Train Sim World
TARGET Profile for Thrustmaster Warthog

Introduction

This document explains how to use the included Thrustmaster TARGET script created for the HOTAS Warthog Throttle intended to work with Train Sim World (TSW). The script only uses the Warthog Throttle. The Warthog Joystick is not used, though adding mapping for the Joystick is easy if desired. The Warthog axes are mapped as follows:

- The Friction Control axis controls the reverser
- The Right Throttle axis controls the engine throttle or throttle/brake lever depending on the engine type.
- The Left Throttle axis controls the dynamic brake, automatic brake, and independent brake, depending on the setting of the FLAPS switch. It is easy to use all three modes on one lever due to a memory feature that remembers the position of each brake lever in TSW and makes sure you
- Supported engines:
    - SD40-2, GP38-2, GP40-2
    - GE AC4400CW
    - British Rail Class 43 HST
    - British Rail Class 166
    - British Rail Class 66
- Support for the Amtrak ACS-64 will be provided once Northeast Corridor New York is released.
- Support for the DB BR 442 Talent 2 EMU is not planned at this time as I don't have this DLC.
    - Contact me and if you can provide notch patterns, I can provide support with your testing help.
- All engines supported in a single script with quick profile changes using a button combination.
- Logitech/Saitek Throttle Quadrant profiles are also included to complement the Warthog by controlling the independent brake and automatic brake separately, allowing the Warthog Left axis to control the dynamic brake alone if desired. It is possible, however, to use just the Warthog Throttle to control all the engine levers. Use of these profiles is optional.
- TrackIR support is also offered using a FreePIE script. Use of this script is optional.

The TARGET script has several files included which are explained in the installation section. It is not necessary to understand how to program TARGET scripts to use this, but it will help a great deal if customizing the script is desired. This script is also a good example for showing what can be done with TARGET using more advanced techniques.

Installation

Installation is a manual process. Here is a list of the included files.

- *TARGET Script files*
  The Thrustmaster Warthog Throttle Script files. *Drakoz_TrainSimWorld.ttm* and *Drakoz_TrainSimWorld_Warthog.tmc*
- *Train Simi World – TARGET Script Doc.pdf*
  The document you are reading now.
- *TrainSimWorld_Generic_Warthog Controller Layout.xlsx*
  Excel file that shows the layout of all controls mapped in the TARGET script file. Also includes mapping for the Saitek Throttle Quadrant if you use those profiles.
- *wBeep files*
  The wBeep.exe program used in the TARGET script – for audible feedback when using the script.
- *Logitech-Saitek Throttle Quadrant Profiles*
  Profiles for the Logitech/Saitek Pro Flight Throttle Quadrant and Pro Flight Yoke (which has a Throttle Quadrant). It is optional if you want to use these.
- *TrackIR Support using FreePIE*
  A FreePIE python script and instructions to use FreePIE to get TrackIR support in TSW. See the README file for details on how to install it and use it. It is optional if you want to use this.

To install the TARGET script…

- Copy the *Drakoz_TrainSimWorld.ttm* and *Drakoz_TrainSimWorld_Warthog.tmc* files from "*Target Script files*" to where you normally store your TARGET script files. By default, this should be:
  c:\users\<your username>\AppData\Roaming\Thrustmaster\TARGET\Scripts
  The AppData folder is normally hidden. You can access it easily by hitting the Windows key and typing %appdata% <enter> (for Windows 7, 8, 10) and it will open the Appdata folder. Or it is OK to save the files anywhere as long as the TARGET Script editor can brose to them (see below).
- Copy wBeep.exe to C:\bin\wbeep.exe. You will likely have to create the C:\bin folder.
  - If you don't want to use or install wBeep.exe, see the *How to Disable wBeep.exe and the Beep() Function* section below.
- If you want to use the Logitech/Saitek profiles, see the README file in that directory for where to copy them. You only need one file or the other (the Pro Flight Yoke or Pro Flight Quadrant files) as each profile does the same thing. These profiles are easily modified through the Logitech Profile software.

To run the TARGET script…

- Make sure your Thrustmaster HOTAS Warthog Throttle is plugged in. The Warthog Joystick is not needed, but it does not hurt to have it plugged in. The same is true for any other Thrustmaster TARGET compatible controllers. This script will ignore them.
- Run the TARGET Script Editor (not the TARGET GUI).
- Select the TOOLS tab.
- Make sure FILE PATHS in TARGET points to where you copied the .tmc and .ttm files. If you copied the script files to a non-standard place, this is where you need to tell TARGET where they are.
  - Click on Options

    - Select “FILE PATHS” tab
    - Make sure the *Drakoz_TrainSimWorld.ttm* and *Drakoz_TrainSimWorld_Warthog.tmc* files are in a folder that is in the list of INCLUDE FILE LOCATIONS.
- Load the *Drakoz_TrainSimWorld_Warthog.tmc* file.
    - Click on MENU
    - Click on OPEN
    - Find the file “Drakoz_TrainSimWorld_Warthog.tmc” and open it.
    - The .ttm file will be loaded automatically for you when you run the script.
- Compile and run the script by clicking “RUN”.
- If all was installed correctly, it should compile with no errors, and you should hear a beep. The beep is part of the script using the wBeep.exe program. If you do not hear a beep, recheck that you have copied wBeep.exe to C:\bin (the script will not run without wBeep.exe unless you have disabled it) or check your audio settings. The beep sounds are part of the Windows “System Sounds” volume in the Windows Volume Mixer. Make sure the System Sounds volume is high enough (this should be true by default).
- If it fails to run for some other reason, it is probably a TARGET Script Editor setup thing, or you forgot to plug in your Warthog. Check the error log in the TARGET Script Editor, resolve the problem and try again. If all else fails, ask for help in the forum where you downloaded this.
- The script is now running and ready to be used in TSW. Please see the section “Using the TARGET Script below before continuing.

Once running, be careful about moving the axes on your Warthog Throttle when Train Sim World or the TARGET Script Editor are not the selected application as it will send lots of keystrokes to the computer with unintended consequences. To see what I mean, open up Notepad and move the axis back and forth. You’ll see a spew of characters show up on the screen.

To Install the Logitech Profiles (optional)

- See the README file in the *Logitech-Saitek Throttle Quadrant Profiles* folder.
- These profiles are only needed if you have and plan to use a Logitech Throttle Quadrant or the Throttle Quadrant that came with a Pro Flight Yoke. They are functionally the same, so I created a version of the profile for each as I have both. I only use one or the other, not both at the same time.

Using the TARGET Script – Basic Usage

- Run the TARGET Script.
- Select the desired engine profile - press and hold MSP and then press LDGH several times and it will cycle through the profiles.
- Load Train Sim World and select a scenario or service.
- Before entering the scenario, set the axes and switches on the Warthog to match the current state of the engine.
- See the included Excel file for button and axes configuration.

Please see below for detailed usage instructions.

Function Reference – *TrainSimWorld_Generic_Warthog Controller Layout.xlsx*

Open the *TrainSimWorld_Generic_Warthog Controller Layout.xlsx* file which has a layout of the controller and how the buttons, switches, and axes are mapped. Here are some notes on how to read this file.

- Each button and axis on the Throttle is listed in the gray boxes.
- **Black** text lists the **primary function** of a button (e.g. push the APENG button and it blows the horn).
- **Red** text is a **shifted function** controlled by the Thrustmaster IO shift button. The shift button is MSP (Mic Switch Hat center button). E.g. press MSP and the LDGH button, and it will change to the next engine profile (more on engine profiles below).
- **Orange** text is a **Press and Hold function**. E.g. Press and hold the Mic Switch HAT forward, and it will switch to Free View (view 8 in TSW).
- **Blue** text indicates **mode based functions** that change based on the position of the **Flaps Switch**. This is the Thrustmaster UMD mode switch.
- **Light Blue** text is **setup notes** or additional functional notes related to the sim.
- The file also includes mapping information for the Saitek Throttle Quadrant and the Logitech G700S mouse which I use. You can ignore the G700S settings as that is not part of this collection of files.

Changing the Engine Profile

To change profiles, press and hold the MSP button (you will hear a "bip" sound to confirm it was pressed). Then press the LDGH button. Each press of the LDGH button will change to the next profile. After reaching the last profile, it will switch back to the first profile. Text in the TARGET Script Editor window will identify which profile has been selected. The LEDs on the Warthog Throttle will display a binary number indicating the current profile number. Also, an increasing pitch sound will be heard with each selected profile. With the pitch sounds, or LEDs you can identify which profile you have switched to without leaving or alt-tabbing out of TSW.

The Profiles are (in order of selection):

Text in TARGET Window	Engines Supported	LED Pattern [1]
Profile 1: pro_SD40GP38	EMD SD40-2, GP38-2, GP40-2	LED 00001 = 1
Profile 2: pro_AC4400CW	GE AC4400CW	LED 00010 = 2
Profile 3: pro_BR43	British Rail Class 43 HST	LED 00011 = 3
Profile 4: pro_BR166	British Rail Class 166	LED 00100 = 4
Profile 5: pro_BR66	British Rail Class 66	LED 00101 = 5
Profile 6: pro_DBBR422	[2] DB BR 442 Talent 2 EMU	LED 00110 = 6
Profile 7: pro_ACS64	[2] Amtrak ACS-64	LED 00111 = 7

[1] A 1 means the LED is lit, a 0 means it is dark.
[2] Support for the Amtrak ACS-64 will be provided once Northeast Corridor New York is released. Support for the DB BR 442 Talent 2 EMU is not planned at this time as I don't have this DLC. Contact me and if you can provide lever notch patterns and key press times, I am happy to provide support with your testing assistance.

Using the TARGET Script – Detailed Usage

Following is a detailed description of how to use the TARGET script to control various engines.

Warthog Axis out of Sync with In-Game Lever

Occasionally, a Warthog axes will get out of sync with the engine lever in TSW. Great effort has been taken to avoid this, so it is possible to throw the Warthog axes back and forth quickly and have every axis movement accurately translate to moving the lever in game. This is the big difference between using this script and simply using key mapping features in other programming software like I did for the Logitech/Saitek profiles. For example, it should be possible to throw an axis full forward to full backward several times, and though there is a delay between moving the Warthog axes and the in game lever, the lever in game should eventually follow your every movement.

But again, this is only an axis to keypress mapping script, and occasionally a keypress is missed by TSW. This causes the Warthog axis to be out of sync with the sim. Sadly, it happens more often than I would like. I am working on ways to make it better, but it is difficult to overcome. See the Technical Details section for further explanation.

If you experience significant difficulties with the Warthog axes getting out of sync with the TSW levers, let me know. There may be issues with how timing on one computer compares to another, and they have to be adjusted for your computer.

When the Warthog axis and the in game lever get out of sync, either use the keyboard keys to move the in game lever to match the Warthog, or press ESC to pause the game, and move the Warthog axis to match the lever in game.

Lever Notch Feedback

The levers in TSW have notches for some lever positions. For example, the throttle lever on the SD40-2 has 9 notches, IDLE and 1 through 8, or the AC4400CW has a hump to overcome to move from throttle to dynamic braking. Sadly, the Warthog Throttle does not have these physical notches to give feedback (I have a solution for that using the afterburner notch piece, but it will be a while). Even the RailDriver product does not match these notches perfectly for every engine.

Feedback as you move the in game levers is given with HUD messages and the HUD compass in the lower right corner, but operating the engines with the HUD off, you lose this feedback. For some levers, a click can be heard in TSW that indicates lever movement. This is not consistent with all engines. Hence, the TARGET script uses various tones and beeps to indicate the most important notch points such as the hump or notch between throttle and braking on the AC4400CW or Class 166 throttle/brake lever, or important positions on the automatic brake lever such as Release, Handle Off and Emergency Brakes.

Because there are no physical notches on the Warthog, start out moving the Warthog axes slowly while getting used to the various feedback (HUD, in game clicks, and the TARGET Script beeps). This will help prevent over shooting the desired lever position in game.

Reverser (Warthog TFC Axis)

The Warthog Throttle Friction Control axis (TFC axis) is mapped to control the engine reverser lever for all engines.

If the Warthog TFC axis gets out of sync with the engine's reverser (which happens usually when starting a new scenario without first setting the TFC to match the reverser in game or when changing engine profiles) just move the TFC axis lever full forward, then full backward, then back to center, and the TFC axis will be in sync with TSW again.

For engines like the Class 43 HST and Class 166, there is an Engine Off position. The TFC axis is programmed to reach this position only in the last 2% of axis travel. Moving the TFC only 10% forward or 10% backward from center, however, will select forward and reverse. This should avoid accidentally moving to the Engine Off position.

Throttle Lever (Warthog Right Axis)

This applies to engines like the SD/GP engines, Class 43 HST, and Class 66. These engines use a dedicated throttle lever which has an IDLE position and several notched power positions (e.g. 1 through 8). For these engines the Warthog Right axis maps directly to the throttle lever on the engine.

Combined Throttle & Brake Lever

This applies to engines like the AC4400CW and Class 166. These engines have a combined throttle and brake lever with several notched throttle positions, a notch or hump in the middle, and several notched brake positions (e.g. Class 166), or a continuous (non-notched) brake zone (e.g. AC4400CW). The Warthog Right axis maps directly to the throttle/brake lever on these engines.

When using these engines, it is best to move the Right axis slowly so as to not overshoot the center detent or hump between throttle and braking. This is especially important for the AC4400CW as the center hump between throttle and braking requires a 1 second key press to overcome the hump. Hence it is easy to get confused as to where the Warthog lever is vs. the lever in TSW. There is some delay between moving the Warthog Right axis and the lever movement in the sim making it easy to get out of sync or overshoot the desired position. For the AC4400CW, extra beeps were used to help identify the movement over this hump. It is a good idea to practice moving back and forth over the hump to get used to the feel and the beeps.

Be aware that if the reverser on the AC4400CW is in neutral, TSW will not allow the throttle/brake lever to move into the braking region. Of course, the Warthog axis has no such limitation which gets things out of sync immediately.

Dynamic/Automatic/Independent Brake Lever Selection using the FLAPS Switch (Warthog Left Axis)

There are three types of brake levers in TSW, the dynamic brake, automatic brake, and independent brake. Not all engines have all three levers, and some engines combine one or more of these brake functions with the throttle lever (e.g. AC4400CW and Class 166). All three brake levers are controlled by the Warthog Throttle Left axis. You select which brake lever you want to control with the position of the FLAPS switch.

The Warthog Left axis is mapped as follows:

FLAPS Switch Position	**Left Axis Controls**
FLAPU (forward position)	Dynamic Brakes
FLAPM (middle position)	Automatic Brakes
FLAPD (backward position)	Independent Brakes

For engines that only have a single brake lever (e.g. Class 43 HST), the position of the FLAPS switch does not matter. For other engines, only those positions that match the engine are functional.

The TARGET script remembers the position of all three in game brake levers so when switching from one to another, the script will prevent the Left axis from moving the in game lever until it has been moved back to the previous position of the selected lever.

For example, in a GP38-2 going down a hill with the automatic and independent brakes set to release, and the dynamic brake set to position 7, say you are still gaining speed, but can't increase the dynamic brake more due to a brake warning (over current) light. Adding a little automatic braking will slow things down. To do this with the Warthog Left axis, flip the Flaps switch to FLAPM to select the automatic brakes and pull the Left axis all the way back. You will hear a beep indicating that the Left Axis is now in sync with the automatic brake lever in game. Apply a little automatic braking to slow down, then move the automatic brake lever back to release. Flip the Flaps switch back to FLAPU to select dynamic brakes and move the Left axis forward a little. Notice

the dynamic brake lever in game does not move. Keep pushing the lever forward until you hear a beep. Now the Left axis is in sync with the in game dynamic brake lever, and lever movement is enabled again.

For those that want separate lever controls, that is why the Logitech/Saitek Throttle Quadrant profiles were provided. They allow controlling the automatic (left lever) and independent (middle lever) brakes. When using the Throttle Quadrant, set the Warthog FLAPS switch to FLAPU to control the dynamic brake with the Warthog Left axis. I did not set up the Throttle Quadrant right lever to control the dynamic brakes because I always use the Warthog for dynamic brakes regardless. Or for the AC4400CW, I use the Warthog Left axis to control the automatic brakes, and the Saitek Throttle Quadrant only to control the independent brake.

Here are some specifics about each brake setting.

Dynamic Brake

This applies to engines like the SD40-2, GP38-2 and GP40-2. These engines use a dedicated dynamic brake lever which has OFF and SETUP notched positions and an un-notched continuous region (e.g. marked 1 through 8). When FLAPU is selected, the Warthog Left axis maps directly to the dynamic brake lever in game.

Automatic Brake

Almost all the engines in TSW have an automatic brake, or a brake that is mapped to work with the automatic brake keys in TSW (; and ‘). The automatic brake usually has several notched positions (e.g. Release, Initial Reduction, Suppression, Handle Off, and Emergency) as well as a continuous non-notched region between Initial Reduction and Suppression. When FLAPM is selected the Warthog Left axis maps directly to the automatic brake lever in game.

For the Class 66 locomotive, the automatic brake is a three position lever with Release, Hold, and Apply. The lever is spring loaded to default to the Hold position. The Warthog Left axis maps so that the lower 20% range is the Release setting, the 20-40% range is the Hold setting, and 40%-100% is the Apply setting. Of course, the Warthog is not spring loaded, so don’t forget to move the Left axis back to the Hold position.

For the Class 166 engine, the brake lever is just a simple notched lever. The Warthog Left axis maps directly to this lever in game and is chosen regardless of the Flaps switch position.

Independent Brake

Most engines have an independent brake. When FLAPD is selected the Warthog Left axis maps directly to the independent brake lever in game.

To actuate Bail Off, with the Warthog Left axis pulled all the way back, lift the Left axis up and pull back to the Engine OFF position (called IDLELOFF). As long as the axis is in IDLELOFF, the independent brake will be held in Bail Off. Push the Left axis forward to IDLE and Bail Off is released.

For the Class 66 locomotive, the independent brake is a three position lever with Release, Hold, and Apply. The Warthog Left axis maps so that the lower 20% range is the Release setting, the 20-40% range is the Hold setting, and 40%-100% is the Apply setting.

Views – Mic Switch

The Mic Switch is mapped to control TSW views and enable/disable HUD elements. See the Excel file for all mappings. Here are a few notes on the Mic Switch.

- MSR – In Cab View – TSW 1 key
- MSR Press and Hold – Free View – TSW 8 key
- MSL – Exterior View – TSW 3 key
- MSL Press and Hold – Boom View – TSW 2 key
- MSD – Toggle HUD Markers toggles the 3 icon markers that show next way point, next speed zone, and next signal.
- MSD Press and Hold – HUD Toggle turns all HUD elements on or off (TSW CTRL-1, CTRL-2, CTRL-3)
- MSU – 2D Map View – TSW 9 key
- MSU Press and Hold – Score Toggle (CTRL-6) toggles display of the Score HUD element, but because of what I believe is a bug in TSW, it also toggles the next signal and next speed zone HUD elements. These elements have their own toggle key (CTRL-4) but using them seems to have problems with them not working (a bug). So I just use Score Toggle.

Using the Mic Switch for views, the Coolie Switch as the arrow keys (see below), and some mappings on my Logitech G700S Mouse (see the Excel file) allows me to control most external views and First Person activities without touching the keyboard.

Arrow Keys – Coolie Switch

I normally have my left hand on the Warthog Throttle and right hand on the mouse. I mapped the following Warthog buttons so I don't have to use the keyboard arrow keys to move around. I can quickly switch views using the Mic Switch, and then fly around.

The keyboard arrow keys are mapped to the Coolie HAT switch. This controls the UP, DOWN, LEFT, and RIGHT arrow keys for switching views in the cab or moving around in external views.

MSP + the Left or Right Coolie buttons (CSL and CSR), does a CTRL-LEFT ARROW or CTRL-RIGHT ARROW which is used in external views.

The Slew Control center button (SC) maps to the keyboard SHIFT key. When SHIFT is held with an arrow key, you move faster in external views. This also works with First Person walk around mode.

Headlight Switches

TSW has a forward headlight switch (keyboard hot key h and SHIFT-h) and a backward headlight switch (keyboard hot key CTRL-h and SHIFT-CTRL-h). These switches work differently for different engines. The Warthog Engine Operate Switches are mapped to control these switches. See the Excel cheat sheet for details. Try them to understand what they are doing. One press or toggle of the switch forward or backward will move the respective headlight switch forward or backward one position.

Wiper Switch

The wipers work differently for different engines.

For the GP/SD engines, press and hold v, and the wipers are turned on or increase in speed. Press and hold SHIFT-v and the wipers are decreased in speed or turned off. The TARGET script maps this to a single toggle switch on the Warthog, the Engine Flow R toggle switch (EFRNORM and EFROVER). It is programmed to simply turn the wipers on or off. For example, on the SD/GP engines, it defaults to 50% wiper speed.

For the Class 43 HST, the wiper switch has 4 positions. The Warthog switch, again, simply turns the wipers on, or off. But it will move the Class 43 wiper switch from ON to OFF to PARK, wait a moment to park the wipers, and then back to OFF.

Other Switches and Buttons

All other switches and functions in TSW should be easy to figure out. See the Excel cheat sheet for details. If you want to change the mapping of some switches and buttons, see the Technical Details section below for a few important things to be aware of before modifying the TARGET Script.

How to Disable wBeep.exe and the Beep() Function

If you don't want to use wBeep.exe or get audible feedback from the scripts, make the following change. This variable is found near the top of the *Drakoz_TrainSimWorld_Warthog.tmc* file. When disabled, it is not necessary to copy wBeep.exe to C:\bin on your computer. I strongly recommend trying the beeps first before deciding to disable it. Because there are no detents on the Thrustmaster Throttle Left and Right axis, the beeps are a solution to give feedback. To understand more about wBeep.exe and what it is doing, see the section titled *Beep() Function – Calling wBeep.exe* in the *Technical Details* section.

Default setting – this enables beeps:

```
// Disable Beep() output - disabled =0, enabled =1
define     BeepEnabled          1
```

To disable beeps, make the following change:

```
// Disable Beep() output - disabled =0, enabled =1
define     BeepEnabled          0
```

Using the Logitech/Saitek Throttle Quadrant Profiles

Refer to the Excel spreadsheet to see the mapping for the Saitek Throttle Quadrants. Although you can control all the brake levers with the Warthog Left axis, it is much nicer to have a Throttle Quadrant set up so you have three separate levers to control the brakes.

The Saitek left and middle (X and Y) axes were configured to use Directional Axis mode. This mode is very simple. Move the lever up and it presses one key. Move the lever down and it presses another key. The right (Z) axis is not used.

When moving the Throttle Quadrant levers, move them slowly. Unlike the Warthog that guarantees that every key press will be sent to TSW, which keeps things in sync and gives great precision, the Saitek

programming software will skip keypresses if you move the lever too quickly and fall short when you reach the end of travel. It works OK for the automatic brake and independent brake, but this way of doing keypresses simply does not work for the throttle or throttle/brake lever in TSW.

The Throttle Quadrant, however, has a nice feature missing on the Warthog – the Reverser position at the bottom of the lever throw. The Warthog has a similar position, but the Throttle Quadrant's position has a physical notch you have to overcome to use it which makes it perfect for the automatic and independent brakes.

Automatic Brake using the Saitek Throttle Quadrant

The Throttle Quadrant's left lever is mapped to move the engine's automatic brake lever, but it uses only short keypresses. It is not intended to overcome the notched areas that require long presses. Instead, the T1 and T2 buttons perform a long press to get past any notches.

When the Left lever is pulled back to the Saitek's Reverser position, it does two long presses which will move the automatic brake from Service to Minimum Reduction to Release. Then when moved out of the Reverser position, two long presses are made again, which moves from Release to Service.

To move to the Minimum Reduction, Handle Off, or Emergency positions, press the T1 or T2 buttons as needed to create the long presses.

Independent Brake using the Saitek Throttle Quadrant

The Throttle Quadrant's middle lever is mapped to move the engine's independent brake lever. It only does short presses as long presses are not needed.

To actuate Bail Out, press the middle lever down into the Saitek's Reverser position. To release Bail Out, pull the lever out of the reverser position.

Technical Details

This section assumes you have some familiarity with TARGET scripting (not the TARGET GUI, but the TARGET Scripting Editor). It also helps to understand C Programming at least a little. If you don't have such experience, don't be afraid to dive in. That's how you learn. But you'll want to keep the TARGET Scripting Documentation handy to understand things. TARGET is a C like language. It has a lot of power, but lacks a few basic features of C. For the average TARGET Script user, however, Thrustmaster has simplified the language down to relatively simple MapKey(), MapAxis() and KeyAxis() commands.

The information below was written before my final release of the script, so some details may not match the actual script. Refer to my actual TARGET scripts to see exactly what I did. But the information below is still correct from a learning perspective.

To follow along, open up the .tmc and .ttm TARGET scripts for Train Sim World in the TARGET Script. Or use a program like Notepad++ and tell it that .tmc and .ttm files are C code to add proper coloring. Notepad++ is a much better text editor than using the TARGET Script editor or Windows Notepad. You can edit the files in Notepad++ and load them in the TARGET Script Editor at the same time. Make edits in Notepad++, save the file, and press the RUN button in the TARGET Script Editor to compile and run file. It will always compile and run the saved file even if the open file in TARGET doesn't match anymore. Just never save the file in the TARGET Script Editor. That will over write the saved copy from Notepad++.

Editing the TARGET Script to Change Button/Switch Assignments

If all you care about is changing a few of the button assignments, you only need to understand basic TARGET Scripts functions like MapKey(), but also be aware of the following that is specific to my script.

A normal TARGET script has a main() section where the user implements their MapKey() statements, and the EventHandle() routine that is just there and not meant to be touched. In the case of my script, the normal MapKey() statements are implemented in a separate function called CommonConfig(). Any changes to key mappings should be done there. CommonConfig() is called by the pro_CallCustomConfigs() function which is called at the bottom of main(). So it is really the same as if it was in main().

To implement key mappings specific to one engine, add those changes to the engine specific pro_Configure<engine_name>() function. For example, for the SD40-2, GP38-2, and GP40-2 engines, add custom key mapping in the pro_ConfigureSD40GP38() function. The engine specific functions are called as you cycle through the profiles.

Any mapping in the engine specific functions will override any previous mapping. For example, EACON is mapped to blow the horn for all engines in the CommonConfig() function. If you want to change EACON to turn on the cab lights only for one particular engine, add a MapKey() function to the engine specific function and it will replace the mapping to blow the horn only for that engine, even though it was previously mapped in CommonConfig(). When switching the profile back to another engine, EACON will go back to blowing the horn as CommonConfig() is executed again every time an engine profile change occurs.

Changing the IO Shift and UMD Mode Buttons

The exception to all this is the MSP button must be mapped in main() and nowhere else because it is the TARGET IO Shift button. If you want to use a different key for the IO Shift key, change both the following two lines:

```
// IO Shift and UMD Setup (%DEV1, I button, $DEV, U, D, Toggle settings)
SetShiftButton(&Throttle, MSP, &Throttle, FLAPU, FLAPD, 0);
```

and

```
// MSP is the IO Shift Key.  Only map it here. Do no map in custom profiles or CommonConfig)()
//   Doing so will break rotating through profiles.
//   Makes a bip sound when pressing MSP
MapKey(&Throttle, MSP, EXEC("Beep(8000,30);"));
```

To change the UMD Mode Switch, modify the SetShiftButton() function shown above, but also make sure to update the following MapKey() functions (found in CommonConfig()) to point to your new chosen mode buttons (e.g. replace FLAPU, FLAPM, and FLAPD with your chosen switch positions):

```
// Flaps Switch
// UMD Switch - also used to select function of Warthog Left Throttle Axis with
KeyAxisDirectional()
MapKeyIO(&Throttle, FLAPU, 0, EXEC("SwitchUMD(SWITCHFORWARD);"));
MapKeyIO(&Throttle, FLAPM, 0, EXEC("SwitchUMD(SWITCHMIDDLE);"));
MapKeyIO(&Throttle, FLAPD, 0, EXEC("SwitchUMD(SWITCHBACK);"));
```

The above EXEC() statements call the function SwitchUMD() which important for allowing Left axis to control different engine levers and keep them in sync when switching modes.

Long Keypresses required by TSW

Many of the keyboard keys used to control TSW require a longer than average press, and the length of that press is different for different locomotives. For example, controlling the headlight switch (h) requires a 200-300 ms key press for some engines and a 300-400 ms keypress for other engines.

Normally, in TARGET scripts, you map a key with MapKey() as follows:

```
MapKey(&Throttle, LTB, ‘h’);
```

When the LTB button is pressed, the h key will be pressed and held as long as you press and hold the LTB button. Often, the PULSE+ modifier is used as follows:

```
MapKey(&Throttle, LTB, PULSE+’h’);
```

When LTB is pressed, no matter how long you press and hold LTB, the h key is pressed only for a moment. It is pulsed for a period of milliseconds defined by the default time for a PULSE. This is set by the SetKBRate() function, or 25ms, which I believe is the default if there is no SetKBRate() function.

The first method above works for TSW but is sometimes inconvenient as you must always remember to press and hold the button long enough to have the desired result. Some functions in TSW require such a long press

that I often “missed’ getting the button or lever to move because I didn’t press the button long enough. So, I tend to program using the PULSE+ modifier. The problem with the PULSE+ modifier is there is only one default delay for all PULSE+ key presses. But as pointed out, some keys require a 100 or 200 ms press, while others require a 300 or 400 ms press.

The solution is to use a CHAIN as follows:

```
MapKey(&Throttle, LTB, CHAIN(LOCK+DOWN+'h', D(250), UP+'h', D(50), LOCK);
```

Using DOWN+ and UP+ in a chain command like this allows us to set a specific delay for each button. DOWN+ presses the h key down. UP+ releases the h key. The delay between pressing and releasing h is set by D(250) in the middle which causes a 250ms delay. The LOCK+ modifier makes sure that the entire string of commands in the CHAIN() are executed without interruption by additional presses of the LTB button. The LOCK at the end unlocks it to allow other LTB presses to fire off a new pressing of the h key. The D(50) at the end is just to make sure that if you do multiple presses of the button, there is a slight delay between them. This is because when using LOCK, if you press the LTB button 10 times rapidly (faster than the D(250)), TARGET will queue up all 10 presses of the button and spit them out one after the other. Without LOCK, they would be sent coincidentally, which would cause chaos. But you also need TSW to register the button as being unpressed, and a small delay between each button press is required for that. Hence the D(50).

So if you desire to modify the MapKey() functions to fit your tastes, keep the above in mind as you may have to use the CHAIN(..DOWN+…UP+) version of a MapKey() to get the key to work correctly.

Changing the Lever Mapping and KeyPress Delays for Each Engine

This topic is too big to explain in detail, but I’ll make a “few” comments and then direct the reader to follow the existing script as an example.

Engine Brake and Lever controls are mapped using the CallKeyAxisDirectional() function. This function simply calls KeyAxisDirectional() with the proper data tables specific to each engine. The data tables are configured in the .ttm file.

CallKeyAxisDirectional() looks surprisingly similar to EventHandle(). That is because CalKeyAxisDirectional() is called by EventHandle(). Every time a button is pressed, or a switch is flipped, or an axis moves on the Warthog, the TARGET service calls EventHandle(). This is our backdoor to write custom code to take over. When we are done, EventHandle() then calls DefaultMapping() to continue handling the event normally.

EventHandle() takes in three parameters, type, o, and x. Parameter o is the device (e.g. &Throttle). Parameter x is the button or axis that made the event (e.g. MSP for the Mic center push button, or THR_RIGHT for the Warthog Right Throttle axis). Parameter type is not used. In EventHandle(), CallKeyAxisDirectional() is called with the same o and x parameters. CallKeyAxisDirectional() then determines which engine profile is active, and which axis was moved, and sends the data to KeyAxisDirectional() which performs the actual key presses to make the TSW engine levers move. So to be clear, KeyAxisDirectional() is really the main point of all this. Everything else wrapped around it is just the fluff to make it work conveniently (customizable profiles, profile switching, formatted tables that are easy to edit, etc.).

KeyAxisDirectional() performs the equivalent of an TARGET AXMAP1() and AXMAP2() function at the same time. Similar to AXMAP1, KeyAxisDirectional() presses one key when an axis moves up, and another key when the axis moves down. If LIST() is used with AXMAP1(), several zones of different size can be configured so the

key press occurs when crossing the oddly sized zones. Similar to AXMAP2(), KeyAxisDirectional() can press a different key depending on what zone the axis is in. Hence KeyAxisDirectional() can send one set of keys based on zones when moving up, and another set of keys based on zones when moving down. It is designed to use the same zones for axis up and axis down. Though it wouldn't be difficult to modify KeyAxisDirectional() to allow different zones for axis up vs. axis down, such a feature is not needed for TSW.

For AXMAP1() and AXMAP2(), these zones are defined by LIST(), followed by a comma separated list of key press events for each zone. For KeyAxisDirectional(), the zones and the key presses are defined by the tables in the .ttm file, but it is the same basic concept.

Finally, the tables for KeyAxisDirectional() have a pair of columns to specify a frequency and duration for a tone or beep so that beeps can be heard at specific zones to give feedback to the user.

Here is the definition of KeyAxisDirectional():

```
int KeyAxisDirectional(alias dev, int axis, alias zone, alias zonesList, alias actionUP, alias actionDN)
```

An explanation of each parameter:

- dev – Device (e.g. &Throttle). dev is an alias for Throttle, Joystick, HCougar, etc. The variable is passed into KeyAxisDirectional() using the "address of" notation &. Hence &Throttle means the address of Throttle. Throttle is an array of int that contains all the current button and axis states for the Warthog Throttle. For example, Throttle[MSP] equals the current state of the MSP button. The & notation is used throughout TARGET scripts to pass global variables around and allow functions to edit those variables without knowing what they are editing. Normally in C Programming this is done with pointer notation (*), but TARGET does not support pointers. It only supports &.
- axis – This is the axis from dev (e.g. THR_RIGHT) that moved and must be processed by KeyAxisDirectional().
- zone – This is an alias to the current zone for the axis. Zone is a number between 1 and the number of zones defined by zoneList. If zone = 0, the axis hasn't moved since we started the script (there is no current zone yet).
- zonesList - alias to array of int listing zone boundaries by percent (e.g. {0,250,500,750,1000} = 0%, 25%, 50%, 750%, 100%). Whereas LIST() only supports whole numbers (e.g. 25%), KeyAxisDirectional() supports fractions of a percent (e.g. 25.5%). But integers are used (e.g. 250) to avoid using floating point math and keep things efficient in the middle of the EventHandle() routine. The number of zones defined is 1 less than the number of elements in zonesList. E.g. zonesList = {0,250,500,750,1000} is 5 elements, but there are only 4 zones as the zones are between the numbers, not on the numbers. This is the same as LIST().
- actionUP, actionDN - alias to an array of type "struct axisAction" which contains actions and beeps for each zone. Each element of the array applies to a zone defined by zonesList and contains the data action, delay, freq, and dur. The action value is the key to press (e.g. 'h'), delay is how long to hold it down in ms, freq and dur are the frequency (in Hz) and duration (in ms) of a beep for that zone. actionUP is for increasing axis values (the axis moved up), actionDN for decreasing axis values (the axis moved down). If action is 0, no action is performed (no key press). If delay is -1, it means press and hold the action. If delay is 0, it means release the action (be careful with this – always make sure there is a path to release the key!). If dur is 0, no beep is performed. The value of freq doesn't matter if dur is 0.

The parameters dev and axis are passed in directly from EventHandle(). The parameters zone, zonesList, actionUP, and actionDN are defined on a per engine profile basis using the tables in the .ttm file. CallAxisDirectional() figures out which engine profile is the current profile and which axis moved, and passes all this to KeyAxisDirectional() which then performs key presses without having any knowledge of what profile is current, or what axis just moved. It only cares about the current position of the axis (newzone) relative the previous position of the axis (zone), determines if the axis crossed a zone barrier as determined by zonesList , and if so, which direction the axis moved. KeyAxisDirectional() then performs the appropriate actionUP or acitonDN key presses (and beeps) to move the lever in TSW.

The parameter zone (which is called the previous zone) is modified by KeyAxisDirectional() to contain the new zone (newzone) that the axis moved to (if it moved zones at all). All other parameters are not changed by KeyAxisDirectional().

Looking at KeyAxisDirectional(), you'll see that it does the following in this order:

- Determines what zone the axis is currently in (newzone).
- If the previous zone = 0, set zone = newzone and exit. There was no previous zone. First time through the function for this axis/engine combo.
- If the zone didn't change, exit.
    - But first check UMDJoySynced[UMDMode] to see if the axis has now synced. If so, beep and clear the UMDJoySynced[UMDMode] flag – enabling the axis to move the levers in TSW.
- If the zone changed, check UMDJoySynced[UMDMode] to see if the axis is not synced. If not synced, exit – perform no action.
- Determine if Axis moved UP or Down and perform the actionUP or actionDN action
    - Beep according to the action table
    - Perform the action (key press)
- Set zone = newzone and exit

UMDJoySynced[] is an array of 3 elements, one for each mode of the UMD switch (e.g. FLAPU, FLAPM, FLAPD) and hence one for each engine brake lever in TSW (dynamic, automatic, independent). When the UMD mode switch is changed, all elements of UMDJoySynced[] are set to 0 indicating that all 3 brake levers are out of sync with the Warthog Left axis. The next time the Left axis is moved, KeyAxisDirectional() determines if the axis is in the current zone for the brake lever in TSW. If not, then KeyAxisDirectional() ignores the axis movement and exits. If true, then UMDJoySynced[UMDMode] is set to 1 and axis movement is now allowed to move the engine brake lever in TSW. UMDMode is simply the current brake mode as determined by the Flaps switch.

Sometimes it is possible to move the Warthog Left axis faster than the script can keep up. You really have to throw the Warthog lever fast to do this, or it happens due to a Windows interruption. KeyAxisDirectional() figures this out and makes sure that the proper action is performed for every zone that was crossed. No actions or key presses are missed due to moving the Warthog too quickly. This is one of the key problems using the common key mapping software for other game controllers. The Saitek Throttle Quadrant is nice to have, but if you move the lever too fast, the Saitek software starts missing zones and sends too few key presses, causing the Saitek lever to get out of sync with TSW. Or it will mash the key presses together and send them at the same time - chaos. This TARGET script prevents that.

Again, it isn't perfect. There are still incidents where TSW seems to miss a key press. This is related to the time delay for each key press. TARGET is very good at guaranteeing the D() delays within the bounds that

Windows allows. But the TARGET service cannot make it perfect, resulting in some key presses being too short and TSW fails to register them.

There is a range of delay with TSW where too short, and the key doesn't register, too long and the key will register twice and move a lever two steps. For example, for the a key to increase the throttle, for some engines the delay is between 60 and 210 ms. I will choose a delay on the high end of that spectrum (e.g. 170) in the hopes that if an interruption occurs, it will still meet the 60 ms window. But choose too close to the 210 ms value, and TSW might interpret the key press as a long key press and move the throttle two steps. The entire reason I had to write all this scripting was to overcome this issue. It is variable from one lever/button to the next and even from one engine to the next for the same lever/button control. Why Dovetail did it this way is beyond me. No other simulator/game I mess with has this problem. But it is nothing new. Trains Simulator 20xx has the same issue. Which means that this TARGET script could be updated to work with Train Simulator also, but there is already a good mod for Train Simulator to handle all this far more elegantly and for all game controllers, not just Thrustmaster. The point is, this TARGET script is capable of doing this for any train simulator (RUN 8, TS 20xx, etc.) using the same techniques.

There is a lot more that could be said about the tables in the .ttm file and why they were done the way they were done. There is no great insight I can depart here. It was just hours of time spent figuring out the required key delays for each lever on each engine and finding the right combination of number of zones and keypress delays to get the result I wanted. Now that the groundwork is done, it is very easy to add new engines as they come out.

Engine Profiles and Too Many Engines

Right now, there are only a few engine profiles, so cycling through them with a button press, beeps, and LEDs for feedback is OK. But as more engines are released by Dovetail Games, it will be ridiculous pressing the LDGH button a couple dozen times to get to your engine. And who really wants to read binary code on an LED display (well, actually, if you read this far, you are probably someone that doesn't mind binary). Anyway, this will be addressed in the future either by splitting up the script into multiple scripts based on engine type and having to run each script manually in TARGET, or by asking for user input through the TARGET Script output window, which I believe is possible.

The real issue is, I don't plan to buy every DLC for TSW, but I can help define the engine tables in the .ttm file with input from other users. That is really the reason I took the time to write this document – so that people can help define the engine profile tables for DLC that I don't own.

Beep() Function – Calling wBeep.exe

Beep() is a function that simply calls wBeep.exe using the system() command. Here is the coding for Beep():

```
//*************************************************************
// Beep()
//  Beep function - calls wbeep.exe which must be located at the path below.
int Beep(int freq, int duration)            // Beep(frequency, duration in ms)
{
    char buffer;Dim(&buffer, 64);
    sprintf (&buffer, "spawn c:\\bin\\wbeep.exe %d %d", freq, duration);
    system(&buffer);
}
```

The sprintf() function "builds" the command to be sent to the Windows command prompt to call wBeep.exe. The command it executes is:

```
spawn c:\bin\wbeep.exe %d %d
```

Where the %d %d is the frequency and duration of the beep, and spawn is just a command to run the program as its own process. To use the system() function correctly with a Windows command prompt pathname, it is required to escape the \ with a \\. If the location for wBeep.exe is changed, the path should be modified in the Beep() function, remembering the \\ in place of \.

See the readme file in the "*wBeep files*" directory for more details including the source code of the wBeep.exe program and how to make and compile it yourself. The program literally just passes the frequency and duration to the Windows Beep(f, d) system call using 1 line of code.

I have considered upgrading to something that can play .wav or .mp3 files to allow for a variety of custom sounds. Other people have done this with TARGET Scripts using the same technique used for wBeep.exe. See this forum topic at SimHQ.com for further details on wBeep.exe and other ideas.

http://simhq.com/forum/ubbthreads.php/topics/3852299

Conclusion on the Technical Details

Most of the rest of the TSW TARGET script is using common methods that are clearly explained in the TARGET Script Editor Basics manual.

Support

I am posting these files in the following locations. Any questions/comments, I would prefer you post to the UKTrainSim forums as it is a much older and well established forum, but I will check the Dovetail Games forums as well of course.

UKTrainSim Forums:
http://forums.uktrainsim.com/viewtopic.php?f=386&t=149086

DoveTail Games TSW forums:
https://forums.dovetailgames.com/threads/thurstmaster-target-script-for-warthog-throttle-saitek-tq-profile.3634

Or if all else fails, email me…
Michael Lohmeyer
mike@akhara.com

Other Works and Links of Interest

The UKTrainSim forum is where I found the mods by CobraOne and Havner and older works by other people (see the Links below). Without their work, I would never have considered Train Simulator 20xx anything more than a novelty. I took a gamble and bought TSW hoping I could make TARGET do what I need. Sadly, extending this capability to other game controllers is a much bigger task. I would suggest starting with AutoHotKeys as it can do it, but also tell Dovetail that you want game controller support. My TSW TARGET Script has nothing to do with CobraOne and Havner's work (no code in common), but I was inspired by their desire to solve the problem.

TS2015 Raildriver Interface

CobraOne's mod – a plugin/mod for Train Simulator 20xx that provides better RailDriver support than the actual RailDriver software, as well as support for all DirectX game controllers including axis and button mapping. Also includes an overlay that provides useful information for people that want to turn off the TS 20xx HUD information at the bottom of the screen.
http://forums.uktrainsim.com/viewtopic.php?f=361&t=139830

TrainSim Helper (Joystick/Overlay) release thread

Havner's mod – a plugin/mod for Train Simulator 20xx that provides support for all DirectX game controllers but including only axis mapping to TS 20xx. Button mapping is not provided, nor is the enhanced RailDriver support provided. Also provides the overlay feature like in CobraOne's mod. In fact, the overlay started here and CobraOne adopted it into his mod. Havner and CobraOne have collaborated a great deal on these two items and deserve huge recognition for their efforts.
http://forums.uktrainsim.com/viewtopic.php?f=361&t=139304