

Validierung der Basis-Konfiguration des Audi- Autonomous-Driving-Cup-Fahrzeuges

STUDIENARBEIT

im Studiengang „Fahrzeugelektronik und mechatronische Systeme“

an der DHBW Ravensburg
Campus Friedrichshafen

von

Jan Mittelstädt, Patrick Lang, Laurent Eherler

17.12.2014

Bearbeitungszeitraum:

Oktober bis Dezember 2014

Matrikelnummern:

3215315, 9308051, 2332035

Betreuer:

Prof. Dr. Konrad Reif

Eidesstattliche Erklärung

gemäß § 5 (3) der „Studien- und Prüfungsordnung DHBW Technik“ vom
22. September 2011.

Hiermit erklären wir, dass wir die vorliegende Arbeit mit dem Titel

„Validierung der Basis-Konfiguration des Audi-Autonomous-Driving-Cup- Fahrzeuges“

selbständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt, keine
anderen als die angegebenen Hilfsmittel benutzt und wörtliche sowie sinngemäße
Zitate als solche gekennzeichnet haben.

Friedrichshafen, den 16. Dezember 2014

Jan Mittelstädt

Patrick Lang

Laurent Eherler

Inhaltsverzeichnis

Inhaltsverzeichnis	III
Abkürzungsverzeichnis.....	V
Abbildungsverzeichnis	VI
Formelverzeichnis	VIII
1 Einleitung	1
1.1 Motivation	1
1.2 Gegenstand und Zielsetzung.....	2
2 Theoretische Grundlagen.....	5
2.1 Fahrzeug	5
2.1.1 Energieversorgung.....	5
2.1.2 Steuerrechner & Messrechner	5
2.1.3 Sensoren	6
2.1.4 Aktuatoren	9
2.2 Automotive Data and Time-Triggered Framework (ADTF)	11
2.2.1 Begriffsbestimmung ADTF	11
2.2.2 Funktionsweise ADTF	12
2.3 Filterbeschreibung	13
2.3.1 Arduino Communication.....	14
2.3.2 Arduino Sensors	15
2.3.3 Arduino Actuator	18
2.3.4 RPM Calculation	18
2.3.5 Attitude Converter	19
2.3.6 Calibration Scaling	19
2.3.7 Xtion Camera	20
2.3.8 Watchdog Trigger	21
2.3.9 Jury Module	21
3 Definition von Testfällen.....	22
3.1 Das V-Modell.....	22
3.1.1 Komponententest.....	23
3.1.2 Integrationstest	24
3.2 Komponententest der ADTF Filter	25
3.2.1 Arduino Communication.....	26

3.2.2	Arduino Actuator	27
3.2.3	Arduino Sensors	28
3.3	Integrationstest der ADTF Filter	29
4	Filter-Tests	31
4.1	Umsetzung der Komponententests	31
4.1.1	Arduino Communication.....	31
4.1.2	Arduino Actuator	53
4.1.3	Arduino Sensors	58
4.2	Umsetzung des Integrationstests	62
5	Zusammenfassung	65
5.1	Ergebnis und Bewertung	65
5.2	Ausblick	66
	Glossar	IX
	Literaturverzeichnis	XI
	Anhang	XII

Abkürzungsverzeichnis

AADC	Audi Autonomous Driving Cup
ADTF	Automotive Data and Time-Triggered Framework
AG	Aktiengesellschaft
DTM	Deutsche Tourenwagen Masters
HDMI	High Definition Multimedia Interface
ID	Identifikation
LED	Licht-emittierende Diode
PWM	Pulsweitenmodulation
RGB	Rot-Gelb-Grün-Farbraum
SDK	Software Development Kit
SD-Karte	„Secure Digital Memory“-Karte
USB	Universal Serial Bus
VW	Volkswagen
WLAN	Wireless Local Area Network

Abbildungsverzeichnis

Abbildung 1 Audi RS7 piloted driving concept	2
Abbildung 2 Basiskonfiguration Software	3
Abbildung 3 Übersicht Sensoren	6
Abbildung 4 Ultraschallsensor MB1242	6
Abbildung 5 Infrarotsensor GP2Y0A21YK0F	7
Abbildung 6 Raddrehzahlsensor ROB – 09454	7
Abbildung 7 Lagesensor SEN - 11028	8
Abbildung 8 Asus Xtion	9
Abbildung 9 Servomotor-Regelkreis	10
Abbildung 10 Fahrtregler	10
Abbildung 11 Funktionsprinzip bürstenloser Motor	11
Abbildung 12 Funktionsweise ADTF – Übersicht.....	13
Abbildung 13 Arduino Protokoll	15
Abbildung 14 Sensor Datenpakete	15
Abbildung 15 Aktuator Datenpakete	18
Abbildung 16 Spline mit acht Messdaten.....	20
Abbildung 17 Das V-Modell	22
Abbildung 18 Schematische Darstellung des Sensor-Filters	29
Abbildung 19 Testdefinition SteeringAngle	32
Abbildung 20 Erstellen des Output-Pins	33
Abbildung 21 Erstellen eines MediaSample	34
Abbildung 22 Werte in MediaSample festlegen	35
Abbildung 23 TransmitDriveCommand	35
Abbildung 24 TransmitLightSignal	36
Abbildung 25 TransmitWatchdog.....	36
Abbildung 26 Testing – Initialisierung	37
Abbildung 27 Testing – Rampendefinition	38
Abbildung 28 Testing – Watchdog.....	38
Abbildung 29 Testing - Acceleration and SteerAngle	39
Abbildung 30 Testing - LightTest	40
Abbildung 31 <i>SampleSink</i>	41

Abbildung 32 Verknüpfung SampleSink	41
Abbildung 33 Map zur Sortierung und Abspeicherung der <i>MediaSamples</i>	42
Abbildung 34 Auslesen der empfangenen <i>MediaSamples</i>	43
Abbildung 35 Abspeichern der <i>MediaSamples</i>	44
Abbildung 36 Speicherbereinigung Arduino Communication Filter	44
Abbildung 37 Anzahlprüfung <i>MediaSamples</i>	45
Abbildung 38 Raddrehzahlsensor - Festlegung wichtiger Grenzen und Werte	46
Abbildung 39 Auswertung Header	46
Abbildung 40 Auslesen und Gewichten der Daten	47
Abbildung 41 Rampenerkennung Raddrehzahlsensor	48
Abbildung 42 Auswertung der Rampenerkennung	48
Abbildung 43 Umspeichern der Werte und Auswertung Testdurchlauf	49
Abbildung 44 Lenkwinkelsensor - Festlegung wichtiger Grenzen und Werte	50
Abbildung 45 Failed Test.....	51
Abbildung 46 Test Result	52
Abbildung 47 Passed Test.....	52
Abbildung 48 Konfiguration des Schleifendurchlaufs.....	54
Abbildung 49 Überprüfen der Nutzdaten – Steering Angle	54
Abbildung 50 Test-Samples werden generiert.....	56
Abbildung 51 Anzahl der Samples überprüfen	57
Abbildung 52 Differenzierung der Datenlängen des Watchdogs und der restlichen Pins	58
Abbildung 53 Deklaration der Soll-Datenlängen	59
Abbildung 54 Generierung der Test-Daten.....	60
Abbildung 55 Übersicht der Senken-Funktionen	61
Abbildung 56 Ausschnitt der Senke für die Ultraschall-Sensorik	61
Abbildung 57 Verbinden der zu testenden Filter.....	62
Abbildung 58 : Erstellen der Quellen für den Aktuator-Filter.....	63
Abbildung 59 Erstellung der Senken für den Sensor-Filter	63
Abbildung 60 Maximaler Schleifenzähler und Vorlaufzeit deklarieren	64
Abbildung 61 Aufgreifen des Watchdog-Output-Pins	64

Formelverzeichnis

Formel 1 Abrollumfang	8
Formel 2 Kalibration Accelerometer Signals.....	16
Formel 3 Spannungsteiler Systemspannung	17
Formel 4 Kalibration Systemspannung	17
Formel 5 Umrechnung Bogenmaß in Grad.....	19

1 Einleitung

1.1 Motivation

„Überholt die Maschine den Menschen?“¹

Dieser Fragestellung begegnen nicht nur Wissenschaftler des Themengebietes der Robotik, sondern immer intensiver große namhafte Automobilhersteller in Deutschland. Wo moderne Fahrerassistenzsysteme fast veraltet wirken, setzen diese intelligenten Recheneinheiten, gepaart mit einer Unzahl an verschiedenen Sensoren und weiteren elektrischen Feinessen erst an.

Einsatz soll diese Technik immer dann finden, wenn die Fahrzeugkontrolle des eigentlichen Fahrers kritisch ist. Dieses Szenario spiegelt sich vor allem in zwei konträren Situationen wieder. Wird es dem Fahrer zu hektisch, so werden meist falsche oder gar keine Entscheidungen getroffen, die das Fahrzeug stabilisieren würden. Tritt hingegen Langeweile auf, so verfällt der Fahrer oft der eigenen Konzentrationsschwäche indem er beispielsweise Ablenkung im Internet findet.

Das Unternehmen AUDI AG stellte dazu am 19. Oktober 2014 zum DTM-Finale in Hockenheim das Audi RS7 piloted driving concept vor, welches „auf dem Level eines Rennfahrers, und das dauerhaft und ohne zu ermüden“² die Strecke abfuhr. Hierbei soll allerdings nicht der Mensch durch die Maschine abgelöst werden, vielmehr soll demonstriert werden, auf welchem Stand der Technik sich das Unternehmen heute schon befindet. Abbildung 1 zeigt die dazugehörige Sensoren-Vielfalt, mit der es bald möglich sein soll, dass sich das Fahrzeug im Stau selbstständig steuert oder eigenständig auf Parkplatzsuche geht, wenn der eigentliche Fahrer schon die Einkäufe in der Innenstadt erledigt.

¹ (Decker, 2012)

² (Hackenberg, 2014)

Audi RS 7 piloted driving concept

Fahrerassistenzsysteme
 10/14

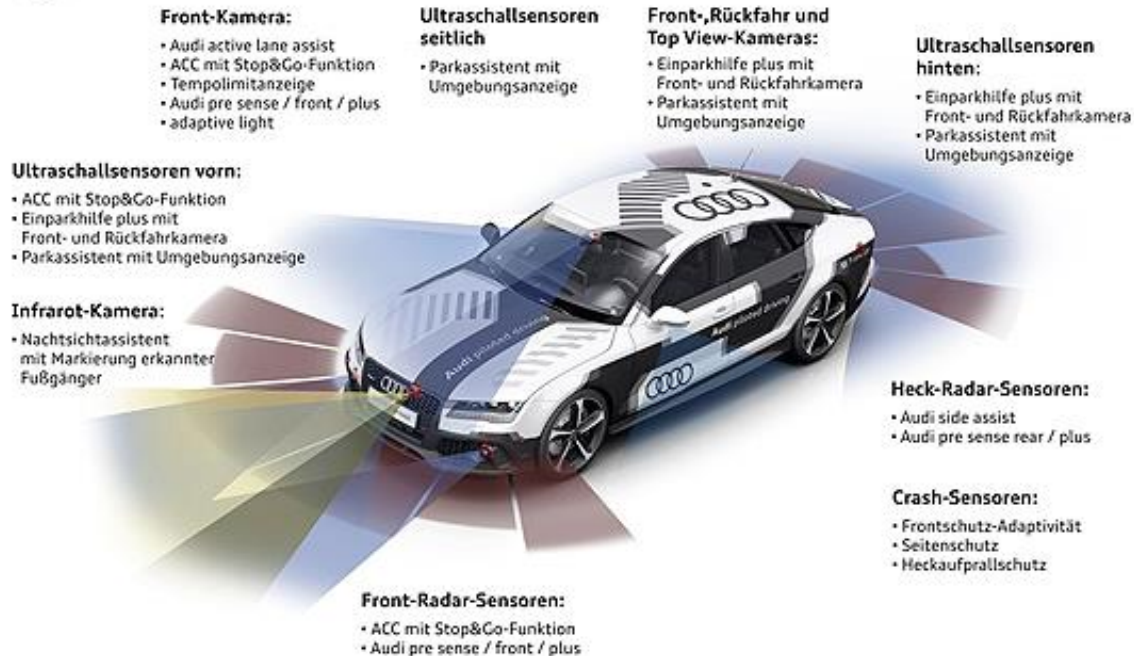


Abbildung 1 Audi RS7 piloted driving concept

1.2 Gegenstand und Zielsetzung

Audi organisiert speziell zu dem Themengebiet des „autonomen Fahrens“ einen eigenen Wettbewerb mit dem Audi Autonomous Driving Cup. Ziel dieses Wettbewerbs ist es, vollautomatische Fahrfunktionen sowie zugehörige Softwarearchitekturen zu entwickeln. Speziell für den Wettbewerb entwickelte Modellfahrzeuge werden für den Wettbewerb bereitgestellt. Die Modellfahrzeuge im Maßstab 1:8 sind mit reichlich Rechenleistung ausgestattet und haben eine ähnlich hohe Anzahl an Sensoren, wie der in Abbildung 1 gezeigte RS7. Auch eine Basis-Software wird bereitgestellt und vorgegeben, um Sensoren sowie Aktuatoren anzusteuern. Eine Optimierung der Ausführungsgeschwindigkeit oder Genauigkeit wurde nicht vorgenommen, um auch eine entsprechende Fahr-Performance zu bewerten.

Die Basissoftware besteht aus verschiedenen Filtern, die in der Software-Umgebung ADTF erstellt wurden. Dabei werden Sensoren und Aktoren als Regelschleife verknüpft, wobei ein Arduino Mikrocontroller die Verbindung zur Hardware darstellt.

Im Rahmen der vorliegenden Studienarbeit werden hierzu die drei Haupt-Filter der bestehenden Basissoftware getestet, die sich als Arduino Communication, Arduino Actuator und Arduino Sensors Filter darstellen. Hierzu werden im zweiten Kapitel die theoretischen Grundlagen dieser, sowie der übrigen Filter der Basissoftware erarbeitet. Auch die Hardware-Komponenten des zur Verfügung stehenden Fahrzeugmodells werden kurz dargestellt und erläutert. Anschließend werden im

dritten Kapitel die Testfälle definiert, bei denen die Filter sowohl als einzelne Unit, als auch zusammenhängend geprüft werden sollen. Auf diesen Definitionen aufbauend, werden die Filter-Tests anhand von Software erstellt, auf dem Fahrzeug durchgeführt und abschließend in Kapitel vier in Aufbau und Vorgehen dokumentiert. Zuletzt folgt im fünften Kapitel die Zusammenfassung, in der noch einmal die gewonnen Erkenntnisse schlussgefolgert werden und ein Ausblick dargelegt wird.

Ziel dieser Arbeit und dieses Vorgehens ist es, eine voll funktionsfähige sowie robuste Software zu gewährleisten, auf deren Grundlage weitere Filter hinzugefügt werden können.

2 Theoretische Grundlagen

Dieses Kapitel soll dem Leser sowohl wichtige Grundlagen in Punkto Fahrzeug und Software vermitteln, als auch einen theoretischen Überblick verschaffen, in welche Theorieaspekte sich die vorliegende Arbeit eingliedern lässt. Die Gewichtung dieser Grundlagen liegt dabei vor allem auf der Software, da diese das Kernelement der vorliegenden Studienarbeit bildet.

2.1 Fahrzeug

Das Fahrzeug wurde speziell für den Wettbewerb des AADC entwickelt und bildet ein Audi-Modell im Chassis-Maßstab 1:8 ab, um die benötigte Bordelektronik zu verbauen. Die integrierten Hauptkomponenten werden zunächst in den folgenden Unterkapiteln näher erläutert.

2.1.1 Energieversorgung

Die Energieversorgung des Modellautos wird durch zwei separierte Energiekreise gewährleistet, womit sowohl die Versorgung der Messtechnik als auch die des Antriebes abgedeckt wird. Beide Akkumulatoren sind Lithium-Polymer-Akkus. Dieser Akku-Typ ist v.a. im Modellbau sehr beliebt, da dieser sehr häufig erneut aufgeladen werden kann, ohne dass Leistungsverluste spürbar sind.

Speziell bei dem AADC-Modellfahrzeug werden zusätzlich kleine Messgeräte verwendet, die mittels Balancer-Anschluss verbunden sind. Mittels dieses Anschlusses wird eine Überwachung der einzelnen Zellen sowie der Gesamtspannung vorgenommen. Fällt eine Zelle unterhalb von 3,3 Volt, so ertönt ein akustisches Warnsignal, welches den entsprechenden Akkumulator vor einer Tiefenentladung bzw. irreversiblen Schädigung der Zellen schützen soll.

2.1.2 Steuerrechner & Messrechner

Den Messrechner bildet ein X2 Odroid Board mit 1,7 GHz Quad Core ARM Prozessor und 2GB RAM. Mittels dieses Rechners werden Mess- und Videodaten verarbeitet. Als Schnittstellen werden dabei Ethernet, WLAN, USB und HDMI bereitgestellt, die die Verbindung vom oder zum Fahrzeug ermöglichen. Auch ein SD-Karten-Einschub ist zur Datenablage verbaut.

Als Steuerrechner kommt ein Arduino Due zum Einsatz, welcher zum einen Schnittstellen zu Sensoren und Aktoren bereitstellt und zum anderen mittels USB mit dem Messrechner kommuniziert.

2.1.3 Sensoren

Das AADC-Modellfahrzeug ist mit einer großen Anzahl von Sensoren ausgestattet, welche in Abbildung 3 ersichtlich sind. Die Sensoren liefern permanent Daten, welche als sogenannte Rohwerte noch kalibriert werden müssen. Diese Kalibrierung der Rohwerte, in Form von analogen Spannungen oder Registerwerten findet in entsprechenden ADTF-Filtern statt.

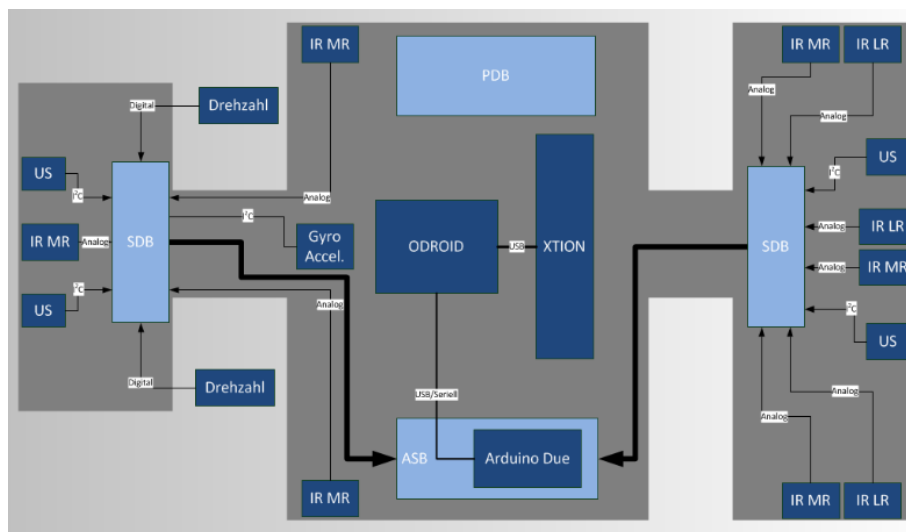


Abbildung 3 Übersicht Sensoren

2.1.3.1 Ultraschallsensoren

Das AADC-Fahrzeug besitzt jeweils im Front- und Heckbereich des Fahrzeuges zwei Ultraschallsensoren der Firma MaxBotix (MB1242I 2CXL - MaxSonar® - EZ4™). Ultraschallsensoren senden kurze Ultraschallimpulse aus, wobei die Schallwellen durch verschiedene Objekte reflektiert werden.³ Die empfangenen Signale werden registriert, wobei der Minimalabstand dieses Ultraschallsensors bei 20 cm liegt.



Abbildung 4 Ultraschallsensor MB1242

³ Vgl. (Bosch Mobility Solutions, 2014)

2.1.3.2 Infrarotsensoren

Infrarotsensoren sind im Modell neun Stück integriert, wobei diese nochmals unterschieden werden müssen. Die Sensoren im mittleren Infrarotbereich (Sharp GP2Y0A02YK0F Mid - Range) sind denen im niederen Infrarotbereich (Sharp GP2Y0A21YK0F Short - Range) im Verhältnis 2:1 stückmäßig überlegen. Diese Differenzierung ist allerdings nur für den Entfernungsmessbereich entscheidend, wobei die prinzipielle Funktionsweise identisch ist. Im Aufbau besteht der Infrarotsensor der Firma Sharp aus einem Sender und einem Empfänger. Der Sensorsender sendet einen Infrarot-Strahl aus, welcher wiederum von einem Objekt reflektiert wird. Abhängig von der Entfernung des Objektes trifft der reflektierte Strahl an einer divergierenden Stelle des Sensorempfängers auf. Diese Abweichung des Auftreffpunktes am Empfänger wird in einen analogen Spannungswert umgewandelt.⁴



Abbildung 5 Infrarotsensor GP2Y0A21YK0F

2.1.3.3 Raddrehzahlsensoren

Die Raddrehzahlsensoren (ROB – 09454 mit Fairchild QRE1113 Infrarot) der Marke Sparkfun befinden sich jeweils an den beiden Hinterrädern des Fahrzeuges, wodurch neben der Drehrichtung auch die zurückgelegte Strecke ermittelt werden kann.



Abbildung 6 Raddrehzahlsensor ROB – 09454

Auf der Innenseite des Rades befinden sich Streifen, durch die der Sensor pro Umdrehung genau acht Ticks auslöst. Der Abrollumfang lässt sich durch das

⁴ Vgl. (Sharp, 2006)

Bekanntsein des Reifendurchmessers von 10 cm und mittels der Kreisumfang-Formel ermitteln.

Darüber hinaus ergibt sich pro Tick eine Strecke von 0,04m, was anhand der Rechnung in Formel 1 nachziehbar ist.⁵

Formel 1 Abrollumfang

$$\frac{0,1m \times \pi}{8} \approx 0,04m$$

2.1.3.4 Lagesensor

Ein weiterer Sensor stellt der Sechs-Achsen-Lagesensor (SEN – 11028 mit MPU6050) der Marke Sparkfun dar. Dieser Sensor vereint sowohl einen Beschleunigungs- sowie Drehratensensor, womit die Beschleunigung und Drehrate in der Längs-, Quer- sowie Hochachse gemessen wird.⁶



Abbildung 7 Lagesensor SEN - 11028

2.1.3.5 Kamera

Die 3D Kamera der Marke Asus, ist eine weitere Sensorart, die sowohl Entfernung- als auch Farbbilder liefert.

Zur Gewinnung von Tiefenbildern und Informationen werden zwei der drei Linsen benötigt. Bestehend aus Sender und Empfänger, sendet die Kamera einen großflächigen Infrarotstrahl aus. Der Tiefensensor empfängt die reflektierten Infrarotstrahlen und wertet diese Information anschließend aus.

Das farbliche Fotografieren wird mit der RGB-Kameralinse realisiert.

⁵ Vgl. (BFFT Gesellschaft für Fahrzeugtechnik GmbH, 2014)

⁶ Vgl. (Grewal, Weill, & Andrews, 2007, S. 61 f.)



Abbildung 8 Asus Xtion

2.1.4 Aktuatoren

Das AADC-Modellfahrzeug besitzt drei verschiedene Aktoren, die jeweils durch PWM-Signale angesteuert werden. Auch eine Kalibrierung der entsprechenden Komponenten ist notwendig, welche im ADTF-Calibration Filter stattfindet.

2.1.4.1 Lenkwinkelservomotor

Der Lenkwinkelservomotor beschreibt einen Motorentyp, welcher zur Ansteuerung in einem geschlossenen Regelkreis betrieben wird. Dieser Motor ermöglicht die Kontrolle der Winkelposition, sowie Drehgeschwindigkeit und Beschleunigung. Neben dem eigentlichen elektrischen Motor ist zur Realisierung der Funktion eine weitere Leitung angebracht, damit der Lenkwinkel zurückgelesen werden kann.

Der grundsätzliche Vorgang einer Regelkette des Servomotors ist im Aufbau gleich. Zunächst wird das PWM-Eingangssignal in eine Gleichspannung gewandelt und mit dem Spannungswert des Endpotentiometers verglichen. Stimmen diese Werte überein, so tritt keine Veränderung des Winkels auf und der Servomotor hält die Position. Treten bei den Werten allerdings Differenzen auf, so wird der Motor aktiv und verändert seine Position. Diese Positionsänderung ist in Abbildung 9 nachvollziehbar wobei der Motor durch einen Verstärker angesteuert wird, der das Signal des Vergleichers bekommt. Der Motor treibt das Getriebe an, welches wiederum mit dem Potentiometer verbunden ist. Stimmt die zurückgeführte Potentiometer-Spannung mit der des PWM-Signals wieder überein, so ist der korrekte Winkel erreicht und der Servomotor hält diese Position erneut.⁷

⁷ Vgl. (Schenke, 2013)

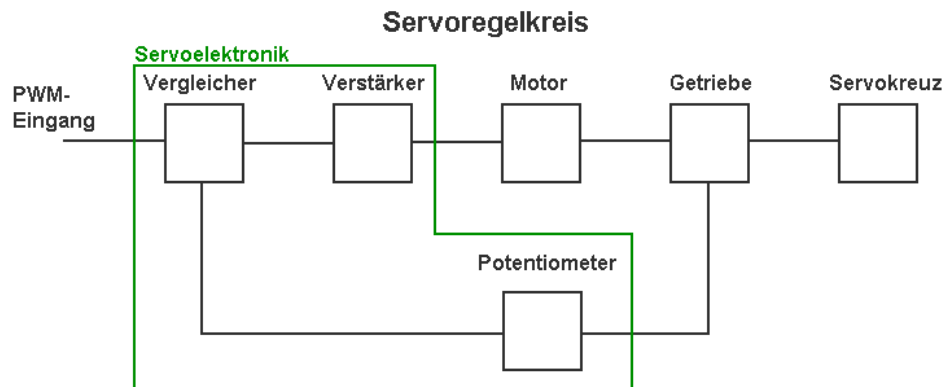


Abbildung 9 Servomotor-Regelkreis

2.1.4.2 Fahrtregler

Das AADC-Fahrzeug besitzt einen Fahrtregler der Marke Robitronic, welcher zur Drehzahlregelung des Motors verwendet wird und somit eine Entkoppelung von Motor und Mikrokontroller bewirkt. Der Fahrtregler erhält Impulse als Eingangssignal des Mikrokontrollers und setzt diese zu einem PWM-Signal für den Motor um. Das Bremsen sowie Vorwärts- und Rückwärtsfahren wird mittels dieser Technik realisiert.



Abbildung 10 Fahrtregler

2.1.4.3 Motor

Der bürstenlose Elektromotor der Marke Robitronic ist ein leichter, drehmomentstarker Motor, welcher benötigt wird, um das hohe Gewicht des Modellfahrzeugs auch aus dem Stand oder in niedrigen Geschwindigkeitsbereichen optimal zu steuern.

Die Funktionsweise bei einem bürstenlosen Gleichstrommotor unterscheidet sich zu herkömmlichen Bürstenmotoren in einem umgekehrten Funktionsprinzip. Der Rotor eines bürstenlosen Motors besteht dabei aus einem starken Permanentmagneten, welcher über Kraftschluss mit der Motorwelle verbunden ist. Zur Drehung des Motors ist hierzu noch ein Stator nötig. Dieser besteht aus mehreren Elektromagneten, die durch unterschiedliche Ansteuerung ein Wechselfeld bilden, dem der Rotor folgt.

Dazu wird des Weiteren eine Steuerelektronik benötigt, die zur Ansteuerung mit drei Kabeln über den Fahrtregler verbunden ist. Abbildung 11 zeigt hierzu den schematischen Aufbau, wobei zusätzlich in dieser Abbildung die Motorsteuerung mittels Sensoren realisiert wird. Dabei ermitteln kleine Sensoren im Innenraum des Motors alle Daten, um Position Drehzahl und Drehrichtung des Rotors exakt für den Regler zu bestimmen. Es kommen hierbei fünf Sensoren zum Einsatz, sodass der Regler mehr Informationen über den Rotor erhält, als über den Induktionsstrom der jeweiligen Phasen. Der Induktionsstrom ist ein Nebeneffekt der Drehbewegung des Motors, wobei auch mit diesem Strom, in Kombination mit einem komplexen Programm, der Regler mit den notwendigen Daten versorgt werden kann. Die Ermittlung mittels Sensordaten ist somit zugleich exakter in den Ergebnissen, als auch in der Anforderung geringer, um die passende Frequenz für den Drehstrom zu erzeugen.

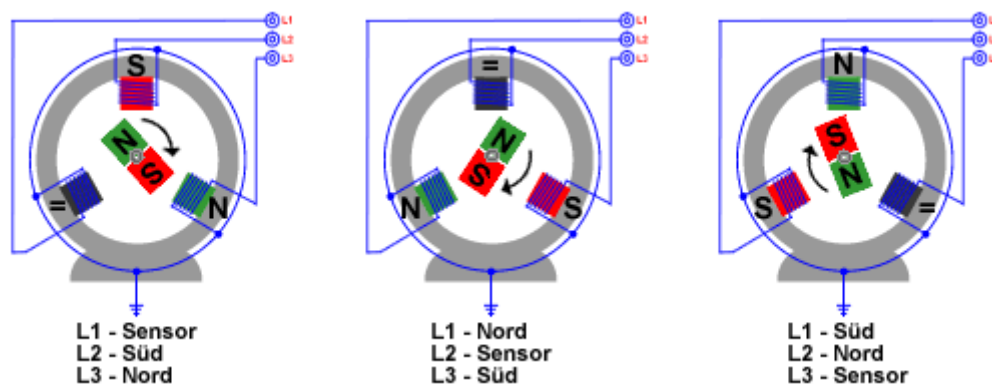


Abbildung 11 Funktionsprinzip bürstenloser Motor

2.2 Automotive Data and Time-Triggered Framework (ADTF)

2.2.1 Begriffsbestimmung ADTF

ADTF ist eine Entwicklungsumgebung, die speziell für die Software-Entwicklung von Fahrerassistenzsystemen geschaffen wurde. Seit dem Jahre 2001 wird ADTF in Zusammenarbeit mit der Audi Electronics Venture GmbH, eine 100 prozentige Tochter der Audi AG, entwickelt.

Der enorme Vorteil dieser Umgebung ist die zeitsynchrone Erfassung, Aufzeichnung und Evaluierung aller Daten aus dem Fahrzeug, zudem wird durch eine Vielzahl an Weiterverarbeitungsmöglichkeiten und Visualisierung der vorhandenen Daten eine flexible Plattform dargestellt. Diese Plattform findet im gesamten VW-Konzern regen

Einsatz in der Entwicklung. Dadurch werden vor allem Entwicklungszeit und -kosten eingespart.⁸

Egal ob im Bereich Lichttechnik, Fahrerassistenz oder autonomes Fahren bietet ADTF durch den modularen Aufbau, die dynamische Kalibrierung aller vorhandenen Informationsflüsse sowie die Wiederverwendbarkeit von einzelnen Software-Modulen und den zugehörigen Messdaten. Außerdem ist durch die vielen offenen Schnittstellen in der ADTF-Architektur, ein hohes Maß an Portabilität, das Übertragen von Software auf andere Betriebssysteme, vorhanden.

2.2.2 Funktionsweise ADTF

ADTF bietet nicht nur eine Entwicklungsplattform, sondern ist zugleich eine interaktive Arbeitsumgebung, die für den fahrzeugnahen Einsatz konzipiert wurde. Entwickler konzipieren Funktionen und Anwendungen und können diese per Drag & Drop aus einer Komponentenbibliothek beziehen. Im Zweifelsfall kann die Anordnung dieser Komponenten (sog. Filter) auch unmittelbar mit realen Daten aus dem Fahrzeug getestet werden.

Für die schnelle und reibungslose Einbindung der Filter wird das ADTF Plug-In Software Development Kit (SDK) verwendet. Diese Schnittstellen gewährleisten eine einfache Austauschbarkeit und Erweiterbarkeit der Filterbibliothek, welche schließlich abteilungs- oder unternehmensübergreifend genutzt werden kann.

ADTF Plug-In SDK beinhaltet mehrere Funktionalitäten, die im Folgenden näher beschrieben werden:

- Die Kapselung der Basisfunktionalitäten (z.B. Betriebssystemabstraktion) wird durch „adtf_util“ verwirklicht.
- Durch „ucom“ wird definiert wie die Vorgehensweise bei einem Systemaufbau auszusehen hat. Dies beinhaltet die Basisimplementierung von den Schnittstellen, die für die Nutzung der Filterarchitektur notwendig ist.
- Für die Verwendung der ADTF Funktionalität wird die Implementierung einer Schnittstelle von „adtf“ benötigt.
- „adtf_graphics“ definiert die Schnittstelle für die Verwendung von Graphiken innerhalb von ADTF.

⁸ Vgl. (Audi Electronics Venture GmbH, 2014)

Im Folgenden wird mittels Abbildung 12 der Zusammenhang aller ADTF-Bauteile verdeutlicht.

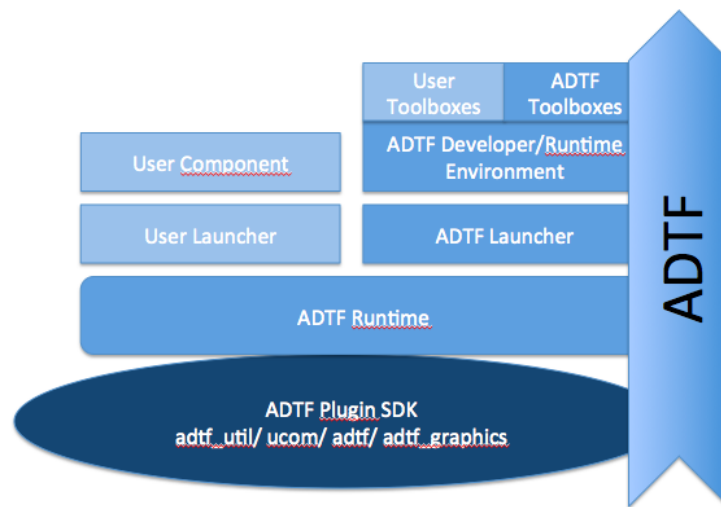


Abbildung 12 Funktionsweise ADTF – Übersicht

Wie der grafischen Darstellung zu entnehmen ist, ist das ADTF Plug-In SDK die grundlegende Architektur von ADTF, welche die Schnittstellen, die Komponenten und die Interaktion der Komponenten untereinander, definiert.

Die Interaktion der Module wird von der Laufzeitumgebung ADTF Runtime sichergestellt. Außerdem wird Plug-In SDK auf das jeweilige Betriebssystem angepasst (Linux, Windows, VxWorks).

Der ADTF Launcher erhält als Input eine System-Spezifikation und lädt anhand dieser Spezifikation bestimmte Funktionen, wie beispielweise Konfigurationsdienste oder Schreib- und Lesezugriffe.

Die Entwicklungsoberfläche besteht aus Bibliotheken und dem ADTF Framework und wird in der ADTF Developer Environment zur Verfügung gestellt. Das ADTF Framework wiederum umfasst die Header für die Entwicklung, sowie ADTF Runtime. Die Oberfläche ADTF Runtime Environment ist der Treiber, der die Nutzung der fertigen Konfigurationen erst ermöglicht.⁹

2.3 Filterbeschreibung

ADTF-Filter sind in der Programmiersprache C++ geschrieben. Ihre Ausführung findet im ADTF Framework statt, wobei eigens für den AADC ein neues ADTF Projekt erschaffen wurde, indem entsprechend neue Filter angelegt wurden.

⁹ Vgl. (Belke, 2009, S. 16)

Die mittels ADTF erstellte Basissoftware wird hierbei auf dem Messrechner, dem X2-Odroid-Board, verarbeitet. Wie einleitend erwähnt ist darüber hinaus die Software in eine Sensoren- sowie Aktoren-Seite aufgeteilt. Der Communication Filter übernimmt hierzu die Realisierung einer Regelschleife mittels Verbindung zwischen den lokal getrennten Aktoren und Sensoren. Die Hauptfunktion, physikalische Werte anzunehmen, diese zu verarbeiten und danach wieder in Form von physikalischen Werten auszugeben, soll durch diese Basissoftware im Zusammenspiel mit dem Gesamtsystem gewährleistet werden.

In den folgenden Unterkapiteln werden dabei die Funktionsmerkmale der einzelnen Filter näher erläutert.

2.3.1 Arduino Communication

Der Arduino Communication Filter bildet die Verbindung zwischen Arduino- und X2 Odroid Board und versetzt somit beide Komponenten in die Lage eine Informationsübermittlung aufzubauen.

Als Schnittstelle zu den Sensoren und Aktoren implementiert der Filter das zur Kommunikation notwendige Übertragungsprotokoll zwischen ADTF und Arduino. Dabei nimmt der Eingang die Befehle der Aktoren an, wobei der Ausgang die Sensorsignale im festgelegten Protokoll liefert.

Abbildung 13 zeigt den Aufbau des Übertragungsprotokolls, was im Folgenden näher erläutert wird. Der Arduino Frame ist in seiner Struktur in ein Header- sowie Daten-Segment gegliedert, wobei das Header-Segment sich nochmals unterteilen lässt. Das Arduino-Protokoll beginnt mit dem Start-of-Frame-Byte. Dieses Datenfeld dient der Kennzeichnung eines neuen Datenpaketes, welches aus einzelnen Datenfeldern besteht. Das darauffolgende ID-Byte gibt hierzu an, welche Nutzdaten das Paket enthält. Entweder enthalten diese die Werte der Sensoren und werden von dem Arduino- zum X2 Odroid Board gesendet, oder diese Daten enthalten die Stellwerte für die Aktuatoren und werden vom X2 Odroid zum Arduino-Board gesendet.

Der Zeitstempel wird genutzt, um das Empfangen der zeitlichen Abfolge von Datenframes festzuhalten. Dieser Eintrag wird dann getätigt, wenn das Arduino-Board die Daten vom X2 Odroid-Board empfangen hat.

Das Datenfeld mit der Bezeichnung „dataLenght“ ist zugleich das letzte des Header-Segmentes und gibt die Anzahl der Bytes der zu übertragenden Nutzdaten an.

Im Daten-Segment dagegen befinden sich die Nutzdaten, die tatsächlich übertragen werden.

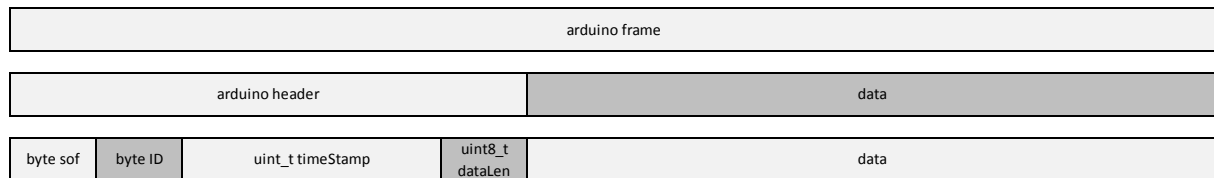


Abbildung 13 Arduino Protokoll

2.3.2 Arduino Sensors

Der Arduino Sensors Filter ist ein Teil der ersten Abstraktionsschicht, die die serielle Ausgabe des Arduino Communication Filters verwaltet. Dabei werden die Daten des Arduino Communication Filters entgegengenommen und die im Protokoll enthaltenen Werte in einzelne Sensorarten aufgeteilt.

Im Übertragungsprotokoll sind dabei sieben verschiedene Sensor-Datenpakete spezifiziert, deren Aufbau in Abbildung 14 dargestellt wird. Dabei haben die Pakete des Lenkwinkel-, Raddrehzahl, Lage-, Infrarot-, Helligkeit- und Ultraschallsensors sowie die der Spannungsmessung eine unterschiedliche ID. Anhand dieser ist eine schnelle, effiziente und fehlerfreie Zuordnung der Daten möglich.

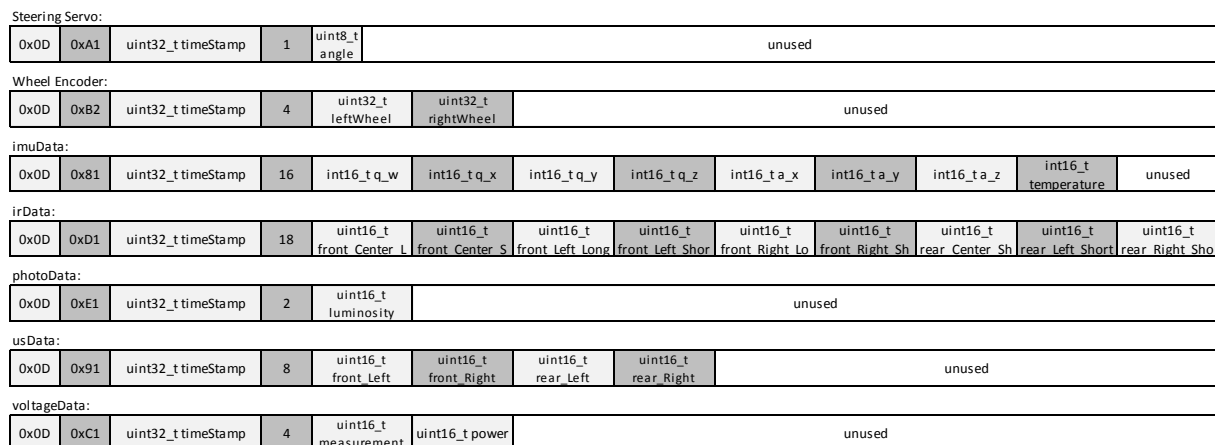


Abbildung 14 Sensor Datenpakete

2.3.2.1 Ultrasonic Signals

Dieser Filter hat die Aufgabe die Sensordaten der Ultraschallsensoren ortsbezogen aufzuspalten. Somit gehen in den Input-Pin die Daten der Sensoren direkt ein. Anschließend wird der Datenstrom in vier Output-Pins aufgespalten, die genau zugeordnet sind auf den jeweiligen Sensor. Der Filter hat ausschließlich Rohdaten

als Output, d.h. es muss mindestens ein Filter für die Aufbereitung der Daten vorhanden sein (Calibration Scaling).

2.3.2.2 Infrared Signals

Der Filter ist in seiner Funktionsweise identisch mit dem „Ultrasonic Signals“-Filter, jedoch werden in diesem Fall keine Ultraschall-Daten zugeordnet sondern Infrarot-Daten. Das Fahrzeug besitzt insgesamt neun Infrarot-Sensoren und dementsprechend sind neun Output-Pins vorhanden. Auch hier handelt es sich noch um Rohdaten die nachfolgend mit einem Kalibrierungsfilter aufbereitet werden müssen.

2.3.2.3 Steering Servo Signals

Der Lenkwinkel wird über ein Potentiometer bestimmt. Wenn das Lenkrad maximal eingeschlagen ist, dann wird ein bestimmter Spannungswert geliefert. Der Spannungswert geht als analoger Input-Wert in den „Steering Servo Signals“-Filter ein und wird innerhalb des Filters zu einem digitalen Integer-Wert umgewandelt. Anschließend wird der Integer über einen „Calibration Filter Extended“ auf einen Winkel (Float-Wert) umgerechnet.

2.3.2.4 Accelerometer Signals

Bei dem Beschleunigungsfilter handelt es sich wieder um dieselbe Funktionsweise wie bei dem „Ultrasonic- oder Infrared Signals“ Filter. Die Sensor-Daten werden auf die einzelnen Achsen bezogen und geben nach der Aufbereitung der Rohdaten, die aus diesem Filter hervorgehen, Auskunft über ein mögliches Gieren, Nicken oder Wanken des Fahrzeugs. Durch einen „Calibration Scaling“ Filter werden die Integer-Werte, der Sensoren auf die Beschleunigung in Meter pro Sekunde zum Quadrat umgelegt. Hierbei entspricht der Wert von 8000 der Erdbeschleunigung von einem g. Die daraus definierte Formel 2 hat somit eine Konstante zur Folge, mit der die Beschleunigungswerte multipliziert werden müssen, um korrekt skaliert zu werden.

Formel 2 Kalibration Accelerometer Signals

$$\frac{9,81 \text{ m/s}^2}{8000} = 0,00122625$$

2.3.2.5 Gyroscope Signals

Dieser Filter verarbeitet den Datenstrom aus dem Gyroskop. Die Daten werden zu Quaternionen umgerechnet und auf vier Output-Pins aufgeteilt. Mit Quaternionen kann man die Bewegung eines Objekts im dreidimensionalen Raum besser berechnen, insbesondere dann, wenn es um Drehbewegungen geht.

2.3.2.6 Wheel Speed Sensor Signals

Die Raddrehzahlsensoren erfassen die Raddrehzahl indem die Wechselhäufigkeit der Schwarz-Weiß-Übergänge in der Felgeninnenseite mittels einer Zählervariablen festgehalten wird. Diese Daten werden dann in den Filter auf einen Input-Pin gegeben und innerhalb des Filters der rechten und linken Seite des Fahrzeugs zugeordnet. Die zwei Output-Pins geben die unbehandelten Rohdaten der Sensoren aus, daher muss mit einem „Calibration“ Filter nachbereitet werden.

2.3.2.7 System Voltage Signals

Die Eingangsdaten von diesem Filter stammen direkt von dem Arduino Board. Über einen Spannungsteiler wird die Spannung ermittelt, was in Formel 3 dargestellt wird. Innerhalb des Filters findet mittels der Formel 4 eine Aufspaltung in die Spannung des Messkreises und des Steuerungs- und Antriebskreises statt.

Formel 3 Spannungsteiler Systemspannung

$$U_{Kalib} = U_{mess} \times \frac{R_1 + R_2}{R_2} = U_{mess} \times \frac{4,7k\Omega + 15k\Omega}{4,7k\Omega}$$

Formel 4 Kalibration Systemspannung

$$c = \frac{4,7k\Omega + 15k\Omega}{4,7k\Omega} \times \frac{3,3V}{1023} = 0,013521V$$

2.3.3 Arduino Actuator

Der „Arduino Actuator“-Filter macht die Ansteuerung der Aktuatoren im Fahrzeug möglich. Beispielsweise wird ein boolescher Wert, verwendet um das Licht anzusteuern, da es hier nur zwei Zustände (An und Aus) gibt. Bei der Beschleunigung wird ein Float-Wert übergeben, da die Gaspedalstellung viele Zwischenwerte hat. Der Filter hat insgesamt neun Input-Pins. Unter anderem werden so die Bremsleuchten oder die Blinker links und rechts angesteuert. Der einzelne Output-Pin des Filters gibt die Signale weiter und steuert so die jeweiligen Aktuatoren an.

In der nachfolgenden Abbildung 15 ist das Übertragungsprotokoll des Filters dargestellt.

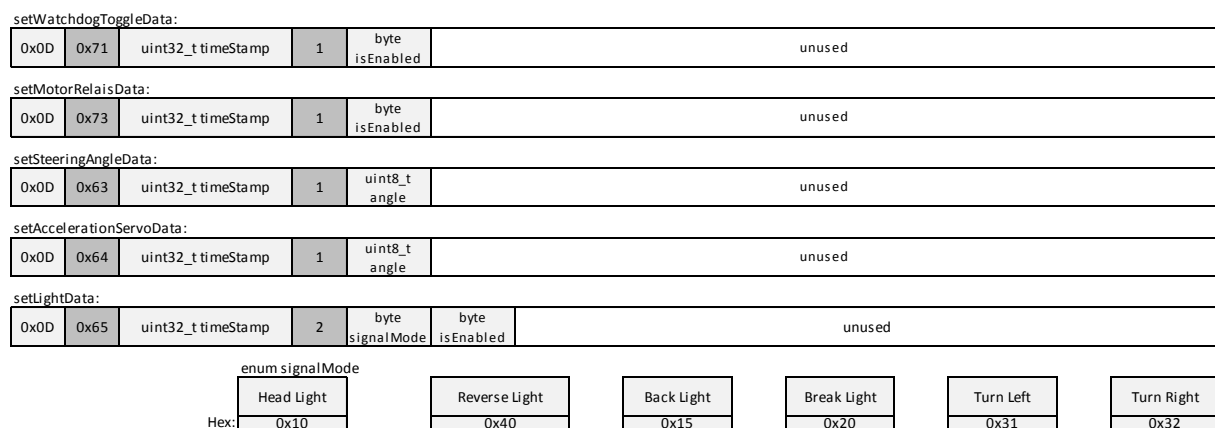


Abbildung 15 Aktuator Datenpakete

Der Aufbau eines Übertragungsprotokolls wird in Abschnitt 2.3.1 erläutert.

In dem Protokoll des Aktuator-Filters ist der Watchdog-Datensatz vorhanden, der periodisch an das Arduino-Board gesendet wird (siehe Abschnitt 2.3.8). Zusätzlich werden die Stellgrößen Lenkwinkel und Gaspedalstellung sowie die Ansteuerung des Lichtes und des Motorrelais durch unterschiedliche Bus-Adressen gekennzeichnet.

2.3.4 RPM Calculation

Dieser Filter ist der „Calibration“-Filter für die Raddrehzahl. Die Zählervariablen aus dem „Wheel Speed Sensor Signals“-Filter werden übergeben und in diesem Filter mittels eines Sliding-Window-Filters, welcher in definierten Zeitperioden die Zählerwerte auswertet, umgerechnet. Der Filter besitzt zwei Eingänge sowie Ausgänge. Die Ausgangswerte sind in Umdrehungen pro Minute angegeben.

2.3.5 Attitude Converter

Die Eingangswerte des „Attitude Converter“-Filters sind die vier Outputs des „Gyroscope Signals“-Filters. Die Quaternionen-Koordinaten werden übergeben und in Euler Winkel konvertiert. Es gibt zwar viele verschiedene Euler Systeme, jedoch ist für die Lage des Fahrzeugs nur das Yaw-Pitch-Roll-System (Roll-Nick-Gier-Winkel) von Relevanz. Folglich beschränkt sich die Konvertierung auf dieses eine System. Die drei Output-Winkel sind im Bogenmaß angegeben. Für die Umrechnung in Grad kann ein einfacher „Calibration Scaling“-Filter, mit der vorhandenen Formel 5, angehängt werden.

Formel 5 Umrechnung Bogenmaß in Grad

$$\alpha^{\circ} = \frac{180^{\circ}}{\pi}$$

2.3.6 Calibration Scaling

Dieser Filter multipliziert den Eingangswert mit einem zuvor definierten Faktor und sorgt so für die richtige Skalierung. Zum Beispiel der Integer des Beschleunigungs-Filters hat eine Skalierung bei der ein g einem Integer-Wert von 8000 entspricht. Um die gegebene Größe in eine verwertbare Einheit umrechnen zu können braucht man einen „Calibration Scaling“-Filter.

2.3.6.1 Calibration Filter Extended

Für die Aufbereitung mancher Daten ist eine lineare Skalierung nicht ausreichend, daher werden die Messdaten, die dem Filter zugeführt werden, interpoliert. Das bedeutet, es wird über einen komplizierten Algorithmus, eine definierte Funktion $f(x)$ errechnet, die die Messdaten als Stützstellen beinhaltet.

Die Funktion $f(x)$, auch Spline genannt, wird in der folgenden Grafik exemplarisch dargestellt.

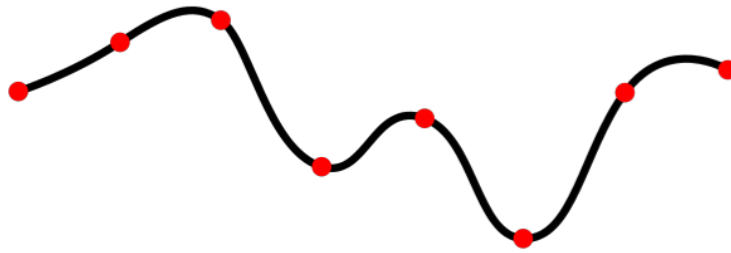


Abbildung 16 Spline mit acht Messdaten

Der Input dieses Filters sind meistens vorbehandelte Rohdaten, die ohne eine Darstellung als Polynom nicht weiterverarbeitet werden können. Zu diesem Zweck ist der Output dieses Filters ein Spline (Abbildung 16), welches die Input-Daten beinhaltet.

2.3.7 Xtion Camera

Die Daten der Kamera werden über diesen Filter eingestellt. Daher beinhaltet er eine Konfigurationsdatei, die sowohl die gewünschte Auflösung als auch die Bildrate entsprechend beeinflusst.

Die Konfigurationsdatei ist folgendermaßen aufgebaut:

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
<xtionSettings xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <depthImage>
    <x_resolution>640</x_resolution>
    <y_resolution>480</y_resolution>
    <fps>30</fps>
  </depthImage>
  <colorImage>
    <x_resolution>640</x_resolution>
    <y_resolution>480</y_resolution>
    <fps>30</fps>
  </colorImage>
  <colorImageLowRes>
    <x_resolution>320</x_resolution>
    <y_resolution>240</y_resolution>
    <fps>30</fps>
  </colorImageLowRes>
  <options>
    <setDepthColorSync>0</setDepthColorSync>
```

```
<setRegistration>0</setRegistration>  
</options>  
</xtionSettings>
```

2.3.8 Watchdog Trigger

Dieser Filter ist ein Überwachungsfilter, welcher in immer gleichem Zeitabstand ein Signal an den Watchdog (Komponente im Arduino) sendet. Fällt dieses Signal aus oder es kommt aus irgendeinem Grund nicht im Arduino an, wird ein Fehler in der Kommunikation des Bordnetzes detektiert und das Relais für die Antriebselektronik wird abgeschaltet, d.h. das Fahrzeug bleibt abrupt stehen.

2.3.9 Jury Module

Dieser Filter ist zum Zweck des AADC-Wettbewerbs entwickelt worden und soll von den Juroren verwendet werden, um die Grundfunktionen und –manöver des Fahrzeugs zu testen und zu bewerten. Hierfür gibt es eine speziell vorgegebene Manöverliste, welche die Reihenfolge der Manöver ohne zeitlichen Rahmen vorgibt. Diese Liste kann in etwa so aussehen:

- Rechts abbiegen
- Einparken
- Überholen
- Anhalten

Außerdem wird in dem Filter ein Sektor definiert, d.h. es wird ein Streckenabschnitt, welcher zugleich eine Zwischenwertung beinhaltet, vorgegeben in dem bestimmte Manöver und/oder Reaktion auf Hindernisse geprüft werden.

Über das sogenannte NotAusFlag kann ein abrupter Stillstand des Fahrzeugs hervorgerufen werden, was einen Abbruch aller Manöver zur Folge hat.

3 Definition von Testfällen

Das präzise und sorgsame Testen von Softwarebausteinen wird häufig als Trivialität wahrgenommen, aber ebenso häufig falsch gedeutet und umgesetzt.

Eine fundierte Basis ist daher umso wichtiger, um einen soliden Stand- oder Ausgangspunkt des Testens zu ermöglichen. Im Folgenden wird hierbei zunächst auf den Titel dieser Studienarbeit Bezug genommen, sodass die Intention, die Validierung von Softwarebausteinen zu erreichen, eine definierte Zielausrichtung erhält. Anschließend werden die einzelnen Testfälle in Anforderung sowie Inhalt beschrieben und definiert, womit das weitere praktische Vorgehen, logisch aufeinander aufbauend, entworfen und geplant werden kann.

3.1 Das V-Modell

Das V-Modell ist die eine etablierte Darstellungs-Form eines Entwicklungsprozesses des Software-Engineerings. Nach mehreren Weiterentwicklungen des Wasserfallmodells nach Royce wurde das V-Modell 1979 von Boehm entwickelt. Das Modell wird auf zwei Achsen aufgetragen, welche den Vertiefungsgrad und den zeitlichen Fortschritt beschreiben. Daher ist ein Aufteilen in verschiedene, korrelierende Testphasen möglich.

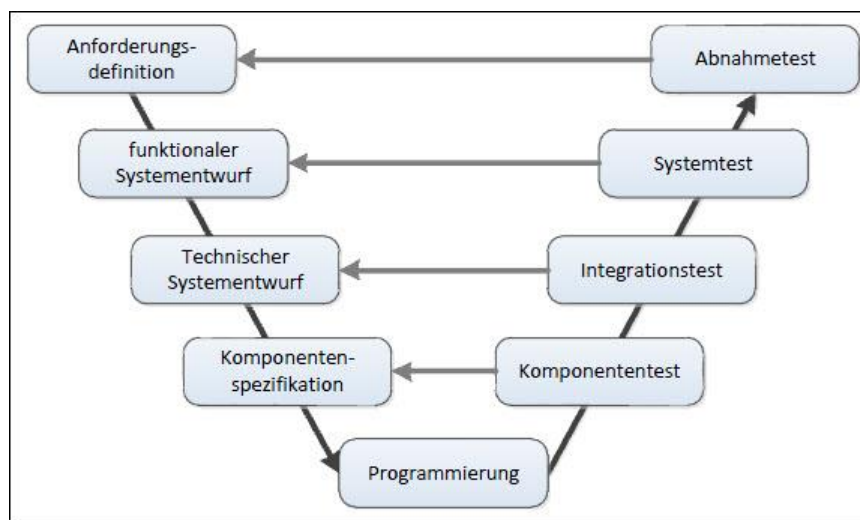


Abbildung 17 Das V-Modell

Auf dieser Darstellung des V-Modells sind die einzelnen Phasen als abgerundete Rechtecke dargestellt. Die Phasen verlaufen sequenziell in Pfeilrichtung. Die Anforderungsdefinition ist die erste Phase, hier werden die Anforderungen an die

Software formuliert, welche später die grundlegenden Kriterien für die Abnahme des Produkts bilden.

In der zweiten Phase werden die Anforderungen an das Produkt auf einen konkreten Systementwurf angewandt. Dabei geht es ausschließlich um die Funktion, die mit der Software erfüllt wird. Weitergehend wird der Systementwurf vertieft, indem die technischen Möglichkeiten miteinbezogen werden. Im nächsten Schritt wird eine Detailierungsebene tiefer gegangen, d.h. es werden die einzelnen Komponenten, die von dem System beinhaltet werden sollen, betrachtet.

Nach der Planung erfolgen die eigentliche Programmierung und das Umsetzen der zuvor festgelegten Systemspezifikationen.

In Abbildung 17 ist ersichtlich, dass nach der Programmierung die ganzen Prüfverfahren und Tests folgen. Diese Seite des V-Modells wird auch „Validierung“ genannt.

Eine Validierung in der Informatik bezeichnet einen Prüf-Vorgang, bei dem die Tauglichkeit der Software, d.h. die Funktionalität, in Bezug auf den jeweiligen Verwendungszweck, festgestellt wird. In der Umgangssprache könnte man bei der Validierung fragen: „Wird das richtige Produkt entwickelt?“

Der Komponenten- und Integrationstest sind die Hauptbestandteile des V-Modells, die in dieser Studienarbeit relevant sind und werden in den nachfolgenden Kapiteln erläutert.

Bei dem Systemtest und dem finalen Abnahmetest werden die zuvor definierten Systemspezifikationen mit den realen Gegebenheiten verglichen. Ist alles in Ordnung wird die Freigabe erteilt.

3.1.1 Komponententest

Ein Komponententest, auch Unit- oder Modultest genannt, ist das Abprüfen der kleinsten testbaren Einheit eines Computerprogramms. Das bedeutet es werden einzelne Funktionen oder Klassenmethoden getestet. Diese Vorgehensweise hat den Vorteil, dass sich sehr einfach automatische Tests implementieren lassen, die bei Bedarf immer wiederholt werden können.

Ein typischer Komponententest beinhaltet das Konfrontieren der zu testenden Funktion oder Methode mit einer Reihe von Testparametern. Anschließend wird die

Reaktion des Moduls, d.h. die Ausgabewerte, mit der zu erwartenden Verhaltensweise (Soll-Werte) verglichen.

Ein Test besteht in der Regel aus einer ganzen Reihe von Testfällen, die viele unterschiedliche Parametersätze prüfen. Wie umfangreich diese Tests letztendlich sind, entscheidet der Entwickler selbst. Meistens werden spezielle Fälle, wie zum Beispiel Grenzwerte, Null-Pointer oder andere spezielle Zustände, die das System instabil werden lassen könnten, betrachtet.

Ein übliches Vorgehen beim Erstellen solcher Tests ist, dass zu jeder Klasse eine Testklasse konstruiert wird. Auf diese Weise können die zu testenden Methoden der Klasse verifiziert werden. Zusätzlich besteht die Möglichkeit, mittels unabhängiger Testklassen, die in sog. Test-Suiten zusammengefasst werden, ganze Softwaresysteme zu testen, um somit die einwandfreie Kooperation der Module sicherzustellen.

Ein weiterer Vorteil der Komponententests ist das „saubere“ Programmieren, welches die Qualität und die Übersichtlichkeit des Codes garantiert. Denn ein zu komplexer Source-Code führt schnell zu Komplikationen beim Implementieren des Komponententests.

Allerdings sollte im Hinterkopf behalten werden, dass bei einer falschen Implementierung des Tests oder einer unglücklichen Wahl der Test-Parameter, der Test ein positives Ergebnis ausgibt, welches unter Umständen nicht positiv ist.¹⁰

3.1.2 Integrationstest

Der Integrationstest behandelt viele verschiedene Aspekte der Software-Qualität. Während bei einem Komponententest (siehe Abschnitt 3.1.1) die Schwerpunkte auf der inneren Software-Qualität (zum Beispiel Übertragbarkeit) liegen, liegt bei einem Integrationstest der Fokus in der äußeren Software-Qualität. Äußere Software-Qualität kennzeichnet sich durch Aspekte wie Funktionalität und Zuverlässigkeit. Ein intuitives User-Interface zählt ebenfalls zu den Merkmalen der äußeren Qualität einer Software-Architektur. Ein Integrationstest kann auf vielen verschiedenen Detaillierungsebenen stattfinden. Es können einzelne Funktionen und Methoden, die

¹⁰ Vgl. (Müller, 2008)

in eine Klasse zu integrieren sind, oder ganze Systeme und Teilsysteme, welche miteinander verbunden oder verschachtelt sind, betrachtet werden.¹¹

In einem Integrationstest wird also das reibungslose Zusammenspiel von Hardware- und Software-Modulen in einem Verbund getestet.

Eine Bedingung die vor dem Integrationstest erfüllt sein muss, ist das positive Ergebnis des Komponententests. Erst wenn die Software-Komponenten autark funktionstüchtig sind können die Anforderungen an das Gesamtsystem geprüft werden.

Jede Verbindung zweier Komponenten wird in einem Testfall für den Integrationstest geprüft. Hierbei werden nicht nur die Schnittstellen und die korrekte Kommunikation überprüft sondern auch die Fehlerbehandlung von Ausnahmesituationen.

3.2 Komponententest der ADTF Filter

Die Umsetzung des Komponententests dieser Studienarbeit erfolgt auf der erarbeiteten Grundlage aus Abschnitt 3.1.1. Als kleinste prüfbare Einheit der zu testenden Basissoftware stellen sich dabei die einzelnen Filter dar, wie sie zu Beginn in Abbildung 2 zu sehen sind. Diese werden eingangsseitig (Input) mit Daten beaufschlagt, um in einem automatischen Testfall das daraus resultierende Ergebnis mit den zu erwartenden Soll-Daten abzugleichen.

Im Allgemeinen wird hierbei von einem Quelle-Senke-Prinzip gesprochen, indem die Quelle den Ursprung des Datenflusses und die Senke den dazugehörigen Zielort beschreibt.^{12,13}

Je nach Filter werden die Eingänge durch die Variation an Parametersätzen abgeprüft, da jeder Filter sich aus unterschiedlichen Eingängen zusammensetzt. Folgerichtig können diese Eingänge nur in jener Anzahl abgeprüft werden, in der sie existieren. Die Testparameter werden dabei durch das jeweilige Testprogramm bereitgestellt und an den Filter übermittelt.

Auch Grenzfälle sollen möglichst effizient abgeprüft werden. Dabei richtet die Grenzfallbetrachtung ihren Fokus vorrangig auf die Variation von verschiedenen Datenraten sowie auf Grenzen der zulässigen beziehungsweise unzulässigen Werte.

¹¹ Vgl. (Mario Winter, 2012, S. 9 ff.)

¹² Vgl. (Feess, 2014)

¹³ Vgl. (Krieger, 2014)

Ein Test, der auf Grund seines halbautomatischen Ablaufs von den anderen zu unterscheiden ist, ist der des Arduino Communication Filters. Die dabei entstehende Differenz des Testaufbaus wird in dem gleichnamigen Kapitel 3.2.1 nochmals detailliert erläutert.

3.2.1 Arduino Communication

Der Arduino Communication Filter stellt, wie bereits erwähnt, die serielle Verbindung zum Arduino Board her und differenziert sich in seinem Testablauf von anderen Filtern. Aus diesem Grund ist es nicht möglich, ein klassisches Quelle-Senke-Prinzip anzuwenden, da hier Ein- und Ausgang nicht einen direkten Zusammenhang in Ihrer Verbindung darstellen. Zur Ansteuerung der Aktoren werden dabei Eingangssignale zur Verarbeitung an das Arduino Board mittels USB-Schnittstelle weitergegeben. In umgekehrter Richtung werden analog zum eben beschriebenen Prozess der Aktuator-Signale die Sensor-Signale vom Arduino- an das Odroid-Board gesendet. Ein korrekter Protokollaufbau ist sowohl Ein- sowie Ausgangsseitig zwingend notwendig, um die Datenströme noch richtig in die verschiedenen Signale zu untergliedern.

Auch ein vollautomatischer Test ist nicht möglich. Die Testeingangssignale, die schon als korrekt aufgebautes Protokoll vom Communication Filter an das Arduino Board gesendet werden, steuern in unterschiedlichster Weise die Aktuatoren an. Werden dabei die verschiedenen LEDs angesteuert, so kann nicht softwareseitig überprüft werden, ob diese wirklich angeschaltet wurden, da keine Sensortechnik verwendet wird, die diese Ansteuerung zurückprüft. Diese Kontrolle oder auch Sichtprüfung wird manuell von dem Tester vorgenommen, der mit dem Testablauf vertraut ist.

Der grundsätzliche Testablauf beinhaltet das Beaufschlagen des Eingangs mit den zu erwartenden Eingangssignalen. Hierbei werden zunächst die Geschwindigkeitswerte in vollem Umfang abgeprüft. Dieser involviert das Vorwärts- und Rückwärtsfahren und das über den vollständigen Bereich der Geschwindigkeitswerte. Der vollständige Wertebereich wird dabei gleichmäßig über eine lineare Rampe abgefahren. Ebenso wird der Lenkwinkelwertebereich über eine lineare Rampe abgedeckt. Der Unterschied zwischen beiden Rampen liegt folgerichtig an der Art der Werte, die sich durch Geschwindigkeit und

Lenkwinkелеinschlag charakterisieren. Auch alle Lampen des Fahrzeugs werden angesteuert. Die Lampen gliedern sich in Front-, Heck-, Brems-, Rückfahrlicht sowie Blinker. Abschließend wird der Arduino Communication Filter mit einem Datenstrom beaufschlagt, der die drei genannten Eingangssignale zusammenhängend versendet und nicht wie zu Beginn separiert.

Die Grenzfallbetrachtung des Communication Filter Test konzentriert sich dabei in der Anforderung auf zwei Fälle. Zum einen auf das Definieren einer grenzwertigen Datenrate und zum anderen auf das Auswerten der beiden somit entstandenen Datenratenbereiche.

Das Testprogramm wird nach Fertigstellung zunächst so verwendet und eingesetzt, um den Grenzwert der Datenrate experimentell zu ermitteln. Dazu wird eine sogenannte Sleep-Zeit verwendet, die durch Hinzufügen von Werten die Taktrate verlangsamt.

Wurde der Grenzbereich der Datenrate festgestellt, so wird die nächste Anforderung überprüft. Hierbei findet erneut eine Untergliederung statt, indem einerseits die Werte oberhalb und andererseits unterhalb der Datenratengrenze abgeprüft werden. Oberhalb der grenzwertigen Datenrate werden die Datenpakete, die sogenannten Mediasamples, lediglich in ihrem Auftreten überprüft. Hierbei soll festgestellt werden, ob überhaupt noch Mediasamples verarbeitet und weitergeleitet werden. Dieser Test stellt sozusagen einen Stresstest für das Arduino-Board dar, welcher im Worst-Case bei Überbeanspruchung keine Sensordaten mehr absetzen kann. Unterhalb der grenzwertigen Datenrate werden die Mediasamples nicht nur in ihrem Auftreten überprüft, es wird des Weiteren auch nachvollzogen, ob die Werte der Geschwindigkeit und des Lenkwinkels richtig vom Arduino umgesetzt wurden. Dazu ist es notwendig die zurückgelesenen Sensordaten des Arduino-Boards über Lenkwinkel- und Drehzahlsensor zu erfassen. Die Sensordaten müssen dabei ebenso, analog zu den zu beaufschlagenden Eingangswerten, in Ihrem Ergebnis eine lineare Rampe darstellen.

3.2.2 Arduino Actuator

Wie bereits in Kapitel 2.3.3 erwähnt ist der Aktuator Filter für das korrekte Ansteuern der Aktuatoren zuständig. In diesem Fall lässt sich eine herkömmliche Testumgebung aufbauen, die nach dem Quelle-Senke-Prinzip funktioniert. Das

bedeutet, dass vor den zu testenden Filter eine Quelle implementiert wird, die den Filter mit ausgewählten Test-Daten konfrontiert. Anschließend wird in der Senke, die sich nach dem Filter befindet, die Reaktion des Filters ausgewertet.

Der Test kann vollautomatisch programmiert werden, d.h., dass der Test später in der Anwendung ohne weiteres Zutun des Benutzers ausgeführt wird.

Bei dem Aktuator-Filter wird der Protokollaufbau von verschiedenen Aktuatoren überprüft. Nach dem Start-of-Frame-Bit wird geprüft ob die ID, der Zeitstempel und die Datenlänge richtig angefügt sind. Die Nutzdaten, also die relevanten Informationen, sollen anschließend über einen Algorithmus auf ihre Plausibilität geprüft werden.

Von der Quelle aus werden die Daten für die Lenkung, Beschleunigung und die Ansteuerung des Lichts sowie des Watchdogs und des Emergency-Stops auf die Pins des Filters gegeben. Dabei sind die Werte, die übergeben werden, auf einen bestimmten Werte-Bereich begrenzt. Der Lenkwinkel zum Beispiel ist begrenzt auf 65 bis 125 oder die Beschleunigungswerte liegen zwischen 0 und 180. Diese dimensionslosen Werte werden durch einen Calibration Scaling Filter später in die entsprechende Einheit überführt. Für welchen Aktuator die Daten letztendlich bestimmt sind kann über den Pin-Namen und der daraus resultierenden Frame-ID ausgelesen werden.

Das Fahrzeug besitzt fünf verschiedene Licht-Modi, die angesteuert werden können. Das reguläre Licht umfasst die zwei Front- und Heckleuchten. Zur Richtungsanzeige gibt es einen Blinker rechts und links sowie ein Brems- und Rückfahrlicht.

Das generelle Ziel dieses Tests ist das Sicherstellen der vollumfänglichen Funktionsfähigkeit des Filters. Das beinhaltet das Beaufschlagen des kompletten Wertebereiches und der Grenzwerte. Darüber hinaus muss mit einer ausreichenden Periodizität getestet werden. Das bedeutet, die Schrittweite, die den Umfang an Test-Werten ausmacht, muss an den jeweiligen Testfall angepasst werden.

3.2.3 Arduino Sensors

Der Arduino Sensors Filter ist, betrachtet man die Filterarchitektur, das Gegenstück zu dem Aktuator Filter. Das bedeutet, dass es einen Input-Pin und mehrere Ausgänge gibt. Daher ist die Test-Struktur ebenfalls entgegengesetzt zu implementieren.

Wie bereits in Abschnitt 2.3.2 beschrieben, teilt der Arduino Sensor Filter das empfangene Übertragungsprotokoll des Arduino Boards in die diversen Sensor-Arten auf. Um einen Test-Fall zu erzeugen, muss es also eine Quelle geben, die die Daten des Boards simuliert. Zur Veranschaulichung des Test-Prinzips wird in Abbildung 18 die Filterarchitektur schematisch dargestellt.

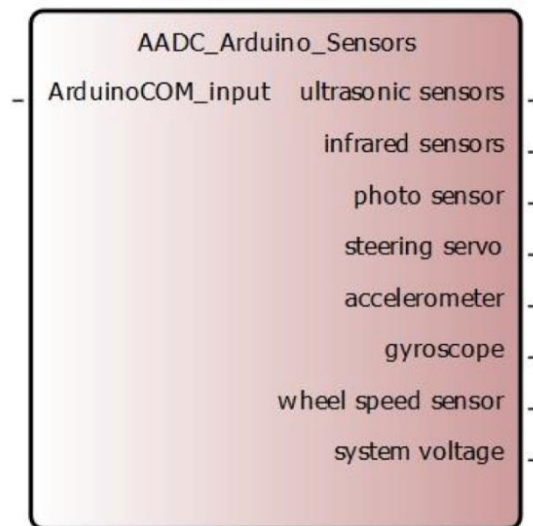


Abbildung 18 Schematische Darstellung des Sensor-Filters

In diesem Fall gibt es zwei Möglichkeiten eine Senke an den Filter anzuhängen. Entweder es wird pro Ausgang eine Senke installiert oder es wird eine universelle Senke implementiert, die im Test-Betrieb abwechselnd an die Output-Pins gehängt werden kann.

Neben den Nutzdaten werden wie bei dem Aktuator-Test die restlichen Bits des Übertragungsprotokolls (SOF, Timestamp etc.) auf ihre Richtigkeit überprüft.

Auch hier ist es Ziel die Funktionsfähigkeit des Filters sicherzustellen. Die Herausforderung hierbei besteht in dem Generieren des abzuprüfenden Wertebereiches und der jeweiligen Grenzfallbetrachtung.

3.3 Integrationstest der ADTF Filter

Die Realisierung des Integrationstest der drei ADTF-Filter richtet sich stark nach dem Verlauf des Komponententests. Diese müssen zunächst erfolgreich durchgeführt werden, bevor die jeweiligen Komponenten zu einem Teilsystem integriert und abgeprüft werden können.

Der Aufbau grenzt sich im Vergleich zur Definition in Abschnitt 3.1.2 etwas ab. Nicht nur die äußere Softwarequalität soll abgeprüft werden, sondern auch die Innere.

Dazu werden einzelne Untermodule der jeweiligen Filter so miteinander verknüpft, dass die Eingangsseite des Filters A mit der Ausgangsseite des Filters B korreliert. Der Vorgang orientiert sich hierzu wieder nach dem Quelle-Senke-Prinzip, wie schon in Abschnitt 3.2 beschrieben. Allerdings werden hier die Ein- und Ausgänge nicht jeweiligen einzelnen Filtern betrachtet, sondern diese nach der entsprechenden Modulzusammenschaltung beaufschlagt und ausgewertet. Ein- und Ausgänge innerhalb des Moduls sollten dabei problemlos die jeweiligen Signale zwischen den Filtern verarbeiten und weiterleiten. Somit kann ein Modul in seiner Funktionalität abgeprüft werden.

Um auch Aspekte der äußeren Softwarequalität zu prüfen, werden nicht nur die eben genannten Schnittstellen und die Kommunikation getestet, sondern auch Grenzwertbetrachtungen vorgenommen, die auf die Zuverlässigkeit des Moduls und der Software schließen lassen.

Ziel ist es, somit das Zusammenspiel von Softwarebausteinen, in Form von den drei behandelten Filtern, mit den Hardwaremodulen des Modellfahrzeugs im Verbund zu testen. Dabei ist zu erwarten, dass mittels dieser Integrationen keine Änderungen der Ergebnisse aus den bereits abgeprüften Komponenten entstehen. Die zu beaufschlagende Eingangsseite bildet der Arduino Actuator Filter, der die empfangenen Daten zu einem einheitlichen Datenstrom packen sollte, um diesen danach an den Arduino Communication Filter weiterzuleiten. Der Arduino Communication Filter steht nach wie vor in Austausch mit dem Arduino Board und sollte unterhalb des ermittelten Datenratenraten-Grenzwertes Sensordaten zurückliefern. Diese sollen darauffolgend von dem Arduino Sensors Filter in ihre einzelnen Signale aufgeteilt werden und anschließend nach dem bekannten Prinzip ausgewertet werden.

4 Filter-Tests

Anknüpfend und aufbauend an den dritten Abschnitt beschäftigt sich Abschnitt 4 mit der konkreten softwaretechnischen Umsetzung der Filter-Tests. Dabei werden die Inhalte von Abschnitt 3 angewendet und praktisch realisiert. Der dadurch entstehende Code wird zu jedem Filter-Test anhand seiner wichtigsten Bausteine aufgegriffen und detailliert erläutert.

4.1 Umsetzung der Komponententests

4.1.1 Arduino Communication

Der Arduino Communication Test setzt sich aus vier einzelnen Testfällen zusammen. Diese werden, wie in Abbildung 19 definiert, wobei diese Abbildung nur einen Testfall aufzeigt, und exemplarisch für die weiteren Tests steht.

Das `DEFINE_TEST` Makro besteht dabei aus verschiedenen Parametern. Hierzu wird zunächst in Zeile 594 eine Testklasse zugeordnet. Diese Testklassen werden in einem Test ausgeführt. Zeile 595 definiert hierzu den Methodennamen des Testfalls, der wiederum in der Testklasse ausgeführt wird.

Darüber hinaus werden mehrere Textbausteine bestimmt, die im Folgenden stichwortartig beschrieben und mit den entsprechenden Zeilennummern versehen werden, um die Nachvollziehbarkeit zu bewahren.

- 596: bezieht die Testfall-ID mit der ID-Nummer 2.2
- 597: definiert den Titel des Tests, der in diesem Beispiel Lenkwinkeltest genannt wird
- 598/599: Testbeschreibung – Sicherstellung, dass der Ausgang des Communication Filters funktionsfähig ist, wenn der Eingang mit verschiedenen Lenkwinkel-Werten beaufschlagt wird
- 600: beschreibt die Teststrategie, nach dieser soll der Communication Filter stets Daten senden
- 601: Beschreibt die Bedingung, was erfolgen muss, dass der Test bestanden ist
- 602: bietet Platz für zusätzliche Beschreibungen, die in diesem Fall nicht notwendig sind.

- 603: legt die Testanforderung fest – laut dieser Anforderung muss der Communication Filter stets in der Lage sein Lenkwinkel-Werte zu senden, während er Sensor-Werte erhält
- 604: bezieht sich auf die Testausführung, die in diesem Fall automatischer Herkunft ist

So wie eben die Parameter des Testmakros festgelegt wurden, beschreibt Zeile 605 bis 608 die tatsächliche Ausführung des Makros. Hierzu wird dem Communication Filter Test zwei notwendige Parameter übergeben. Einerseits die Parameter-ID, dass die Ausführung des Tests als Lenkwinkel-Test abläuft und zum anderen die bereits erwähnte Sleep-Zeit, die in diesem Fall willkürlich auf zehn Mikrosekunden festgelegt wurde. Die Sleep-Zeit stellt die Wartezeit zwischen den einzelnen Mediasamples dar und ist somit ein gewisser Zeitpuffer zwischen den verschiedenen Daten-Frames.

```
594 DEFINE_TEST(cTesterAADCArduinoComm,  
595             TestSteeringAngle,  
596             "2.2",  
597             "TestSteeringAngle",  
598             "This test makes sure, that the output of the communication filter works while the input"\  
599             "is set to different steering angle values",  
600             "The communication filter sends allways data",  
601             "See above",  
602             "none",  
603             "The filter must be able to receive steering data while delivering sensor data.",  
604             "automatic")  
605 {  
606     // steering angle pin with expected frameID  
607     return DoCommunicationFilterTest(ID_ARD_ACT_STEER_ANGLE, 10);  
608 }
```

Abbildung 19 Testdefinition SteeringAngle

Jeder Testfall des Arduino Communication Filters beginnt mit dem beaufschlagen des Eingangs mit den zu erwartenden Datenströmen von Geschwindigkeit, Lenkwinkel und den verschiedenen Lichtern, was bereits in Abschnitt 3.2.1 erwähnt wurde.

Eingangsseitig muss dazu zunächst ein Pin als Datenquelle erstellt werden, der den Zugriff auf den Arduino Communication Filter ermöglicht. Dieser Vorgang ist im Programmausschnitt der Abbildung 20 zu sehen und wird im folgenden Abschnitt erläutert.

Die Datenquelle wird, wie in Zeile 257 zu sehen, als generischer Output-Pin erstellt. Dabei braucht der Pin noch eine weitere Beschreibung, in der festgelegt wird, welche Daten er zu senden hat. In Zeile 258 wird dazu die Festlegung getroffen, in der die Methode *Create*, der Klasse Pin, zwei Parameter übergeben bekommt. Diese

Parameter prägen dem Pin ein spezielles Verhalten auf. Einerseits wird der Name des Pins mit *sampleSource* bestimmt und andererseits wird mit der Methode *cMediaType* der gewünschte Datentyp erzeugt. Bei der Methode *cMediaType* ist darauf zu achten, dass die korrekte Datenbeschreibung gewählt wird, da der Konstruktor der Klasse *cMediaType* anhand des Namens erkennt, wie er sich zu initialisieren hat.

Zugleich wird in Zeile 261 der Input-Pin des Communication Filters, mit dem eben erstellten Pin verknüpft.

```
256 | // create a simple sample source
257 | cObjectPtr<cOutputPin> pSampleSource = new cOutputPin();
258 | __adtf_test_result(pSampleSource->Create("sampleSource", new cMediaType(0,0,0,"tArduinoData")));
259 |
260 | // connect the Input pin of the communication filter with the sample source
261 | __adtf_test_result(pComInputPin->Connect(pSampleSource));
```

Abbildung 20 Erstellen des Output-Pins

Nachdem die Erstellung des Pins erläutert wurde, wird auf die Hilfsfunktionen eingegangen, die diesen Pin bedienen.

Die Hilfsfunktion *transmitMediaSample* ist für den Protokollaufbau zuständig und stellt den eigentlichen Sender dar. Wie in Zeile 165 zu sehen ist, müssen dieser Funktion zwei Arten von Parametern übergeben werden. Einerseits der Parameter *pSampleSource* als Objekt, welches die *MediaSamples* sendet. Mittels dieses Parameters weiß die Hilfsfunktion über welchen Pin Daten gesendet werden sollen. In diesem Fall ist das der zuvor erklärte und erstellte Output Pin, der an den Input Pin des Arduino Communication Filters gehängt wurde. Andererseits werden die Parameter übergeben, die die Daten darstellen, die wiederum im *MediaSample* vorhanden sein sollen. Im den folgenden Stichpunkten werden diese knapp aufgeführt und erklärt:

- *ch8ID*: ID des Frames
- *i8DataLength*: Datenlänge der Nutzdaten
- *chData*: tatsächlichen Nutzdaten

Abbildung 21 zeigt den ersten Teil der Hilfsfunktion *transmitMediaSample* und behandelt das generelle Erstellen eines *MediaSamples*. Zunächst wird je nach Filter der passende *MediaType* festgelegt. Der *MediaType* stellt die Datenbeschreibung bzw. den Datentyp dar. Falls der *MediaType* nicht erkannt werden kann, fängt eine triviale Abfrage diesen Fall ab, was in Zeile 169 zu sehen ist. Ebenso wird weiterhin

die Typbeschreibung festgelegt. Dieser Programmablauf ist notwendig, um *MediaSamples* in ihrer Grundstruktur zu erstellen.

```

165 tTestResult transmitMediaSample(cOutputPin *pSampleSource, tInt8 ch8Id, tInt8 i8DataLength, const tChar *chData)
166 {
167     // get the mediatype of the sample source
168     cObjectPtr<IMediaType> pType;
169     if(IS_FAILED(pSampleSource->GetMediaType(&pType)))
170     {
171         __adtf_test_ext(tFalse, "unable to get mediatype");
172     }
173
174     // get the type description
175     cObjectPtr<IMediaTypeDescription> pTypeDesc;
176     pType->GetInterface(IID_ADTF_MEDIA_TYPE_DESCRIPTION, (tVoid**)&pTypeDesc);
177
178     // get the serializer from description to get the deserialized size
179     cObjectPtr<IMediaSerializer> pSerializer;
180     pTypeDesc->GetMediaSampleSerializer(&pSerializer);
181     tInt nSize = pSerializer->GetDeserializedSize();
182     pSerializer = NULL;
183
184     // init the values of the protocol
185     tInt8 i8SOF = ID_ARD_SOF;          // Set the start of frame
186     tUInt32 ui32ArduinoTimestamp = 0;  // Set the Timestamp
187
188     // create and allocate the sample
189     cObjectPtr<IMediaSample> pSample;
190     _runtime->CreateInstance(OID_ADTF_MEDIA_SAMPLE, IID_ADTF_MEDIA_SAMPLE, (tVoid**)&pSample);
191     pSample->AllocBuffer(nSize);

```

Abbildung 21 Erstellen eines MediaSample

Im zweiten Teil der Hilfsfunktion werden die tatsächlichen Werte der *MediaSamples* gesetzt. Hierzu muss zunächst ein *MediaCoder* erstellt werden. Dieser ist notwendig, um die gewünschten Daten an den entsprechenden Bits des *MediaSamples* anzulegen. Das Setzen der Werte wird im Folgenden kurz mit den entsprechenden Zeilennummern vorgenommen:

- 198: Start of Frame
- 199: ID des Frames
- 200: Zeitstempel der Daten
- 201: Datenlänge der Nutzdaten
- 202: tatsächlichen Nutzdaten

Das eigentliche Senden des *MediaSamples* wird über den Aufruf der Methode in Zeile 210 realisiert. Hierbei wird das soeben erstellte *MediaSample* übergeben.

```

193 // get the coder
194 cObjectPtr<IMediaCoder> pCoder;
195 pTypeDesc->WriteLock(pSample, &pCoder);
196
197 // use the coder to set the value
198 pCoder->Set("i8SOF", (tVoid*)&i8SOF);
199 pCoder->Set("chID", (tVoid*)&ch8Id);
200 pCoder->Set("ui32ArduinoTimestamp", (tVoid*)&(ui32ArduinoTimestamp));
201 pCoder->Set("i8DataLength", (tVoid*)&i8DataLength);
202 pCoder->Set("chData", (tVoid*)&chData);
203
204
205 // unlock the coder
206 pTypeDesc->Unlock(pCoder);
207 // set the sample time (in this case the time doesn't matter and must not be the stream time)
208 pSample->SetTime(0);
209 // transmit the media sample
210 __adtf_test_result(pSampleSource->Transmit(pSample));
211
212 }

```

Abbildung 22 Werte in MediaSample festlegen

Die Hilfsfunktion *transmitDriveCommand* (Abbildung 23) behandelt die erhaltenen Werte, wandelt Nutzdaten in den korrekten Datentyp und übergibt die gesamten Werte der eben erläuterten Funktion *transmitMediaSample*. Allerdings gilt es zu beachten, dass diese Funktion sich lediglich auf Geschwindigkeits- und Lenkwinkelwerte bezieht.

Nachfolgend wird der Aufbau dieser Funktion näher erläutert. Die Parameter *SampleSource* sowie *chFrameID* werden der Funktion lediglich übergeben, um diese im unteren Funktionsablauf in Zeile 223 der Funktion *transmitMediaSample* zu überreichen. Den Parameter *f32Value* nutzt die Funktion *transmitDriveCommand* aktiv und castet den Wert vom Typ *Float* in ein Datenpaket mit dem Datentyp *Char*. Zur exakten Rundung wird der Float-Wert einer Addition von 0,5f unterzogen. Somit wird bei Werten mit der Kommastelle fünf auf den nächsthöheren Wert aufgerundet. Der Char-Wert wird danach ebenfalls der Funktion *transmitMediaSample* übergeben. Ebenso wird der Funktion die Datenlänge, die in diesem Fall ein Byte beträgt, überreicht.

```

218 tTestResult transmitDriveCommand(cOutputPin *pSampleSource, tFloat32 f32Value, const char chFrameId)
219 {
220     // correct rounding of the value // Kommentar
221     tChar chData = static_cast<tChar> (f32Value + 0.5f);
222     // sizeof
223     return transmitMediaSample(pSampleSource, chFrameId, 1, &chData);
224 }

```

Abbildung 23 TransmitDriveCommand

Analog zu *transmitDriveCommand* behandelt auch die Hilfsfunktion *transmitLightSignal* die erhaltenen Werte und übergibt diese ebenso der Funktion *transmitMediaSample*. Die Funktion ist im Gegensatz zur Vorhergehenden nur für die Licht-Daten zuständig. In Abbildung 24 ist zu erkennen, dass die Nutzdaten allerdings nicht wie in Abbildung 23 gecastet werden müssen, sondern schon im korrekten Datentyp vorliegen. Auch die Datenlänge erhöht sich im Funktionsaufruf *transmitMediaSample*, was sich folgerichtig zur Erklärung und Definition in Abschnitt 2.3.3 gestaltet.

```
220 tTestResult transmitLightSignal(cOutputPin *pSampleSource, tChar *chData, const char chFrameId)
221 {
222     return transmitMediaSample(pSampleSource, chFrameId, 2, chData);
223 }
```

Abbildung 24 TransmitLightSignal

Der Watchdog repräsentiert ein Signal, dass zur Überwachung genutzt wird und ohne dieses keine Datenpakete gesendet werden.

Die Funktion *transmitWatchdog* bekommt im Vergleich zu den zuvor genannten Hilfsfunktionen lediglich den Parameter *SampleSource*, da keine verschiedenen Frame-ID oder Daten zu erwarten sind.

In der Programmzeile 231 werden die tatsächlichen Daten behandelt. In dieser wird nur ein Bool-Wert gecastet, da der Watchdog, wie bereits erwähnt, nur ein Überwachungssignal darstellt und sendet.

Die Übergabeparameter ergeben sich somit aus der überreichten *SampleSource*, der stets gleichen Frame-ID, der festen Datenlänge von einem Byte und den tatsächlichen Daten aus Programmzeile 231.

```
229 tTestResult transmitWatchdog(cOutputPin *pSampleSource)
230 {
231     tChar chData = static_cast<tChar> (true);
232     return transmitMediaSample(pSampleSource, ID_ARD_ACT_WD_TOGGLE, 1, &chData);
233 }
```

Abbildung 25 TransmitWatchdog

Das Testing beinhaltet die Ansteuerungsseite des Testablaufs, was sich wiederum in vier verschiedene For-Schleifen unterteilt, die jeweils einen zu testenden Bereich der Rampe definieren. Die vier Rampen beschreiben den Testablauf und sollen anhand des Lenkwinkel-Beispiels die notwendige Anzahl von vier Rampen verdeutlichen.

1. Lenkeinschlag von Grundstellung nach rechts
2. Lenkeinschlag von rechts nach Grundstellung
3. Lenkeinschlag von Grundstellung nach links
4. Lenkeinschlag links nach Grundstellung

Die For-Schleifen sind dabei nochmals in Fahr- und Lichtansteuerung gegliedert.

Im Folgenden wird lediglich auf das erste Viertel der Rampe (erste For-Schleife) eingegangen, da die restlichen drei Rampensegmente kohärent zu dem ersten Teil sind.

Abbildung 26 zeigt zunächst den Beginn des Testings. Hierbei wird in Zeile 354 eine For-Schleife genutzt, um genügend Zeit für das Senden des Watchdog-Signals (Zeile 357) zu gewährleisten. Mit dem Senden des Watchdog-Signals wird das Motorrelais geschlossen und der Motorcontroller kann sich initialisieren. Eine Initialisierung muss vor dem ersten Senden von Daten stattfinden, da sonst die Initialisierung durch die gesendeten Daten unterbrochen wird.

Grundsätzlich wird nach jedem Senden über die Funktion *transmitMediaSample* ein Sleep gesendet, um die Taktrate immer zu gewährleisten. Zur Fokussierung auf die wesentlichen Inhalte wird daher nicht mehr in den kommenden Programmausschnitten auf das Senden der Sleep-Zeit eingegangen.

```
350 //#####
351 // send some samples
352 //#####
353 // to give the motor controller time for initialisation send for a short time only wd signal
354 for (tUInt32 nIdx = 0; nIdx < ui32Waittime; nIdx++)
355 {
356     // Allways send Watchdog Signal
357     transmitWatchdog(pSampleSource);
358     cSystem::Sleep(nSleep);
359 }
```

Abbildung 26 Testing – Initialisierung

Jede der vier Rampen benötigt einen oberen und unteren Wert zur Definition der jeweiligen Rampengrenze. Diese Festlegung wird in Abbildung 27 getroffen, wobei zu Beginn eine Abfrage getroffen wird, ob die jeweilige ID eines Lenkwinkels (Zeile 362), einer Geschwindigkeit (Zeile 367) oder eines allgemeinen Gesamttests überreicht wird. Die ID *0xff* wird dabei stets für den Gesamttest verwendet. Der Gesamttest, der sowohl ein Test der Geschwindigkeit als auch des Lenkwinkels darstellt, benötigt den Wertebereich, in dem einerseits Lenkwinkel- und andererseits auch Geschwindigkeitsdaten gesendet werden können. Würde der Wertebereich der

Geschwindigkeit gewählt werden, würde die physikalische Grenze des Lenkwinkelstellmotors überschritten werden, da der Arduino Communication Filter keinen Mechanismus beinhaltet, der eine Überschreitung des Wertebereichs verhindert.

```
361 // define a Ramp from one border to init Value
362 if(chFrameId == ID_ARD_ACT_STEER_ANGLE || chFrameId == 0xff)
363 {
364     f32LowerRange = 95.0f;
365     f32UpperRange = 125.0f;
366
367 }else if (chFrameId == ID_ARD_ACT_ACCEL_SERVO){
368     f32LowerRange = 90.0f;
369     f32UpperRange = 180.0f;
370 }
```

Abbildung 27 Testing – Rampendefinition

Das tatsächliche Senden der ersten Rampe beginnt mit der For-Schleife in Programmzeile 373 (Abbildung 28). Diese Schleife wird dazu genutzt, um die gewünschte Anzahl an *MediaSamples* senden zu können. Dadurch, dass die Anzahl festgelegt wird und bekannt ist, können statistische Aussagen getroffen werden, indem man in der Auswertung die erhaltenen Daten erneut zählt. Da vier For-Schleifen existieren, wird der definierte *ui32MaxLoopCount* durch vier geteilt, sodass jede Rampe zeitlich identisch ist.

Ebenfalls wird in jedem Schleifendurchlauf ein Watchdog-Signal (Zeile 376) gesendet, das wie bereits im oberen Abschnitt erwähnt, für das Schließen des Motorrelais verantwortlich ist. Liegt dieses Signal nicht an, so wird das Motorrelais geöffnet und der Motor kann nicht mehr angesteuert werden.

```
372 // send the samples according to the specified ramp
373 for (tUInt32 nIdx = 0; nIdx < ui32MaxLoopCount/4; nIdx++)
374 {
375     // Always send Watchdog Signal
376     transmitWatchdog(pSampleSource);
377     cSystem::Sleep(nSleep);
}
```

Abbildung 28 Testing – Watchdog

Der Fahrttest in Abbildung 29 tritt ein, wenn entweder die Frame-ID der Geschwindigkeit, des Lenkwinkels oder die allgemeine ID 0xff überreicht wird. In Programmzeile 385 wird die Berechnung des aktuellen Wertes vorgenommen, der tatsächlich gesendet wird. Mittels dieser Berechnung wird der zu sendende Wertebereich in äquidistante Werte eingeteilt. Für den Gesamttest müssen sowohl

Geschwindigkeits- sowie Lenkwinkelwerte gesendet werden, was im Programmablauf in Zeile 387 bis 393 realisiert wird. Ansonsten wird wie ab Zeile 394 zu sehen, die entsprechende ID übergeben, mit der wiederum entschieden wird, ob Lenkwinkel- oder Geschwindigkeitswerte zu senden sind.

```
379 // Acceleration and steer angle test
380 if(chFrameId==ID_ARD_ACT_ACCEL_SERVO || chFrameId ==ID_ARD_ACT_STEER_ANGLE || chFrameId == 0xff)
381 {
382     // set the data of the protocol
383     // Calculate the value by dividing the range by the loop count times the loop counter
384     // through this every Value will be tested
385     tFloat32 f32Value = (f32UpperRange - f32LowerRange) / (ui32MaxLoopCount/4) * nIndex + f32LowerRange;
386
387     if(chFrameId == 0xff)
388     {
389         transmitDriveCommand(pSampleSource, f32Value, ID_ARD_ACT_ACCEL_SERVO);
390         cSystem::Sleep(nSleep);
391         transmitDriveCommand(pSampleSource, f32Value, ID_ARD_ACT_STEER_ANGLE);
392         cSystem::Sleep(nSleep);
393     }
394     else
395     {
396         transmitDriveCommand(pSampleSource, f32Value, chFrameId);
397         cSystem::Sleep(nSleep);
398     }
399 }
```

Abbildung 29 Testing - Acceleration and SteerAngle

Abbildung 30 zeigt den Lichttest, der eintritt, wenn eine ID der Licht-Frames oder die des Gesamttests übergeben wird. Für das Festlegen der unterschiedlichen Zeitabschnitte ist eine Kalkulation für die verschiedenen Lichter notwendig, die allerdings nicht auf der Abbildung zu sehen ist, sondern schon im vorgegangenen Programmablauf errechnet wurde. Diese Kalkulation ist notwendig, um die Lichter kontrolliert nacheinander anschalten zu können.

In der Zeile 401 findet die Festlegung der Lichtdaten statt, die nur die booleschen Werte True und False annehmen können. Diese Werte kennzeichnen, ob das entsprechende Licht an- oder ausgeschaltet ist. Da die Daten für jedes Licht zwei Byte groß sind, wird ein Array benötigt, das in Programmzeile 402 erzeugt wird. Die zwei Byte sind notwendig, dass bei den verschiedenen Lichtern je die Entscheidung getroffen werden kann, welches Licht ausgewählt wird und ob dieses aktiviert oder deaktiviert wird.

Die Abfrage, in welchem Zeitabschnitt der Test sich befindet, findet von Zeile 408 bis 413 statt. Hier wird auch die Reihenfolge des Licht-Ablaufs festgelegt, welche in Abbildung 30 nachvollziehbar ist. Zeile 416 der Funktion wird genutzt, um den Wert zum Einschalten des Lichts zu setzen, vorausgesetzt, dass die vordere Abfrage den booleschen Wert True ergibt.

Die Funktion *transmitMediaSample* bekommt, nach dem bekannten Prinzip, die notwendigen Parameter über die Hilfsfunktion *transmitLightSignal* in Zeile 418 übergeben, um die entsprechenden *MediaSamples* zu senden.

```

400 // Light Test
401 if(chFrameId == ID_ARD_ACT_LIGHT_DATA || chFrameId == 0xff)
402 {
403     // set the data of the protocol
404     tBool bValue = tTrue;
405     tChar achData[2] = {0,0};
406
407     // Divide the loop into equal sections for every Light
408     if(nIdx > (f32StartOfHeadlight) && nIdx < (f32StartOfBacklight) ) achData[0] = ID_ARD_ACT_LIGHT_DATA_HEAD;
409     if(nIdx > (f32StartOfBacklight) && nIdx < (f32StartOfBrakelight) ) achData[0] = ID_ARD_ACT_LIGHT_DATA_BACK;
410     if(nIdx > (f32StartOfBrakelight) && nIdx < (f32StartOfTurnleftlight) ) achData[0] = ID_ARD_ACT_LIGHT_DATA_BRAKE;
411     if(nIdx > (f32StartOfTurnleftlight) && nIdx < (f32StartOfTurnrightlight) ) achData[0] = ID_ARD_ACT_LIGHT_DATA_TURNLEFT;
412     if(nIdx > (f32StartOfTurnrightlight) && nIdx < (f32StartOfReverselight) ) achData[0] = ID_ARD_ACT_LIGHT_DATA_TURNRIGHT;
413     if(nIdx > (f32StartOfReverselight) && nIdx < ui32MaxLoopCount) achData[0] = ID_ARD_ACT_LIGHT_DATA_REVERSE;
414
415     // Set the Lights to on
416     bValue == tTrue ? achData[1] = 1: achData[1] = 0;
417
418     transmitLightSignal(pSampleSource, achData, ID_ARD_ACT_LIGHT_DATA);
419     cSystem::Sleep(nSleep);
420 }
421 }

```

Abbildung 30 Testing - LightTest

Die in Abbildung 31 gezeigte *SampleSink* wird verwendet, um die *MediaSamples* vom getesteten Filter, für die spätere Evaluation, anhand einer Liste zwischen zu speichern. Diese *SampleSink* ist somit notwendig und wird, wie in Zeile 95 zu sehen, als eigene Klasse erstellt.

Folglich braucht diese Klasse auch einen Konstruktor, der ein Objekt der Klasse erzeugt und einen Destruktor, der die Speicherbereinigung durchführt. Auf den genauen Ablauf dieser Speicherbereinigung wird an geeigneter Stelle der einzigen Methode der Klasse nochmals Bezug genommen.

Die Variablen der Klasse werden in Programmzeile 112 und 113 erstellt. In Zeile 112 handelt es sich dabei um einen Zähler, zum Erfassen der ankommenden *MediaSamples* und in Zeile 113 um die Liste, an die die *MediaSamples* aneinander gehängt werden.

Das praktische Aneinanderreihen der *MediaSamples* wird von der Methode *OnPinEvent* übernommen, die dazu verschiedene Parameter von der Softwareumgebung ADTF übergeben bekommt. Liegen dabei tatsächlich neue Daten am Pin an und stimmt somit wirklich der *EventCode* überein, dass *MediaSamples* übermittelt wurden, so springt der Programmablauf in die If-Abfrage. Hierbei wird in Zeile 120 zunächst der Samplezähler inkrementiert und das *MediaSample* wird an letzter Stelle der Liste angehängt. Ebenso wird das neue *MediaSample* referenziert, sodass eine ungewollte Speicherbereinigung seitens ADTF vermieden wird. Hierzu

muss der bereits erwähnte Bezug zum Destruktor hergestellt werden, da dieser eine beabsichtigte Speicherbereinigung durchführt, die allerdings durch die Referenz verhindert werden würde. Um dieses Problem zu beheben, werden in Zeile 105 bis 109, alle *MediaSamples*, im Destruktor dereferenziert. Das hat gleichzeitig ein Bereinigen der jeweiligen Daten zu Folge, da das Programm gerade am Abbauen ist. In Zeile 126 wird darüber hinaus noch ein Rückgabewert generiert, der das erfolgreiche Aufbauen der Liste bestätigt.

```

95 class cMediaSampleSink: public IPinEventSink
96 {
97     // ucom helper macro
98     UCOM_OBJECT_IMPL(IID_ADTF_PIN_EVENT_SINK, adtf::IPinEventSink);
99 public:
100     // constructor
101     cMediaSampleSink(): m_ui32SampleCount(0) { }
102     // destructor
103     virtual ~cMediaSampleSink()
104     {
105         // loop over all mediasamples to unref
106         for (tUInt nIdx = 0; nIdx < m_vecMediaSamples.size(); nIdx++)
107         {
108             // after unref the mediasample will destroy itself
109             m_vecMediaSamples[nIdx]->Unref();
110         }
111         // members to count and hold the received media samples
112         tUInt32 m_ui32SampleCount;
113         std::vector<IMediaSample*> m_vecMediaSamples;
114 public:
115         tResult OnPinEvent(IPin* pSource, tInt nEventCode, tInt nParam1, tInt nParam2, IMediaSample *pMediaSample)
116         {
117             if (nEventCode == IPinEventSink::PE_MediaSampleTransmitted)
118             {
119                 // count the samples
120                 ++m_ui32SampleCount;
121                 // make a ref on sample to avoid self destruction
122                 pMediaSample->Ref();
123                 // push sample into vector for later compare
124                 m_vecMediaSamples.push_back(pMediaSample);
125             }
126             RETURN_NOERROR;
127         }
128     };

```

Abbildung 31 *SampleSink*

Die *SampleSink* bildet wie eben beschrieben die Senke, die die Daten entgegennimmt. Sie stellt ein eigenes kleines Modul dar, weshalb sie am Ausgang des zu testenden Filters angeschlossen werden muss, was in Abbildung 32 gezeigt wird. In Programmzeile 276 wird dabei zuerst ein Objekt der Klasse *SampleSink* erstellt. Das eigentliche Verknüpfen des Output Pins mit der eben erstellten *SampleSink* findet dagegen in Zeile 277 statt.

```

275 // register the sample sink to receive the data from the output pin of the Arduino Communication Filter
276 cObjectPtr<cMediaSampleSink> pSampleSink = new cMediaSampleSink();
277 __adtf_test_result(pComOutputPin->RegisterEventSink(pSampleSink));

```

Abbildung 32 Verknüpfung *SampleSink*

Nachdem alle notwendigen Schritte zum Erstellen der Quelle und Senke, sowie der zu beaufschlagenden Daten am Eingang durchgeführt wurden, können nun die Daten am Ausgang ausgewertet werden. Dieser Vorgang läuft in der Evaluation ab, die im Folgenden beschrieben wird.

Zu Beginn wird eine Map erstellt, was in Abbildung 33 dargestellt wird. Diese Map wird im weiteren Verlauf der Evaluation verwendet, um die empfangenen *MediaSamples* nach der Frame-ID zu sortieren.

```
609 // map with channel/frameid as key and vector of mediasamples...
610 // ...as value to sort the different kinds of sensors
611 //cSampleMap mapSensors;
612 std::map<uchar, std::vector<ArduinoFrame> > mapSensors2;
```

Abbildung 33 Map zur Sortierung und Abspeicherung der *MediaSamples*

Bevor die empfangenen Daten geordnet werden, ist es zuerst notwendig die empfangenen *MediaSamples* auszuwerten. Abbildung 34 beginnt hierzu mit einer For-Schleife, die von dem Zählerwert null bis zur entsprechenden Anzahl der gespeicherten Mediasamples ein Datenpaket nach dem anderen pro Schleifendurchgang selektiert. Der *MediaCoder* in Zeile 617 wird genutzt, um die gewünschten Daten an den entsprechenden Bitpositionen zu bestimmen, wie es bereits detailliert in Abbildung 22 gezeigt wurde. Mittels dem Vektor *m_vecMediaSamples* in Zeile 618 wird das gewünschte *MediaSamples* mittels jeweiliger ID *nIdx* ausgewählt. Die Daten, die aus dem Frame gelesen werden sollen, sind in Zeile 621 bis 625 erstmals bestimmt und werden zunächst initialisiert. Die Initialisierung ist dabei erforderlich, dass die Daten, die nicht empfangen wurden, einen spezifizierten Wert besitzen und nicht undefiniert bleiben. Ein Programmabbruch wäre die Folge. Das eigentliche Auslesen der Daten findet in Zeile 628 bis 632 statt. Des Weiteren muss der *MediaCoder* noch vor dem nächsten Schleifendurchlauf entriegelt werden, um danach auf das nächste *MediaSample* zugreifen zu können.

```
613 // build the map by iterating over all received mediasamples
614 for(tUInt32 nIdx = 0; nIdx < pSampleSink->m_vecMediaSamples.size(); ++nIdx)
615 {
616     // get the coder
617     cObjectPtr<IMediaCoder> pCoder;
618     m_pCoderDescArduinoData->Lock(pSampleSink->m_vecMediaSamples[nIdx], &pCoder);
619
620     // init the values
621     tInt8 i8SOF = 0;
622     tChar ch8Id = 0;
623     tUInt32 arduinoTimestamp = 0;
624     tInt8 i8DataLength = 0;
625     tChar frameData[25];
626
627     // get values from coder
628     pCoder->Get("chID", (tVoid*)&ch8Id);
629     pCoder->Get("i8SOF", (tVoid*)&i8SOF);
630     pCoder->Get("ui32ArduinoTimestamp", (tVoid*)&arduinoTimestamp);
631     pCoder->Get("i8DataLength", (tVoid*)&i8DataLength);
632     pCoder->Get("chData", (tVoid*)&(frameData));
633
634     m_pCoderDescArduinoData->Unlock(pCoder);
```

Abbildung 34 Auslesen der empfangenen *MediaSamples*

Abbildung 35 zeigt das Abspeichern der Inhalte der *MediaSamples*, wozu die Map notwendig ist, die in Abbildung 33 gezeigt und darüber erläutert wurde. In Programmzeile 637 findet eine Plausibilitätsprüfung der *MediaSamples* statt. Dabei findet eine Kontrolle statt, ob die Datenlänge des ausgelesenen *MediaSamples* den Vorgaben hinsichtlich Datenlänge entspricht. Nach diesem Vorgang wird ein Arduino Frame erstellt (Zeile 640), um die relevanten Daten aufnehmen und abspeichern zu können. Die Speichergröße des Frames des *MediaSamples* wird dazu in Zeile 641 auf die Größe des Arduino Frames gesetzt. Grund dafür ist, dass ein reiner Arduino Frame nur Daten enthält die benötigt werden und ein *MediaSample* noch weitere Inhalte besitzt, die für diese Tests keine weitere Bedeutung haben. Durch dieses Vorgehen wird kein Speicherplatz unnötig vergeudet.

Das Füllen des Arduino Frames mit den wichtigen Daten findet in Zeile 644 bis 648 statt. Anschließend wird dieser Frame in Zeile 650 in der Map nach der jeweiligen ID sortiert und abgespeichert.

```
636 // check for plausible datalength
637 i8DataLength = i8DataLength > sizeof(tArduinoDataUnion) ? sizeof(tArduinoDataUnion) : i8DataLength;
638
639 // create a arduino frame
640 tArduinoFrame sFrame;
641 cMemoryBlock::MemSet(&sFrame, 0, sizeof(tArduinoFrame));
642
643 // fill the arduino frame
644 sFrame.sHeader.chID = ch8Id;
645 sFrame.sHeader.i8SOF = i8SOF;
646 sFrame.sHeader.ui32ArduinoTimestamp = arduinoTimestamp;
647 sFrame.sHeader.i8DataLength = i8DataLength;
648 cMemoryBlock::Memcpy(&sFrame.sData, &frameData, i8DataLength);
649
650 mapSensors2[ch8Id].push_back(sFrame);
651 }
```

Abbildung 35 Abspeichern der *MediaSamples*

Zur Speicherbereinigung wird die *SampleSink* anschließend gelöscht und der Filter wird heruntergefahren, um nicht noch weitere *MediaSamples* zu senden. Diese Abfolge wird in Abbildung 36 verdeutlicht.

```
653 // clear the sample sink
654 pSampleSink->m_vecMediaSamples.clear();
655 // Shutdown the Filter
656 SET_STATE_SHUTDOWN(pFilter);
```

Abbildung 36 Speicherbereinigung Arduino Communication Filter

Abbildung 37 nutzt die Sortierung der Map aus, um zu prüfen, ob für jeden Sensor genug *MediaSamples* empfangen wurden. Dieser Abgleich geschieht durch einen Vergleich der empfangenen *Samples* mit der Anzahl von *MediaSamples* aus empirischen Versuchen. Wenn nicht genug *Samples* empfangen wurden, bricht der Test ab und erzeugt eine entsprechende Fehlermeldung mit der Information, welcher Sensor zu wenig *MediaSamples* gesendet hat. Diese Überprüfung ist wichtig, um zu überprüfen, ob der Arduino Communication Filter noch genügend Daten versendet. Gerade bei zu hohen Datenraten ist die Versendung von Daten oft nicht mehr gewährleistet.

```
660 // Do a quick test to see if enough samples are received
661 __adtf_test_ext(mapSensors2[ID_ARD_SENS_STEER_ANGLE].size() >= 100,
662 cString::Format("does not receive enough samples for Steerangle Sensor. Number of samples: %d",
663 mapSensors2[ID_ARD_SENS_STEER_ANGLE].size()));
664 __adtf_test_ext(mapSensors2[ID_ARD_SENS_WHEELENC].size() >= 100,
665 cString::Format("does not receive enough samples for RPM Sensor. Number of samples: %d",
666 mapSensors2[ID_ARD_SENS_WHEELENC].size()));
667 __adtf_test_ext(mapSensors2[ID_ARD_SENS_IMU].size() >= 100,
668 cString::Format("does not receive enough samples for Motion Sensor. Number of samples: %d",
669 mapSensors2[ID_ARD_SENS_IMU].size()));
670 __adtf_test_ext(mapSensors2[ID_ARD_SENS_IR].size() >= 100,
671 cString::Format("does not receive enough samples for Infrared Sensor. Number of samples: %d",
672 mapSensors2[ID_ARD_SENS_IR].size()));
673 __adtf_test_ext(mapSensors2[ID_ARD_SENS_PHOTO].size() >= 100,
674 cString::Format("does not receive enough samples for Photo Sensor. Number of samples: %d",
675 mapSensors2[ID_ARD_SENS_PHOTO].size()));
676 __adtf_test_ext(mapSensors2[ID_ARD_SENS_US].size() >= 50,
677 cString::Format("does not receive enough samples for Ultra Sonic Sensor. Number of samples: %d",
678 mapSensors2[ID_ARD_SENS_US].size()));
679 __adtf_test_ext(mapSensors2[ID_ARD_SENS_VOLTAGE].size() >= 5,
680 cString::Format("does not receive enough samples for Voltage Sensor. Number of samples: %d",
681 mapSensors2[ID_ARD_SENS_VOLTAGE].size()));
```

Abbildung 37 Anzahlprüfung *MediaSamples*

Der Beginn der konkreten Auswertung des Geschwindigkeits- und des allgemeinen Tests findet in Abbildung 38 statt. Zur Erkennung der vier definierten Rampenabschnitte in der Auswertung werden die Grenzen der Rampe sowie deren Toleranz in Programmzeile 690 bis 692 festgelegt. Mittels dieser Werte kann in der späteren Rampenerkennung ein Start sowie Endpunkt detektiert werden.

Angrenzend werden noch weitere Werte initialisiert, die relevant zur Rampenerkennung sind:

- Zeile 695: Variable zum Zwischenspeichern des zu vorigen Raddrehzahl-Differenzwertes, um einen entsprechenden Vergleichswert zur aktuellen Differenz zu generieren. Wenn das Rad beispielsweise schneller dreht, so muss auch der RPM-Wert eine höhere Differenzanzahl an Daten in einem gleichen Zeitraum besitzen
- Zeile 696: Diese Variable bildet den absoluten zu vorigen Wert der Raddrehzahl, um aus diesem und dem Wert der aktuellen Drehzahl die Differenz zu bilden
- Zeile 697: Stellt einen booleschen Wert dar, der zur Erkennung genutzt wird, ob sich der Wertedurchlauf noch auf der Rampe befindet
- Zeile 698: Ist eine Variable zum Zwischenspeichern, auf welcher der aktuelle Durchlauf stattfindet

- Zeile 699: Bildet einen Wert, mit dem am Ende des Rampendurchlaufs ein Vergleich gezogen werden kann, ob die Ausgangsposition wieder erreicht wurde

```

683 // if the sleep time is long enough do a deeper check for the ramps
684 if(nSleep > 10 && (chFrameId == ID_ARD_SENS_WHEELENC || chFrameId == 0xff))
685 {
686     // received sensor data should show the actor ramp
687     // now check the received data
688
689     // Define the Ranges of the ramp and a tolerance Value to ignore single value jumps
690     tUInt16 ui16LowerRampRange = 1;
691     tUInt16 ui16UpperRampRange = 10;
692     tUInt16 ui16ToleranceValue = 4;
693
694     // Init some values we need to recognize the ramp
695     tUInt32 ui32RPMDiffBefore = 0;
696     tUInt32 ui32RPMBefore = 0;
697     tBool bStartOfRamp = tFalse;
698     tInt8 i8RampSegment = 0;
699     tUInt16 ui16StartRPMDiff = 0;

```

Abbildung 38 Raddrehzahlsensor - Festlegung wichtiger Grenzen und Werte

Abbildung 39 verdeutlicht zunächst, dass nur jeder dritte Wert der *MediaSamples* verwendet wird. Grund dafür ist, dass die Samples so schnell gesendet werden, dass keine große Differenz der Werte besteht.

Des Weiteren wird auch der Header ausgewertet. Dabei finden Abfragen von Zeile 710 bis 714 statt, ob der richtige SOF, die richtige ID und die richtige Datenlänge übertragen wurden.

```

705 // only take every third value to receive stronger differences
706 if(nIdx % 3)
707 {
708     // test the received data
709     // SOF
710     __adtf_test(mapSensors2[ID_ARD_SENS_WHEELENC][nIdx].sHeader.i8SOF == ID_ARD_SOF);
711     // Frame ID
712     __adtf_test(mapSensors2[ID_ARD_SENS_WHEELENC][nIdx].sHeader.chID == ID_ARD_SENS_WHEELENC);
713     // Check for the right datalength
714     __adtf_test(mapSensors2[ID_ARD_SENS_WHEELENC][nIdx].sHeader.i8DataLength == 8);

```

Abbildung 39 Auswertung Header

Um die Lesezugriffe auf die Map zu minimieren, werden die Werte der aktuellen Nutzdaten des Frames, des linken Rades, zwischengespeichert. Dieser Vorgang ist in Zeile 717 (Abbildung 40) sichtbar.

Darüber hinaus wird in der gleichen Abbildung die Bildung eines Mittelwertes (Zeile 719) aufgezeigt. Diese Mittelwertbildung hat eine Glättung der Werte zu Folge, um einzelne Höchstwerte zu egalisieren.

```
716 // Extract the RPM Data
717 tUInt32 ui32RPM = mapSensors2[ID_ARD_SENS_WHEELENC][nIdx].sData.sWheelEncData.ui32LeftWheel;
718 // weigh the new value statistically with the old one to filter out peak values
719 ui32RPM = (ui32RPMBefore + ui32RPM)/2;
```

Abbildung 40 Auslesen und Gewichten der Daten

Die Rampenerkennung findet in Abbildung 41 statt.

Zu Beginn wird in Zeile 726 geprüft, ob der Wertedurchlauf sich auf der Rampe befindet. Ist das nicht der Fall und es wurde auch noch kein Rampensegment durchlaufen, so erfolgt die Abfrage, ob die Raddrehzahlsensor-Differenz größer null ist. Ist diese größer als null, so hat das Fahrzeug mit dem Fahren begonnen. Dementsprechend wird die Wertedurchlauf-Erkennung auf True (Zeile 729) und die anfängliche Raddrehzahl-Differenz auf null gesetzt. Das Rampensegment wird ebenfalls um einen Wert nach oben gezählt und befindet sich auf dem ersten der vier Rampensegmente. Diese Inkrementierung wird auf jedem Rampensegment wiederholt, bis die Rampe alle vier Segmente durchlaufen hat.

Da die tatsächlichen Rampengrenzen durch Messungenauigkeiten schwanken, ist der obere Rampengrenzwert etwas unterhalb des tatsächlichen Wertes definiert. Dies hat gleichzeitig zur Folge, dass die Rampe sowohl über den oberen als auch unter den unteren Grenzwert herausläuft. Aus diesem Grund wird jeweils auf einem Rampenabschnitt die Wertedurchlauf-Erkennung auf False gesetzt, wie es in Programmzeile 735, 742 und auch 748 zu sehen ist. Die Erkennung der Rampe wird mittels dem Programmabschnitt von Zeile 751 bis 755 wieder auf den Wert True gesetzt, wenn erkannt wird, dass sich die Werte wieder in dem Bereich zwischen dem oberen und unteren Rampengrenzwert befinden.

Nach diesem Prinzip kann jeweils ein neues Rampensegment begonnen werden. Das Rampensegment zwei und drei wurde zusammengelegt, da beide Segmente eine zusammenhängende und gleichbleibend fallende Rampe bilden, die als solche erkannt werden kann.


```

724 // Test for the ramp
725 // recognize the start of the ramp
726 if(!bStartOfRamp && ui32RPM - ui32RPMBefore > 0 && i8RampSegment == 0)
727 {
728     ui16StartRPMDiff = 0;
729     bStartOfRamp = tTrue;
730     i8RampSegment++;
731 }
732 // recognize the first turn
733 if(bStartOfRamp && ui32RPM - ui32RPMBefore > ui16UpperRampRange && i8RampSegment == 1)
734 {
735     i8RampSegment++;
736     bStartOfRamp = tFalse;
737 }
738 // recognize the second turn
739 if(bStartOfRamp && ui32RPM - ui32RPMBefore < ui16LowerRampRange && i8RampSegment == 2)
740 {
741     i8RampSegment++;
742     bStartOfRamp = tFalse;
743 }
744 // recognize the end of the ramp
745 if(bStartOfRamp && ui32RPM - ui32RPMBefore == ui16StartRPMDiff && i8RampSegment == 3)
746 {
747     i8RampSegment++;
748     bStartOfRamp = tFalse;
749 }
750 // start the ramp recognition again after a turn if the values are in the range
751 if(bStartOfRamp && ui32RPM - ui32RPMBefore < ui16UpperRampRange &&
752    ui32RPM - ui32RPMBefore > ui16LowerRampRange && i8RampSegment < 4)
753 {
754     bStartOfRamp = tTrue;
755 }

```

Abbildung 41 Rampenerkennung Raddrehzahlsensor

Die Auswertung dieser Rampenwerte bezieht sich auf die Rampenerkennung, welche in Abbildung 42 dargestellt wird. Befindet sich die Rampe beispielsweise in dem generierten Segment eins oder drei aus Abbildung 41, so muss in beiden Fällen eine steigende Rampe erkannt werden. Dieser Vorgang läuft in Zeile 760 ab. Dabei wird dem aktuellen absoluten Wert der Vorherige abgezogen und eine entsprechende Toleranz aufaddiert. Ist dieses Ergebnis größer als die zu vorige Differenz, so steigen die Geschwindigkeiten und es wird folgerichtig eine Rampe erkannt.

```

757 // test the values in the rising ramps
758 if(bStartOfRamp && (i8RampSegment == 1 || i8RampSegment == 3))
759 {
760     __adtf_test(ui32RPM - ui32RPMBefore + ui16ToleranceValue > ui32RPMDiffBefore);
761 }
762 // test the values of the falling ramp
763 if(bStartOfRamp && i8RampSegment == 2)
764 {
765     __adtf_test(ui32RPM - ui32RPMBefore < ui32RPMDiffBefore + ui16ToleranceValue);
766 }

```

Abbildung 42 Auswertung der Rampenerkennung

Bezogen auf den nächsten Werte-Durchlauf, müssen die Werte der Variablen sozusagen umgespeichert werden. Die aktuelle Differenz entspricht der vorigen

Differenz (Zeile 769) und ebenso entspricht der aktuelle Drehzahlwert der Drehzahl zuvor (Zeile 770). Dieses Vorgehen lässt sich in Abbildung 43 nachvollziehen.

Des Weiteren wird auf der Einrückungsebene der Auswertung der Drehzahlsensoren, das in Abbildung 38 dargestellt ist, eine Abfrage implementiert, die kontrolliert, ob alle Rampensegmente durchlaufen wurden (Zeile 775).

```
768 | // Go to the next Values
769 | ui32RPMDiffBefore = ui32RPM - ui32RPMBefore;
770 | ui32RPMBefore = ui32RPM;
771 |
772 |     }
773 | }
774 | // Check if the ramp was recognized
775 | __adtf_test(i8RampSegment == 4);
776 | }
```

Abbildung 43 Umspeichern der Werte und Auswertung Testdurchlauf

Die Auswertung des Lenkwinkelsensors schließt sich der des Drehzahlsensors in Aufbau und Inhalt identisch an. Der einzige Unterschied lässt sich in Abbildung 44 erkennen und spiegelt sich in den Grenzwerten und Variablen wieder.

Im Vergleich zur Abbildung 38 unterscheidet sich nicht nur die Toleranz sondern auch der obere und untere Grenzwert erheblich von den Werten des Drehzahlsensors.

Auch die initialisierten Variablen, die für die Rampenerkennung relevant sind, weisen diese Differenz auf. Diese werden nach dem Muster aus der Beschreibung zur Abbildung 38 fortgesetzt, um die Parallelitäten festzustellen.

- Zeile 787: Diese Variable bildet, analog zum Drehzahlsensor, den vorigen absoluten Wert des Lenkwinkels
- Zeile 788: Auch bei dem Lenkwinkel wird ein boolescher Wert zur Erkennung genutzt, ob sich der Wertedurchlauf auf der Rampe befindet
- Zeile 789: Diese Variable wird genutzt, um zwischen zu speichern, bei welchem Rampensegment sich der Durchlauf aktuell befindet. Die Initialisierung mit dem Wert zehn ist begründet durch den allgemeinen Test, bei dem zunächst noch auf eine Initialisierung des Lenkwinkels gewartet werden muss. Ist diese Initialisierung beendet, so wird die zehn mit einer null überschrieben, und der Vorgang läuft nach dem gleichen Prinzip des Drehzahlsensors ab

- Zeile 790: Der Lenkwinkelsensor benutzt wie der Drehzahlsensor einen Wert, mit dem das Ende des Rampendurchlaufs in der Ausgangsposition erkannt werden kann

Ein Differenzwert muss in dem Falle des Lenkwinkelsensors nicht berücksichtigt werden, da dieser Sensor jeweils einen absoluten Wert zurückgibt, der direkt verwendet werden kann.

```
782 // Define the Ranges of the ramp and a tolerance Value to ignore single value jumps
783 tUInt16 ui16LowerRampRange = 338;
784 tUInt16 ui16UpperRampRange = 520;
785 tUInt16 ui16ToleranceValue = 20;
786
787 // Init some values we need to recognize the ramp
788 tUInt16 ui16SteerAngleBefore = 0;
789 tBool bStartOfRamp = tFalse;
790 tInt8 i8RampSegment = 10;
791 tUInt16 ui16StartAngle = 0;
```

Abbildung 44 Lenkwinkelsensor - Festlegung wichtiger Grenzen und Werte

Nach dem vollständigen Umsetzen der theoretischen Testanforderung in Software, konnte mit der Grenzfallbetrachtung begonnen werden.

Zunächst wurde die grenzwertige Datenrate experimentell ermittelt. Dabei konnte durch die Variation der Sleep-Zeit immer eine unterschiedliche Taktrate simuliert werden, anhand derer sich ein Datenraten-Grenzfall von zehn Mikrosekunden bezüglich der Sleep-Zeit herauskristallisierte.

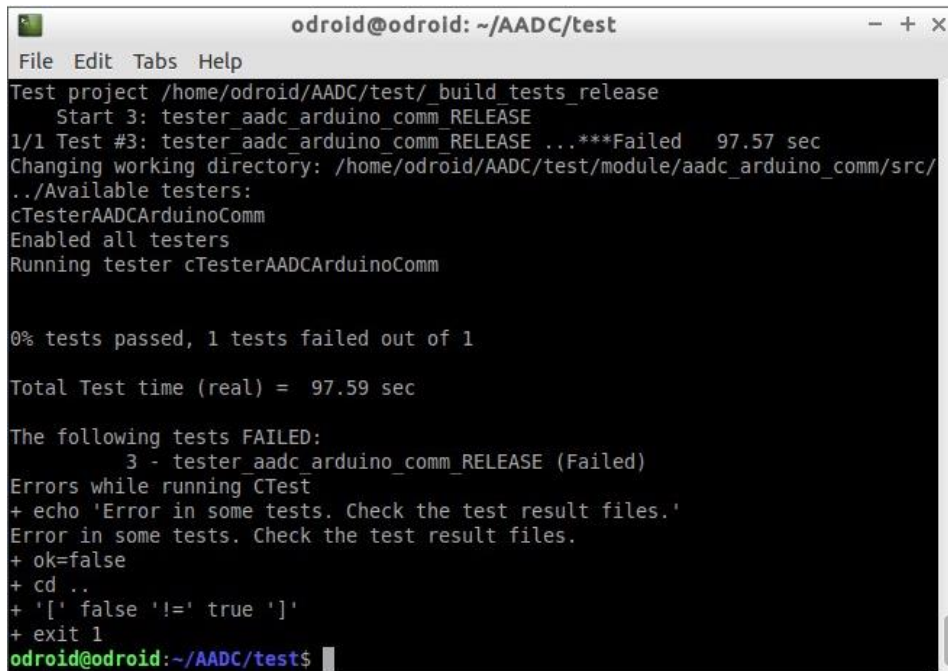
Da im ersten Programmentwurf eine willkürliche Sleep-Zeit unterhalb von zehn Mikrosekunden gewählt wurde, traten dabei vermehrt Probleme auf, die durch das anschließende Erhöhen dieser Zeit behoben werden konnte.

Die grenzwertige Datenrate wurde dadurch ermittelt, dass nicht mehr genügend *MediaSamples* gesendet wurden. Dies ist begründet durch das Arduino-Board, welches bei einer zu hohen Datenrate überlastet wird und nicht mehr korrekt Samples zu den jeweiligen Sensordaten liefern kann.

Wie in Abschnitt 3.2.1 beschrieben, sieht der darauffolgende Test eine Untergliederung in einen Testbereich unterhalb der Sleep-Zeit und einen oberhalb dieser vor. Beide Prüfungen wurden mit dem, in diesem Abschnitt beschriebenen, Arduino Communication Filter Test durchgeführt.

Der Testfall oberhalb der grenzwertigen Datenrate wies dabei nochmals unterschiedliche Ergebnisse auf. Knapp oberhalb der Datenrate konnten teilweise noch *MediaSamples* gesendet werden, was sich dadurch bemerkbar machte, dass

das Arduino-Board noch weiterhin arbeitete, aber das Testergebnis der Testsoftware (Abbildung 45) ein „FAILED“ anzeigt. Grund dafür ist, dass im Testablauf eine gewisse Anzahl von *MediaSamples* benötigt und abgefragt wird, wie es bereits in der Erläuterung zu Abbildung 37 aufgezeigt wurde. Bei etwa neun Mikrosekunden arbeitete auch das Arduino-Board nicht mehr, sodass ebenfalls der Test ein zu erwartendes negatives Ergebnis hervorbrachte.



```
odroid@odroid: ~/AADC/test
File Edit Tabs Help
Test project /home/odroid/AADC/test/_build_tests_release
Start 3: tester_aadc_arduino_comm_RELEASE
1/1 Test #3: tester_aadc_arduino_comm_RELEASE ...***Failed 97.57 sec
Changing working directory: /home/odroid/AADC/test/module/aadc_arduino_comm/src/
../Available testers:
cTesterAADCArduinoComm
Enabled all testers
Running tester cTesterAADCArduinoComm

0% tests passed, 1 tests failed out of 1

Total Test time (real) = 97.59 sec

The following tests FAILED:
 3 - tester_aadc_arduino_comm_RELEASE (Failed)
Errors while running CTest
+ echo 'Error in some tests. Check the test result files.'
Error in some tests. Check the test result files.
+ ok=false
+ cd ..
+ '[' false '!=' true ']'
+ exit 1
odroid@odroid:~/AADC/test$
```

Abbildung 45 Failed Test

Ebenfalls wird in dieser Abbildung ein Verweis auf die Testergebnisse gegeben, die nachfolgend überprüft werden müssen. Die Ergebnisse des Tests können in Abbildung 46 nachvollzogen werden.

In Zeile 13 ist dabei zu erkennen, dass ein Fehler im Testfall des Gesamttests aufgetreten ist. Weiterhin ist bei genauerer Betrachtung zu erkennen, was genau fehlgeschlagen ist. Dabei handelt es sich um die Überprüfung, ob alle Rampensegmente durchlaufen wurden, was in Abbildung 43 bereits behandelt wurde.

Darüber hinaus lässt sich ein vollständiger Auszug aus dem Log-File einsehen (Zeile 19), um die genaue Fehlerursache zu ermitteln.

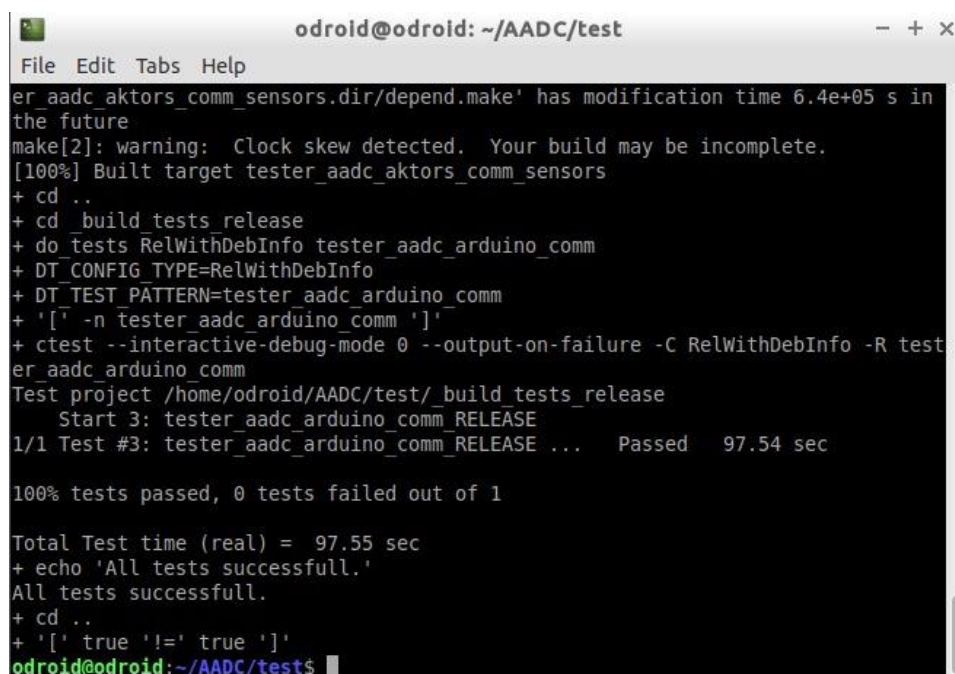
```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <testsuites>
3  <testsuite name="AADC Arduino Communication filter_RELEASE">
4  <properties>
5  <property name="id" value="2" />
6  <property name="short_description" value="This test makes sure that the AADC Arduino Communication filter works as expected" />
7  <property name="requirements" value="" />
8  <property name="build_nr" value="2751" />
9  </properties>
10 <testcase classname="AADC Arduino Communication filter_RELEASE" name="TestAcceleration" />
11 <testcase classname="AADC Arduino Communication filter_RELEASE" name="TestSteeringAngle" />
12 <testcase classname="AADC Arduino Communication filter_RELEASE" name="TestLights" />
13 <testcase classname="AADC Arduino Communication filter_RELEASE" name="TestAllActors">
14 <error message="(i8RampSegment == 4) failed" type="Error">date: 2014-12-03 03:01:20
15 expression: (i8RampSegment == 4)
16 position: /home/odroid/AADC/test/module/aadc_arduino_comm/src/tester_aadc_arduino_comm.cpp:709
17
18 </error>
19 <system-out>...</system-out>
2136 </testcase>
2137 </testsuite>
2138 </testsuites>

```

Abbildung 46 Test Result

Abbildung 47 zeigt das Ergebnis unterhalb der grenzwertigen Datenrate des Testfalls. Wie nach der Ermittlung des Datenraten-Grenzwertes und des Abprüfens oberhalb dieser Datenrate zu erwarten, stellte sich ein positives Ergebnis ein. Dieses Resultat bedeutet, dass nicht nur die *MediaSamples* in der gewünschten Anzahl gesendet wurden, sondern auch eine Rampe mittels Testablauf zurückgelesen werden konnte, was wiederum dem Beaufschlagen des Eingangs entspricht. Ebenfalls konnte die Ansteuerung der Lichter durch eine manuelle Sichtprüfung positiv bestätigt werden.



```

odroid@odroid: ~/AADC/test
File Edit Tabs Help
er_aadc_aktors_comm_sensors.dir/depend.make' has modification time 6.4e+05 s in
the future
make[2]: warning: Clock skew detected. Your build may be incomplete.
[100%] Built target tester_aadc_aktors_comm_sensors
+ cd ..
+ cd _build_tests_release
+ do_tests RelWithDebInfo tester_aadc_arduino_comm
+ DT_CONFIG_TYPE=RelWithDebInfo
+ DT_TEST_PATTERN=tester_aadc_arduino_comm
+ '[' -n tester_aadc_arduino_comm ']'
+ ctest --interactive-debug-mode 0 --output-on-failure -C RelWithDebInfo -R test
er_aadc_arduino_comm
Test project /home/odroid/AADC/test/ _build_tests_release
Start 3: tester_aadc_arduino_comm RELEASE
1/1 Test #3: tester_aadc_arduino_comm RELEASE ... Passed 97.54 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) = 97.55 sec
+ echo 'All tests successfull.'
All tests successfull.
+ cd ..
+ '[' true != true ']'
odroid@odroid: ~/AADC/test$

```

Abbildung 47 Passed Test

Durch das Ermitteln der grenzwertigen Datenrate für den Arduino Communication Filter und die daraus folgenden positiven und erwarteten Testergebnisse, kann dieser halbautomatische Testfall für weitere Softwareanpassungen am Arduino Communication Filter verwendet werden, um diese zu evaluieren.

4.1.2 Arduino Actuator

Der Aktuator Test prüft den Arduino Actuator Filter auf seine korrekte Arbeitsweise. Das bedeutet, es müssen vier verschiedene Tests entwickelt werden, die alle Testfälle des Filters abdecken. Die Testfälle behandeln die Datenpakete (*samples*), des Lenkwinkels und der Gaspedalstellung sowie der Ansteuerung des Lichtes und des Watchdogs.

Die Funktion *DoActuatorTest()* übernimmt den Test der Lenkwinkel- und Beschleunigungs-Samples, denn diese beiden Datenpaket-Varianten enthalten den identischen Datentyp. Als Übergabewert werden der Pin-Name des zu testenden Pins und die Frame-ID benötigt. Wie bereits bei dem Communication-Filter-Test beschrieben, wird mit der Konfiguration und Initialisierung des Filters begonnen, d.h. die Quelle und die Senke werden erstellt und an den Filter angebunden.

Der Lenkwinkel-Wertebereich, der von den weiterführenden Filtern verarbeitet wird, reicht von 0° bis 180°. Folglich muss der Aktuator-Filter alle Werte kleiner Null gleich Null und alle Werte größer 180 gleich 180 setzen. Um diese Testfälle zu generieren wird eine obere und untere Grenze festgelegt, demzufolge können die Grenzen in einen Schleifendurchlauf miteinbezogen werden. Zu diesem Zweck werden zwei Variablen deklariert. *f32LowerRange* gibt die untere Grenze an und wird als -1 gewählt, die *f32UpperRange* (obere Grenze) hingegen wird als 181 festgelegt. Anschließend wird eine Schleife benötigt, um in einer äquidistanten Schrittweite Test-Werte auf den Filter zu „feuern“. Daher wird eine Variable mit dem Namen *ui32MaxLoopCount* deklariert, eine maximale Anzahl der Schleifendurchläufe festzulegen.

Abbildung 48 zeigt die Konfiguration des Schleifendurchlaufs des Aktuator Filters. Mit der Bedingung in Zeile 204 wird sichergestellt, dass der erste Schleifendurchlauf die untere Grenze des Lenkwinkels (Zeile 207) behandelt. Die Anzahl der Durchläufe wird mit dem Schleifenzähler *nIdx* festgehalten. Eine weitere Bedingung in Zeile 209 legt den letzten Schleifendurchlauf fest, der die obere Grenze abprüfen soll. Die

Werte zwischen den Variablen *f32LowerRange* und *f32UpperRange* werden im letzten Abschnitt (ab Zeile 214) generiert. Über die zwei Grenzen (*range*), den maximalen Schleifendurchgängen und dem aktuellen Schleifenzähler wird die Schrittweite der Test-Werte errechnet (Zeile 217).

In einer weiteren Schleife wird der Aufbau jedes Übertragungsprotokolls und des jeweiligen Test-Wertes geprüft. Zu diesem Zweck wurde eine Hilfs-Klasse erzeugt, die alle erhaltenen Datenpakete in einem Speicher ablegt und anschließend über einen Zeiger auf das benötigte Datenpaket zugreifen kann. In der folgenden Schleife werden alle erhaltenen Daten chronologisch durchgegangen, so dass in jedem Schleifendurchlauf ein Sample auf die Plausibilität des Aufbaus geprüft wird.

```

202         // set the value
203         tFloat32 f32Value = 0.0f;
204         if (0 == nIndex)
205         {
206             // first value must test the lower border of data range
207             f32Value = f32LowerRange;
208         }
209         else if (ui32MaxLoopCount - 1 == nIndex)
210         {
211             // last value must test the upper border of data range
212             f32Value = f32UpperRange;
213         }
214         else
215         {
216             // the rest will be calculated by index, range and loop count
217             f32Value = (f32UpperRange - f32LowerRange) / ui32MaxLoopCount * nIndex;
218         }

```

Abbildung 48 Konfiguration des Schleifendurchlaufs

In der nachfolgenden Abbildung 49 ist die Überprüfung der Nutzdaten am Beispiel des Lenkwinkels aufgeführt.

```

288         if (nIndex == 0)
289         {
290             // lower range check (values smaller than 0 must be set to 0)
291             __adtf_test(achFrameData[0] == 0);
292         }
293         else if (nIndex == ui32MaxLoopCount)
294         {
295             // upper range check (values greater than 180 must be set to 180)
296             __adtf_test(achFrameData[0] == 180);
297         }
298         else
299         {
300             // the value must be the same (with rounding) as set while transmit
301             tFloat32 f32Value = ((f32UpperRange - f32LowerRange) * static_cast<tFloat32> (nIndex) / ui32MaxLoopCount);
302
303             __adtf_test_ext(static_cast<tChar> (f32Value + 0.5f) == achFrameData[0],
304                             cString::Format("Received data (%d) not as expected (%d)", achFrameData[0], static_cast<tChar> (f32Value + 0.5f)));
305         }

```

Abbildung 49 Überprüfen der Nutzdaten – Steering Angle

Von Zeile 288 bis 292 wird die untere Grenze geprüft. Mit dem Befehl `adtf_test(achFrameData[0] == 0)` wird überprüft, ob der Filter den Test-Wert auf null gesetzt hat. Ist dies der Fall wird durch die Hilfs-Klasse der Rückgabewert `NOERROR` zurückgegeben, um dem Anwender zu signalisieren, dass kein Fehler vorliegt. Dieses Verfahren wird mit der oberen Grenze von Zeile 293 bis 297 fortgeführt, indem der gespeicherte Wert mit 180 abgeglichen wird. Wenn der Wert nicht überschrieben wird, ist der Filter nicht funktionsfähig und es wird ein Fehler ausgegeben.

Der nächste Abschnitt (Zeile 298 bis 305) prüft alle Zwischenwerte, die zuvor über eine errechnete Schrittweite ermittelt wurden. In dem Filter werden die Nutzdaten von dem Datentyp Float in Integer umgewandelt, bei dieser Konvertierung wird die Nachkommastelle gerundet.

Um den Integer mit dem Soll-Wert abzugleichen wird in Zeile 301 die Schrittweite erneut berechnet und anschließend in Zeile 303 mit dem Korrekturwert 0,5 addiert. Mit dieser Vorgehensweise wird die korrekte Rundung sichergestellt, da die Nachkommastelle bei der Umwandlung in Integer abgeschnitten wird. Im Folgenden ist eine Rundung exemplarisch aufgeführt:

- Rundung im Filter $12,7 \rightarrow 13$
- Rundung durch Abschneiden $12,7 \rightarrow 12$
- Rundung mit Korrekturwert $12,7 + 0,5 = 13,2 \rightarrow 13$

Demzufolge können die korrekten Soll-Daten in der Senke zum Soll-Ist-Abgleich zur Verfügung gestellt werden. Anschließend wird im Fehlerfalle, d.h. wenn die Ist-Werte mit den Soll-Daten nicht übereinstimmen, ein formatierter String ausgegeben, der das Fehlerausmaß beschreibt. Ein formatierter String beinhaltet Platzhalter, die durch bestimmte Variablen vor der Ausgabe ersetzt werden können. Der String der zur Fehlerausgabe dient ist „*Sample count (Platzhalter 1) does not match (Platzhalter 2)*“. Platzhalter 1 wird durch den zuvor inkrementierten Datenpaket-Zähler ersetzt, während Platzhalter 2 die maximale Schleifenanzahl anzeigt.

Der Datentyp des Lights, des Emergency-Stops und des Watchdogs ist ebenfalls identisch, aus diesem Grund wurden die Testfälle dieser Pins in einer generischen Funktion zusammengefasst. Die Funktion `DoActuatorBoolTest` benötigt ebenfalls den Pin-Namen und die Frame-ID als Übergabewerte. Die Initialisierung des Tests hat ebenfalls einen ähnlichen Aufbau wie der Beschleunigungs- und Lenkwinkeltests.

Wie bereits beim Beschleunigungs- und Lenkwinkeltest werden die Test-Samples in einer For-Schleife generiert. Die For-Schleife, welche die Quelle bei diesem Test-Prinzip darstellt, wird im Folgenden dargestellt.

```
367     for (tUInt32 nIdx = 0; nIdx < ui32MaxLoopCount; nIdx++)
368     {
369         // create and allocate the sample
370         cObjectPtr<IMediaSample> pSample;
371         __adtf_test_result(_runtime->CreateInstance(OID_ADTF_MEDIA_SAMPLE, IID_ADTF_MEDIA_SAMPLE, (tVoid**)&pSample));
372         __adtf_test_result(pSample->AllocBuffer(nSize));
373
374         // get the coder
375         cObjectPtr<IMediaCoder> pCoder;
376         __adtf_test_result(pTypeDesc->WriteLock(pSample, &pCoder));
377
378         // set the value
379         tBool bValue = tFalse;
380
381         // send every the first and then every second time on signal
382         if(!(nIdx % 2))bValue = tTrue;
383
384         // use the coder to set the value
385         if(chFrameId != ID_ARD_ACT_WD_ENABLE){
386             __adtf_test_result(pCoder->Set("bValue", &bValue));
387         }else{
388             __adtf_test_result(pCoder->Set("bEmergencyStop", (tVoid*)&bValue));
389         }
390     }
```

Abbildung 50 Test-Samples werden generiert

Auf das Erstellen von Samples und den Coder wird in dem Kapitel 4.1.1 näher eingegangen. Um die Ansteuerung des Lichts zu realisieren wird eine boolesche Variable benötigt. Diese Variable wird in Zeile 279 zur Initialisierung auf den Wert „Falsch“ gesetzt. Allerdings ist in Zeile 382 eine Bedingung definiert, die mittels des Modulo-Operators (%) jeden zweiten Schleifendurchlauf ermittelt und wenn dies gegeben ist, den Boolean auf „Wahr“ (*tTrue*) setzt. Der erste Schleifendurchlauf steuert die Leuchten an und der darauffolgende Durchlauf schaltet diese wieder aus. In dem Abschnitt von Zeile 385 bis 386 wird der Boolean unter der Bedingung, dass die Frame-ID nicht von dem Emergency-Stop ist, in dem Sample an die richtige Stelle gesetzt. Ansonsten setzt der Coder in dem *else*-Block (ab Zeile 387) den Boolean zur Ansteuerung des Emergency-Stops ein. Der Vorgang wird explizit durch den String „*bEmergencyStop*“ eingeleitet.

In der Senke wird zu Beginn die richtige Anzahl der Datenpakete überprüft.


```
406 // first check for the right number of samples
407 if (strInputPinName == "headLightEnabled"){
408     // by sending headLight data the aktors filter automatically sends head and backlight data
409     // so the received samples have to be twice as much as the transmitted
410     __adtf_test_ext(pSampleSink->m_ui32SampleCount == ui32MaxLoopCount * 2,
411         cString::Format("Sample count (%d) does not match (%d)", pSampleSink->m_ui32SampleCount, ui32MaxLoopCount * 2));
412 }else if(strInputPinName == "Watchdog_Alive_Flag" || chFrameId == ID_ARC_ACT_WD_ENABLE){
413     // only enabled watchdog signals with the value true shall be send to the arduino,
414     // so only have of the send samples should be received
415     __adtf_test_ext(pSampleSink->m_ui32SampleCount == ui32MaxLoopCount / 2,
416         cString::Format("Sample count (%d) does not match (%d)", pSampleSink->m_ui32SampleCount, ui32MaxLoopCount / 2));
417 }else{
418     // every send sample should generate one received sample
419     __adtf_test_ext(pSampleSink->m_ui32SampleCount == ui32MaxLoopCount,
420         cString::Format("Sample count (%d) does not match (%d)", pSampleSink->m_ui32SampleCount, ui32MaxLoopCount));
```

Abbildung 51 Anzahl der Samples überprüfen

In Zeile 407 ist mittels einer If-Abfrage der Pin der Front- und Heckleuchten ausgewählt. Bei diesem Test wird die Gesamtanzahl der Schleifendurchläufe auf das Doppelte angehoben (Zeile 410), da bei der Ansteuerung der Front- und Heckleuchten immer zwei Samples in einem Durchlauf geschickt werden. Anschließend wird zur Fehlerausgabe ein formatierter String genutzt, der in Zeile 411 implementiert ist.

Bei dem Watchdog kommt es ausschließlich auf die True-Werte an. Die Bits, die auf eins gesetzt sind werden von dem Watchdog verarbeitet, um das Motorrelais geschlossen zu halten. Aus diesem Grund werden halb so viele Samples in der Senke ankommen, wie ursprünglich gesendet wurden. In Zeile 415 wird folglich die maximale Schleifenanzahl durch zwei geteilt. Bei Emergency ist es so, dass jedes zweite Mal zwei True werte gesendet werden einmal an Watchdog und das andere Mal an das Relais.

Der Else-Block ab der Zeile 417 behandelt die restlichen Leuchten, welche mit einem Sample angesteuert werden und daher die Schleifenanzahl bei der ursprünglichen Größe belassen wird.

Anschließend wird noch der Test des Übertragungsprotokolls ähnlich wie bei dem Communication-Filter realisiert. Ein wesentlicher Unterschied ist die Größe der Nutzdaten, da bei den Samples für den Watchdog lediglich ein Byte benötigt wird, während bei den restlichen Daten zwei Byte angehängt sind. In der folgenden Darstellung ist die If-Bedingung aufgeführt, die zwischen den beiden Soll-Datengrößen hin- und herschaltet.

```
486         if(chFrameId == ID_ARD_ACT_WD_TOGGLE || chFrameId == ID_ARD_ACT_WD_ENABLE)
487         {
488             __adtf_test(i8DataLength == 1);
489         }
490         else
491         {
492             __adtf_test(i8DataLength == 2);
493         }
```

Abbildung 52 Differenzierung der Datenlängen des Watchdogs und der restlichen Pins

Mit diesem Test kann die Tauglichkeit dieses Filters für den Einsatz im Gesamtsystem validiert werden, sowie die Vollfunktionsfähigkeit des Software-Moduls ermittelt werden.

4.1.3 Arduino Sensors

Der Test des Sensor-Filters unterscheidet sich in dem grundlegenden Test-Aufbau nicht von dem Aktuator-Filter-Test. Wie bei dem Aktuator-Test gibt es ein Set-Up, welches die Testumgebung schafft und in dem Communication-Filter-Test ausführlich beschrieben wurde, eine Hilfsklasse für die Realisierung der Senke und eine Test-Funktion für den korrekten Aufbau des Übertragungsprotokolls.

Die Besonderheit bei diesem Filter-Test ist die große Bandbreite an unterschiedlichen Nutzdatenlängen. Beispielsweise hat das Sample des Lenkwinkels eine Datenlänge von zwei Byte, während ein Infrarot-Sensor 18 Byte in Anspruch nimmt. Folglich müssen die Datengrößen vor dem Senden der Test-Daten festgelegt werden, indem die einzelnen Sensoren über die jeweilige Frame-ID identifiziert und die entsprechenden Datenlängen deklariert werden. Dieser Vorgang ist in der folgenden Abbildung 53 dargestellt.

```
972     if(chFrameId == ID_ARD_SENS_STEER_ANGLE)
973     {
974         i8DataLength = 2;
975     }
976     if(chFrameId == ID_ARD_SENS_WHEELENC)
977     {
978         i8DataLength = 8;
979         ui8SizeOfValue = 4;
980     }
981     if(chFrameId == ID_ARD_SENS_IMU)
982     {
983         i8DataLength = 16;
984     }
985     if(chFrameId == ID_ARD_SENS_IR)
986     {
987         i8DataLength = 18;
988     }
989     if(chFrameId == ID_ARD_SENS_PHOTO)
990     {
991         i8DataLength = 2;
992     }
993     if(chFrameId == ID_ARD_SENS_US)
994     {
995         i8DataLength = 8;
996     }
997     if(chFrameId == ID_ARD_SENS_VOLTAGE)
998     {
999         i8DataLength = 4;
1000     }
```

Abbildung 53 Deklaration der Soll-Datenlängen

Die Test-Daten sollen den kompletten Bereich der Nutzdaten enthalten, d.h. bei einer Datenlänge von 8 Byte, also 64 Bit, ergeben sich hierfür $1,84 \cdot 10^{19}$ Möglichkeiten. Folglich kommt ein 32-Bit Integer schnell an seine Grenzen, daher werden die Test-Daten durch die nachfolgende For-Schleife, in 8-Bit Abschnitten, generiert.

```
1003     for (tUInt32 nIdx = 0; nIdx < ui32MaxLoopCount; nIdx++)
1004     {
1005         // Because of 8 bit in each byte we want to increase the next higher byte with each full lower byte
1006         if(chData[ui8PositionToWrite] == static_cast<char> (255))
1007         {
1008             chData[ui8PositionToWrite+1] += 1;
1009             ui16Value = 0;
1010         }
1011         // If the current value range is full go to the next value range
1012         if(chData[ui8PositionToWrite+1] == static_cast<char> (255))
1013         {
1014             ui16Value = 0;
1015             ui8PositionToWrite += ui8SizeOfValue;
1016         }
1017         // Check if the end of the current protocol is reached if yes quit the loop
1018         // but set the goal of received samples to match the send samples
1019         if(ui8PositionToWrite == i8DataLength)
1020         {
1021             ui32MaxLoopCount = nIdx;
1022             break;
1023         }
1024
1025         // set the actual value
1026         chData[ui8PositionToWrite] = ui16Value;
1027
1028         // transmit the value
1029         transmitMediaSample(pSampleSource, chFrameId, i8DataLength, chData);
1030         // increase the value for the next time.
1031         ui16Value++;
1032     }
```

Abbildung 54 Generierung der Test-Daten

Die maximale Anzahl an Schleifendurchgängen ist auf zwei Millionen festgelegt worden, um die hohen Datenlängen abzudecken. In einem Array von dem Datentyp *Char* werden die Test-Daten in kleinen Einheiten über die Variable *ui16Value* gespeichert (Zeile 1026). In Zeile 1029 wird der Test-Wert an den jeweiligen Input-Pin übergeben, hierbei kann durch die Übergabewerte genau festgelegt werden an welchem Pin Daten beaufschlagt werden. In der letzten Zeile der Schleife wird der Test-Wert inkrementiert und so für den nächsten Schleifendurchlauf vorbereitet.

Nach 255 Schleifendurchgängen ist ein Byte voll und somit die maximale Datenlänge eines *Char's* ausgereizt, für diesen Fall wird in der ersten if-Bedingung (Zeile 1006 bis 1010) die Schreibposition im Array um eine Stelle weitergeschoben und die Variable zum Speichern der Test-Daten wird wieder auf null zurückgesetzt. Diese Vorgehensweise wird in dem Bereich von Zeile 1012 bis 1016 fortgesetzt.

Ist die maximale Datenlänge, die zuvor deklariert wurde (Abbildung 53), erreicht, wird die Schleife automatisch unterbrochen (Zeile 1022).

Wie in Abschnitt 2.3.2 bereits beschrieben hat der Sensor-Filter Eingang-Pin für die Sensordaten von dem Arduino-Board und mehrere Ausgangspins. Aus diesem Grund sind mehrere Senken-Funktionen implementiert (für jeden Pin eine Funktion),

die jeden Datenstrom einzeln auswerten. In der nachfolgenden Abbildung sind die Funktionen zur Realisierung der einzelnen Senken aufgelistet.

```

195  /******
196  /* These are just a helper functions to evaluate the received data
197  /******
198
199  tTestResult evaluateSteerAngle(cMediaSampleSink *pSampleSink, tUInt32 ui32MaxLoopCount) { ... }
251  tTestResult evaluateRPM(cMediaSampleSink *pSampleSink, tUInt32 ui32MaxLoopCount) { ... }
317  tTestResult evaluateGyro(cMediaSampleSink *pSampleSink, tUInt32 ui32MaxLoopCount) { ... }
431  tTestResult evaluateAcc(cMediaSampleSink *pSampleSink, tUInt32 ui32MaxLoopCount) { ... }
534  tTestResult evaluateIR(cMediaSampleSink *pSampleSink, tUInt32 ui32MaxLoopCount) { ... }
655  tTestResult evaluatePHOTO(cMediaSampleSink *pSampleSink, tUInt32 ui32MaxLoopCount) { ... }
707  tTestResult evaluateUS(cMediaSampleSink *pSampleSink, tUInt32 ui32MaxLoopCount) { ... }
792  tTestResult evaluateVOLTAGE(cMediaSampleSink *pSampleSink, tUInt32 ui32MaxLoopCount) { ... }

```

Abbildung 55 Übersicht der Senken-Funktionen

In jeder Senke wird, ähnlich wie bei dem Aktuator-Test, der Coder initialisiert und das Übertragungsprotokoll überprüft. Um den Aufbau und die Funktionsweise einer dieser Senken zu verstehen ist in der nächsten Darstellung exemplarisch ein Ausschnitt der Funktion zur Evaluierung der Ultraschall-Sensorik aufgeführt.

```

756  if(nIdx < ui32MaxValue*1)
757  {
758      if(ui16FrontLeft!=nIdx)
759      {
760          __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should be (%d)",ui16FrontLeft,nIdx));
761      }
762      // then test the Front right data
763      }else if(nIdx < ui32MaxValue*2)
764      {
765          if(ui16FrontRight!=nIdx-ui32MaxValue*1)
766          {
767              __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should be (%d)",ui16FrontRight,nIdx));
768          }
769      // now the rear left data
770      }else if(nIdx < ui32MaxValue*3)
771      {
772          if(ui16RearLeft!=nIdx-ui32MaxValue*2)
773          {
774              __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should be (%d)",ui16RearLeft,nIdx-ui32MaxValue*2));
775          }
776      }

```

Abbildung 56 Ausschnitt der Senke für die Ultraschall-Sensorik

Das Fahrzeug besitzt insgesamt vier Ultraschall-Sensoren, zwei an der Fahrzeug-Front und zwei am Heck. Jeder Werte-Bereich wird über eine If-Bedingung aufgegriffen und einem Sensor zugeteilt. In Zeile 760 wird der linke Frontsensor mit den Test-Daten beaufschlagt, zu diesem Zweck wird der Schleifenzähler eingesetzt, da dieser Wert sich ohnehin in jedem Durchlauf inkrementiert und folglich keine Test-Lücken entstehen. Mittels einer verschachtelten If-Bedingung in Zeile 758 werden die Test-Daten nach einer bestimmten Anzahl abgebrochen und der zweite Wertebereich wird in Zeile 767 auf den rechten Frontsensor beaufschlagt. Um den Schleifenzähler weiter zu diesem Zweck verwenden zu können, wird in der betreffenden Bedingung der eigentliche Wertebereich verdoppelt (Zeile 763: *nIdx* <

*ui32MaxValue*2*). Demnach wird der Schleifenzähler von dem letzten Wert weiter inkrementiert und in dem If-Block von Zeile 763 bis 768 der identische Werte-Bereich abgeprüft. Die verschachtelte If-Bedingung, die das Abbruch-Kriterium beinhaltet, subtrahiert von dem maximalen Schleifendurchlauf, der für diesen Sensor vorgesehen ist, den Werte-Bereich (Zeile 765: *ui16FrontRight != nldx - ui32MaxValue*1*). Diese Vorgehensweise wird mit den Hecksensoren fortgesetzt.

Die restlichen Sensoren werden durch ähnliche Senken-Funktionen geprüft.

Die Sensorik stellt die Verbindung zwischen dem Fahrzeug und der Umwelt her, somit ist das autonome Fahren nur möglich, indem der Filter alle Sensordaten korrekt erfasst und diese weiterverarbeitet. Aus diesem Grund stellt der Test alle erdenklichen Testfälle auf und gewährleistet somit eine umfängliche Funktionsprüfung. Demzufolge erfüllt der Sensortest alle an ihn gestellten Anforderungen.

4.2 Umsetzung des Integrationstests

Wie bereits in Abschnitt 3.1.2 erwähnt ist der Integrationstest ein Verbundtest. Der Test ist in vier verschiedenen Teilen realisiert worden. In den ersten drei Modulen werden die Tests für Beschleunigung, Lenkwinkel und Licht nochmals separiert ausgeführt. Infolgedessen wird im letzten Teil der Verbund der drei Filter geprüft, mit dem Beaufschlagen der drei Aktoren. Die einzelnen Filter werden miteinander verbunden und es findet ein Test auf Systemebene statt. Zu diesem Zweck werden die Ausgangspins des Aktuator-Filters mit den Eingangs-Pins des Communication-Filters verknüpft, während die Ausgänge des Communication-Filters mit dem Sensor-Filter verbunden werden. Dieser Vorgang ist nachfolgend in Quellcode-Form dargestellt.

```
610 // connect the output pin of the Aktors Filter with the input pin of the Communication Filter
611 __adtf_test_result(pArdCommInPin->Connect(pArdAktOutPin));
612
613 // connect the output pin of the Communication Filter with the input pin of the Sensors Filter
614 __adtf_test_result(pArdSensInPin->Connect(pArdCommOutPin));
```

Abbildung 57 Verbinden der zu testenden Filter

Aus diesem Grund wird der Communication-Filter-Test als Basis für den Integrationstest verwendet. Der gravierende Unterschied zu dem herkömmlichen Communication-Filter-Test ist, dass die Test-Daten nicht direkt über die erstellten Quellen, sondern über den angebotenen Aktuator-Filter beaufschlagt werden. Das

Anbinden einer universalen Quelle, die die Adressierung manuell übernimmt ist somit nicht möglich. An jeden Input-Pin wird eine Quelle angehängt, denn die Adressierung findet letztendlich, über die Eingangs-Pins, im Filter statt. In der folgenden Abbildung 58 wird gezeigt, wie die Quellen erstellt werden.

```

581 // create a simple sample source for every input pin
582 cObjectPtr<cOutputPin> pAccSampleSource = new cOutputPin();
583 __adtf_test_result(pAccSampleSource->Create("sampleSource", new cMediaType(0,0,0,"tSignalValue")));
584 cObjectPtr<cOutputPin> pSteerAngleSampleSource = new cOutputPin();
585 __adtf_test_result(pSteerAngleSampleSource->Create("sampleSource", new cMediaType(0,0,0,"tSignalValue")));
586 cObjectPtr<cOutputPin> pHeadLightSampleSource = new cOutputPin();
587 __adtf_test_result(pHeadLightSampleSource->Create("sampleSource", new cMediaType(0,0,0,"tBoolSignalValue")));
588 cObjectPtr<cOutputPin> pTurnLeftSampleSource = new cOutputPin();
589 __adtf_test_result(pTurnLeftSampleSource->Create("sampleSource", new cMediaType(0,0,0,"tBoolSignalValue")));
590 cObjectPtr<cOutputPin> pTurnRightSampleSource = new cOutputPin();
591 __adtf_test_result(pTurnRightSampleSource->Create("sampleSource", new cMediaType(0,0,0,"tBoolSignalValue")));
592 cObjectPtr<cOutputPin> pBrakeLightSampleSource = new cOutputPin();
593 __adtf_test_result(pBrakeLightSampleSource->Create("sampleSource", new cMediaType(0,0,0,"tBoolSignalValue")));
594 cObjectPtr<cOutputPin> pReverseLightSampleSource = new cOutputPin();
595 __adtf_test_result(pReverseLightSampleSource->Create("sampleSource", new cMediaType(0,0,0,"tBoolSignalValue")));

```

Abbildung 58 : Erstellen der Quellen für den Aktuator-Filter

Die Senken des Sensor-Filters werden ebenfalls einzeln an die Ausgänge des Sensor-Filters angebunden, da die Output-Daten des Communication-Filters direkt in den angebundenen Sensor-Filter übergehen und dort aufgeteilt werden.

In der folgenden Abbildung wird die Erstellung der Senken für die Ausgänge des Sensor-Filters dargestellt.

```

616 // create a sample sink for every output of the sensors filter
617 cObjectPtr<cMediaSampleSink> pUSSampleSink = new cMediaSampleSink();
618 cObjectPtr<cMediaSampleSink> pIRSampleSink = new cMediaSampleSink();
619 cObjectPtr<cMediaSampleSink> pSteerAngleSampleSink = new cMediaSampleSink();
620 cObjectPtr<cMediaSampleSink> pAccSampleSink = new cMediaSampleSink();
621 cObjectPtr<cMediaSampleSink> pGyroSampleSink = new cMediaSampleSink();
622 cObjectPtr<cMediaSampleSink> pRPMSampleSink = new cMediaSampleSink();
623 cObjectPtr<cMediaSampleSink> pVoltageSampleSink = new cMediaSampleSink();

```

Abbildung 59 Erstellung der Senken für den Sensor-Filter

Zuvor werden die üblichen Set-Ups, Initialisierungen und die Konstruktionen der Quellen- und Senkenstruktur erstellt. Vor dem Test, müssen neben den Initialisierungen und Konfigurationen der maximale Schleifenzähler, der die Testdauer und die Genauigkeit des Tests ausmacht, und die sogenannte *Waittime* festgelegt werden. Für den optimalen Betrieb braucht der Motor eine bestimmte Vorlaufzeit, die durch die Variable *Waittime* dargestellt wird. Für andere Betriebssysteme werden andere Parameter verwendet.

```
649 | // define the Test duration and the Waittime after Watchdog Signal is on
650 | ui32MaxLoopCount = 250;
651 | // Defines the Time from when the Watchdog is active to when the first actuator sample is sent.
652 | // This is because of the initialisation of the motor controller
653 | ui32Waittime = 5000;
```

Abbildung 60 Maximaler Schleifenzähler und Vorlaufzeit deklarieren

Die Beaufschlagung der Test-Daten wird durch eine Rampen-Ansteuerung realisiert, die bereits in Abschnitt 4.1.1 ausführlich beschrieben wurde. Bei den Test-Daten für die Beschleunigung und den Lenkwinkel ist eine rampenförmige Ansteuerung leicht nachzuvollziehen. Die Räder werden linear vom Stillstand bis zur Höchstgeschwindigkeit, sowohl vorwärts als auch rückwärts, beschleunigt, während der Lenkwinkel von der Geradeausfahrt linear auf den rechten und anschließend den linken Anschlag geführt wird. Die Leuchten werden ebenfalls über die vier verschiedenen Rampensegmente an- und abgeschaltet, da eine einheitliche Testfunktion Programmieraufwand und Rechenarbeit spart.

Ein weiterer entscheidender Unterschied ist das Generieren des Watchdog-Signals. Zuvor wurde über eine manuell abgestimmte Periode das Watchdog-Signal simuliert, um das Motor-Relais geschlossen zu halten und den Communication-Test ablaufen lassen zu können. In der folgenden Abbildung 61 ist zu sehen, dass Ausgang des Watchdog-Filters aufgegriffen wird.

```
525 | // get the output pin of the WD Filter
526 | cObjectPtr<IPin> pWDOutPin;
527 | __adtf_test_result_ext(pWDFilter->FindPin("WatchdogAliveSignal", IPin::PD_Output, &pWDOutPin), "Unable to find pin WatchdogAliveSignal");
```

Abbildung 61 Aufgreifen des Watchdog-Output-Pins

Da der Aktuator-Filter einen Input-Pin für das Watchdog-Signal besitzt, ist es sinnvoll den Watchdog-Filter anzubinden und das Signal von dem Filter zu beziehen. Diese Vorgehensweise hat den großen Vorteil, dass keine weiteren Maßnahmen zur Generierung des Signals ergriffen werden müssen.

Außerdem ist der Integrationstest auf diese Weise ein Stück näher an der Realität, bzw. dem regulären Betrieb des Fahrzeugs. Durch diesen Umstand erfüllt der Integrationstest seine Zielsetzung in noch höherem Maße. Die Anforderung an den Integrationstest ist die Praktikabilität und Funktionalität der einzelnen Software-Module auf Gesamtsystem-Ebene sicherzustellen. Demzufolge wurde der Test vollumfänglich umgesetzt.

5 Zusammenfassung

Der fünfte und letzte Abschnitt dieser Studienarbeit arbeitet das Ergebnis, die Schlussfolgerung und den Ausblick heraus. Hierbei werden die Erwartungen und Anforderungen aus Abschnitt 3, sowie deren verschiedenen Testresultate und Bedeutungen aus Abschnitt 4 hinsichtlich einer Ergebnisauswertung in Zusammenhang gesetzt. Zuletzt wird noch ein Ausblick aufgezeigt, der die weiterführenden Möglichkeiten dieser Studienarbeit verdeutlichen soll.

5.1 Ergebnis und Bewertung

Die Ergebnisse der Komponenten-, als auch des Integrationstests konnten vollständig mit einer durchweg positiven Erkenntnis durchlaufen und somit auch ebenso positiv bewertet werden.

Der Komponententest des Communication Filters lieferte die grenzwertige Sleep-Zeit von zehn Mikrosekunden. Unterhalb dieser Grenze konnte ebenfalls erkannt werden, ob Datenpakete noch gesendet werden oder nicht. Liegt die Sleep-Zeit circa zwischen neun und zehn Mikrosekunden, so wurden zwar noch Datenpakete empfangen, der Test allerdings lieferte ein negatives Ergebnis. Fällt die Datenrate unter neun Mikrosekunden, so konnten keine Datenpakete mehr empfangen werden, da das Arduino-Board auf Grund von Überbelastung keine Sensorwerte mehr liefern konnte. Oberhalb von zehn Mikrosekunden konnte sowohl die benötigte Anzahl der Datenpakete, als auch die, durch die ansteigenden Daten entstandene, lineare Rampe nachvollzogen werden.

Die Quintessenz dieses Tests und somit auch des Filters ist, dass der Arduino Communication Filter sehr gut arbeitet, solange die richtige Datenrate beibehalten wird.

Auch beim Test des Arduino Actuator Filter konnten alle Funktionen wie erwartet abgeprüft werden. Der Wertebereich wurde beschränkt, sodass keine zu großen oder zu kleinen Werte gesendet werden konnten und alle eingehenden Daten wurden ebenfalls ordnungsgemäß verpackt und weitergeleitet. Die Ausnahme bildet hier das Signal des *Emergency Stops*. Dieses Signal konnte nicht erfolgreich durchgesendet werden. Bei der Weiterleitung des Problems an die Softwareentwicklung der Firma BFFT Fahrzeugtechnik, die für den Aufbau und die Software des Modellfahrzeugs verantwortlich sind, konnte der Fehler in der

Basissoftware gefunden und behoben werden. Noch während der Testphase konnte anschließend mit einem funktionierenden *Emergency Stop* Signal weitergearbeitet werden.

Der Arduino Sensors Filter verhielt sich auch wie gewünscht. Die Sensorendaten wurden alle durchgesendet und wie erwartet aufgeteilt. Im Lauf der Testdurchläufe wurde mit einem anderen Target als dem Rechner des Fahrzeugs geprüft. Dabei zeigten sich Probleme bei der Übergabe der Frame-IDs. Dieses Problem war auf die Wertebereichszuweisung hinsichtlich eines Datentyps in der Basissoftware zurückzuführen und konnte ebenfalls behoben werden.

Beim Integrationstest der drei Komponenten konnte gezeigt werden, dass die Basissoftware, bezüglich der drei getesteten Filter, gut funktioniert. Jeder Softwarebaustein erfüllt, integriert in das Teilsystem, die Aufgabe, die angefordert wird. Bezüglich der Datenrate gibt es allerdings keinen Mechanismus in der Basissoftware, der eine zu hohe Datenrate abfangen würde, ähnlich wie es mit der Wertebereichsbeschränkung, hinsichtlich Arduino Actuator Filter, geschieht. Wird hier eine zu niedrige Sleep-Zeit in zuvor geschalteten Filtern vorgenommen, so wird das Arduino-Board überlastet werden. Dieser Vorschlag wurde bereits an die Firma BFFT Fahrzeugtechnik weitergeleitet und ist ein positives Beispiel für den nutzvollen Gehalt dieser Arbeit, da diese fehlende Datenratenbeschränkung bereits in einer zukünftigen Softwareaktualisierung integriert werden soll.

5.2 Ausblick

Die vorliegende Studienarbeit hat Tests für die grundlegenden Filter zur Kommunikation mit dem Arduino-Board abgehandelt. Dabei konnte die Funktionsfähigkeit der Basissoftware geprüft werden.

Infolgedessen konnte eine Basis für das kommende Vorgehen geschaffen werden. Weitere Tests, der nicht behandelten Filter können an die bereits bestehenden Tests angelehnt werden und auf deren Grundstruktur aufbauen. Änderungen an den bestehenden Filtern können unkompliziert evaluiert werden. Die Verifikation der Ergebnisse mittels der erstellten Filter-Tests liefert dazu ein schnelles und aussagekräftiges Ergebnis.

Diese Arbeit hat somit den Grundstein zu einer voll funktionsfähigen und robusten Software gelegt.

Glossar

Array: Das *Array* beschreibt eine Aneinanderreihung von Elementen des gleichen Datentyps.

Boolean: Der Typ *Boolean* (kurz: *Bool*) ist ein Datentyp in der Informatik und dient zum Speichern der logischen Werte True und False.

Bit: Das *Bit* ist eine Binärziffer und ebenso die kleinste darstellbare Datenmenge. Dabei kann jedes einzelne Bit stets zwei Zustände annehmen.

Byte: Ein *Byte* besteht aus acht aufeinanderfolgenden Bits und wird u.a. zur Adressierung in Rechnern verwendet.

Casten: *Casten* bezeichnet das Vorgehen in der Informatik, einen bestimmten Datentyp in einen entsprechend anderen Datentyp zu überführen.

Char: Der Typ *Char* ist ein Datentyp in der Informatik und dient zum Speichern einzelner Unicode-Zeichen. Mit diesem Typ können alle 65536 Unicode-Zeichen dargestellt werden.

Float: Der Typ *Float* ist ein Datentyp in der Informatik und dient zum Speichern von Gleitkommazahlen. Diese Zahl wird als 32-Bit großer Wert dargestellt.

For-Schleife: Die *For-Schleife* ist eine Schleifenform in der Informatik und besitzt Schleifenbedingungen, die noch vor dem ersten Schleifendurchlauf überprüft werden. Diese Schleifenart wird v.a. bei zählergesteuerten Schleifen bevorzugt, die eine konstante Anzahl an Durchläufen besitzen.

If-Abfrage: Die *If-Abfrage* bildet in der Informatik eine einfache Verzweigung, die nach der Bedingungsabfrage (durch If oder Else verzweigt) eine Anweisung ausführt.

Inkrementierung: Die Begrifflichkeit der *Inkrementierung* bezeichnet in der Informatik das Erhöhen einer Variablen um den Wert eins.

Integer: Der Typ *Integer* ist ein Datentyp in der Informatik und dient zum Speichern von ganzen Zahlen. Diese Zahl wird als 16- oder 32-Bit großer Wert dargestellt.

Modulo-Operator: Der Modulo-Operator ist ein Divisionsalgorithmus der als Rückgabewert den Rest einer Division ausgibt. Zwei Zahlen die Vielfache voneinander sind haben demnach den Rückgabewert Null.

Signed: Der Begriff *Signed* kennzeichnet vorzeichenbehaftete Werte. Beispiel hierfür wäre ein 32 Bit großer Integer-Wert: *i32Value*.

Unsigned: Der Begriff *Unsigned* kennzeichnet vorzeichenlose Werte. Beispiel hierfür wäre ein 32 Bit großer Integer-Wert: ui32Value.

Literaturverzeichnis

- Audi Electronics Venture GmbH. (2014). *ADTF Software-Umgebung für Applikationen und Tool*. Abgerufen am 13. November 2014 von <http://www.audi-electronics-venture.de/aev/brand/de/leistungen/Entwicklungstools/adtf.html>
- Belke, R. (2009). *Schulungsunterlagen - Automotive Data and Time Triggered Framework*. Ingolstadt.
- BFFT Gesellschaft für Fahrzeugtechnik GmbH. (10. März 2014). Audi Autonomous Driving Cup - Manual 2015. Gaimersheim.
- Bosch Mobility Solutions. (2014). Ultraschallsensoren Fahrerassistenzsysteme - Einparkhilfe.
- Decker, P. D. (31. Mai 2012). Überholt die Maschine den Menschen. (E. Blum, Interviewer)
- Feess, P. D. (2014). *GABLER WIRTSCHAFTSLEXIKON*. Abgerufen am 6. November 2014 von Definition Senke: <http://wirtschaftslexikon.gabler.de/Definition/senke.html>
- Grewal, M., Weill, L., & Andrews, A. (2007). *Global Positioning Systems, Inertial Navigation, and Integration*. Wiley-Interscience.
- Hackenberg, P. D.-I. (17. Oktober 2014). Audi RS7 piloted driving concept . (M. Sommer, Interviewer)
- Krieger, P. D. (2014). *GABLER WIRTSCHAFTSLEXIKON*. Abgerufen am 11. November 2014 von Definition Quelle: <http://wirtschaftslexikon.gabler.de/Definition/quelle.html>
- Mario Winter, M. E.-M. (2012). *Der Integrationstest - Von Entwurf und Architektur zur Komponenten- und Systemintegration*. Hanser.
- Müller, A. (2008). *Evolutionary Computation*. Abgerufen am 5. November 2014 von Unit-Tests (Komponententests): <http://www.evocomp.de/softwareentwicklung/unit-tests/unittests.html>
- Schenke, G. (Januar 2013). *Mechatronik - Servomotoren*. Abgerufen am 28. Oktober 2014 von http://www.et-inf.fho-emden.de/~elmalab/mechatronik/download/Mechatronik_4.pdf

Sharp. (1. Dezember 2006). Datenblatt Sharp GP2Y0A21YK0F Short - Range & Sharp GP2Y0A02YK0F Mid - Range.

Anhang

Programm-Code: Arduino Communication Filter (Modul)

```
/**
 *
 * AADC Arduino Communication filter tests
 *
 * @file
 * Copyright &copy; Audi Electronics Venture GmbH. All rights reserved
 *
 * $Author: VG8D3AW $
 * $Date: 2013-02-06 16:30:41 +0100 (Mi, 06. Feb 2013) $
 * $Revision: 18162 $
 *
 * @remarks
 *
 */

#include "stdafx.h"
#include "tester_aadc_arduino_comm.h"

IMPLEMENT_TESTER_CLASS(cTesterAADCArduinoComm,
    "2",
    "AADC Arduino Communication filter",
    "This test makes sure that the AADC Arduino Communication filter works
as expected",
    "");

/*
Setup for this test
*/
void cTesterAADCArduinoComm::setUp()
{
    // create the environment
    SERVICE_ENV_SETUP;

    // minimum needed ADTF services
    SERVICE_ENV_ADD_PLUGIN("adtf_clock.srv");
    SERVICE_ENV_ADD_PLUGIN("adtf_kernel.srv");
    SERVICE_ENV_ADD_PLUGIN("adtf_sample_pool.srv");
    SERVICE_ENV_ADD_PLUGIN("adtf_namespace.srv");
    SERVICE_ENV_ADD_PLUGIN("adtf_media_description.srv");

    // register services
    SERVICE_ENV_REGISTER_SERVICE(OID_ADTF_REFERENCE_CLOCK, "referenceclock",
IRuntime::RL_Kernel);
    SERVICE_ENV_REGISTER_SERVICE(OID_ADTF_KERNEL, "kernel", IRuntime::RL_System);
    SERVICE_ENV_REGISTER_SERVICE(OID_ADTF_SAMPLE_POOL, "samplepool",
IRuntime::RL_System);
    SERVICE_ENV_REGISTER_SERVICE(OID_ADTF_NAMESPACE, "namespace",
IRuntime::RL_System);
    SERVICE_ENV_REGISTER_SERVICE(OID_ADTF_MEDIA_DESCRIPTION_MANAGER, "mediadesc",
IRuntime::RL_System);

    // add filter plugin
    FILTER_ENV_ADD_PLUGIN("aadc_arduinoCommunication.plb");
```

```

    // set runlevel system
    SERVICE_ENV_SET_RUNLEVEL(IRuntime::RL_System);

    // set the path to the MediaDescription-Files
    // (relative to the current working dir which is the tester dir)
    cFilename strDescrFileADTF =
    "../../../src/adtfBase/AADC_ADTF_BaseFilters/description/aadc.description";
    cFilename strDescrFile =
    "../../../src/adtfBase/AADC_ADTF_BaseFilters/description/aadc.description";

    // create absolute path
    strDescrFileADTF =
    strDescrFileADTF.CreateAbsolutePath(cFileSystem::GetCurDirectory());
    strDescrFile = strDescrFile.CreateAbsolutePath(cFileSystem::GetCurDirectory());

    // get the media description manager
    cObjectPtr<IMediaDescriptionManager> pMediaDesc;
    __adtf_test_result_ext(_runtime->GetObject(OID_ADTF_MEDIA_DESCRIPTION_MANAGER,
    IID_ADTF_MEDIA_DESCRIPTION_MANAGER, (tVoid**) &pMediaDesc),
    "unable to get media description manager");

    // get the config from the media description manager
    cObjectPtr<IConfiguration> pConfig;
    __adtf_test_result(pMediaDesc->GetInterface(IID_ADTF_CONFIGURATION, (tVoid**)
    &pConfig));

    // set the path to the description files
    __adtf_test_result(pConfig->SetPropertyStr("media_description_files",
    cString::Format("%s;%s", strDescrFileADTF.GetPtr(), strDescrFile.GetPtr())));

    // check if the media description manager loads the description correctly
    __adtf_test(pMediaDesc->GetMediaDescription("tArduinoData") != NULL);

    // set runlevel application
    SERVICE_ENV_SET_RUNLEVEL(IRuntime::RL_Application);
}

/*
Tear down for this test
*/
void cTesterAADCArduinoComm::tearDown()
{
    // give the kernel some time to quit its threads
    //cSystem::Sleep(200000);
    FILTER_ENV_TEAR_DOWN;
}

/*****
/* This is just a helper class to receive data from output pins.
*/
*****/
class cMediaSampleSink: public IPinEventSink
{
    // ucom helper macro
    UCOM_OBJECT_IMPL(IID_ADTF_PIN_EVENT_SINK, adtf::IPinEventSink);
public:
    // constructor
    cMediaSampleSink(): m_ui32SampleCount(0)
    {
    }
}

```



```

// destructor
virtual ~cMediaSampleSink()
{
}

// members to count and hold the received media samples
tUInt32 m_ui32SampleCount;
std::vector<cObjectPtr<IMediaSample> > m_vecMediaSamples;

public:
    tResult OnPinEvent(IPin* pSource, tInt nEventCode, tInt nParam1, tInt nParam2,
IMediaSample *pMediaSample)
    {
        if (nEventCode == IPinEventSink::PE_MediaSampleTransmitted)
        {
            // count the samples
            ++m_ui32SampleCount;
            // push sample into vector for later compare
            m_vecMediaSamples.push_back(pMediaSample);
        }

        RETURN_NOERROR;
    }
};

/*****
/* This are just helper functions to send Media Samples.
*/
*****/

tTestResult transmitMediaSample(cOutputPin *pSampleSource, tUInt8 ui8Id, tUInt8
ui8DataLength, const tUInt8 *ui8Data)
{
    // get the mediatype of the sample source
    cObjectPtr<IMediaType> pType;
    if(IS_FAILED(pSampleSource->GetMediaType(&pType)))
    {
        __adtf_test_ext(tFalse, "unable to get mediatype");
    }

    // get the type description
    cObjectPtr<IMediaTypeDescription> pTypeDesc;
    pType->GetInterface(IID_ADTF_MEDIA_TYPE_DESCRIPTION, (tVoid**)&pTypeDesc);

    // get the serializer from description to get the deserialized size
    cObjectPtr<IMediaSerializer> pSerializer;
    pTypeDesc->GetMediaSampleSerializer(&pSerializer);
    tInt nSize = pSerializer->GetDeserializedSize();
    pSerializer = NULL;

    // init the values of the protocol
    tUInt8 ui8SOF = ID_ARD_SOF;           // Set the start of frame
    tUInt32 ui32ArduinoTimestamp = 0;     // Set the Timestamp

    // create and allocate the sample
    cObjectPtr<IMediaSample> pSample;
    _runtime->CreateInstance(OID_ADTF_MEDIA_SAMPLE, IID_ADTF_MEDIA_SAMPLE,
(tVoid**)&pSample);
    pSample->AllocBuffer(nSize);

```

```

// get the coder
cObjectPtr<IMediaCoder> pCoder;
pTypeDesc->WriteLock(pSample, &pCoder);

// use the coder to set the value
pCoder->Set("ui8SOF", (tVoid*)&ui8SOF);
pCoder->Set("ui8ID", (tVoid*)&ui8Id);
pCoder->Set("ui32ArduinoTimestamp", (tVoid*)&(ui32ArduinoTimestamp));
pCoder->Set("ui8DataLength", (tVoid*)&ui8DataLength);
pCoder->Set("ui8Data", (tVoid*)&ui8Data);

// unlock the coder
pTypeDesc->Unlock(pCoder);
// set the sample time (in this case the time doesn't matter and must not be the
stream time)
pSample->SetTime(0);
// transmit the media sample
//__adtf_test_result(pSampleSource->Transmit(pSample));
if(IS_FAILED(pSampleSource->Transmit(pSample)))
{
    __adtf_test_ext(tFalse, "unable to send media sample");
}
}

tTestResult transmitDriveCommand(cOutputPin *pSampleSource, tFloat32 f32Value, const
tUInt8 chFrameId)
{
    // correct rounding of the value // Kommentar
    tUInt8 ui8Data = static_cast<tUInt8> (f32Value + 0.5f);
    // sizeof
    return transmitMediaSample(pSampleSource, chFrameId, 1, &ui8Data);
}

tTestResult transmitLightSignal(cOutputPin *pSampleSource, tUInt8 *ui8Data, const
tUInt8 chFrameId)
{
    return transmitMediaSample(pSampleSource, chFrameId, 2, ui8Data);
}

tTestResult transmitWatchdog(cOutputPin *pSampleSource)
{
    tUInt8 ui8Data = static_cast<tUInt8> (true);
    return transmitMediaSample(pSampleSource, ID_ARD_ACT_WD_TOGGLE, 1, &ui8Data);
}

tTestResult DoCommunicationFilterTest(const tUInt8 chFrameId, tInt nSleep = 0)
{
    // initialize filter
    INIT_FILTER(pFilter, pFilterConfig, "adtf.aadc.arduinoCOM");

#ifdef WIN32 // WIN32 is also defined on WIN64
    // set the COM-Port on the Windows machine (on ODROID we use the default setting
from filter)
    __adtf_test_result(pFilterConfig->SetPropertyStr("COM Port", " \\.\\COM9"));
#endif

    // set filter to state ready
    SET_STATE_READY(pFilter);

    // get the input pin of the Arduino Communications Filter
    cObjectPtr<IPin> pComInputPin;

```

```

__adtf_test_result_ext(pFilter->FindPin("COM_input", IPin::PD_Input,
&pComInputPin), "Unable to find input pin COM_input on Arduino Communications
Filter");

// get the output pin of the Arduino Communications Filter
cObjectPtr<IPin> pComOutputPin;
__adtf_test_result_ext(pFilter->FindPin("COM_output", IPin::PD_Output,
&pComOutputPin), "Unable to find pin COM_output on Arduino Communications Filter");

// register the sample sink to receive the data from the output pin of the Arduino
Communication Filter
cObjectPtr<cMediaSampleSink> pSampleSink = new cMediaSampleSink();
__adtf_test_result(pComOutputPin->RegisterEventSink(pSampleSink));

// create a simple sample source
cObjectPtr<cOutputPin> pSampleSource = new cOutputPin();
__adtf_test_result(pSampleSource->Create("sampleSource", new
cMediaType(0,0,0,"tArduinoData")));

// connect the Input pin of the communication filter with the sample source
__adtf_test_result(pComInputPin->Connect(pSampleSource));

// set the filter state running
SET_STATE_RUNNING(pFilter);

// init the loop count
tUInt32 ui32MaxLoopCount = 0;
// init the wait time
tUInt32 ui32Waittime = 0;

#####
#####
//Testing

#####
#####

// define the Test duration and the Waittime after Watchdog Signal is on
ui32MaxLoopCount = 10000;
// Dafines the Time from when the Watchdog is active to when the first actuator
sample is sent. This is because of the initialisation of the motor controler
ui32Waittime = 10000;

#ifdef WIN32 // WIN32 is also defined on WIN64
// Because of the different execution time on windows and linux there have to be
different loop cycle numbers to simulate the same time
ui32MaxLoopCount = 1000;
ui32Waittime = 1000;
#endif

// init the data range
tFloat32 f32LowerRange = 0.0f;
tFloat32 f32UpperRange = 0.0f;

// define the data range
if(chFrameId == ID_ARD_ACT_STEER_ANGLE || chFrameId == 0xff)
{
    f32LowerRange = 65.0f;
    f32UpperRange = 125.0f;

}else if (chFrameId == ID_ARD_ACT_ACCEL_SERVO){

```

```

    f32LowerRange = 0.0f;
    f32UpperRange = 180.0f;
}

// define the LightCycle
// Number of lights
tFloat32 f32NumberOfLights = 6;
// Loopcount when the lights are activated
tUInt32 ui32NumberOfCycles = ui32MaxLoopCount/4;

// calculate the Start of each Lighttest
tFloat32 f32StartOfHeadlight = ui32NumberOfCycles / f32NumberOfLights * 0;
tFloat32 f32StartOfBacklight = ui32NumberOfCycles / f32NumberOfLights * 1;
tFloat32 f32StartOfBrakelight = ui32NumberOfCycles / f32NumberOfLights * 2;
tFloat32 f32StartOfTurnleftlight = ui32NumberOfCycles / f32NumberOfLights * 3;
tFloat32 f32StartOfTurnrightlight = ui32NumberOfCycles / f32NumberOfLights * 4;
tFloat32 f32StartOfReverselight = ui32NumberOfCycles / f32NumberOfLights * 5;

#####
// send some samples
#####
// to give the motor controller time for initialisation send for a short time only
wd signal
for (tUInt32 nIdx = 0; nIdx < ui32Waittime; nIdx++)
{
    // Always send Watchdog Signal
    transmitWatchdog(pSampleSource);
    cSystem::Sleep(nSleep);
}
#####
// define a Ramp from one border to init Value
if(chFrameId == ID_ARD_ACT_STEER_ANGLE || chFrameId == 0xff)
{
    f32LowerRange = 95.0f;
    f32UpperRange = 125.0f;

}else if (chFrameId == ID_ARD_ACT_ACCEL_SERVO){
    f32LowerRange = 90.0f;
    f32UpperRange = 180.0f;
}
#####
// send the samples according to the specified ramp
for (tUInt32 nIdx = 0; nIdx < ui32MaxLoopCount/4; nIdx++)
{
    // Always send Watchdog Signal
    transmitWatchdog(pSampleSource);
    cSystem::Sleep(nSleep);

    // Acceleration and steer angle test
    if(chFrameId==ID_ARD_ACT_ACCEL_SERVO || chFrameId ==ID_ARD_ACT_STEER_ANGLE ||
chFrameId == 0xff)
    {
        // set the data of the protocol
        // Calculate the value by dividing the range by the loop count times the
loop counter
        // through this every Value will be tested
        tFloat32 f32Value = (f32UpperRange - f32LowerRange) / (ui32MaxLoopCount/4)
* nIdx + f32LowerRange;

        if(chFrameId == 0xff)

```

```

    {
        transmitDriveCommand(pSampleSource, f32Value, ID_ARD_ACT_ACCEL_SERVO);
        cSystem::Sleep(nSleep);
        transmitDriveCommand(pSampleSource, f32Value, ID_ARD_ACT_STEER_ANGLE);
        cSystem::Sleep(nSleep);
    }
    else
    {
        transmitDriveCommand(pSampleSource, f32Value, chFrameId);
        cSystem::Sleep(nSleep);
    }
}
// Light Test
if(chFrameId == ID_ARD_ACT_LIGHT_DATA || chFrameId == 0xff)
{
    // set the data of the protocol
    tBool bValue = tTrue;
    tUInt8 au8Data[2] = {0,0};

    // Divide the loop into equal sections for every Light
    if(nIdx > (f32StartOfHeadlight) && nIdx < (f32StartOfBacklight) )
    au8Data[0] = ID_ARD_ACT_LIGHT_DATA_HEAD;
    if(nIdx > (f32StartOfBacklight) && nIdx < (f32StartOfBrakelight) )
    au8Data[0] = ID_ARD_ACT_LIGHT_DATA_BACK;
    if(nIdx > (f32StartOfBrakelight) && nIdx < (f32StartOfTurnleftlight) )
    au8Data[0] = ID_ARD_ACT_LIGHT_DATA_BRAKE;
    if(nIdx > (f32StartOfTurnleftlight) && nIdx < (f32StartOfTurnrightlight) )
    au8Data[0] = ID_ARD_ACT_LIGHT_DATA_TURNLEFT;
    if(nIdx > (f32StartOfTurnrightlight) && nIdx < (f32StartOfReverselight) )
    au8Data[0] = ID_ARD_ACT_LIGHT_DATA_TURNRIGHT;
    if(nIdx > (f32StartOfReverselight) && nIdx < ui32MaxLoopCount) au8Data[0]
    = ID_ARD_ACT_LIGHT_DATA_REVERSE;

    // Set the Lights to on
    bValue == tTrue ? au8Data[1] = 1: au8Data[1] = 0;

    transmitLightSignal(pSampleSource, au8Data, ID_ARD_ACT_LIGHT_DATA);
    cSystem::Sleep(nSleep);
}
}
//#####
// Go the same ramp backwards
//#####
// send the samples according to the specified ramp
for (tUInt32 nIdx = ui32MaxLoopCount/4; nIdx > 0 ; nIdx--)
{
    // Always send Watchdog Signal
    transmitWatchdog(pSampleSource);
    cSystem::Sleep(nSleep);

    // Acceleration and steer angle test
    if(chFrameId==ID_ARD_ACT_ACCEL_SERVO || chFrameId ==ID_ARD_ACT_STEER_ANGLE ||
chFrameId == 0xff)
    {
        // set the data of the protocol
        // Calculate the value by dividing the range by the loop count times the
loop counter
        // through this every Value will be tested
        tFloat32 f32Value = (f32UpperRange - f32LowerRange) / (ui32MaxLoopCount/4)
* nIdx + f32LowerRange;

        if(chFrameId == 0xff)

```

```

    {
        transmitDriveCommand(pSampleSource, f32Value, ID_ARD_ACT_ACCEL_SERVO);
        cSystem::Sleep(nSleep);
        transmitDriveCommand(pSampleSource, f32Value, ID_ARD_ACT_STEER_ANGLE);
        cSystem::Sleep(nSleep);
    }
    else
    {
        transmitDriveCommand(pSampleSource, f32Value, chFrameId);
        cSystem::Sleep(nSleep);
    }
}
// Light Test
if(chFrameId == ID_ARD_ACT_LIGHT_DATA || chFrameId == 0xff)
{
    // set the data of the protocol
    tBool bValue = tFalse;
    tUInt8 au8Data[2] = {0,0};

    // Divide the loop into equal sections for every Light
    if(nIdx > (f32StartOfHeadlight) && nIdx < (f32StartOfBacklight) )
        au8Data[0] = ID_ARD_ACT_LIGHT_DATA_HEAD;
    if(nIdx > (f32StartOfBacklight) && nIdx < (f32StartOfBrakelight) )
        au8Data[0] = ID_ARD_ACT_LIGHT_DATA_BACK;
    if(nIdx > (f32StartOfBrakelight) && nIdx < (f32StartOfTurnleftlight) )
        au8Data[0] = ID_ARD_ACT_LIGHT_DATA_BRAKE;
    if(nIdx > (f32StartOfTurnleftlight) && nIdx < (f32StartOfTurnrightlight) )
        au8Data[0] = ID_ARD_ACT_LIGHT_DATA_TURNLEFT;
    if(nIdx > (f32StartOfTurnrightlight) && nIdx < (f32StartOfReverselight) )
        au8Data[0] = ID_ARD_ACT_LIGHT_DATA_TURNRIGHT;
    if(nIdx > (f32StartOfReverselight) && nIdx < ui32MaxLoopCount) au8Data[0]
        = ID_ARD_ACT_LIGHT_DATA_REVERSE;

    // Set the Lights to on
    bValue == tTrue ? au8Data[1] = 1: au8Data[1] = 0;

    transmitLightSignal(pSampleSource, au8Data, ID_ARD_ACT_LIGHT_DATA);
    cSystem::Sleep(nSleep);
}
}
//#####
// define a Ramp from init Value to the other border
if(chFrameId == ID_ARD_ACT_STEER_ANGLE || chFrameId == 0xff)
{
    f32LowerRange = 65.0f;
    f32UpperRange = 95.0f;

} else if (chFrameId == ID_ARD_ACT_ACCEL_SERVO){
    f32LowerRange = 0.0f;
    f32UpperRange = 90.0f;
}
//#####
// send the samples according to the specified ramp
for (tUInt32 nIdx = ui32MaxLoopCount/4; nIdx > 0 ; nIdx--)
{
    // Always send Watchdog Signal
    transmitWatchdog(pSampleSource);
    cSystem::Sleep(nSleep);

    // Acceleration and steer angle test
    if(chFrameId==ID_ARD_ACT_ACCEL_SERVO || chFrameId ==ID_ARD_ACT_STEER_ANGLE ||
chFrameId == 0xff)

```

```

    {
        // set the data of the protocol
        // Calculate the value by dividing the range by the loop count times the
loop counter
        // through this every Value will be tested
        tFloat32 f32Value = (f32UpperRange - f32LowerRange) / (ui32MaxLoopCount/4)
* nIdx + f32LowerRange;

        if(chFrameId == 0xff)
        {
            transmitDriveCommand(pSampleSource, f32Value, ID_ARD_ACT_ACCEL_SERVO);
            cSystem::Sleep(nSleep);
            transmitDriveCommand(pSampleSource, f32Value, ID_ARD_ACT_STEER_ANGLE);
            cSystem::Sleep(nSleep);
        }
        else
        {
            transmitDriveCommand(pSampleSource, f32Value, chFrameId);
            cSystem::Sleep(nSleep);
        }
    }
    // Light Test
    if(chFrameId == ID_ARD_ACT_LIGHT_DATA || chFrameId == 0xff)
    {
        // set the data of the protocol
        tBool bValue = tTrue;
        tUInt8 au8Data[2] = {0,0};

        // Divide the loop into equal sections for every Light
        if(nIdx > (f32StartOfHeadlight) && nIdx < (f32StartOfBacklight) )
au8Data[0] = ID_ARD_ACT_LIGHT_DATA_HEAD;
        if(nIdx > (f32StartOfBacklight) && nIdx < (f32StartOfBrakelight) )
au8Data[0] = ID_ARD_ACT_LIGHT_DATA_BACK;
        if(nIdx > (f32StartOfBrakelight) && nIdx < (f32StartOfTurnleftlight) )
au8Data[0] = ID_ARD_ACT_LIGHT_DATA_BRAKE;
        if(nIdx > (f32StartOfTurnleftlight) && nIdx < (f32StartOfTurnrightlight) )
au8Data[0] = ID_ARD_ACT_LIGHT_DATA_TURNLEFT;
        if(nIdx > (f32StartOfTurnrightlight) && nIdx < (f32StartOfReverselight) )
au8Data[0] = ID_ARD_ACT_LIGHT_DATA_TURNRIGHT;
        if(nIdx > (f32StartOfReverselight) && nIdx < ui32MaxLoopCount) au8Data[0]
= ID_ARD_ACT_LIGHT_DATA_REVERSE;

        // Set the Lights to on
        bValue == tTrue ? au8Data[1] = 1: au8Data[1] = 0;

        transmitLightSignal(pSampleSource, au8Data, ID_ARD_ACT_LIGHT_DATA);
        cSystem::Sleep(nSleep);
    }
}

#####
// Go the same ramp backwards
#####
// send the samples according to the specified ramp
for (tUInt32 nIdx = 0; nIdx < ui32MaxLoopCount/4; nIdx++)
{
    // Always send Watchdog Signal
    transmitWatchdog(pSampleSource);
    cSystem::Sleep(nSleep);

    // Acceleration and steer angle test

```

```

    if(chFrameId==ID_ARD_ACT_ACCEL_SERVO || chFrameId ==ID_ARD_ACT_STEER_ANGLE ||
chFrameId == 0xff)
    {
        // set the data of the protocol
        // Calculate the value by dividing the range by the loop count times the
loop counter
        // through this every Value will be tested
        tFloat32 f32Value = (f32UpperRange - f32LowerRange) / (ui32MaxLoopCount/4)
* nIdx + f32LowerRange;

        if(chFrameId == 0xff)
        {
            transmitDriveCommand(pSampleSource, f32Value,ID_ARD_ACT_ACCEL_SERVO);
            cSystem::Sleep(nSleep);
            transmitDriveCommand(pSampleSource, f32Value,ID_ARD_ACT_STEER_ANGLE);
            cSystem::Sleep(nSleep);
        }
        else
        {
            transmitDriveCommand(pSampleSource, f32Value,chFrameId);
            cSystem::Sleep(nSleep);
        }
    }
    // Light Test
    if(chFrameId == ID_ARD_ACT_LIGHT_DATA || chFrameId == 0xff)
    {
        // set the data of the protocol
        tBool bValue = tFalse;
        tUInt8 auI8Data[2] = {0,0};

        // Divide the loop into equal sections for every Light
        if(nIdx > (f32StartOfHeadlight) && nIdx < (f32StartOfBacklight) )
auI8Data[0] = ID_ARD_ACT_LIGHT_DATA_HEAD;
        if(nIdx > (f32StartOfBacklight) && nIdx < (f32StartOfBrakelight) )
auI8Data[0] = ID_ARD_ACT_LIGHT_DATA_BACK;
        if(nIdx > (f32StartOfBrakelight) && nIdx < (f32StartOfTurnleftlight) )
auI8Data[0] = ID_ARD_ACT_LIGHT_DATA_BRAKE;
        if(nIdx > (f32StartOfTurnleftlight) && nIdx < (f32StartOfTurnrightlight) )
auI8Data[0] = ID_ARD_ACT_LIGHT_DATA_TURNLEFT;
        if(nIdx > (f32StartOfTurnrightlight) && nIdx < (f32StartOfReverselight) )
auI8Data[0] = ID_ARD_ACT_LIGHT_DATA_TURNRIGHT;
        if(nIdx > (f32StartOfReverselight) && nIdx < ui32MaxLoopCount) auI8Data[0]
= ID_ARD_ACT_LIGHT_DATA_REVERSE;

        // Set the Lights to on
        bValue == tTrue ? auI8Data[1] = 1: auI8Data[1] = 0;

        transmitLightSignal(pSampleSource, auI8Data, ID_ARD_ACT_LIGHT_DATA);
        cSystem::Sleep(nSleep);
    }
}
//#####

//#####
#####

// clean up
__adtf_test_result(pComInputPin->Disconnect(pSampleSource));
__adtf_test_result(pComOutputPin->UnregisterEventSink(pSampleSink));

```



```

#####
#####
// Start of evaluation

#####
#####

// get description manager
cObjectPtr<IMediaDescriptionManager> pDescManager;
__adtf_test_result(_runtime-
>GetObject(OID_ADTF_MEDIA_DESCRIPTION_MANAGER, IID_ADTF_MEDIA_DESCRIPTION_MANAGER, (tVoid**)
&pDescManager));

// get media description for tArduinoData
tChar const * strDescArduino = pDescManager->GetMediaDescription("tArduinoData");
__adtf_test_pointer(strDescArduino);

// create mediatype for coder
cObjectPtr<IMediaType> pTypeArduinoData = new cMediaType(0, 0, 0, "tArduinoData",
strDescArduino, IMediaDescription::MDF_DDL_DEFAULT_VERSION);

// get the mediatype description from the mediatype
cObjectPtr<IMediaTypeDescription> m_pCoderDescArduinoData;
__adtf_test_result(pTypeArduinoData->GetInterface(IID_ADTF_MEDIA_TYPE_DESCRIPTION,
(tVoid**) &m_pCoderDescArduinoData));

// map with channel/frameid as key and vector of mediasamples as value to sort the
different kinds of sensors
//cSampleMap mapSensors;
std::map<tChar, std::vector<tArduinoFrame> > mapSensors;

// build the map by iterating over all received mediasamples
for(tUInt32 nIdx = 0; nIdx < pSampleSink->m_vecMediaSamples.size(); ++nIdx)
{
    // get the coder
    cObjectPtr<IMediaCoder> pCoder;
    m_pCoderDescArduinoData->Lock(pSampleSink->m_vecMediaSamples[nIdx], &pCoder);

    // init the values
    tUInt8 ui8SOF = 0;
    tUInt8 ui8Id = 0;
    tUInt32 arduinoTimestamp = 0;
    tUInt8 ui8DataLength = 0;
    tUInt8 frameData[25];

    // get values from coder
    pCoder->Get("ui8ID", (tVoid*)&ui8Id);
    pCoder->Get("ui8SOF", (tVoid*)&ui8SOF);
    pCoder->Get("ui32ArduinoTimestamp", (tVoid*)&arduinoTimestamp);
    pCoder->Get("ui8DataLength", (tVoid*)&ui8DataLength);
    pCoder->Get("ui8Data", (tVoid*)&(frameData));

    m_pCoderDescArduinoData->Unlock(pCoder);

    // check for plausible datalength
    ui8DataLength = ui8DataLength > sizeof(tArduinoDataUnion) ?
sizeof(tArduinoDataUnion) : ui8DataLength;

    // create a arduinoFrame
    tArduinoFrame sFrame;
    cMemoryBlock::MemSet(&sFrame, 0, sizeof(tArduinoFrame));

```

```

    // fill the arduino frame
    sFrame.sHeader.ui8ID = ui8Id;
    sFrame.sHeader.ui8SOF = ui8SOF;
    sFrame.sHeader.ui32ArduinoTimestamp = arduinoTimestamp;
    sFrame.sHeader.ui8DataLength = ui8DataLength;
    cMemoryBlock::MemCopy(&sFrame.sData, &frameData, ui8DataLength);

    mapSensors[ui8Id].push_back(sFrame);
}

// clear the sample sink
pSampleSink->m_vecMediaSamples.clear();
// Shutdown the Filter
SET_STATE_SHUTDOWN(pFilter);

tUInt32 ui32test=mapSensors[ID_ARD_SENS_STEER_ANGLE].size();
// Do a quick test to see if enough samples are received
__adtf_test_ext(mapSensors[ID_ARD_SENS_STEER_ANGLE].size() >= 100,
cString::Format("does not receive enough samples for Steerangle Sensor. Number of
samples: %d", mapSensors[ID_ARD_SENS_STEER_ANGLE].size()));
__adtf_test_ext(mapSensors[ID_ARD_SENS_WHEELENC].size() >= 100,
cString::Format("does not receive enough samples for RPM Sensor. Number of samples:
%d", mapSensors[ID_ARD_SENS_WHEELENC].size()));
__adtf_test_ext(mapSensors[ID_ARD_SENS_IMU].size() >= 100, cString::Format("does
not receive enough samples for Motion Sensor. Number of samples: %d",
mapSensors[ID_ARD_SENS_IMU].size()));
__adtf_test_ext(mapSensors[ID_ARD_SENS_IR].size() >= 100, cString::Format("does
not receive enough samples for Infrared Sensor. Number of samples: %d",
mapSensors[ID_ARD_SENS_IR].size()));
__adtf_test_ext(mapSensors[ID_ARD_SENS_PHOTO].size() >= 100, cString::Format("does
not receive enough samples for Photo Sensor. Number of samples: %d",
mapSensors[ID_ARD_SENS_PHOTO].size()));
__adtf_test_ext(mapSensors[ID_ARD_SENS_US].size() >= 50, cString::Format("does not
receive enough samples for Ultra Sonic Sensor. Number of samples: %d",
mapSensors[ID_ARD_SENS_US].size()));
__adtf_test_ext(mapSensors[ID_ARD_SENS_VOLTAGE].size() >= 5, cString::Format("does
not receive enough samples for Voltage Sensor. Number of samples: %d",
mapSensors[ID_ARD_SENS_VOLTAGE].size()));

// if the sleep time is long enough do a deeper check for the ramps
if(nSleep > 10 && (chFrameId == ID_ARD_SENS_WHEELENC || chFrameId == 0xff))
{
    // received sensor data should show the actor ramp
    // now check the received data

    // Define the Ranges of the ramp and a tolerance Value to ignore single value
jumps
    tUInt16 ui16LowerRampRange = 1;
    tUInt16 ui16UpperRampRange = 10;
    tUInt16 ui16ToleranceValue = 4;

    // Init some values we need to recognize the ramp
    tUInt32 ui32RPMDiffBefore = 0;
    tUInt32 ui32RPMBefore = 0;
    tBool bStartOfRamp = tFalse;
    tInt8 i8RampSegment = 0;
    tUInt16 ui16StartRPMDiff = 0;

    // loop over the received data
    // rpm evaluation
    for(tSize nIdx = 0; nIdx < mapSensors[ID_ARD_SENS_WHEELENC].size(); ++nIdx)

```

```

{
    // only take every third value to receive stronger differences
    if(nIdx % 3)
    {
        // test the received data
        // SOF
        __adtf_test(mapSensors[ID_ARD_SENS_WHEELENC][nIdx].sHeader.ui8SOF ==
ID_ARD_SOF);
        // Frame ID
        __adtf_test(mapSensors[ID_ARD_SENS_WHEELENC][nIdx].sHeader.ui8ID ==
ID_ARD_SENS_WHEELENC);
        // Check for the right datalength

        __adtf_test(mapSensors[ID_ARD_SENS_WHEELENC][nIdx].sHeader.ui8DataLength == 8);

        // Extract the Steerangle Data
        tUInt32 ui32RPM =
mapSensors[ID_ARD_SENS_WHEELENC][nIdx].sData.sWheelEncData.ui32LeftWheel;
        // weigh the new value statistically with the old one to filter out
        peak values
        ui32RPM = (ui32RPMBefore + ui32RPM)/2;

        __adtf_test_log(cString::Format("RPM: %d", ui32RPM - ui32RPMBefore));
        __adtf_test_log(cString::Format("Rampsegment: %d", i8RampSegment));

        // Test for the ramp
        // recognize the start of the ramp
        if(!bStartOfRamp && ui32RPM - ui32RPMBefore > 0 && i8RampSegment == 0)
        {
            ui16StartRPMDiff = 0;
            bStartOfRamp = true;
            i8RampSegment++;
        }
        // recognize the first turn
        if(bStartOfRamp && ui32RPM - ui32RPMBefore > ui16UpperRampRange &&
i8RampSegment == 1)
        {
            i8RampSegment++;
            bStartOfRamp = false;
        }
        // recognize the second turn
        if(bStartOfRamp && ui32RPM - ui32RPMBefore < ui16LowerRampRange &&
i8RampSegment == 2)
        {
            i8RampSegment++;
            bStartOfRamp = false;
        }
        // recognize the end of the ramp
        if(bStartOfRamp && ui32RPM - ui32RPMBefore == ui16StartRPMDiff &&
i8RampSegment == 3)
        {
            i8RampSegment++;
            bStartOfRamp = false;
        }
        // start the ramp recognition again after a turn if the values are in
        the range
        if(!bStartOfRamp && ui32RPM - ui32RPMBefore < ui16UpperRampRange &&
ui32RPM - ui32RPMBefore > ui16LowerRampRange && i8RampSegment < 4)
        {
            bStartOfRamp = true;
        }
    }
}

```

```

        // test the values in the rising ramps
        if(bStartOfRamp && (i8RampSegment == 1 || i8RampSegment == 3))
        {
            __adtf_test(ui32RPM - ui32RPMBefore + ui16ToleranceValue >
ui32RPMDiffBefore);
        }
        // test the values of the falling ramp
        if(bStartOfRamp && i8RampSegment == 2)
        {
            __adtf_test(ui32RPM - ui32RPMBefore < ui32RPMDiffBefore +
ui16ToleranceValue);
        }

        // Go to the next Values
        ui32RPMDiffBefore = ui32RPM - ui32RPMBefore;
        ui32RPMBefore = ui32RPM;

    }
}
// Check if the ramp was recognized
__adtf_test(i8RampSegment == 4);
}
//#####
//Steerangle evaluation
if(nSleep > 10 && (chFrameId == ID_ARD_SENS_STEER_ANGLE || chFrameId == 0xff))
{
    // received sensor data should show the actor ramp
    // 0xff bei der Auswertung nicht vergessen
    // now check the received data

    // Define the Ranges of the ramp and a tolerance Value to ignore single value
jumps
    tUInt16 ui16LowerRampRange = 338;
    tUInt16 ui16UpperRampRange = 520;
    tUInt16 ui16ToleranceValue = 20;

    // Init some values we need to recognize the ramp
    tUInt16 ui16SteerAngleBefore = 0;
    tBool bStartOfRamp = tFalse;
    tInt8 i8RampSegment = 10;
    tUInt16 ui16StartAngle = 0;

    // loop over the received data
    for(tSize nIdx = 0; nIdx < mapSensors[ID_ARD_SENS_STEER_ANGLE].size(); ++nIdx)
    {
        // test the received data
        // SOF
        __adtf_test(mapSensors[ID_ARD_SENS_STEER_ANGLE][nIdx].sHeader.ui8SOF ==
ID_ARD_SOF);
        // Frame ID
        __adtf_test(mapSensors[ID_ARD_SENS_STEER_ANGLE][nIdx].sHeader.ui8ID ==
ID_ARD_SENS_STEER_ANGLE);
        // Check for the right datalength

        __adtf_test(mapSensors[ID_ARD_SENS_STEER_ANGLE][nIdx].sHeader.ui8DataLength == 2);

        // Extract the Steerangle Data
        tUInt16 ui16Angle =
mapSensors[ID_ARD_SENS_STEER_ANGLE][nIdx].sData.sSteeringData.ui16Angle;

        __adtf_test_log(cString::Format("Steerangle: %d", ui16Angle));
        __adtf_test_log(cString::Format("Rampsegment: %d", i8RampSegment));
    }
}

```

```

    // Test for the ramp
    // recognize initial state
    if(ui16Angle == 0)
    {
        i8RampSegment = 0;
    }
    // recognize the start of the ramp
    if(!bStartOfRamp && ui16Angle > 0 && i8RampSegment == 0)
    {
        ui16StartAngle = ui16Angle;
        bStartOfRamp = tTrue;
        i8RampSegment++;
    }
    // recognize the first turn
    if(bStartOfRamp && ui16Angle > ui16UpperRampRange && i8RampSegment == 1)
    {
        i8RampSegment++;
        bStartOfRamp = tFalse;
    }
    // recognize the second turn
    if(bStartOfRamp && ui16Angle < ui16LowerRampRange && i8RampSegment == 2)
    {
        i8RampSegment++;
        bStartOfRamp = tFalse;
    }
    // recognize the end of the ramp
    if(bStartOfRamp && ui16Angle > ui16StartAngle - ui16ToleranceValue &&
i8RampSegment == 3)
    {
        i8RampSegment++;
        bStartOfRamp = tFalse;
    }
    // start the ramp recognition again after a turn if the values are in the
range
    if(!bStartOfRamp && ui16Angle < ui16UpperRampRange && ui16Angle >
ui16LowerRampRange && i8RampSegment < 4)
    {
        bStartOfRamp = tTrue;
    }

    // test the values in the rising ramps
    if(bStartOfRamp && (i8RampSegment == 1 || i8RampSegment == 3))
    {
        __adtf_test(ui16Angle > ui16SteerAngleBefore - ui16ToleranceValue);
    }
    // test the values of the falling ramp
    if(bStartOfRamp && i8RampSegment == 2)
    {
        __adtf_test(ui16Angle < ui16SteerAngleBefore + ui16ToleranceValue);
    }

    // Go to the next Values
    ui16SteerAngleBefore = ui16Angle;
}
// Check if the ramp was recognized
__adtf_test(i8RampSegment == 4);
}
}

```

```
DEFINE_TEST(cTesterAADCArduinoComm,
            TestAcceleration,
            "2.1",
            "TestAcceleration",
            "This test makes sure, that the output of the communication filter works
while the input\"
            "is set to different acceleration values",
            "The communication filter sends allways data",
            "See above",
            "none",
            "The filter must be able to receive acceleration data while delivering
sensor data.",
            "automatic")
{
    // accelerate pin with expected frameID
    return DoCommunicationFilterTest(ID_ARD_ACT_ACCEL_SERVO, 20);
}
```

```
DEFINE_TEST(cTesterAADCArduinoComm,
            TestSteeringAngle,
            "2.2",
            "TestSteeringAngle",
            "This test makes sure, that the output of the communication filter works
while the input\"
            "is set to different steering angle values",
            "The communication filter sends allways data",
            "See above",
            "none",
            "The filter must be able to receive steering data while delivering sensor
data.",
            "automatic")
{
    // steering angle pin with expected frameID
    return DoCommunicationFilterTest(ID_ARD_ACT_STEER_ANGLE, 20);
}
```

```
DEFINE_TEST(cTesterAADCArduinoComm,
            TestLights,
            "2.2",
            "TestLights",
            "This test makes sure, that the output of the communication filter works
while the input\"
            "is set to different steering angle values",
            "The communication filter sends allways data",
            "See above",
            "none",
            "The filter must be able to receive steering data while delivering sensor
data.",
            "automatic")
{
    // steering angle pin with expected frameID
    return DoCommunicationFilterTest(ID_ARD_ACT_LIGHT_DATA, 10);
}
```

```

DEFINE_TEST(cTesterAADCArduinoComm,
            TestAllActors,
            "2.3",
            "TestAllActors",
            "...\"
            \"is set to different steering angle values\",
            \"The communication filter sends allways data\",
            \"See above\",
            \"none\",
            \"The filter must be able to receive steering data while delivering sensor
data.\",
            \"automatic\")
{
    // steering angle pin with expected frameID
    return DoCommunicationFilterTest(0xff, 20);
}

```

Programm-Code: Arduino Sensors Filter (Modul)

```

/**
 *
 * AADC Arduino sensors filter tests
 *
 * @file
 * Copyright &copy; Audi Electronics Venture GmbH. All rights reserved
 *
 * $Author: VG8D3AW $
 * $Date: 2013-02-06 16:30:41 +0100 (Mi, 06. Feb 2013) $
 * $Revision: 18162 $
 *
 * @remarks
 *
 */

#include "stdafx.h"
#include "tester_aadc_sensors.h"

IMPLEMENT_TESTER_CLASS(cTesterAADCsensors,
                      "1",
                      "AADC Arduino sensors filter",
                      "This test makes sure that the AADC Arduino sensors filter works as
expected",
                      "");

/**
Setup for this test
*/
void cTesterAADCsensors::setUp()
{
    // create the environment
    SERVICE_ENV_SETUP;

    // minimum needed ADTF services
    SERVICE_ENV_ADD_PLUGIN("adtf_clock.srv");
    SERVICE_ENV_ADD_PLUGIN("adtf_kernel.srv");
    SERVICE_ENV_ADD_PLUGIN("adtf_sample_pool.srv");
    SERVICE_ENV_ADD_PLUGIN("adtf_namespace.srv");
    SERVICE_ENV_ADD_PLUGIN("adtf_media_description.srv");

    // register services

```



```

    SERVICE_ENV_REGISTER_SERVICE(OID_ADTF_REFERENCE_CLOCK, "referenceclock",
IRuntime::RL_Kernel);
    SERVICE_ENV_REGISTER_SERVICE(OID_ADTF_KERNEL, "kernel", IRuntime::RL_System);
    SERVICE_ENV_REGISTER_SERVICE(OID_ADTF_SAMPLE_POOL, "samplepool",
IRuntime::RL_System);
    SERVICE_ENV_REGISTER_SERVICE(OID_ADTF_NAMESPACE, "namespace",
IRuntime::RL_System);
    SERVICE_ENV_REGISTER_SERVICE(OID_ADTF_MEDIA_DESCRIPTION_MANAGER, "mediadesc",
IRuntime::RL_System);

    // add filter plugin
    FILTER_ENV_ADD_PLUGIN("aadc_sensors.plb");

    // set runlevel system
    SERVICE_ENV_SET_RUNLEVEL(IRuntime::RL_System);

    // set the path to the MediaDescription-Files
    // (relative to the current working dir which is the tester dir)
    cFilename strDescrFileADTF =
"..../src/adtfBase/AADC_ADTF_BaseFilters/description/aadc.description";
    cFilename strDescrFile =
"..../src/adtfBase/AADC_ADTF_BaseFilters/description/aadc.description";

    // create absolute path
    strDescrFileADTF =
strDescrFileADTF.CreateAbsolutePath(cFileSystem::GetCurDirectory());
    strDescrFile = strDescrFile.CreateAbsolutePath(cFileSystem::GetCurDirectory());

    // get the media description manager
    cObjectPtr<IMediaDescriptionManager> pMediaDesc;
    __adtf_test_result_ext(_runtime->GetObject(OID_ADTF_MEDIA_DESCRIPTION_MANAGER,
IID_ADTF_MEDIA_DESCRIPTION_MANAGER, (tVoid**) &pMediaDesc),
    "unable to get media description manager");

    // get the config from the media description manager
    cObjectPtr<IConfiguration> pConfig;
    __adtf_test_result(pMediaDesc->GetInterface(IID_ADTF_CONFIGURATION, (tVoid**)
&pConfig));

    // set the path to the description files
    __adtf_test_result(pConfig->SetPropertyStr("media_description_files",
cString::Format("%s;%s", strDescrFileADTF.GetPtr(), strDescrFile.GetPtr())));

    // check if the media description manager loads the description correctly
    __adtf_test(pMediaDesc->GetMediaDescription("tArduinoData") != NULL);

    // set runlevel application
    SERVICE_ENV_SET_RUNLEVEL(IRuntime::RL_Application);
}

/*
Tear down for this test
*/
void cTesterAADCsensors::tearDown()
{
    // give the kernel some time to quit its threads
    //cSystem::Sleep(200000);
    FILTER_ENV_TEAR_DOWN;
}

/*****

```

```

/* This is just a helper class to receive data from output pins.
*/
/*****
class cMediaSampleSink: public IPinEventSink
{
    // ucom helper macro
    UCOM_OBJECT_IMPL(IID_ADTF_PIN_EVENT_SINK, adtf::IPinEventSink);
public:
    // constructor
    cMediaSampleSink(): m_ui32SampleCount(0)
    {

    }

    // destructor
    virtual ~cMediaSampleSink()
    {
        // loop over all mediasamples to unref
        for (tInt nIndex = 0; nIndex < m_vecMediaSamples.size(); nIndex++)
        {
            // after unref the mediasample will destroy itself
            m_vecMediaSamples[nIndex]->Unref();
        }

        // members to count and hold the received media samples
        tUInt32 m_ui32SampleCount;
        std::vector<IMediaSample*> m_vecMediaSamples;
    }

public:
    tResult OnPinEvent(IPin* pSource, tInt nEventCode, tInt nParam1, tInt nParam2,
    IMediaSample* pMediaSample)
    {
        if (nEventCode == IPinEventSink::PE_MediaSampleTransmitted)
        {
            // count the samples
            ++m_ui32SampleCount;
            // make a ref on sample to avoid self destruction
            pMediaSample->Ref();
            // push sample into vector for later compare
            m_vecMediaSamples.push_back(pMediaSample);
        }

        RETURN_NOERROR;
    }
};

/*****
/* This is just a helper function to send data with the right protocoll.
*/
/*****
tTestResult transmitMediaSample(cOutputPin *pSampleSource, const tUInt8 ui8Id, tInt8
ui8DataLength, const tUInt8 *ui8Data)
{
    // get the mediatype of the sample source
    cObjectPtr<IMediaType> pType;
    if(IS_FAILED(pSampleSource->GetMediaType(&pType)))
    {
        __adtf_test_ext(tFalse, "unable to get mediatype");
    }
}

```

```

// get the type description
cObjectPtr<IMediaTypeDescription> pTypeDesc;
pType->GetInterface(IID_ADTF_MEDIA_TYPE_DESCRIPTION, (tVoid**)&pTypeDesc);

// get the serializer from description to get the deserialized size
cObjectPtr<IMediaSerializer> pSerializer;
pTypeDesc->GetMediaSampleSerializer(&pSerializer);
tInt nSize = pSerializer->GetDeserializedSize();
pSerializer = NULL;

// init the values of the protocol
tUInt8 ui8SOF = ID_ARD_SOF;          // Set the start of frame
tUInt32 ui32ArduinoTimestamp = 0;    // Set the Timestamp

// create and allocate the sample
cObjectPtr<IMediaSample> pSample;
_runtime->CreateInstance(OID_ADTF_MEDIA_SAMPLE, IID_ADTF_MEDIA_SAMPLE,
(tVoid**)&pSample);
pSample->AllocBuffer(nSize);

// get the coder
cObjectPtr<IMediaCoder> pCoder;
pTypeDesc->WriteLock(pSample, &pCoder);

// use the coder to set the value
pCoder->Set("ui8SOF", (tVoid*)&ui8SOF);
pCoder->Set("ui8ID", (tVoid*)&ui8Id);
pCoder->Set("ui32ArduinoTimestamp", (tVoid*)&(ui32ArduinoTimestamp));
pCoder->Set("ui8DataLength", (tVoid*)&ui8DataLength);
pCoder->Set("ui8Data", (tVoid*)&ui8Data);

// unlock the coder
pTypeDesc->Unlock(pCoder);
// set the sample time (in this case the time doesn't matter and must not be the
stream time)
pSample->SetTime(0); // cHighResTimer::GetTime();
// transmit the media sample
//__adtf_test_result(pSampleSource->Transmit(pSample));
if(IS_FAILED(pSampleSource->Transmit(pSample)))
{
    __adtf_test_ext(tFalse, "unable to send media sample");
}
}

/*****
/* These are just a helper functions to evaluate the received data
*/
*****/

tTestResult evaluateSteerAngle(cMediaSampleSink *pSampleSink, tUInt32
ui32MaxLoopCount)
{
    // now check the received data
    __adtf_test_ext(pSampleSink->m_ui32SampleCount == ui32MaxLoopCount,
cString::Format("Sample count (%d) does not match (%d)", pSampleSink-
>m_ui32SampleCount, ui32MaxLoopCount));

    // get description manager
cObjectPtr<IMediaDescriptionManager> pDescManager;

```

```

    __adtf_test_result(_runtime-
>GetObject(OID_ADTF_MEDIA_DESCRIPTION_MANAGER, IID_ADTF_MEDIA_DESCRIPTION_MANAGER, (tVoid**)
&pDescManager));

    // get media description for tArduinoData
    tChar const * strDescSteerAngle = pDescManager-
>GetMediaDescription("tSteeringAngleData");
    __adtf_test_pointer(strDescSteerAngle);

    // create mediatype for coder
    cObjectPtr<IMediaType> pTypeSteerAngleData = new cMediaType(0, 0, 0,
    "tSteeringAngleData", strDescSteerAngle, IMediaDescription::MDF_DDL_DEFAULT_VERSION);

    // get the mediatype description from the mediatype
    cObjectPtr<IMediaTypeDescription> m_pCoderDescSteerAngleData;
    __adtf_test_result(pTypeSteerAngleData-
>GetInterface(IID_ADTF_MEDIA_TYPE_DESCRIPTION, (tVoid**) &m_pCoderDescSteerAngleData));

    // loop over the received data
    for(tSize nIndex = 0; nIndex < pSampleSink->m_vecMediaSamples.size(); ++nIndex)
    {
        // get the coder
        cObjectPtr<IMediaCoder> pCoder;
        m_pCoderDescSteerAngleData->Lock(pSampleSink->m_vecMediaSamples[nIndex],
        &pCoder);

        // init the values
        tUInt16 ui16SteerAngle = 0;
        tUInt32 ui32ArduinoTimestamp = 0;

        // get values from coder
        pCoder->Get("ui16Angle", (tVoid*)&ui16SteerAngle);
        pCoder->Get("ui32ArduinoTimestamp", (tVoid*)&ui32ArduinoTimestamp);

        // unlock
        m_pCoderDescSteerAngleData->Unlock(pCoder);

        // test the received data
        // SteerAngle
        if(ui16SteerAngle != nIndex)
        {
            __adtf_test_ext(tFalse, cString::Format("Wrong Value is(%d) should be
            (%d)", ui16SteerAngle, nIndex));
        }
        // timestamp is always 0
        if(ui32ArduinoTimestamp != 0)
        {
            __adtf_test_ext(tFalse, cString::Format("Wrong Value is(%d) should be
            0", ui32ArduinoTimestamp));
        }
    }
}

tTestResult evaluateRPM(cMediaSampleSink *pSampleSink, tUInt32 ui32MaxLoopCount)
{
    // now check the received data
    __adtf_test_ext(pSampleSink->m_ui32SampleCount == ui32MaxLoopCount,
    cString::Format("Sample count (%d) does not match (%d)", pSampleSink-
    >m_ui32SampleCount, ui32MaxLoopCount));
}

```

```

// get description manager
cObjectPtr<IMediaDescriptionManager> pDescManager;
__adtf_test_result(_runtime-
>GetObject(OID_ADTF_MEDIA_DESCRIPTION_MANAGER, IID_ADTF_MEDIA_DESCRIPTION_MANAGER, (tVoid**)
&pDescManager));

// get media description for tArduinoData
tChar const * strDescRPM = pDescManager->GetMediaDescription("tWheelEncoderData");
__adtf_test_pointer(strDescRPM);

// create mediatype for coder
cObjectPtr<IMediaType> pTypeSteerAngleData = new cMediaType(0, 0, 0,
"tWheelEncoderData", strDescRPM, IMediaDescription::MDF_DDL_DEFAULT_VERSION);

// get the mediatype description from the mediatype
cObjectPtr<IMediaTypeDescription> m_pCoderDescSteerAngleData;
__adtf_test_result(pTypeSteerAngleData-
>GetInterface(IID_ADTF_MEDIA_TYPE_DESCRIPTION, (tVoid**) &m_pCoderDescSteerAngleData));

// loop over the received data
for(tSize nIndex = 0; nIndex < pSampleSink->m_vecMediaSamples.size(); ++nIndex)
{
    // get the coder
    cObjectPtr<IMediaCoder> pCoder;
    m_pCoderDescSteerAngleData->Lock(pSampleSink->m_vecMediaSamples[nIndex],
    &pCoder);

    // init the values
    tUInt32 ui32RPMLeft = 0;
    tUInt32 ui32RPMRight = 0;
    tUInt32 ui32ArduinoTimestamp = 0;

    // get values from coder
    pCoder->Get("ui32LeftWheel", (tVoid*)&ui32RPMLeft);
    pCoder->Get("ui32RightWheel", (tVoid*)&ui32RPMRight);
    pCoder->Get("ui32ArduinoTimestamp", (tVoid*)&ui32ArduinoTimestamp);

    // unlock
    m_pCoderDescSteerAngleData->Unlock(pCoder);

    // test the received data
    // RPM
    // Test for the first samples for Left RPM data, then test the right RPM data
    if(nIndex < 65280)
    {
        if(ui32RPMLeft != nIndex)
        {
            __adtf_test_ext(tFalse, cString::Format("Wrong Value is(%d) should be
(%d)", ui32RPMLeft, nIndex));
        }
    }
    if(nIndex >= 65280)
    {
        if(ui32RPMRight != nIndex-65280)
        {
            __adtf_test_ext(tFalse, cString::Format("Wrong Value is(%d) should be
(%d)", ui32RPMRight, nIndex-65280));
        }
    }
    // timestamp is always 0
    if(ui32ArduinoTimestamp != 0)
    {

```

```
        __adtf_test_ext(tFalse, cString::Format("Wrong Value is(%d) should be
0", ui32ArduinoTimestamp));
    }

}

tTestResult evaluateGyro(cMediaSampleSink *pSampleSink, tUInt32 ui32MaxLoopCount)
{
    // now check the received data
    __adtf_test_ext(pSampleSink->m_ui32SampleCount == ui32MaxLoopCount,
cString::Format("Sample count (%d) does not match (%d)", pSampleSink-
>m_ui32SampleCount, ui32MaxLoopCount));

    // get description manager
    cObjectPtr<IMediaDescriptionManager> pDescManager;
    __adtf_test_result(_runtime-
>GetObject(OID_ADTF_MEDIA_DESCRIPTION_MANAGER, IID_ADTF_MEDIA_DESCRIPTION_MANAGER, (tVoi
d**) &pDescManager));

    // get media description for tArduinoData
    tChar const * strDescRPM = pDescManager->GetMediaDescription("tGyroData");
    __adtf_test_pointer(strDescRPM);

    // create mediatype for coder
    cObjectPtr<IMediaType> pTypeSteerAngleData = new cMediaType(0, 0, 0, "tGyroData",
strDescRPM, IMediaDescription::MDF_DDL_DEFAULT_VERSION);

    // get the mediatype description from the mediatype
    cObjectPtr<IMediaTypeDescription> m_pCoderDescSteerAngleData;
    __adtf_test_result(pTypeSteerAngleData-
>GetInterface(IID_ADTF_MEDIA_TYPE_DESCRIPTION, (tVoid**) &m_pCoderDescSteerAngleData));

    // define the maximum Value for each sensor
    const tUInt32 ui32MaxValue = 65280;

    // loop over the received data
    for(tSize nIdx = 0; nIdx < ui32MaxValue*4; ++nIdx)
    {
        // get the coder
        cObjectPtr<IMediaCoder> pCoder;
        m_pCoderDescSteerAngleData->Lock(pSampleSink->m_vecMediaSamples[nIdx],
&pCoder);

        // init the values
        tInt16 i16Q_w = 0;
        tInt16 i16Q_x = 0;
        tInt16 i16Q_y = 0;
        tInt16 i16Q_z = 0;
        tUInt32 ui32ArduinoTimestamp = 0;

        // get values from coder
        pCoder->Get("i16Q_w", &i16Q_w);
        pCoder->Get("i16Q_x", &i16Q_x);
        pCoder->Get("i16Q_y", &i16Q_y);
        pCoder->Get("i16Q_z", &i16Q_z);
        pCoder->Get("ui32ArduinoTimestamp", &ui32ArduinoTimestamp);

        // unlock
        m_pCoderDescSteerAngleData->Unlock(pCoder);

        // test the received data
        // Gyro
    }
}
```

```

// Test the Gyro data, as in the acc test there are also negative values.
if(nIdx < ui32MaxValue*1)
{
    if(i16Q_w>=0)
    {
        if(i16Q_w!=nIdx)
        {
            __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should
be (%d)",i16Q_w,nIdx));
        }
    }else{
        if(i16Q_w!=-65536+nIdx)
        {
            __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should
be (%d)",i16Q_w,-65536+nIdx));
        }
    }
}else if(nIdx < ui32MaxValue*2)
{
    if(i16Q_x>=0)
    {
        if(i16Q_x!=(nIdx-ui32MaxValue*1))
        {
            __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should
be (%d)",i16Q_x,(nIdx-ui32MaxValue*1)));
        }
    }else{
        if(i16Q_x!=-65536+(nIdx-ui32MaxValue*1))
        {
            __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should
be (%d)",i16Q_x,-65536+(nIdx-ui32MaxValue*1)));
        }
    }
}else if(nIdx < ui32MaxValue*3)
{
    if(i16Q_y>=0)
    {
        if(i16Q_y!=(nIdx-ui32MaxValue*2))
        {
            __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should
be (%d)",i16Q_y,(nIdx-ui32MaxValue*2)));
        }
    }else{
        if(i16Q_y!=-65536+(nIdx-ui32MaxValue*2))
        {
            __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should
be (%d)",i16Q_y,-65536+(nIdx-ui32MaxValue*2)));
        }
    }
}else
{
    if(i16Q_z>=0)
    {
        if(i16Q_z!=(nIdx-ui32MaxValue*3))
        {
            __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should
be (%d)",i16Q_z,(nIdx-ui32MaxValue*3)));
        }
    }else{
        if(i16Q_z!=-65536+(nIdx-ui32MaxValue*3))
        {

```



```

        __adtf_test_ext(tFalse, cString::Format("Wrong Value is(%d) should
be (%d)", i16Q_z, -65536+(nIdx-ui32MaxValue*3)));
    }
}
// timestamp is always 0
if(ui32ArduinoTimestamp != 0)
{
    __adtf_test_ext(tFalse, cString::Format("Wrong Value is(%d) should be
(0)", ui32ArduinoTimestamp));
}
}
tTestResult evaluateAcc(cMediaSampleSink *pSampleSink, tUInt32 ui32MaxLoopCount)
{
    // now check the received data
    __adtf_test_ext(pSampleSink->m_ui32SampleCount == ui32MaxLoopCount,
cString::Format("Sample count (%d) does not match (%d)", pSampleSink-
>m_ui32SampleCount, ui32MaxLoopCount));

    // get description manager
    cObjectPtr<IMediaDescriptionManager> pDescManager;
    __adtf_test_result(_runtime-
>GetObject(OID_ADTF_MEDIA_DESCRIPTION_MANAGER, IID_ADTF_MEDIA_DESCRIPTION_MANAGER, (tVoi
d**)&pDescManager));

    // get media description for tArduinoData
    tChar const * strDescRPM = pDescManager->GetMediaDescription("tAccData");
    __adtf_test_pointer(strDescRPM);

    // create mediatype for coder
    cObjectPtr<IMediaType> pTypeSteerAngleData = new cMediaType(0, 0, 0, "tAccData",
strDescRPM, IMediaDescription::MDF_DDL_DEFAULT_VERSION);

    // get the mediatype description from the mediatype
    cObjectPtr<IMediaTypeDescription> m_pCoderDescSteerAngleData;
    __adtf_test_result(pTypeSteerAngleData-
>GetInterface(IID_ADTF_MEDIA_TYPE_DESCRIPTION, (tVoid**)&m_pCoderDescSteerAngleData));

    // define the maximum Value for each sensor
    const tUInt32 ui32MaxValue = 65280;

    // loop over the received data
    for(tSize nIdx = ui32MaxValue*4; nIdx < pSampleSink->m_vecMediaSamples.size();
++nIdx)
    {
        // get the coder
        cObjectPtr<IMediaCoder> pCoder;
        m_pCoderDescSteerAngleData->Lock(pSampleSink->m_vecMediaSamples[nIdx],
&pCoder);

        // init the values
        tInt16 i16A_x = 0;
        tInt16 i16A_y = 0;
        tInt16 i16A_z = 0;
        tUInt32 ui32ArduinoTimestamp = 0;

        // get values from coder
        pCoder->Get("i16A_x", &i16A_x);
        pCoder->Get("i16A_y", &i16A_y);
        pCoder->Get("i16A_z", &i16A_z);
        // useless because there are the same values as in i16A_x
    }
}

```

```
// pCoder->Get("i16Temperature", &i16A_x);
pCoder->Get("ui32ArduinoTimestamp", &ui32ArduinoTimestamp);

// unlock
m_pCoderDescSteerAngleData->Unlock(pCoder);

// test the received data
// Acc
// Test the three axes one after the other
if(nIdx < ui32MaxValue*5)
{
    // Because there are also negative values we have to test them too
    if(i16A_x>=0)
    {
        if(i16A_x!=(nIdx-ui32MaxValue*4))
        {
            __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should
be (%d)",i16A_x,(nIdx-ui32MaxValue*4)));
        }
    }else{
        if(i16A_x!=-65536+(nIdx-ui32MaxValue*4))
        {
            __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should
be (%d)",i16A_x,-65536+(nIdx-ui32MaxValue*4)));
        }
    }
}else if(nIdx < ui32MaxValue*6)
{
    if(i16A_y>=0)
    {
        if(i16A_y!=(nIdx-ui32MaxValue*5))
        {
            __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should
be (%d)",i16A_y,(nIdx-ui32MaxValue*5)));
        }
    }else{
        if(i16A_y!=-65536+(nIdx-ui32MaxValue*5))
        {
            __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should
be (%d)",i16A_y,-65536+(nIdx-ui32MaxValue*5)));
        }
    }
}else if(nIdx < ui32MaxValue*7)
{
    if(i16A_z>=0)
    {
        if(i16A_z!=(nIdx-ui32MaxValue*6))
        {
            __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should
be (%d)",i16A_z,(nIdx-ui32MaxValue*6)));
        }
    }else{
        if(i16A_z!=-65536+(nIdx-ui32MaxValue*6))
        {
            __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should
be (%d)",i16A_z,-65536+(nIdx-ui32MaxValue*6)));
        }
    }
}else{
    // Do nothing because there is the temperature data which is not usefull
}
// timestamp is always 0
```

```

        if(ui32ArduinoTimestamp != 0)
        {
            __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should be
(0)",ui32ArduinoTimestamp));
        }
    }
}
tTestResult evaluateIR(cMediaSampleSink *pSampleSink, tUInt32 ui32MaxLoopCount)
{
    // now check the received data
    __adtf_test_ext(pSampleSink->m_ui32SampleCount == ui32MaxLoopCount,
cString::Format("Sample count (%d) does not match (%d)", pSampleSink-
>m_ui32SampleCount, ui32MaxLoopCount));

    // get description manager
    cObjectPtr<IMediaDescriptionManager> pDescManager;
    __adtf_test_result(_runtime-
>GetObject(OID_ADTF_MEDIA_DESCRIPTION_MANAGER,IID_ADTF_MEDIA_DESCRIPTION_MANAGER,(tVoi
d**)&pDescManager));

    // get media description for tArduinoData
    tChar const * strDescRPM = pDescManager->GetMediaDescription("tIrData");
    __adtf_test_pointer(strDescRPM);

    // create mediatype for coder
    cObjectPtr<IMediaType> pTypeSteerAngleData = new cMediaType(0, 0, 0, "tIrData",
strDescRPM,IMediaDescription::MDF_DDL_DEFAULT_VERSION);

    // get the mediatype description from the mediatype
    cObjectPtr<IMediaTypeDescription> m_pCoderDescSteerAngleData;
    __adtf_test_result(pTypeSteerAngleData-
>GetInterface(IID_ADTF_MEDIA_TYPE_DESCRIPTION, (tVoid**)&m_pCoderDescSteerAngleData));

    // loop over the received data
    for(tSize nIdx = 0; nIdx < pSampleSink->m_vecMediaSamples.size(); ++nIdx)
    {
        // get the coder
        cObjectPtr<IMediaCoder> pCoder;
        m_pCoderDescSteerAngleData->Lock(pSampleSink->m_vecMediaSamples[nIdx],
&pCoder);

        // init the values
        tUInt16 ui16Front_Center_Longrange=0;
        tUInt16 ui16Front_Center_Shortrange=0;
        tUInt16 ui16Front_Left_Longrange=0;
        tUInt16 ui16Front_Left_Shortrange=0;
        tUInt16 ui16Front_Right_Shortrange=0;
        tUInt16 ui16Front_Right_Longrange=0;
        tUInt16 ui16Rear_Center_Shortrange=0;
        tUInt16 ui16Rear_Left_Shortrange=0;
        tUInt16 ui16Rear_Right_Shortrange=0;
        tUInt32 ui32ArduinoTimestamp = 0;

        // get values from coder
        pCoder->Get("ui16Front_Center_Longrange", &ui16Front_Center_Longrange);
        pCoder->Get("ui16Front_Center_Shortrange", &ui16Front_Center_Shortrange);
        pCoder->Get("ui16Front_Left_Longrange", &ui16Front_Left_Longrange);
        pCoder->Get("ui16Front_Left_Shortrange", &ui16Front_Left_Shortrange);
        pCoder->Get("ui16Front_Right_Shortrange", &ui16Front_Right_Shortrange);
        pCoder->Get("ui16Front_Right_Longrange", &ui16Front_Right_Longrange);
        pCoder->Get("ui16Rear_center_Shortrange", &ui16Rear_Center_Shortrange);
        pCoder->Get("ui16Rear_Left_Shortrange", &ui16Rear_Left_Shortrange);
    }
}

```

```
pCoder->Get("ui16Rear_Right_Shortrange", &ui16Rear_Right_Shortrange);

// unlock
m_pCoderDescSteerAngleData->Unlock(pCoder);

tUInt32 ui32MaxValue = 65280;

// test the received data
// IR
// Test every single Sensor because of the send data they are counted up one
after the other
if(nIdx < ui32MaxValue*1)
{
    if(ui16Front_Center_Longrange!=nIdx)
    {
        __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should be
(%d)",ui16Front_Center_Longrange,nIdx));
    }
}else if(nIdx < ui32MaxValue*2)
{
    if(ui16Front_Center_Shortrange!=nIdx-ui32MaxValue*1)
    {
        __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should be
(%d)",ui16Front_Center_Shortrange,nIdx));
    }
}else if(nIdx < ui32MaxValue*3)
{
    if(ui16Front_Left_Longrange!=nIdx-ui32MaxValue*2)
    {
        __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should be
(%d)",ui16Front_Left_Longrange,nIdx-ui32MaxValue*2));
    }
}else if(nIdx < ui32MaxValue*4)
{
    if(ui16Front_Left_Shortrange!=nIdx-ui32MaxValue*3)
    {
        __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should be
(%d)",ui16Front_Left_Shortrange,nIdx-ui32MaxValue*3));
    }
}else if(nIdx < ui32MaxValue*5)
{
    if(ui16Front_Right_Longrange!=nIdx-ui32MaxValue*4)
    {
        __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should be
(%d)",ui16Front_Right_Longrange,nIdx-ui32MaxValue*5));
    }
}else if(nIdx < ui32MaxValue*6)
{
    if(ui16Front_Right_Shortrange!=nIdx-ui32MaxValue*5)
    {
        __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should be
(%d)",ui16Front_Right_Shortrange,nIdx-ui32MaxValue*4));
    }
}else if(nIdx < ui32MaxValue*7)
{
    if(ui16Rear_Center_Shortrange!=nIdx-ui32MaxValue*6)
    {
        __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should be
(%d)",ui16Rear_Center_Shortrange,nIdx-ui32MaxValue*6));
    }
}else if(nIdx < ui32MaxValue*8)
{
    if(ui16Rear_Right_Shortrange!=nIdx-ui32MaxValue*7)
    {
        __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should be
(%d)",ui16Rear_Right_Shortrange,nIdx-ui32MaxValue*7));
    }
}
```

```

        if(ui16Rear_Left_Shortrange!=nIdx-ui32MaxValue*7)
        {
            __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should be
(%d)",ui16Rear_Left_Shortrange,nIdx-ui32MaxValue*7));
        }
    }else if(nIdx < ui32MaxValue*9)
    {
        if(ui16Rear_Right_Shortrange!=nIdx-ui32MaxValue*8)
        {
            __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should be
(%d)",ui16Rear_Right_Shortrange,nIdx-ui32MaxValue*8));
        }
    }
    // timestamp is always 0
    if(ui32ArduinoTimestamp != 0)
    {
        __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should be
(0)",ui32ArduinoTimestamp));
    }
}
}
/**
tTestResult evaluatePHOTO(cMediaSampleSink *pSampleSink, tUInt32 ui32MaxLoopCount)
{
    // now check the received data
    __adtf_test_ext(pSampleSink->m_ui32SampleCount == ui32MaxLoopCount,
cString::Format("Sample count (%d) does not match (%d)", pSampleSink-
>m_ui32SampleCount, ui32MaxLoopCount));

    // get description manager
    cObjectPtr<IMediaDescriptionManager> pDescManager;
    __adtf_test_result(_runtime-
>GetObject(OID_ADTF_MEDIA_DESCRIPTION_MANAGER,IID_ADTF_MEDIA_DESCRIPTION_MANAGER,(tVoi
d**)&pDescManager));

    // get media description for tArduinoData
    tChar const * strDescSteerAngle = pDescManager->GetMediaDescription("tPhotoData");
    __adtf_test_pointer(strDescSteerAngle);

    // create mediatype for coder
    cObjectPtr<IMediaType> pTypeSteerAngleData = new cMediaType(0, 0, 0, "tPhotoData",
strDescSteerAngle,IMediaDescription::MDF_DDL_DEFAULT_VERSION);

    // get the mediatype description from the mediatype
    cObjectPtr<IMediaTypeDescription> m_pCoderDescSteerAngleData;
    __adtf_test_result(pTypeSteerAngleData-
>GetInterface(IID_ADTF_MEDIA_TYPE_DESCRIPTION, (tVoid**)&m_pCoderDescSteerAngleData));

    // loop over the received data
    for(tSize nIdx = 0; nIdx < pSampleSink->m_vecMediaSamples.size(); ++nIdx)
    {
        // get the coder
        cObjectPtr<IMediaCoder> pCoder;
        m_pCoderDescSteerAngleData->Lock(pSampleSink->m_vecMediaSamples[nIdx],
&pCoder);

        // init the values
        tUInt32 ui32luminosity = 0;
        tUInt32 ui32ArduinoTimestamp = 0;

        // get values from coder

```

```

pCoder->Get("ui32luminosity", (tVoid*)&ui32luminosity);
pCoder->Get("ui32ArduinoTimestamp", (tVoid*)&ui32ArduinoTimestamp);

// unlock
m_pCoderDescSteerAngleData->Unlock(pCoder);

// test the received data
// SteerAngle
if(ui32luminosity != nIdx)
{
    __adtf_test_ext(tFalse, cString::Format("Wrong Value is(%d) should be
(%d)", ui32luminosity, nIdx));
}
// timestamp is always 0
if(ui32ArduinoTimestamp != 0)
{
    __adtf_test_ext(tFalse, cString::Format("Wrong Value is(%d) should be
0", ui32ArduinoTimestamp));
}

}
}
**/
tTestResult evaluateUS(cMediaSampleSink *pSampleSink, tUInt32 ui32MaxLoopCount)
{
    // now check the received data
    __adtf_test_ext(pSampleSink->m_ui32SampleCount == ui32MaxLoopCount,
cString::Format("Sample count (%d) does not match (%d)", pSampleSink-
>m_ui32SampleCount, ui32MaxLoopCount));

    // get description manager
    cObjectPtr<IMediaDescriptionManager> pDescManager;
    __adtf_test_result(_runtime-
>GetObject(OID_ADTF_MEDIA_DESCRIPTION_MANAGER, IID_ADTF_MEDIA_DESCRIPTION_MANAGER, (tVoi
d**) &pDescManager));

    // get media description for tArduinoData
    tChar const * strDescRPM = pDescManager->GetMediaDescription("tUsData");
    __adtf_test_pointer(strDescRPM);

    // create mediatype for coder
    cObjectPtr<IMediaType> pTypeSteerAngleData = new cMediaType(0, 0, 0, "tUsData",
strDescRPM, IMediaDescription::MDF_DDL_DEFAULT_VERSION);

    // get the mediatype description from the mediatype
    cObjectPtr<IMediaTypeDescription> m_pCoderDescSteerAngleData;
    __adtf_test_result(pTypeSteerAngleData-
>GetInterface(IID_ADTF_MEDIA_TYPE_DESCRIPTION, (tVoid**) &m_pCoderDescSteerAngleData));

    // loop over the received data
    for(tSize nIdx = 0; nIdx < pSampleSink->m_vecMediaSamples.size(); ++nIdx)
    {
        // get the coder
        cObjectPtr<IMediaCoder> pCoder;
        m_pCoderDescSteerAngleData->Lock(pSampleSink->m_vecMediaSamples[nIdx],
&pCoder);

        // init the values
        tUInt16 ui16FrontLeft=0;
        tUInt16 ui16FrontRight=0;
        tUInt16 ui16RearLeft=0;
        tUInt16 ui16RearRight=0;
    }
}

```

```

tUInt32 ui32ArduinoTimestamp = 0;

// get values from coder
pCoder->Get("ui16Front_Left", &ui16FrontLeft);
pCoder->Get("ui16Front_Right", &ui16FrontRight);
pCoder->Get("ui16Rear_Left", &ui16RearLeft);
pCoder->Get("ui16Rear_Right", &ui16RearRight);
pCoder->Get("ui32ArduinoTimestamp", &ui32ArduinoTimestamp);

// unlock
m_pCoderDescSteerAngleData->Unlock(pCoder);

tUInt32 ui32MaxValue = 65280;

// test the received data
// US
// Test for the first samples for US Front Left data,
if(nIdx < ui32MaxValue*1)
{
    if(ui16FrontLeft!=nIdx)
    {
        __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should be
(%d)",ui16FrontLeft,nIdx));
    }
    // then test the Front right data
}else if(nIdx < ui32MaxValue*2)
{
    if(ui16FrontRight!=nIdx-ui32MaxValue*1)
    {
        __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should be
(%d)",ui16FrontRight,nIdx));
    }
    // now the rear left data
}else if(nIdx < ui32MaxValue*3)
{
    if(ui16RearLeft!=nIdx-ui32MaxValue*2)
    {
        __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should be
(%d)",ui16RearLeft,nIdx-ui32MaxValue*2));
    }
    // and finally the rear right data
}else if(nIdx < ui32MaxValue*4)
{
    if(ui16RearRight!=nIdx-ui32MaxValue*3)
    {
        __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should be
(%d)",ui16RearRight,nIdx-ui32MaxValue*3));
    }
}
// timestamp is always 0
if(ui32ArduinoTimestamp != 0)
{
    __adtf_test_ext(tFalse,cString::Format("Wrong Value is(%d) should be
(0)",ui32ArduinoTimestamp));
}
}

tTestResult evaluateVOLTAGE(cMediaSampleSink *pSampleSink, tUInt32 ui32MaxLoopCount)
{
    // now check the received data

```

```

    __adtf_test_ext(pSampleSink->m_ui32SampleCount == ui32MaxLoopCount,
cString::Format("Sample count (%d) does not match (%d)", pSampleSink-
>m_ui32SampleCount, ui32MaxLoopCount));

    // get description manager
    cObjectPtr<IMediaDescriptionManager> pDescManager;
    __adtf_test_result(_runtime-
>GetObject(OID_ADTF_MEDIA_DESCRIPTION_MANAGER, IID_ADTF_MEDIA_DESCRIPTION_MANAGER, (tVoi
d**) &pDescManager));

    // get media description for tArduinoData
    tChar const * strDescRPM = pDescManager->GetMediaDescription("tVoltageData");
    __adtf_test_pointer(strDescRPM);

    // create mediatype for coder
    cObjectPtr<IMediaType> pTypeSteerAngleData = new cMediaType(0, 0, 0,
"tVoltageData", strDescRPM, IMediaDescription::MDF_DDL_DEFAULT_VERSION);

    // get the mediatype description from the mediatype
    cObjectPtr<IMediaTypeDescription> m_pCoderDescSteerAngleData;
    __adtf_test_result(pTypeSteerAngleData-
>GetInterface(IID_ADTF_MEDIA_TYPE_DESCRIPTION, (tVoid**) &m_pCoderDescSteerAngleData));

    // loop over the received data
    for(tSize nIdx = 0; nIdx < pSampleSink->m_vecMediaSamples.size(); ++nIdx)
    {
        // get the coder
        cObjectPtr<IMediaCoder> pCoder;
        m_pCoderDescSteerAngleData->Lock(pSampleSink->m_vecMediaSamples[nIdx],
&pCoder);

        // init the values
        tUInt16 ui16Measurement=0;
        tUInt16 ui16Power=0;
        tUInt32 ui32ArduinoTimestamp = 0;

        // get values from coder
        pCoder->Get("ui32MeasurementCircuit", &ui16Measurement);
        pCoder->Get("ui32PowerCircuit", &ui16Power);
        pCoder->Get("ui32ArduinoTimestamp", &ui32ArduinoTimestamp);

        // unlock
        m_pCoderDescSteerAngleData->Unlock(pCoder);

        tUInt32 ui32MaxValue = 65280;

        // test the received data
        // voltage
        // Test for the first samples for MeasurementCircuit data, then test the
PowerCircuit data
        if(nIdx < ui32MaxValue*1)
        {
            if(ui16Measurement!=nIdx)
            {
                __adtf_test_ext(tFalse, cString::Format("Wrong Value is(%d) should be
(%d)", ui16Measurement, nIdx));
            }
        }else
        {
            if(ui16Power!=nIdx-ui32MaxValue*1)
            {

```



```

        __adtf_test_ext(tFalse, cString::Format("Wrong Value is(%d) should be
(%d)", ui16Power, nIndex));
    }
}
// timestamp is always 0
if(ui32ArduinoTimestamp != 0)
{
    __adtf_test_ext(tFalse, cString::Format("Wrong Value is(%d) should be
(0)", ui32ArduinoTimestamp));
}
}
}

tTestResult DoSensorTest(const cString & strInputPinName, const tUInt8 chFrameId)
{
    // initialize filter
    INIT_FILTER(pFilter, pFilterConfig, "adtf.aadc.sensors");

    // set filter to state ready
    SET_STATE_READY(pFilter);

    // get the input pin
    cObjectPtr<IPin> pCommPin;
    __adtf_test_result_ext(pFilter->FindPin("ArduinoCOM_input", IPin::PD_Input,
&pCommPin), "Unable to find input pin");

    // get the output pin
    cObjectPtr<IPin> pOutput;
    __adtf_test_result_ext(pFilter->FindPin(strInputPinName.GetPtr(), IPin::PD_Output,
&pOutput), cString::Format("Unable to find pin %s", strInputPinName.GetPtr()));

    // register the sample sink to receive the data from the output pin
    cObjectPtr<cMediaSampleSink> pSampleSink = new cMediaSampleSink();
    __adtf_test_result(pOutput->RegisterEventSink(pSampleSink))

    // for each imu sample the sensors filter sends a acc sample and a gyro sample, so
we need two seperated sample sinks.
    // beacause of the test parameter gyroscope there is already a gyroscope sink
registered, so now we need a accelerometer sink
    cObjectPtr<IPin> pOutputAcc;
    cObjectPtr<cMediaSampleSink> pSampleSinkAcc = new cMediaSampleSink();
    if(chFrameId == ID_ARD_SENS_IMU)
    {
        // get the output pin
        __adtf_test_result_ext(pFilter->FindPin("accelerometer", IPin::PD_Output,
&pOutputAcc), "Unable to find Accelerometer pin" );
        // register the sample sink to receive the data from the output pin
        __adtf_test_result(pOutputAcc->RegisterEventSink(pSampleSinkAcc));
    }

    ///#####
    ///#####
    /// Sample Sink on every output pin

    ///#####
    ///#####

    /// get the output pin
    ///cObjectPtr<IPin> pOutput2;

```

```

    __adtf_test_result_ext(pFilter->FindPin("gyroscope", IPin::PD_Output,
&pOutput2), CString::Format("Unable to find pin %s",strInputPinName));

    /// register the sample sink to receive the data from the output pin
    //cObjectPtr<cMediaSampleSink> pSampleSink2 = new cMediaSampleSink();
    __adtf_test_result(pOutput2->RegisterEventSink(pSampleSink2));

    /// get the output pin
    //cObjectPtr<IPin> pOutput3;
    __adtf_test_result_ext(pFilter->FindPin("infrared_sensors", IPin::PD_Output,
&pOutput3), CString::Format("Unable to find pin %s",strInputPinName));

    /// register the sample sink to receive the data from the output pin
    //cObjectPtr<cMediaSampleSink> pSampleSink3 = new cMediaSampleSink();
    __adtf_test_result(pOutput3->RegisterEventSink(pSampleSink3));

    /// get the output pin
    //cObjectPtr<IPin> pOutput4;
    __adtf_test_result_ext(pFilter->FindPin("ultrasonic_sensors", IPin::PD_Output,
&pOutput4), CString::Format("Unable to find pin %s",strInputPinName));

    /// register the sample sink to receive the data from the output pin
    //cObjectPtr<cMediaSampleSink> pSampleSink4 = new cMediaSampleSink();
    __adtf_test_result(pOutput4->RegisterEventSink(pSampleSink4));

    /// get the output pin
    //cObjectPtr<IPin> pOutput5;
    __adtf_test_result_ext(pFilter->FindPin("system_voltage", IPin::PD_Output,
&pOutput5), CString::Format("Unable to find pin %s",strInputPinName));

    /// register the sample sink to receive the data from the output pin
    //cObjectPtr<cMediaSampleSink> pSampleSink5 = new cMediaSampleSink();
    __adtf_test_result(pOutput5->RegisterEventSink(pSampleSink5));

    ///#####
    ///#####
    /// Sample Sink on every output pin

    ///#####
    ///#####

    // create a simple sample source
    cObjectPtr<cOutputPin> pSampleSource = new cOutputPin();
    __adtf_test_result(pSampleSource->Create("sampleSource", new
cMediaType(0,0,0,"tArduinoData")));

    // connect the communication pin with the sample source
    __adtf_test_result(pCommPin->Connect(pSampleSource));

    // set the filter state running
    SET_STATE_RUNNING(pFilter);

    // get the mediatype of the sample source
    cObjectPtr<IMediaType> pType;
    __adtf_test_result(pSampleSource->GetMediaType(&pType));

    // get the type description
    cObjectPtr<IMediaTypeDescription> pTypeDesc;
    __adtf_test_result(pType->GetInterface(IID_ADTF_MEDIA_TYPE_DESCRIPTION,
(tVoid*)&pTypeDesc));

    // get the serializer from description to get the deserialized size

```

```

cObjectPtr<IMediaSerializer> pSerializer;
__adtf_test_result(pTypeDesc->GetMediaSampleSerializer(&pSerializer));
tInt nSize = pSerializer->GetDeserializedSize();
pSerializer = NULL;

// init the data of the protocol
tUInt8 ui8Data[25]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

// define the loop count, also defines the datarange
tUInt32 ui32MaxLoopCount = 2000000;
// define the position to write data
tUInt8 ui8PositionToWrite = 0;
// define the size of a single value
tUInt8 ui8SizeOfValue = 2;
// define the value to write
tUInt16 ui16Value = 0;
// init the Datalength of the protocol
tUInt8 ui8DataLength = 0;
// define the Datalength and the size of the values of the cuurent protocol
if(chFrameId == ID_ARD_SENS_STEER_ANGLE)
{
    ui8DataLength = 2;
}
if(chFrameId == ID_ARD_SENS_WHEELENC)
{
    ui8DataLength = 8;
    ui8SizeOfValue = 4;
}
if(chFrameId == ID_ARD_SENS_IMU)
{
    ui8DataLength = 16;
}
if(chFrameId == ID_ARD_SENS_IR)
{
    ui8DataLength = 18;
}
/**
if(chFrameId == ID_ARD_SENS_PHOTO)
{
    ui8DataLength = 2;
}
**/
if(chFrameId == ID_ARD_SENS_US)
{
    ui8DataLength = 8;
}
if(chFrameId == ID_ARD_SENS_VOLTAGE)
{
    ui8DataLength = 4;
}

// send some samples
for (tUInt32 nIdx = 0; nIdx < ui32MaxLoopCount; nIdx++)
{
    // Because of 8 bit in each byte we want to increase the next higher byte with
    each full lower byte
    if(ui8Data[ui8PositionToWrite] == static_cast <tUInt8> (255))
    {
        ui8Data[ui8PositionToWrite+1] += 1;
        ui16Value = 0;
    }
}

```

```

    }
    // If the current value range is full go to the next value range
    if(ui8Data[ui8PositionToWrite+1] == static_cast<tUInt8> (255))
    {
        ui16Value = 0;
        ui8PositionToWrite += ui8SizeOfValue;
    }
    // Check if the end of the current protocol is reached if yes quit the loop
    but set the goal of received samples to match the send samples
    if(ui8PositionToWrite == ui8DataLength)
    {
        ui32MaxLoopCount = nIdx;
        break;
    }

    // set the actual value
    ui8Data[ui8PositionToWrite] = ui16Value;

    // transmit the value
    transmitMediaSample(pSampleSource, chFrameId, ui8DataLength, ui8Data);
    // increase the Value for the next time.
    ui16Value++;
}

// clean up
__adtf_test_result(pCommPin->Disconnect(pSampleSource));
__adtf_test_result(pOutput->UnregisterEventSink(pSampleSink));

//#####
// Evaluation
//#####

if(chFrameId == ID_ARD_SENS_STEER_ANGLE) evaluateSteerAngle(pSampleSink,
ui32MaxLoopCount);
if(chFrameId == ID_ARD_SENS_WHEELENC) evaluateRPM(pSampleSink, ui32MaxLoopCount);
if(chFrameId == ID_ARD_SENS_IMU)
{
    evaluateGyro(pSampleSink, ui32MaxLoopCount);
    evaluateAcc(pSampleSinkAcc, ui32MaxLoopCount);
}
if(chFrameId == ID_ARD_SENS_IR) evaluateIR(pSampleSink, ui32MaxLoopCount);
//if(chFrameId == ID_ARD_SENS_PHOTO) evaluatePHOTO(pSampleSink, ui32MaxLoopCount);
if(chFrameId == ID_ARD_SENS_US) evaluateUS(pSampleSink, ui32MaxLoopCount);
if(chFrameId == ID_ARD_SENS_VOLTAGE) evaluateVOLTAGE(pSampleSink,
ui32MaxLoopCount);

// shutdown the filter
SET_STATE_SHUTDOWN(pFilter);
}

DEFINE_TEST(cTesterAADC Sensors,
    TestSteeringAngle,
    "3.1",
    "TestSteeringAngle",
    "This test makes sure, that the 'SteeringAngle' pin is present. After that
some SteeringAngle data\
    "will be send to the filter and the filter output will be compared to some
reference data.",
    "Pin can be found and the output data are as expected.",
    "See above",

```

```
        "none",
        "The filter must be able to receive Arduino data and convert them into
SteeringAngle data.",
        "automatic")
{
    // accelerate pin with expected frameID
    return DoSensorTest("steering_servo", ID_ARD_SENS_STEER_ANGLE);
}
```

```
DEFINE_TEST(cTesterAADCSensors,
            TestRPM,
            "3.2",
            "TestRPM",
            "This test makes sure, that the 'RPM' pin is present. After that some RPM
data"\
            "will be send to the filter and the filter output will be compared to some
reference data.",
            "Pin can be found and the output data are as expected.",
            "See above",
            "none",
            "The filter must be able to receive arduino data and convert them into rpm
data.",
            "automatic")
{
    // steering angle pin with expected frameID
    return DoSensorTest("wheel_speed_sensor", ID_ARD_SENS_WHEEL_ENC);
}
```

```
DEFINE_TEST(cTesterAADCSensors,
            TestIMU,
            "3.3",
            "TestIMU",
            "This test makes sure, that the 'Acc and Gyro' pins are present. After
that some IMU data"\
            "will be send to the filter and the filter output will be compared to some
reference data.",
            "Pin can be found and the output data are as expected.",
            "See above",
            "none",
            "The filter must be able to receive arduino data and convert them into acc
and gyro data.",
            "automatic")
{
    // steering angle pin with expected frameID
    return DoSensorTest("gyroscope", ID_ARD_SENS_IMU);
}
```

```
DEFINE_TEST(cTesterAADCSensors,
            TestIR,
            "3.4",
```

```
        "TestIR",
        "This test makes sure, that the 'IR' pin is present. After that some ir
data"\
        "will be send to the filter and the filter output will be compared to some
reference data.",
        "Pin can be found and the output data are as expected.",
        "See above",
        "none",
        "The filter must be able to receive arduino data and convert them into IR
data.",
        "automatic")
{
    // steering angle pin with expected frameID
    return DoSensorTest("infrared_sensors", ID_ARD_SENS_IR);
}
```

```
DEFINE_TEST(cTesterAADCSensors,
            TestUS,
            "3.5",
            "TestUS",
            "This test makes sure, that the 'US' pin is present. After that some US
data"\
            "will be send to the filter and the filter output will be compared to some
reference data.",
            "Pin can be found and the output data are as expected.",
            "See above",
            "none",
            "The filter must be able to receive arduino data and convert them into US
data.",
            "automatic")
{
    // steering angle pin with expected frameID
    return DoSensorTest("ultrasonic_sensors", ID_ARD_SENS_US);
}
```

```
DEFINE_TEST(cTesterAADCSensors,
            TestVoltage,
            "3.6",
            "TestVoltage",
            "This test makes sure, that the 'Voltage' pin is present. After that some
voltage data"\
            "will be send to the filter and the filter output will be compared to some
reference data.",
            "Pin can be found and the output data are as expected.",
            "See above",
            "none",
            "The filter must be able to receive arduino data and convert them into
voltage data.",
            "automatic")
{
    // steering angle pin with expected frameID
    return DoSensorTest("system_voltage", ID_ARD_SENS_VOLTAGE);
}
```

```

/**
DEFINE_TEST_INACTIVE(cTesterAADC Sensors,
    TestPHOTO,
    "3.7",
    "TestPhoto",
    "This test makes sure, that the 'Photo' pin is present. After that some
photo data"\
    "will be send to the filter and the filter output will be compared to some
reference data.",
    "Pin can be found and the output data are as expected.",
    "See above",
    "none",
    "The filter must be able to receive arduino data and convert them into
photo data.",
    "automatic")
{
    // steering angle pin with expected frameID
    return DoSensorTest("steerAngle", ID_ARD_SENS_PHOTO);
}
**/

```

Programm-Code: Arduino Actuators Filter (Modul)

```

/**
 *
 * AADC Arduino Aktors filter tests
 *
 * @file
 * Copyright &copy; Audi Electronics Venture GmbH. All rights reserved
 *
 * $Author: VG8D3AW $
 * $Date: 2013-02-06 16:30:41 +0100 (Mi, 06. Feb 2013) $
 * $Revision: 18162 $
 *
 * @remarks
 *
 */

#include "stdafx.h"
#include "tester_aadc_aktors.h"

IMPLEMENT_TESTER_CLASS(cTesterAADC Aktors,
    "1",
    "AADC Arduino Aktors filter",
    "This test makes sure that the AADC Arduino Aktors filter works as
expected",
    "");

/*
Setup for this test
*/
void cTesterAADC Aktors::setUp()
{
    // create the environment
    SERVICE_ENV_SETUP;

    // minimum needed ADTF services
    SERVICE_ENV_ADD_PLUGIN("adtf_clock.srv");
}

```

```

SERVICE_ENV_ADD_PLUGIN("adtf_kernel.srv");
SERVICE_ENV_ADD_PLUGIN("adtf_sample_pool.srv");
SERVICE_ENV_ADD_PLUGIN("adtf_namespace.srv");
SERVICE_ENV_ADD_PLUGIN("adtf_media_description.srv");

// register services
SERVICE_ENV_REGISTER_SERVICE(OID_ADTF_REFERENCE_CLOCK, "referenceclock",
IRuntime::RL_Kernel);
SERVICE_ENV_REGISTER_SERVICE(OID_ADTF_KERNEL, "kernel", IRuntime::RL_System);
SERVICE_ENV_REGISTER_SERVICE(OID_ADTF_SAMPLE_POOL, "samplepool",
IRuntime::RL_System);
SERVICE_ENV_REGISTER_SERVICE(OID_ADTF_NAMESPACE, "namespace",
IRuntime::RL_System);
SERVICE_ENV_REGISTER_SERVICE(OID_ADTF_MEDIA_DESCRIPTION_MANAGER, "mediadesc",
IRuntime::RL_System);

// add filter plugin
FILTER_ENV_ADD_PLUGIN("aadc_aktors.plb");

// set runlevel system
SERVICE_ENV_SET_RUNLEVEL(IRuntime::RL_System);

// set the path to the MediaDescription-Files
// (relative to the current working dir which is the tester dir)
cFilename strDescrFileADTF =
"..\\..\\..\\src\\adtfBase\\AADC_ADTF_BaseFilters\\description\\aadc.description";
cFilename strDescrFile =
"..\\..\\..\\src\\adtfBase\\AADC_ADTF_BaseFilters\\description\\aadc.description";

// create absolute path
strDescrFileADTF =
strDescrFileADTF.CreateAbsolutePath(cFileSystem::GetCurDirectory());
strDescrFile = strDescrFile.CreateAbsolutePath(cFileSystem::GetCurDirectory());

// get the media description manager
cObjectPtr<IMediaDescriptionManager> pMediaDesc;
__adtf_test_result_ext(_runtime->GetObject(OID_ADTF_MEDIA_DESCRIPTION_MANAGER,
IID_ADTF_MEDIA_DESCRIPTION_MANAGER, (tVoid**) &pMediaDesc),
    "unable to get media description manager");

// get the config from the media description manager
cObjectPtr<IConfiguration> pConfig;
__adtf_test_result(pMediaDesc->GetInterface(IID_ADTF_CONFIGURATION, (tVoid**)
&pConfig));

// set the path to the description files
__adtf_test_result(pConfig->SetPropertyStr("media_description_files",
cString::Format("%s;%s", strDescrFileADTF.GetPtr(), strDescrFile.GetPtr())));

// check if the media description manager loads the description correctly
__adtf_test(pMediaDesc->GetMediaDescription("tArduinoData") != NULL);

// set runlevel application
SERVICE_ENV_SET_RUNLEVEL(IRuntime::RL_Application);
}

/*
Tear down for this test
*/
void cTesterAADCAktors::tearDown()
{
    // give the kernel some time to quit its threads

```



```

    //cSystem::Sleep(200000);
    FILTER_ENV_TEAR_DOWN;
}

/*****
/* This is just a helper class to receive data from output pins.
*/
*****/
class cMediaSampleSink: public IPinEventSink
{
    // ucom helper macro
    UCOM_OBJECT_IMPL(IID_ADTF_PIN_EVENT_SINK, adtf::IPinEventSink);
public:
    // constructor
    cMediaSampleSink(): m_ui32SampleCount(0)
    {

    }

    // destructor
    virtual ~cMediaSampleSink()
    {
        // loop over all mediasamples to unref
        for (tInt nIndex = 0; nIndex < m_vecMediaSamples.size(); nIndex++)
        {
            // after unref the mediasample will destroy itself
            m_vecMediaSamples[nIndex]->Unref();
        }
    }

    // members to count and hold the received media samples
    tUInt32 m_ui32SampleCount;
    std::vector<IMediaSample*> m_vecMediaSamples;

public:
    tResult OnPinEvent(IPin* pSource, tInt nEventCode, tInt nParam1, tInt nParam2,
    IMediaSample* pMediaSample)
    {
        if (nEventCode == IPinEventSink::PE_MediaSampleTransmitted)
        {
            // count the samples
            ++m_ui32SampleCount;
            // make a ref on sample to avoid self destruction
            pMediaSample->Ref();
            // push sample into vector for later compare
            m_vecMediaSamples.push_back(pMediaSample);
        }

        RETURN_NOERROR;
    }
};

tTestResult DoActuatorTest(const cString & strInputPinName, const tUInt8 chFrameId)
{
    // initialize filter
    INIT_FILTER(pFilter, pFilterConfig, "adtf.aadc.aktors");

    // set filter to state ready
    SET_STATE_READY(pFilter);

    // get the input pin

```

```
cObjectPtr<IPin> pAccPin;
__adtf_test_result_ext(pFilter->FindPin(strInputPinName.GetPtr(), IPin::PD_Input,
&pAccPin), "Unable to find input pin");

// get the output pin
cObjectPtr<IPin> pOutput;
__adtf_test_result_ext(pFilter->FindPin("ArduinoCOM_output", IPin::PD_Output,
&pOutput), "Unable to find pin ArduinoCOM_output");

// register the sample sink to receive the data from the output pin
cObjectPtr<cMediaSampleSink> pSampleSink = new cMediaSampleSink();
__adtf_test_result(pOutput->RegisterEventSink(pSampleSink));

// create a simple sample source
cObjectPtr<cOutputPin> pSampleSource = new cOutputPin();
__adtf_test_result(pSampleSource->Create("sampleSource", new
cMediaType(0,0,0,"tSignalValue")));

// connect the accelerate pin with th sample source
__adtf_test_result(pAccPin->Connect(pSampleSource));

// set the filter state running
SET_STATE_RUNNING(pFilter);

// get the mediatype of the sample source
cObjectPtr<IMediaType> pType;
__adtf_test_result(pSampleSource->GetMediaType(&pType));

// get the type description
cObjectPtr<IMediaTypeDescription> pTypeDesc;
__adtf_test_result(pType->GetInterface(IID_ADTF_MEDIA_TYPE_DESCRIPTION,
(tVoid**)&pTypeDesc));

// get the serializer from description to get the deserialized size
cObjectPtr<IMediaSerializer> pSerializer;
__adtf_test_result(pTypeDesc->GetMediaSampleSerializer(&pSerializer));
tInt nSize = pSerializer->GetDeserializedSize();
pSerializer = NULL;

// define values outside data range
const tFloat32 f32LowerRange = -1.0f;
const tFloat32 f32UpperRange = 181.0f;

// define the loop count
const tUInt32 ui32MaxLoopCount = 100;

// send some samples
for (tUInt32 nIdx = 0; nIdx < ui32MaxLoopCount; nIdx++)
{
    // create and allocate the sample
    cObjectPtr<IMediaSample> pSample;
    __adtf_test_result(_runtime->CreateInstance(OID_ADTF_MEDIA_SAMPLE,
IID_ADTF_MEDIA_SAMPLE, (tVoid**)&pSample));
    __adtf_test_result(pSample->AllocBuffer(nSize));

    // get the coder
    cObjectPtr<IMediaCoder> pCoder;
    __adtf_test_result(pTypeDesc->WriteLock(pSample, &pCoder));

    // set the value
    tFloat32 f32Value = 0.0f;
    if (0 == nIdx)
```

```

{
    // first value must test the lower border of data range
    f32Value = f32LowerRange;
}
else if (ui32MaxLoopCount - 1 == nIdx)
{
    // last value must test the upper border of data range
    f32Value = f32UpperRange;
}
else
{
    // the rest will be calculated by index, range and loop count
    f32Value = (f32UpperRange - f32LowerRange) / ui32MaxLoopCount * nIdx;
}

// use the coder to set the value
__adtf_test_result(pCoder->Set("f32Value", &f32Value));
// unlock the coder
__adtf_test_result(pTypeDesc->Unlock(pCoder));
// set the sample time (in this case the time doesn't matter and must not be
the stream time)
pSample->SetTime(nIdx);
// transmit the media sample
__adtf_test_result(pSampleSource->Transmit(pSample));
}

// clean up
__adtf_test_result(pAccPin->Disconnect(pSampleSource));
__adtf_test_result(pOutput->UnregisterEventSink(pSampleSink));

// now check the received data
__adtf_test_ext(pSampleSink->m_ui32SampleCount == ui32MaxLoopCount,
cString::Format("Sample count (%d) does not match (%d)", pSampleSink-
>m_ui32SampleCount, ui32MaxLoopCount));

// get description manager
cObjectPtr<IMediaDescriptionManager> pDescManager;
__adtf_test_result(_runtime-
>GetObject(OID_ADTF_MEDIA_DESCRIPTION_MANAGER, IID_ADTF_MEDIA_DESCRIPTION_MANAGER, (tVoi
d**) &pDescManager));

// get media description for tArduinoData
tChar const * strDescArduino = pDescManager->GetMediaDescription("tArduinoData");
__adtf_test_pointer(strDescArduino);

// create mediatype for coder
cObjectPtr<IMediaType> pTypeArduinoData = new cMediaType(0, 0, 0, "tArduinoData",
strDescArduino, IMediaDescription::MDF_DDL_DEFAULT_VERSION);

// get the mediatype description from the mediatype
cObjectPtr<IMediaTypeDescription> m_pCoderDescArduinoData;
__adtf_test_result(pTypeArduinoData->GetInterface(IID_ADTF_MEDIA_TYPE_DESCRIPTION,
(tVoid**) &m_pCoderDescArduinoData));

// loop over the received data
for(tSize nIdx = 0; nIdx < pSampleSink->m_vecMediaSamples.size(); ++nIdx)
{
    // get the coder //Search Dummy
    cObjectPtr<IMediaCoder> pCoder;
    m_pCoderDescArduinoData->Lock(pSampleSink->m_vecMediaSamples[nIdx], &pCoder);

```

```

// init the values
tUInt8 ui8SOF = 0;
tUInt8 ui8Id = 0;
tUInt32 ui32ArduinoTimestamp = 0;
tUInt8 ui8DataLength = 0;
tUInt8 achFrameData[25];
cMemoryBlock::MemSet(achFrameData, 0x00, sizeof(achFrameData));

// get values from coder
pCoder->Get("ui8SOF", (tVoid*)&ui8SOF);
pCoder->Get("ui8ID", (tVoid*)&ui8Id);
pCoder->Get("ui32ArduinoTimestamp", (tVoid*)&(ui32ArduinoTimestamp));

pCoder->Get("ui8DataLength", (tVoid*)&ui8DataLength);
pCoder->Get("ui8Data", (tVoid*)&(achFrameData));

// unlock
m_pCoderDescArduinoData->Unlock(pCoder);

// test the received data // Search Dummy
// SOF
__adtf_test(ui8SOF == ID_ARD_SOF);
// Frame ID
__adtf_test(ui8Id == chFrameId);
// timestamp is always 0 in this filter
__adtf_test(ui32ArduinoTimestamp == 0);
__adtf_test(ui8DataLength == 1);

if (nIdx == 0)
{
    // lower range check (values smaller than 0 must be set to 0)
    __adtf_test(achFrameData[0] == 0);
}
else if (nIdx == ui32MaxLoopCount)
{
    // upper range check (values greater than 180 must be set to 180)
    __adtf_test(achFrameData[0] == 180);
}
else
{
    // the value must be the same (with rounding) as set while transmit
    tFloat32 f32Value = ((f32UpperRange - f32LowerRange) *
static_cast<tFloat32> (nIdx) / ui32MaxLoopCount);

    __adtf_test_ext(static_cast<tUInt8> (f32Value + 0.5f) == achFrameData[0],
cString::Format("Received data (%d) not as expected (%d)", achFrameData[0],
static_cast<tUInt8> (f32Value + 0.5f)));
}
}

// shutdown the filter
SET_STATE_SHUTDOWN(pFilter);
}

tTestResult DoActuatorBoolTest(const cString & strInputPinName, const tUInt8
chFrameId)
{
    // initialize filter
    INIT_FILTER(pFilter, pFilterConfig, "adtf.aadc.aktors");

```

```
// set filter to state ready
SET_STATE_READY(pFilter);

// get the input pin
cObjectPtr<IPin> pBoolPin;
__adtf_test_result_ext(pFilter->FindPin(strInputPinName.GetPtr(), IPin::PD_Input,
&pBoolPin), "Unable to find input pin");

// get the output pin
cObjectPtr<IPin> pOutput;
__adtf_test_result_ext(pFilter->FindPin("ArduinoCOM_output", IPin::PD_Output,
&pOutput), "Unable to find pin ArduinoCOM_output");

// register the sample sink to receive the data from the output pin
cObjectPtr<cMediaSampleSink> pSampleSink = new cMediaSampleSink();
__adtf_test_result(pOutput->RegisterEventSink(pSampleSink));

// create a simple sample source
cObjectPtr<cOutputPin> pSampleSource = new cOutputPin();

if(chFrameId != ID_ARD_ACT_WD_ENABLE)
{
    __adtf_test_result(pSampleSource->Create("sampleSource", new
cMediaType(0,0,0,"tBoolSignalValue")));
}
else
{
    // we need another typ of samplesource
    __adtf_test_result(pSampleSource->Create("sampleSource", new cMediaType(0, 0,
0, "tJuryEmergencyStop")));
}

// connect the accelerate pin with th sample source
__adtf_test_result(pBoolPin->Connect(pSampleSource));

// set the filter state running
SET_STATE_RUNNING(pFilter);

// get the mediatype of the sample source
cObjectPtr<IMediaType> pType;
__adtf_test_result(pSampleSource->GetMediaType(&pType));

// get the type description
cObjectPtr<IMediaTypeDescription> pTypeDesc;
__adtf_test_result(pType->GetInterface(IID_ADTF_MEDIA_TYPE_DESCRIPTION,
(tVoid**)&pTypeDesc));

// get the serializer from description to get the deserialized size
cObjectPtr<IMediaSerializer> pSerializer;
__adtf_test_result(pTypeDesc->GetMediaSampleSerializer(&pSerializer));
tInt nSize = pSerializer->GetDeserializedSize();
pSerializer = NULL;

tUInt32 ui32MaxLoopCount = 50;

// send some samples
for (tUInt32 nIdx = 0; nIdx < ui32MaxLoopCount; nIdx++)
{
    // create and allocate the sample
    cObjectPtr<IMediaSample> pSample;
    __adtf_test_result(_runtime->CreateInstance(OID_ADTF_MEDIA_SAMPLE,
IID_ADTF_MEDIA_SAMPLE, (tVoid**)&pSample));
```

```

__adtf_test_result(pSample->AllocBuffer(nSize));

// get the coder
cObjectPtr<IMediaCoder> pCoder;
__adtf_test_result(pTypeDesc->WriteLock(pSample, &pCoder));

// set the value
tBool bValue = tFalse;

// send every the first and then every second time on signal
if(!(nIdx % 2))bValue = tTrue;

// use the coder to set the value
if(chFrameId != ID_ARD_ACT_WD_ENABLE)
{
    __adtf_test_result(pCoder->Set("bValue", &bValue));
}
else
{
    __adtf_test_result(pCoder->Set("bEmergencyStop", (tVoid*)&bValue));
}

// unlock the coder
__adtf_test_result(pTypeDesc->Unlock(pCoder));
// set the sample time (in this case the time doesn't matter and must not be
the stream time)
pSample->SetTime(nIdx);
// transmit the media sample
__adtf_test_result(pSampleSource->Transmit(pSample));
}

// clean up
__adtf_test_result(pBoolPin->Disconnect(pSampleSource));
__adtf_test_result(pOutput->UnregisterEventSink(pSampleSink));

// now check the received data
// first check for the right number of samples
if (strInputPinName == "headLightEnabled")
{
    // by sending headLight data the aktors filter automatically sends head and
    backlight data so the received samples have to be twice as much as the transmitted
    __adtf_test_ext(pSampleSink->m_ui32SampleCount == ui32MaxLoopCount * 2,
    cString::Format("Sample count (%d) does not match (%d)", pSampleSink-
    >m_ui32SampleCount, ui32MaxLoopCount * 2));
}
else if(strInputPinName == "Watchdog_Alive_Flag")
{
    // only enabled watchdog signals with the value true shall be send to the
    arduino, so only have of the send samples should be received
    __adtf_test_ext(pSampleSink->m_ui32SampleCount == ui32MaxLoopCount / 2,
    cString::Format("Sample count (%d) does not match (%d)", pSampleSink-
    >m_ui32SampleCount, ui32MaxLoopCount / 2));
}
else if(chFrameId == ID_ARD_ACT_WD_ENABLE)
{
    // only if the emergency stop value is true, the filter will send samples BUT
    the filter transmits a frame for the relais and a frame for watchdog
    // because we alternating the value this ends up in the same sample count

```

```

    __adtf_test_ext(pSampleSink->m_ui32SampleCount == ui32MaxLoopCount,
cString::Format("Sample count (%d) does not match (%d)", pSampleSink-
>m_ui32SampleCount, ui32MaxLoopCount / 2));
    }
    else
    {
        // every send sample should generate one received sample
        __adtf_test_ext(pSampleSink->m_ui32SampleCount == ui32MaxLoopCount,
cString::Format("Sample count (%d) does not match (%d)", pSampleSink-
>m_ui32SampleCount, ui32MaxLoopCount));
    }
    // get description manager
    cObjectPtr<IMediaDescriptionManager> pDescManager;
    __adtf_test_result(_runtime-
>GetObject(OID_ADTF_MEDIA_DESCRIPTION_MANAGER, IID_ADTF_MEDIA_DESCRIPTION_MANAGER, (tVoi
d**) &pDescManager));

    // get media description for tArduinoData
    tChar const * strDescArduino = pDescManager->GetMediaDescription("tArduinoData");
    __adtf_test_pointer(strDescArduino);

    // create mediatype for coder
    cObjectPtr<IMediaType> pTypeArduinoData = new cMediaType(0, 0, 0, "tArduinoData",
strDescArduino, IMediaDescription::MDF_DDL_DEFAULT_VERSION);

    // get the mediatype description from the mediatype
    cObjectPtr<IMediaTypeDescription> m_pCoderDescArduinoData;
    __adtf_test_result(pTypeArduinoData->GetInterface(IID_ADTF_MEDIA_TYPE_DESCRIPTION,
(tVoid**) &m_pCoderDescArduinoData));

    // loop over the received data
    for(tSize nIdx = 0; nIdx < pSampleSink->m_vecMediaSamples.size(); ++nIdx)
    {
        // get the coder //Search Dummy
        cObjectPtr<IMediaCoder> pCoder;
        m_pCoderDescArduinoData->Lock(pSampleSink->m_vecMediaSamples[nIdx], &pCoder);

        // init the values
        tUInt8 ui8SOF = 0;
        tUInt8 ui8Id = 0;
        tUInt32 ui32ArduinoTimestamp = 0;
        tUInt8 ui8DataLength = 0;
        tUInt8 achFrameData[25];
        cMemoryBlock::MemSet(achFrameData, 0x00, sizeof(achFrameData));

        // get values from coder
        pCoder->Get("ui8SOF", (tVoid*)&ui8SOF);
        pCoder->Get("ui8ID", (tVoid*)&ui8Id);
        pCoder->Get("ui32ArduinoTimestamp", (tVoid*)&(ui32ArduinoTimestamp));

        pCoder->Get("ui8DataLength", (tVoid*)&ui8DataLength);
        pCoder->Get("ui8Data", (tVoid*)&(achFrameData));

        // unlock
        m_pCoderDescArduinoData->Unlock(pCoder);

        // test the received data // Search Dummy
        // SOF
        __adtf_test(ui8SOF == ID_ARD_SOF);
        // Frame ID only in the emergency test there will be samples send with
different IDs
        if(chFrameId != ID_ARD_ACT_WD_ENABLE)

```

```
{
    __adtf_test(ui8Id == chFrameId);
}
// timestamp is always 0 in this filter
__adtf_test(ui32ArduinoTimestamp == 0);
// datalength has to be 2 because of the light ID for watchdog only 1
if(chFrameId == ID_ARD_ACT_WD_TOGGLE || chFrameId == ID_ARD_ACT_WD_ENABLE)
{
    __adtf_test(ui8DataLength == 1);
}
else
{
    __adtf_test(ui8DataLength == 2);
}

// Test the data of the protocol
if (strInputPinName == "headLightEnabled")
{
    // the aktors filter sends two samples for head and back light so they
    have to be checked seperatedly
    if(nIdx == 0 || !(nIdx % 4))
    {
        // the first sample sends on to the head light
        __adtf_test(achFrameData[0] == ID_ARD_ACT_LIGHT_DATA_HEAD);
        __adtf_test(achFrameData[1] == tUInt8(tTrue));
    }
    else if(nIdx == 1 || !((nIdx-1) % 4))
    {
        // the second sample sends on to the back light
        __adtf_test(achFrameData[0] == ID_ARD_ACT_LIGHT_DATA_BACK);
        __adtf_test(achFrameData[1] == tUInt8(tTrue));
    }
    else if(nIdx == 2 || !((nIdx-2) % 4))
    {
        // the third sample sends off to the front light
        __adtf_test(achFrameData[0] == ID_ARD_ACT_LIGHT_DATA_HEAD);
        __adtf_test(achFrameData[1] == tUInt8(tFalse));
    }
    else if(nIdx == 3 || !((nIdx-3) % 4))
    {
        // the fourth sample sends off to the back light
        __adtf_test(achFrameData[0] == ID_ARD_ACT_LIGHT_DATA_BACK);
        __adtf_test(achFrameData[1] == tUInt8(tFalse));
    }
}
else if (strInputPinName == "brakeLightEnabled")
{
    // First on command then off command for brake light
    if(!(nIdx % 2))
    {
        __adtf_test(achFrameData[0] == ID_ARD_ACT_LIGHT_DATA_BRAKE);
        __adtf_test(achFrameData[1] == tUInt8(tTrue));
    }
    if(nIdx % 2)
    {
        __adtf_test(achFrameData[0] == ID_ARD_ACT_LIGHT_DATA_BRAKE);
        __adtf_test(achFrameData[1] == tUInt8(tFalse));
    }
}
else if (strInputPinName == "turnSignalLeftEnabled")
{
    // First on command then off command for turn left signal
```



```

    if(!(nIdx % 2))
    {
        __adtf_test(achFrameData[0] == ID_ARD_ACT_LIGHT_DATA_TURNLEFT);
        __adtf_test(achFrameData[1] == tUInt8(tTrue));
    }
    if(nIdx % 2)
    {
        __adtf_test(achFrameData[0] == ID_ARD_ACT_LIGHT_DATA_TURNLEFT);
        __adtf_test(achFrameData[1] == tUInt8(tFalse));
    }
}
else if (strInputPinName == "turnSignalRightEnabled")
{
    // First on command then off command for turn right signal
    if(!(nIdx % 2))
    {
        __adtf_test(achFrameData[0] == ID_ARD_ACT_LIGHT_DATA_TURNRIGHT);
        __adtf_test(achFrameData[1] == tUInt8(tTrue));
    }
    if(nIdx % 2)
    {
        __adtf_test(achFrameData[0] == ID_ARD_ACT_LIGHT_DATA_TURNRIGHT);
        __adtf_test(achFrameData[1] == tUInt8(tFalse));
    }
}
else if (strInputPinName == "reverseLightsEnabled")
{
    // First on command then off command for reverse light
    if(!(nIdx % 2))
    {
        __adtf_test(achFrameData[0] == ID_ARD_ACT_LIGHT_DATA_REVERSE);
        __adtf_test(achFrameData[1] == tUInt8(tTrue));
    }
    if(nIdx % 2)
    {
        __adtf_test(achFrameData[0] == ID_ARD_ACT_LIGHT_DATA_REVERSE);
        __adtf_test(achFrameData[1] == tUInt8(tFalse));
    }
}
else if (strInputPinName == "Watchdog_Alive_Flag")
{
    // Only true commands are send as watchdog signal
    __adtf_test(achFrameData[0] == tUInt8(tTrue));
}
else if (strInputPinName == "Emergency_Stop")
{
    // First Motor relais will be reseted then the watchdog will be killed
    if(!(nIdx % 2))
    {
        __adtf_test(achFrameData[0] == 0x00);
        __adtf_test(ui8Id == ID_ARD_ACT_MOT_RELAIS);
    }
    if(nIdx % 2){
        __adtf_test(achFrameData[0] == 0x00);
        __adtf_test(ui8Id == ID_ARD_ACT_WD_ENABLE);
    }
}
}

// shutdown the filter
SET_STATE_SHUTDOWN(pFilter);
}

```

```
// Definition of the Test cases
```

```
DEFINE_TEST(cTesterAADCAktors,  
            TestAcceleration,  
            "1.1",  
            "TestAcceleration",  
            "This test makes sure, that the 'accelerate' pin is present. After that  
some acceleration data"\  
            "will be send to the filter and the filter output will be compared to some  
reference data.",  
            "Pin can be found and the output data are as expected.",  
            "See above",  
            "none",  
            "The filter must be able to receive acceleration data and convert them  
into arduino data.",  
            "automatic")  
{  
    // accelerate pin with expected frameID  
    return DoActuatorTest("accelerate", ID_ARD_ACT_ACCEL_SERVO);  
}
```

```
DEFINE_TEST(cTesterAADCAktors,  
            TestSteeringAngle,  
            "1.2",  
            "TestSteeringAngle",  
            "This test makes sure, that the 'steerAngle' pin is present. After that  
some steering data"\  
            "will be send to the filter and the filter output will be compared to some  
reference data.",  
            "Pin can be found and the output data are as expected.",  
            "See above",  
            "none",  
            "The filter must be able to receive steering data and convert them into  
arduino data.",  
            "automatic")  
{  
    // steering angle pin with expected frameID  
    return DoActuatorTest("steerAngle", ID_ARD_ACT_STEER_ANGLE);  
}
```

```
// helper function for the light tests
```

```
tTestResult LightTests(){  
    // Do the light test for every available Light and test the received samples  
    DoActuatorBoolTest("headLightEnabled", ID_ARD_ACT_LIGHT_DATA);  
    DoActuatorBoolTest("brakeLightEnabled", ID_ARD_ACT_LIGHT_DATA);  
    DoActuatorBoolTest("turnSignalLeftEnabled", ID_ARD_ACT_LIGHT_DATA);  
    DoActuatorBoolTest("turnSignalRightEnabled", ID_ARD_ACT_LIGHT_DATA);  
    DoActuatorBoolTest("reverseLightsEnabled", ID_ARD_ACT_LIGHT_DATA);  
}
```

```
DEFINE_TEST(cTesterAADCAktors,  
            TestLights,  
            "1.3",  
            "TestLights",  
            "This test makes sure, that the 'LightPins' pins are present. After that  
some Lightsignals data"\  
            "will be send to the filter and the filter output will be compared to some  
reference data.",
```

```

        "Pin can be found and the output data are as expected.",
        "See above",
        "none",
        "The filter must be able to receive Light data and convert them into
arduino data.",
        "automatic")
{
    // Specific light test so no parameters needed
    return LightTests();
}

DEFINE_TEST(cTesterAADCAktors,
            TestWatchdog,
            "1.4",
            "TestWatchdog",
            "This test makes sure, that the 'Watchdog' pin is present. After that some
Watchdog data"\
            "will be send to the filter and the filter output will be compared to some
reference data.",
            "Pin can be found and the output data are as expected.",
            "See above",
            "none",
            "The filter must be able to receive Watchdog data and convert them into
arduino data.",
            "automatic")
{
    // Do the BoolTest with the Watchdog pin and test the received samples
    return DoActuatorBoolTest("Watchdog_Alive_Flag", ID_ARD_ACT_WD_TOGGLE);
}

DEFINE_TEST(cTesterAADCAktors,
            TestEmergencyStop,
            "1.5",
            "TestEmergencyStop",
            "This test makes sure, that the 'Emergency' pin is present. After that
some Emergency data"\
            "will be send to the filter and the filter output will be compared to some
reference data.",
            "Pin can be found and the output data are as expected.",
            "See above",
            "none",
            "The filter must be able to receive Emergency data and convert them into
arduino data.",
            "automatic")
{
    // Do the BoolTest with the Watchdog pin and test the received samples
    return DoActuatorBoolTest("Emergency_Stop", ID_ARD_ACT_WD_ENABLE);
}

```

Programm-Code: Integrationstest

```

/**
 *
 * AADC Arduino integration filter tests
 *
 * @file
 * Copyright &copy; Audi Electronics Venture GmbH. All rights reserved
 *
 * $Author: VG8D3AW $
 * $Date: 2013-02-06 16:30:41 +0100 (Mi, 06. Feb 2013) $

```

```
* $Revision: 18162 $
*
* @remarks
*
*/

#include "stdafx.h"
#include "tester_aadc_aktors_comm_sensors.h"

IMPLEMENT_TESTER_CLASS(cTesterAADCAktorsCommSensors,
    "1",
    "AADC Arduino Aktors filter",
    "This test makes sure that the AADC Arduino Aktors filter works as
expected",
    "");

/*
Setup for this test
*/
void cTesterAADCAktorsCommSensors::setUp()
{
    // create the environment
    SERVICE_ENV_SETUP;

    // minimum needed ADTF services
    SERVICE_ENV_ADD_PLUGIN("adtf_clock.srv");
    SERVICE_ENV_ADD_PLUGIN("adtf_kernel.srv");
    SERVICE_ENV_ADD_PLUGIN("adtf_sample_pool.srv");
    SERVICE_ENV_ADD_PLUGIN("adtf_namespace.srv");
    SERVICE_ENV_ADD_PLUGIN("adtf_media_description.srv");

    // register services
    SERVICE_ENV_REGISTER_SERVICE(OID_ADTF_REFERENCE_CLOCK, "referenceclock",
IRuntime::RL_Kernel);
    SERVICE_ENV_REGISTER_SERVICE(OID_ADTF_KERNEL, "kernel", IRuntime::RL_System);
    SERVICE_ENV_REGISTER_SERVICE(OID_ADTF_SAMPLE_POOL, "samplepool",
IRuntime::RL_System);
    SERVICE_ENV_REGISTER_SERVICE(OID_ADTF_NAMESPACE, "namespace",
IRuntime::RL_System);
    SERVICE_ENV_REGISTER_SERVICE(OID_ADTF_MEDIA_DESCRIPTION_MANAGER, "mediadesc",
IRuntime::RL_System);

    // add filter plugin
    FILTER_ENV_ADD_PLUGIN("aadc_aktors.plb");
    FILTER_ENV_ADD_PLUGIN("aadc_arduinoCommunication.plb");
    FILTER_ENV_ADD_PLUGIN("aadc_sensors.plb");
    FILTER_ENV_ADD_PLUGIN("aadc_watchdogTrigger.plb");

    // set runlevel system
    SERVICE_ENV_SET_RUNLEVEL(IRuntime::RL_System);

    // set the path to the MediaDescription-Files
    // (relative to the current working dir which is the tester dir)
    cFilename strDescrFileADTF =
"..\\..\\..\\src\\adtfBase\\AADC_ADTF_BaseFilters\\description\\aadc.description";
    cFilename strDescrFile =
"..\\..\\..\\src\\adtfBase\\AADC_ADTF_BaseFilters\\description\\aadc.description";

    // create absolute path
    strDescrFileADTF =
strDescrFileADTF.CreateAbsolutePath(cFileSystem::GetCurDirectory());
```

```

    strDescrFile = strDescrFile.CreateAbsolutePath(cFileSystem::GetCurDirectory());

    // get the media description manager
    cObjectPtr<IMediaDescriptionManager> pMediaDesc;
    __adtf_test_result_ext(_runtime->GetObject(OID_ADTF_MEDIA_DESCRIPTION_MANAGER,
    IID_ADTF_MEDIA_DESCRIPTION_MANAGER, (tVoid**) &pMediaDesc),
        "unable to get media description manager");

    // get the config from the media description manager
    cObjectPtr<IConfiguration> pConfig;
    __adtf_test_result(pMediaDesc->GetInterface(IID_ADTF_CONFIGURATION, (tVoid**)
    &pConfig));

    // set the path to the description files
    __adtf_test_result(pConfig->SetPropertyStr("media_description_files",
    cString::Format("%s;%s", strDescrFileADTF.GetPtr(), strDescrFile.GetPtr())));

    // check if the media description manager loads the description correctly
    __adtf_test(pMediaDesc->GetMediaDescription("tArduinoData") != NULL);

    // set runlevel application
    SERVICE_ENV_SET_RUNLEVEL(IRuntime::RL_Application);
}

/*
Tear down for this test
*/
void cTesterAADCAktorsCommSensors::tearDown()
{
    // give the kernel some time to quit its threads
    //cSystem::Sleep(200000);
    FILTER_ENV_TEAR_DOWN;
}

/*****
/* This is just a helper class to receive data from output pins.
*/
*****/
class cMediaSampleSink: public IPinEventSink
{
    // ucom helper macro
    UCOM_OBJECT_IMPL(IID_ADTF_PIN_EVENT_SINK, adtf::IPinEventSink);
public:
    // constructor
    cMediaSampleSink(): m_ui32SampleCount(0)
    {
    }

    // destructor
    virtual ~cMediaSampleSink()
    {
    }

    // members to count and hold the received media samples
    tUInt32 m_ui32SampleCount;
    std::vector<cObjectPtr<IMediaSample> > m_vecMediaSamples;

public:
    tResult OnPinEvent(IPin* pSource, tInt nEventCode, tInt nParam1, tInt nParam2,
    IMediaSample *pMediaSample)
    {
        if (nEventCode == IPinEventSink::PE_MediaSampleTransmitted)

```

```

    {
        // count the samples
        ++m_ui32SampleCount;
        // push sample into vector for later compare
        m_vecMediaSamples.push_back(pMediaSample);
    }

    RETURN_NOERROR;
}

};

/*****
/* These are just helper functions to send Media Samples.
*/
*****/
tTestResult transmitSignalMediaSample(cOutputPin *pSampleSource, const tFloat32
f32Value )
{
    // get the mediatype of the sample source
    cObjectPtr<IMediaType> pType;
    if(IS_FAILED(pSampleSource->GetMediaType(&pType)))
    {
        __adtf_test_ext(tFalse, "unable to get mediatype");
    }

    // get the type description
    cObjectPtr<IMediaTypeDescription> pTypeDesc;
    pType->GetInterface(IID_ADTF_MEDIA_TYPE_DESCRIPTION, (tVoid**)&pTypeDesc);

    // get the serializer from description to get the deserialized size
    cObjectPtr<IMediaSerializer> pSerializer;
    pTypeDesc->GetMediaSampleSerializer(&pSerializer);
    tInt nSize = pSerializer->GetDeserializedSize();
    pSerializer = NULL;

    // create and allocate the sample
    cObjectPtr<IMediaSample> pSample;
    _runtime->CreateInstance(OID_ADTF_MEDIA_SAMPLE, IID_ADTF_MEDIA_SAMPLE,
(tVoid**)&pSample);
    pSample->AllocBuffer(nSize);

    // get the coder
    cObjectPtr<IMediaCoder> pCoder;
    pTypeDesc->WriteLock(pSample, &pCoder);

    // use the coder to set the value
    pCoder->Set("f32Value", &f32Value);

    // unlock the coder
    pTypeDesc->Unlock(pCoder);
    // set the sample time (in this case the time doesn't matter and must not be the
stream time)
    pSample->SetTime(0);
    // transmit the media sample
    //__adtf_test_result(pSampleSource->Transmit(pSample));
    if(IS_FAILED(pSampleSource->Transmit(pSample)))
    {
        __adtf_test_ext(tFalse, "unable to send media sample");
    }
}
}

```

```
tTestResult transmitBoolMediaSample(cOutputPin *pSampleSource, const tBool bValue)
{
    // get the mediatype of the sample source
    cObjectPtr<IMediaType> pType;
    if(IS_FAILED(pSampleSource->GetMediaType(&pType)))
    {
        __adtf_test_ext(tFalse, "unable to get mediatype");
    }

    // get the type description
    cObjectPtr<IMediaTypeDescription> pTypeDesc;
    pType->GetInterface(IID_ADTF_MEDIA_TYPE_DESCRIPTION, (tVoid**)&pTypeDesc);

    // get the serializer from description to get the deserialized size
    cObjectPtr<IMediaSerializer> pSerializer;
    pTypeDesc->GetMediaSampleSerializer(&pSerializer);
    tInt nSize = pSerializer->GetDeserializedSize();
    pSerializer = NULL;

    // create and allocate the sample
    cObjectPtr<IMediaSample> pSample;
    _runtime->CreateInstance(OID_ADTF_MEDIA_SAMPLE, IID_ADTF_MEDIA_SAMPLE,
(tVoid**)&pSample);
    pSample->AllocBuffer(nSize);

    // get the coder
    cObjectPtr<IMediaCoder> pCoder;
    pTypeDesc->WriteLock(pSample, &pCoder);

    // use the coder to write the value in the sample
    pCoder->Set("bValue", &bValue);

    // unlock the coder
    pTypeDesc->Unlock(pCoder);
    // set the sample time (in this case the time doesn't matter and must not be the
stream time)
    pSample->SetTime(0);
    // transmit the media sample
    //__adtf_test_result(pSampleSource->Transmit(pSample));
    if(IS_FAILED(pSampleSource->Transmit(pSample)))
    {
        __adtf_test_ext(tFalse, "unable to send media sample");
    }
}

tTestResult transmitEmergencyMediaSample(cOutputPin *pSampleSource, const tBool
bValue)
{
    // get the mediatype of the sample source
    cObjectPtr<IMediaType> pType;
    if(IS_FAILED(pSampleSource->GetMediaType(&pType)))
    {
        __adtf_test_ext(tFalse, "unable to get mediatype");
    }

    // get the type description
    cObjectPtr<IMediaTypeDescription> pTypeDesc;
    pType->GetInterface(IID_ADTF_MEDIA_TYPE_DESCRIPTION, (tVoid**)&pTypeDesc);

    // get the serializer from description to get the deserialized size
    cObjectPtr<IMediaSerializer> pSerializer;
    pTypeDesc->GetMediaSampleSerializer(&pSerializer);
```

```

tInt nSize = pSerializer->GetDeserializedSize();
pSerializer = NULL;

// create and allocate the sample
cObjectPtr<IMediaSample> pSample;
_runtime->CreateInstance(OID_ADTF_MEDIA_SAMPLE, IID_ADTF_MEDIA_SAMPLE,
(tVoid**)&pSample);
pSample->AllocBuffer(nSize);

// get the coder
cObjectPtr<IMediaCoder> pCoder;
pTypeDesc->WriteLock(pSample, &pCoder);

// use the coder to write the value in the sample
pCoder->Set("bEmergencyStop", (tVoid*)&bValue);

// unlock the coder
pTypeDesc->Unlock(pCoder);
// set the sample time (in this case the time doesn't matter and must not be the
stream time)
pSample->SetTime(0);
// transmit the media sample
//__adtf_test_result(pSampleSource->Transmit(pSample));
if(IS_FAILED(pSampleSource->Transmit(pSample)))
{
    __adtf_test_ext(tFalse, "unable to send media sample");
}
}

/*****
/* These are just helper functions to evaluate Media Samples.
*/
*****/
tTestResult evaluateSteerAngle(cMediaSampleSink *pSampleSink)
{
    // get description manager
    cObjectPtr<IMediaDescriptionManager> pDescManager;
    __adtf_test_result(_runtime-
>GetObject(OID_ADTF_MEDIA_DESCRIPTION_MANAGER, IID_ADTF_MEDIA_DESCRIPTION_MANAGER, (tVoi
d**)&pDescManager));

    // get media description for tArduinoData
    tChar const * strDescSteerAngle = pDescManager-
>GetMediaDescription("tSteeringAngleData");
    __adtf_test_pointer(strDescSteerAngle);

    // create mediatype for coder
    cObjectPtr<IMediaType> pTypeSteerAngleData = new cMediaType(0, 0, 0,
    "tSteeringAngleData", strDescSteerAngle, IMediaDescription::MDF_DDL_DEFAULT_VERSION);

    // get the mediatype description from the mediatype
    cObjectPtr<IMediaTypeDescription> m_pCoderDescSteerAngleData;
    __adtf_test_result(pTypeSteerAngleData-
>GetInterface(IID_ADTF_MEDIA_TYPE_DESCRIPTION, (tVoid**)&m_pCoderDescSteerAngleData));

    // received sensor data should show the actor ramp
    // now check the received data

```



```
// Define the Ranges of the ramp and a tolerance Value to ignore single value
jumps
tUInt16 ui16LowerRampRange = 338;
tUInt16 ui16UpperRampRange = 520;
tUInt16 ui16ToleranceValue = 20;

// Init some values we need to recognize the ramp
tUInt16 ui16SteerAngleBefore = 0;
tBool bStartOfRamp = tFalse;
tInt8 i8RampSegment = 10;
tUInt16 ui16StartAngle = 0;

// loop over the received data
for(tSize nIdx = 0; nIdx < pSampleSink->m_vecMediaSamples.size(); ++nIdx)
{
    // get the coder
    cObjectPtr<IMediaCoder> pCoder;
    m_pCoderDescSteerAngleData->Lock(pSampleSink->m_vecMediaSamples[nIdx],
    &pCoder);

    // init the values
    tUInt16 ui16Angle = 0;
    tUInt32 ui32ArduinoTimestamp = 0;

    // get values from coder
    pCoder->Get("ui16Angle", (tVoid*)&ui16Angle);
    pCoder->Get("ui32ArduinoTimestamp", (tVoid*)&ui32ArduinoTimestamp);

    // unlock
    m_pCoderDescSteerAngleData->Unlock(pCoder);

    // test the received data
    __adtf_test(ui32ArduinoTimestamp != 0);

    __adtf_test_log(cString::Format("Steerangle: %u", ui16Angle));
    __adtf_test_log(cString::Format("Rampsegment: %d", i8RampSegment));

    // Test for the ramp
    // recognize initial state
    if(ui16Angle == 0)
    {
        i8RampSegment = 0;
    }
    // recognize the start of the ramp
    if(!bStartOfRamp && ui16Angle > 0 && i8RampSegment == 0)
    {
        ui16StartAngle = ui16Angle;
        bStartOfRamp = tTrue;
        i8RampSegment++;
    }
    // recognize the first turn
    if(bStartOfRamp && ui16Angle > ui16UpperRampRange && i8RampSegment == 1)
    {
        i8RampSegment++;
        bStartOfRamp = tFalse;
    }
    // recognize the second turn
    if(bStartOfRamp && ui16Angle < ui16LowerRampRange && i8RampSegment == 2)
    {
        i8RampSegment++;
        bStartOfRamp = tFalse;
    }
}
```

```

    // recognize the end of the ramp
    if(bStartOfRamp && ui16Angle > ui16StartAngle - ui16ToleranceValue &&
i8RampSegment == 3)
    {
        i8RampSegment++;
        bStartOfRamp = tFalse;
    }
    // start the ramp recognition again after a turn if the values are in the
range
    if(!bStartOfRamp && ui16Angle < ui16UpperRampRange && ui16Angle >
ui16LowerRampRange && i8RampSegment < 4)
    {
        bStartOfRamp = tTrue;
    }

    // test the values in the rising ramps
    if(bStartOfRamp && (i8RampSegment == 1 || i8RampSegment == 3))
    {
        __adtf_test(ui16Angle > ui16SteerAngleBefore - ui16ToleranceValue);
    }
    // test the values of the falling ramp
    if(bStartOfRamp && i8RampSegment == 2)
    {
        __adtf_test(ui16Angle < ui16SteerAngleBefore + ui16ToleranceValue);
    }

    // Go to the next Values
    ui16SteerAngleBefore = ui16Angle;
}
// Check if the ramp was recognized
__adtf_test(i8RampSegment == 4);
}
tTestResult evaluateRPM(cMediaSampleSink *pSampleSink, const tChar chFrameId)
{
    // get description manager
    cObjectPtr<IMediaDescriptionManager> pDescManager;
    __adtf_test_result(_runtime-
>GetObject(OID_ADTF_MEDIA_DESCRIPTION_MANAGER, IID_ADTF_MEDIA_DESCRIPTION_MANAGER, (tVoi
d**) &pDescManager));

    // get media description for tArduinoData
    tChar const * strDescRPM = pDescManager->GetMediaDescription("tWheelEncoderData");
    __adtf_test_pointer(strDescRPM);

    // create mediatype for coder
    cObjectPtr<IMediaType> pTypeSteerAngleData = new cMediaType(0, 0, 0,
"tWheelEncoderData", strDescRPM, IMediaDescription::MDF_DDL_DEFAULT_VERSION);

    // get the mediatype description from the mediatype
    cObjectPtr<IMediaTypeDescription> m_pCoderDescSteerAngleData;
    __adtf_test_result(pTypeSteerAngleData-
>GetInterface(IID_ADTF_MEDIA_TYPE_DESCRIPTION, (tVoid**) &m_pCoderDescSteerAngleData));

    // received sensor data should show the actor ramp
    // now check the received data

    // Define the Ranges of the ramp and a tolerance Value to ignore single value
jumps
    tUInt16 ui16LowerRampRange = 3;
    tUInt16 ui16UpperRampRange = 21;
    tUInt16 ui16ToleranceValue = 4;

```

```

// Init some values we need to recognize the ramp
tUInt32 ui32RPMDiffBefore = 0;
tUInt32 ui32RPMBefore = 0;
tBool bStartOfRamp = tFalse;
tInt8 i8RampSegment = 0;
tUInt16 ui16EndRPMDiff = 12;

// for the combination test take smaller values
if(chFrameId == 0xff)
{
    ui16UpperRampRange = 11;
    ui16EndRPMDiff = 3;
}

// loop over the received data
for(tSize nIdx = 0; nIdx < pSampleSink->m_vecMediaSamples.size(); ++nIdx)
{
    // only take every third value to receive stronger differences
    if(nIdx % 2)
    {
        // get the coder
        cObjectPtr<IMediaCoder> pCoder;
        m_pCoderDescSteerAngleData->Lock(pSampleSink->m_vecMediaSamples[nIdx],
&pCoder);

        // init the values
        tUInt32 ui32RPMLeft = 0;
        tUInt32 ui32RPMRight = 0;
        tUInt32 ui32ArduinoTimestamp = 0;

        // get values from coder
        pCoder->Get("ui32LeftWheel", (tVoid*)&ui32RPMLeft);
        pCoder->Get("ui32RightWheel", (tVoid*)&ui32RPMRight);
        pCoder->Get("ui32ArduinoTimestamp", (tVoid*)&ui32ArduinoTimestamp);

        // unlock
        m_pCoderDescSteerAngleData->Unlock(pCoder);

        // test the received data
        __adtf_test(ui32ArduinoTimestamp!=0);

        // build the average between left and right wheel
        tUInt32 ui32RPM = (ui32RPMLeft + ui32RPMRight)/2;

        // weigh the new value statistically with the old one to filter out peak
values
        ui32RPM = (ui32RPMBefore + ui32RPM)/2;

        __adtf_test_log(cString::Format("RPM: %d", ui32RPM - ui32RPMBefore));
        __adtf_test_log(cString::Format("Rampsegment: %d", i8RampSegment));

        // Test for the ramp
        // recognize the start of the ramp
        if(!bStartOfRamp && ui32RPM - ui32RPMBefore > 0 && i8RampSegment == 0)
        {
            bStartOfRamp = tTrue;
            i8RampSegment++;
        }
        // recognize the first turn
        if(bStartOfRamp && ui32RPM - ui32RPMBefore > ui16UpperRampRange &&
(i8RampSegment == 1 || i8RampSegment == 3))
        {

```

```

        i8RampSegment++;
        bStartOfRamp = tFalse;
    }
    // recognize the second turn
    if(bStartOfRamp && ui32RPM - ui32RPMBefore < ui16LowerRampRange &&
i8RampSegment == 2)
    {
        i8RampSegment++;
        bStartOfRamp = tFalse;
    }
    // recognize the end of the ramp
    if(bStartOfRamp && ui32RPM - ui32RPMBefore < ui16EndRPMDiff &&
i8RampSegment == 4)
    {
        i8RampSegment++;
        bStartOfRamp = tFalse;
    }
    // start the ramp recognition again after a turn if the values are in the
range
    if(!bStartOfRamp && ui32RPM - ui32RPMBefore < ui16UpperRampRange &&
ui32RPM - ui32RPMBefore > ui16LowerRampRange && i8RampSegment < 4)
    {
        bStartOfRamp = tTrue;
    }

    // test the values in the rising ramps
    if(bStartOfRamp && (i8RampSegment == 1 || i8RampSegment == 3))
    {
        __adtf_test(ui32RPM - ui32RPMBefore + ui16ToleranceValue >
ui32RPMDiffBefore);
    }
    // test the values of the falling ramp
    if(bStartOfRamp && i8RampSegment == 2)
    {
        __adtf_test(ui32RPM - ui32RPMBefore < ui32RPMDiffBefore +
ui16ToleranceValue);
    }

    // Go to the next Values
    ui32RPMDiffBefore = ui32RPM - ui32RPMBefore;
    ui32RPMBefore = ui32RPM;
}
}

// Check if the ramp was recognized
__adtf_test(i8RampSegment == 4);
}

tTestResult DoActuatorCommSensorTest(const tChar chFrameId, tInt nSleep = 0)
{
    // initialize filter
    INIT_FILTER(pAktFilter, pAktFilterConfig, "adtf.aadc.aktors");
    INIT_FILTER(pCommFilter, pCommFilterConfig, "adtf.aadc.arduinoCOM");
    INIT_FILTER(pSensFilter, pSensFilterConfig, "adtf.aadc.sensors");
    INIT_FILTER(pWDFilter, pWDFilterConfig, "adtf.aadc.watchdogGuard");

#ifdef WIN32 // WIN32 is also defined on WIN64
    // set the COM-Port on the Windows machine (on ODR0ID we use the default setting
from filter)
    __adtf_test_result(pCommFilterConfig->SetPropertyStr("COM Port", " \\.\\COM9"));
#endif

```

```
// set filter to state ready
SET_STATE_READY(pAktFilter);
SET_STATE_READY(pCommFilter);
SET_STATE_READY(pSensFilter);
SET_STATE_READY(pWDFilter);

// get the output pin of the WD Filter
cObjectPtr<IPin> pWDOutPin;
__adtf_test_result_ext(pWDFilter->FindPin("WatchdogAliveSignal", IPin::PD_Output,
&pWDOutPin), "Unable to find pin WatchdogAliveSignal");

// get the input pins of the Aktors Filter
cObjectPtr<IPin> pAccInPin;
__adtf_test_result_ext(pAktFilter->FindPin("accelerate", IPin::PD_Input,
&pAccInPin), "Unable to find input pin of Aktors filter");
cObjectPtr<IPin> pSteerAngleInPin;
__adtf_test_result_ext(pAktFilter->FindPin("steerAngle", IPin::PD_Input,
&pSteerAngleInPin), "Unable to find input pin of Aktors filter");
cObjectPtr<IPin> pHeadLightInPin;
__adtf_test_result_ext(pAktFilter->FindPin("headLightEnabled", IPin::PD_Input,
&pHeadLightInPin), "Unable to find input pin of Aktors filter");
cObjectPtr<IPin> pTurnLeftInPin;
__adtf_test_result_ext(pAktFilter->FindPin("turnSignalLeftEnabled",
IPin::PD_Input, &pTurnLeftInPin), "Unable to find input pin of Aktors filter");
cObjectPtr<IPin> pTurnRightInPin;
__adtf_test_result_ext(pAktFilter->FindPin("turnSignalRightEnabled",
IPin::PD_Input, &pTurnRightInPin), "Unable to find input pin of Aktors filter");
cObjectPtr<IPin> pBrakeLightInPin;
__adtf_test_result_ext(pAktFilter->FindPin("brakeLightEnabled", IPin::PD_Input,
&pBrakeLightInPin), "Unable to find input pin of Aktors filter");
cObjectPtr<IPin> pReverseLightInPin;
__adtf_test_result_ext(pAktFilter->FindPin("reverseLightsEnabled", IPin::PD_Input,
&pReverseLightInPin), "Unable to find input pin of Aktors filter");
cObjectPtr<IPin> pWDInPin;
__adtf_test_result_ext(pAktFilter->FindPin("Watchdog_Alive_Flag", IPin::PD_Input,
&pWDInPin), "Unable to find input pin of Aktors filter");
cObjectPtr<IPin> pEmergencyInPin;
__adtf_test_result_ext(pAktFilter->FindPin("Emergency_Stop", IPin::PD_Input,
&pEmergencyInPin), "Unable to find input pin of Aktors filter");

// get the output pin of the Aktors Filter
cObjectPtr<IPin> pArdAktOutPin;
__adtf_test_result_ext(pAktFilter->FindPin("ArduinoCOM_output", IPin::PD_Output,
&pArdAktOutPin), "Unable to find pin ArduinoCOM_output");

// get the input pin of the Communication Filter
cObjectPtr<IPin> pArdCommInPin;
__adtf_test_result_ext(pCommFilter->FindPin("COM_input", IPin::PD_Input,
&pArdCommInPin), "Unable to find input pin of Communication Filter");

// get the output pin of the Communication Filter
cObjectPtr<IPin> pArdCommOutPin;
__adtf_test_result_ext(pCommFilter->FindPin("COM_output", IPin::PD_Output,
&pArdCommOutPin), "Unable to find output pin of Communication Filter");

// get the input pin of the Sensors Filter
cObjectPtr<IPin> pArdSensInPin;
__adtf_test_result_ext(pSensFilter->FindPin("ArduinoCOM_input", IPin::PD_Input,
&pArdSensInPin), "Unable to find input pin of Sensors Filter");

// get the output pins of the Sensors Filter
```

```

    cObjectPtr<IPin> pArdSensUSOutPin;
    __adtf_test_result_ext(pSensFilter->FindPin("ultrasonic_sensors", IPin::PD_Output,
&pArdSensUSOutPin), "Unable to find output pin of Sensors Filter");
    cObjectPtr<IPin> pArdSensIROutPin;
    __adtf_test_result_ext(pSensFilter->FindPin("infrared_sensors", IPin::PD_Output,
&pArdSensIROutPin), "Unable to find output pin of Sensors Filter");
    cObjectPtr<IPin> pArdSensSteerAngleOutPin;
    __adtf_test_result_ext(pSensFilter->FindPin("steering_servo", IPin::PD_Output,
&pArdSensSteerAngleOutPin), "Unable to find output pin of Sensors Filter");
    cObjectPtr<IPin> pArdSensAccOutPin;
    __adtf_test_result_ext(pSensFilter->FindPin("accelerometer", IPin::PD_Output,
&pArdSensAccOutPin), "Unable to find output pin of Sensors Filter");
    cObjectPtr<IPin> pArdSensGyroOutPin;
    __adtf_test_result_ext(pSensFilter->FindPin("gyroscope", IPin::PD_Output,
&pArdSensGyroOutPin), "Unable to find output pin of Sensors Filter");
    cObjectPtr<IPin> pArdSensRPMOutPin;
    __adtf_test_result_ext(pSensFilter->FindPin("wheel_speed_sensor", IPin::PD_Output,
&pArdSensRPMOutPin), "Unable to find output pin of Sensors Filter");
    cObjectPtr<IPin> pArdSensVoltageOutPin;
    __adtf_test_result_ext(pSensFilter->FindPin("system_voltage", IPin::PD_Output,
&pArdSensVoltageOutPin), "Unable to find output pin of Sensors Filter");

    // create a simple sample source for every input pin
    cObjectPtr<cOutputPin> pAccSampleSource = new cOutputPin();
    __adtf_test_result(pAccSampleSource->Create("sampleSource", new
cMediaType(0,0,0,"tSignalValue"))));
    cObjectPtr<cOutputPin> pSteerAngleSampleSource = new cOutputPin();
    __adtf_test_result(pSteerAngleSampleSource->Create("sampleSource", new
cMediaType(0,0,0,"tSignalValue"))));
    cObjectPtr<cOutputPin> pHeadLightSampleSource = new cOutputPin();
    __adtf_test_result(pHeadLightSampleSource->Create("sampleSource", new
cMediaType(0,0,0,"tBoolSignalValue"))));
    cObjectPtr<cOutputPin> pTurnLeftSampleSource = new cOutputPin();
    __adtf_test_result(pTurnLeftSampleSource->Create("sampleSource", new
cMediaType(0,0,0,"tBoolSignalValue"))));
    cObjectPtr<cOutputPin> pTurnRightSampleSource = new cOutputPin();
    __adtf_test_result(pTurnRightSampleSource->Create("sampleSource", new
cMediaType(0,0,0,"tBoolSignalValue"))));
    cObjectPtr<cOutputPin> pBrakeLightSampleSource = new cOutputPin();
    __adtf_test_result(pBrakeLightSampleSource->Create("sampleSource", new
cMediaType(0,0,0,"tBoolSignalValue"))));
    cObjectPtr<cOutputPin> pReverseLightSampleSource = new cOutputPin();
    __adtf_test_result(pReverseLightSampleSource->Create("sampleSource", new
cMediaType(0,0,0,"tBoolSignalValue"))));
    cObjectPtr<cOutputPin> pEmergencySampleSource = new cOutputPin();
    __adtf_test_result(pEmergencySampleSource->Create("sampleSource", new
cMediaType(0,0,0,"tJuryEmergencyStop"))));

    // connect the input pins of the aktors filter with the sample sources
    __adtf_test_result(pAccInPin->Connect(pAccSampleSource));
    __adtf_test_result(pSteerAngleInPin->Connect(pSteerAngleSampleSource));
    __adtf_test_result(pHeadLightInPin->Connect(pHeadLightSampleSource));
    __adtf_test_result(pTurnLeftInPin->Connect(pTurnLeftSampleSource));
    __adtf_test_result(pTurnRightInPin->Connect(pTurnRightSampleSource));
    __adtf_test_result(pBrakeLightInPin->Connect(pBrakeLightSampleSource));
    __adtf_test_result(pReverseLightInPin->Connect(pReverseLightSampleSource));
    __adtf_test_result(pWDInPin->Connect(pWDOutPin));
    __adtf_test_result(pEmergencyInPin->Connect(pEmergencySampleSource));

    // connect the output pin of the Aktors Filter with the input pin of the
Communication Filter
    __adtf_test_result(pArdCommInPin->Connect(pArdAktOutPin));

```

```

// connect the output pin of the Communication Filter with the input pin of the
Sensors Filter
__adtf_test_result(pArdSensInPin->Connect(pArdCommOutPin));

// create a sample sink for every output of the sensors filter
cObjectPtr<cMediaSampleSink> pUSSampleSink = new cMediaSampleSink();
cObjectPtr<cMediaSampleSink> pIRSampleSink = new cMediaSampleSink();
cObjectPtr<cMediaSampleSink> pSteerAngleSampleSink = new cMediaSampleSink();
cObjectPtr<cMediaSampleSink> pAccSampleSink = new cMediaSampleSink();
cObjectPtr<cMediaSampleSink> pGyroSampleSink = new cMediaSampleSink();
cObjectPtr<cMediaSampleSink> pRPMSampleSink = new cMediaSampleSink();
cObjectPtr<cMediaSampleSink> pVoltageSampleSink = new cMediaSampleSink();

// connect the output pins of the sensors filter with the sample sinks
__adtf_test_result(pArdSensUSOutPin->RegisterEventSink(pUSSampleSink));
__adtf_test_result(pArdSensIROutPin->RegisterEventSink(pIRSampleSink));
__adtf_test_result(pArdSensAccOutPin->RegisterEventSink(pSteerAngleSampleSink));
__adtf_test_result(pArdSensUSOutPin->RegisterEventSink(pAccSampleSink));
__adtf_test_result(pArdSensGyroOutPin->RegisterEventSink(pGyroSampleSink));
__adtf_test_result(pArdSensRPMOutPin->RegisterEventSink(pRPMSampleSink));
__adtf_test_result(pArdSensVoltageOutPin->RegisterEventSink(pVoltageSampleSink));

// set the filter state running
SET_STATE_RUNNING(pAktFilter);
SET_STATE_RUNNING(pCommFilter);
SET_STATE_RUNNING(pSensFilter);
SET_STATE_RUNNING(pWDFilter);

// init the loop count
tUInt32 ui32MaxLoopCount = 0;
// init the wait time
tUInt32 ui32Waittime = 0;

#####
//Testing
#####

//#####
#####

// define the Test duration and the Waittime after Watchdog Signal is on
ui32MaxLoopCount = 250;
// Dafines the Time from when the Watchdog is active to when the first actuator
sample is sent. This is because of the initialisation of the motor controler
ui32Waittime = 5000;

#ifdef WIN32 // WIN32 is also defined on WIN64
// Because of the different execution time on windows and linux there have to be
different loop cycle numbers to simulate the same time
ui32MaxLoopCount = 1000;
ui32Waittime = 1000;
#endif

// init the data range
tFloat32 f32LowerRange = 0.0f;
tFloat32 f32UpperRange = 0.0f;

// define the data range
if(chFrameId == ID_ARD_ACT_STEER_ANGLE || chFrameId == 0xff)
{

```



```

    f32LowerRange = 65.0f;
    f32UpperRange = 125.0f;

}else if (chFrameId == ID_ARD_ACT_ACCEL_SERVO){
    f32LowerRange = 0.0f;
    f32UpperRange = 180.0f;
}

// define the LightCycle
// Number of lights
tFloat32 f32NumberOfLights = 5;
// Loopcount when the lights are activated
tUInt32 ui32NumberOfCycles = ui32MaxLoopCount/4;

// calculate the Start of each Lighttest
tFloat32 f32StartOfHeadlight = ui32NumberOfCycles / f32NumberOfLights * 0;
tFloat32 f32StartOfBrakelight = ui32NumberOfCycles / f32NumberOfLights * 1;
tFloat32 f32StartOfTurnleftlight = ui32NumberOfCycles / f32NumberOfLights * 2;
tFloat32 f32StartOfTurnrightlight = ui32NumberOfCycles / f32NumberOfLights * 3;
tFloat32 f32StartOfReverselight = ui32NumberOfCycles / f32NumberOfLights * 4;

#####
// send some samples
#####
// to give the motor controller time for initialisation send for a short time only
wd signal
for (tUInt32 nIdx = 0; nIdx < ui32Waittime; nIdx++)
{
    // Always send Watchdog Signal
    //Let the Watchdog send some Signals before starting the test
    cSystem::Sleep(1000);
}
#####
// define a Ramp from one border to init Value
if(chFrameId == ID_ARD_ACT_STEER_ANGLE || chFrameId == 0xff)
{
    f32LowerRange = 95.0f;
    f32UpperRange = 125.0f;

}else if (chFrameId == ID_ARD_ACT_ACCEL_SERVO){
    f32LowerRange = 90.0f;
    f32UpperRange = 180.0f;
}
#####
// send the samples according to the specified ramp
for (tUInt32 nIdx = 0; nIdx < ui32MaxLoopCount/4; nIdx++)
{

    // Acceleration test
    if(chFrameId==ID_ARD_ACT_ACCEL_SERVO || chFrameId == 0xff)
    {
        // set the data of the protocol
        // Calculate the value by dividing the range by the loop count times the
loop counter
        // through this every Value will be tested
        tFloat32 f32Value = (f32UpperRange - f32LowerRange) / (ui32MaxLoopCount/4)
* nIdx + f32LowerRange;
        // send the sample
        transmitSignalMediaSample(pAccSampleSource, f32Value);
        // wait to control the sendrate

```



```

        cSystem::Sleep(nSleep);
    }

    // Steer angle test
    if(chFrameId==ID_ARD_ACT_STEER_ANGLE || chFrameId == 0xff)
    {
        // set the data of the protocol
        // Calculate the value by dividing the range by the loop count times the
loop counter
        // through this every Value will be tested
        tFloat32 f32Value = (f32UpperRange - f32LowerRange) / (ui32MaxLoopCount/4)
* nIdx + f32LowerRange;
        // send the sample
        transmitSignalMediaSample(pSteerAngleSampleSource, f32Value);
        // wait to control the sendrate
        cSystem::Sleep(nSleep);
    }

    // Light Test
    if(chFrameId == ID_ARD_ACT_LIGHT_DATA || chFrameId == 0xff)
    {
        // set the value
        tBool bValue = tTrue;
        // send the sample
        // Divide the loop into equal sections for every Light
        if(nIdx > (f32StartOfHeadlight) && nIdx < (f32StartOfBrakelight) )
transmitBoolMediaSample(pHeadLightSampleSource,bValue);
        if(nIdx > (f32StartOfBrakelight) && nIdx < (f32StartOfTurnleftlight) )
transmitBoolMediaSample(pBrakeLightSampleSource,bValue);
        if(nIdx > (f32StartOfTurnleftlight) && nIdx < (f32StartOfTurnrightlight) )
transmitBoolMediaSample(pTurnLeftSampleSource,bValue);
        if(nIdx > (f32StartOfTurnrightlight) && nIdx < (f32StartOfReverselight) )
transmitBoolMediaSample(pTurnRightSampleSource,bValue);
        if(nIdx > (f32StartOfReverselight) && nIdx < ui32MaxLoopCount)
transmitBoolMediaSample(pReverseLightSampleSource,bValue);
        // wait to control the sendrate
        cSystem::Sleep(nSleep);
    }
}
//#####
// Go the same ramp backwards
//#####
// send the samples according to the specified ramp
for (tUInt32 nIdx = ui32MaxLoopCount/4; nIdx > 0 ; nIdx--)
{

    // Acceleration test
    if(chFrameId==ID_ARD_ACT_ACCEL_SERVO || chFrameId == 0xff)
    {
        // set the data of the protocol
        // Calculate the value by dividing the range by the loop count times the
loop counter
        // through this every Value will be tested
        tFloat32 f32Value = (f32UpperRange - f32LowerRange) / (ui32MaxLoopCount/4)
* nIdx + f32LowerRange;
        // send the sample
        transmitSignalMediaSample(pAccSampleSource, f32Value);
        // wait to control the sendrate
        cSystem::Sleep(nSleep);
    }

    // Steer angle test

```

```

    if(chFrameId==ID_ARD_ACT_STEER_ANGLE || chFrameId == 0xff)
    {
        // set the data of the protocol
        // Calculate the value by dividing the range by the loop count times the
loop counter
        // through this every Value will be tested
        tFloat32 f32Value = (f32UpperRange - f32LowerRange) / (ui32MaxLoopCount/4)
* nIdx + f32LowerRange;
        // send the sample
        transmitSignalMediaSample(pSteerAngleSampleSource, f32Value);
        // wait to control the sendrate
        cSystem::Sleep(nSleep);
    }

    // Light Test
    if(chFrameId == ID_ARD_ACT_LIGHT_DATA || chFrameId == 0xff)
    {
        // set the value
        tBool bValue = tFalse;
        // send the sample
        // Divide the loop into equal sections for every Light
        if(nIdx > (f32StartOfHeadlight) && nIdx < (f32StartOfBrakelight) )
transmitBoolMediaSample(pHeadLightSampleSource,bValue);
        if(nIdx > (f32StartOfBrakelight) && nIdx < (f32StartOfTurnleftlight) )
transmitBoolMediaSample(pBrakeLightSampleSource,bValue);
        if(nIdx > (f32StartOfTurnleftlight) && nIdx < (f32StartOfTurnrightlight) )
transmitBoolMediaSample(pTurnLeftSampleSource,bValue);
        if(nIdx > (f32StartOfTurnrightlight) && nIdx < (f32StartOfReverselight) )
transmitBoolMediaSample(pTurnRightSampleSource,bValue);
        if(nIdx > (f32StartOfReverselight) && nIdx < ui32MaxLoopCount)
transmitBoolMediaSample(pReverseLightSampleSource,bValue);
        // wait to control the sendrate
        cSystem::Sleep(nSleep);
    }
}
//#####
// define a Ramp from init Value to the other border
if(chFrameId == ID_ARD_ACT_STEER_ANGLE || chFrameId == 0xff)
{
    f32LowerRange = 65.0f;
    f32UpperRange = 95.0f;

}
else if (chFrameId == ID_ARD_ACT_ACCEL_SERVO){
    f32LowerRange = 0.0f;
    f32UpperRange = 90.0f;
}
//#####
// send the samples according to the specified ramp
for (tUInt32 nIdx = ui32MaxLoopCount/4; nIdx > 0 ; nIdx--)
{

    // Acceleration test
    if(chFrameId==ID_ARD_ACT_ACCEL_SERVO || chFrameId == 0xff)
    {
        // set the data of the protocol
        // Calculate the value by dividing the range by the loop count times the
loop counter
        // through this every Value will be tested
        tFloat32 f32Value = (f32UpperRange - f32LowerRange) / (ui32MaxLoopCount/4)
* nIdx + f32LowerRange;
        // send the sample
        transmitSignalMediaSample(pAccSampleSource, f32Value);
    }
}

```

```

        // wait to control the sendrate
        cSystem::Sleep(nSleep);
    }

    // Steer angle test
    if(chFrameId==ID_ARD_ACT_STEER_ANGLE || chFrameId == 0xff)
    {
        // set the data of the protocol
        // Calculate the value by dividing the range by the loop count times the
loop counter
        // through this every Value will be tested
        tFloat32 f32Value = (f32UpperRange - f32LowerRange) / (ui32MaxLoopCount/4)
* nIdx + f32LowerRange;
        // send the sample
        transmitSignalMediaSample(pSteerAngleSampleSource, f32Value);
        // wait to control the sendrate
        cSystem::Sleep(nSleep);
    }

    // Light Test
    if(chFrameId == ID_ARD_ACT_LIGHT_DATA || chFrameId == 0xff)
    {
        // set the value
        tBool bValue = tTrue;
        // send the sample
        // Divide the loop into equal sections for every Light
        if(nIdx > (f32StartOfHeadlight) && nIdx < (f32StartOfBrakelight) )
transmitBoolMediaSample(pHeadLightSampleSource,bValue);
        if(nIdx > (f32StartOfBrakelight) && nIdx < (f32StartOfTurnleftlight) )
transmitBoolMediaSample(pBrakeLightSampleSource,bValue);
        if(nIdx > (f32StartOfTurnleftlight) && nIdx < (f32StartOfTurnrightlight) )
transmitBoolMediaSample(pTurnLeftSampleSource,bValue);
        if(nIdx > (f32StartOfTurnrightlight) && nIdx < (f32StartOfReverselight) )
transmitBoolMediaSample(pTurnRightSampleSource,bValue);
        if(nIdx > (f32StartOfReverselight) && nIdx < ui32MaxLoopCount)
transmitBoolMediaSample(pReverseLightSampleSource,bValue);
        // wait to control the sendrate
        cSystem::Sleep(nSleep);
    }
}

//#####
// Go the same ramp backwards
//#####
// send the samples according to the specified ramp
for (tUInt32 nIdx = 0; nIdx < ui32MaxLoopCount/4; nIdx++)
{

    // Acceleration test
    if(chFrameId==ID_ARD_ACT_ACCEL_SERVO || chFrameId == 0xff)
    {
        // set the data of the protocol
        // Calculate the value by dividing the range by the loop count times the
loop counter
        // through this every Value will be tested
        tFloat32 f32Value = (f32UpperRange - f32LowerRange) / (ui32MaxLoopCount/4)
* nIdx + f32LowerRange;
        // send the sample
        transmitSignalMediaSample(pAccSampleSource, f32Value);
        // wait to control the sendrate
        cSystem::Sleep(nSleep);
    }
}

```

```

// Steer angle test
if(chFrameId==ID_ARD_ACT_STEER_ANGLE || chFrameId == 0xff)
{
    // set the data of the protocol
    // Calculate the value by dividing the range by the loop count times the
loop counter
    // through this every Value will be tested
    tFloat32 f32Value = (f32UpperRange - f32LowerRange) / (ui32MaxLoopCount/4)
* nIdx + f32LowerRange;
    // send the sample
    transmitSignalMediaSample(pSteerAngleSampleSource, f32Value);
    // wait to control the sendrate
    cSystem::Sleep(nSleep);
}

// Light Test
if(chFrameId == ID_ARD_ACT_LIGHT_DATA || chFrameId == 0xff)
{
    // set the value
    tBool bValue = tFalse;
    // send the sample
    // Divide the loop into equal sections for every Light
    if(nIdx > (f32StartOfHeadlight) && nIdx < (f32StartOfBrakelight) )
transmitBoolMediaSample(pHeadLightSampleSource,bValue);
    if(nIdx > (f32StartOfBrakelight) && nIdx < (f32StartOfTurnleftlight) )
transmitBoolMediaSample(pBrakeLightSampleSource,bValue);
    if(nIdx > (f32StartOfTurnleftlight) && nIdx < (f32StartOfTurnrightlight) )
transmitBoolMediaSample(pTurnLeftSampleSource,bValue);
    if(nIdx > (f32StartOfTurnrightlight) && nIdx < (f32StartOfReverselight) )
transmitBoolMediaSample(pTurnRightSampleSource,bValue);
    if(nIdx > (f32StartOfReverselight) && nIdx < ui32MaxLoopCount)
transmitBoolMediaSample(pReverseLightSampleSource,bValue);
    // wait to control the sendrate
    cSystem::Sleep(nSleep);
}
}

//#####

//#####
#####

// clean up

// disconnect the input pins of the aktors filter with the sample sources
__adtf_test_result(pAccInPin->Disconnect(pAccSampleSource));
__adtf_test_result(pSteerAngleInPin->Disconnect(pSteerAngleSampleSource));
__adtf_test_result(pHeadLightInPin->Disconnect(pHeadLightSampleSource));
__adtf_test_result(pTurnLeftInPin->Disconnect(pTurnLeftSampleSource));
__adtf_test_result(pTurnRightInPin->Disconnect(pTurnRightSampleSource));
__adtf_test_result(pBrakeLightInPin->Disconnect(pBrakeLightSampleSource));
__adtf_test_result(pReverseLightInPin->Disconnect(pReverseLightSampleSource));
__adtf_test_result(pEmergencyInPin->Disconnect(pEmergencySampleSource));

__adtf_test_result(pArdCommInPin->Disconnect(pArdAktOutPin));
__adtf_test_result(pArdSensInPin->Disconnect(pArdCommOutPin));

// Disconnect the output pins of the sensors filter with the sample sinks
__adtf_test_result(pArdSensUSOutPin->UnregisterEventSink(pUSSampleSink));
__adtf_test_result(pArdSensIROutPin->UnregisterEventSink(pIRSampleSink));
__adtf_test_result(pArdSensAccOutPin->UnregisterEventSink(pSteerAngleSampleSink));

```

```

    __adtf_test_result(pArdSensUSOutPin->UnregisterEventSink(pAccSampleSink));
    __adtf_test_result(pArdSensGyroOutPin->UnregisterEventSink(pGyroSampleSink));
    __adtf_test_result(pArdSensRPMOutPin->UnregisterEventSink(pRPMSampleSink));
    __adtf_test_result(pArdSensVoltageOutPin->UnregisterEventSink(pVoltageSampleSink));

    // shutdown the filter
    SET_STATE_SHUTDOWN(pAktFilter);
    SET_STATE_SHUTDOWN(pCommFilter);
    SET_STATE_SHUTDOWN(pSensFilter);
    SET_STATE_SHUTDOWN(pWDFilter);

    //#####
    // Start of evaluation

    //#####

    // now check the received data
    // Check for 80 samples, that should be received following the send rate of the
    // arduino. Different sample count for US 50 and Voltage 10 due to the lower send rate
    __adtf_test_ext(pUSSampleSink->m_ui32SampleCount >= 50, cString::Format("Sample
count (%d) does not match (%d)", pUSSampleSink->m_ui32SampleCount, 50));
    __adtf_test_ext(pIRSampleSink->m_ui32SampleCount >= 80, cString::Format("Sample
count (%d) does not match (%d)", pIRSampleSink->m_ui32SampleCount, 80));
    __adtf_test_ext(pSteerAngleSampleSink->m_ui32SampleCount >= 80,
cString::Format("Sample count (%d) does not match (%d)", pSteerAngleSampleSink-
>m_ui32SampleCount, 80));
    __adtf_test_ext(pAccSampleSink->m_ui32SampleCount >= 80, cString::Format("Sample
count (%d) does not match (%d)", pAccSampleSink->m_ui32SampleCount, 80));
    __adtf_test_ext(pGyroSampleSink->m_ui32SampleCount >= 80, cString::Format("Sample
count (%d) does not match (%d)", pGyroSampleSink->m_ui32SampleCount, 80));
    __adtf_test_ext(pRPMSampleSink->m_ui32SampleCount >= 80, cString::Format("Sample
count (%d) does not match (%d)", pRPMSampleSink->m_ui32SampleCount, 80));
    __adtf_test_ext(pVoltageSampleSink->m_ui32SampleCount >= 10,
cString::Format("Sample count (%d) does not match (%d)", pVoltageSampleSink-
>m_ui32SampleCount, 10));

    // If we are in the acceleration or the complete test check for the ramp
    if(chFrameId==ID_ARD_ACT_ACCEL_SERVO || chFrameId ==
0xff)evaluateRPM(pRPMSampleSink, chFrameId);
    // If we are in the steerangle or the complete test check for the ramp
    if(chFrameId==ID_ARD_ACT_STEER_ANGLE || chFrameId == 0xff)
evaluateSteerAngle(pSteerAngleSampleSink);

    //#####
    // End of evaluation

    //#####

}

// Definition of the Test cases

DEFINE_TEST(cTesterAADCAktorsCommSensors,
    TestAcceleration,

```

```
        "4.1",
        "TestAcceleration",
        "This test makes sure, that the 'accelerate' pin is present. After that
some acceleration data"\
        "will be send to the filter and the filter output will be compared to some
reference data.",
        "Pin can be found and the output data are as expected.",
        "See above",
        "none",
        "The filter must be able to receive acceleration data and convert them
into arduino data.",
        "automatic")
{
    // accelerate pin with expected frameID
    return DoActuatorCommSensorTest(ID_ARD_ACT_ACCEL_SERVO, 40000);
}
```

```
DEFINE_TEST(cTesterAADCAktorsCommSensors,
            TestSteeringAngle,
            "4.2",
            "TestSteeringAngle",
            "This test makes sure, that the 'steerAngle' pin is present. After that
some steering data"\
            "will be send to the filter and the filter output will be compared to some
reference data.",
            "Pin can be found and the output data are as expected.",
            "See above",
            "none",
            "The filter must be able to receive steering data and convert them into
arduino data.",
            "automatic")
{
    // steering angle pin with expected frameID
    return DoActuatorCommSensorTest(ID_ARD_ACT_STEER_ANGLE, 50000);
}
```

```
DEFINE_TEST(cTesterAADCAktorsCommSensors,
            TestLights,
            "4.3",
            "TestLights",
            "This test makes sure, that the 'LightPins' pins are present. After that
some Lightsignals data"\
            "will be send to the filter and the filter output will be compared to some
reference data.",
            "Pin can be found and the output data are as expected.",
            "See above",
            "none",
            "The filter must be able to receive Light data and convert them into
arduino data.",
            "automatic")
{
    // Specific light test so no parameters needed
    return DoActuatorCommSensorTest(ID_ARD_ACT_LIGHT_DATA, 40000);
}
```

```
DEFINE_TEST(cTesterAADCAktorsCommSensors,
            TestAllActors,
            "4.4",
            "TestAllActors",
```

```
        "..."\n        "is set to different steering angle values",\n        "The communication filter sends allways data",\n        "See above",\n        "none",\n        "The filter must be able to receive steering data while delivering sensor\n        data.",\n        "automatic")\n{\n    // steering angle pin with expected frameID\n    return DoActuatorCommSensorTest(0xff, 40000);\n}\n\nDEFINE_TEST_INACTIVE(cTesterAADCActorsCommSensors,\n    TestEmergencyStop,\n    "1.5",\n    "TestEmergencyStop",\n    "This test makes sure, that the 'Emergency' pin is present. After that\n    some Emergency data"\\\n    "will be send to the filter and the filter output will be compared to some\n    reference data.",\n    "Pin can be found and the output data are as expected.",\n    "See above",\n    "none",\n    "The filter must be able to receive Emergency data and convert them into\n    arduino data.",\n    "automatic")\n{\n    // Do the BoolTest with the Watchdog pin and test the received samples\n    return DoActuatorCommSensorTest(0xfe, 20000);\n}
```