Thrustmaster T300RS Force-Feedback

Contents

Requirements

- Visual Studio 2019 (or higher, with C++ for windows apps, windows 10 SDK)
  - Library only is functional when a Visual Studio installation is on the computer.
- Windows version, at least 10.0.17763 Build 17763
- MATLAB R2018b
  - The racing wheel works perfectly fine with newer versions
  - R2018b is needed for Trackmaker
  - Parallel Toolbox required for Trackmaker
- Trackmaker for Simulink 7.0.3
- DevConnAT repository
  - For truckmaker simulation
  - Branch Force-Feedback
- Racing-Wheel with Force-Feedback functionality
  - Only tried it with Thrustmaster T300RS

UWP/Windows Gaming API (Visual Studio)

The library to interface with the racing-wheel with force-feedback is based on the Windows Runtime API *Windows.Gaming.Input*. References can be found at:

- Guide for gamepad implementation: https://docs.microsoft.com/en-us/windows/uwp/gaming/input-practices-for-games

- Reference for Racing-Wheel and Force-Feedback: https://docs.microsoft.com/en-us/windows/uwp/gaming/racing-wheel-and-force-feedback
- Windows.Gaming.Input Namespace: https://docs.microsoft.com/en-us/uwp/api/Windows.Gaming.Input?view=winrt-19041
- RacingWheel Class: https://docs.microsoft.com/en-us/uwp/api/windows.gaming.input.racingwheel?view=winrt-19041

The dynamic link library is coded in Visual Studio 2019. When installing Visual Studio, check the box to install the Windows 10 SDK as well.

Understanding and building the .dll with Visual Studio

Open the corresponding Visual Studio Solution in the folder FF_UWP_WIN32_dll. The following project-folder in Figure 1 should be visible.

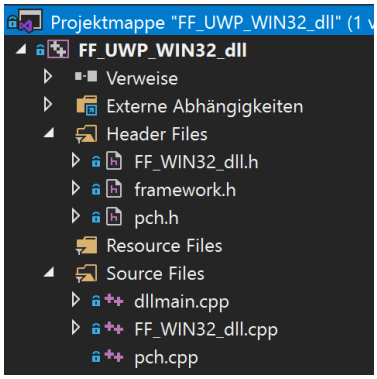

Figure 1 - Visual Studio project folder

```
// Initialize the racing wheel
FF_UWP_API bool initRacingWheel();
FF_UWP_API int initForceFeedback();
FF_UWP_API void readWheelStatus(WheelReadings* wheelValues);
FF_UWP_API void FF_minus(double gain);
FF_UWP_API void FF_plus(double gain);
FF_UWP_API void FF_zero();
FF_UWP_API void readingButton(buttonReadings* bValues);
FF_UWP_API void kill_FF();
```

Figure 2 - The library features eight functions. Two init-functions, two functions for reading the wheel, three for applying force-feedback and one to shut-down the force-feedback.

In *FF_WIN32_dll.cpp* the functions are coded, whereas in *FF_WIN32_dll.h* the function callups are written for the library. Note, that extern "C" functions are defined, therefore the library is callable as a C-Library. Structure definitions for Buttons and Paddles are defined in the header file.

The .dll features eight functions which are given in Figure 2:

- **bool initRacingWheel():** Builds the connection to the racing-wheel.
- **int initForceFeedback():** Loads the two force-feedback-effects (FF_minus, FF_plus) onto the wheel. The effects are initialized with gain zero (no torque). The runtime of the effects are limited to be effective for 27.777h.
- **void readWheelStatus(WheelReadings* wheelValues):** Reads the following values from the wheel
  - `double throttle;`
  - `double brake;`
  - `double clutch;`
  - `double angle;`
  - `double mastergain;`
  - `unsigned long long timestamp;`
- **void readingButton(buttonReadings* bValues):** Reads the following values from the wheel:
  - `bool gearup;`
  - `bool geardown;`
  - `bool ST;`
  - `bool SE;`
  - `bool X;`
  - `bool O;`
  - `bool Square;`

    - `bool Triangle;`
    - `bool L2;`
    - `bool R2;`
    - `bool L3;`
    - `bool R3;`
    - `bool DPadDown;`
    - `bool DPadUp;`
    - `bool DPadLeft;`
    - `bool DPadRight;`
- **void FF_plus(double gain):** Applies a constant force effect with the given gain. Note, the gain is a double value in the range of 0 to 1.
- **void FF_minus(double gain):** Applies a constant force effect with the given gain. Note, the gain is a double value in the range of 0 to 1. Note, the effect applies a torque in the opposite direction to FF_plus.
- **void FF_zero():** Sets the force effect to zero. Gain of FF_plus and FF_minus is set to 0.
- **void kill_FF():** Stops all effects on the wheel.

Setting up Visual Studio for building the .dll

If the project-solution does not already have to following settings, please add them in the property pages:

- General → Windows SDK Version
- C/C++ → General → Additional #using Directories
- C/C++ → General → Consume Windows Runtime Extension
- C/C++ → Preprocessor → Preprocessor Definitions
- C/C++ → Command Line → Additional Options

For the necessary changes to be applied, see the figures attached below.

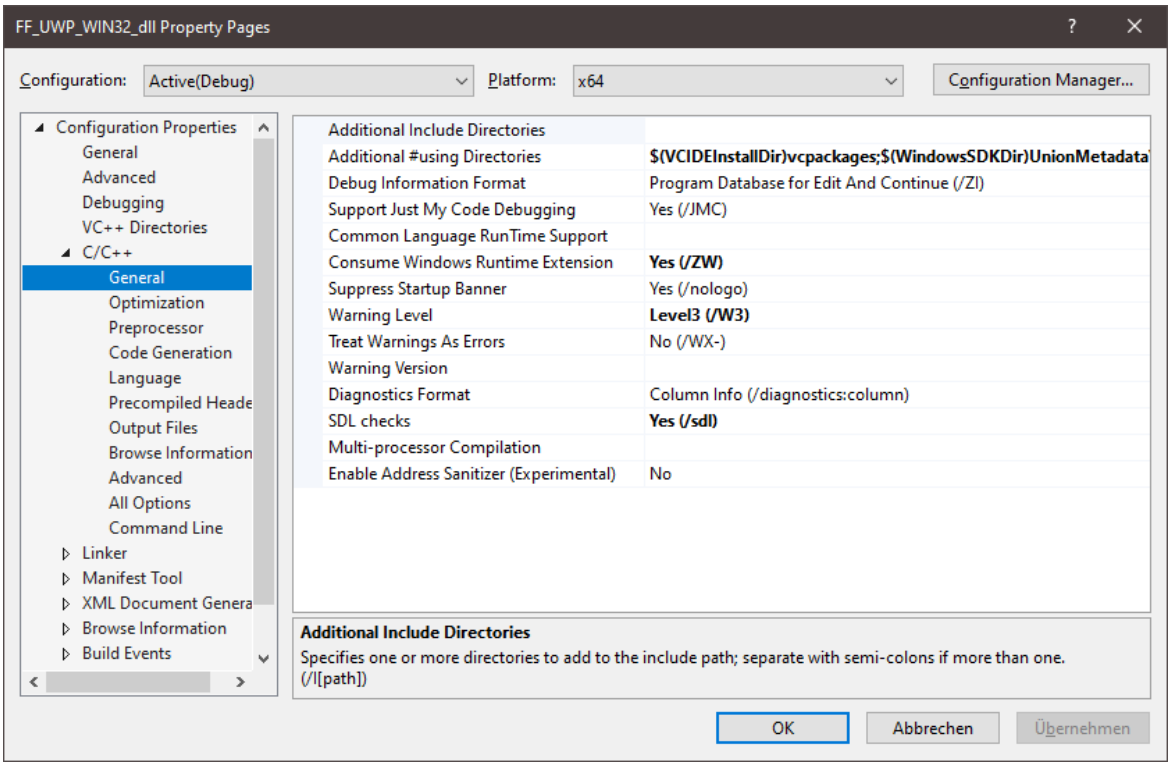

Figure 3 - C/C++ --> General

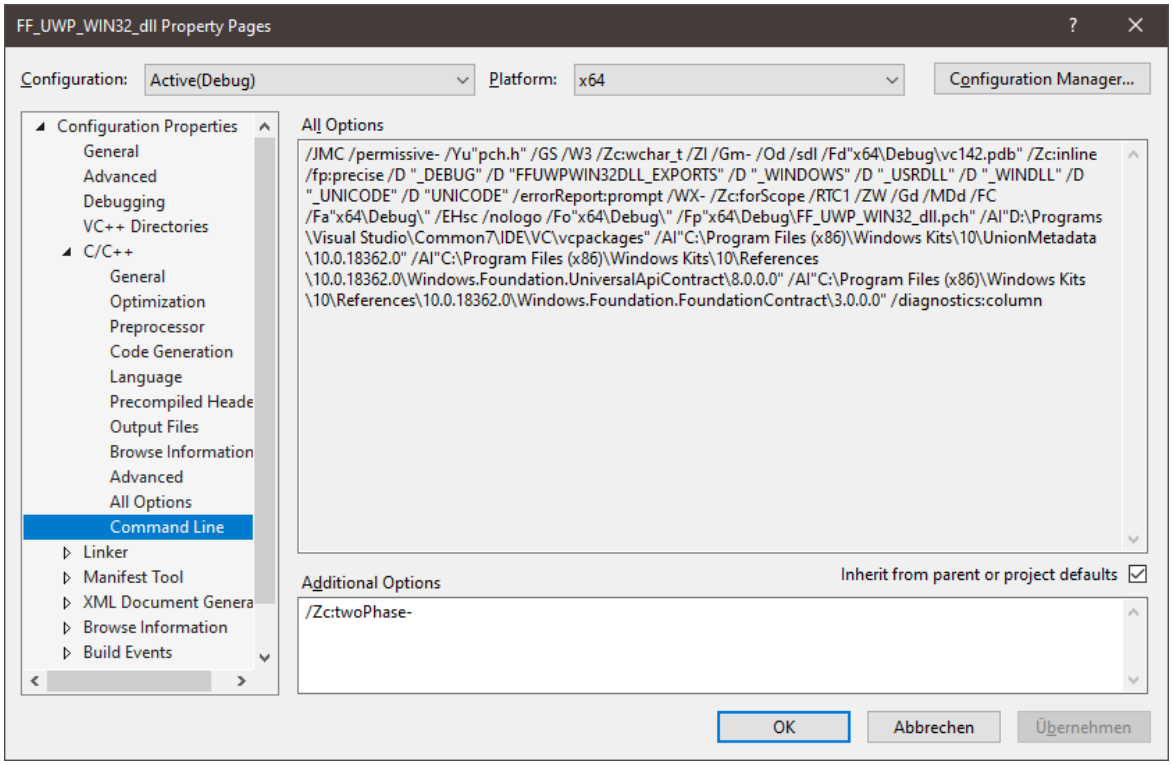

Figure 4 - C/C++ --> Command Line

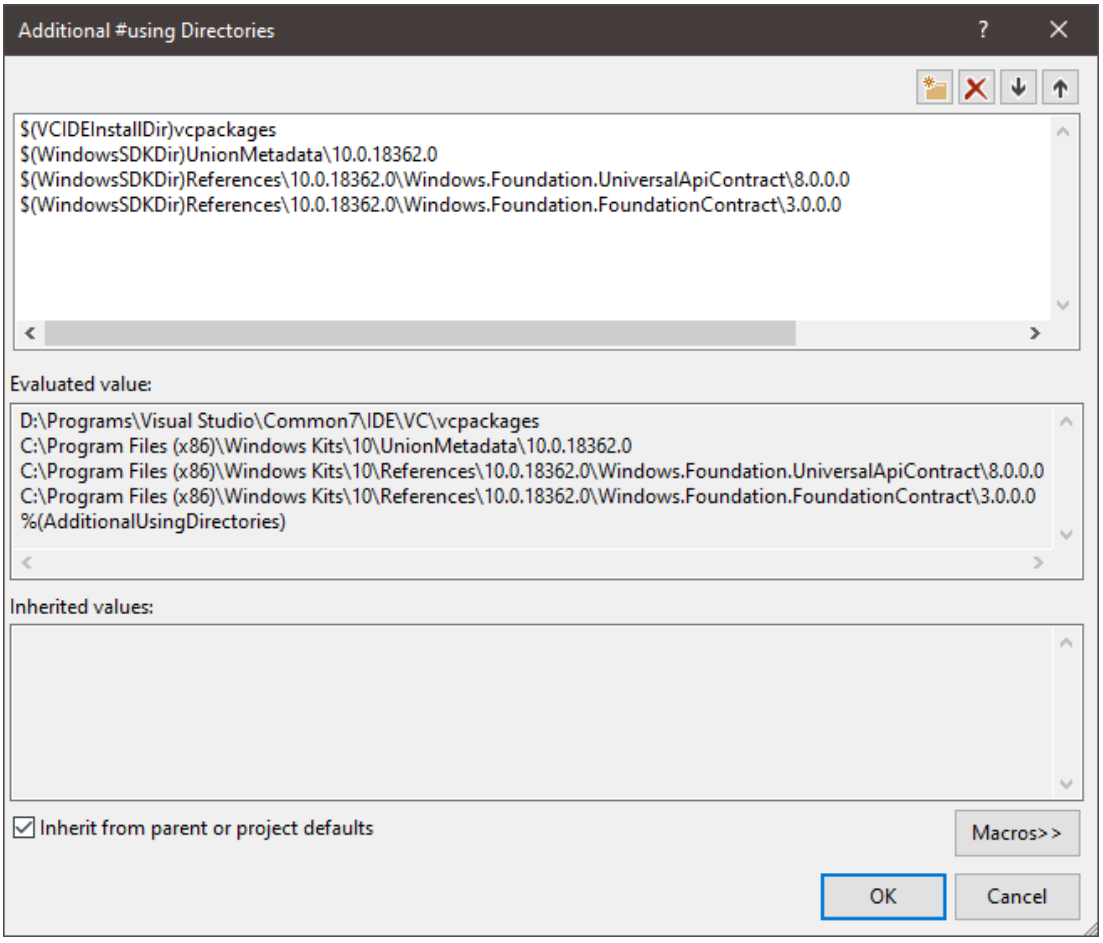

Figure 5 - Additional #using Directories

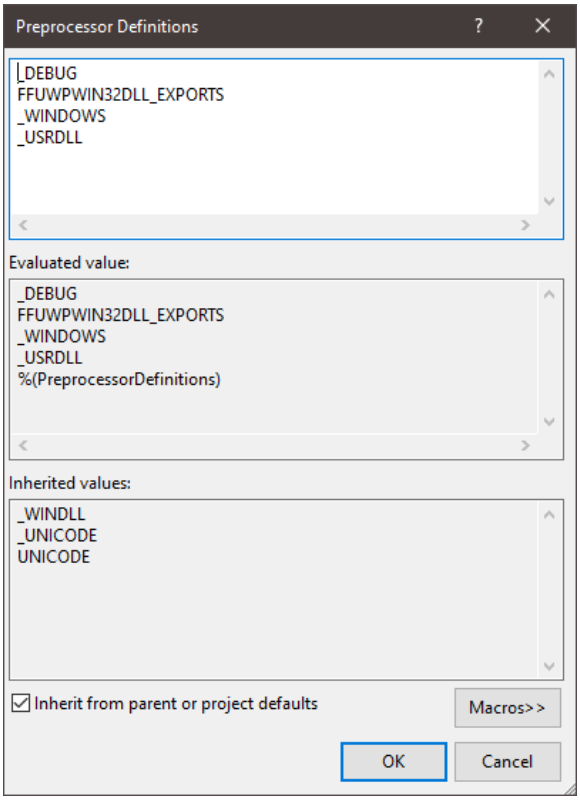

Figure 6 - Preprocessor Definitions

Final step, building!

Don't forget to build the .dll for x64 since MATLAB will be a x64 application. Be certain, that the changes above have been applied to the x64 settings in Visual Studio. Go to Build and Build Solution.

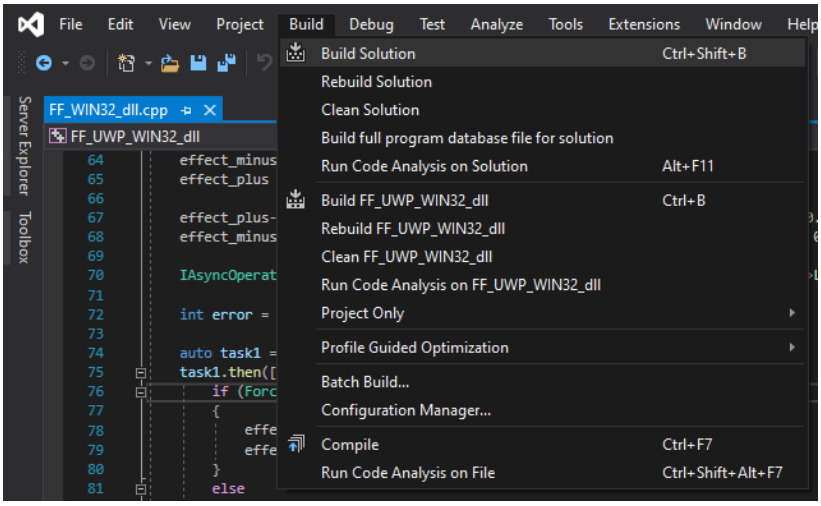

Figure 7 - Build the .dll

Using the library (MATLAB)

There are several ways to utilize the library in MATLAB:

1. Load the library with **loadlibrary**: https://de.mathworks.com/help/matlab/ref/loadlibrary.html

2. Use the **clibgen.buildInterface** (available since R2019a) https://de.mathworks.com/help/matlab/ref/clibgen.buildinterface.html
3. Build a **mex-function** https://de.mathworks.com/help/matlab/ref/mex.html

Building with **loadlibrary** and **clibgen.buildInterface** is similar, however, loadlibrary requires C-extern-functions and clibgen.buildInterface can utilize C++ libraries but needs a MATLAB version equal or higher R2019a. From Visual Studio the following files are required in all implementations followed:

- FF_WIN32_dll.h
- FF_UWP_WIN32_dll.dll
- FF_UWP_WIN32_dll.lib

With loadlibrary

An example on how to use it with the library is given in the file ***tryout_loadlibrary.m*** in the folder Matlab_Simulink\matlab_loadlibrary.

```
% load library
if not(libisloaded('FF_UWP_WIN32_dll'))
    loadlibrary('FF_UWP_WIN32_dll','FF_UWP_WIN32_dll.h');
end
libfunctions('FF_UWP_WIN32_dll')

% INIT RACING WHEEL WITH .DLL and set FORCE FEEDBACK
fprintf('Looking for a Racing Wheel! \n');

assert(calllib('FF_UWP_WIN32_dll','initRacingWheel') == true,...
    'No Racing Wheel found! Connect one!')
fprintf('Racing Wheel found! \n')

assert(calllib('FF_UWP_WIN32_dll','initForceFeedback') == 0,...
    'Problem with Force Feedback');
fprintf('Forece Feedback initialized! \n');

% INIT STURCTS FOR BUTTON AND WHEELREADINGS
WheelReadings = struct();
WheelReadings.throttle = 0;
WheelReadings.brake = 0;
WheelReadings.clutch = 0;
WheelReadings.angle = 0;
WheelReadings.mastergain = 0;
WheelReadings.timestamp = 0;

buttonReadings = struct();
buttonReadings.gearup = 0;
buttonReadings.geardown = 0;
buttonReadings.ST = 0;
buttonReadings.SE = 0;
buttonReadings.X = 0;
buttonReadings.O = 0;
buttonReadings.Square = 0;
buttonReadings.Triangle = 0;
buttonReadings.L2 = 0;
buttonReadings.R2 = 0;
buttonReadings.L3 = 0;
buttonReadings.R3 = 0;
buttonReadings.DPadDown = 0;
buttonReadings.DPadUp = 0;
buttonReadings.DPadLeft = 0;
buttonReadings.DPadRight = 0;

% REED WHEEL AND BUTTONS
buttonReadings = calllib('FF_UWP_WIN32_dll','readingButton',buttonReadings);
WheelReadings = calllib('FF_UWP_WIN32_dll','readWheelStatus',WheelReadings);

% FORCE-FEEDBACK STUFF -> DO SOMETHING ON BUTTON ACTION
calllib('FF_UWP_WIN32_dll','FF_zero')
if buttonReadings.X == true
    calllib('FF_UWP_WIN32_dll','FF_minus',0.5)
    xline(length(angle_container),'--');
elseif buttonReadings.O == true
    calllib('FF_UWP_WIN32_dll','FF_plus',0.5)
    xline(length(angle_container),'-');
end
```

Figure 8 - Code example from tryout_loadlibrary.m

With clibgen

Instructions on how to build a MATLAB readable interface file for the library can be found online or in the ***build_clibgen_FF_UWP_WIN32_dll.m*** in the folder Matlab_Simulink\matlab_clibgen\Build_clibgen_dll.

After building the interface, the library can be used as described in the example in the file ***tryout_clibgen.m***, see Matlab_Simulink\matlab_clibgen\Tryout_clibgen.

```
% INIT RACING WHEEL WITH .DLL and set FORCE FEEDBACK
fprintf('Looking for a Racing Wheel! \n');

assert(clib.FF_UWP_WIN32_dll.initRacingWheel == true,...
    'No Racing Wheel found! Connect one!')
fprintf('Racing Wheel found! \n')

assert(clib.FF_UWP_WIN32_dll.initForceFeedback() == 0,...
    'Problem with Force Feedback');
fprintf('Forece Feedback initialized! \n');

% INIT STURCTS FOR BUTTON AND WHEELREADINGS
buttonReadings = clib.FF_UWP_WIN32_dll.buttonReadings;
WheelReadings = clib.FF_UWP_WIN32_dll.WheelReadings;

% REED WHEEL AND BUTTONS
clib.FF_UWP_WIN32_dll.readingButton(buttonReadings);
clib.FF_UWP_WIN32_dll.readWheelStatus(WheelReadings);

    % FORCE-FEEDBACK STUFF -> DO SOMETHING ON BUTTON ACTION
if buttonReadings.X == true
    clib.FF_UWP_WIN32_dll.FF_minus(0.5)
    xline(length(angle_container),'--');
elseif buttonReadings.O == true
    clib.FF_UWP_WIN32_dll.FF_plus(0.5)
    xline(length(angle_container),'-');
end
```

Figure 9 - Example, full example see tryout_clibgen.m

Using the library (Simulink)

To utilize the library in Simulink the s-function builder is used: See folder simulink_sfunction. Therein a builder and tryout folder are found. In the builder a Simulink model generates the .mexw64 which can be used in the tryout folder to test the wheel. The configuration of s-function-builder is below:

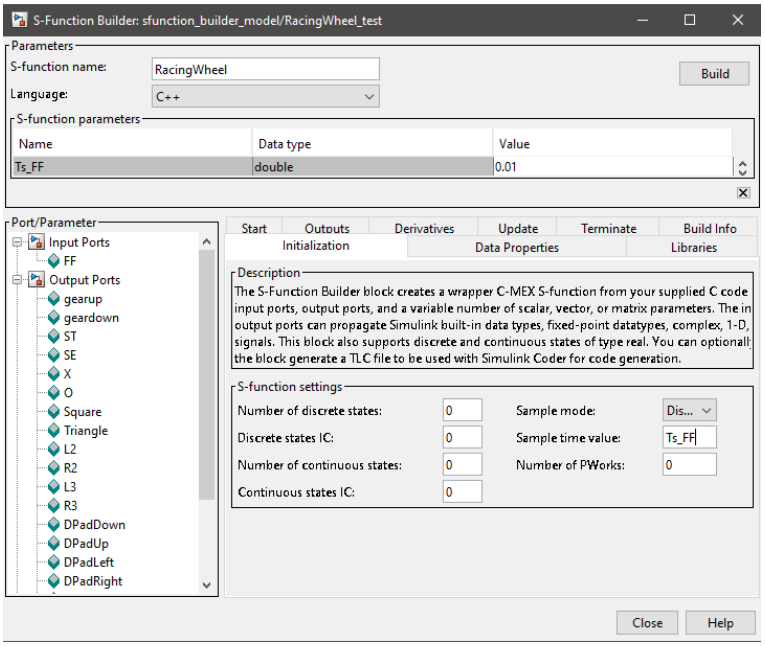

Figure 10 - s-function-builder

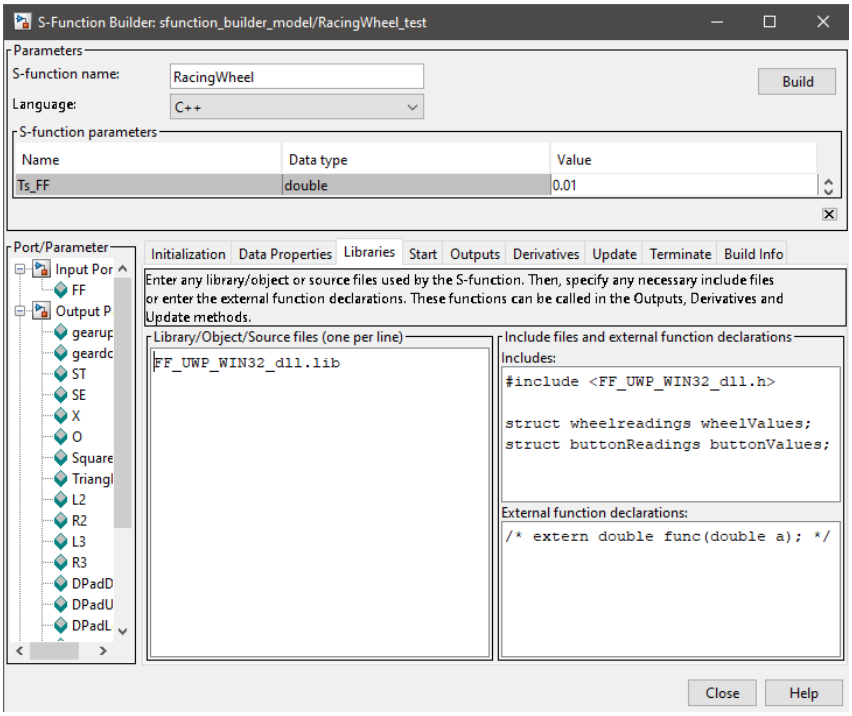

Figure 11- s-function-builder

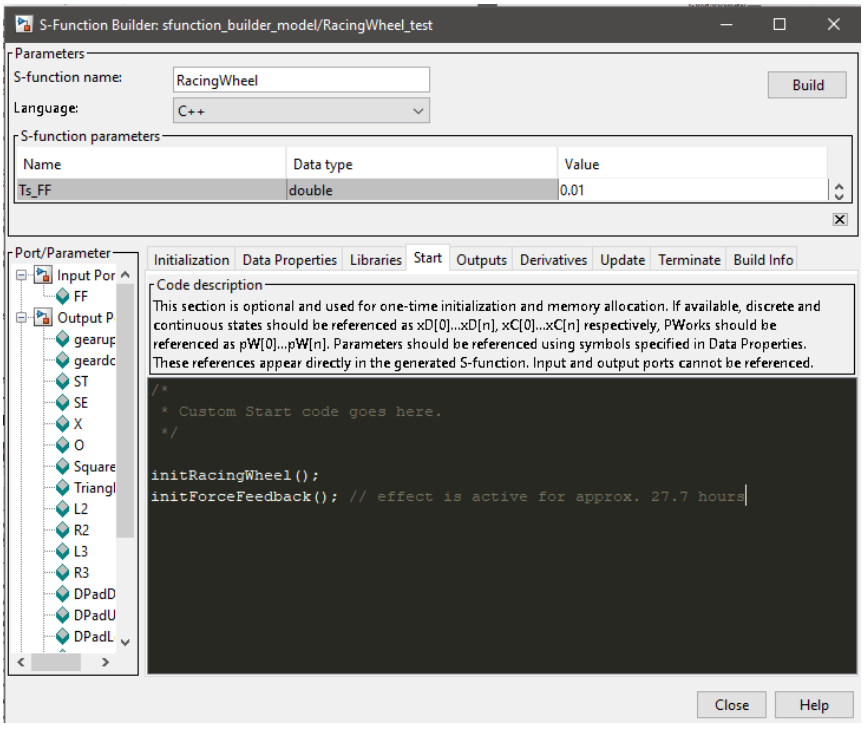

Figure 12 - s-function-builder

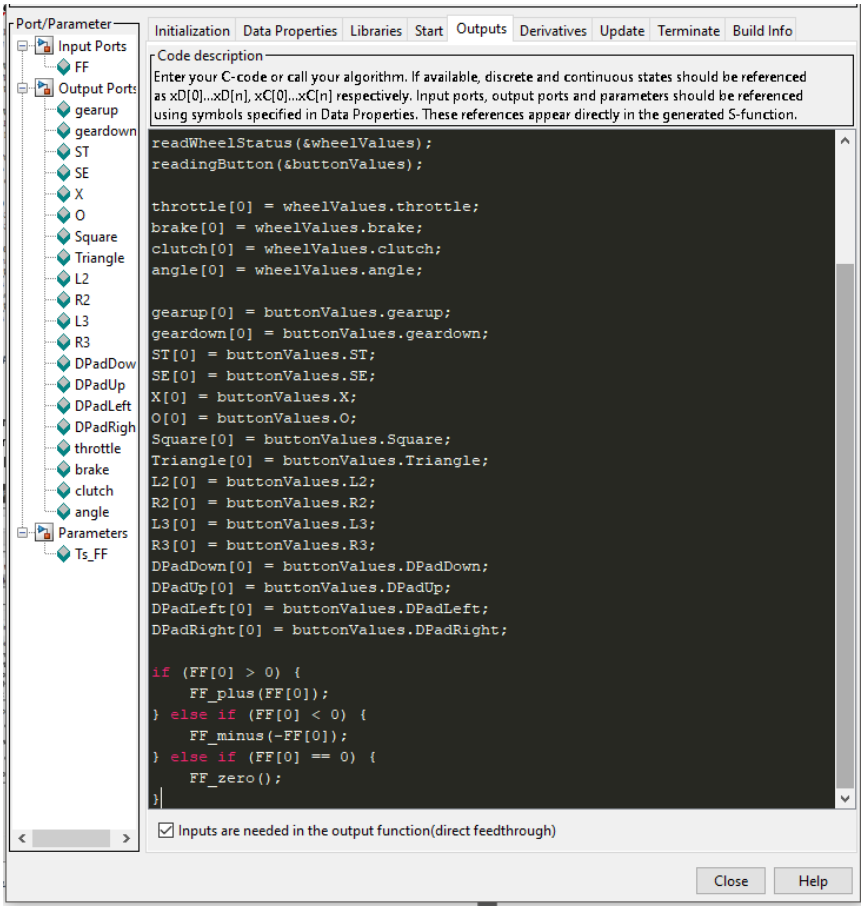

Figure 13 - s-function-builder

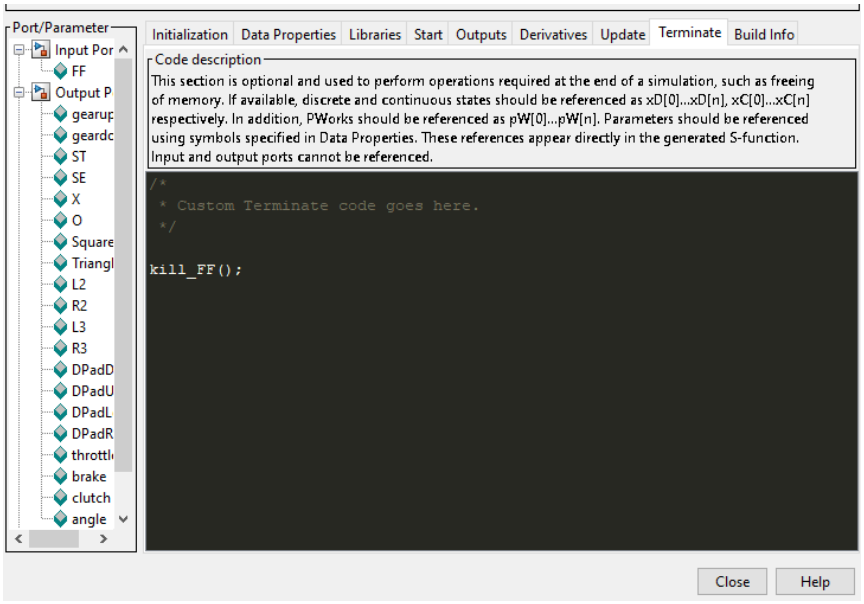

Figure 14 - s-function-builder

To test the Simulink block switch the directory and open the tryout_sfunction. Therein the build_folder contains the library files and the .mexw64. Run tryout_sfunction.m.

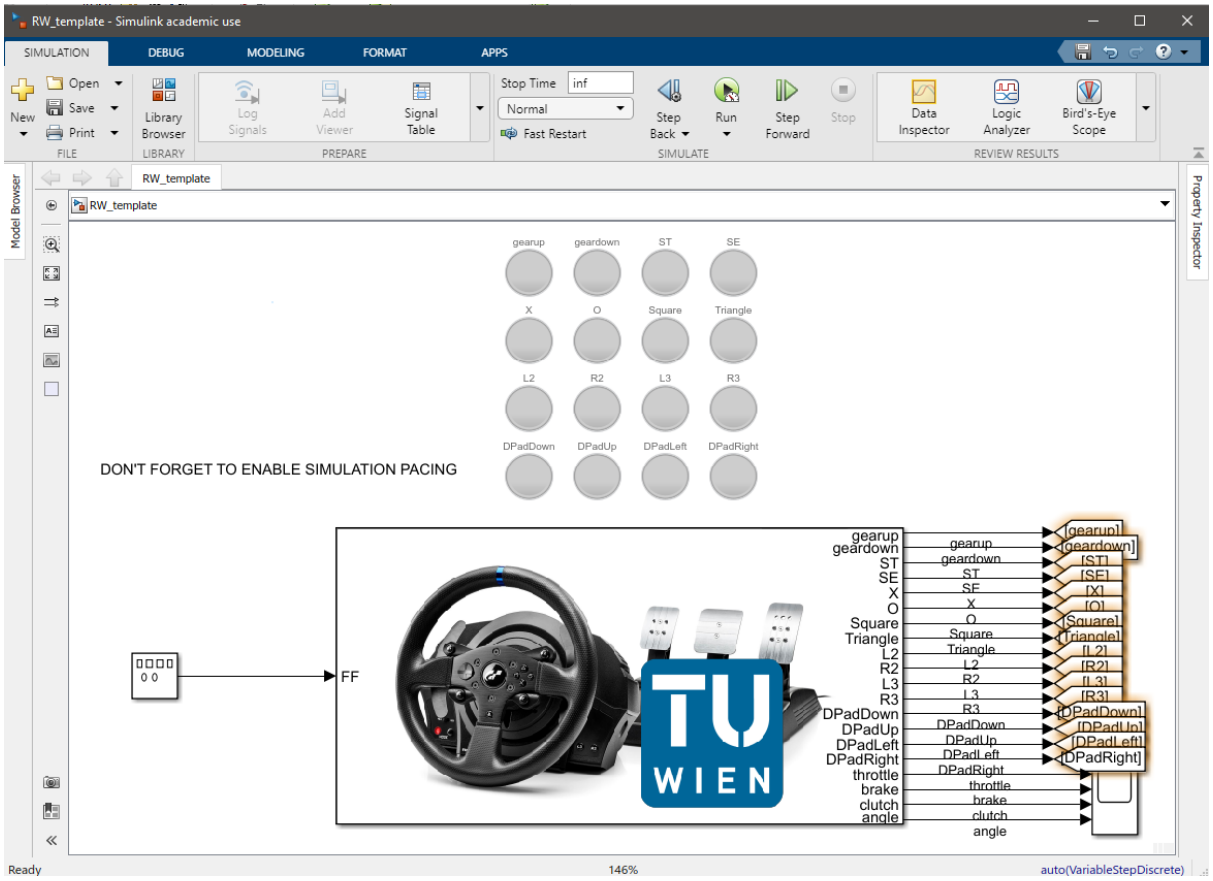

Figure 15 - The simulink block with custom icon

Note for Simulink: To have close to real-time-simulation, the simulation pacing option in Simulink is enabled to 1x.

Using the library (Simulink + Truckmaker)

Based on the DevConnAT git repository, the already existing demo-showcase was modified to feature force-feedback capability. For the purpose of consistence, the folder structure of the DevConnAT repository was untouched and all changes were made to be backwards compatible with the already existing showcase. Necessary modifications were done in the following folders and files, see as well Figure 16:

- **test_showcase_demo:**
  - Sampling Time Ts = 0.05s → PID for Force-Feedback tuned at 0.05s.
  - p.filename_road: Rundkurs
  - p.veh.ScriptControl(end + 1) = `'KeyValue set Vehicle Demo2AxleSemiTruck4x2_Volvo_ImportedClutchGearboxControl_SteerByTrq';`
    To switch from angle or torque steering → choose different vehicle
  - Most of the MPC stuff was disabled for faster simulation.
- **Demo2AxleSemiTruck4x2_Volvo_ImportedClutchGearboxControl_SteerByTrq:**
  - Added this new vehicle which has the torque steering module activated.
- **FF_UWP_WIN32_dll / RacingWheel.mexw64 / icon.png**
  - Library integration for the Simulink block in the truck.mdl file.
- **startup.dict**
  - Definition of custom dictionary variables used in the truck.mdl file.

- **truck.mdl**
  - Truckmaker for Simulink file
  - Modifications done in the DriveMan and Vehicle-Block

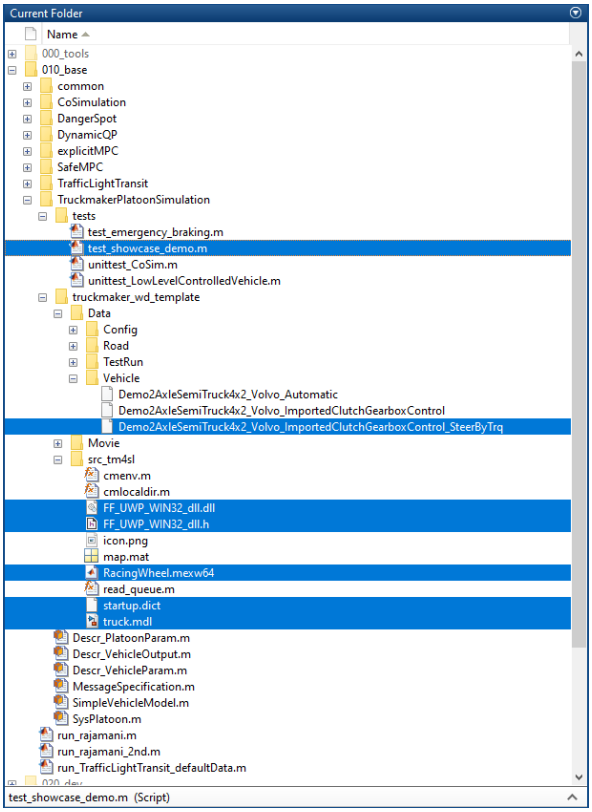

Figure 16 - The highlighted files were either modified or added.

The integration of the block takes place in the truck.mdl file, see figure below.

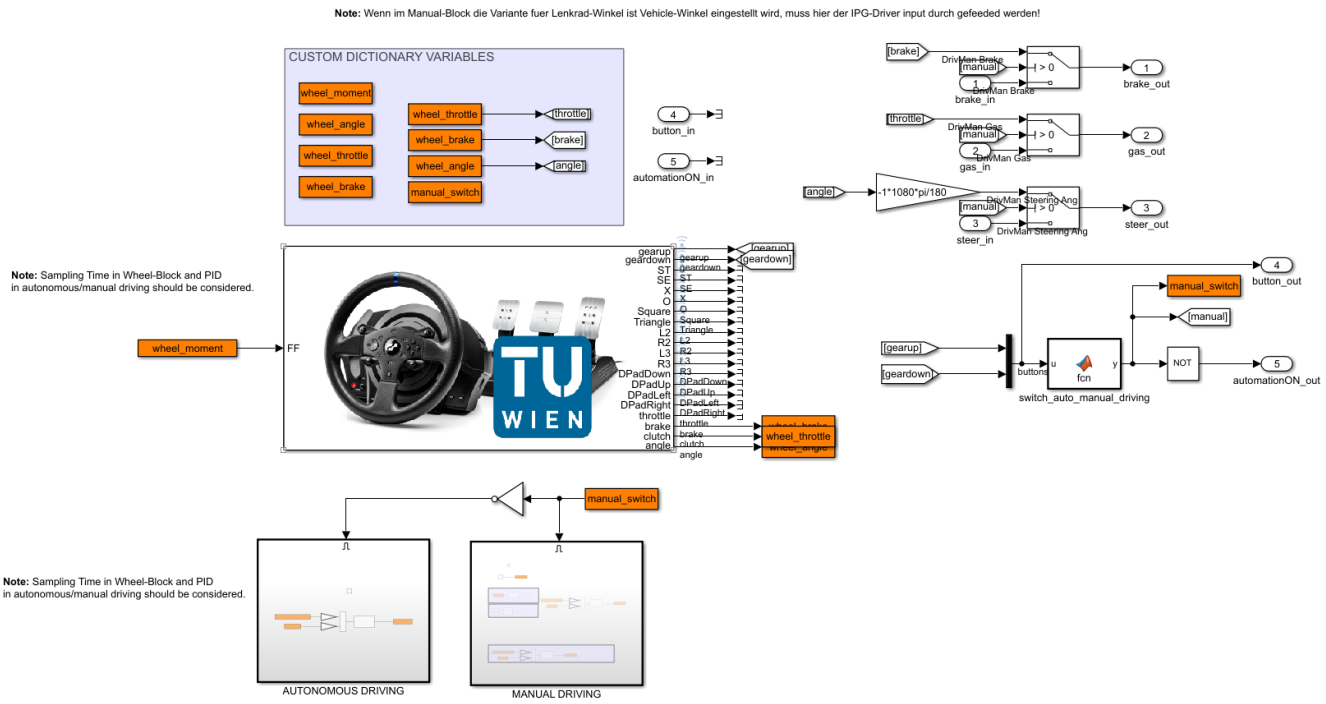

Figure 17 - Integration of Racing-Wheel in the truck.mdl

Force-Feedback for Autonomous-Driving as a show-effect (Angle Steering)

First there is a simple show-case-effect, which emulates the steering wheel of the autonomous driving truck. Make sure, that the vehicle with angle steering is loaded in test_showcase_demo.m:

```
p.veh.ScriptControl(end + 1) = 'KeyValue set Vehicle
Demo2AxleSemiTruck4x2_Volvo_ImportedClutchGearboxControl';
```

Run the test_showcase_demo.m. With the simulation running, the wheel should display the vehicles steering angle in real time. By pulling the geardown paddle, the mode can be changed to manual driving (force-feedback disabled). By pulling the gearup paddle, the show-case-effect can be turned on again.

Figure 18 - Show-Effect running on ThrustMaster T300RS

For the show-case-effect a PID-Controller was tuned. The input is the difference between the IPG-Driver wheel angle and the racing-wheel angle. The control input for the wheel is limited in the range of [-1,1].

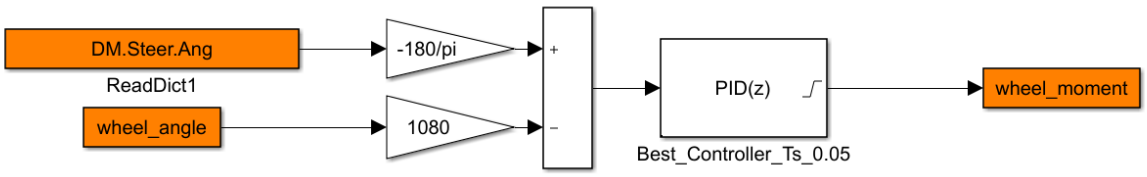

Figure 19 - Force-Feedback PID-Controller

Force-Feedback for Manual-Driving (Torque Steering)

Check for the show-effect, that the vehicle with torque steering is loaded in test_showcase_demo.m:

```
if torque_steering
    p.veh.ScriptControl(end + 1) = 'KeyValue set Vehicle
    Demo2AxleSemiTruck4x2_Volvo_ImportedClutchGearboxControl_SteerByTrq';
    p.relpathForm_simulink_mdl = p.relpath_tm4sl + '/' +'truckTrq%s.mdl';
end
```

In this instance the Simulink model beneath get's enabled. Here a crude model of the racing wheel is utilized to "estimate" the user input on the racing wheel. This user torque is applied to the simulation steering rack.

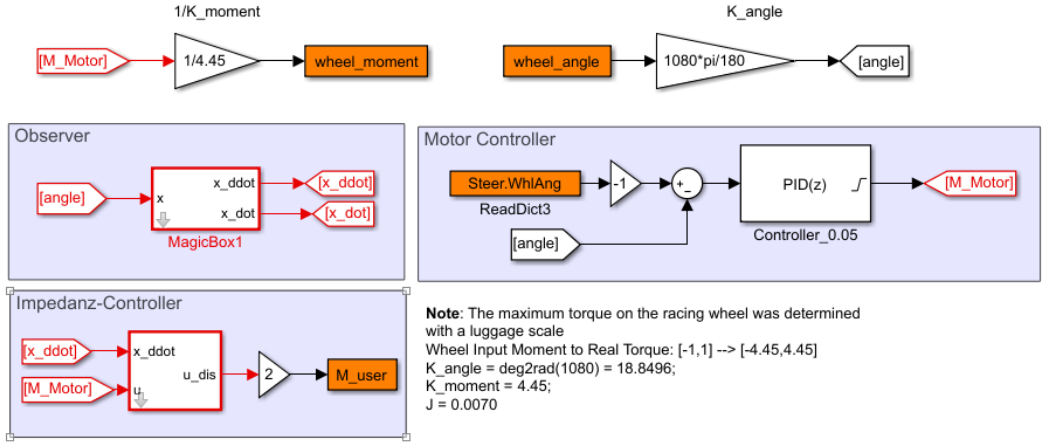

Figure 20 - Torque Steering Simulink File

Based on a crude identified model, a simple impedance controller was designed. The model and controller can be found in the respective Simulink-file. Note, the controller is tuned for 0.05 Ts. The vehicle is now controlled by the input moment given by the user.

Figure 21- Force-Feedback on the Thrustmaster Racing Wheel. The vehicles wheel gives great feedback.

Identifying a model for the racing wheel

For the Force-Feedback with impedance control a model was required to calculate the user input on the wheel. Several non-linear approaches were tried with e.g. the grey-box-model estimation toolbox built in in MATLAB. Due to unknown delays, friction and a unidentifiable dead zone in the racing wheel, estimating a good fitting model was not possible. Therefore, a crude method to estimate a simple model was chosen. The inertia of the wheel was determined by investigating the velocity gradient in a recorded force-feedback effect. Then the inertia was fitted. The simple model has the form of:

$$J\frac{d^2\varphi}{dt^2} = J\frac{d\omega}{dt} = M_{motor}$$

With a constant motor input a linear velocity gradient was visible and used to determine J. This crude model does barely represent the reality and gave only insufficient improvement as a controller. Therefore, a PID was previously chosen over for example a LQR controller.

Note: I have recorded 3 different runs with the racing wheel and saved the data. If one is eager enough to estimate an appropriate model, feel free to use the data. One dataset contains several bounce-backs – these happen if the racing wheel hits the end-stops.