Teknisk Dokumentation

Redaktör: Anders Toverland

Version 0.13

Status

Granskad		
Godkänd		

PROJEKTIDENTITET

Grupp 1, 2005/VT, NightRider
Linköpings Tekniska Högskola, ISY

Gruppdeltagare

Namn	**Ansvar**	**Telefon**	**E-post**
Anders Wikström	Kvalitetsansvarig (KVA)	073-815 58 17	andvi526@student.liu.se
Lina Blom	Projektledare (PL)	070-200 93 36	linbl444@student.liu.se
Anders Bergmark	Testansvarig (TST)	070-555 07 62	andbe552@student.liu.se
Anders Toverland	Dokumentansvarig (DOK)	013-14 59 27 070-954 45 27	andto280@student.liu.se
Rikard Borst	Designansvarig fordon (DFA)	070-636 02 47	rikbo236@student.liu.se
Johan Tunkrans	Designansvarig visualisering (DVA)	070-575 07 68	johtu203@student.liu.se
Stefan Wedell	Kundansvarig (KA)	070-891 12 97	stewe678@student.liu.se

E-postlista för hela gruppen: fordonssimulator@yahoogroups.com
Hemsida:
http://www.vehicular.isy.liu.se/Edu/Courses/TSFS07/Proposals/fordonssimulator.html

Kund: Fordonssystem, ISY, 581 83 Linköping,
Kundtel: 013-28 10 00, Fax: 013-13 92 82, da@isy.liu.se
Kontaktperson hos kund: Lars Eriksson, 013-28 44 09, larer@isy.liu.se
Kursansvarig: Anders Hansson, 013-28 16 81, hansson@isy.liu.se
Handledare: Anders Fröberg, 013-28 40 66, froberg@isy.liu.se
Per Öberg, 013-28 23 69, oberg@isy.liu.se

Innehåll

Dokumenthistorik

Version	**Datum**	**Utförda förändringar**	**Utförda av**	**Granskad**
0.1	2005–05–16	Första utkast.	Alla	
0.11	2005–05–16	Kommunikationsmodulen tilllagd.	JT	
0.12	2005–05–18	Ändrat efter handledares kommentarer (IM, LM, KM).	Alla	
0.13	2005–05–19	Ändrat efter handledares kommentarer (övriga moduler).	Alla	

1 Inledning

Första delen av tekniska dokumentationen är tänkt att ge en snabb översikt av systemet varefter vi fördjupar oss i detaljerna. I detta dokument förklaras delar av den kod som finns men för full förståelse av vår lösning kommer givetvis källkoden behöva studeras. Källkod finns ej inte i detta dokument utan vi hänvisar till kodbilagan. För varje modul som är kodad kommer dess uppgift och vilka meddelanden den skickar och tar emot att finnas beskrivna, för simulinkmodellerna kommer in och utgångar att vara definerade. Detta gör det enkelt att byta ut en godtycklig del av projektet.

1.1 Sökvägar

NRRoot är den katalog där NightRider är inlagd.

1.1.1 IM

Visualstudio projekt: `NRRoot\IM\Gui\Gui.sln`

1.1.2 FM

Simulinkmodell: `NRRott\FM\fordonsmodell.mdl`

1.1.3 KM - Server

Visualstudio projekt: `NRRoot\KM\Project\Server\Server.sln`

1.1.4 KM - Klient

Headerfil för klienten: `NRRoot\KM\Client\Client.h`

1.1.5 LM

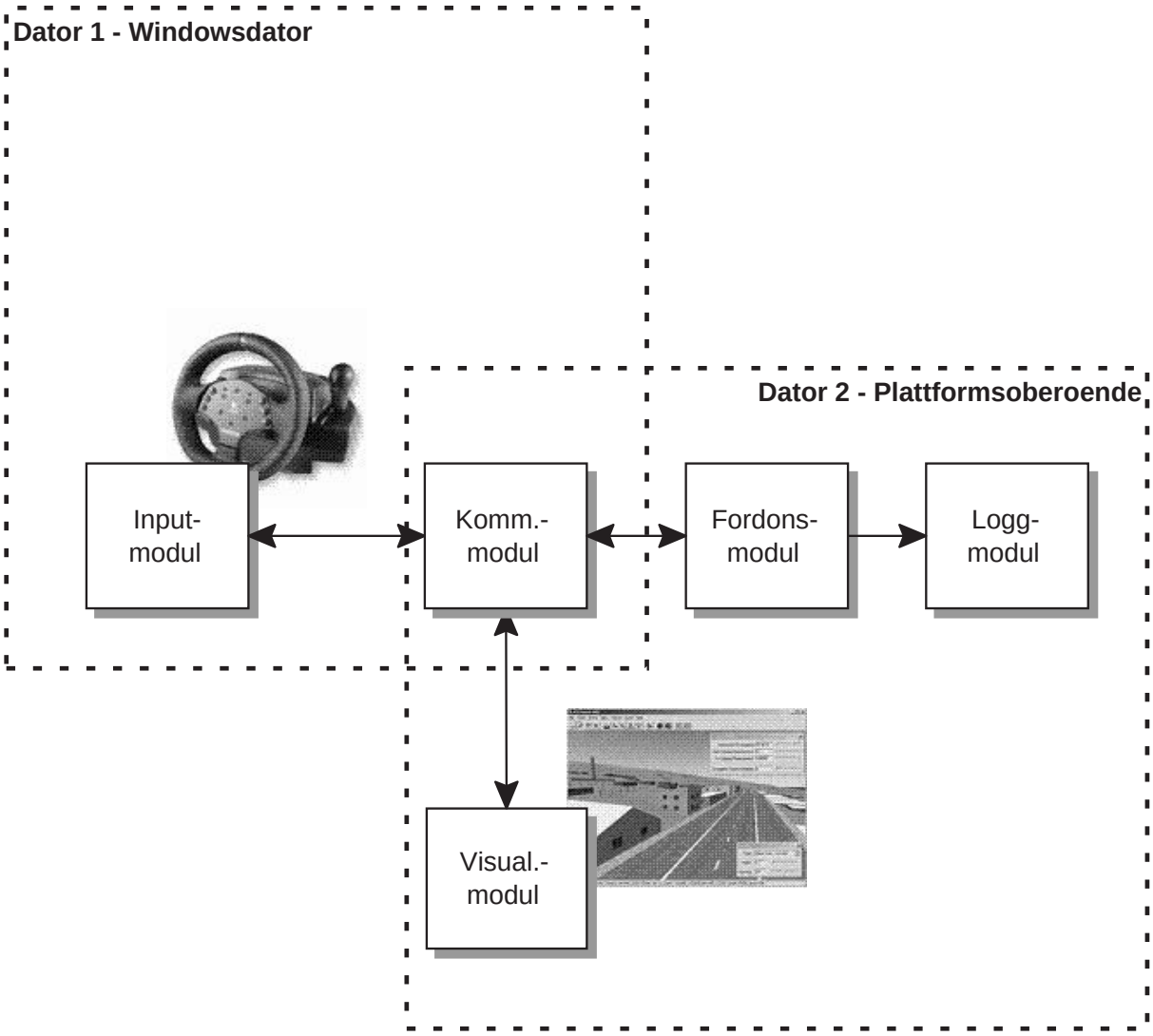
Visualstudio projekt: `NRRoot\LM\Ljudmodul main\Ljudmodul main.sln`

1.1.6 VM

Visualstudio projekt: `NRRoot\VM\Visualisering.sln`

2 Översikt av systemet

Systemet kan delas upp i fem moduler: Inputmodul, fordonsmodul, loggmodul, kommunikationsmodul och visualiseringsmodul. Det ska vara möjligt att dela upp modulerna på en eller flera datorer. Detta gör det dels möjligt för oss att göra visualiseringsmodulen plattformsoberoende och dessutom kan vi erhålla en bättre prestanda genom att köra simuleringen på en dator och visualiseringen på en annan.



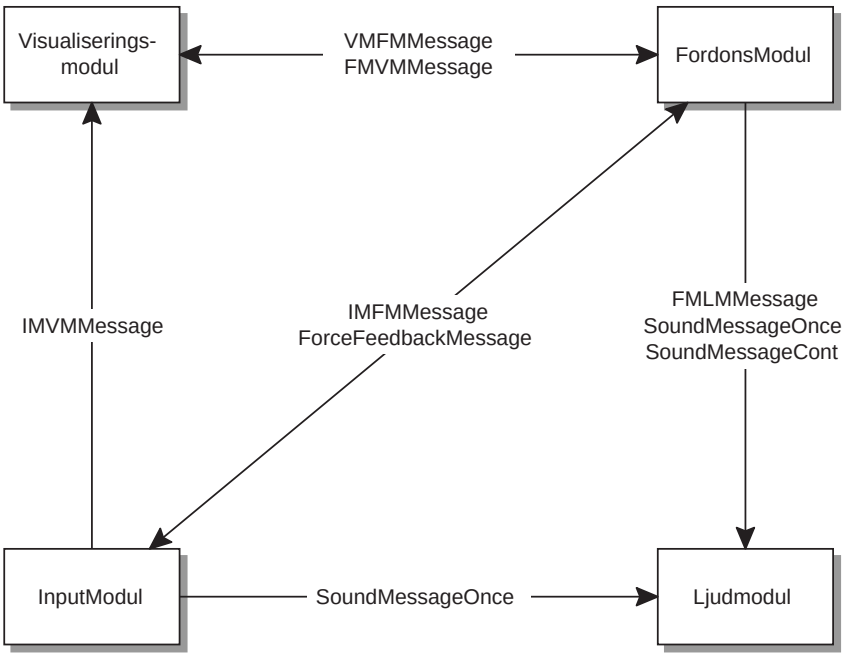
Figur 1: Exempel på hur systemet kan se ut när det körs på 2 datorer

2.1 Ingående delsystem

- Inputmodul (IM). Denna modul har hand om den ratt med tillhörande pedaler som är kopplad till den ena datorn.
- Fordonsmodul (FM). Denna modul simulerar ett riktigt fordon. Består av en simulinkmodell med tillhörande s-funktion som sköter nätverksanvändning. Kompileras med Realtime Workshop.
- Ljudmodul (LM). Denna modul har hand om uppspelning av ljud.
- Kommunikationsmodul (KM). Denna modul har hand om kommunikationen mellan de olika datorerna.
- Visualiseringsmodul (VM). Denna modul har hand om att visualisera fordonet i dess omkringliggande miljö.

2.2 Kommunikation

Modulerna kommunicerar över udp/ip via en server. Varje modul skickar meddelanden till andra moduler men behöver inte veta ifall dessa kommer fram eller ej, du kan alltså skicka meddelanden till ljudmodulen om att den ska spela upp ljud utan att först behöva kolla ifall denna är igång. En snabb översikt över vilka meddelanden som skickas mellan de olika modulerna vilken återfinns i figur 2.



Figur 2: Översikt över vilka meddelande som skickas

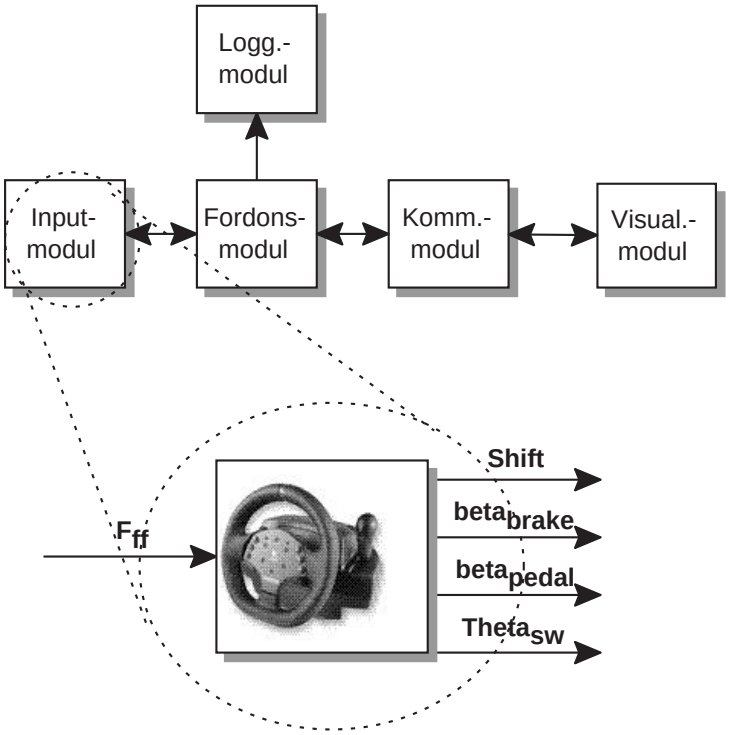
2.3 Realtid

Vår simulinkmodell jobbar med ett tidsteg på tio millisekunder. Den s-funktion som har hand om kommunikationen mellan simulinkmodellen och övriga moduler ser till att man sover den tid som inte gått åt till simulering. Tar exempelvis simuleringen tre millisekunder, sover s-funktionen sju millisekunder. Däremot har vi användt oss utav Windows XP vilket contextswitchar var tionde millisekund, så vi kan med andra ord inte garantera svarstider. Det bästa vore givetvis att kompilera simulinkmodellen med Realtime Workshop och sedan köra denna på en dator med realtidslinux.

3 Inputmodul

3.1 Övergripande modulbeskrivning

Inputmodulen sköter kommmunikationen med den ratt med tillhörande pedaler som kommer användas. DirectInput kommer att användas för att kommunicera med ratten. Detta



Figur 3: Blockschema över Inputmodulen

innebär att den dator som inputmodulen återfinns på måste vara windowsbaserad. Inputmodulen kan även lägga en kraft på ratten. DirectX är gratis att ladda hem ifrån Microsofts hemsida.

3.2 Design

Två klasser används, den ena heter `Form1` och ärver ifrån `System::Windows::Forms::Form` den andra heter `MvisDirectInput` och är en vidareutveckling utav Karls Mårtenssons klass med samma namn.

3.2.1 Timer: tmrUpdateJoy

Timern `tmrUpdateJoy` anropar funktionen `update()` var 10 ms vilken i sin tur anropar `UpdateInputState()` vilken kontrollerar joystickens knappar samt axellägen. Denna funktion har även hand om att skicka meddelanden till ljudmodulen om tutan har blivit nertryckt. Den skickar även ut `ShutDownMessage` till samtliga moduler ifall escapeknappen har blivit nertryckt. Den viktigaste uppgiften är däremot att uppdatera de interna variablerna: `BetaSW, BetaAccBrake, BetaBrake, gear` vilka sedan kommer att skickas vidare till fordonsmodulen. Ifall man vill ändra prioritet mellan tangentbord och ratt är det här man gör det.

3.2.2 Timer: tmrNetwork

Denna timer anropar funktionerna `NetworkReceive()` samt `NetworkSend()` var 40 ms. Dessa funktioner tar emot och skickar information till fordonsmodulen, vi har alltså 25 uppdateringar per sekund till detta vilket har visat sig vara tillräckligt.

3.2.3 Funktion: NetworkReceive()

Header	`void NetworkReceive()`
Argument	-
Returnerar	-

Tar emot `ForceFeedbackMessage` och lägger ut kraften den får ifrån detta meddelande på ratten, tar även emot `ShutdownMessage` och ifall den fått ett sådant kopplar den ifrån sig ifrån servern.

3.2.4 Funktion: NetworkSend()

Header	`void NetworkSend()`
Argument	-
Returnerar	-

Skickar två meddelanden. Ett `IMFMMessage` som innehåller den information fordonsmodulen behöver ifrån inputmodulen alltså rattvinkel, pedallägen samt växlingssignal. Den skickar även ett `IMVMMessage` till visualiseringsmodulen tre värden: `x, y, angle`. I dagsläget används enbart `angle` vilket talar om för visualiseringsmodulen hur mycket den ska vrida ratten.

3.2.5 Klass: MvisDirectInput

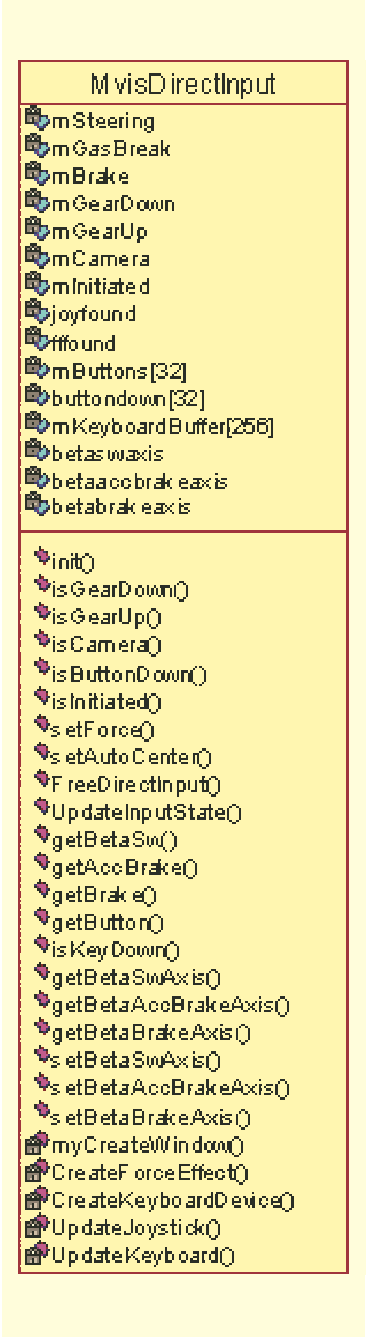
Den viktigaste funktionen är `UpdateInputState()` som uppdaterar interna variabler genom att kontrollera axlar och knapplägen för joystick samt tangentbord. Om man kör den tillräckligt ofta (ungefär 100 gånger per sekund) så missar man inget och kan sen läsa av dessa lägen med hjälp av de `get` funktioner som finns. För närmare dokumentation av denna klass hänvisar vi till Karl Mårtenssons examensarbete och även Microsoft DirectX SDK Documentation.

3.2.6 Klassdiagram

MvisDirectInput är IMs enda klass, ett diagram över denna återfinns i figur 4.

3.2.7 GUI

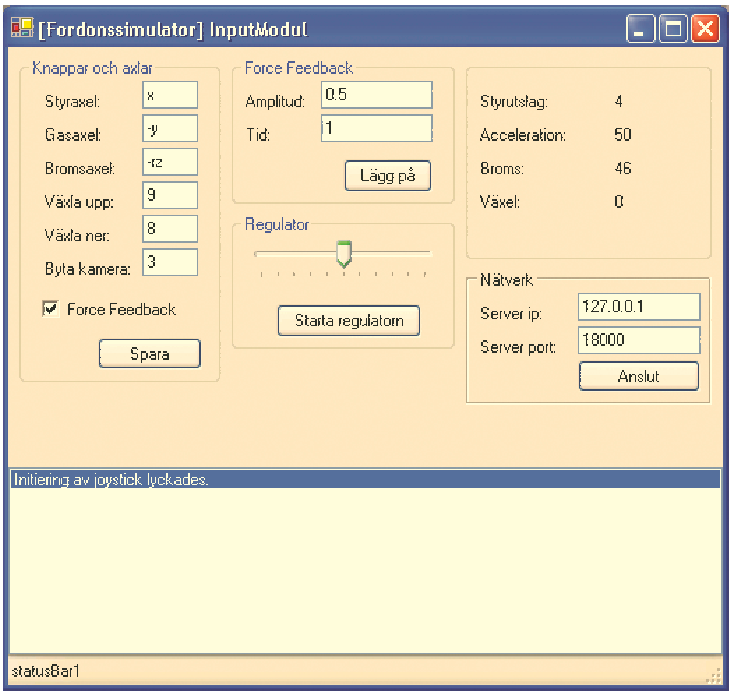
En skärmdump återfinns i figur 5.



Figur 4: Klassdiagram över MvisDirectInput

3.2.8 Kompabilitet

`FsimDirectInput();` har stöd för alla styrdon som har stöd för DirectInput, alltså i princip alla rattar och joysticks på marknaden. För att sedan få det användbart i vår simulator får man ändra mappningen i inputmodulens gui.



Figur 5: Screenshot över Inputmodulens GUI

3.3 Nätverk

Ifall denna modul ska bytas ut ska följande meddelanden behandlas.

3.3.1 Ta emot

- **ForceFeedbackMessage** - Lägg på kraft på ratten.
- **ShutdownMessage** - Koppla ifrån.

3.3.2 Skicka

- **IMVMMessage** - Rattposition.
- **IMFMMessage** - Rattpostion, gas, broms och växelsignal.
- **ShutdownMessage** - Sänd meddelanden om avslut vid escape tryckning.
- **SoundMessageOnce** - Spela upp tuta.

4 Loggmodul

Realiseras genom att dra ut To Fileblock i Simulinkmodellen och koppla dessa till de signaler man vill logga.

5 Ljudmodul

5.1 Modellbeskrivning

Ljudmodulen är uppdelad i två delar. Huvuddelen spelar upp kontinuerliga ljud som kan bero på vissa parametrar i modellen. Ett mycket bra exempel på ett sådant ljud är motorljudet som beror på varvtalet. Den andra delen spelar upp ljud då speciella händelser inträffar. Dessa händelser kan initieras från valfri modul och skickas då över nätverket, argumentet på dessa paket består av filnamnet på den ljudfil som ska spelas upp. Exempel på detta kan vara ett ljud som spelas upp varje gång man växlar.

5.2 Design

Vi har här valt att använda `Fmod` vilken är en samling klasser som används för att spela upp ljud. Ifall den körs på windowssystem kommer DirectSound att användas, ifall man väljer att kompilera för Linux använder den OSS eller ESD. Det viktiga är att `Fmod` är fritt att använda så länge man inte tänkt tjäna pengar på den produkt man utvecklar. En annan viktigt del av valet av att använda den api var att den hade ett mycket bra hjälpsystem, så det bör inte vara några problem att snabbt sätta sig in i och ändra i koden för denna modul.

5.2.1 Huvudloop

Huvudloopen körs tills man har tagit emot ett `ShutdownMessage`. Den uppdaterar klienten, tar emot ett meddelande och utför rätt aktion beroende på vilken meddelandetyp som är mottagen.

5.2.2 Klass: playing

De interna listorna håller koll på vilken fil som spelas på vilken kanal på ljudkortet. Man kan maximalt spela upp 12 stycken kontinuerliga ljud, om inte detta räcker får man helt enkelt gå in och öka storlekarna på variablerna. De viktigaste funktionerna följer här:

5.2.3 Funktion: playing::change

Header	`int change(const char* filename, int freq, int volume)`
Argument	*filename* är den spelande filens namn *freq* är frekvensen och ska ligga mellan 100 och 705600 *volume* är volym och ska ligga mellan 0 och 255
Returnerar	0 ifall lyckat, 1 ifall misslyckat

Tar emot ett filnamn, en frekvens samt en volym. Den letar upp vilken kanal filen spelas i och ändrar sedan denna kanals frekvens samt volym. Ifall filen inte spelas startar man uppspelning av den genom att anropa funktionen `start`.

5.2.4 Funktion: playing::stop

Header	`void stop(const char* filename)`
Argument	*filename* är filnamnet på den fil som ska slutas spela upp.
Returnerar	-

Tar ett filnamn som argument och stoppar då uppspelningen av detta ljud.

5.2.5 Funktion: playing::playonce

Header	`int playonce(const char* filename, int volume)`
Argument	*filename* är namnet på filen *volume* är den volym den spelas upp med, ska ligga mellan 0 och 255
Returnerar	-

Tar ett filnamn och en volym som argument och spelar sedan upp detta ljud med den valda volymen.

5.3 Nätverk

Ifall denna modul ska bytas ut ska följande meddelanden behandlas.

5.3.1 Ta emot

- **FMLMMessage** - Spela upp motorljud samt slirljud med de volymer och frekvenser som specificeras.
- **SoundMessageOnce** - Spela upp det ljud som specificeras i meddelandet en gång.
- **SoundMessageCont** - Ifall ljudet spelas ska frekvens och volym ändras. Ifall det inte spelas ska det börjas spela.

- **SoundMessageStop** - Stäng av ett ljud som spelas kontinuerligt.
- **ShutdownMessage** - Avsluta.

5.4 Klassdiagram

Det finns enbart en klass i LM, ett klassdiagram finns i figur 6.

Figur 6: Klassen playing

6 Visualiseringsmodul

6.1 Övergripande modulbeskrivning

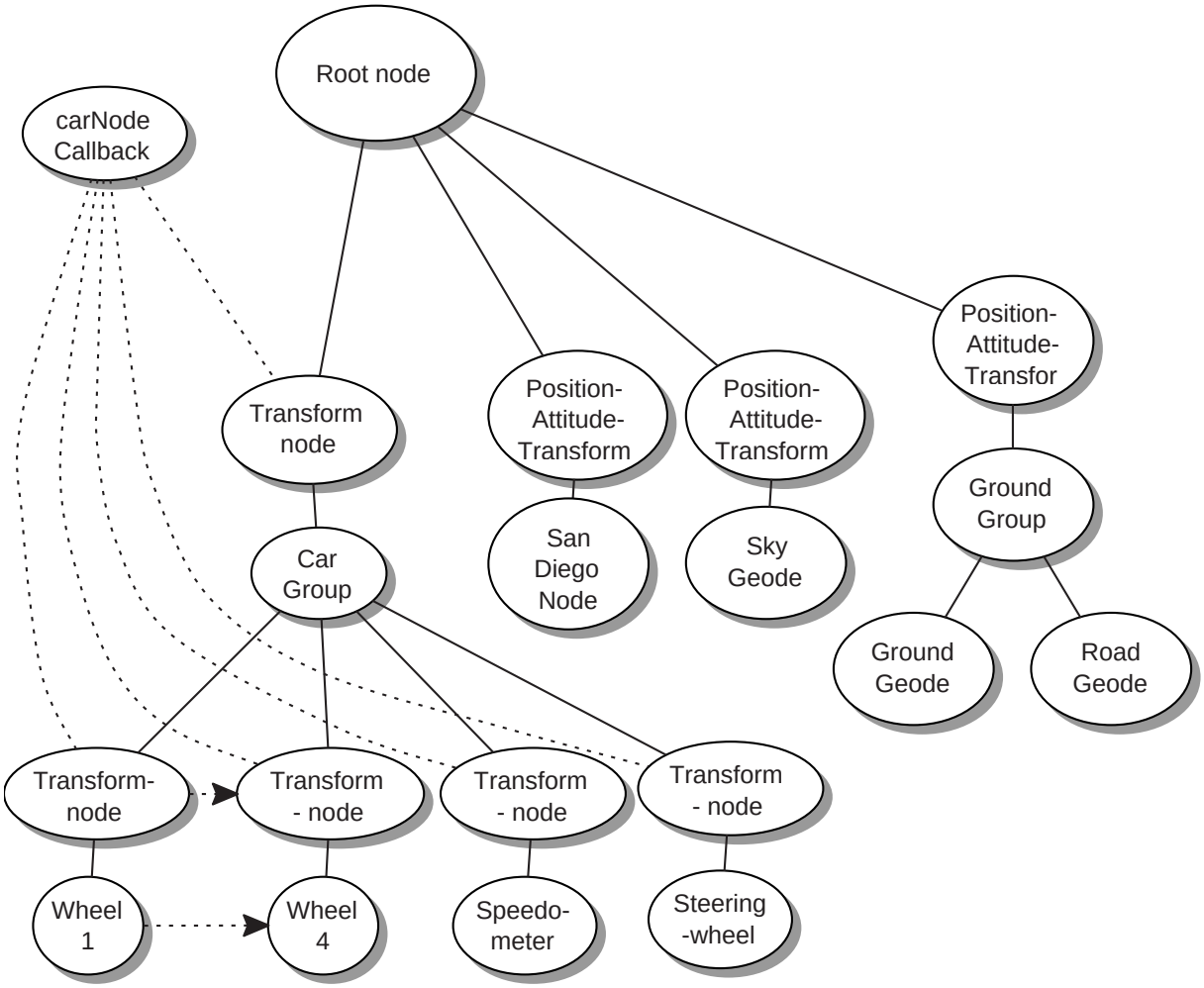
Visualiseringsmodulen får information om fordonets position samt rotation från fordonsmodulen och beräknar utifrån dessa värden den globala höjden hos respektive däck. Resultatet skickas sedan tillbaka till fordonsmodulen. Visualiseringsmodulen tar även emot meddelanden från inputmodulen, detta för att kunna uppdatera rattens samt framdäckens vridning. Dessutom är det givetvis i visualiseringsmodulen som all grafik byggs upp via Open Scene Graph.

6.2 Design

Eftersom visualiseringsmodulen är gjord i Open Scene Graph så kommer referenser till dess inbyggda klasser och funktioner att förekomma i dokumentationen nedan. Vi kommer bara ytligt beröra vad dessa innebär och för en mer ingående beskrivning av Open Scene Graphs klasser, funktioner och finesser så hänvisar vi till deras officiella hemsida, *www.openscenegraph.org*.

6.2.1 Klass: carNodeCallback()

Detta är en speciell klass som ärver från `osg::NodeCallback`. I huvudsimuleringsloopen som körs i main så körs bl.a. `viewer.update()`, denna uppdaterar samtliga noder som har en NodeCallback-funktion associerad till sig. För en mer utförlig beskrivning av hur NodeCallback fungerar så hänvisar vi till *www.openscenegraph.org*. För att applicera vår NodeCallback på en viss nod så anropar vi `setUpdateCallback(new carNodeCallback())`.



Figur 7: Blockschema över visualiseringen

6.2.2 Klass: DataType()

Från vår NodeCallback anropar vi klassen DataType och beroende på vilken nod vi befinner oss i så uppdaterar denna dess position. Detta görs med funktionen `updatePosition()` som utför rotations/positions-uppdatering.

6.2.3 Funktion: updatePosition()

Header	`Producer::CameraConfig* updatePosition()`
Argument	osg::MatrixTransform*, är den aktuella nodens transformnod. Det är denna som ändras vid positions/rotations-uppdateringar. Messages*, skapar ett FMVMMessage samt ett IMVMMessage. Client*, en klient för att ta emot meddelanden från FM och IM Ground*, automatgenererade terräng som bidrar med fordonets globala höjd. int, bestämmer vilken del av bilen vi vill uppdatera, (finns 7 alternativ).
Returnerar	Returnerar inget men i funktionen så skapas ett VMFMMessage och höjden hos de 4 däcken skickas till FM

6.2.4 Klass: Messages()

Består av en konstruktor som deklarerar ett FMVMMessage samt ett IMVMMessage.

6.2.5 Funktion: setupCameras()

Header	`Producer::CameraConfig* setupCameras()`
Argument	-
Returnerar	Producer::CameraConfig *cfg

`setupCameras()` är en funktion som konfigurerar kamerorna med lämpliga parametrar. Den aktuella vyn består för närvarande utav två individuella kameror. Den ena är huvudkameran, `carFollowerCamera()` som är placerad på förarplats för en bra vy dels över fordonets interiör och dels över omgivningen. Den andra kameran, `rearViewCamera()` ska representera backspegeln och konfigureras så att den tar upp en mindre del av skärmen och visar en vy bakåt i förhållande till förarens placering.

6.2.6 Funktion: BuildVehicleTree()

Header	`Producer::CameraConfig* BuildVehicleTree()`
Argument	Messages* Ground*
Returnerar	osg::Group*, en Group-nod som representerar hela fordonets subträd. (Se figur **??**)

Denna funktion bygger upp det subträd som motsvarar fordonet. Trädet består av 14 noder, 7 stycken för att representera fordonets individuellt rörliga delar, dvs fordonskropp,

4 däck, ratt samt nålen till hastigtetsmätaren. Varje nod har dessutom en tillhörande transformnod för att kunna uppdatera dess position/rotation. Samtliga löv i trädet är instanser av `osg::Node` medan fordonskroppen är en instans av `osg::Group`. `BuildVehicleTree()` returnerar den översta noden i det skapade subträdet. Hela fordonet är modellerat i 3D Studio Max av Karl Mårtensson som en del av hans examensarbete.

6.2.7 Klass: TransformAccumulator()

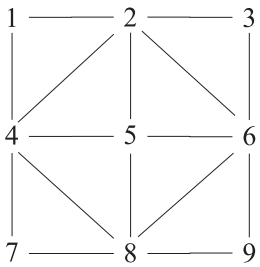
`TransformAccumulator()` har som främsta syfte att beräkna en nods globala koordinater. Klassens funktioner är intuitiva och vi kommer inte gå in närmre på dessa här.

6.2.8 Klass: followNodeMatrixManipulator()

Denna klass använder sig av resultatet av `TranformAccumulator` för att kunna "hänga på"en kamera på en nod. För djupare förståelse för dessa klasser och de ingående funktionerna så hänvisar vi till *www.openscenegraph.org*. Att använda dessa klasser för att positionera sin kamera är tämligen lätt och lämnar stor valfrihet till programmeraren.

6.2.9 Klass: Ground

Denna klass har för uppgift att skapa samt hålla reda på hur marken ser ut. Den bygger upp marken från en bmp fil, där värdet i den blå kanalen blir z-värdet för denna koordinat. Marken består av ett antal texturerade trianglar som är byggda från de punkter som lästes in ifrån .bmp-filen. Dessa är sammanbundna enligt figur 8 för att textureringen skall bli korrekt.



Figur 8: Figur över hur trianglarna är uppbyggda.

Header	`Ground(char* groundBitmap, char* roadBitmap, char* frictionMap, char* groundTexture, char* roadTexture, osg::Group* root, double aHillHeight, int aSize, int aN)`
Argument	*groundBitmap* är den .bmp som bestämmer höjdkurvan. Det är endast den blå kanalen som används och svart är lägst medan helt blått är högst. *roadBitmap* är den .bmp som bestämmer hur vägen ser ut. Återigen är det endast blå kanalen som har betydelse, och här betyder ett värde av 255 väg och resten inte väg. *frictionMap* är den .bmp som bestämmer friktionen i världen. Den blå kanalen används, och ett värde av 255 betyder något mindre än 1 i friktion, medan ett värde av 0 betyder 0 i friktion. *groundTexture* är den textur som skall användas för marken. *roadTexture* är den textur som skall användas till vägen. *root* är den nod i openscenegraph-värden som skall hålla marken. *aHillHeight* är en skalning av hur mycket kullar och dalar man vill ha. *aSize* är en skalning av arean. *aN* säger hur i hur många olika delar världen skall skapas. Många sådana gör att openscenegraph kan culla bort mycket, men innebär samtidigt en högre datamängd.
Returnerar	-

Header	`osg::Vec3f getHeight(double x, double y)`
Argument	*x och y* är koordinaterna för vilka man vill ha ut höjden på marken
Returnerar	Returnerar en vector till den höjdpunkt som efterfrågades. Observera att x, och y är med i denna, oförändrade.

Header	`double GetFriction(double x, double y)`
Argument	*x och y* är koordinaterna för vilka man vill ha ut friktionen på marken
Returnerar	returnerar ett värde mellan 0 och 1

Header	`private int createPartOfGround(int k, int l)`
Argument	*k, l* koordinater för var delen av marken som skapas skall placeras.
Returnerar	Returns success

Header	`osg::StateSet* getaState(char* texture)`
Argument	*texture* är namnet på textur-filen som man vill läsa in
Returnerar	Returnerar det *osg::StateSet** som openscenegraph kan använda sig av för att applicera en textur på ett objekt

6.2.10 PixelInfo

Denna klass läser in .bmp filer. Dessa används sedan för att bestämma hur marken i världen ser ut, samt för att bestämma friktion under hjulen på bilen.

Header	`PixelInfo(char* infile)`
Argument	*infile* är den .bmp fil som man vill läsa in
Returnerar	-

Header	`unsigned char Match(double a, double b)`
Argument	*a* är den x-koordinat som man vill hämta ett färgvärde för *b* är den y-koordinat som man vill hämta ett färgvärde för.
Returnerar	Returnerar färgvärdet i den specificerade punkten.

6.2.11 RGBQuad, BitmapFileHeader och BitmapInfoHeader

Dessa klasser är en hjälpklasser till PixelInfo för att läsa och hålla information från en .bmp-fil.

6.3 Nätverk

Ifall denna modul ska bytas ut ska följande meddelanden behandlas.

6.3.1 Ta emot

- **FMVMMessage** - Position (x,y,z), Rotation(phi,theta,psi)
- **IMVMMessage** - Rattposition

6.3.2 Skicka

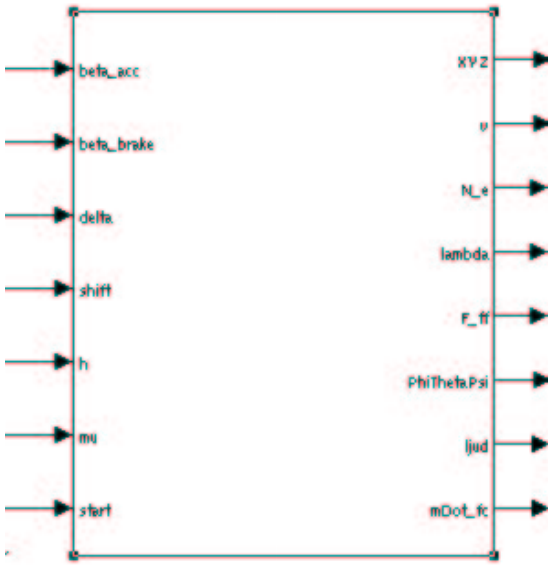
- **VMFMMessage** - Global höjd samt friktion för varje hjul ($h_1, h_2, h_3, h_4, \mu_1, \mu_2, \mu_3, \mu_4$)

7 Fordonsmodul

Fordonsmodulens syfte är att simulera ett fordon utifrån insignaler ifrån (IM) och därefter generera nödvändiga utsignaler till övriga moduler. Fordonsmodulen ska vara plattformsoberoende. Fordonsmodulen är uppbyggd av flera mindre delmoduler skapade i Matlab/

Simulink. Dessa moduler är helt diskretiserade för att kunna köras i realtid. Fordonet består av fem kroppar och kan röra sig i tre dimensioner.

Insignaler	β_{acc}, β_{brake}, δ, $shift$, h, μ, $start$
Utsignaler	x, y, z, v, N_e, λ, F_{ff}, Φ, Θ, Ψ, $ljud$, $\dot{m}_{fc}$



Figur 9: Blockschema över Fordonsmodul

7.1 Huvudprogram

Den diskretiserade modellen implementeras i simulink och kommunicerar med övriga moduler med hjälp av en s-funktion som kör c++ kod. Modellen kommer att köras med en samplingstid på 0.01 s.

7.2 Nätverk

Vi lyssnar här efter:

- **IMFMMessage** - Innehåller $\beta_{acc}, \beta_{brake}, \beta_{sw}$ samt $shift$.
- **VMFMMessage** - Höjd för varje hjul samt friktion på underlaget.
- **ShutdownMessage** - Stänger av simuleringen.

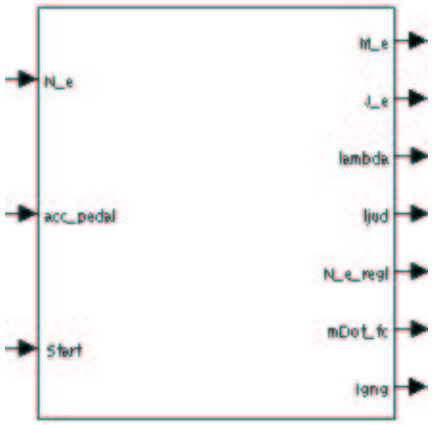
Och skickar sedan vidare:

- **FMVMMessage** - Skickar position samt rotation till vm.
- **SoundMessageOnce** - Ljudmeddelande för exempelvis motorstart.
- **FMLMMessage** - Skickar slip samt motorljud till ljudmodulen.
- **SoundMessageStop** - Stänger av dvs ljud vid simuleringsstop.

7.3 Motor

Motormodellen som använts är av medelvärdestyp påminner starkt om den som användes i lab1c i kursen fordonssystem-TFSF05. Hela motorn är diskretiserad och dessutom har en startmotor införts. I motormodellen finns ett antal block av ad hoc-karaktär för att detektera motorstart/stopp samt för att undvika division med 0 när motorn står still. Utifrån varvtal och gaspådrag bestäms ett moment ut från motorn. detta förs vidare till drivlinan. I kopplingsblocket beräknas motorns rotationshastighet.

Insignaler	N_e, acc_{pedal}, $start$
Utsignaler	M_e, J_e, λ, $ljud$, N_{eregl}, $\dot{m}_{fc}$, $igng$

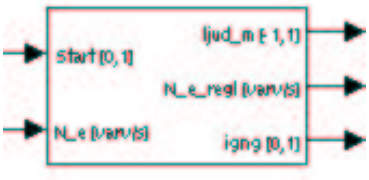


Figur 10: Blockschema över Motor

7.3.1 Startmotor

Signalen igng ändras då motorn startas och stoppas. Start ger igng=1 och stopp ger igng=0. Då varvtalet blir för lågt stoppas motorn och kan startas igen genom att sätta signalen start=1. En signal genereras även till ljudmodulen vid start och stopp.

Insignaler	$start$, N_e
Utsignaler	$ljud_m$, N_{eregl}, $igng$



Figur 11: Blockschema över Startmotor

7.3.2 Engine

Insignaler	acc_{pedal}, $igng$, N_e, t_{inj}
Utsignaler	M_e, J_e, λ, λ_{disc}, $\dot{m}_{at}$, p_{im}, $\dot{m}_{fc}$

Figur 12: Blockschema över Engine

Omvandling av pedalläget till luftmassflödets börvärde:

$$\dot{m}_{AirDemand} = \dot{m}_{atmin} + acc_{pedal} \cdot (\dot{m}_{atmax} - \dot{m}_{atmin}$$

Beräkning av luftmassflödet vid trotteln och i cylinder:

$$\frac{d}{dt}\dot{m}_{atSens} = \frac{1}{\tau_{th}}(\dot{m}_{AirDemand} - \dot{m}_{atSens})$$

$$\dot{m}_{ac} = \eta_{vol}\frac{n_{cyl}p_i NV_d}{n_r RT_i}$$

$$\dot{m}_{at} = \dot{m}_{AirDemand} - \dot{m}_{ac}$$

Beräkning av insugstrycket:

$$\frac{d}{dt}p_{imSens} = \frac{RT_i}{V_i}(\dot{m}_{AirDemand} - \dot{m}_{atSens})$$

Beräkning av fyllnadsgrad. Parametrarna c_0, c_1, c_2 skattades i fordonssystemskursen med minsta kvadratmetoden:

$$\eta_{vol} = c_0 + c_1\sqrt{p_i} + c_2\sqrt{N_e}$$

Beräkning av bränslemassflöde till insugningsröret:

$$\dot{m}_{fi} = \frac{n_{cyl}}{n_r}C_1 N_e(t_i nj - t_0)$$

Modellering av bränslepöl i insugningsröret:

$$\frac{d}{dt}m_{fp} = Xm_{fi} - \frac{1}{\tau_f p}m_{fp}$$

$$\dot{m}_{fc} = (1 - X)\dot{m}_f i + \frac{1}{\tau_{fp}} m_{fp}$$

Beräknar λ i cylindern och i avgasröret:

$$\lambda_{cyl} = \frac{\dot{m}_{ac}}{\dot{m}_{fc}(A/F)_s}$$

$$\frac{d}{dt}\lambda = \frac{1}{\tau_\lambda}(\lambda_{cyl}(t - \tau_d) - \lambda)$$

Beräkning av momentet ut från motorn. Friktionsparametrarna skattades i fordonssystemskursen med minsta kvadratmetoden:

$$M_e = \frac{W_{ig} - W_p - W_f}{n_r 2\pi}$$

$$W_{ig} = m_{fc} q_{HV} \eta_{ig}$$
$$W_p = V_d(p_e - p_i)$$
$$W_f = V_d(N_e^2 C_{f2} + N_e C_{f1} + C_{f0})$$

7.3.3 Bränsleregulator

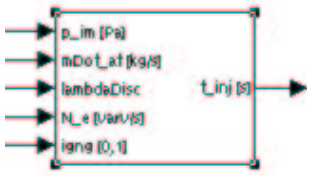
För att få optimal bränsleförbränning regleras λ. Regulatorn består av en återkoppling av λ_{disc} som är en PI-regulator och en framkoppling som predikterar insprutningstiden.

$$reg = K_p \lambda_{disc} + K_i \int \lambda_{disc} dt$$

$$t_{ref} = \frac{\frac{\dot{m}_{ac}}{(A/F)_s}}{kN_e} + t_0$$

$$t_{inj} = t_{ref} reg$$

Insignaler	p_{im}, $\dot{m}_{at}$, λ_{disc}, N_e, $igng$
Utsignaler	t_{inj}



Figur 13: Blockschema över Bränsleregulator

7.4 Tomgångsregulator

Består av en enkel P-regulator som håller varvtalet runt 800 rpm vid tomgångskörning.

Insignaler	RPM
Utsignaler	$Pedal_{Pos}$

Figur 14: Blockschema över Tomgångsregulator

7.5 Varvtalsregulator

Består av en regulator som håller varvtalet begränsat. Den består av en P-regulator som börjar verka efter 7000 RPM. Detta för att skona motorn.

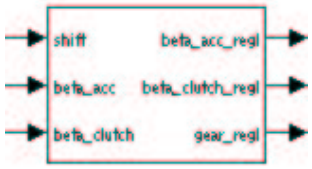
Insignaler	RPM
Utsignaler	$Pedal_{Pos}$

Figur 15: Blockschema över Varvtalsregulator

7.6 Växlare

När signalen $shift$ ändras till 1 eller -1 ska sekvetiell växling ske. En kontroll bör göras så att inte signalen $gear$ har värdet av högsta eller lägsta växeln. I sådant fall ska ändringen av $shift$ ignoreras. Växlingsförloppet sker på samma sätt som en manuell växling. Kopplingen trycks ned samtidigt som gasen släpps upp genom att β_{clutch} ändras från 1 till 0 och β_{acc} ändras till 0. Signalen β_{acc} ska alltså tillfälligt styras automatiskt och inte ta någon hänsyn till ROP. Nu är motorn frikopplad och växling kan ske. Detta görs genom att ändra signalen $gear$ till $gear + shift$. Nu ges kontrollen över β_{acc} åter till ROP och kopllingen släpps upp genom att ändra β_{clutch} till 1.

Insignaler	$shift$, β_{acc}, β_{clutch}
Utsignaler	$\beta_{acc.regl}$, $\beta_{clutch.regl}$, $gear_{regl}$

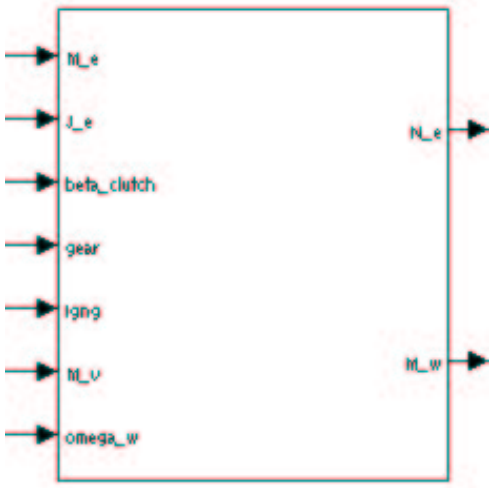


Figur 16: Blockschema över Växlare

7.7 Drivlina

Drivlinan är implementerad som stel. Realistiska förluster, så som friktion, har tagits hänsyn till för att göra drivlinemodellen så realistisk som möjligt. En öppen diffrential finns även implementerad. Från motorn kommer ett drivande moment som skalas genom utväxlingar och minskas med de förluster som finns. Detta moment förs över till blocket orientering där hjulens rotationshastighet beräknas. Denna hastighet skickas sedan tillbaka genom genom subblocken i drivlinan. Slutligen bestäms motorns rotationshastighet utifrån detta samt växelval, kopplingsläge osv. Signalerna *igng* och *start/stopp* används för att ge motorn rätt hastighet vid start och stopp samt för att säkerställa att varvtalet blir 0 när motorn inte är igång.

Insignaler	M_e, J_e, β_{clutch}, $gear$, $igng$, M_v, ω_w, $start/stopp$
Utsignaler	N_e, M_w

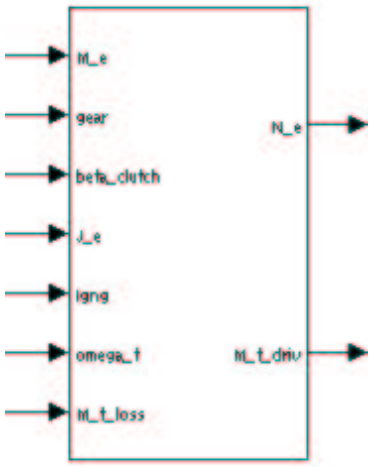


Figur 17: Blockshema över Drivlina

7.7.1 Koppling/Växellåda

Kopplingen är implemeterad så att om ingen växel är i eller om motorn är helt frikopplad kommer motorns utmoment att integreras och läggas till den hastighet motorn har innan frikoppling/friläge. När kopplingen trycks ner eller släpps upp kommer den att slira lite till hastighetsskillnaden är så liten att lamellerna låser ihop. Därefter får motorn samma rotation som drivlinan. Det bromsande moment som verkar på motorn dras ifrån motorns utmoment och skickas ut till växellådan. Växellådan är förlustfri och skalar endast moment och rotation med i_t, utväxlingsförhållandet. Se även Embedded m-filen.

Insignaler	M_e, $gear$, β_{clutch}, J_e, $igng$, ω_t, M_{tloss}
Utsignaler	N_e, M_{tdriv}



Figur 18: Blockshema över Koppling/Växellåda

7.7.2 Kardanaxel

Tar hänsyn till förlusterna i kardanaxeln.

$$M_k = M_t - \omega_k F_k \qquad \omega_k = \omega_t \qquad M_{kloss} = M_{fdloss} + \omega_k F_k$$

Insignaler	M_t, ω_k, M_{fdloss}
Utsignaler	M_k, ω_t, M_{kloss}

Figur 19: Blockshema över Kardanaxel

7.7.3 Slutväxel

Slutväxeln fördelar momentet från kardanaxeln till de drivande hjulen med en utväxlingsfaktor i_f minus friktionsförluster. Kardanaxelns varvtal räknas ut utifrån vinkelhastigheten på drivaxlarna. Detta för att den sekventiella växlingen ska ske så realistiskt som möjligt.

$$M_{fd} = (M_k i_f - dF_{fd}\omega_{fd})c \qquad \omega_k = d\omega_{fd} i_f \qquad M_{fd,loss} = \frac{dF_{fd}\omega_{fd} + M_{d,loss}d}{i_f}$$

d är en 1x4 matris som skickar kraften från de drivande hjulen och noll från de rullande hjulen.
c är en 1x4 matris som fördelar momentet från kardanaxeln ut på drivaxeln.

Insignaler	M_k, ω_{fd}, M_{dloss}
Utsignaler	M_{fd}, ω_k, M_{fdloss}

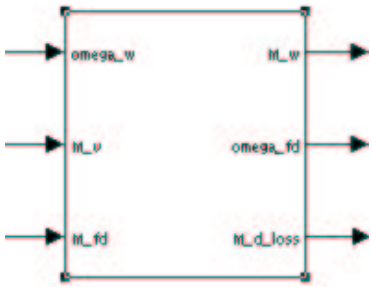
Figur 20: Blockshema över Slutväxel

7.7.4 Drivaxel

Drivaxeln implementeras som stel vilket medför att momentet går direkt ut på hjulen med en viss förlust pga friktion.
M_v är de bromsande krafterna på varje hjul som återkopplas för att bromsa motorn.

$$M_w = M_{fd} - \omega_w F_d \qquad M_{d,loss} = M_v + \omega_w F_d \qquad \omega_w = \omega_{w_{fd}}$$

Insignaler	ω_w, M_v, M_{fd}
Utsignaler	M_w, ω_{fd}, M_{dloss}

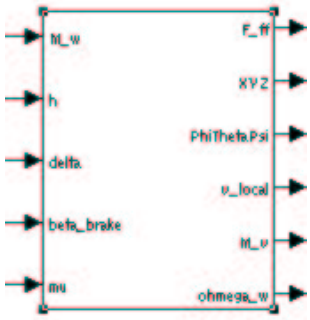


Figur 21: Blockshema över Drivaxel

7.8 Orientering

Modell av fordonets hjul/framdrivning. De krafter och moment som verkar på fordonskroppen beräknas i detta block. De beräknas i bilens respektive hjulens koordinatsystem och därefter beräknas/integreras dessa till hastigheter och rotationshastigheter. Därefter transformeras dessa till globala koordinater och integreras till position och rotation

Insignaler	M_w, h, δ, β_{brake}, μ
Utsignaler	F_{ff}, x, y, z, Φ, Θ, Ψ, v_{local}, M_v, ω_w



Figur 22: Blockschema över Orientering

7.8.1 Hjulupphängning

Här beräknas normalkrafterna på hjulen. De är en olinjär funktion av hoptryckning och hoptryckninshastighet för att simulera att fjädringen bottnar samt att dämparna blir styvare vid högre hoptryckningshastighet. Vidare kan normalkrafterna ej bli negativa eller vara annat än 0 när ingen markkontakt finns.

$$
\begin{aligned}
z_1 &= Z - a\Theta + c\Phi - h_1(X + a\cos\Psi - c\sin\Psi, Y + a\sin\Psi + c\cos\Psi) \\
z_2 &= Z - a\Theta + c\Phi - h_2(X + a\cos\Psi - c\sin\Psi, Y + a\sin\Psi - c\cos\Psi) \\
z_3 &= Z + b\Theta + c\Phi - h_3(X + a\cos\Psi - c\sin\Psi, Y - b\sin\Psi + c\cos\Psi) \\
z_4 &= Z + b\Theta + c\Phi - h_4(X + a\cos\Psi - c\sin\Psi, Y - b\sin\Psi - c\cos\Psi)
\end{aligned}
$$

där $h_i(X, Y)$ beskriver vägens profil.

$$\dot{z}_i = (v + \omega \times r_i) \cdot \hat{z}, i = 1, 2, 3, 4$$

Uttrycken nedan beskriver den olinjära hjulupphängningen. Uttrycken i tangensfunktionen är begränsade för att undvika simuleringsproblem som uppstår när tangensfunktionen går mot oändligheten samt byter tecken.

$$F_{fjader} = -k_{spring} \tan(\frac{\pi}{2} \frac{z_i}{z_{max}})$$

$$F_{dampare} = -c_{dampare} \tan(\frac{\pi}{2} \frac{\dot{z}_i}{\dot{z}_{i\,max}})$$

Insignaler	x, y, z, Φ, Θ, Ψ, v, ω, h
Utsignaler	N

Figur 23: Blockschema över Hjulupphängning

7.8.2 Pacejkas magic formula

Innehåller Pacejkas empiriska magic formula. Beräknar longitudinella och laterala krafter på alla hjul. De beräknas utifrån hjulhastigheter och rotationshastigheter i hjulens koordinatsystem. Dessutom bestäms M_z som skickas till force feedback för att generera den kraft som läggs på ratten. F_i är en 2-vektor som innehåller krafter i x- och y-led i hjulens lokala koordinatsystem. F_{ff} är den kraft som ska läggas på ratten och ligger mellan -1 och 1. Friktionselipsen är inte implementerad.

Beräkning av longitudinellt slip, individuellt för varje hjul:

$$s = \begin{cases} \frac{r\omega_{hjul} - v_x}{|r\omega_{hjul}|}, & \text{om hjul snurrar} \\ \frac{r\omega_{hjul} - v_x}{|v_x|}, & \text{om hjul ej snurrar och bil rör sig} \\ r\omega_{hjul} - v_x, & \text{om bil och hjul stilla.} \end{cases}$$

Beräkning av slipvinkeln, individuellt för varje hjul:

$$\alpha = \begin{cases} -\arctan(\frac{v_y}{v_x})\frac{2}{\pi}, & \text{om bil kör framåt} \\ \arctan(\frac{v_y}{v_x})\frac{2}{\pi}, & \text{om bil kör bakåt} \\ -\arctan(\frac{v_y}{0.1})\frac{2}{\pi}, & \text{om bil glider rakt i sidled åt vänster} \\ \arctan(\frac{v_y}{-0.1})\frac{2}{\pi}, & \text{om bil glider rakt i sidled åt höger} \end{cases}$$

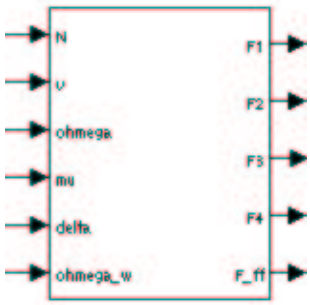
Däckskrafterna beräknas med Pacejkas magic formula med s och α som parametrar:

$$F_x = D\sin(C\arctan(Bs - E(Bs - \arctan Bs)))$$

$$F_y = D\sin(C\arctan(B\alpha - E(B\alpha - \arctan B\alpha)))$$

Parametrarna B, C, D, E är däcksparametrar, se bifogad m-fil.

Insignaler	N, v, ω, μ, δ, ω_w
Utsignaler	$F1$, $F2$, $F3$, $F4$, F_{ff}

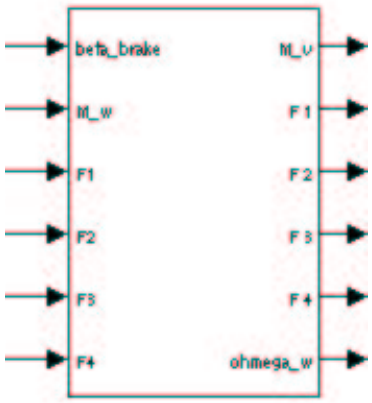


Figur 24: Blockschema över Pacejkas magic formula

7.8.3 Hjulhastighet

I detta block beräknas hjulhastighet, bromsande moment från bromssystem på hjul, krafter från däck på fordonet och bromsande moment från bromssytem och hjul på drivlinan. Här finns även ett rullmotstånd implementerat. Krafter på hjul i x-led överförs till fordonet men begränsas av hur mycket fordonet själv sätter in, se embedded m-filen Kraft på bil. Fritt rullande hjul kan inte överföra någon kraft till bilen men om man bromsar det kan den överföra kraft. I y-led överförs alla däckskrafter.

Insignaler	β_{brake}, M_w, $F1$, $F2$, $F3$, $F4$
Utsignaler	M_v, $F1$, $F2$, $F3$, $F4$, ω_w



Figur 25: Blockschema över Hjulhastighet

7.8.4 Fartvind och rullmotstånd

Luft- och rullmotstånd beräknas och summeras på enklaste sätt med hjälp av parameterskattning av en andragradskurva. Hämtat från Lab. 1c i Fordonssystem.

Insignaler	v
Utsignaler	F_{fordon}

Figur 26: Blockschema över Fartvind och rullmotstånd

7.8.5 F tot

Krafter på hjulen transformeras från hjulens koordinatsystem till bilens och därefter summeras de. Vidare räknas tyngdkraft och normalkraft in.

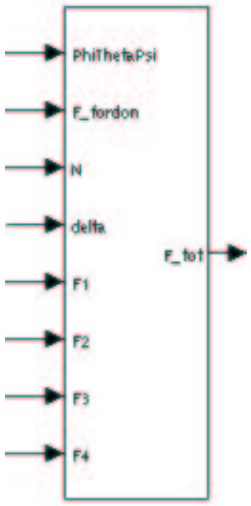
$$F_x = cos\alpha Fx_1 - sin\alpha F_{y_1} + cos\alpha Fx_2 - sin\alpha F_{y_2} + Fx_3 + F_{x_4} - mg \cdot f_x(\Theta, \Phi, \Psi) - F_{fordon}$$

$$F_y = sin\alpha Fx_1 + cos\alpha F_{y_1} + sin\alpha Fx_2 + cos\alpha F_{y_2} + Fy_3 + F_{y_4} - mg \cdot f_y(\Theta, \Phi, \Psi)$$

$$F_z = N_1 + N_2 + N_3 + N_4 - mg \cdot f_z(\Theta, \Phi, \Psi)$$

där $N_i = f_N(z_i, \dot{z}_i), F_{x_i}, F_{y_i}$ är kraft på hjul i hjulets lokala koordinatsystem och $f_i(\Theta, \Phi, \Psi)$ är projektionen av global Z-axel på lokal axel i.

Insignaler	F_{fordon}, Φ, Θ, Ψ, N, δ, $F1$, $F2$, $F3$, $F4$
Utsignaler	F_{tot}



Figur 27: Blockschema över F_{tot}

7.8.6 M tot

Krafter på bile n från hjulen på bilen summeras till ett resulterande moment.

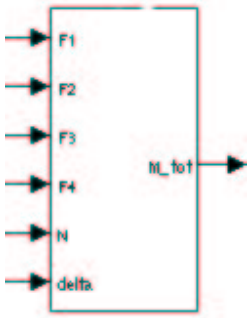
$$M_x = cN_1 - cN_2 + cN_3 - cN_4$$

$$M_y = -aN_1 - aN_2 + bN_3 + bN_4$$

$$M_z = (a\sin\alpha - c\cos\alpha)F_{x_1} + (a\cos\alpha + c\sin\alpha)F_{y_1} +$$
$$+(a\sin\alpha + c\cos\alpha)F_{x_2} + (a\cos\alpha - c\sin\alpha)F_{y_2} -$$
$$-cF_{x_3} - bF_{y_3} - cF_{x_4} + cF_{y_4}$$

d.v.s $M_{tot} = \sum r \times F$

Insignaler	$F1$, $F2$, $F3$, $F4$, N, δ
Utsignaler	M_{tot}

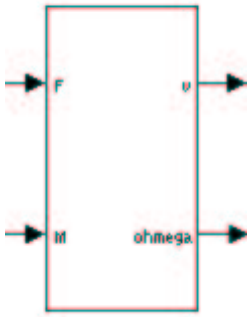


Figur 28: Blockschema över M_{tot}

7.8.7 Dynamik

Krafter och moment på fordonet integreras till hastighet och rotationshastighet för fordonet.

Insignaler	F_{tot}, M_{tot}
Utsignaler	v, ω

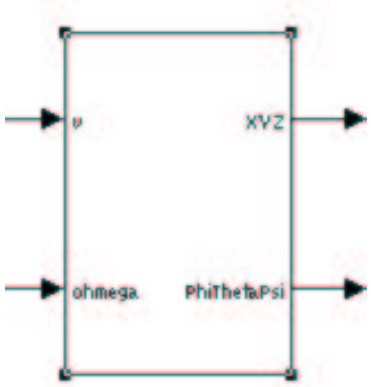


Figur 29: Blockschema över Dynamik

7.8.8 Orientering

Hastighet och rotationshastighet transformeras från bilens koordinatsystem till globala koordinatsystemet. Därefter integreras dessa till position och rotation på bilen i globala koordinater.

Insignaler	v, ω
Utsignaler	XYZ, $\Phi\Theta\Psi$



Figur 30: Blockschema över Orientering

8 Kommunikationsmodul

8.1 Klienten

Klienten är den klassen som varje program som vill använda sig av kommunikationsgränssnittet skall instantiera. Klienten har en lista med meddelanden som den har mottagit. En parameter i meddelandet säger om det skall köas upp eller om ett eventuellt tidigare obehandlat meddelande av samma typ skall skrivas över. Meddelanden av olika typ kommer alltid att köas upp. För att klienten skall uppdatera sin lista med meddelanden från nätverket måste funktionen `Client::UpdateThreads()` köras med jämna mellanrum.

Klienten har endast en UDP-uppkoppling, och denna är till servern. Vilken address klienten skall koppla upp sig mot tas in som parameter till konstruktorn av klientklassen.

8.2 Meddelanden

Det skall finnas en meddelandeklass för varje sorts meddelande som skall kunna skickas. Förutom dessa finns även meddelandetypen FsimMessage, som alla meddelanden skall ärva ifrån. Varje meddelandeklass måste förutom det som man vill kunna lagra/skicka i meddelandet innehålla en konstruktor för att skapa ett nytt meddelande från parametrar, samt ett en konstruktor för att skapa/parsa ett meddelande från en inkommande sträng. Dessa två konstruktorer måste även köra respektive konstruktor i basklassen FsimMessage, för att de skall bli korrekta meddelanden.

Nya meddelandeklasser kan skapas genom följande steg:

1. Skapa klassen och skriv dess konstruktorer på samma sätt som för en befintlig meddelandeklass. Glöm inte get- och set- funktioner.
2. Lägg till ett ID för meddelandetypen i msgtype (finns i `Message.h`)
3. Lägg till att `FsimMessage::IdentifyMsgType()` returnerar en instans av den nya meddelandetypen.

Nya gemensamma parametrar kan läggas till genom att ändra i `FsimMessage::ParseMessage()` samt i konstruktorn som tar in vilka parametrar som skall tas in.

8.3 Meddelandestruktur

Meddelanden är skrivna i ASCII-format. Alla meddelanden börjar med startsymbolen §. Därefter kommer ett antal parametrar som är gemensamma för alla sorters meddelanden. Dessa är en enum för identifikation av meddelandet, samt en integer för mottagaren av detta meddelande. Mellan varje parameter finns symbolen ¶ för att separera parametrarna från varandra. Den gemensamma delen av meddelandet avslutas med symbolen +.

Direkt efter den gemensamma delen följer de unika parametrarna. Dessa är också separerade av symbolen ¶. Antalet unika parametrar varierar mellan olika meddelandetyper. Slutligen avslutas varje meddelande av att slutsymbolen þkommer. För närvarande finns en begränsning av att ett meddelande högst får innehålla 256 tecken. Detta kan dock lätt utökas. Dessutom är alla separatorsymboler lätta att byta ut.

Som exempel på hur ett meddelande kan vara uppbyggt kan visas hur meddelandetypen IMFMMessage lagras:

§MessageType¶ToID¶enqueue+Acceleration¶Brake¶SteeringWheel¶Shiftþ

Detta kan se ut i verkligheten som:

§4¶16¶0+100¶150¶156¶0þ och om fordonsmodulen tog emot detta meddelande skulle den tolka det som att gaspedalen är nedtryckt 100/255, bromsen 150/255, ratten är på position 156/255, och vi växlar varken upp eller ned. Enqueue är satt till 0, vilket innebär att meddelandet inte ska köas i den mottagande klienten.

8.4 Mottagande av ett meddelande av klienten

Meddelanden i klienten lagras som en länkad lista med objekt av typen FsimMessage. Detta är basklassen för alla meddelanden, och alla meddelanden som ärver från denna klass kan lagras i klienten. För att läsa informationen i varje meddelande måste man typecasta den till rätt sorts meddelande igen. Funktionen `Client::UpdateThreads()` tar emot alla meddelanden som hittills inte har behandlats, och stoppar in dem i den länkade listan av meddelanden som klienten tillhandahåller.

Då ett meddelande tas emot kommer klienten att köra funktionen `FsimMessage::IdentifyMsgType()`. Denna returnerar ett objekt av rätt typ, parsad så att alla parametrar är korrekt satta. Det är dessa objekt som lagras i klientens lista med meddelanden. Konstruktorn för att parsa ett meddelande körs med andra ord inte av någon annan än denna funktion.

När sedan applikationen som instantierar klienten vill ta emot ett meddelande körs `Client::GetNextMessage()`. Denna tar bort meddelandet ut klientens lista och returnerar detta till användaren av klienten. Denna måste sedan undersöka om meddelandet är något som är intressant genom att typecasta till de klasser av meddelanden som användaren av klienten bryr sig om. Observera att användaren av klienten måste ta bort meddelandet efter sin användning, då detta inte görs av klienten.

Ett exempel på mottagning kan då se ut som följande, om en applikation är intresserad av meddelanden av typen SoundMessageOnce och klienten är instantierad som myClient*.

```
⋮
FsimMessage *mess = myClient->GetNextMessage();
SoundMessageOnce* soundmess = dynamic_cast<SoundMessage>(mess);
if (soundmess)
{
        //Gör allt som skall göras med meddelandet.
        soundmess->GetFilename();
        ...
        delete soundmess;
}
else delete mess;
⋮
```

8.5 Servern

Servern skiljer sig från klienten på två punkter:

1. Den har lika många UDP-uppkopplingar som klienter som har kopplat upp sig mot denna.
2. Den tar emot alla meddelanden, och sållar inte bort meddelanden även om de skulle råka vara av samma typ.

I övrigt kan sägas att det enda servern gör är att ta emot ett meddelande från en klient, analysera till vilken annan klient det skall skickas, och skicka det. För närvarande stödjer servern upp till 10 uppkopplingar. Detta nummer kan dock lätt ökas.

8.6 Uppkopplingsförfarandet

Det första som händer när en klient vill koppla upp sig mot servern är att klienten skickar strängen "ID,myID,END", där myID är ett godtyckligt nummer som denna klient vill kännas under. Alla meddelanden som är addresserade till detta nummer kommer att skickas till denna klienten. När servern har mottagit detta skickar servern sitt ID tillbaka till klienten på samma sätt, så att klienten vet om att den är uppkopplad.

8.7 Klasser

Router - basklass för Client och Server.

Header	`virtual void ReceiveMessage(std::string theString)`
Argument	*theString* är den sträng som det skall byggas ett meddelande ifrån.
Returnerar	

Client - gränssnitt för användning av kommunikationsmodulen.

Header	`Client(tID)`
Argument	*tID* är id-numret som denna klient skall ha.
Returnerar	-

Header	`void Connect(char tIP[15], int tPort)`
Argument	*tIP* är IP:t som man klienten vill koppla upp sig mot. *tPort* är porten som man vill göra detta på
Returnerar	-

Header	`void Disconnect()`
Argument	-
Returnerar	-

Header	`void UpdateThreads()`
Argument	-
Returnerar	-

Header	`void ReceiveMessage(std::string theString)`
Argument	*theString* är en sträng som har mottagits från nätverket och är redo att parsas.
Returnerar	-

Header	`FsimMessage* GetNextMessage()`
Argument	-
Returnerar	Returnerar ett färdigparsat meddelande som har mottagits från nätverket

Header	`bool Send(FsimMessage *theMessage)`
Argument	*theMessage* är det meddelande som man vill skicka. Det kan vara av vilken klass som helst så länge som den ärver från FsimMessage.
Returnerar	returnerar *true* om den lyckades att sända meddelandet.

Header	`private void AddToReceivedMessageQueue(FsimMessage *newElement)`
Argument	*newElement* är det meddelande som skall läggas in i klientens meddelandelista
Returnerar	-

Server - gränssnitt för användning av servern. I samma fil som klassen Server finns en main-funktion som instantierar denna.

Header	`Server(long int tID)`
Argument	*tID* är det id som man önskar att servern skall kännas vid
Returnerar	-

Header	`void UpdateThreads()`
Argument	-
Returnerar	-

Header	`void UpdateMessages()`
Argument	-
Returnerar	-

Header	`void ReceiveMessage(std::string theString)`
Argument	*theString* är den sträng som man vill skapa meddelandet från
Returnerar	-

Header	`private bool SendToUser(FsimMessage *theMessage)`
Argument	*theMessage* är det meddelande som skall skickas
Returnerar	returnerar ifall försändelsen lyckades

Header	`void AddToReceivedMessageQueue(FsimMessage *newElement)`
Argument	*newElement* är det meddelande som man vill stoppa in i serverns meddelandelista
Returnerar	-

Connection - basklass för uppkopplingar. Innehåller det som är gemensamt för ClientConnection och ServerConnection. Denna bör skrivas om något om man skulle vilja använda något annat operativsystem än windows. Dock bör detta inte inte några större omstruktureringar.

Header	`Connection()`
Argument	-
Returnerar	-

Header	`void Disconnect()`
Argument	-
Returnerar	-

Header	`virtual bool SendThisMessage(FsimMessage* theMessage)`
Argument	*theMessage* är det meddelande som man önskar sända
Returnerar	returnerar true om uppkopplingen har lyckats sända meddelandet

Header	`virtual void GetFromSockets()`
Argument	-
Returnerar	-

Header	`protected void SetNonBlocking(const bool b)`
Argument	*b* = true om man önskar icke-blockerande sockets
Returnerar	-

ClientConnection - tillhandahåller UDP-uppkopplingen till Client. Denna bör också skrivas om något om Windows inte används.

Header	`ClientConnection(char* address, int port, long int aID, Router* aRouter)`
Argument	*address* är addressen som man ska koppla upp sig mot. *port* är den port som man vill koppla upp sig mot. *aID* är klientens ID. *aRouter* är en callback till Client.
Returnerar	-

Header	`bool SendThisMessage(FsimMessage* theMessage)`
Argument	*theMessage* är det meddelande som man vill att ClientConnection skall sända.
Returnerar	Returnerar om den ClientConnection lyckades skicka meddelandet.

Header	`void GetFromSockets()`
Argument	-
Returnerar	-

Header	`private void ConnectToServer()`
Argument	-
Returnerar	-

ServerConnection - tillhandahåller UDP-uppkopplingen till Server. Denna bör också skrivas om något om Windows inte används.

Header	`ServerConnection(long int aID, Router* aRouter)`
Argument	*aID* är serverns ID *aRouter* är en callback till ServerConnection.
Returnerar	-

Header	`bool SendThisMessage(FsimMessage* theMessage)`
Argument	*theMessage* är meddelandet som man önskar att servern ska skicka vidare till en klient
Returnerar	Returnerar *true* om man lyckades skicka iväg meddelandet

Header	`void GetFromSockets()`
Argument	-
Returnerar	-

FsimMessage - basklass för meddelanden. Innehåller det som alla meddelanden måste innehålla.

Header	`FsimMessage(int aToID, msgtype aType, bool aEnqueue)`
Argument	*aToID* är vilken modul som är mottagare av meddelandet. *aType* är vilken typ av meddelande detta är. *aEnqueue* = true betyder att meddelandet skall köas upp på mottagarsidan.
Returnerar	-

Header	`FsimMessage(std::string aMessage)`
Argument	*aMessage* är den meddelandesträng som meddelandet skall byggas upp ifrån
Returnerar	-

Header	`std::string GetBody()`
Argument	-
Returnerar	Returnerar strängen som meddelandet är uppbyggd ifrån.

Header	`long int GetToID()`
Argument	-
Returnerar	Returnerar vilket ID meddelandet är addresserat till.

Header	`void SetToID(long int aToID)`
Argument	*aToID* sätter vem meddelandet är addresserat till.
Returnerar	-

Header	`char GetType()`
Argument	-
Returnerar	Returnerar vilken typ meddelandet är.

Header	`bool GetEnqueue()`
Argument	-
Returnerar	Returnerar *true* om meddelandet skall köas på mottagarsidan.

Header	`static FsimMessage* IdentifyMsgType(std::string& aMessage)`
Argument	*aMessage* är den sträng som skall parsas, och identifiera typen av meddelande som skall skapas.
Returnerar	Returnerar ett meddelande av en klass som ärver från FsimMessage.

Header	`protected void ParseMessage(std::string& tMessageBuf)`
Argument	*tMessageBuf* är den meddelandesträng som skall parsas
Returnerar	-

ForceFeedbackMessage - Meddelande för överföring av ForceFeedback till inputmodulen.

Header	`ForceFeedbackMessage(long int aToID, double aForce, double aTime)`
Argument	*aToID* är vem meddelandet är till. *aForce* är vilken kraft som skall läggas på. *aTime* är under hur lång tid kraften skall läggas på
Returnerar	-

Header	`ForceFeedbackMessage(std::string theMessage)`
Argument	*theMessage* är den meddelandesträng som skall parsas.
Returnerar	-

IMVMMessage - Meddelande för överföring av rattvinkel till visualiseringsmodulen.

Header	`IMVMMessage(long int aToID, int aX, int aY, double aAngle)`
Argument	*aToID* är vem meddelandet är till. *aAngle* är hur mycket ratten är vriden.
Returnerar	-

Header	`IMVMMessage(std::string theMessage)`
Argument	*theMessage* är den meddelandesträng som skall parsas.
Returnerar	-

IMVMCameraMessage - Meddelande för att byta kameravinkel.

Header	`IMVMCameraMessage(long int aToID)`
Argument	*aToID* är vem meddelandet är till.
Returnerar	-

Header	`IMVMCameraMessage(std::string theMessage)`
Argument	*theMessage* är den meddelandesträng som skall parsas.
Returnerar	-

IMFMMessage - Meddelande för att överföra information om gaspedal, rattvinkel och broms till fordonsmodulen.

Header	`IMFMMessage(long int aToID, int aAcc, int aBrake, int aSw, int aShift)`
Argument	*aToID* är vem meddelandet är till. *aAcc* är hur mycket gaspedalen är nedtryckt. *aBrake* är hur mycket bromspedalen är nedtryckt. *aSw* är hur mycket ratten är vriden. *aShift* sätt till 1 för uppväxling och -1 för nedväxling.
Returnerar	-

Header	`IMFMMessage(std::string theMessage)`
Argument	*theMessage* är den meddelandesträng som skall parsas.
Returnerar	-

FMVMMessage - Meddelande för att överföra position samt vinkel på bilen till visualiseringsmodulen.

Header	`FMVMMessage(long int aToID, double aX, double aY, double aZ, double aRx, double aRy, double aRz, double aSpeed)`
Argument	*aToID* är vem meddelandet är till. *aX* är bilens X-position i världen. *aY* är bilens Y-position i världen. *aZ* är bilens Z-position i världen. *aRx* är bilens rotation kring X-axeln. *aRy* är bilens rotation kring Y-axeln. *aRz* är bilens rotation kring Z-axeln.
Returnerar	-

Header	`FMMVMessage(std::string theMessage)`
Argument	*theMessage* är den meddelandesträng som skall parsas.
Returnerar	-

VMFMMessage - Meddelande för att överföra information om väglag samt höjd på väg under hjulen till fordonsmudulen.

Header	`VMFMMessage(long int aToID, double aHfl, double aHfr, double aHbl, double aHbr, double aMyfl, double aMyfr, double aMybl, double aMybr)`
Argument	*aToID* är vem meddelandet är till. *aHfl* är den absoluta höjden på marken under bilens främre vänstra hjul. *aHfr* är den absoluta höjden på marken under bilens främre högra hjul. *aHbl* är den absoluta höjden på marken under bilens bakre vänstra hjul. *aHbr* är den absoluta höjden på marken under bilens bakre högra hjul. *aMyfl* är friktionen under bilens främre vänstra hjul. *aMyfr* är friktionen under bilens främre högra hjul. *aMrbl* är friktionen under bilens bakre vänstra hjul. *aMybr* är friktionen under bilens bakre högra hjul.
Returnerar	-

Header	`VMFMMessage(std::string theMessage)`
Argument	*theMessage* är den meddelandesträng som skall parsas.
Returnerar	-

ShudownMessage - Meddelande för att säga till att det är dags att avsluta simuleringen.

Header	`ShutdownMessage(long int aToID)`
Argument	*aToID* är vem meddelandet är till. .
Returnerar	-

Header	`ShudownMessage(std::string theMessage)`
Argument	*theMessage* är den meddelandesträng som skall parsas.
Returnerar	-

SoundOnceMessage - Meddelande om att ett ljud skall spelas en gång.

Header	`SoundMessageOnce(long int aToID, std::string aFilename, int aVolume)`
Argument	*aToID* är vem meddelandet är till. *aFilename* är vilket ljud som skall spelas. *aVolume* är vilken volym som ljudet skall spelas upp med.
Returnerar	-

Header	`SoundOnceMessage(std::string theMessage)`
Argument	*theMessage* är den meddelandesträng som skall parsas.
Returnerar	-

SoundContMessage - Meddelande om att ett ljud skall spelas upp kontinuerligt.

Header	`SoundMessageCont(long int aToID, std::string aFilename, int aFreq, int aVolume)`
Argument	*aToID* är vem meddelandet är till. *aFilename* är vilket ljud som skall spelas. *aFreq* är med vilken frekvens ljudet skall spelas upp. *aVolume* är vilken volym som ljudet skall spelas upp med.
Returnerar	-

Header	`SoundContMessage(std::string theMessage)`
Argument	*theMessage* är den meddelandesträng som skall parsas.
Returnerar	-

SoundStopMessage - Meddelande om att ett ljud som spelas upp kontinuerligt skall sluta spelas upp.

Header	`SoundMessageStop(long int aToID, std::string aFilename)`
Argument	*aToID* är vem meddelandet är till. *aFilename* är filnamnet på ljudet som skall stoppas.
Returnerar	-

Header	`SoundStopMessage(std::string theMessage)`
Argument	*theMessage* är den meddelandesträng som skall parsas.
Returnerar	-

FMLMMessage - Meddelande om ändring av volym/frekvens för sladd- och motorljud.

Header	`FMLMMessage(long int aToID, int aVol1, int aFreq1, int aVol2, int aFreq2)`
Argument	*aToID* är vem meddelandet är till. *aVol1* är vilken volym motorljudet skall ha. *aFreq1* är vilken frekvens motorljudet skall spelas upp med *aVol2* är vilken volym sladdljudet skall ha. *aFreq2* är vilken frekvens sladdljudet skall spelas upp med.
Returnerar	-

Header	`FMLMMessage(std::string theMessage)`
Argument	*theMessage* är den meddelandesträng som skall parsas.
Returnerar	-

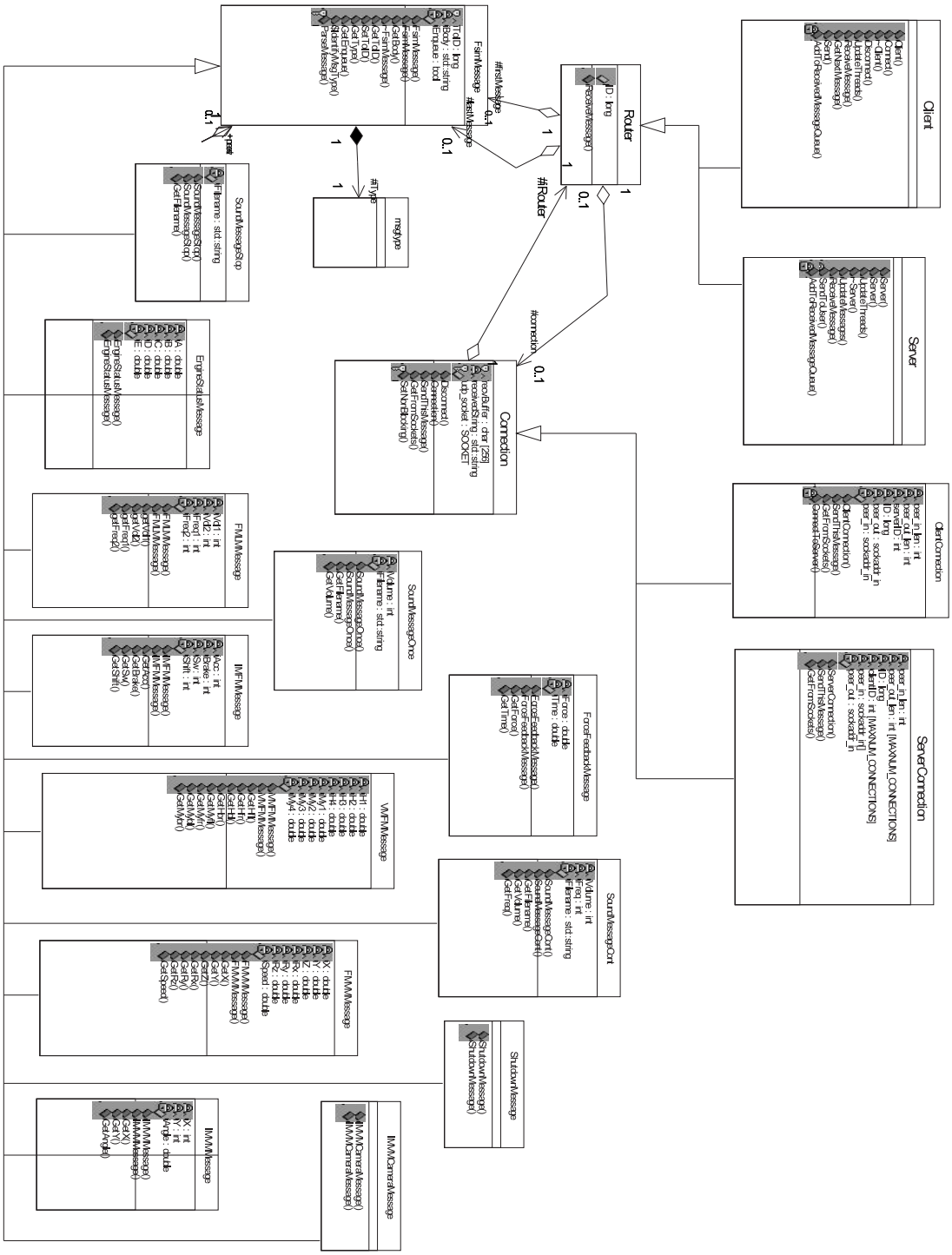
EngineStatusMessage - Debugmeddelande för att skicka diverse status om fordonsmodulen.

Header	`EngineStatusMessage(long int aToID, double aA, double aB, double aC, double aD, double aE)`
Argument	*aToID* är vem meddelandet är till. Alla övriga argument är godtyckliga.
Returnerar	-

Header	`EngineStatusMessage(std::string theMessage)`
Argument	*theMessage* är den meddelandesträng som skall parsas.
Returnerar	-

8.8 Klassdiagram

Klassdiagrammet för kommunikationsmodulen visas i figur 31.



Figur 31: Klassdiagram över kommunikationsmodulen

A Appendix

A.1 fordonsparametrar.m

```
%TSRT71 Reglerteknisk projektkurs CDIO
%Projekt NightRider, fordonssimulator
%Parametrar för fordonet

T = 0.01;                   %Simuleringstid
rps2rpm = 60;               %Omvandlingsfaktor

%==========Växlare==========
acc_releasetime = 0.2;      %Gaspedalens uppsläppningstid (1-0).
acc_presstime = 0.2;        %Gaspedalens nedtryckningsstid (0-1).
clutch_presstime = 0.1;     %Kopplingspedalens nedtryckningstid.
clutch_releasetime = 0.5;   %Kopplingspedalens uppsläppningstid.
total_shifttime = 0.8;      %Önskad total tid för växlingsförloppet.
min_shifttime = max(acc_releasetime,clutch_presstime) + ...
    max(clutch_releasetime,acc_presstime); %Minsta växlingstid
init_gear = 0;              %Startväxel.

%==========Orientering==========
g = 9.81;                   %Gravitation
r = 0.25;                   %Hjulradie
maxratt = 35*pi/180;        %Maximalt hjulvinkel
%Geometri
%beräkning av tyngdpunkt och tröghetsmoment för ett fordon
%som approximeras till en punkmassa mellan framhjulen(motor)
%och en platta med uniformt fördelad massa mellan hjulen
m_c = 1350;                  %Chassits massa
m_m = 250;                  %Motorns massa
B = 2;                      %Bredd mellan hjulen
H = 1;                      %Plattans höjd
L = 4;                      %Längd mellan hjulen
I_G_platta = diag([m_c*(B^2+H^2)/12,m_c*(L^2+H^2)/12,m_c*(L^2+B^2)/12]);
md_2 = diag([0,m_c*m_m/(m_c+m_m)*(L^2)/4,m_c*m_m/(m_c+m_m)*(L^2)/4]);
I_G_car = I_G_platta+md_2;  %Steiners sats: I_G = I_G' +md^2
inv_I_G = inv(I_G_car);     %Tröghetsmatris
m = m_c+m_m;                %Bilens massa
a = m_c/(m_c+m_m)*L/2;      %Längd från tyngdpunkten till fronten
b = L-a;                    %Längd från tyngdpunkten till baken
c = B/2;                    %Längd från tyngdpunkten till sidan
d = H/2;                    %Längd från tyngdpunkten till bottenplattan
Z_0 = 0.69;                 %Höjd från marken till fjädern

%Hjulupphängning
```

```
k_spring = 20000;           %fjäderkonstant
c_damper = 100000;          %dämpkoefficient
z_max = 0.2;                %maximal ihoptryckning
z_dot_max = 15;             %maximal hoptryckningshastighet

%Hjulhastighet
J_w = 40*r^2/2;             %Hjulens tröghetsmoment
C_rullmots = m*r^2/4;

%Fartvind, rullmotstånd
air_res = 0.0180;            %F_luft=air_res*|v|^2
J_v = 3.6875;                %F_rull=J_v*|v|
%Bromskraft
brake_f = 3000;              %Bromskraft fram
brake_b = 2000;              %Bromskraft bak

%Pacejka
F_w = 0.1;                  %Rullmotstånd
brus = 0.001;                %Varians på brus i mu
F_ff = 1/1000;              %Normering av force-feedback kraften
B_c = 18;
B_l = 10;
B_M = 15;
C_c = 1.5;
C_l = 1.4;
C_M = 2;
E_c = -10;
E_l = -4;
E_M = -4;
magicx = [-1:0.01:1];
magicy_c = sin(C_c*atan(B_c*magicx-E_c*(B_c*magicx-atan(B_c*magicx))));
magicy_l = sin(C_l*atan(B_l*magicx-E_l*(B_l*magicx-atan(B_l*magicx))));
magicy_M = 0.2*sin(C_M*atan(B_M*magicx-E_M*(B_M*magicx-atan(B_M*magicx))));

%==========Tomgångsregulator==========
tomgang = 800;              %Tomgångsvarvtal
K_p_tomgang = 1/4000;        %P-förstärkning

%==========Motor==========
N_min = 200/60;             %Minsta tillåtna varvtal
p_amb = 1e5;                %Tryck i insugsröret
PI_bl = 1.85;               %Boost layout
V_d = 0.00229;              %Total slagvolym
V_i = 0.002;                %Insugningsrörets volym
n_r = 2;                    %Antal varv per tändcykel
R = 277;                    %Gaskanstanten
```

```
T_i = 310.7;                %Temperatur i insugsröret
J_e = 0.1;                  %Motorns tröghetsmoment
AF_s = 14.6;                %Stökiometriskt luft-bränsle förhållande

%Parametrar för trotteln
mDot_at_max = p_amb*PI_bl*V_d/(n_r*R*T_i); %Maximalt luftmassflöde
mDot_at_min = 0.0036;      %Minimalt luftmassflöde
min_p_i = 0;                %Minsta tillåtna insugstryck

%Parametrar för fyllnadsgrad
C_0 = 0.4781;
C_1 = 0.0011;
C_2 = -0.0036;

%Parametrar för bränslepölsdynamiken
tau_fp = 0.25;              %Tidskonstant
X = 0.25;                   %Andel bränsle som fastnar i bränslepölen

%Parametrar för bränsleinjektor
t_0 = 0.00037;              %Tidskonstant
k = 0.0078;                 %Tryckpåverkan

%Parametrar för lambdasond
tau_d = 0.08;               %Tidskonstant
tau_lambda = 0.05;          %Tidskonstant
tau_th = 0.05;              %Tidskonstant för luftmassflödsregulator.

%Parametrar för momentmodellen
C_f0 = 233820*2; %Teständring!
C_f1 = -394.8;
C_f2 = 12.06;
C_p_e = 0.0282;             %Tryckkonstant i avgasröret
n_ign = 1;                  %Optimal tändtidpunkt
n_ig_ch = 0.8582;           %Förluster i förbränningen
q_HV = 44e6;                %Uppvärmningskonstant
r_c = 9.3;                  %Kompressionsförhålllande
gamma = 1.3;                %c_v/c_p, termodynamisk egenskap hos bränslet
mDot_Fc0 = 0.0036;
N_e0 = 1;
K_motorstopp = 1;

%Bränsleregulator
K_p_lambda = 0.02;          %P-förstärkning
K_i_lambda = 0.1;           %I-förstärkning

%Varvtalsregulator
```

```
N_max = 7000;                %Maximalt varvtal
K_p_varvtal = 1/10;           %P-förstärkning

%==========Drivlina==========
%Koppling
i_t = [-12 0 16 7.5 5.5 4.5 3.2]/4.1; %Utväxlingsförhållande växellåda

%Kardanaxel
F_k = 0.01;                   %Förluster i kardanaxeln

%Slutväxel                    %Förluster i slutväxeln
F_s = 0.01;
i_f = 4.1;                    %Utväxlingsförhållande slutväxel
drivande_hjul = [0 0 1/2 1/2]; %Framhjulsdrift eller bakhjulsdrift
kraftfordelning= [0 0 1/2 1/2]; %Kraftfördelning i slutväxeln

%Drivaxel                     %Förluster i drivaxeln
F_d = 0.01;
```

A.2 Signaler

Signal	Beskrivning
acc_{pedal}	Det gaspedalsläge som når motorn då tomgångsreglering och växling tagits hänsyn till.
$gear$	Antar värdet 1 eller -1 beroend på upp- eller nerväxling.
$gear_{regl}$	Växelläge. Kan anta värden 0-5 där friläge=0 och växlarna representeras av 1-5.
$F1$,$F2$,$F3$,$F4$	Däckskrafter i lateral och longitudinell led för varje hjul i N.
F_{ff}	Force feedback som ska kännas i ratten, genereras från momentet på hjulen.
F_{fordon}	Summan av vind och rullmotstånd i N.
F_{tot}	Fordonets totala kraft i N.
h	Markens höjd vid varje hjul i meter från en referenspunkt.
$igng$	Startmotorns startsignal till motorn.
J_e	Motorns tröghetsmoment i kgm^2
$ljud$	Skickar signaler till ljudmodulen som genererar motorljudet
$\dot{m}_{fc}$	Bränslemassflöde in i cylindrarna i kg/s.
M_{dloss}	Momentförluster i drivaxeln.
M_e	Momentet ut från motorn i Nm
M_{fd}	Moment efter slutväxeln i Nm.
M_{fdloss}	Momentförlust i slutväxeln.
M_k	Moment efter kardanaxel i Nm.
M_{kloss}	Momentförlust i kardanaxeln.
M_t	Moment ut från växellådan i Nm.
M_{tloss}	Momentförlust i växellådan.
M_{tot}	Fordonets totala moment i Nm.
M_v	Bromsandemoment på drivlinan i Nm.
M_w	Hjulens moment i Nm.
N	Normalkraften vid varje hjul.
N_e	Motorns varvtal i rps.
N_{eregl}	Reglerat varvtal. Sätts till 0 vid motorstopp.
$Pedalpos$	Reglerar gaspådraget vid tomgång.
RPM	Varvtalet i rpm.
$shift$	Växlingssignal för sekventiell växling. $shift = -1$ betyder växla ned, $shift = 1$ betyder växla upp.
$start$	Startsignal till startmotorn.
v	Fordonets hastighet i m/s.
v_{lokal}	Fordonets hastighet i det egna koordinatsystemet, m/s.
x, y, z	Fordonets position i meter från en referenspunkt.

β_{acc}	Gaspedalläge. $\beta_{acc} = 0$ betyder helt uppsläppt pedal, $\beta_{acc} = 1$ betyder gasen i botten.
$\beta_{acc.regl}$	Reglerat gaspådrag vid växling .
β_{brake}	Bromspedalläge. $\beta_{brake} = 0$ betyder helt uppsläppt pedal, $\beta_{brake} = 1$ betyder maximal bromsning.
β_{clutch}	Kopplingspedalläge. $\beta_{clutch} = 1$ eftersom vi ej har någon koppling på ROP.
$\beta_{clutch.regl}$	Reglerad koppling. Kopplar ur vid växling.
δ	Rattläge. $\delta = 0$ betyder att ratten ej är vriden, $\delta = -1$ och $\delta = 1$ betyder att ratten är vriden maximalt åt vänster respektive höger.
λ	Kontinuerligt lambda.
μ	Markens friktion vid varje hjul.
ϕ, θ, ψ	Fordonets orientering i radianer
ω	Fordonets vinkelhastighet i rad/s.
ω_{fd}	Slutväxelns vinkelhastighet i rad/s.
ω_k	Kardanaxelns vinkelhastighet i rad/s.
ω_t	Växellådans vinkelhastighet i rad/s.
ω_w	Hjulens vinkelhastighet i rad/s.

Referenser

[1] *Tire and vehicle dynamics* Hans B. Pacejka. Society of Automotive Engineers, Inc, 2002.

[2] *Course material Vehicular Systems* Lars Nielsen och Lars Eriksson. Kompendium, LiTH, 2004.

[3] *LIPS – nivå 1. Version 1.0.* Tomas Svensson och Christian Krysander. Kompendium, LiTH, 2002.

[4] *Nightrider - Systemskiss. Version 1.0.* Anders Toverland. Linköping, 2005.

[5] *Nightrider - Förstudie. Version 1.0.* Anders Toverland. Linköping, 2005.