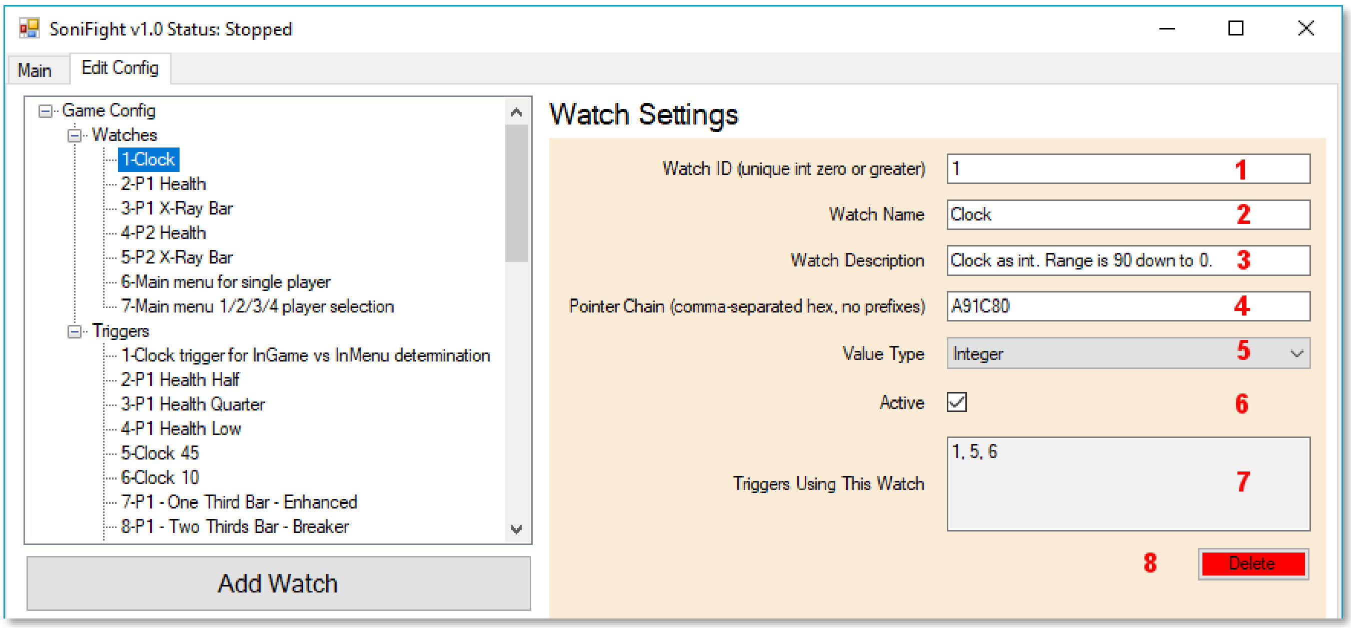
To clone an existing watch, select a watch from the left-hand watch sub-tree and then click the **[Clone Current Watch]** button. After which a new watch will exist with the details copied from the original watch and the text “-CLONE” appended to the name.

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 value of interest.

As such, a watch must use a kind of **relative address** in the form of a ***pointer chain***. This is a series of ‘hops’, starting at where the game process is loaded into memory, jumping to the next ‘hop’, reading the address at that location and then repeating the same process until we get to the final hop which – which rather than containing a memory address will contain the value of interest such as the clock or a player's health or ammo etc.

A watch has the following user interface elements:

Figure 6 – Example settings and user interface for a Watch.


SoniFight v1.0 Status: Stopped

Main | Edit Config

- Game Config
  - Watches
    - 1-Clock
    - 2-P1 Health
    - 3-P1 X-Ray Bar
    - 4-P2 Health
    - 5-P2 X-Ray Bar
    - 6-Main menu for single player
    - 7-Main menu 1/2/3/4 player selection
  - Triggers
    - 1-Clock trigger for InGame vs InMenu determination
    - 2-P1 Health Half
    - 3-P1 Health Quarter
    - 4-P1 Health Low
    - 5-Clock 45
    - 6-Clock 10
    - 7-P1 - One Third Bar - Enhanced
    - 8-P1 - Two Thirds Bar - Breaker

Add Watch

Watch Settings

Watch ID (unique int zero or greater)	1	1
Watch Name	Clock	2
Watch Description	Clock as int. Range is 90 down to 0.	3
Pointer Chain (comma-separated hex, no prefixes)	A91C80	4
Value Type	Integer	5
Active	☑	6
Triggers Using This Watch	1, 5, 6	7
	Delete	8

The numbered UI elements 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 Chain** – This is a series of one or more comma-separated hexadecimal values used to offset from the game process' base address to find a value of interest. Do not include any ***0x*** prefixes or such to indicate that offsets are in hexadecimal format.

   For example, the following pointer chain 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 chains work and can be found is provided in section ***8 Finding and Using Watches and Triggers***.

5. **Value Type** – The above ***Pointer Chain*** provides enough information to locate a given value of interest, but once there we need to know what **type** of data we should read from that memory address, that is – how much data should we read and how should we interpret it?

The value type dropdown provides the following options for data types to read from the address:

- **Integer** (whole numbers, 4 bytes),
- **Short** (whole numbers, 2 bytes),
- **Long** (whole numbers, 8 bytes),
- **Unsigned Integer** (positive whole numbers, 4 bytes),
- **Float** (numbers with decimal places, 4 bytes),
- **Double** (numbers with decimal places, 8 bytes),
- **Boolean** (true or false [0 or 1], 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)** – Up to 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. It's possible that you may wish to deactivate a given watch in a config to temporarily disable it without losing the saved watch data – 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**. Please note that the active status of any given 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 that trigger is marked as active or not.

8. **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.

6.5 Creating Triggers

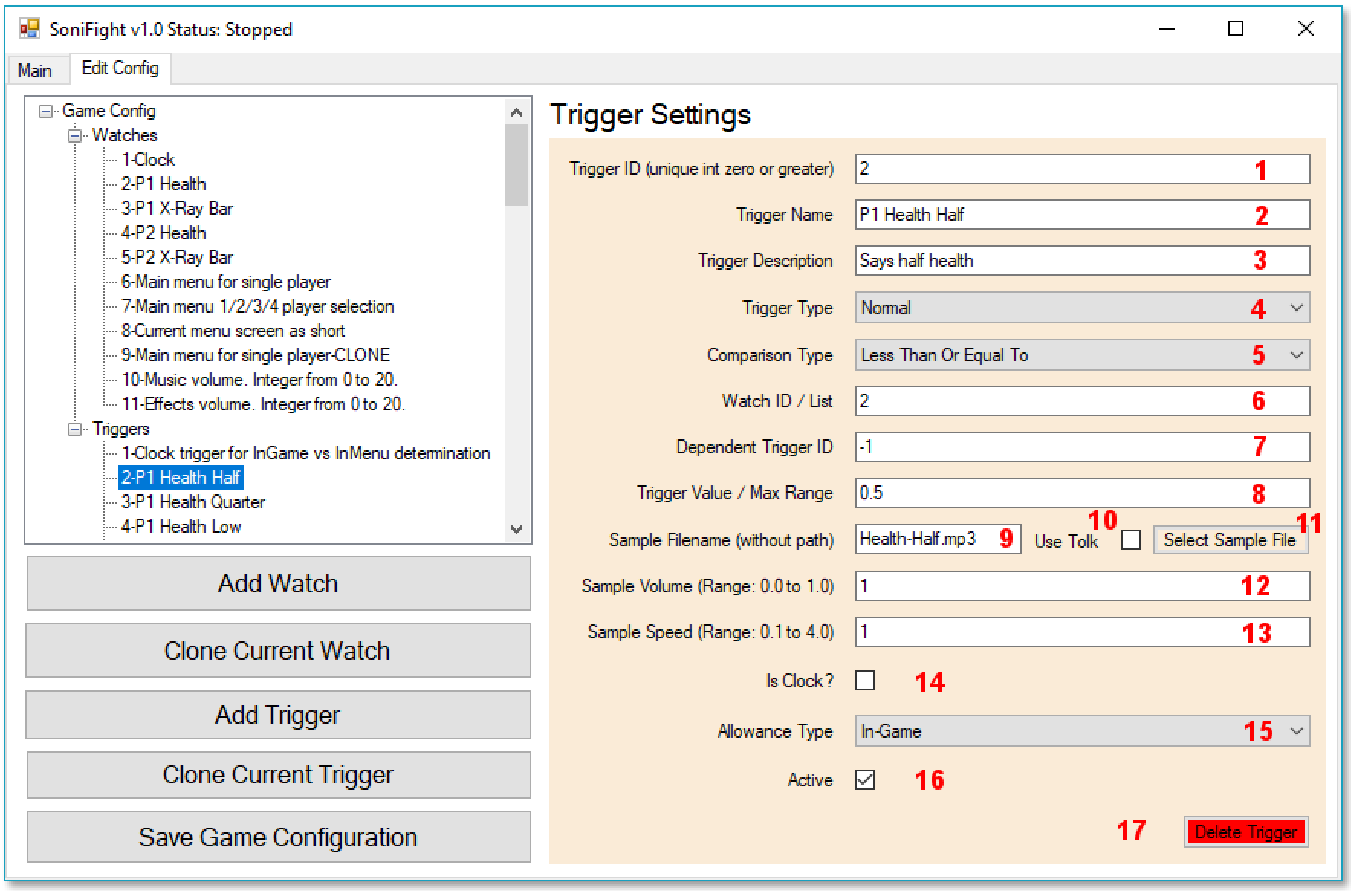
A **trigger** is a condition that we check against to determine whether we should play a sample (i.e. provide a sonification event) or not. Each trigger has its own unique ID, but will also depend on at least one watch (as specified by the **Watch 1 ID** field) along with some ***comparison criteria*** 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 trigger click the **[Add New Trigger]** button from the **Edit Config** tab. It will be given a unique ID but all other values will be placeholders.

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 as shown below in Figure 7:

Figure 7 – An example Trigger and its configuration details.


SoniFight v1.0 Status: Stopped

Main | Edit Config

- Game Config
  - Watches
    - 1-Clock
    - 2-P1 Health
    - 3-P1 X-Ray Bar
    - 4-P2 Health
    - 5-P2 X-Ray Bar
    - 6-Main menu for single player
    - 7-Main menu 1/2/3/4 player selection
    - 8-Current menu screen as short
    - 9-Main menu for single player-CLONE
    - 10-Music volume. Integer from 0 to 20.
    - 11-Effects volume. Integer from 0 to 20.
  - Triggers
    - 1-Clock trigger for InGame vs InMenu determination
    - 2-P1 Health Half
    - 3-P1 Health Quarter
    - 4-P1 Health Low

Add Watch | Clone Current Watch | Add Trigger | Clone Current Trigger | Save Game Configuration

Trigger Settings

Trigger ID (unique int zero or greater)	2	1
Trigger Name	P1 Health Half	2
Trigger Description	Says half health	3
Trigger Type	Normal	4
Comparison Type	Less Than Or Equal To	5
Watch ID / List	2	6
Dependent Trigger ID	-1	7
Trigger Value / Max Range	0.5	8
Sample Filename (without path)	Health-Half.mp3	9
Use Tolk	☐	10
Select Sample File		11
Sample Volume (Range: 0.0 to 1.0)	1	12
Sample Speed (Range: 0.1 to 4.0)	1	13
Is Clock?	☐	14
Allowance Type	In-Game	15
Active	☑	16
Delete Trigger		17

The numbered UI elements 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. **Trigger Type –** The type of the trigger. There are four possible options:

   - **Normal** – A standard trigger which may play only when 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).

   - **Dependent** – Similar to a normal trigger but without a sound to activate, it's condition check is only "is the condition met" without looking at a previous value for a threshold comparison (asides from if the comparison type is "changed").

   - **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. 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. For example, if a modifier trigger has a volume of 0.5 then 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.

5. **Comparison Type** – The comparison type is how we compare the trigger value (described below) against its previous value. For normal triggers the comparison must pass through this threshold to trigger a sonification event, while for dependent triggers a simple match is good enough.

   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.

6. **Watch 1 ID** – The ID of the watch associated with this trigger. The watch contains the pointer chain to memory address of a value of interest, and the type of the data to read from that address (e.g. integer, float etc).

7. **Dependent Trigger ID List / 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 List** – If this trigger is a **Normal** or **Dependent** trigger, then this optional value may specify one or more space-separated IDs of normal or dependent triggers which must also meet their own criteria in order for ***this*** trigger to be allowed to activate and provide a sonification event.

This is one of the most complex aspects of SoniFight. It sounds simple, that this trigger can only activate if another trigger's condition (as specified by the ID in this list) is met, but each trigger in the list may be a normal trigger that provides sound, or a dependent trigger which does not. The difference between how these triggers are evaluated is that normal triggers must pass a threshold value to activate (e.g. from more than to less than or equal to or such), while dependent triggers simply have to meet the required condition (e.g. equal to 3, or less than 50 etc).

If the dependent trigger ID is **-1** then it means "this trigger does not depend on any further triggers". Also, while normal triggers may have more than a single value in their dependency list, all other trigger types only use the first element of the list, or don't use it at all if it's **-1**.

- **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.

8. **Trigger Value / Max Range** – For a trigger with a type of ***Normal, Dependent*** 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***), for a dependent trigger it must meet the threshold, 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.

   Dependent triggers do not use a sample filename, while normal triggers with the "Use Tolk" flag output the text in the sample filename field, with substitutions as specified below.

10. **Use Tolk Checkbox** – Specifies whether this trigger should load and play the sample in the **Sample Filename** field, or use the text in that field as the speech to be output to a screen reader. You may substitute **{}** for the value of the current watch, or **{33}** to substitute the value of watch 33 (in this example) within the screen reader output.

11. **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.

12. **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.

13. **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.

14. **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.

15. **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***).

16. **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.

17. **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.

7 Different Types of Watches and Triggers

The most important watch that we find, and the trigger we should associate with it when creating a new game config for a fighting game, is the ***clock*** or ***round-timer***. That is, the time value that counts down during each round, 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 are still 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 to determine the in-game or in-menu state, then we can start to add additional triggers to provide further information about the game state.

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.

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. You will now have the cloned trigger selected, so change the value to activate the trigger to **85**, and finally
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. 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! s***ection in the FAQ at the end of this document.

You can also opt to check the "Use Tolk" checkbox and type some text into the sample filename field for that text to be output directly to your screen reader.

7.1 Normal Triggers

Normal triggers are the most common type of triggers used and 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 state ("Any").

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).

There are currently computer generated voices in English for many 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, or may have to wait for that sample to finish playing before another sample can play in the case of Normal triggers.

Alternatively, when creating a normal trigger you may check the "Use Tolk" checkbox and directly enter some text you want the screen reader to output into the sample filename field. When using tolk, you may also put a pair of curly braces to substitute the value of those braces with the exact value of this trigger's watch, or you may place a number inside curly braces to read out the value of that specific watch.

For example, placing a tolk output string of: "Health is {}" would output "Health is 373" if that was what the value of this trigger's watch happened to be. Placing a tolk output sting of "Health is {}, Opponent has {3}" would say something along the lines of "Health is 373, Opponent has 282" if the watch associated with this trigger had a value of 383 and the watch with ID 3 had the value 282.

7.2 Dependent Triggers

Dependent triggers are mostly the same as normal triggers but they don't make a sound. Instead, they are used as dependent conditions for normal triggers (as entered into the "Secondary ID / List" field) to be allowed to activate.

Normal triggers must pass a threshold to activate - that is, if a normal trigger has a condition to be less than 500 for it to activate – then it must have a previous value from the last poll of greater than or equal to 500 and a current value of less than 500. Dependent triggers do not have to 'cross the matching threshold' from their previous value and a naïve comparison based on the dependent triggers comparison type is made. The exception to this is the "changed" comparison type which compares against the previously polled value of this dependent trigger.

As an example usage scenario, in Mortal Kombat 9 the same memory address is used for all menu options for 1 player, 2 player, 3 players and 4 players. However, each of these player numbers may have different menu options, so a watch may be found that determines which screen the game is on. Then, we can use a bunch of normal triggers which activate the correct sonification for the 1 player settings while the dependent trigger indicates that the "1 player" option is active, the correction sonification for the 2 player settings while the dependent trigger indicates that the "2 players" option is active, and so on.

7.3 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 idea behind continuous triggers was that in a fighting game, and 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 4 game config, there is a continuous trigger which plays a 'rushing-wind' sound that changes volume based on the distance between players.

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 **PointerChainTester** software for this) when they are moved to opposite sides of the screen.

Only volume ***or*** pitch (not both) can vary on a continuous trigger, but this can be augmented so that either can be changed using a ***modifier trigger*** as discussed below.

7.4 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 4.

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 commonly no such audio cue when a player crouches (although exceptions to this do exist). 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. In this scenario 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 4 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.

8 Finding and Using Watches and Triggers

As described previously in this document, we need both a **Watch** (i.e. a memory address specified as a pointer chain along with the type of value to read from that address) as well as a **Trigger** (i.e. a condition that must be met in order to play a sample or output speech from a screen reader) for SoniFight to perform any useful work.

While there are no-doubt a number of different pieces of software which could help you locate a pointer chain 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/

If you decide to install Cheat Engine yourself then please be careful during install because it will offer to install some third-party software such as toolbars or anti-virus software, and it’s likely you neither want or need these.

Unfortunately, the actual process of finding pointer chains to specific values involves a series of steps that will likely be difficult for non-sighted people to perform. My hope is that with some determination and perhaps a little sighted assistance that the gaming community can help create some great configs which can then be freely shared to anyone who may wish to benefit from them.

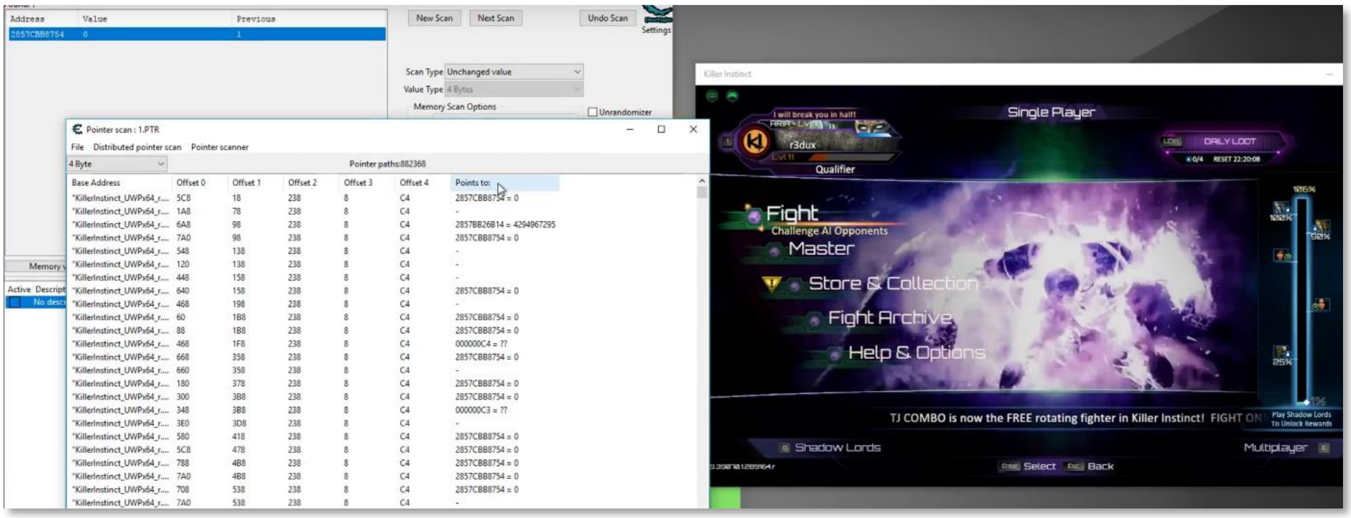
If you don't want to read through the below written description of finding pointer chains I've put together a few YouTube videos that demonstrate the entire process. These videos can be found at the following URLS:

Part 1 – Pointer Finding Basic Filtering Process: https://www.youtube.com/watch?v=qIw8nuOtnM8

Part 2 – Different Strategies for Locating Values: https://www.youtube.com/watch?v=omFAopx8FUk

Part 3 – Pointer Finding Tips, Tricks and Traps: https://www.youtube.com/watch?v=3XcFg5YbrPw

Figure 8 – An example screenshot of the pointer chain finding videos.



Address	Value	Previous
2857CBB8754	0	1

Base Address	Offset 0	Offset 1	Offset 2	Offset 3	Offset 4	Points to:
"KillerInstinct_UWPx64_r....	5C8	18	238	8	C4	2857CBB8754 = 0
"KillerInstinct_UWPx64_r....	1A8	78	238	8	C4	-
"KillerInstinct_UWPx64_r....	6A8	98	238	8	C4	2857BB26B14 = 4294967295
"KillerInstinct_UWPx64_r....	7A0	98	238	8	C4	2857CBB8754 = 0
"KillerInstinct_UWPx64_r....	548	138	238	8	C4	-
"KillerInstinct_UWPx64_r....	120	138	238	8	C4	-
"KillerInstinct_UWPx64_r....	448	158	238	8	C4	-
"KillerInstinct_UWPx64_r....	640	158	238	8	C4	2857CBB8754 = 0
"KillerInstinct_UWPx64_r....	468	198	238	8	C4	-
"KillerInstinct_UWPx64_r....	60	1B8	238	8	C4	2857CBB8754 = 0
"KillerInstinct_UWPx64_r....	88	1B8	238	8	C4	2857CBB8754 = 0
"KillerInstinct_UWPx64_r....	468	1F8	238	8	C4	000000C4 = ??
"KillerInstinct_UWPx64_r....	668	358	238	8	C4	2857CBB8754 = 0
"KillerInstinct_UWPx64_r....	660	358	238	8	C4	-
"KillerInstinct_UWPx64_r....	180	378	238	8	C4	2857CBB8754 = 0
"KillerInstinct_UWPx64_r....	300	3B8	238	8	C4	2857CBB8754 = 0
"KillerInstinct_UWPx64_r....	348	3B8	238	8	C4	000000C3 = ??
"KillerInstinct_UWPx64_r....	3E0	3D8	238	8	C4	-
"KillerInstinct_UWPx64_r....	580	418	238	8	C4	2857CBB8754 = 0
"KillerInstinct_UWPx64_r....	5C8	478	238	8	C4	2857CBB8754 = 0
"KillerInstinct_UWPx64_r....	788	4B8	238	8	C4	2857CBB8754 = 0
"KillerInstinct_UWPx64_r....	7A0	4B8	238	8	C4	2857CBB8754 = 0
"KillerInstinct_UWPx64_r....	708	538	238	8	C4	2857CBB8754 = 0
"KillerInstinct_UWPx64_r....	7A0	538	238	8	C4	-

Once you understand the pattern of what's happening it's not incredibly complex – but it can be a little repetitive and time-consuming. The upside is that once you've positively identified a pointer chain 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!

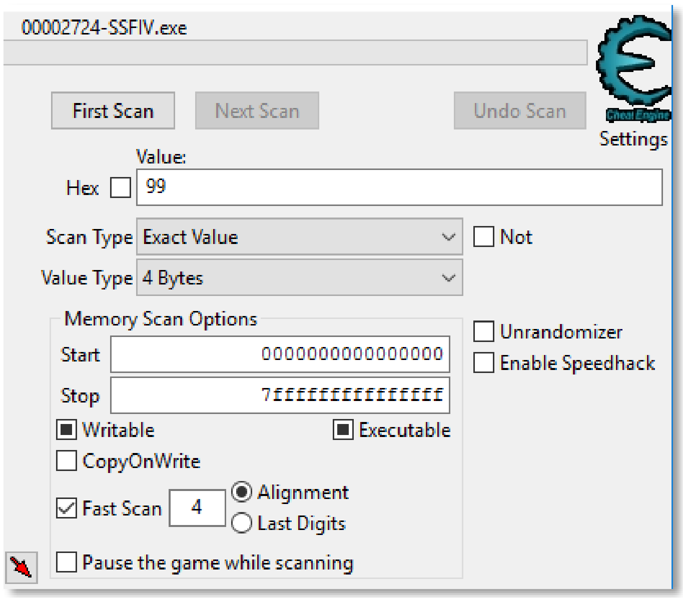
8.1 Finding Pointer Chains in Cheat Engine via Pointer Scans

Part 1 – Finding an Initial Value

The basic process for finding a pointer chain 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 out 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 9 - Our initial scan for the value 99 as a 4 byte value where results may be writable or read-only.



```
00002724-SSFIV.exe
Value:
Hex [ ]  99
Scan Type   Exact Value      [ ] Not
Value Type  4 Bytes
Memory Scan Options             [ ] Unrandomizer
Start  0000000000000000         [ ] Enable Speedhack
Stop   7fffffffffffffff
[■] Writable        [■] Executable
[ ] CopyOnWrite
[✓] Fast Scan  4   (●) Alignment
                   ( ) Last Digits
[ ] Pause the game while scanning
```

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 cheat table panel, and then click the checkbox on the left of the value to freeze it. 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 that we searched for. As such, our next step is to generate a pointer list 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.

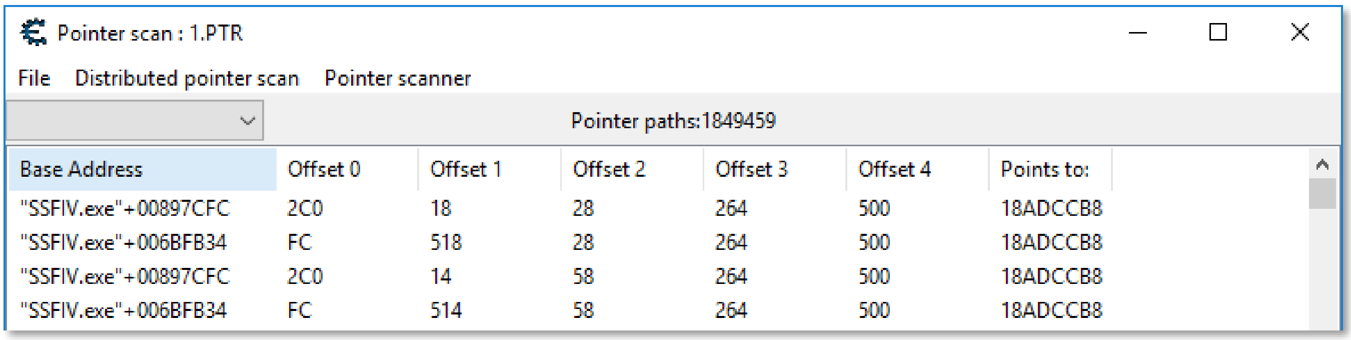
- **Max level**
    - This is the maximum number of ***hops*** to take to reach the memory address and has a default value of 5 hops.

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 **pointer-scans** and save the first scan to that directory with the name **1.PTR**.

When re-finding the Street Fighter 4 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 to run and return significantly more results resulting in gigabytes of potential pointer chains, especially if you increase the max level of hops to a value greater than 5.

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 five additional hexadecimal 'hop' values (because we previously specified our maximum depth as 5) – and each of these pointer chains points to the memory address we identified, which in this example is the Street Fighter IV clock. While writing this documentation I happened to find at the clock's memory address at: **18ADCCB8**.

Figure 10 - An initial scan for pointers to the Street Fighter 4 clock returns a little over 1.8 million pointer chains.



Pointer paths:1849459

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

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

Part 3 – Filtering our Pointer List

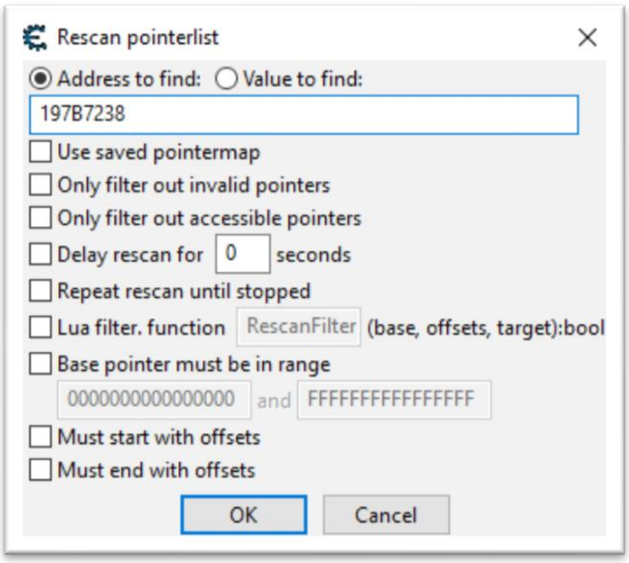
The basic idea of how we narrow down the list of potential pointer chains is as follows:

1. Close down the game so the process ends,
2. 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 again (so in this example we'd re-find the clock value),
3. Then, when the memory address for the clock has been located 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 chains that 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 memory address into the textbox at the top (in this case the default **Address to find** radio button is selected above it – which is exactly what we want to filter on) and then click the **[OK]** button and save the results as **2.PTR** in our **pointer-scans** directory.

Figure 8 - Filtering our pointer scan results for the new feature address.



Rescan pointerlist

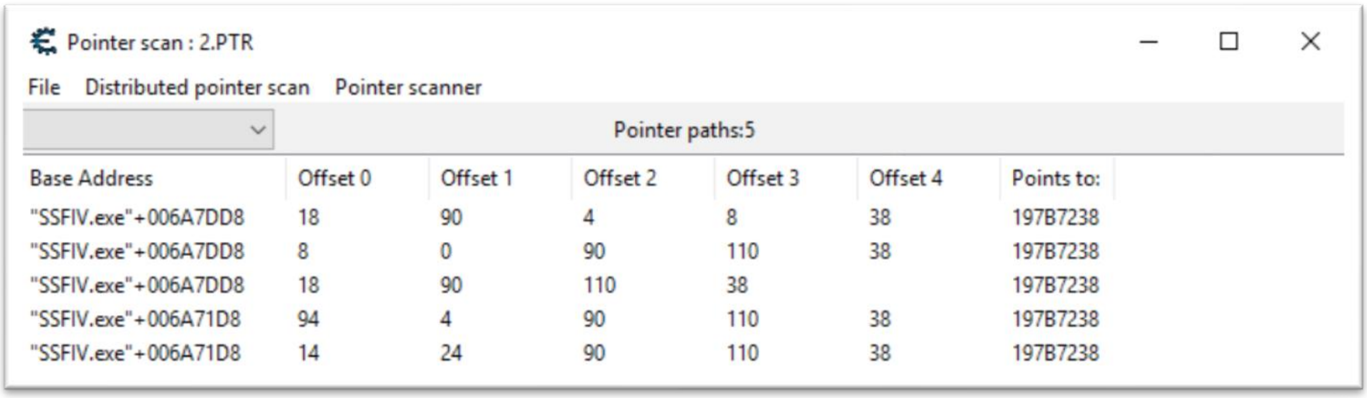
[✓ Address to find:] Value to find:

197B7238

- ☐ Use saved pointermap
- ☐ Only filter out invalid pointers
- ☐ Only filter out accessible pointers
- ☐ Delay rescan for 0 seconds
- ☐ Repeat rescan until stopped
- ☐ Lua filter. function RescanFilter (base, offsets, target):bool
- ☐ Base pointer must be in range
  0000000000000000 and FFFFFFFFFFFFFFF
- ☐ Must start with offsets
- ☐ Must end with offsets

After performing this filtering, in this example, we've cut down the list of pointer chains from 1.8 million results to a mere **five** results! This means that it's very likely that any one of these pointer chains will be able to consistently find the Street Fighter 4 clock across reboots.

Figure 9 - A manageable number of results after filtering our initial pointer scan of 1.8 million potential pointer chains.



Pointer paths:5

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

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

So our first result says: **"SSFIV.exe"+006A7DD8 18 90 4 8 38**

To use this result as a pointer chain to the Street Fighter 4 clock in SoniFight, we just take those offsets and put commas between them, which would make our pointer chain: **6A7DD8, 18, 90, 4, 8, 38**

We can omit the initial **00** before the first offset ***00**6A7DD8* because those zeros don't change the value of the number.

8.2 Pointer Chain Tester

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

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

To use the pointer chain tester you need to provide the following information to it:

1. The process name without the dot-exe suffix, such as **SSFIV** for Street Fighter 4,
2. The pointer chain as a series of comma separated hex values, such as **6A7DD8, 18, 90, 110, 38** for the Street Fighter 4 clock, and finally
3. The type of data to read, such as **Integer** to read the Street Fighter 4 clock as whole numbers.

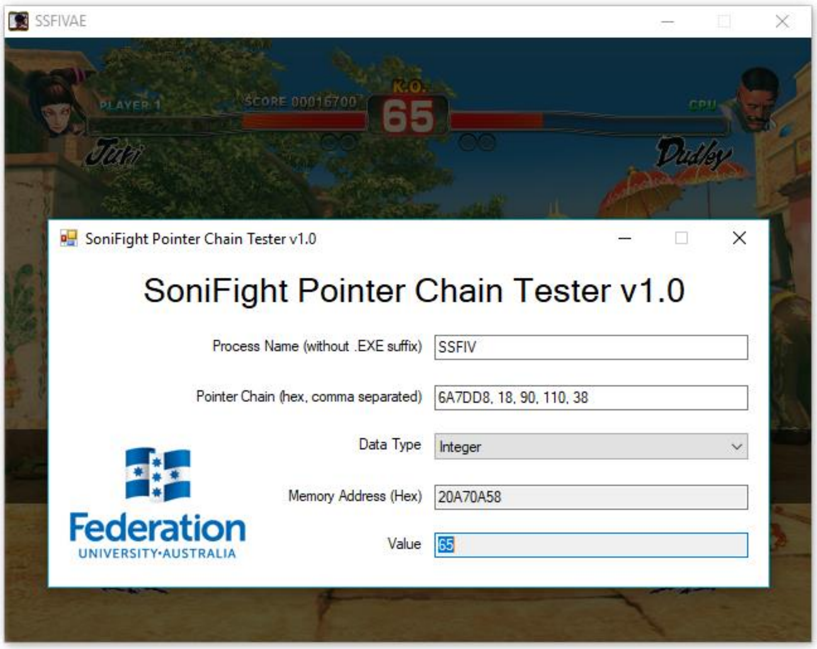
After that it'll show you the value at that memory address which is updated 10 times per second. This is an easy way to determine that a pointer chain in a watch is working, and to examine what possible values the watch may have while you do various things in the game.

So once you've found a pointer and want to be sure it's working, you could close the game application then re-launch it and start a match, then pause the match and launch the chain test app and feed it the above details to ensure everything's working as planned.

In the figure below, I've entered the street fighter 4 clock details and it shows the in-game clock and found value matching up identically with a value of 65 – so in this case I can be relatively sure that the pointer chain is correct and will successfully locate 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.

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

Figure 13 - The found pointer chain successfully locating the Street Fighter 4 clock which is showing a value of 65.



SoniFight Pointer Chain Tester v1.0

SoniFight Pointer Chain Tester v1.0

Process Name (without .EXE suffix) SSFIV

Pointer Chain (hex, comma separated) 6A7DD8, 18, 90, 110, 38

Data Type Integer

Memory Address (Hex) 20A70A58

Value 65

9 Frequently Asked Questions

9.1 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:
  - Have a 'good' (i.e. working) pointer chain,
  - Have the correct data type set so it reads the correct amount of data in the correct format, and
  - Be marked as active.

- The trigger using the watch must:
  - Use the correct watch ID for the data you're keeping track of,
  - Have a value and comparison type which is actually met by the watch,
  - Have a valid sample in the current config directory or have a compatible screen reader running if the trigger uses tolk-based sonification,
  - Have sane volume and playback speeds (1.0 for each would be fine),
  - Be marked with an allowance type of **In-Game**, **In-Menu** or **Any** as appropriate for what you want to happen, and
  - Be marked as active.

- The game configuration itself should also have a normal and continuous trigger master volume set at a value that will allow the trigger sample to be heard (again, 1.0 for each means 'full volume' and would be fine).

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

Also, if the config name or description contains a precise executable version number, then it could be worth checking to ensure that the version of the game executable you're running matches up with the version the config is designed to work with. To find the version number of an executable you can typically right-click on it and select **Properties** from the pop-up menu, and then go to the **Details** tab and look at the **File version** or **Product version** values.

Finally, while SoniFight is a windows forms application, it also creates a console window and writes some debug output it so you can see what's going on. If you wanted to write this debug output to file then you can simply launch a SoniFight executable from the command prompt and pipe the output to a file using a command like this: **SoniFight.exe > log.txt**

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

At present SoniFight only ships with a small number of configs developed as proof of concept. However, SoniFight was built to run configurations for various games with the idea being that users can create configs for any game they'd like 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 chains to values for use in new game configs for whatever game you might be interested in. Unfortunately, the process to find these pointer chains is likely to be 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 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 chain 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.

9.3 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 400 individual triggers, a large number of which use file-based samples for sonification) we still don’t go above around 70MB RAM usage.

9.4 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 all 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 4 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.

9.5 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 then 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 simply don't have the rights to distribute copyrighted audio.

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

Yup! It’s possible to create or modify configs 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).

9.7 I want to add additional triggers, is it a difficult process?

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 that would be it. That's the simple scenario.

If there isn't a watch for the specific value you want, then a pointer chain 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 chains are provided above in the ***Finding and Using Watches and Triggers*** section of this document. Once a pointer chain 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.

9.8 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 pain 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.

9.9 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 own **Configs** folder for it to be added to the list of available configs.

9.10 What platforms does SoniFight run on?

SoniFight runs on Windows 7 SP1 and above only. The SoniFight application itself comes in both 32-bit and 64-bit flavours. The 32-bit version can only connect to games which run as 32-bit processes, and the 64-bit version to games that run as 64-bit processes. SoniFight cannot run on consoles because the software would have to be signed by official parties (Microsoft, Sony etc) to run on console like an Xbox or Playstation – and even if it could run on consoles, suitable memory scanning software akin to Cheat Engine would need to be available to find the pointer chains for the console builds of any particular game.

9.11 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/irrklang/irrklang_pro.html

9.12 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

10 Appendix – Working with Multilanguage App Toolkit (MAT)

If you'd like to add an additional localisation language to SoniFight, then you'll need to do so via Microsoft's Multilanguage App Toolkit. To do so, open the SoniFight project in Visual Studio 2017. If you do not have the MAT installed then it can be freely obtained from the following URL:

https://marketplace.visualstudio.com/items?itemName=MultilingualAppToolkit.MultilingualAppToolkit-18308

Once installed, then to add a new language, right click on SoniFight's **Multilanguage Resources** folder and from the pop-up menu select "Multilanguage App Toolkit | Add translation languages" then add the language you want to localise for. Try to pick a more general language over a specific dialect where possible, for example "de" for German rather than the more specific "de-BE" for German (Belgian).

You should not add resource strings directly to ***Resources.fr.resx*** or such as they get automatically overwritten. Instead, open the localised resources file – for example **SoniFight.fr.xlf** with the **Multilingual Editor**, find the string you want to localise and add the correct translation to that.

Don't worry about properties of the form like size, width, location etc. being set for translation - the default translation is the exact same as the original values, and if you remove those strings or alter the **Form.resx** file then it either breaks the build or gets automatically regenerated to put them back – so it's best just to ignore these strings that you don't want to translate and only provide translations for the actual text fields you want to localise.

To test out the translation you can force the app to work in your desired language by importing the following references:

```
using System.Diagnostics;
using System.Globalization;
using System.Threading;
```

And then add the following code to the beginning of the Main constructor in **Program.cs**:

```
// Replace "fr" with your language of choice
CultureInfo cultureOverride = new CultureInfo("fr");
Thread.CurrentThread.CurrentUICulture = cultureOverride;
Thread.CurrentThread.CurrentCulture = cultureOverride;
```

Just as a note of warning: if you decide to manually modify the ***Form1.resx*** file (which is ill-advised as it's an automatically generated file), then MAT will break the file on next rebuild by inserting an additional `<?xml version="1.0" encoding="utf-8"?>` tag into one of the fields half way through the file. It will have the brackets HTML encoded to be things like `%lt;` - to fix this problem you'll have to manually remove that xml encoding string from the tag, which will allow you to clean and rebuild the project successfully.