Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Andy Lesmana

Integration eines Forcefeedback-Lenkrades in die CarMaker für Simulink- Simulationsumgebung

Fakultät Technik und Informatik
Department Fahrzeugtechnik und Flugzeugbau

Faculty of Engineering and Computer Science
Department of Automotive Engineering and Aircraft Construction

Andy Lesmana

Integration eines Forcefeedback-Lenkrades in die CarMaker für Simulink-Simulationsumgebung

Bachelorthesis eingereicht im Rahmen der Bachelorprüfung

im Studiengang Flugzeugbau
am Department Fahrzeugtechnik und Flugzeugbau
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Dirk Adamski
Zweitgutachter: Prof. Dr. Thomas Netzel

Abgegeben am 17.10.2016

Zusammenfassung

Andy Lesmana

Thema der Bachelorthesis

Integration eines *Forcefeedback*-Lenkrades in die *CarMaker* für *Simulink*-Simulationsumgebung

Stichworte

MATLAB/*Simulink*, C++, *G27* Software Development Kit (SDK), Integration, Schnittstelle, *MEX*-Datei, *S-function*, Testumgebung, Fehlerüberwachung

Kurzzusammenfassung

In der vorliegenden Arbeit wird eine Integration des Logitech SDKs für das *Forcefeedback*-Lenkrad G27 in die *MATLAB*/*Simulink* Umgebung durchgeführt. Um das SDK unter *MATLAB*/*Simulink* verwenden zu können wird eine *S-function* benötigt welche in diesem Fall in der Programmiersprache C++ verfasst und als *C MEX*-Datei kompiliert wird. Zum Testen der geforderten Funktionalität wird unter *Simulink* eine entsprechende Simulationsumgebung erstellt

Andy Lesmana

Title of the paper

Integration of a *forcefeedback*-steering wheel into the *CarMaker* for *Simulink*-simulation environment

Keywords

MATLAB/*Simulink*, C++, *G27* Software Development Kit (SDK), integration, interface, *MEX*-file, *S-function*, testing environment, error control

Abstract

In this study, the integration of the Logitech SDK for the *forcefeedback*-steering wheel *G27* into the *MATLAB*/*Simulink* environment will be described. To be able to use the SDK under *MATLAB*/*Simulink* a *S-function* is required which in this case is written using the programming language C++ and compiled as a *C MEX*-file. For testing purposes of the required functionality a simulation environment for *Simulink* is created

Danksagung

An dieser Stelle möchte ich mich herzlichst bei meinem betreuenden Professor Dr. Dirk Adamski bedanken, welcher mir mit hilfreichen Ideen und Vorschlägen jederzeit zur Seite stand und mir die Erstellung dieser Bachelorarbeit ermöglichte.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

Die Simulationsumgebung *CarMaker* wird für die Analyse der Fahrdynamik von Personenkraftwagen eingesetzt. Eine Variante von *CarMaker* ist in *MATLAB/Simulink* integriert. Hier sind die Schnittstellen zwischen dem vorhandenen Fahrerregler *IPG-Driver* und dem Fahrzeugmodell offen zugänglich. Um ein Fahrzeugmodell auch durch einen realen Fahrer fahren lassen zu können, kann an dieser Stelle die Verbindung zum Fahrerregler unterbrochen werden und durch eine neue Schnittstelle zu einem handelsüblichen *Forcefeedback*-Lenkrad ersetzt werden. In der *Simulink*- Toolbox 3D-Animation steht eine einfache Anbindungsmöglichkeit zur Verfügung, die aber nicht alle Optionen des Lenkrades nutzt. Die Firma Logitech stellt für ihr Lenkrad G27 ein Software Development Kit (SDK) zur Verfügung. Mithilfe dieses SDK soll eine vollständige Schnittstelle in C++ erzeugt werden, die dann als *S-function* in *Simulink* eingebunden werden kann. Hier sollen alle angebotenen Möglichkeiten des Lenkrades, der Pedalerie, der LEDs und des Gangwahlhebels verwendet werden können.

Um die Funktionstüchtigkeit der Schnittstelle testen zu können, wird eine Testumgebung in *Simulink* erstellt. Die Eingänge (*Forcefeedback*, Straßenmodi, LEDs) sollen auf diese Weise vom Benutzer vorgegeben werden können. Die Ausgänge der Schnittstelle sollen geeignet dargestellt werden, damit eine ordnungsgemäße Funktion überprüft werden kann.

Die Schnittstelle soll vollständig dokumentiert und mit Fehlerüberwachungen versehen werden, die aussagekräftige Warnungen und Fehlermeldungen an *MATLAB* weiterleiten. Die Fehlermöglichkeiten sollen dabei analysiert und das umgesetzte Fehlerkonzept beschrieben werden.

Diese Arbeit ist inklusive der Einführung in sechs Teile zerlegt. Zunächst soll ein Überblick über die vorliegenden Elemente der Arbeit geschaffen und das benötigte Know-How vermittelt werden. Dafür wird zuallererst die Funktionalität sowie Verwendung von *S-functions* erläutert und auf ihre Eigenschaften eingegangen. Um ein genaueres Bild von dem *Logitech* SDK zu bekommen, wird dieses in Kapitel drei als Ganzes angerissen und dessen Voraussetzungen geklärt. In Kapitel vier wird zum einen der Erstversuch der Integration dokumentiert und ein Überblick über den Entwicklungsprozess der entstandenen *S-function* geliefert, sowie die Funktionalität und das eingebaute Error-Handling erklärt. Das fünfte Kapitel dient der Erläuterung der Simulationsumgebung/Schnittstelle und der daran gebundenen Funktionen und wird mit Anschluss des sechsten Kapitels mit einer Schlussbetrachtung abgeschlossen. Die Informationen zur Bearbeitung wurden zum Teil vorliegenden Dokumentationen entnommen, deren Quellen in dem Literaturverzeichnis hinterlegt sind.

2 Simulink S-functions

In diesem Kapitel soll ein Überblick über die in dieser Arbeit verwendete *S-function* API geschaffen werden. Es soll geklärt werden was *S-functions* sind, wie sie arbeiten, wozu sie verwendet und wie genau sie aufgebaut werden. Da S-functions einen Teil der *Simulink*-Funktionalität darstellen, soll auch ein Überblick über *Simulink* und dessen Arbeitsweise geschaffen werden. Informationen und Abbildungen sind dabei den *MATLAB* und *Simulink* Dokumentationen entnommen worden, deren Quellen im Literaturverzeichnis hinterlegt sind.

2.1 MATLAB/Simulink

MATLAB ist eine Software des Unternehmens *The Mathworks* und wurde entwickelt, um vorwiegend mathematisch-technische Probleme zu lösen. Der Name *MATLAB* leitet sich dabei aus MATrix LABoratory ab. Dies ist eine Anspielung auf die größte Stärke des Programmes, nämlich dem Rechnen mit Matrizen. *MATLAB* bietet zahlreiche Erweiterungsmöglichkeiten über so genannte *Toolboxen*, wodurch beispielsweise mechanische, statistische, regelungstechnische oder sogar wirtschaftswissenschaftliche und biologische Probleme gelöst werden können.
Simulink stellt dabei eine dieser *Toolboxen* dar und liefert eine Blockdiagrammumgebung für die Mehrdomänen-Simulation und das Model-Based Design. Dabei unterstützt *Simulink* den Entwurf und die Simulation auf Systemebene und ermöglicht außerdem die automatische Codegenerierung und das kontinuierliche Testen und Verifizieren von Embedded Systems. Das Programm umfasst einen Grafikeditor, benutzerdefinierbare Blockbibliotheken und Solver für die Modellierung und Simulation von dynamischen Systemen. Da *Simulink* in *MATLAB* integriert ist, lassen sich so *MATLAB*-Algorithmen einfach in Modelle aufnehmen und Simulationsergebnisse wiederum in *MATLAB* analysieren und weiterverarbeiten.

2.2 Was ist eine S-function?

Eine *S-function* (*system-function*) ist eine in Computersprache verfasste Beschreibung eines *Simulink*-Blocks. Dies ermöglicht Benutzern die vorhandene Bibliothek an *Simulink*-Blöcken mit eigenen zu erweitern und so eine umfangreichere Funktionalität und Flexibilität zu gewährleisten. *S-functions* können auf unterschiedliche Weise und in unterschiedlichen Sprachen wie *MATLAB*, C, C++ oder Fortran erstellt werden. Es gibt fünf Möglichkeiten eine *S-function* zu erstellen:

- Eine Level-1 *MATLAB S-function* bietet eine einfache *MATLAB* Schnittstelle, um mit einem kleinen Teil der *S-function API* zu interagieren.

- Eine Level-2 *MATLAB S-function* ermöglicht es ein umfangreicheres Set der *S-function API* zu nutzen und unterstützt Codegenerierung.
- Eine handgeschriebene *C MEX S-function* welche die meiste Flexibilität hinsichtlich der Programmierung bietet. Diese bietet die Möglichkeit Algorithmen als *C MEX S-function* zu implementieren oder existierenden C, C++ oder Fortran Code einzubinden. Dies setzt Wissen über die *S-function API*, sowie natürlich der verwendeten Programmiersprache voraus.
- Die Benutzung des *S-function builders*, welcher eine grafische Schnittstelle zur Programmierung einer Teilmenge von *S-functions* bietet. Dieser bietet die Möglichkeit *S-functions* zu bauen oder existierenden C/C++ Code zu implementieren ohne sich mit der *S-function API* auseinandersetzen zu müssen.
- Das Legacy Code Tool welches ein Set aus *MATLAB*-Befehlen darstellt, die helfen eine *S-function* zu erstellen oder *legacy C/C++ Code* zu implementieren. Im Vergleich mit dem *S-function builder* oder hangeschrieben *C MEX S-functions* hat das Legacy Code Tool allerdings einen beschränkteren Zugang zu den Methoden der *S-function API.*

Diese Arbeit beschränkt sich auf die dritte Methode, der Erstellung einer handgeschriebenen *C MEX S-function*.

Zur Kompilierung einer *C MEX S-function* muss zunächst eine *MEX*-Datei (siehe 2.4) erstellt werden. *S-functions* benutzen eine spezielle Syntax, der es diesen erlaubt mit *Simulink* zu kommunizieren. Die Interaktion zwischen *S-functions* und *Simulink* ähnelt sehr der Interaktion zwischen *Simulink* und den bereits vorhandenen Standardblöcken.

Die Form einer *S-function* ist sehr allgemein und nahezu alle *Simulink* Modelle können als *S-functions* beschrieben werden.

2.3 Wann macht es Sinn eine S-function zu verwenden?

Wie schon bereits erwähnt ermöglichen *S-functions* die *Simulink* Bibliothek mit eigenen Blöcken zu erweitern. Dies stellt auch die am meist genutzte Funktion von *S-functions* dar. Es gibt jedoch auch noch weitere Gründe wie:

- Bereits existierenden C-Code in die Simulation einzubeziehen
- Neue Universalblöcke zur Bibliothek hinzuzufügen
- Ein System als mathematisches Set von Gleichungen zu beschreiben
- Die Möglichkeit zu nutzen, graphische Animationen einzubeziehen

Ein Vorteil von Universalblöcken ist, dass man sie öfters in einem Modell verwenden kann, mit jeweils anderen Parametern und/oder Eingabewerten.

2.4 MEX-files

Wie der Name bereits vermuten lässt, kommt man ohne eine *MEX*-Datei bei einer *C MEX S-function* nicht weit. *C MEX S-functions* werden nämlich als *MEX*-Dateien kompiliert. Die Abkürzung *MEX* steht für *MATLAB Executable*. *MEX*-Dateien sind dynamisch gelinkte Subroutinen, welche innerhalb von *MATLAB* geladen und ausgeführt werden können. Dynamisch gelinkt heißt, dass das Programm nach den *runtime libraries* sucht, sobald die Datei ausgeführt wird. Diese Subroutinen entspringen dabei, im Falle einer *C MEX-S-function*, einer Source-Datei, welche in C, C++ oder Fortran geschrieben ist. Um eine *MEX*-Datei auszuführen, schreibt man einfach den Namen der Datei ohne Endung in das *MATLAB command window*.

MEX-Dateien können dabei unterschiedliche Bedeutungen haben, wie die folgende Abbildung 1 zeigt:

MEX Term	Definition
source MEX file	C, C++, or Fortran source code file.
binary MEX file	Dynamically linked subroutine executed in the MATLAB environment.
MEX function library	MATLAB C/C++ and Fortran API Reference library to perform operations in the MATLAB environment.
mex build script	MATLAB function to create a binary file from a source file.

Abbildung 1: MEX Definitionen

Um eine Source-Code-*MEX*-Datei zu erstellen benötigt man:

- Einen für *MEX*-Dateien gültigen Source-Code (viele Beispiele zu finden unter `MATLABroot/extern/examples`)
- Einen Compiler, welcher von *MATLAB* unterstützt wird
- Die *C/C++ Matrix Library API* und die *C MEX Library API*-Funktionen
- Das *MEX build script*

Sind alle diese Voraussetzungen erfüllt, kann die *MEX*-Datei über die *MATLAB command window* Eingabe „MEX filename.cpp“ erstellt werden. Werden mehrere Source-Dateien für die Erstellung benötigt, schreibt man diese einfach mit einem Leerzeichen getrennt dahinter: „MEX filename.cpp otherfile.cpp yetanotherfile.cpp…“.
Erwähnenswert dabei ist, dass die Header nicht mit angegeben werden, aber *MATLAB* in einem Verzeichnis vorliegen müssen. Wichtig dabei ist, dass der Pfad über *MATLAB* auch als solcher hinzugefügt wurde.

Möchte man seine *MEX*-Datei bzw. *S-function* zusätzlich auch debuggen können, muss eine *PDB*-Datei (*Program Database*) erstellt werden. Diese lässt sich durch die zusätzliche Eingabe von „-g“ erzeugen: „MEX -g filename.cpp“. Diese Datei enthält dabei „Debug- und Projektzustandsinformationen, die die inkrementelle Verknüpfung einer Debugkonfiguration des Programms ermöglichen“. (*PDB*-Dateien *Microsoft* Dokumentation)

Sollte eine erhaltene *MEX*-Datei nicht ausführbar sein, ist es sehr wahrscheinlich, dass dies an Plattform und/oder Versionskompatibilitätsproblemen oder fehlenden *runtime libraries* liegt.

2.5 Funktionsweise von S-functions

Jeder Block in *Simulink* interagiert über drei Vektoren: einem Vektor aus Eingabewerten (*inputs*) u, einem Vektor aus Zuständen (*states*) und einem Vektor von Ausgabewerten (*outputs*) y, wie die folgende Abbildung 2 zeigt.

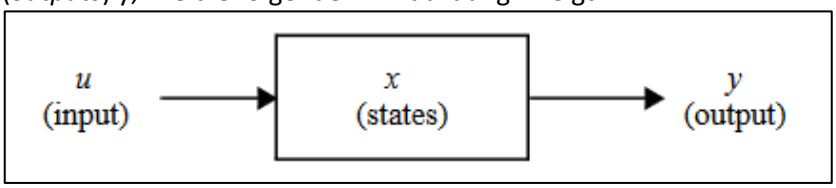

Abbildung 2: Input/States/Outputs eines Simulink-Blockes

Eines *C MEX S-function* besitzt dabei zwei *state* Vektoren. Einen für *continuous states* und einen weiteren für die *discrete states*.
Der Unterschied zwischen *continuous states* und *discrete states* ist dabei folgender:

- Ein *discrete state* wird an einem *time step* (siehe Abbildung 6 für Visualisierung eines *time steps*) gespeichert und später wieder abgerufen. *Simulink* tut dabei nichts außer den Wert zu speichern.
- Ein *continuous state* wird dabei von dem *Simulink Solver* integriert. Dabei gibt man dem Solver die Ableitung „dx/dt“ und bekommt den integrierten Wert „x“ zurück.

Continuous und *discrete states* sind dabei nicht mit *continuous* und *discrete sample times (siehe 2.6)* zu verwechseln, welche für eine „gleichbleibebende“ und „unabhängige“ *sample time* stehen.

Die mathematischen Beziehungen zwischen *inputs*, *states* und *ouputs* lassen sich im Folgenden (Abbildung 3) so ausdrücken:

$$y = f_0(t, x, u) \qquad \text{(output)}$$

$$\dot{x}_c = f_d(t, x, u) \qquad \text{(derivative)}$$

$$x_{d_{k+1}} = f_u(t, x, u) \qquad \text{(update)}$$

$$\text{where} \quad x = x_c + x_d$$

Abbildung 3: Input/States/Output Beziehungen MEX-Datei

2.6 S-function Konzepte

Um eine *S-function* zu aufzubauen sollten die folgenden Konzepte verstanden werden:

- *Direct feedthrough*
- Dynamische Input-Port-Weiten
- Das Setzen von *sample times* und *offsets*

Direct Feedthrough

Direct feedthrough bedeutet, dass der Output oder die variable *sample time* direkt durch einen Eingangswert des Input-Ports kontrolliert wird. Ein *S-function* Input-Port hat dabei meist *direct feedthrough* wenn:

- Innerhalb der Output-Methode *mdlOutputs* das Eingangssignal innerhalb von dort definierten Gleichungen verwendet wird.
- Die *S-function* eine Funktion mit variabler sample time darstellt und die Berechnung des nächsten *sample hits* von der Eingangsvariable u abhängt.

Ein Beispiel für ein System, welches für seine Eingangssignale *direct feedthrough* benötigt, wäre die Operation y = k * u, wobei u das Input-Signal, k den Multiplikator und y das Output-Signal darstellt.

Ein Beispiel für ein System, welches für seine Eingangssignale kein *direct feedthrough* benötigt, wäre folgender Integrationsalgorithmus:

Outputs: $y = x$
Ableitung: $\dot{x} = u$

Die Variable x steht hier für einen *state*, $\dot{x}$ ist stellt eine Ableitung der Zeit dar, u ist das Eingangs- und y das Ausgangssignal.

Es ist sehr wichtig die *direct feedthrough*-Option richtig zu setzen, denn sie bestimmt die Reihenfolge in welcher die Blöcke in einem *Simulink*-Modell ausgeführt werden und wird benutzt um algebraische Loops zu erkennen.

Dynamische Input-Port-Weiten

S-function können mit dynamischen Input-Port-Weiten versehen werden. Dabei wird die Port-Weite beim Start der Simulation über das angeschlossene Eingangssignal festgestellt. Über die Port-Weite des Input-Signals können weiterhin die Anzahl von *continuous* oder *discrete states* und die Weite von Output-Ports bestimmt werden.

Innerhalb einer *C MEX S-function* geschieht dies über die Verwendung der Option *DYNAMICALLY_SIZED* innerhalb der *ssSetInputPortWidth*-Funktion in der *mdlInitializeSizes*-Methode (siehe 2.7). Der Nutzen dieser Funktion liegt darin, dass die *S-function* unterschiedliche Weiten von Eingangssignalen mit demselben Code bearbeiten kann und nicht erst ein neuer Block dafür erstellt werden muss.

Das Setzen von sample times und offsets

Bezüglich dem Zeitpunkt, wann eine *S-function* genau ausgeführt wird, besitzen diese einen hohen Grad an Flexibilität. Simulink liefert dabei die folgenden Optionen:

- *Continuous sample time*: Für S-functions, welche *continuous states* und/oder *nonsampled zero crossings* besitzen. Der Output verändert sich in *minor time steps*.
- *Continuous* aber fixiert in *minor time step sample time*: Für *S-functions,* deren Output nur während *major simulation steps* variiert.
- *Discrete sample time*: Für *S-functions* mit eigenständigen *sample times*. Dadurch kann genau eingestellt werden, wann *Simulink* den Block ausführt. Es kann außerdem ein Offset definiert werden, mit dem sich jeder *sample hit* verzögern lässt. Ein *sample time hit* wird über folgende Formel berechnet:
  Timehit = (n * period) + offset
  „n“ stellt dabei den derzeitigen *simulation step* dar. *Simulink* ruft dann die *mdlOutput* und *mdlUpdate* Methoden zu jedem *sample hit* auf.
- *Variable sample time*: Für *S-functions,* deren Intervalle von *sample hits* variieren können. Zum Anfang jedes Simulationsschrittes wird die *S-function* zur Bestimmung der Zeit des nächsten *hits* integriert.
- *Inherited sample time*: Für *S-functions,* deren *sample time* von einem angeschlossenen Block, einem Zielblock oder der schnellsten *sample time* im System abhängt. Ein Beispiel dafür wäre *Simulinks Gain*-Block. Eine solche *S-function* besitzt also keine eigene definierte *sample time*.

Außerdem können *S-functions* anstatt einer auch mehrere *sample times* besitzen.

2.7 Aufbau einer S-function

Der Aufbau einer *S-function* bezüglich des Source Codes muss einem bestimmten Muster folgen und bestimmte Routinen enthalten, um unter *Simulink* richtig zu funktionieren. Zu den benötigten Routinen gehören:

- *mdlInitializeSizes*
- *mdlInitializeSampleTimes*
- *mdlOutputs*
- *mdlTerminate*

Mithilfe von Abbildung 4 und der darin enthaltenen *MATLAB S-function times-two.c* soll der Aufbau einer einfachen *S-function* erklärt werden:

```
#define S_FUNCTION_NAME  timestwo
#define S_FUNCTION_LEVEL 2

#include "simstruc.h"

static void mdlInitializeSizes(SimStruct *S)
{
    ssSetNumSFcnParams(S, 0);
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        return;
    }

    if (!ssSetNumInputPorts(S, 1)) return;
    ssSetInputPortWidth(S, 0, DYNAMICALLY_SIZED);
    ssSetInputPortDirectFeedThrough(S, 0, 1);

    if (!ssSetNumOutputPorts(S,1)) return;
    ssSetOutputPortWidth(S, 0, DYNAMICALLY_SIZED);

    ssSetNumSampleTimes(S, 1);

    ssSetSimStateCompliance(S, USE_DEFAULT_SIM_STATE);

    ssSetOptions(S,
                 SS_OPTION_WORKS_WITH_CODE_REUSE |
                 SS_OPTION_EXCEPTION_FREE_CODE |
                 SS_OPTION_USE_TLC_WITH_ACCELERATOR);
}
```

```
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
    ssSetModelReferenceSampleTimeDefaultInheritance(S);
}

static void mdlOutputs(SimStruct *S, int_T tid)
{
    int_T             i;
    InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
    real_T            *y    = ssGetOutputPortRealSignal(S,0);
    int_T             width = ssGetOutputPortWidth(S,0);

    for (i=0; i<width; i++) {
        *y++ = 2.0 *(*uPtrs[i]);
    }
}

static void mdlTerminate(SimStruct *S)
{
}



#ifdef  MATLAB_MEX_FILE
#include "simulink.c"
#else
#include "cg_sfun.h"
#endif
```

Abbildung 4: Source Code der timestwo S-function

Das Beispiel startet mit den folgenden *define*-Statements:

```
#define S_FUNCTION_NAME  timestwo
#define S_FUNCTION_LEVEL 2
```

Das erste *define*-Statement beschreibt den Namen der *S-function*, mit welcher sie später in *Simulink* eingebunden wird. Das zweite Statement sagt aus, dass es sich hierbei um eine *Level 2 S-function* handelt. Es gibt *Level 1* und *Level 2 S-functions*. Die *Level 2 S-functions* stellen dabei eine Weiterentwicklung dar und bringen eine umfangreichere Funktionalität mit sich. Zum Beispiel wird mit einer *Level 1 S-function* nur *continous sample time* unterstützt. Das heißt der Block unterstützt keine eigene oder variable sample time. Gefolgt werden die *define*-Statements von der Zeile:

```
#include "simstruc.h"
```

Die Datei *simstruc.h* ist eine Header-Datei, welche eine Datenstruktur namens *SimStruct* definiert. Diese benötigt *Simulink,* um Informationen über die *S-function* zu erhalten. Diese Datenstruktur wird dabei jeder Methode der *S-function* übergeben und ist eine Voraussetzung für deren Funktionalität. Des Weiteren werden dort einige Makros definiert, um Werte der *MEX*-Datei zu lesen und zu schreiben. Beispiele für solche Makros wären Funktionen wie *mxCalloc*, *mexPrintF* oder *mexEvalString*. Die Verwendung solcher Makros innerhalb einer *S-function* führen allerdings zum *long jumping*, da *Simulink* zusätzlichen Overhead für das *exception handling* bereitstellen muss. Ein Verzicht auf solche Funktionen erbringt daher einen Performance Boost.

mdlinitialize

Im Folgenden Code schließen die Callback-Methoden an. Zeile 6 bis 28 beschreibt hier die *mdlInitializeSizes* Routine, welche die Größe und Weite der Input- und Output-Ports, sowie die Anzahl der Parameter, festelegt. Die Funktion *ssSetNumSFcnParams* gibt die Zahl der zu erwartenden Parameter an und ist in diesem Beispiel auf 0 gesetzt, da diese hier nicht benötigt werden. Bei jedem Initialisierungsprozess wird dabei die Anzahl der eingegebenen Parameter überprüft. Stimmt diese nicht mit der Anzahl der geforderten Parameter überein, gibt *Simulink* eine Fehlermeldung aus und beendet die Simulation. Diese Überprüfung wird in Zeilen 9-11 ausgeführt und sollte jeder *S-function* mitgegeben werden, um Fehler zu vermeiden.
In Zeilen 13 bis 18 werden die Anzahl der Input- und Output-Ports gesetzt. Bei einem Fehlschlag des Vorgangs wird auch hier die Simulation abgebrochen. In Zeile 14 und 18 wird mit *ssSetInputPortWidth* bzw. *ssSetOutputPortWidth* die Weite dieser Vektoren gesetzt. *DYNAMICALLY_SIZED* bedeutet hier, dass die Weite zu diesem Zeitpunkt noch nicht bekannt ist und von dem Eingangssignal abhängt. Die Funktion *ssSetInputPortDirectFeedThrough* in Zeile 15 erlaubt mit den angegebenen Vektoren der Eingangssignale innerhalb von *mdlOutputs* zu arbeiten. Ist diese Option nicht gesetzt, ist es nicht möglich ein Ausgangssignal direkt durch das Eingangssignal zu beeinflussen.
In Zeile 20 wird die Anzahl der *sample times* gesetzt. Zeilen 24-27 dienen dazu sonstige Optionen zu setzen, um festzulegen wie Simulink die erstelle *S-function* behandelt. Erwähnenswert hierbei ist die Option *SS_OPTION_EXCEPTION_FREE_CODE*. Diese legt fest, dass

auf den Overhead zum *exception handling* verzichtet wird und beschleunigt den Prozess. Wie bereits erwähnt sollte diese Option nur gesetzt werden, wenn es sich sicher ist, dass die *S-function* keine *long jumping* provozieren wird.

mdlSetSampleTimes

Die Routine *mdlSetSampleTimes* wird aufgerufen, um die *sample times* zu definieren. In diesem Fall erhält die *S-function* die Einstellungen durch *INHERITED_SAMPLE_TIME* von dem am *Input* angeschlossenen Block.

mdlOutputs

In *mdlOutputs* wird nun die eigentliche Berechnung durchgeführt. Zeilen 39 bis 42 zeigen dabei den üblichen Weg, wie in einer *S-function* auf die Eingangswerte und Ausgabewerte zugegriffen wird. Die Typdefinitionen *int_t* und *real_T* sind *MATLAB* eigen und erlauben eine größere Plattformunabhängigkeit. Der *real_T typedef* wechselt zum Beispiel für 16,32 und 64 Bit Systeme zwischen unterschiedlichen Datentypen. In Zeile 44 und 45 findet letztendlich die eigentliche Berechnung und Ausgabe statt.

mdlTerminate

In der Methode *mdlTerminate* werden normalerweise Funktionen ausgeführt, die am Ende einer Simulation notwendig werden, wie zum Beispiel reservierten Speicher freizugeben.
In diesem Fall ist dies jedoch nicht notwendig, da kein Speicher reserviert wurde.

Mit den Statements:

```
#ifdef  MATLAB_MEX_FILE
#include "Simulink.c"
#else
#include "cg_sfun.h"
#endif
```

Wird die *S-function* abgeschlossen. Diese sind notwendig, da *Simulink* diese Syntax benötigt, um die Datei richtig zu verarbeiten.

2.8 Einbindung in Simulink

Ist die *MEX*-Datei erzeugt, kann die *S-function* erstellt und eingebunden werden. Dazu wird *Simulink* gestartet und ein leeres Modell geöffnet. In der *Simulink*-Block-Bibliothek findet man unter „User-Defined-Functions“ den Block „S-function“. Wie alle andere Blöcke wird dieser einfach per Drag und Drop dem Modell hinzugefügt. Öffnet man nun diesen Block per Doppelklick, lässt sich in das Feld „S-function name“ der Name der zuvor erzeugten *MEX*-Datei eintragen. Dies wird in Abbildung 5 illustriert.

Um die Parameter einer *S-function* nutzen zu können, müssen diese, was Verwendung, Anzahl und Reihenfolge angeht, bekannt sein und im Zweifelsfall dem Source Code oder der

Dokumentation des Erstellers entnommen werden. Wird die *S-function* jedoch mit bereits vordefinierten Parametern in einer benutzerdefinierten Block-Bibliothek hinterlegt, ist dies nicht notwendig.

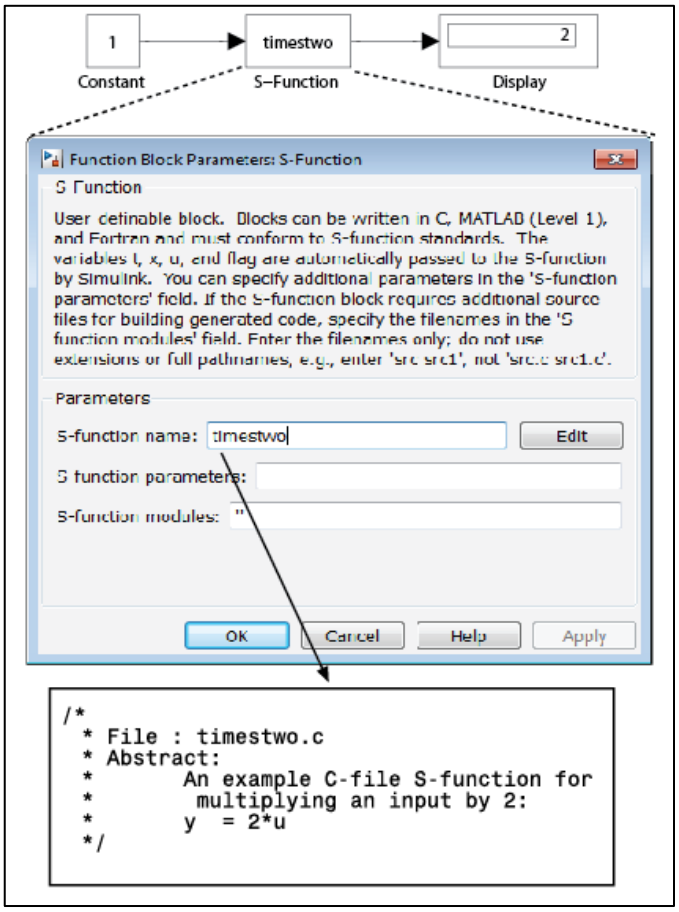

Abbildung 5: S-function Einbindung in Simulink

2.9 Arbeitsschritte einer Simulation

Die Schritte, welche bei einer Simulation durchlaufen werden lassen sich anhand von Abbildung 6 deutlich machen.

Am Anfang jeder Simulation steht die Initialisierung. Funktionen wie *mdlInitializeSizes*, *mdlCheckParameters* oder *mdlInitializeSampleTimes* werden hier ausgeführt. Dies findet nur einmal zur Beginn der Simulation statt und wird wieder gestartet, nachdem die Simulation unterbrochen wurde. Im Detail wird hier folgendes ausgeführt:

- Die Initialisierung von *SimStruct* (siehe. 2.7)
- Das setzen der Anzahl und Weite von Input- und Output-Ports
- Die Einstellung der für den Block vorgesehenen sample time(s)
- Die Reservierung von Speicher für zum Beispiel Arbeitsvektoren

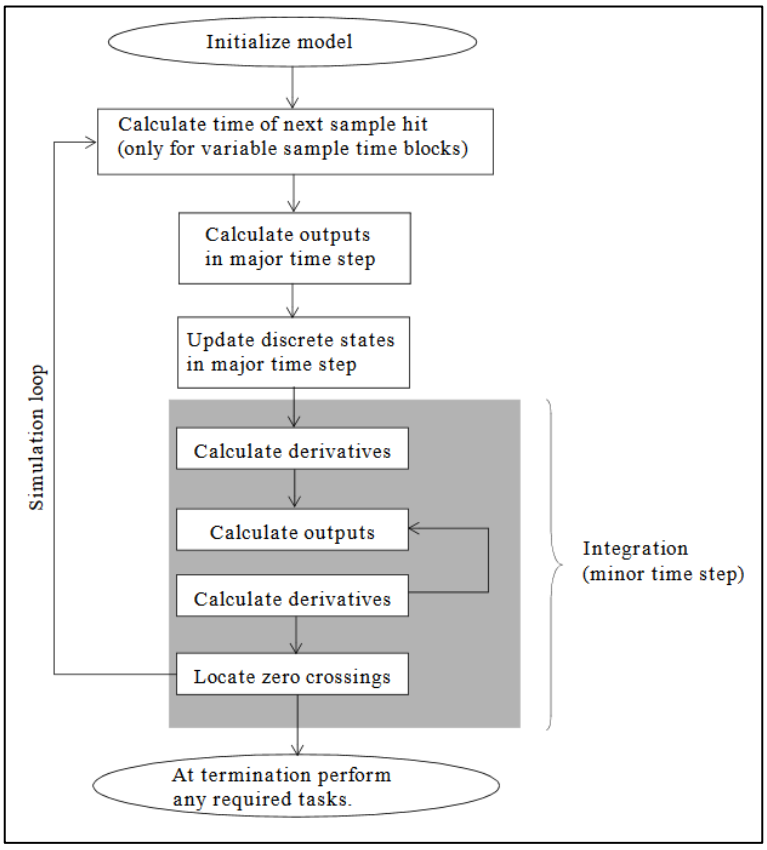

Abbildung 6: Arbeitsschritte einer Simulation

Daraufhin wird eine Simulationsschleife (*simulation loop*) aktiviert. Jeder Durchlauf dieser Schleife wird als Simulationsschritt bezeichnet (*simulation step*). Während jedes Simulationsschritts startet *Simulink* jeden Block in der Reihenfolge wie es zur Initialisierung festgelegt wurde. Für jeden Block werden dann die Zustände (*states*) und Outputs zur derzeitigen *sample time* berechnet.

Am Anfang dieser Schleife steht die Berechnung der Zeit für den nächsten *sample hit*. Dieser Schritt wird allerdings nur durchlaufen, wenn mit *ssSetSampleTime* eine variable Schrittweite eingestellt wurde.

Im nächsten Schritt werden die Outputs für die *major time steps* berechnet und daraufhin die *discrete states* aktualisiert. Die Integration in *minor time step* Intervallen findet nur dann statt, wenn der Block *continuous states* oder *zero crossings* aufweist und als solcher deklariert wurde (siehe 2.6).

2.10 Arbeitsschritte bezüglich einer C-MEX S-function

In welcher Form *Simulink* eine Simulation durchläuft wurde bereits geklärt. Dies lässt sich hinsichtlich der in dieser Arbeit verwendeten *C-MEX S-function* allerdings noch detaillierter und genauer betrachten. Die folgenden zwei Grafiken zeigen welche Callbacks in welcher

Reihenfolge durchlaufen werden und sollen unter anderem aufzeigen, welche Möglichkeiten eine *C-MEX S-function* bietet. Die für eine *S-function* notwendigen Funktionen sind geschlossen umrandet, die optionalen Callbacks gestrichelt.
Die Bedeutungen der einzelnen Callbacks und zu welchen anderen Zeitpunkten diese außerdem noch aufgerufen werden, kann man der Dokumentation auf mathworks.com entnehmen.

Abbildung 7 stellt den Initialisierungsprozess dar. Die in dieser Arbeit verwendeten Callbacks sind die notwendigen Funktionen *mdlInitializeSizes* und *mdlInitializeSampleTime*, sowie die optionalen Funktionen *mdlCheckParameters* und *mdlStart*.

Abbildung 8 zeigt die bereits vorgestellte Simulationsschleife. Im Zuge dieser Arbeit werden lediglich die *major time steps* von Interesse sein, da Integration oder *zero crossing detection* in diesem Fall nicht notwendig ist. Ein Großteil der Arbeit wird sich in *mdlOutputs* abspielen.

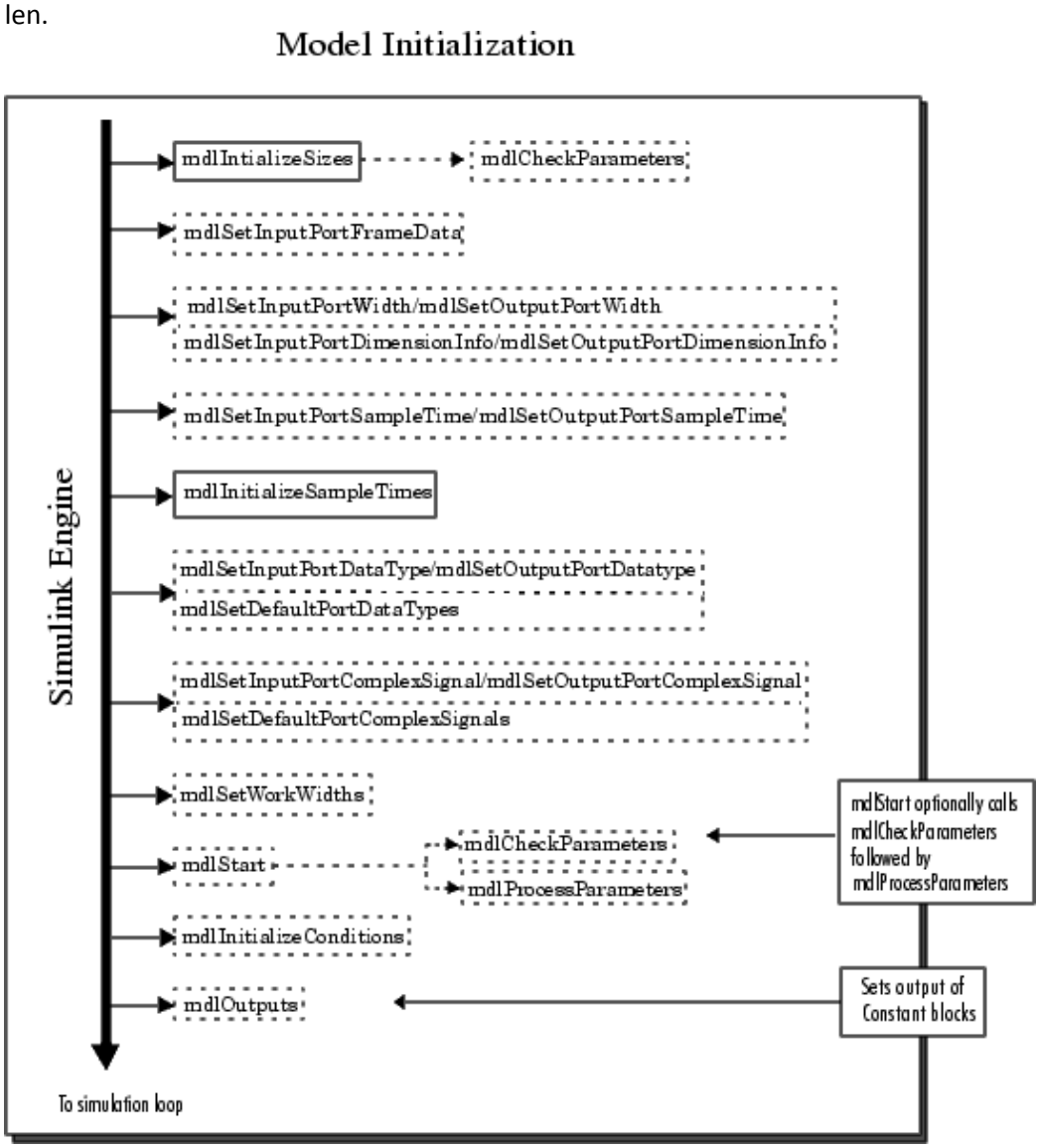

Abbildung 7: Arbeitsschritte einer C MEX S-function Teil 1

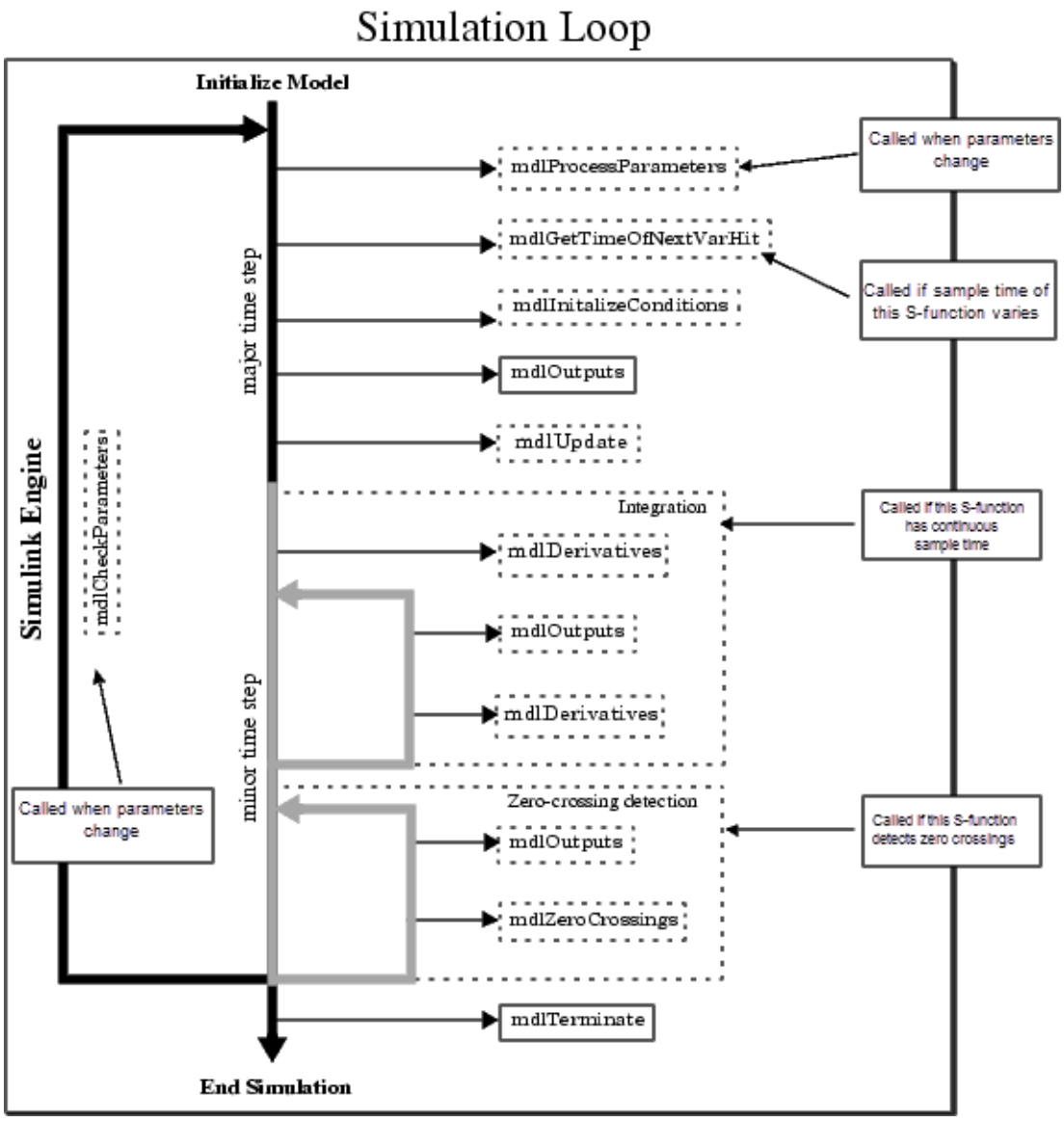

Abbildung 8: Arbeitsschritte einer C MEX S-function Teil 2

2.11 Maskieren und Speichern eines benutzerdefinierten Simulink-Blocks

Ist die *S-function* funktionsfähig, kann der die *S-function* geladene *Simulink*-Block über den *Mask Editor* individualisiert werden. Dies kann zum Beispiel genutzt werden, um die Input- und Output-Ports eines Blockes zu beschriften oder vordefinierte Parameter mitzugeben. Der Block kann dann in einer selbsterstellten Bibliothek gespeichert werden. Um Zugriff innerhalb von *Simulink* auf die Bibliothek und die darin selbsterstellten Blöcke zu haben, muss diese zusätzlich mit einem Skript versehen und vor dem Speichern der *MATLAB*-Befehl „set_param(gcs,'EnableLBRepository','on')“ ausgeführt werden. Liegen die Bibliothek und das zugehörige Skript in einem *MATLAB* hinzugefügten Pfad vor, kann dann auf diese innerhalb des *Simulink Library Browsers* zugegriffen werden.

3 Logitech Software Development Kit (SDK) und G27

In diesem Kapitel soll ein Überblick über die Spezifikationen des verwendeten *Logitech G27* Lenkrad geschaffen werden. Außerdem soll ein erster Blick auf das für die Programmierung notwendige *Logitech* SDK geworfen werden.

3.1 Allgemeine Beschreibung und Funktionalität des G27

Das *Logitech G27* (Abbildung 9) ist ein *Forcefeedback*-Lenkrad welches von *Logitech* für Rennsimulatoren/Spiele entwickelt wurde und stellt eine Weiterentwicklung des *G25* dar. Seit Dezember 2015 ist es aufgrund von neueren Modellen (*G920* und *G29*) nicht mehr offiziell im Handel verfügbar. Aufgrund seiner Haptik ist es jedoch für viele Simulationsfans immer noch die erste Wahl und wird immer noch in etwaigen Onlineportalen angeboten.

Es ist kompatibel mit dem PC, der *Playstation 2* und der *Playstation 3* und besitzt die folgenden Spezifikationen:

Lenkrad:

- Einen Durchmesser von ca. 270mm
- Einstellbare Lenkradrotation bis 900 Grad
- Zwei *Forcefeedback*-Motoren
- Ein Set Zahnräder mit *Antibacklash* Design
- Zwei Lenkrad-Schaltwippen
- Sechs Buttons

Abbildung 9: Logitech G27

Pedale:

- Gaspedal (leichte Federung)
- Kupplung (mittlere Federung)
- Bremse (starke Federung)

Schaltvorrichtung:

- Mit sechs Gängen und einem Rückwärtsgang
- Acht Buttons
- Ein D-Pad

3.2 Logitech Software Development Kit (SDK)

Das *Logitech* SDK stellt eine Erweiterung von *Microsoft's DirectInput* in *DirectX* dar. Es wurde entwickelt um Entwicklungszeiten und die Qualität der Integration verschiedener Controller zu verbessern. Unter anderem zielt es darauf ab eine gewisse Gleichheit zu gewährleisten, was das *Forcefeedback* der verschiedenen Controller betrifft. Die letzte aktualisierte und hier verwendete Version des SDK*s* ist von 2009 und wird bei jeder Installation der *Logitech Gaming Software* in jenem Ordner hinterlegt.
Das SDK enthält eine Sammlung an Source-Dateien und Implementierungsbeispielen zu Controllerinput, Controllermapping, Joysticks und Lenkrädern.

Die Verwendung des *Logitech* SDK*s* unterliegt folgenden Voraussetzungen:

- *Microsoft DirectX 9.0c* oder neuer
- *Visual Studio 2005* oder neuer
- Treiber für die verwendete Hardware
- Die Installation der *Logitech Gaming Software*

(*SteeringWheel* API Dokumentation)

4 Entwicklungsprozess

Dieses Kapitel soll einen Überblick über den Entwicklungsprozess geben. Das Projektsetup, sowie das Integrieren des *Logitech* SDK*s* in eine *S-function* sollen hier dargestellt werden. Dabei wird auf die Wahl bestimmter Lösungsansätze und die dahinterstehenden Überlegungen und Erfahrungen eingegangen.

4.1 Erster Test und Probleme des Logitech SDKs

Wie bereits erwähnt, sind dem *Logitech* SDK Implementierungsbeispiele beigelegt. Die Dateien *SteeringWheel*SDK*Demo.cpp* sowie *SampleInGameImplementation.cpp* sind dabei von besonderem Interesse. Die *SteeringWheel*SDK*Demo.cpp* ist außerdem als ausführbare *EXE*-Datei bereits verfügbar.

Ein Ausführen der genannten Datei öffnet ein Fenster, welches einen Überblick über die Werte von Achsen (Lenkrad und Pedalerie), D-Pad und Buttons liefert. Außerdem ist es möglich durch Drücken einiger Buttons voreingestellte Straßenmodi zu aktivieren und zu testen, sowie die Parameter des Lenkrades durch Get- und Setfunktionen zu verändern.

Da das SDK mit seiner letzten Aktualisierung von 2009 etwas älter ist, soll natürlich auch die Kompilierung auf dem Arbeitscomputer unter *Windows 7 Professional 64bit* und *Visual Studio 2015* getestet werden. Ein Öffnen des vorliegenden *Visual Studio* Projekts zeigt auf, dass dies nicht unter *Visual Studio 2015* läuft. *Visual Studio 2015* bietet glücklicherweise die Nützliche Funktion ältere Projekte automatisch zu migrieren. Die Migration verläuft ohne Probleme und der Migrationsreport gibt lediglich 9 Warnungen aber keinen Error aus. Der Code wird erfolgreich kompiliert, beim anschließenden Starten erscheint dennoch die folgende Fehlermeldung (Abbildung 10):

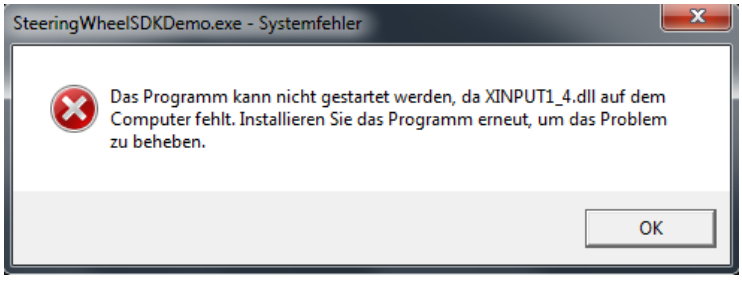

Abbildung 10: Fehlermeldung SDK *Demo*

Fehlende *DLL*-Dateien können in vielen Fällen nachgerüstet werden. Der erste Ansatz ist also die Datei aus dem Internet zu besorgen. Diverse Seiten bieten *DLL*-Dateien an, dabei

sollte man allerdings auf die Vertrauenswürdigkeit der Quellen achten, um sich unerwünschte Viren zu ersparen.
Ein Nachrüsten der Datei bringt trotzdem nicht den gewünschten Erfolg, da in der *DLL*-Datei andere Dateien verlinkt werden, welche in dem *DLL*-Verzeichnis nicht vorhanden sind.

Eine Suche nach der Datei zeigt auf, dass es sich hierbei um eine *cross-platform API* für neuere *XInput*-Geräte handelt, welche allerdings jeder Windows Version mitgegeben sein sollte. Bei *XInput*-Geräten handelt es sich vor allem um den *Xbox 360 Controller* für *Windows*. In diesem Fall wird jedoch die *XINPUT1_4.dll* benötigt, welche eine neuere Variante darstellt und erst mit *Windows 8* verschifft wird. Mit der letzten Aktualisierung des SDK*s* in 2009 und dem Erscheinen von *Windows 8* in 2012 ist dies jedoch etwas merkwürdig.

Der Fehler muss also im SDK Code selber liegen. Da das *G27* über *DirectInput* läuft und nicht über *XInput,* muss der Zugriff also an irgendeiner Stelle eingeschränkt werden. Es hilft nur die manuelle Suche. Bei der Suche innerhalb des Projekts erscheint innerhalb der *LogiControllerInput.cpp* Zeile 53 die vielversprechende Variable *g_ignoreXInputControllers* (Abbildung 11):

```
53      BOOL g_ignoreXInputControllers = FALSE;
```

Abbildung 11: g_ignoreXInputControllers LogiControllerInput.cpp

Das Ändern dieser bringt leider nicht den gewünschten Erfolg, da diese lediglich die Durchnummerierung angeschlossener *XInput*-Geräte unterbindet und trotzdem weiterhin zu der nichtexistierenden *DLL*-Datei verlinkt wird.
In der zugehörigen Header-Datei stößt man auf die folgenden Zeilen (Abbildung 12):

```
15  #pragma comment(lib,"dxguid.lib")
16  #pragma comment(lib,"dinput8.lib")
17  #pragma comment(lib,"xinput.lib")
```

Abbildung 12: Headerausschnitt LogiControllerInput.h

Das Ändern von *xinput.lib* zu der unter *Windows 7* vorhandenen *xinput9_1_0.lib* lässt die zuvor erschienene Fehlermeldung verschwinden, bringt aber zwei neue, auf den ersten Blick nicht aussagekräftige, *Linker Errors* zum Vorschein (Abbildung 13):

```
LNK2019 unresolved external symbol _XInputEnable@4 referenced in function "long __stdcall NewWindowProc(struct HWND__ *,unsigned int,unsigned
        int,long)" (?NewWindowProc@@YGJPAUHWND__@@IJ@Z)
LNK1120 1 unresolved externals
```

Abbildung 13: Linker Errors des SDK*s*

Die Lösung des Problems birgt jedoch der erste *Linker Error*. *XInputEnable* ist eine Funktion der unter *Windows 8* verwendeten *XINPUT1_4.lib* und wird genau zweimal innerhalb der *LogiControllerInput.cpp*, nämlich in Zeile 1485 und 1491, verwendet. Das Auskommentieren dieser bringt den gewünschten Erfolg und das SDK kann letztendlich erfolgreich kompiliert und ausgeführt werden.

4.2 Wrappen des SDKs als S-function

Ein Öffnen der *SampleInGameImplementation.cpp* zeigt auf, in welcher Weise das SDK normalerweise in ein Programm integriert wird. Es gibt zwei globale Variablen mit den Namen *g_wheel* und *g_controllerInput*, welche Pointer der Klassen *Wheel* und *ControllerInput* darstellen. Die Klasse *ControllerInpt* dient dazu *DirectInput*-Geräte zu verwalten und aufzulisten, sowie das Anschließen und Entfernen dieser zu verarbeiten. Über die Klasse *Wheel* lassen sich Funktionen des Lenkrades wie z.B. *PlaySpringForce* oder *PlayDamperForce* ansteuern. Außerdem können über diese Klasse Achsen und Buttons ausgelesen werden.
Der größte Teil findet dabei im Callback *MainDlgProc* statt (Abbildung 14):

```
36    INT_PTR CALLBACK MainDlgProc( HWND hDlg,
37                                  UINT msg,
38                                  WPARAM wParam,
39                                  LPARAM lParam )
40    {
```

Abbildung 14: MainDlgProc der SampleInGameImplementation.cpp

Dieses Callback muss in jeder Simulationsschleife durchlaufen werden, damit die Werte des Lenkrads ausgelesen und das entsprechende Verhalten des *Forcefeedbacks* gesteuert werden kann. Die im *MainDlgProc* mitgegebenen Parameter gehören zur Windowsprogrammierung und werden dafür genutzt Informationen zwischen Fenstern und *Windows* auszutauschen.
Um diese Prozesse nun mit dem geringsten Aufwand in eine *S-function* zu integrieren, ist der erste Schritt, alles, was in diesem Callback passiert (bis auf den Terminierungsprozess), in die *mdlOutputs*-Funktion einer *S-function* zu *wrappen*. Natürlich ohne die für einen Windowsfensterprozess wichtigen Variablen, denn die Initiative soll nicht an das Öffnen des Modellfensters, sondern an das Starten der Simulation gebunden sein. Dies wäre als *S-function* auch gar nicht möglich.
Für die Erstellung der *S-function* wird die Datei *g27SimulinkIntegration.cpp* erzeugt. Die von der *S-function* benötigten Parameter werden zur Übersicht bereits in Zeile 16-37 als *defines* deklariert. Abbildung 15 stellt dies für die Lenkradeinschlag und *Forcefeedback* Parameter beispielhaft dar:

```
16    #define WHEEL_RANGE_IDX 0
17    #define WHEEL_RANGE_PARAM(S) ssGetSFcnParam(S,WHEEL_RANGE_IDX)
18    #define FORCE_ENABLE_IDX 1
19    #define FORCE_ENABLE_PARAM(S) ssGetSFcnParam(S,FORCE_ENABLE_IDX)
```

Abbildung 15: Parameter Define Sample in g27SimulinkIntgetration.cpp

Insgesamt besitzt die *S-function* 11 Parameter. Dabei handelt es sich um die Parameter *Wheel_Range, Force_Enable, Overall_Gain, Overall_Spring_Gain, Overall_Damper_Gain, Combine_Pedals, Wheel_Linearity, Speed_Divisor, Rpm_LedStart, Rpm_LedRed und Set_Preferred*. Diese müssen der *S-function* zur Verfügung stehen. Die genaue Funktion der Parameter wird in Kapitel 5 behandelt.
Gemäß der in 2.7 vorgestellten Beispielfunktion werden die Anzahl und Weite aller Outputs und Inputs, sowie die Nummer der Parameter, in der *mdlInitializeSizes*-Methode der *S-function* definiert. Um die Parameter während der Simulation verändern zu können, werden diese über die *ssSetSFcnParamTunable*-Funktion bestimmt. Darunter fallen alle Para-

meter bis auf *Force_Enable*, da es bei einer Veränderung dieser während der Laufzeit zu Problemen kommt.
Die Objekte *g_controllerInput* und *g_wheel* werden, entgegen der von *Simulink* empfohlenen Verwendung von *pWork*-Vektoren, global deklariert. Dies bringt einen kleinen Performance Boost, da diese Objekte dadurch nicht zwischen den einzelnen *S-function*-Methoden über die Verwendung von Makros hin und her gereicht werden müssen. Ein Nachteil, der mit dieser Methode einhergeht, ist, dass nicht mehrere Lenkräder gleichzeitig in einem Modell verwendet werden können. Aufgrund der Architektur und der von *Simulink* verwendeten *Execution Order* würden die Blöcke, falls mehrere *S-functions* in einem Modell verwendet werden, nacheinander das Lenkrad über die gleichen Objekte ansteuern. Da aber zu Simulations- oder Testzwecken sowieso nicht mehr als ein Lenkrad in Betrieb sein wird, ist dies ein Manko, welches toleriert werden kann.

4.3 Problematiken beim Wrappen

Aufgrund der Programmierung des SDK*s* muss das Objekt *g_controllerInput* jedoch trotzdem an einen Fensterprozess geheftet werden. Die naheliegendste Option ist es, dafür das Fenster des *Simulink*-Modells zu verwenden.

Dabei ergeben sich zwei Probleme für die Integration. Die erste Problematik ergibt sich bei dem Versuch die *S-function* zu debuggen, die Zweite mit der Intention das Modell zu Simulationszwecken an ein weiteres Programm anzuknüpfen. Wird nämlich das Fenster gewechselt, das heißt, wird irgendein Fenster außer dem Hauptfenster des *Simulink*-Modells angeklickt, wird das Lenkrad auf seine Standardeinstellungen zurückgesetzt und ist nicht mehr ansprechbar für jegliche *Forcefeedback*-Befehle. Da sich das gleiche Phänomen bei der *SteeringWheel*SDK*Demo* beobachten lässt, kann ein Fehler diesbezüglich in der *S-function* mit großer Wahrscheinlichkeit ausgeschlossen werden.
Hinsichtlich der Intention des SDK*s* macht diese Funktion Sinn, denn Programme wie Rennsimulatoren finden auch nur innerhalb eines Fensters statt.

Die Problematiken beim Debuggen ergeben sich wie folgt:

Zum Debuggen einer *S-function* muss wie in 2.4 beschrieben, eine *PDB*-Datei erzeugt werden. Das Debugging Tool von *Visual Studio* wird dann an das bereits geöffnete *Simulink*-Modell geheftet. Wie beim Debuggen üblich können in den Source-Dateien Breakpoints zum Überprüfen, Überwachen oder zur Fehlerfindung eingesetzt werden. Wird nun ein Breakpoint erreicht, wechselt der Fokus vom *Simulink*-Modell zum Debuggingfenster von *Visual Studio*. Das Lenkrad wird über den *Logitech Event Monitor* auf seine Standardeinstellungen zurückgesetzt und nimmt keine *Forcefeedback*-Befehle mehr an, während die Simulation von *Simulink* weiterläuft. Dies produziert unerwünschte Komplikationen und Fehler.
Aufgrund dieser Eigenschaft ist das Debugging dieses Verhaltens ein nicht zu verachtender Aufwand.

4.3.1 Forcefeedback und DirectInput

Da *Forcefeedback* im Grunde durch die *DirectInput* Schnittstelle kontrolliert wird, macht es Sinn, sich die *DirectInput* Dokumentation genauer anzuschauen.
Ein Durcharbeiten der *DirectInput* Dokumentation bringt die Methode *IDirectInputDevice8::SetCooperativeLevel* zum Vorschein. Mit dieser Methode lässt sich die Art und Weise einstellen, in welcher das Programm mit anderen zugehörigen Instanzen oder dem restlichen System interagiert. Als Argumente nimmt die Methode einmal die zugehörige *handle* zum Fenster des Prozesses und weitere zum Einstellen des *CooperativeLevels.* Die Suche nach dieser Methode innerhalb des *Logitech* SDK*s* verläuft erfolgreich. Innerhalb der *LogiControllerInput.cpp* taucht die Methode unter Zeilen 491-492 auf. Schaut man sich diese an, fällt das Argument *DISCL_FOREGROUND* auf (Abbildung 16).

```
hr_ = devicesInfo[ll].device->SetCooperativeLevel( m_gameHWnd, DISCL_EXCLUSIVE |
    DISCL_FOREGROUND );
```

Abbildung 16: DirectInput-Funktion SetCooperativeLevel in LogiControllerInput.cpp

Dieses bewirkt, dass das Lenkrad automatisch „aufgegeben“ wird, wenn das Fenster des Prozesses in den Hintergrund rückt. Innerhalb des SDK Codes würde es dann wiederaufgenommen werden, sobald das Fenster der zuvor übergebenen *handle* wieder in den Vordergrund rückt. Dies entspricht allerdings nicht dem gewünschten Verhalten. Ein Ändern dieser Variablen auf *DISCL_BACKGROUND* lässt das Lenkrad auch im Hintergrund ansprechbar für *DirectInput*-Befehle und ist im Sinne der Integration.

4.3.2 Lenkradparameter und Window-Callbacks

Die zweite Problematik, nämlich das Zurücksetzen des Lenkrades auf dessen Standardeinstellungen beim Wechseln des Fensters ist, in dieser Hinsicht, um einiges hartnäckiger. Da es zu diesem Verhalten keine Dokumentation gibt, bleibt nichts Anderes übrig, als sich durch den Code des SDK*s* zu arbeiten und erst einmal festzustellen, an welchen Stellen etwas mit den *Properties* geschieht. Etwas Licht ins Dunkel bringt die Update-Funktion der *LogiControllerProperties.cpp* in Zeilen 400-447. Diese wird innerhalb der Updateschleife der *LogiWheel.cpp* in jedem Iterationsschritt aufgerufen und sendet *setMessages* an die *Logitech Gaming Software* für jeden Lenkradparameter dessen Variable *g_setIsNecessary* auf *TRUE* steht. Die *Logitech Gaming Software*, welche in diesem Fall eine Blackbox darstellt, wird im Konstruktor der *LogiControllerProperties.cpp* als *handle* mit dem Namen *_hEventMonitor* hinterlegt. Bleibt die Frage wo die Variablen *g_setIsNecessary* beeinflusst werden. Innerhalb der selbigen *CPP*-Datei findet sich in den Zeilen 460-516 die Funktion *NewSteeringWheel*SDK*WindowProc.* Diese stellt eine *Windows* eigene Operation dar.

Um zu verstehen wie diese genau in das Programm eingreift, soll die Funktionalität kurz angerissen werden. Ob direkt sichtbar oder nicht, jedem Programm in *Windows* steht ein Fenster zu Verfügung. Soweit sichtbar, können diese zum Beispiel geschlossen, minimiert, verkleinert oder auf Vollbild geschaltet werden. Jede Interaktion mit einem Windows Fenster ruft dessen Callback-Funktion auf, welche dann Nachrichten über den Zustand bzw. bevorstehenden Zustand des Fensters an das jeweilige Programm weiterleitet noch bevor

die Änderung in Kraft tritt. Dabei nimmt die Callback-Funktion vier Parameter. Zum einen die schon vorher erwähnte *window handle*, die Nachrichtenkennung *UINT,* sowie zwei weitere Datentypen *WPARAM* und *LPARAM*. *WPARAM* und *LPARAM* stellen dabei *typedefs* für *UINT_PTR* und *LONG_PTR* dar. Unter einem 32bit System stehen diese für *unsigned int* und *long* Datentypen. Unter einem 64bit System für *unsigned __int64* und *__int64*. Über diese können dann anwendungsspezifische Informationen übergeben werden. (*Windows Procedures* Dokumentation)
Wird nun das Fenster gewechselt, wird die *WM_ACTIVATEAPP* Nachricht an beide Fenster, nämlich dem Neuen und dem Alten gesendet. Innerhalb des *Window*-Callbacks *NewSteeringWheel*SDK*WindowProc* wird dann über *wParam* festgestellt, ob die Variable *activateValue* auf *TRUE* gesetzt werden muss. Die *wParam* Information wird dabei höchstwahrscheinlich von dem *Logitech Event Monitor* der *LGS* übergeben. Wird die Variable *activateValue* auf *TRUE* gesetzt, wird durch alle Lenkradparameter iteriert und deren *g_setIsNecessary*-Variablen auf *TRUE* gesetzt. Sind diese Variablen auf *TRUE* gesetzt, aktualisiert die *Logitech Gaming Software* die Lenkradparameter. Damit dies in jedem Fall eintritt, war der erste Versuch die Variable *activateValue* dauerhaft auf *TRUE* zu setzen und so bei jedem Fensterwechsel die Übernahme der Parameter zu provozieren. Dies löst das Problem allerding nur bedingt. Abbildung 17 soll den Sachverhalt verdeutlichen:

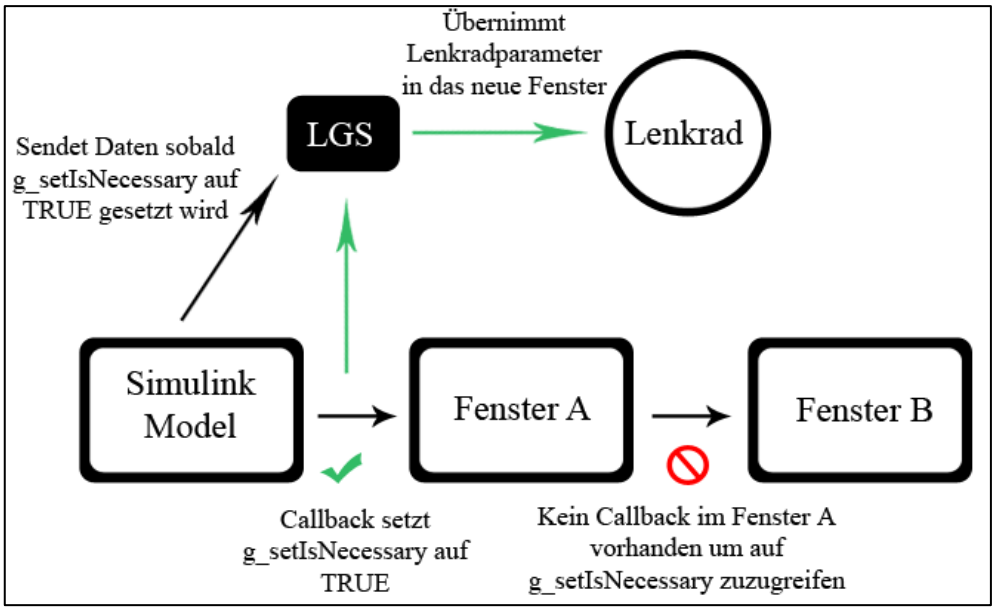

Abbildung 17: Window-Callback-Problematik

Wechselt man aus dem *Simulink* Model in ein beliebiges Fenster, wird das Callback der *S-function* aktiviert und durch die geänderte Variable *activateValue* alle *g_setIsNecessary*-Variablen auf *TRUE* gesetzt. Die *LGS* wird durch eine *SendMessage* (*Windows*-Funktion) angesprochen und übernimmt die Parameter in das neue Fenster für das Lenkrad. Wechselt man aber nun von dem neuen „Fenster A“ in ein weiteres „Fenster B“, oder ist man zufällig den Weg über den Desktop oder „Shift+Tab“ gegangen, existieren für diese Fenster keine Callbacks mehr um mit der *Logitech Gaming Software* zu kommunizieren. Wie zuvor erwähnt erhalten nämlich nur das neue und vorherige Fenster die Meldungen eines bevorstehenden Fensterwechsels. Da dies äußerst unkomfortabel und fehleranfällig ist, muss

eine andere Lösung, beziehungsweise eine Erweiterung gefunden werden. Da die *Logitech Gaming Software* quasi eine Blackbox darstellt und das Herausnehmen der Kommunikation mit selbiger ein Umschreiben des ganzen SDK*s* zur Folge hätte, wurde eine andere Lösung gewählt. Dafür wurde der *LogiControllerProperties.cpp* eine neue öffentliche Funktion mit dem Namen *SetG_SetIsNecessaryTRUE* gegeben (Abbildung 18).

```
450  VOID ControllerProperties::SetG_SetIsNecessaryTRUE()
451  {
452      std::map <DWORD, BOOL>::iterator itr_;
453      for (itr_ = g_setIsNecessary.begin(); itr_ != g_setIsNecessary.end(); itr_++)
454      {
455          g_setIsNecessary[itr_->first] = TRUE;
456      }
457  }
```

Abbildung 18: SetG_SetIsNecessaryTRUE-Funktion in LogiControllerProperties.cpp

Diese kann dann innerhalb der Updateschleife des Lenkrads über das gleichzeitige Drücken des rechten Wippschalters und des rechten mittleren roten Buttons am Lenkrad (Button 4 und 19) aufgerufen werden und wurde innerhalb der *Logiwheel.cpp* in Zeile 183-186 so eingefügt (Abbildung 19).

```
183          if (ButtonIsPressed(index_, 19) && ButtonIsPressed(index_, 4))
184          {
185              m_controllerProperties->SetG_SetIsNecessaryTRUE();
186          }
```

Abbildung 19: SetG_SetIsNecessaryTRUE-Funktion Implementierung in LogiWheel.cpp

Wechselt man nun in ein neues Fenster und stellt fest, dass die Parameter nicht übernommen wurden, kann durch diese Tastenkombination die Kommunikation mit der *LGS* provoziert werden. Diese stellt dann die Lenkradparameter in dem neuen Fenster zur Verfügung.

4.3.3 Struktur und Terminierungsprozess

Um möglichst nah an der für das SDK intentionierten Struktur zu bleiben, wird ein *switch case* erzeugt mit den Status *kInit* und *kUpdate*. *kInit* wird nach dem ersten Start der Simulation durchlaufen, dann auf *kUpdate* gesetzt und danach nicht mehr verändert. In *kInit* werden dabei die *g_wheel*- und *g_controllerInput*-Objekte erzeugt und die Lenkradparameter vorgegeben. Ein Unterbrechen und erneutes Starten der Simulation würde nun die außerhalb der Simulation eingestellten Lenkradparameter unberührt lassen. Um dies zu umgehen, wird auf die *S-function*-Methode *mdlStart* zurückgegriffen. Existiert das Objekt *g_wheel* bereits, werden in *mdlStart* die Parameter neu gesetzt.

Bevor dies allerdings passieren kann, muss die Simulation ohne Probleme unterbrochen und neu gestartet werden können. Auf den ersten Blick ist es naheliegend die in der *SampleInGameImplementation.cpp* vorliegende Methode zum Zerstören von den Objekten *g_wheel* und *g_controllerInput* innerhalb der *S-function*-Methode *mdlTerminate* zu benutzen. Nach dem Stoppen und anschließendem Neustarten der Simulation, würden diese Objekte dann innerhalb von *mdlStart* oder innerhalb von *mdlOutputs* wieder erzeugt wer-

den. Der Versuch einer solchen Implementierung führt allerdings beim Neustarten der Simulation zu Speicherzugriffsverletzungen und letztendlich zum Absturz von *MATLAB* oder dem Rechner.
Selbst ein Verwenden der von *Simulink* zur Verfügung gestellten *pWork*-Vektoren hilft nicht das Problem zu beheben. Der Fehler hierbei muss an der *Logitech Gaming Software* liegen, welche bei einem Terminierungsprozess höchstwahrscheinlich von einer Fensterschließung ausgeht. Bestätigen lässt sich diese Vermutung durch das Schließen und anschließende Öffnen des Models nach dem Stoppen der Simulation. Hält man sich an diese Reihenfolge würden mit dieser Implementierung keine Fehler auftreten, was aber hinsichtlich der Benutzerfreundlichkeit in *Simulink* große Abzüge mit sich bringen würde.
Umgehen lässt sich dieses Problem, indem man die Objekte einfach nicht löscht, heißt also, den Speicher nicht freigibt. Bei einem Neustart der Simulation, sowie einem Schließen und Wiederöffnen des Modells, würden die Objekte dann noch bestehen. Da durch den *switch case* die Initialisierung der Objekte nicht ein weiteres Mal durchlaufen wird, treten allerdings keine Probleme auf. Ein Löschen, des der *MEX function* zur Verfügung gestellten Speichers über „clear MEX*",* hätte dann eine erneute Initialisierung zur Folge. Da *MATLAB* beim Schließen den zuvor verwendeten Speicher automatisch freigibt, kann man das in diesem Fall so machen. Dieser Ansatz ist zwar nicht empfehlenswert, lässt sich aber aufgrund der Kommunikation mit der *Logitech Gaming Software* nicht anders umgehen.

4.3.4 Problematik bei Anschluss mehrerer Geräte

Da die *Logitech Gaming Software* durchaus in der Lage ist mehrere angeschlossene Geräte über das *ControllerInput*-Objekt zu verwalten, wird hier im auf den ersten Blick kein Fehler vermutet. Der Versuch bei dem Anschluss von mehreren Controllern auf das Lenkrad über *Indexing* zuzugreifen funktioniert zwar auch, allerdings nicht bedingungslos. Bei einem Index größer Null führt die *PlaySpringForce*-Funktion der *Wheel* Klasse aus vorerst unersichtlichen Gründen zu Speicherzugriffsverletzungen. Genauer in der *GetType*-Funktion der *Force* Klasse.
Da der Fehler nicht auf Anhieb ersichtlich war, wurden mehrere Lösungsansätze getestet. Als Notlösung dient zuerst einmal nur Geräte mit *Forcefeedback* zuzulassen. Dies lässt sich über die Option *DIEDFL_FORCEFEEDBACK* innerhalb der *EnumDevices*-Methode (Zeile 185-187, *LogiControllerInput.cpp*), welche Teil der *DirectInput* Schnittstelle darstellt, einstellen. Allerdings würde dies natürlich trotzdem andere *Logitech Controller* mit *Forcefeedback* listen. Eine weitere Möglichkeit ist, mit einem *Counter* die Zahl der angeschlossenen Geräte zu überprüfen und dem Benutzer gegebenenfalls mitzuteilen diese zu entfernen. Diese beiden Herangehensweisen zusammen decken alle Fälle ab, sind jedoch teilweise umständlich für den Anwender.
Eine bessere Möglichkeit findet sich in der *PopulateDeviceDescriptions*-Methode der *ControllerInput*-Klasse. Diese findet sich auch in der *LogiControllerInput.cpp*. In dieser Methode werden die Informationen der einzelnen Geräte in einen *DEVICE_INFO_VECTOR* geladen. Um sicherzustellen, dass nur das *G27* angesprochen wird, sind folgende Zeilen hinzugefügt worden (Abbildung 20):

```
317         if (deviceInfo_.vid == VID_LOGITECH && deviceInfo_.pid == PID_G27)
318         {
319             devicesInfo.push_back(deviceInfo_);
320         }
```

Abbildung 20: G27 Produkt ID Abfrage in LogiControllerInput.cpp

Dies erspart dem Benutzer den Aufwand möglicherweise Geräte entfernen zu müssen und lässt die Notwendigkeit der vorigen Notlösungen entfallen.

An einem späteren Zeitpunkt des Entwicklungsprozesses fiel auf, dass innerhalb der *LogiWheelGlobals.h* die Variable *LG_MAX_CONTROLLERS* in Zeile 28 zu einem früheren Zeitpunkt von 2 auf 1 gesetzt wurde. Diese hat letzten Endes die Speicherzugriffsverletzungen provoziert. Dass die Speicherzugriffsverletzung erst in der *Force* Klasse auftrat, liegt daran, dass es zwei *LG_MAX_CONTROLLERS*-Variablen gibt, wobei die Variable der *LogiWheelGlobals.h* erst innerhalb der *Force* Klasse verwendet wurde. Unabhängig davon bleibt die Filterung von Geräten abseits des *G27* und der Zugriff auf den ersten Index, im Hinblick auf die geforderte Funktionalität und mögliche Fehleranfälligkeit, weiterhin eine gute Lösung.

4.4 Anwendungsfehler und Fehlerüberwachung

4.4.1 Initialisierungsfehler

Durch den Benutzer verursachte Initialisierungsfehler können durch folgende falsche Anwendungen verursacht werden:

- Starten der ersten Simulation nach Öffnung des Modells außerhalb des Hauptfensters. Zum Beispiel, indem man die Simulation aus einem *Scope*-Fenster startet.
- Starten der ersten Simulation nach Öffnung des Modells ohne das Lenkrad per USB oder an das Stromnetz angeschlossen zu haben.

Aktionen dieser Art führen mit der bisher verwendeten Struktur (siehe 4.3.3) bei einem erneuten Versuch des Startens der Simulation zu Speicherzugriffsverletzungen und dem Absturz von *MATLAB* oder ggfls. des Rechners. Startet der Benutzer die Simulation bei der ersten Ausführung aus einem anderen Fenster als dem Hauptfenster des *Simulink* Modells, wird eine unzureichende *window handle* übergeben. Die Folge ist, dass die Objekte *g_controllerInput* und somit *g_wheel* nicht erfolgreich erstellt werden können. Den selben Effekt hat auch das erstmalige Starten mit einem unzureichend angeschlossenen Lenkrad (kein Strom oder USB). In diesem Fall wird *g_controllerInput* aufgebaut, allerdings ohne das Lenkrad als *DirectInput*-Gerät zu hinterlegen. Das *Logitech* SDK bietet zwar die Möglichkeit einer *hot plug*-Methode, allerdings wurde im Hinblick auf die Kompatibilität mit *Simulink* und der Fehlermeldung darauf verzichtet.
Um diese Fälle abzufangen wird kurz nach der Initialisierung überprüft, ob das Lenkrad über die SDK Software angesprochen werden kann. Ist dies nicht der Fall wird der *g27SimulationState* gar nicht erst auf *kUpdate* gesetzt. Die Objekte werden gelöscht und der *g27SimulationState* auf *kInitFailed* gesetzt. Dies verhindert einen erneuten Initialisie-

rungsversuch, welcher, aufgrund der Architektur und der Kommunikation mit der *Logitech Gaming Software*, einen Absturz von *MATLAB* mit sich bringen würde (siehe 4.3.3). Folgende Fehlermeldung wird in dem Fall über die *S-function*-Methode *ssSetErrorStatus* ausgegeben (Abbildung 21):

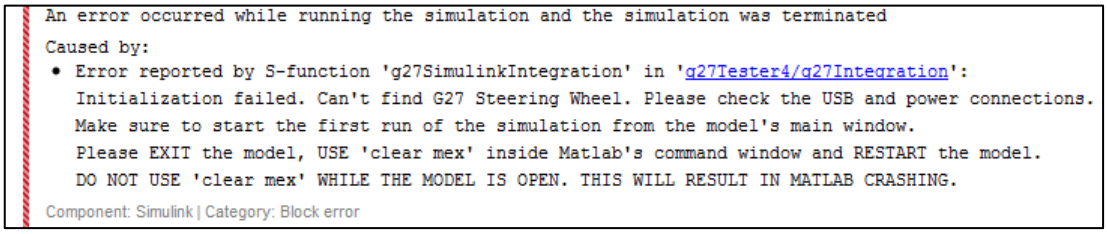

Abbildung 21: Fehlermeldung bei Initialisierungsfehler

Der Benutzer wird gebeten das Modell zu schließen, über das *MATLAB command window* den Befehl „clear MEX“ einzugeben und das Modell anschließend neu zu starten. Ein Fehlverhalten die Simulation einfach neu zu starten wird durch den *kInitFailed case* ebenfalls abgefangen, welcher in diesem Fall wieder die obige Fehlermeldung ausgibt. Was nicht abgefangen werden kann, ist, dass der Anwender „clear MEX“ eingibt während das Modellfenster noch geöffnet ist. Dies resultiert unausweichlich mit dem Absturz von *MATLAB*, weswegen in der Fehlermeldung darauf ausdrücklich hingewiesen wird. Es existiert zwar die Möglichkeit über die *MATLAB*-Funktion *mexLock()* ein Löschen der *MEX*-Funktion zu verhindern, die Freigabe dieser müsste allerdings an das Schließen des Simulationsfensters gebunden sein, auf welches die *S-function* keinen Einfluss mehr hat, da sie zu diesem Zeitpunkt schon gar nicht mehr ausgeführt wird. Erreichen lässt sich dies über die Modelleigenschaften, indem *mexLock()* und *mexUnlock()* über die *PreLoadFcn*- und *CloseFcn*-Callbacks aufgerufen werden.
Als alternativer Ansatz wurde getestet *MATLAB* die *S-function* im Falle eines Initialisierungsfehlers selbst schließen zu lassen. Dies ist möglich über das Erstellen und Aufrufen einer *timer*-Funktion. Diese würde im Falle eines Initialisierungsfehlers das im Vordergrund liegende Modell, in einem gewissen Abstand zum Terminierungsprozess, über den *MATLAB*-Befehl „close_system“ automatisch schließen. Ignoriert der Benutzer jedoch die angezeigte Fehlermeldung und startet das Modell in diesem Zeitraum neu, kommt es zum Absturz von *MATLAB*. Um gegen solch ein Fehlverhalten vorzugehen, ist die einzige Möglichkeit in diesem Fall, die Zeitspanne vom Terminierungsprozess bis zur Schließung des Fensters so gering wie möglich zu halten (Ein Sperren des Simulationsstarts kann nach derzeitigen Erkenntnissen nicht verhindert werden). Da die benötigte Zeit zum Abschließen des Terminierungsprozesses jedoch von Rechner zu Rechner variiert, wurde diese Methode als zu fehleranfällig eingestuft.

4.4.2 Entfernen der Hardware nach erfolgreicher Initialisierung

Ein Unbeabsichtigtes Entfernen des *G27* während der Simulation wird innerhalb der *g27SimulinkIntegration.cpp* in Zeile 565 über die folgende Abfrage überwacht (Abbildung 22):

```
if (g_wheel != NULL && g_wheel->IsConnected(index, LG_MODEL_G27))
```

Abbildung 22: G27 Anschluss Kontrolle in g27SimulinkIntegration.cpp

Wird diese Bedingung nicht erfüllt, erhält der Benutzer die Fehlermeldung, dass das Gerät nicht gefunden wurde und möglicherweise die USB Verbindung oder die Stromzufuhr unterbrochen wurde. Die Simulation wird dabei unterbrochen. Ein erneutes Anschließen des *G27* erfordert in diesem Fall keinen Modell Neustart.

Zu Beginn der Integration wurde dem *g_controllerInput*-Objekt die *window handle* von dem *Simulinkfenster* über die *Windows*-Funktion *getActiveWindow()* übergeben. Dies führt nun allerdings zu Problemen und hängt mit den unter 4.3.2 angesprochenen Callback-Funktionen zusammen, welche an das Simulationsfenster „geheftet“ werden. Wird ein *DirectInput*-Gerät angeschlossen oder entfernt, schickt *Windows* diese Meldung an alle Hauptfenster und das zu diesem Zeitpunkt aktive Fenster. Befindet der Benutzer sich nun in einem anderen Fenster als dem Modell Hauptfenster, erreicht die Meldung die *S-function* nicht, da das Fenster des Modells nicht als Hauptfenster erkannt wird. Das *g_controllerInput*-Objekt erhält in diesem Fall keine Mitteilung darüber und kann auch so die geführte Liste der angeschlossenen Geräte nicht aktualisieren. Letztendlich wird das Entfernen des *G27* in diesem Fall also nicht erkannt und auch keine Fehlermeldung ausgegeben. Glücklicherweise lässt sich dies durch einen kleinen Eingriff beheben. Anstatt dem *g_controllerInput* die *handle* des Simulationsfensters zu übergeben, wird gleich die *handle* zu *MATLAB* übergeben. Erreichen lässt sich dies über eine Verschachtelung der *GetActiveWindow()* Funktion in die *GetAncestor(GetActiveWindow(), GA_ROOT)* Funktion. Mit der Option *GA_ROOT* geht diese Funktion dann die *parent windows* des Simulationsfensters durch, bis es bei der *root* angekommen ist. Provoziert man nun den vorigen Fehler erhält *MATLAB* die Meldung, dass das Gerät entfernt wurde, woraufhin die *S-function* diese verarbeitet. Erfreulicherweise bringt dies keine weiteren Problematiken mit den *Window*-Callbacks oder dem *Logitech Event Monitor* der *LGS* mit sich, da das *MATLAB* Hauptfenster als *parent* des *Simulink*-Modells eine tragbare *handle* darstellt.

Eine weitere Fehlerquelle birgt das Entfernen und erneute Anschließen des Lenkrades während die Simulation gestoppt ist. Über die *window process* Mechanik werden dann nämlich zwei Nachrichten verarbeitet. Zum einen, dass entsprechende Hardware entfernt wurde und zum anderen, dass ein Gerät angeschlossen wurde. Bei einem erneuten Simulationsstart wird innerhalb der *Update*-Funktion des *g_controllerInput* Objekts dann erst einmal versucht ein neues Gerät hinzuzufügen. Dies scheitert, da der Controller bereits existiert. Im Anschluss wird das Entfernen der Controller verarbeitet, wodurch die *S-function* letztendlich feststellt, dass kein Gerät vorhanden ist, obwohl das Lenkrad angeschlossen ist. Tritt dies einmal auf, wird das *G27* bei der derzeitigen Struktur der Überprüfung nicht mehr erkannt werden. Um diesen Fehler zu umgehen wird die *AddNewControllers*-Funktion der *LogiControllerInput.cpp* als *public* deklariert, damit diese innerhalb der *S-function* verwendet werden kann. Anstatt bei nicht vorhandenem *G27* einfach eine Fehlermeldung auszugeben, wird nun der Code in Abbildung 23 durchlaufen. Der erneute Durchlauf der *AddNewControllers*-Funktion, bevor der endgültigen Abfrage des Lenkrades und der anschließenden Ausgabe eines Fehlers, verhindert das beschriebene Phänomen erfolgreich.

```
820             HRESULT ret_ = S_OK;
821             if (ret_ != g_controllerInput->AddNewControllers(FALSE))
822             {
823                 ssSetErrorStatus(S, "Trying to add new controllers failed.");
824                 return;
825             }
826
827             if (!(g_wheel != NULL && g_wheel->IsConnected(index, LG_MODEL_G27)))
828             {
829                 ssSetErrorStatus(S, "Can't find G27 Steering Wheel. Please Check the USB and Power connections.");
830                 return;
831             }
```

Abbildung 23: G27 Produkt ID Abfrage in LogiControllerInput.cpp

4.4.3 Monitoring und Setzen der Parameter

Damit die zur Verfügung stehenden Lenkradparameter nicht überschritten und vor Falscheingaben geschützt sind, müssen diese zur Initialisierung, sowie während der Simulation, überprüft werden.
Die Überprüfung zur Initialisierung geschient noch bevor der eigentlichen Erstellung der für das Lenkrad wichtigen Objekte, in der *S-function*-Methode *mdlCheckParameters*. Diese wird direkt nach der *mdlInitializeSizes*-Methode aufgerufen. Jeder der zur Verfügung stehenden Parameter wird dabei auf die Einhaltung seines Toleranzbereichs, sowie auf seinen Typ überprüft. Um die Typenprüfung zu erleichtern wird folgender *define* verwendet (Abbildung 24):

```
10      #define IS_PARAM_DOUBLE(pVal) (mxIsNumeric(pVal) && !mxIsLogical(pVal) &&\
11      !mxIsEmpty(pVal) && !mxIsSparse(pVal) && !mxIsComplex(pVal) && mxIsDouble(pVal))
```

Abbildung 24: IS_PARAM_DOUBLE Define in g27SimulinkIntegration.cpp

Die zur Überprüfung verwendeten Funktionen sind Teil von *MATLAB* und erleichtern in diesem Fall die Arbeit. Durch die Verwendung von *IS_PARAM_DOUBLE* wird die Eingabe von komplexen, logischen oder leeren Datentypen und Strings somit abgefangen.

Damit die Parameter auch während der Laufzeit abgefragt und kontrolliert werden können muss auf zwei weitere *S-function*-Methoden zurückgegriffen werden. Zuallererst müssen die Parameter, welche einstellbar sein sollen, innerhalb der *mdlInitializeSizes*-Methode über die Funktion *ssSetSFcnParamTunable* als solche aktiviert werden.
Damit diese auch während der Simulation überprüft werden kann, bedarf es der Methoden *mdlSetWorkWidths* und *mdlProcessParameters*. Die Namen, der zur Laufzeit zu überprüfenden Parameter, werden in ein Array des Typs *char_T* (welcher ein *typedef* des Typs *char* darstellt) geschrieben und der Funktion *ssRegAllTunableParamsAsRunTimeParams* übergeben. Diese registriert, wie der Name bereits andeutet, alle einstellbaren Parameter als Laufzeitparameter. In der Methode *mdlProcessParameters* wird mithilfe der *ssUpdateAllTunableParamsAsRunTimeParams*-Funktion dann eingestellt, dass alle Parameter welche mit der *ssRegAllTunableParamsAsRunTimeParams*-Makro erstellt wurden, aktualisiert werden sollen. Dabei wird zum Checken der Parameter in jeder Simulationsschleife, in der die Parameter verändert wurden, die *mdlCheckParameters*-Methode aufgerufen, um diese zu überprüfen.

Weiterhin gibt es den Fall, dass die *Logitech Gaming Software* nicht installiert ist oder läuft. Normalerweise wird diese mit dem ersten Start der Simulation gestartet, falls eine instal-

lierte Version vorliegt. Liegt diese nicht vor, ist es nicht möglich dem Lenkrad neue Parameter zu übergeben. Folgendes wird in der *SetPreferred*-Funktion der *LogiControllerProperties.cpp* in Zeilen 110 bis 118 abgefragt (Abbildung 25):

```
110     if (m_gamingSoftwareManager.IsEventMonitorRunning() && m_wingmanVersion.major == 5 && m_wingmanVersion.minor >= 3
111         || m_wingmanVersion.major > 5)
112     {
113         // version is good
114     }
115     else
116     {
117         return E_FAIL;
118     }
```

Abbildung 25: Event Monitor Abfrage in LogiControllerProperties.cpp

Der Check nach dem *Logitech Gaming Software Manager* wurde nachträglich hinzugefügt. Ist es nun bei dem ersten Start der Simulation nicht möglich die *Logitech Gaming Software* zu starten, wird die Simulation beendet und eine entsprechende Meldung erscheint. Ein erneuter Versuch die Simulation zu starten wird zwar glücken, jedoch wird das Lenkrad nur die Standardeinstellungen verwenden. Das Drücken des *Set Preferred*-Buttons in der Simulationsumgebung wird dann ebenfalls eine solche Fehlermeldung ausgeben.

4.4.4 Monitoring der Input-Ports

Das Monitoring der Input-Ports geschieht in der *mdlOutputs*-Methode innerhalb des *kUpdate switch cases*. Da *Simulink* Input-Signale nur numerische Werte sein können, muss hier keine Typenüberprüfung vorgenommen werden. Die Inputs, werden dabei lediglich auf die Einhaltung ihrer Grenzen überprüft. Alle Input-Ports des *g27Integration*-Blocks sind dabei optional. Um Speicherzugriffsverletzungen zu vermeiden wird deswegen zusätzlich jeder Input-Port auf eine bestehende Verbindung überprüft, bevor dieser ausgelesen wird. Die Einhaltung der Input-Port-Weiten wird von *Simulink* überwacht.

5 Simulationsumgebung

5.1 Simulationsumgebung Überblick

Mit der aufgebauten Simulationsumgebung (Abbildung 26) lassen sich alle zur Verfügung stehenden Funktionen des Lenkrads überprüfen. In erster Linie dient dieses Modell zu Testzwecken, ist aber auch in der Lage ohne Probleme mit anderen Programmen zu kommunizieren, welche mit *Simulink* kompatibel sind. Parameter und Variablen, die zusammengehören, wurden zur Übersichtlichkeit mithilfe der Area-Funktion von *Simulink* zusammengefasst. Die von *Simulink* verwendeten Standardblöcke beinhalten Dashboard- (z.B. Lampen und Geschwindigkeitsanzeigen), *Sink-*, *Constant-*, *Gain-* und *Signalroutingblöcke*. *Dashboard*-Blöcke wurden mit *MATLAB 2015* eingeführt, weswegen eine Verwendung dieses Modells mindestens diese Version voraussetzt. Die Simulationsumgebung wurde innerhalb *MATLAB 2016a* erstellt und für die 2015er Version exportiert und getestet. Die *MEX*-Funktionen wurden für 32bit und 64bit Systeme kompiliert und sind dem Ordner beigelegt. Alle für die *S-function* wichtigen Parameter werden im *OpenFcn*-Callback des Modells bereits erstellt und mit Werten versehen. Das Paket beinhaltet außerdem eine selbst angelegte Bibliothek, welche *MATLAB* als Pfad hinzugefügt werden kann. Dies wird zum Beispiel benötigt um im *Library Browser* von *Simulink* innerhalb anderer Modelle Zugriff auf die erstellte *S-function*, inklusive der darüber liegenden Maske und der benötigten Parameter, zu haben.

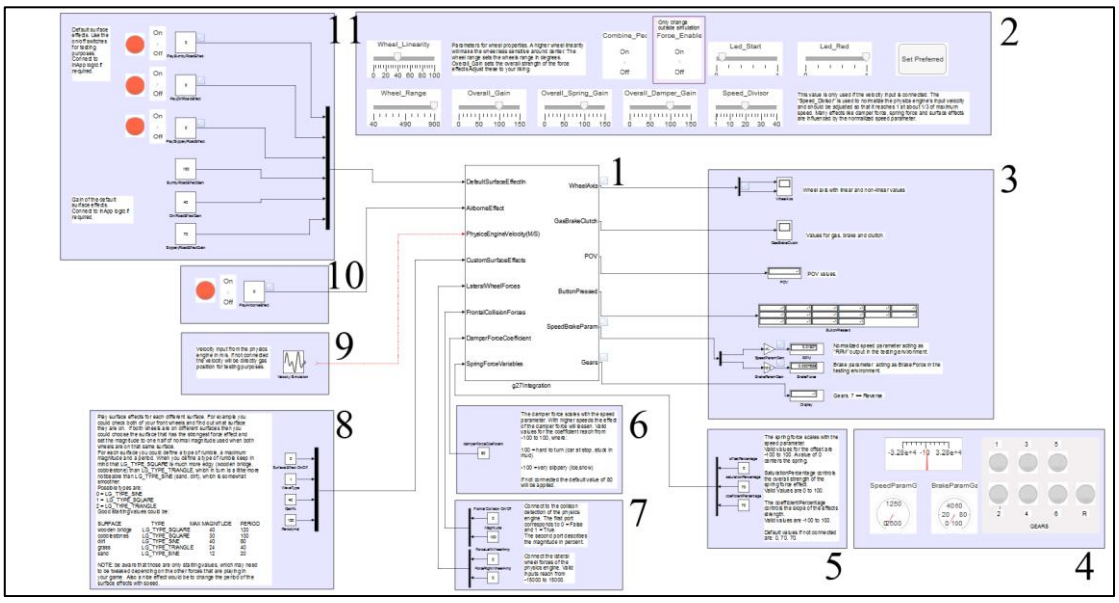

Abbildung 26: Simulationsumgebung G27 Tester

5.2 Bereiche und Funktionen

Gemäß der Nummerierung in Abbildung 26 sollen in diesem Unterkapitel die Funktionen, Voraussetzungen, sowie Inputs und Outputs der einzelnen Bereiche dargestellt und erläutert werden.

5.2.1 G27Integration S-function-Block

Der erstellte g27Integration-*Simulink*-Block besitzt acht Input- und sechs Output-Ports, sowie 11 Parameter. Die Weite der Ports kann Abbildung 27 entnommen werden. Das Anschließen der Ports ist optional und hängt von der intentionierten Verwendung ab. Voraussetzung ist die Installation der für das *G27* notwendigen Treiber. Möchte man Parameter an das Lenkrad übergeben können, muss die *Logitech Gaming Software* installiert sein. Der *S-function*-Block besitzt eine Maskierung mit bereits voreingestellten Parametern und einer dem *Help*-Button hinterlegten Dokumentation der Funktionalität. Sollte die Maskierung des Blockes nicht genutzt werden, haben die Parameter festgelegte Namen und müssen in der richtigen Reihenfolge eingetragen werden: *Wheel_Range, Force_Enable, Overall_Gain, Overall_Spring_Gain, Overall_Damper_Gain, Combine_Pedals, Wheel_Linearity, Speed_Divisor, Led_Start, Led_Red und Set_Preferred*. Die Funktionen der Eingänge und Parameter wird in den folgenden Abschnitten erläutert.

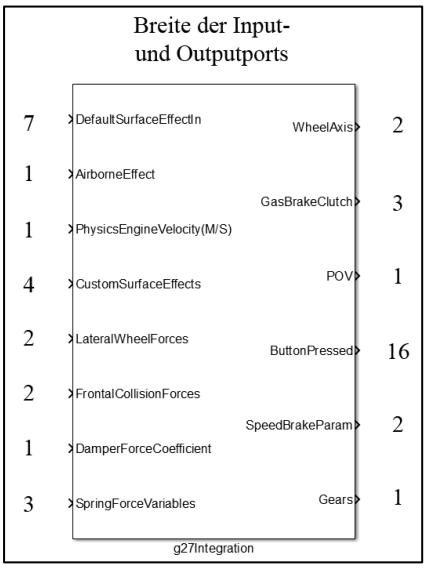

Abbildung 27: Breite der Input/Output Ports des G27-Blocks

5.2.2 Parameter

Der Parameterbereich (Abbildung 28) umfasst alle der *S-function* zu übergebenden Parameter. Für einen einfachen Zugriff wurden diese über *Dashboard*-Blöcke wie Schieberegler und Schalter zugänglich gemacht. Im Folgenden sollen die Parameter gelistet und erläutert werden.

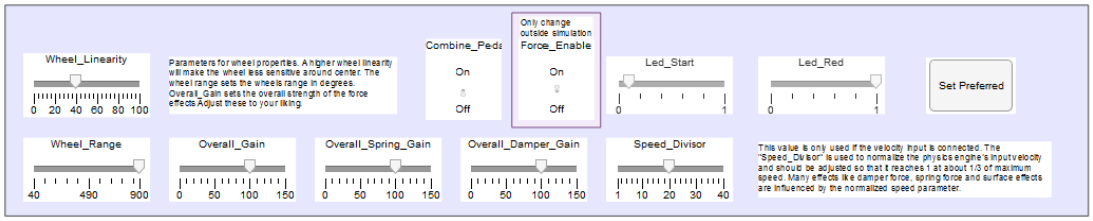

Abbildung 28: Parameter der G27 Simulationsumgebung

Wheel_Linearity

Der *Wheel_Linearity*-Parameter entspringt einer SDK-Funktion und wird dazu verwendet die Empfindlichkeit des Lenkrads um die Nullposition herum zu verringern. Eine hoher *Wheel_Linearity*-Wert hat somit die Folge, dass ein verhältnismäßig großer Lenkradausschlag um die Nullposition einen geringeren Versatz für die Räder bedeutet. Ein Wert von Null führt dabei zu einer 1 zu 1 Übersetzung. Durch einen höheren Wert lässt sich somit eine zu starke Response des Fahrzeugs gerade bei hohen Geschwindigkeiten tunen. Akzeptierte Werte reichen dabei von 0 bis 100 und sind als prozentuale Werte zu verstehen.

Wheel_Range

Der *Wheel_Range*-Parameter stellt den gewünschten Ausschlag des Lenkrads ein. Wird dieser überschritten, greift die Federung des Lenkrads ein und erzeugt ein Drehmoment, welches eine Zurückführung des Lenkrads in den tolerierten Ausschlagbereich zur Folge hat. Einzustellende Werte Reichen von 40 bis 900 und sind als Grad-Angaben zu verstehen.

Overall_Gain

Der *Overall_Gain*-Parameter dient dazu die Stärke der Effekte Federung (*Overall_Spring_Gain*) und Dämpfung (*Overall_Damper_Gain*) zu regeln. Einzustellende Werte reichen von 0 bis 150 und sind als prozentuale Angaben zu verstehen.

Overall_Spring_Gain

Der *Overall_Spring_Gain*-Parameter stellt die Stärke der Federung bei Überschreitung des Lenkradbereichs oder simulierten Bodeneffekten ein. Die Werte reichen dabei von 0 bis 150 und sind als prozentuale Angaben zu verstehen.

Overall_Damper_Gain

Der *Overall_Damper_Gain*-Parameter beeinflusst die Stärke der Dämpfung des Lenkrades. Diese nimmt mit zunehmender Geschwindigkeit ab und simuliert das Verhalten von stehenden bzw. in Bewegung befindlichen Rädern. Die Werte reichen dabei von 0 bis 150 und sind als prozentuale Angaben zu verstehen.

Speed_Divisor

Der *Speed_Divisor*-Parameter wird nur dann benutzt, wenn der *PhysicsEngineVelocity*-Input belegt ist. Dieser wird dazu genutzt die eingehenden Geschwindigkeitswerte in m/s zu normalisieren, sodass sie bei etwa einem Drittel der der Maximalgeschwindigkeit einen Wert von 1 betragen. Viele Effekte wie Stärke von Dämpfung, Federung oder die Intensität

von Bodeneffekten sind indirekt oder direkt von diesem Parameter abhängig. Hinsichtlich der gewünschten Simulationsumgebung lässt sich somit das Verhalten dieser feintunen. Akzeptierte Werte reichen von 0 bis 40 und sind als numerische Teiler zu verstehen.

Combine_Pedals

Der *Combine_Pedals*-Parameter wurde hinsichtlich von Programmen eingeführt, welche die Positionen von Gas- und Bremspedal aus einem Signal herauslesen. Ist dieser Wert auf null beziehungsweise Off geschaltet, wird über den *SpeedBrakeClutch*-Output ein Signal mit Werten für Positionen von [Gaspedal, Bremspedal, Kupplung] ausgegeben. Ist dieser Wert auf 1 bzw. On geschaltet, fallen Positionen von Gas- und Bremspedal auf einen Kanal. Das Signal würde als [Gaspedal/Bremspedal, NULL, Kupplung] ausgegeben werden. Akzeptierte Werte sind 0 und 1 und als logische Werte zu verstehen.

Force_Enable

Der *Force_Enable*-Parameter stellt ein ob *Forcefeedback* aktiviert ist oder nicht. Dabei kann dieser während der Simulation nicht geändert werden und führt bei solch einem Versuch zu einer Fehlermeldung und der Beendigung der Simulation. Ein andersfarbiger Hintergrund und der eingebaute Warnhinweis sollen dies verdeutlichen. Akzeptierte Werte sind 0 und 1 und sind als logische Operation zu verstehen.

Led_Start und Led_Red

Der Parameter *Led_Start* stellt ein, ab welchem normalisierten Geschwindigkeitswert die Leds anfangen zu leuchten. Mit dem *Led_Red*-Parameter lässt sich einstellen wann diese anfangen rot zu blinken. Akzeptierte Werte liegen zwischen 0 und 1.

Set_Preferred

Der *Set_Preferred*-Parameter gibt die eingestellten Parameter an die *Logitech* Software weiter. Je nach Geschwindigkeit der Simulation kann es sein, dass dieser für einen kurzen Augenblick gehalten werden muss, damit die Werte übernommen werden. Akzeptierte Werte für diesen Parameter sind 0 und 1 und als logische Operatoren zu verstehen.

Wird festgestellt, dass die Parameter trotz längeren Drückens von *Set_Preferred* nicht übernommen werden, lässt sich dies durch gleichzeitiges Drücken des rechten Wippschalters und mittleren roten Buttons am Lenkrad beheben. Beim Wechseln in ein anderes Fenster ist dieser Vorgang Voraussetzung.

5.2.3 Outputs

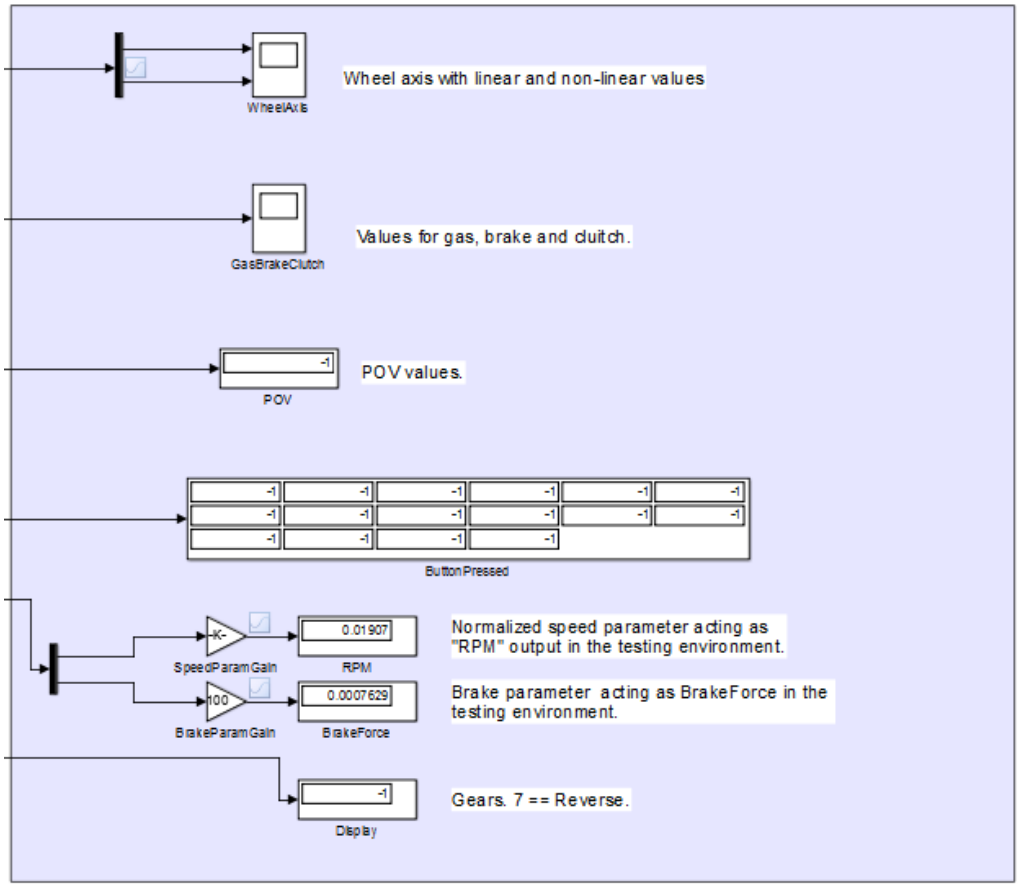

Abbildung 29: Outputs der G27 Simulationsumgebung

Der Output-Bereich (Abbildung 29) fängt alle Ausgangswerte der g27Integration *S-function* auf (für die Portweite der Signale siehe Abbildung 27). Der *Scope*-Block *WheelAxis* zeichnet die Positionen des Lenkrades auf. Sowohl die linearen als auch die nicht linearen Werte werden dabei aufgenommen. An ein angeschlossenes Programm würde der nicht lineare trimmbare Wert übergeben werden. Die Reichweite des Outputs reicht dabei von – 32768 bis 32767. Dies hat seinen Ursprung daher, dass durch einen 16bit Integer 2^16 oder 65,536 Werte ausgedrückt werden können.
Selbiges gilt für die Positionswerte von Gaspedal, Bremspedal und Kupplung, solange diese nicht im *Combined Pedal*-Modus ausgeführt werden. Ist dies der Fall, reicht zwar der Wertebereich der Kupplung weiterhin von -32768 bis 32767, das Gaspedal reicht jedoch von 0 bis -32768 und das Bremspedal von 0 bis 32767. Gas– und Bremspedal teilen sich dann die erste Position des *GasBrakeClutch* Arrays.
Die *POV*-Werte entspringen dem Steuerkreuz der Schalteinheit und werden innerhalb eines numerischen Displays ausgegeben. Ist das Steuerkreuz unbenutzt wird der Wert -1 ausgegeben. Die ausgegebenen Werte spiegeln Gradangaben wieder (mit der 0 für oben, 90 Grad für rechts etc.).
ButtonsPressed ist dabei eine Ansammlung nummerischer Displays. Jedes Display steht für einen verfügbaren Button. Solange diese nicht gedrückt werden, geben diese wie auch das

Steuerkreuz den Wert -1 aus. Wird ein Button gedrückt erscheint seine Nummer auf dem zugehörigen Display.
RPM und *BrakeForce* sind eigentlich die normalisierten *Speed*- und *Brakeparameter*, welche innerhalb der *S-function* genutzt werden um verschiedene Effektstärken zu bestimmen. Da ohne ein externes Programm keine RPM Werte oder Bremskräfte vorliegen, werden die *Speed*- und *Brakeparameter* hier mit jeweils einem *Gain*-Block multipliziert um diese zu „simulieren". Je nach Art der angeschlossenen Software, steht es einem frei die RPM Werte oder Gaspedalposition an das Programm zu übergeben.
Das *Gears*-Display ist dem Button-Display sehr ähnlich. Diese sind eigentlich nur Buttons, welche innerhalb der *S-function* korrigiert werden, um die richtige Gangzahl anzuzeigen. Ist kein Gang eingelegt, wird der Wert -1 ausgegeben. Danach steigen die Gänge von 1 bis 7, wobei der Gang 7 für den Rückwärtsgang steht.

5.2.4 Dashboard

Der Dashboard-Bereich gibt einen Überblick über die derzeitigen Parameter, welche während der Simulation für den Fahrer wichtig sind. Die Werte gehen dabei alle aus den Outputs in 5.2.3 hervor. Ist ein Gang eingelegt leuchtet die entsprechende Lampe grün auf.

5.2.5 Federkräfte

In dem Bereich der Federkraft (Abbildung 30) lassen sich drei Variablen einstellen. Anders als der *Spring_Force*-Parameter haben diese Variablen keinen Einfluss auf die Federkraft bei simulierten Bodeneffekten. Zum einen gibt es die Variable *offsetPercentage*. Mit dieser lässt sich die Nullposition des Lenkrades relativ zum eingestellten Lenkradausschlag verschieben. Dies kann benutzt werden um zum Beispiel Schäden zu simulieren. Akzeptierte Werte reichen von -100 bis 100 und sind als Prozentangaben zu verstehen. Die Variable *saturationPercentage* stellt die maximale Stärke dieses Effekts ein, wobei *coefficientPercentage* die Steigung der Stärke einstellt. Beide Variablen nehmen Werte von 0 bis 100 an und sind ebenfalls als Prozentangaben zu verstehen. Wie alle Inputs ist auch das Anschließen der Federkräfte optional. Werden diese nicht angeschlossen, werden die Werte 0, 70, 70 verwendet.

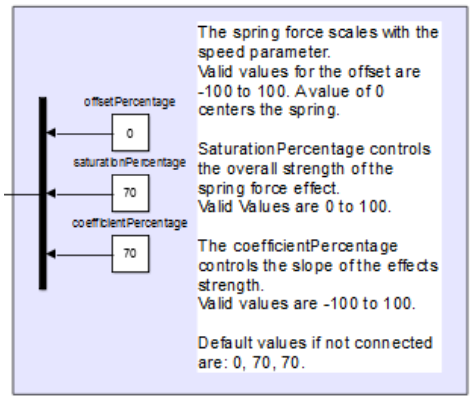

Abbildung 30: Federkraftbereich der G27 Simulationsumgebung

5.2.6 Dämpfungsbeiwert

In diesem Bereich befindet sich eine Variable namens *damperForceCoefficient*. Anders als der Parameter *Overall_Damper_Gain* stellt diese nicht die Stärke des Dämpfungseffekts ein. Der Wertebereich dieser Variablen reicht von -100 bis 100, wobei 100 sehr schwer zu lenken bedeutet. Dies wird verwendet um zum Beispiel das Lenkverhalten eines Autos im Matsch zu simulieren. -100 dagegen simuliert ein Lenkverhalten wie auf Eis, bei dem die Reifen schnell ausschlagen. Diese Variable geht quasi mit den Bodeneffekten Hand in Hand. Solange dieser Input nicht angeschlossen wird, wird ein Wert von 80 als Standard gesetzt.

5.2.7 Kollision und Reifenkräfte

In dem Kollisions- und Reifenkraft-Bereich (Abbildung 31) kommen zwei Funktionen zusammen. Zum einen findet man hier die Kollisionsdetektion, welche im Falle einer frontalen Kollision diese meldet. Der erste Wert des Arrays steht dabei für die logische Operation. 1 für eine Kollision und 0 andererseits. Der zweite Wert ist dafür da die Stärke dieses Effekts einzustellen und akzeptiert Werte von 0 bis 100. Die zweite Funktion, welche über die Eingangswerte *ForceLeftWheelAmp* und *ForceRightWheelAmp* angesteuert wird, ist dafür da, seitliche Reifenkräfte zu simulieren. Diese entstehen zum Beispiel, wenn man auf einer Strecke durch eine geneigte Kurve fährt. Akzeptierte Werte pro Vorderrad reichen von -15000 bis 15000. Dabei stehen die Minuswerte für Kräfte, welche die Reifen nach links auslenken würden. Die positiven Werte lenken die Reifen entsprechend nach rechts aus.

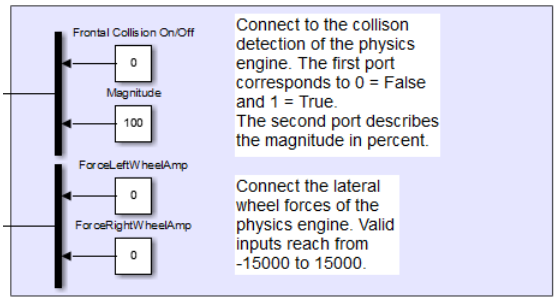

Abbildung 31: Kollision und Reifenkräfte der G27 Simulationsumgebung

5.2.8 Benutzerdefinierte Bodeneffekte

Über die benutzerdefinierten Bodeneffekte (Abbildung 32) lassen sich eine Vielzahl von Untergründen simulieren. Ob und inwieweit eine Überprüfung des Bodenkontakts der Reifen möglich ist, hängt von dem angeschlossenen Programm ab. Das Eingangssignal dieser Funktion besitzt eine Portweite von 4. Wie auch bei der Kollisionsdetektion dient der erste Wert dazu den Effekt jeweils an- und auszuschalten. Die zweite Variable *WaveType* bestimmt die Wellenform des erzeugten Effekts. Akzeptierte Werte reichen von 0 bis 2 und stehen für die folgenden Wellenformen:

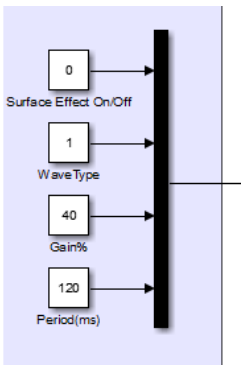

Abbildung 32: Benutzerdefinierte Bodeneffekte der G27 Simulationsumgebung

- 0 = Sinuswelle
- 1 = Rechteckwelle
- 2 = Dreieckwelle

Dabei ist die Sinuswelle die „weichste“ Welle und eignet sich für staubige oder sandige Untergründe. Die Rechteckwelle ist „kantiger“ und eignet sich für Untergründe wie eine hölzerne Brücke oder Pflastersteine. Die Dreieckwelle liegt dabei von der Art des Effektes zwischen diesen beiden. Die dritte Variable *Gain%* stellt die Stärke des Effektes ein und ist eine prozentuale Angabe mit Werten von 0 bis 100. Mit der vierten Variable *Period(ms)* lässt sich die Periode der Welle in Millisekunden einstellen. Akzeptierte Werte reichen dabei von 0 bis 150. Sofern die Überprüfung des Bodenkontakts der Reifen möglich ist, könnte man nun den Kontakt beider Vorderreifen checken, den Bodeneffekt mit dem stärkeren *Forcefeedback*-Effekt wählen und diesen dann mit der halben Stärke abspielen. Die Perioden dieser Effekte könnten auch mit der Geschwindigkeit skaliert werden. Das *Logitech* SDK liefert folgende Werte als Startwerte:

SURFACE	TYPE	MAX MAGNITUDE	PERIOD
Raues Holz	Rechteckwelle	40	120
Pflastersteine	Rechteckwelle	30	100
Dreck	Sinuswelle	40	80
Gras	Dreieckwelle	24	40
Sand	Sinuswelle	12	20

5.2.9 Geschwindigkeit

Die Geschwindigkeit welche dem angeschlossenen Programm entspringt, muss in Meter/Sekunde übergeben werden. Solange keine externe Geschwindigkeit geliefert wird, wird diese zu Testzwecken direkt an die Position des Gaspedals gebunden. Viele Effektstärken werden intern mit der Geschwindigkeit skaliert. An die Um diese irgendwie zu begrenzen liegen die akzeptierten Werte für die Geschwindigkeit zwischen -120 und 120.

5.2.10 Airborne-Effekt

Der *Airborne*-Effekt wird nur über 1 und 0 an- und ausgeschaltet. Ist der *Airborne*-Effekt an, überlagert dieser alle *Forcefeedback*-Befehle. Feder- und Dämpfungskräfte sowie Bodeneffekte werden ausgeschaltet solange der *Airborne*-Effekt läuft.

5.2.11 Standardbodeneffekte

Die Standardbodeneffekte sind im Grunde voreingestellte Bodeneffekte, welche bereits mit der Geschwindigkeit skalieren. Die ersten 3 Plätze dieses Signals werden dafür genutzt diese Effekte über 1 und 0 an- und auszuschalten. Dabei belegen die Effekte diese Plätze in folgender Reihenfolge: *Bumpy road, dirt road* und *slippery road*. Die darauffolgenden Variablen stellen die Stärke der ersten drei Effekte in selbiger Reihenfolge ein und akzeptieren Werte von 0 bis 100. Zweckmäßig sind neben den Eingangswerten zum An- und Ausschalten Schalter angebracht, um diese Effekte schnell testen zu können.

6 Schlussbetrachtung

Im Rahmen dieser Arbeit wurde die Integration des *Logitech* SDK*s* in die *MATLAB/Simulink* Umgebung durchgeführt. Um sich mit der Materie befassen zu können, musste sich in einem ersten Schritt mit *S-functions* und dem Aufbau des *Logitech* SDK*s* auseinandergesetzt werden. Trotz der eigenen Architektur des SDK*s* und der damit verbundenen Kommunikation mit der *Logitech Gaming Software* ist es über mehrere Iterationsschritte geglückt die volle Funktionalität des *G27* Lenkrades unter *MATLAB/Simulink* zur Verfügung zu stellen. Die Funktionen des *G27* wurden dabei nach und nach eingebaut und getestet um einen Überblick über die Fehlerquellen zu gewährleisten. Änderungen im Source-Code des SDK*s* wurden an gegebener Stelle erläutert, innerhalb der Source-Dateien namentlich gekennzeichnet und veränderte Dateien im Kapitel „Veränderte Source-Dateien des SDKs“ genannt.

Während dieser Iteration fiel auf, dass zusätzliches Wissen zu *Windows*-Fensterprozessen, sowie den *DirectInput*-Methoden, benötigt wird. Entsprechende Dokumentationen wurden daraufhin durchgearbeitet und problemspezifische Aspekte genannter APIs erläutert.

Dabei wurden mögliche Anwendungsfehler analysiert, mit entsprechenden Mitteln eliminiert und mit Fehlermeldungen versehen. Das Ziel den Anwender über etwaige Fehler und entsprechende Lösungsansätze zu informieren wurde so vorerst erreicht. Da unterschiedliche Computerkonfigurationen jedoch möglicherweise andere Fehler zum Vorschein bringen, wäre es wünschenswert weitere Testpersonen heranzuziehen.

Die Simulationsumgebung wurde über mehrere Iterationsschritte aufgebaut und soll dem Benutzer einen Überblick über die Möglichkeiten des *G27* Lenkrads bieten. Entsprechende Bereiche und Funktionen wurden zusätzlich erklärt.

Die *S-function* wurde zusätzlich als Block in einer benutzerdefinierten Bibliothek gespeichert und hinterlegt, sodass die Einbindung in *Simulink* schnell und einfach vollzogen werden kann. Für den Block wurde dabei eine Maskierung erzeugt, sodass die Parameter nicht erst eigenständig erstellt werden müssen. Die Dokumentation der entsprechenden Funktionen des Blocks wurde dabei dem *Help* Button der *S-function* hinterlegt und steht so jederzeit zur Verfügung.

Die in dieser Arbeit erzeugten *Simulink*-Modelle und *MEX*-Dateien sowie die verwendeten und bearbeiteten Source-Dateien des *Logitech* SDKs, stehen auf der beigelegten CD zur Verfügung.

Veränderte Source-Dateien des SDKs

LogiControllerInput.cpp

LogiControllerInput.h

LogiControllerProperties.cpp

LogiControllerProperties.h

LogiWheel.cpp

LogiWheelGlobals.h

Literaturverzeichnis

(*MATLAB* Dokumentation)	https://de.mathworks.com/help/*MATLAB*/
(*Simulink* Dokumentation)	https://de.mathworks.com/help/Simulink/
(*SteeringWheel* API Dokumentation)	Logitech_SDK_For_PC_1.00.002/ SteeringWheel/Doc/api.html#robo13
(*ControllerInput* API Dokumentation)	Logitech_SDK_For_PC_1.00.002/ ControllerInput/Doc/api.html
(*DirectInput* Dokumentation)	https://msdn.microsoft.com/en-us/library/windows/desktop/ee416842(v=vs.85).aspx
(*Window Procedures* Dokumentation)	https://msdn.microsoft.com/en-us/library/windows/desktop/ms632593(v=vs.85).aspx
(*PDB*-Dateien *Microsoft* Dokumentation)	https://msdn.microsoft.com/de-de/library/aa292304(v=vs.71).aspx

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Hamburg, den _____ _____