

UNIVERSITEIT ANTWERPEN  
Academiejaar 2017-2018

Faculteit Toegepaste Ingenieurswetenschappen

## **Labo Automotive-II**

**MA-automotive**

### **Student**

Kris Van de Voorde

**Master of Science in de industriële wetenschappen**

## INHOUD

1.	Inleiding.....	4
2.	Low-level-programming.....	5
2.1	LED .....	5
2.1.1	Introductie .....	5
2.1.2	Implementatie.....	5
2.1.3	Testen.....	5
2.2	Button .....	6
2.2.1	Introductie .....	6
2.2.2	Implementatie.....	7
2.2.3	Testen.....	9
2.3	ADC.....	10
2.3.1	Introductie .....	10
2.3.2	Implementatie.....	11
2.3.3	Testen.....	12
2.4	PWM .....	13
2.4.1	Introductie .....	13
2.4.2	Implementatie.....	14
2.4.3	Testen.....	15
2.5	CAN .....	16
2.5.1	Introductie .....	16
2.5.2	Implementatie.....	16
2.5.3	Testen.....	22
3.	Model-Driven Engineering .....	25
3.1	Model.....	25
3.1.1	Plant .....	25
3.1.2	Controller .....	26
3.2	Requirements.....	27
3.3	Maintain speed .....	28
3.3.1	Implementatie.....	28
3.3.2	Resultaat .....	29
3.4	Set desired speed.....	30
3.4.1	Implementatie.....	30
3.4.2	Resultaat .....	31
3.5	Maintain speed on slope.....	32
3.5.1	Implementatie.....	32
3.6	Resultaat .....	36
4.	States.....	37
4.1	Requirements.....	37
4.1.1	Requirements toevoegen via Excel.....	37

4.1.2	Resuirements manueel toevoegen .....	38
4.2	Implementatie.....	38
4.3	Testing.....	41
4.4	Equivalence testen.....	43
4.4.1	Discreet model .....	44
4.4.2	Discrete fixed-point.....	45
5.	Simulink model naar C code.....	47
6.	Integreren in een real-time OS .....	49
6.1	OSEK .....	49
6.1.1	Instellen van GOIL .....	49
6.1.2	Gebruik van GOIL .....	49
6.2	Samenvoegen van de verschillende functionaliteiten.....	51
6.2.1	ECU voor de cruise control.....	53
6.2.2	ECU voor de wielen .....	53
6.2.3	ECU voor het dashboard .....	53
6.2.4	Resultaat van de verschillende ECU's samen.....	54

## 1. Inleiding

In dit portfolio wordt beschreven hoe een cruise control gemaakt kan worden. Er wordt gebruik gemaakt van twee verschillende methodes om het einderesultaat te bekomen. Als eerst zijn we begonnen met het maken van een model-based project met behulp van Simulink. In dit onderdeel wordt de model, plant en controller gemaakt, daaromtrent worden er ook nog test suites geschreven.

De tweede methode die we gebruiken is Low-Level coding. Hier schrijven we in C enkele drivers gespecificeerd voor AT90CAN128. Met behulp van de IDE ATMEL studio. Hier wordt dan uiteindelijk alle gecombineerd met een REAL-TIME OS genaamd OSEK.

## 2. Low-level-programming

Dit portfolio begint bij het beschrijven van het Low-Level Programming. Omdat dit logischerwijze de basis zal vormen voor verder uitbreidingen die gemaakt zullen worden. We programmeren vijf verschillende drivers. De code zal beschikbaar zijn in de documentatie. In de subsecties zal vooral beschreven worden hoe de drivers getest worden. Of dat ze werken en wat ze juist moeten doen.

Bij elke driver wordt ook een interface voorzien die in team is afgesproken.

### 2.1 LED

#### 2.1.1 Introductie

De eerste driver die geschreven werd, was de LED driver. Deze driver zal ervoor dienen om de juiste LEDs te tonen. Op basis van een char, die als parameter naar de driver gestuurd wordt, gaan de juiste lichtjes van het bord aan.

#### 2.1.2 Implementatie

Om de LEDs van de DVK90CAN aan te sturen moeten we gebruik maken van de registers die zich op PORTA bevinden. We zullen deze ook als output moeten definiëren.

```
DDRA = 0b11111111;
```

#### 2.1.3 Testen

Dit werd dan getest door verschillende combinaties te proberen. Al kijkend naar de lichtjes kan men dan zien of de driver werkend is. Omdat we voor deze driver nog geen gebruik kunnen maken van de buttons, gaan we deze testen in de main functie. Gebruikmakend van de volgende functies kunnen we de lichtjes aansturen.

```
void ledsInit(void);  
void ledsOn(char bitmask);  
void ledsOff(char bitmask);  
void ledsToggle(char bitmask);  
void ledsSet(char bitmask);
```

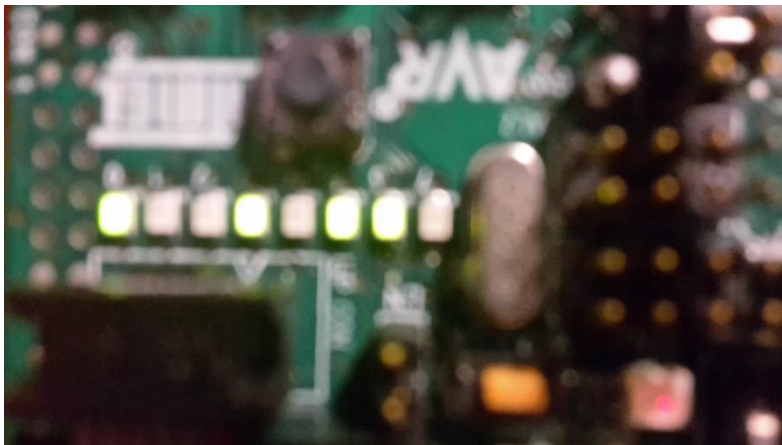
Om te testen zullen we gebruik maken van de volgende code in de main:

```

ledsInit();
ledsSet(0b00111010);
ledsOn(0b00000001);
ledsToggle(0b01000010);
ledsOff(0b00010000);

```

Wanneer we de code op het bordje zetten dan zullen de volgende ledjes zien branden :



We kunnen er vanuit gaan dat de LED driver werkt.

## 2.2 Button

### 2.2.1 Introductie

Ook voor de buttons zal er een driver geschreven worden. Deze buttons kunnen gebruikt worden om de Cruise Control te regelen. De buttons werken via een interrupt handler. Op basis hiervan wordt een deel van de code uitgevoerd als een rising edge interrupt wordt gedetecteerd.

Als we het document bekijken van DVK90CAN, kunnen we concluderen dat de buttons zich bevinden op PortE.



MA-Automotive  
Bert Van Acker

```
//set ports as inports
PORTE = PORTE | (1 << PORT7) | (1 << PORT6) | (1 << PORT5)
| (1 << PORT4) ;
DDRE = ~(~DDRE | (1 << PORT7) | (1 << PORT6)
| (1 << PORT5) | (1 << PORT4));

PORTD = PORTD | (1 << PORT1);
DDRD = ~(~DDRD | 1 << PORT1) ;

//activate interrupt 1 on rising edge
EICRA = EICRA | (1<< ISC11) | (1 << ISC10);
// activate interrupt 7,6,5,4 on rising edge
EICRB = EICRA | (1<< ISC71) | (1 << ISC70) | (1<< ISC61) | (1 << ISC60)
| (1<< ISC51) | (1 << ISC50) | (1<< ISC41) | (1 << ISC40);
//enable the interrupts
EIMSK = EIMSK | (1 << INT7) | (1 << INT6) | (1 << INT5) | (1 << INT5)
| (1 << INT4) | (1 << INT1);
```

Als eerst hebben we de buttons getest door geen gebruik te maken van interrupts, zodat er geweten is of de driver werkt. Dan zijn we overgeschakeld naar het gebruik van ISR. In dit geval moeten we INT\_VECT[7-4] aanspreken en INT\_VECT1. Belangrijk is dat in de ISR code de specifieke interrupt vlag afgezet wordt.

Voor de driver onafhankelijk te maken van andere drivers, gebruiken we in dit geval een function pointer zodat bij het testen de LED driver en Button driver niet afhankelijk zijn van elkaar en dat er makkelijk geswitcht kan worden van de driver die samen moet werken met de button driver.

```
buttonsInit(&ledsSetFunc);
```

Door een adres van een functie mee te geven, die gedefinieerd is in de main, kunnen we deze makkelijk in de main veranderen voor eventuele andere drivers.

```
void buttonsInit(void *lp) {
    funcX = lp;
```

Door in de init van de button de functie pointer gelijk te zetten aan de meegegeven functie pointer, kunnen we gebruik maken van een soortgelijke generieke functie in de ISR waardoor de drivers volledig onafhankelijk worden.

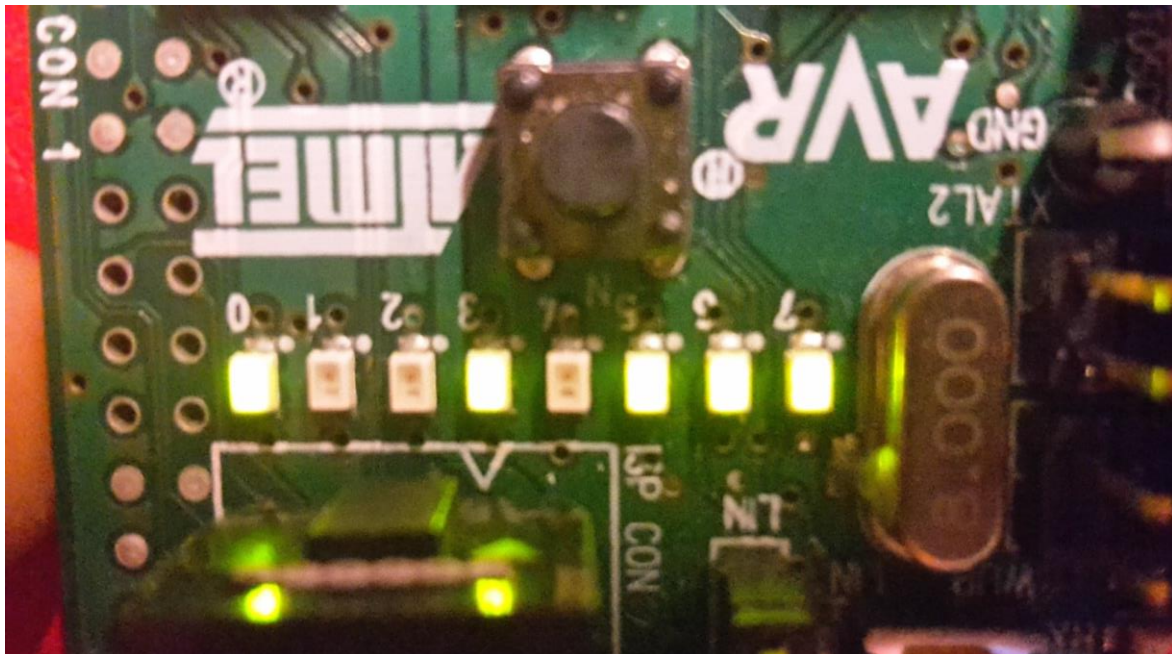
```
ISR(boutonISR7) {
    funcX(0b10000000);
```



### 2.2.3 Testen

Zoals bij het gebruik van de LEDS, kunnen we de buttons ook met het oog testen. We maken in dit geval gebruik van de reeds geteste LED driver. Door een button in te drukken zal de juist led combinatie moeten branden.

Het bordje start in deze toestand op.

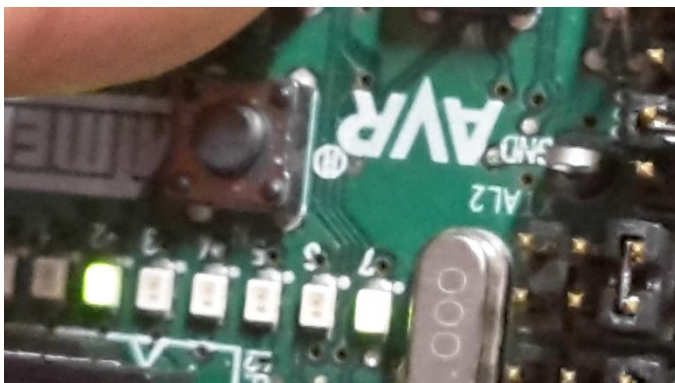


Als we bijvoorbeeld de de center knop indrukken:

De ISR bevat de volgende functie

```
funcX(0b00000100);
```

In de volgende figuur kunnen we zien dat inderdaad het derde lichtje gaat branden.

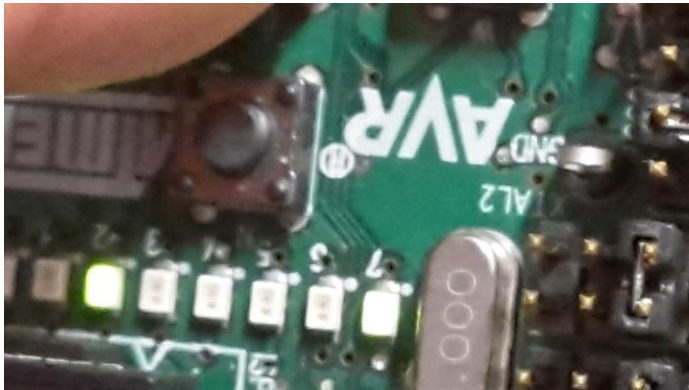


Om zeker te zijn gaan we ook nog eens de N-button indrukken:

De ISR bevat de volgende functie

```
funcX(0b00010000);
```

In de volgende figuur kunnen we zien dat inderdaad het vijfde lichtje gaat branden.



We kunnen concluderen dat de button driver werkt.

## 2.3 ADC

### 2.3.1 Introductie

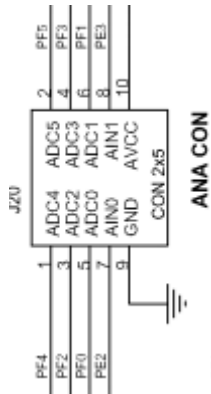
De volgende driver die we geschreven hebben is de ADC (Analog Digital Convert) driver. In deze driver gaan we ook gebruik maken van een function pointer zoals eerder uitgelegd is geweest in de button driver. Deze driver maakt ook gebruik van een ISR.

De volgende functie zijn momenteel nodig om de ADC werkend te krijgen.

```
void(*funcADC)(int c) ;
```

```
void adcInit(void *lp) ;
```

We gaan in het geval van de ADC driver gebruik maken van een potentiometer die verbonden is op de 3<sup>de</sup> ADC pin, die zich bevindt op PortF 3.



Wat de ADC intern zal doen, is gebruik maken van een analoog signaal die afkomstig is van de potentiometer die aan het bordje is gesoldeerd. De MCU (Microcontroller Unit) zal dit signaal quantiseren op een schaal van 0-255.

### 2.3.2 Implementatie

Als eerst gebruiken we de ADCSRA register om de ADC te activeren. De ADCSRA register zal ook gebruikt worden om de triggering van interrupts afkomstig van de potentiometers te activeren. De ADMUX register zal gebruikt worden om de analoge input van ADC3 te activeren. De register wordt ook gebruikt om een voltage referentie in te stellen.

REFS1	REFS0	Voltage Reference Selection
0	0	AREF, Internal Vref turned off
0	1	AV <sub>CC</sub> with external capacitor on AREF pin
1	0	Reserved
1	1	Internal 2.56V Voltage Reference with external capacitor on AREF pin

In het project wordt gebruik gemaakt van internal 2.56V Voltage Reference met een externe condensator op de AREF pin. De volgende code zorgt voor een werkende ADC init.

```
//enabeling ADC & enabeling trigger
ADCSRA = ADCSRA | (1 << ADEN) | ADCSRA /*| (1 << ADIFSC)*/ | (1 << ADIFR);

//Enabeling analog input for ADC 3 & set voltage reference
ADMUX = ADMUX | (1 << MUX1) | ADMUX | (1 << MUX0) | ADMUX
| (1 << REFS0) | ADMUX | (1 << REFS1);
//ADCSRA = ADCSRA | 1 << ADSC;

funcADC = 1p;
```

De volgende lijn code is nog veranderd, omdat beter lijkt te werken:

```
//Enabeling analog input for ADC 3 & set voltage reference
ADMUX = ADMUX | (1 << MUX1) | ADMUX | (1 << MUX0) | ADMUX
| (1 << REFS0) | ADMUX | (1 << REFS1);
```

Nu is het belangrijk dat we de ADC starten om een conversion uit te voeren. Hierdoor wordt de ADC gestart.

```
//Start single conversion
ADCSRA = ADCSRA | 1 << ADSC;
```

Bij deze is de init functie klaar, we kunnen nu overschakelen naar het verwerken van de interrupt. Voor de ISR moet er gekeken worden naar de ADC\_vect. Het is in dit geval weer belangrijk dat we de interrupt vlag afzetten door een één te schrijven naar ADIF. De waardes kunnen uit de ADCW register gelezen worden. We moeten in dit geval ervoor zorgen dat de twee least significant bits van de register verwijderd worden.

Eerst hebben we geprobeerd om de registers ADCL & ADCH te gebruiken. Deze registers gaven ongewenste waardes.

```
ISR(adcISR1)
{
    //turn interrupt flag off
    ADCSRA |= (1<<ADIF);
    //Getting ADC data 2 bitshift right
    funcADC(ADCW>>2);
    //end ISR
    CallTerminateISR2();
}
```

### 2.3.3 Testen

Omdat de ADC moeilijk te testen was door gebruik te maken van lichten, hebben we in dit geval geopteerd om gebruik te maken van het RS232 protocol. De code voor gebruik te maken van RS232 is door de docent beschikbaar gesteld.

De volgende functie is gedefinieerd in de main:

```
void ADCinterrupt(int c)
{
    printf("The integer is %d\r", c);
}
```

Door gebruik te maken van function pointers zal deze code opgeroepen worden als de ADC een interrupt ondervindt. We nemen de proef op de som en zullen de code testen aan de hand van de COM port. Hiervoor zal er gebruik gemaakt worden van RealTerm omdat Putty in ons geval niet werkt.

De potentiometer is ongeveer half opengedraaid op dit moment.

```
The integer is 159
```

De potentiometer is volledig opengedraaid op dit moment.

```
integer is 255
```

De potentiometer is volledig dicht gedraaid op dit moment.

```
the integer is 0  
the integer is 0
```

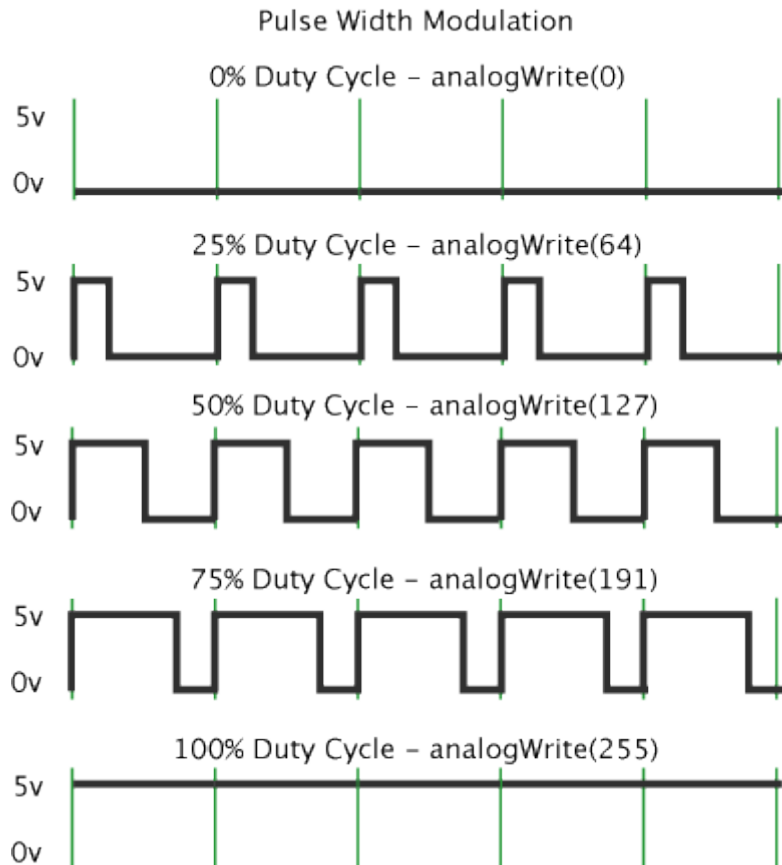
//momenteel niet testbaar, indien mogelijk later potentiometer werkt niet goed meer...

We kunnen concluderen dat deze driver ook werkt.

## 2.4 PWM

### 2.4.1 Introductie

De volgende driver die we geschreven hebben, is een Pulse Width Modulation (PWM) driver aan de hand van deze driver kunnen we een signaal met een duty cycle uitsturen. Zo'n signaal kan onder andere gebruikt worden om een elektromotor aan te sturen.



In ons project maken we gebruik van een fast PWM. In dat geval maken we gebruik van de counter. Die counter wordt vergeleken met de OCR0A register. Wanneer de counter over de waarde zit van de OCR0A zal er een één gestuurd worden. Wanneer de counter onder de OCR0A zal er een 0 gestuurd worden. Zoals te lezen valt in de datasheet, kan portB 7 gebruikt worden om de output te genereren.

### 2.4.2 Implementatie

De volgende code zorgt ervoor dat de PWM driver geïnitieerd kan worden.

```
//
//COM0A --> SET OC0A for comparing, WGM -> fast PWM, CS -> ci/0 = prescaler/256
TCCR0A |= (1 << FOC0A) | (1<<COM0A1) | (1 <<COM0A0) | (1<<WGM00) | (1<<WGM01) | (1 << CS02);

//external output for the OCR0A -> PWM
DDRB |= (1 << DDB7);
```

Met de volgende lijn code kunnen we ervoor zorgen dat de juiste duty cycle ingesteld wordt. Dit zal een getal zijn van 0-255.

```
//Output compare regist will compare with timer to generate PWM
OCR0A = pwmLength;
```

### 2.4.3 Testen

Met de volgende code kunnen we makkelijk de PWM testen.

```
pwmInit();  
  
#if DEBUG  
    printf("INIT PWM done \r");  
#endif  
pwmSet(0x50);
```

We zouden ook de pwmSet in de ADC fuction pointer kunnen steken zodat we dynamisch de duty cycle kunnen veranderen.

We kunnen op twee makkelijke manieren testen of dit werkt. Afhankelijk van welke duty cycle is ingesteld zullen we verschillende toonhoogtes horen. Een andere manier is gebruik te maken van een oscilloscoop. Door de oscilloscoop aan te sluiten op de juiste pin kunnen we het signaal zien dat wordt uitgestuurd.

Foto volgt nog!

## 2.5 CAN

### 2.5.1 Introductie

In deze sectie beschrijven we de CAN (Controller Area Network) die we in ons project gebruikt hebben. CAN is belangrijk wanneer we onze verschillende ECU's samen willen laten werken. CAN zorgt ervoor dat de communicatie tussen de verschillende IMU's mogelijk is. CAN werkt op basis van een CAN low en een CAN high signaal.

We gaan gebruik maken van PCAN-View software om onze messages te kunnen bekijken, of messages te versturen over een CAN bus. Vooraleer we begonnen zijn met de CAN driver hebben we eerst als groep samen besloten hoe we de messages zullen verdelen. In de volgende file wordt beschreven hoe we de verschillende messages per ECU opdelen.



Map1.xlsx

### 2.5.2 Implementatie

In deze sectie zullen we de implementatie van CAN beschrijven. We hebben onze CAN driver in acht verschillende functies onderverdeeld zodat de verschillende functies makkelijk opgeroepen kunnen worden wanneer deze nodig zijn. Een belangrijke note bij de CAN driver is dat men zeker de interrupt niet mag vergeten te includen.

```

void (*functionCANSpeed)(char value) ;
void (*functionCANButton)(char value);
void (*functionCANForce)(uint16_t force);
void (*functionCANSetSpeed)(char speed);
void (*functionCANCCON) (char CANOn);

void canInit(void* FunctionCANSpeed, void* FunctionCANButton, void* FunctionCANForce, void* FunctionCANSetSpeed, void* FunctionCANCCON);
void canSetBaud(void);
void canInitMob(void);
void canInitSend(void);
void canInitRecieve(void);
void canSend(char, uint16_t);
void canStop(void);
void canStart(void);
  
```



## CanInit

In deze functie gaan we de verschillende registers aanspreken die algemeen ingesteld moeten worden. Deze register bevinden zich boven de verschillende mobs.

De belangrijkste registers die we hier gebruiken, zijn de volgende:

`CANIE1 = 0x7F;`

`CANIE2 = 0xFF;`

Met deze registers zetten we de interrupts voor alle mobs aan.

## CanSetBaud

Deze functie spreekt voor zich. In deze functie zullen we de baudrate zetten. De gewenste baudrate hebben we op voorhand afgesproken dat deze 125kb/s moet zijn. De verschillende ECU's zullen uiteraard allemaal op dezelfde baudrate moeten staan. We houden rekening dat we te maken hebben met een MCU die een clock van 8MHz bevat. Wanneer we de gewenste baudrate willen instellen zullen we in de datasheet de verschillende CANBTx register kunnen aflezen in de volgende tabel:

**Table 19-2. Examples of CAN Baud Rate Settings for Commonly Frequencies (Continued)**

fclk <sub>MCU</sub> (MHz)	CAN Baudrate (Kbps)	Description			Segments				Registers		
		Sampling Point	TQ (μs)	Tbit (TQ)	Tprs (TQ)	Tph1 (TQ)	Tph2 (TQ)	Tsjw (TQ)	CANBT1	CANBT2	CANBT3
8.000	1000	63 % <sup>(1)</sup>		x	- - - no data - - -						
			0.125	8	3	2	2	1	0x00	0x04	0x12 <sup>(2)</sup>
	500	69 % <sup>(1)</sup>	0.125	16	7	4	4	1	0x00	0x0C	0x36 <sup>(2)</sup>
		75 %	0.250	8	3	2	2	1	0x02	0x04	0x13
	250	75 %	0.250	16	7	4	4	1	0x02	0x0C	0x37
			0.500	8	3	2	2	1	0x06	0x04	0x13
	200	75 %	0.250	20	8	6	5	1	0x02	0x0E	0x4B
			0.625	8	3	2	2	1	0x08	0x04	0x13
	125	75 %	0.500	16	7	4	4	1	0x06	0x0C	0x37
			1.000	8	3	2	2	1	0x0E	0x04	0x13
	100	75 %	0.625	16	7	4	4	1	0x08	0x0C	0x37
			1.250	8	3	2	2	1	0x12	0x04	0x13

In deze functie zullen we elke MOB zelf moeten initialiseren voor gebruik. Het is belangrijk dat elke MOB zeker geïnitieerd wordt als we de CAN werkend willen krijgen. Wanneer we door elke MOB willen gaan moeten we gebruik maken van de register CANPAGE en een loop.

```
for(i = 0; i < 15; i++) {
    CANPAGE = i << 4; //select mob
```

Als we de juiste mob willen selecteren, moeten we er zeker opletten dat het gewenste nummer 4 bits naar links wordt geshift vooraleer deze toegekend wordt tot de CANPAGE register.

Bit	7	6	5	4	3	2	1	0	
	MOBNB3	MOBNB2	MOBNB1	MOBNB0	AINC	INDX2	INDX1	INDX0	CANPAGE
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Wanneer de CANPAGE is ingesteld, zijn de volgende mob registers die veranderd worden, gelinkt aan het nummer dat we in de CANPAGE gestoken hebben.

Per mob zetten we de volgende registers op nul, omdat deze registers geen initiële waardes bevatten:

#### V2.0 part A

Bit	15/7	14/6	13/5	12/4	11/3	10/2	9/1	8/0	
	-	-	-	-	-	RTRTAG	-	RB0TAG	CANIDT4
	-	-	-	-	-	-	-	-	CANIDT3
	IDT2	IDT1	IDT0	-	-	-	-	-	CANIDT2
	IDT10	IDT9	IDT8	IDT7	IDT6	IDT5	IDT4	IDT3	CANIDT1
Bit	31/23	30/22	29/21	28/20	27/19	26/18	25/17	24/16	
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	-	-	-	-	-	-	-	-	

#### V2.0 part B

#### V2.0 part A

Bit	15/7	14/6	13/5	12/4	11/3	10/2	9/1	8/0	
	-	-	-	-	-	RTRMSK	-	IDEMSK	CANIDM4
	-	-	-	-	-	-	-	-	CANIDM3
	IDMSK2	IDMSK1	IDMSK0	-	-	-	-	-	CANIDM2
	IDMSK10	IDMSK9	IDMSK8	IDMSK7	IDMSK6	IDMSK5	IDMSK4	IDMSK3	CANIDM1
Bit	31/23	30/22	29/21	28/20	27/19	26/18	25/17	24/16	
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	-	-	-	-	-	-	-	-	

Ook wordt de message helemaal leeg gemaakt.

## CanInitSend

In deze functie zetten we de mobs klaar die verzonden moeten worden. We stellen de juiste mob in aan de hand van de CANPAGE. We gebruiken de CANSTMOB om alle flags op 0 te zetten. De CANIDT( 2 & 1) registers gaan we gebruiken om het juiste ID van de CAN message in te stellen.

De CANCDMOB gaan we gebruiken om de transmission te enabelen voor een bepaalde mob maar ook de lengte van de CAN message zal hier ingesteld worden.

## CanInitReceive

Deze functie is gelijkaardig aan de CanInitSend. In dit geval zetten we de mob klaar om een bericht te ontvangen. We selecteren als eerst de mob die we nodig achten om een bepaalde Canmessage op te halen. In dit gedeelte zullen we de CANIDTx registers gebruiken om een ID toe te kennen van het CAN berichtje dat verwacht wordt. We kunnen ook de CANIDMx registers gebruiken om de acceptance filter te gebruiken deze filter zal de message ID's die binnen de filter liggen accepteren.

**Full filtering:** to accept only ID = 0x317 in part A.

- ID MSK = 111 1111 1111<sub>b</sub>  
 - ID TAG = 011 0001 0111<sub>b</sub>

**Partial filtering:** to accept ID from 0x310 up to 0x317 in part A.

- ID MSK = 111 1111 1000<sub>b</sub>  
 - ID TAG = 011 0001 0xxx<sub>b</sub>

**No filtering:** to accept all ID from 0x000 up to 0x7FF in part A.

- ID MSK = 000 0000 0000<sub>b</sub>  
 - ID TAG = xxx xxxx xxxx<sub>b</sub>

In ons geval zullen we alle ID's accepteren.

```
CANIDM1 = 0xFF;           // accept only this ID
CANIDM2 = 0xFF;
CANIDM3 = 0x00;
CANIDM4 = (1 << RTRMSK);
```

Zoals hierboven te lezen valt, maken we ook nog gebruik van een RTRMSK bit. Door de RTRMSK op één te zetten, zullen we de acceptance filter enabelen.

Verder zullen we ook gebruik maken van CANCDMOB om de bepaalde mobs te enabelen als een receive. Zoals bij CanInitSend zullen we deze register ook gebruiken om de verwachte lengte van de CAN message in te stellen. Als de lengte van een bericht niet overeenkomt met de verwachte lengte zal er een warning interrupt plaatsvinden.

We gaan in dit geval ook weer alle flags clearen van de mob door CANSTMOB op nul in te stellen.

### CanSend

Nu moeten we ervoor zorgen dat een message effectief nog verstuurd wordt over de CAN bus. Dit zullen we doen in deze functie. Eerst moeten we er wel nog voor zorgen dat het register CANMSG gevuld wordt met het juiste bericht. Uiteindelijk kunnen we de geselecteerd mob versturen doormiddel van de volgende lijn code:

```
CANCDMOB |= (1 << CONMOB0);
```

Door de CONMOB0 bit op één te zetten, zeggen we dat deze mob dient om te verzenden, waardoor deze mob ineens ook effectief verzonden zal worden.

### CanStop

In deze functie beschrijven we de logica hoe we de CAN moeten stoppen. We mogen enkel de CAN stoppen wanneer er niet meer verstuurd of ontvangen wordt. Eerst en vooral moet er gekeken worden of er geen transmissie meer is. Dit zullen we doen aan de hand van de CANGIT register. Deze register bevat een CANIT bit, die aantoont of de CAN nog met een bepaalde interrupt zit. We moeten in dit geval gebruik maken van een while, wat er kan voor zorgen dat volledig het programma blokkeert. We zullen daarom de canstop functie in een Task steken die we gedefinieerd hebben in de OIL file. We hebben de prioriteit niet hoog gezet omdat we belangrijkere functies niet willen storen met deze while:

```
TASK stopCANTask {
    PRIORITY = 20;
    AUTOSTART = FALSE;
    ACTIVATION = 1;
    SCHEDULE = FULL;
    STACKSIZE = 256;
};
```

Verder gebruiken we de CANGIE register om de verschillende interrupts te disabelen. De CANGCON kunnen we dan gebruiken om de CAN controller te disabelen.

```
while(CANGIT & (1 << CANIT)){
    CANGIE = ~(~CANGIE | (1 << ENIT) | (1 << ENRX) | (1 << ENTX)); // all
    CANGCON = ~(~CANGCON | (1 << ENASTB)); // disable CAN controller*/
```

## CanStart

Voor de CanStart functie moeten we het omgekeerde van de CanStop toepassen.

```
void canStart(void){
    CANGIE = CANGIE | (1 << ENIT) | (1 << ENRX) | (1 << ENTX); // all interrupts enabled
    // activate CAN controller
    CANGCON |= (1 << ENASTB);
}
```

## CANIT\_VECT

Een zeer belangrijke functie voor het ontvangen van een CAN bericht is de ISR die de CANIT\_vect behandelt. Dit is het stuk code dat uitgevoerd wanneer er een bericht verzonden of ontvangen wordt. In ons project zal dit enkel gebruikt worden om berichten te ontvangen. We moeten in dit geval weer gebruik maken van de CANPAGE om de bepaalde mob te selecteren. Het probleem dat zich kan vormen hierbij is dat een interrupt een ander stuk code kan onderbreken. Wanneer dit gebeurt zal de CANPAGE herschreven worden en zal de code dat onderbroken werd verder gaan met de verkeerde CANPAGE. Daarom gaan we gebruik maken van een temp variabelen waarin de vorige CANPAGE opgeslagen wordt.

We kunnen gebruik maken van de CANHPMOB register. In deze register wordt de CANPAGE van de mob die de hoogste prioriteit heeft en die een interrupt heeft bevonden opgeslagen. Wanneer we deze register zijn waarde in de CANPAGE steken kunnen we makkelijk de data afleiden van de bepaalde mob. We doen voor de zekerheid nog een controle op de RXOK flag van die bepaalde mob, zodat we weten dat het om een receive gaat en niet een transmit interrupt.

```
if (CANSTMOB & (1 << RXOK))
{
    //printf("voor de switch: %d \n",
    switch(CANPAGE >> 4) {
        case 1:
            data1 = CANMSG;
            functionCANSPEED(data1);
    }
```

Nadat we onze logica gedaan hebben voor de receive gaan we deze mob terug instellen voor te ontvangen.

```
CANCDMOB |= 1 << CONMOB1;
```

Als laatste zullen we de interrupt flags afzetten.

```
CANSTMOB = 0x00; // clear MOB1 RXOK Interrupt Flag
```

## Main

De volgende code kunnen we gebruiken wanneer we de CAN willen initialiseren:

```
canInit(&FunctionCANSpeed,&FunctionCANButton,&FunctionCANForce,&FunctionCANSetSpeed,&FunctionCANCCOn);
canSetBaud();
canInitMob();
canInitSend();
canInitRecieve();
canStart();
```

De volgende snippet kunnen we gebruik als we iets via CAN willen verzenden. De eerste parameter staat voor het ID en de tweede parameter gebruiken we voor een waarde mee te sturen. We moeten in ons geval geen lengte meesturen want elke MOB is vast gedefinieerd in de CAN driver.

```
canSend(2,c);
```

Voor het receive moeten we geen functie aanroepen vanuit de main, deze worden geregeld in de CAN driver zelf.

```
void canInitRecieve(void){
    uint32_t can_id;
    char mob;
    #if defined(CC)
    //wheelSpeed ID: 1 length: 1 ECU2 en ECU3
    can_id = 0x00000001; // which ID to accept
    mob = 0x10; // MOb0 Select
    canInitReceiveOne(can_id, mob, 1);
```

## 2.5.3 Testen

Zoals in de introductie vermeld is, maken we gebruik van een programma genaamd 'PCAN-View' via dit programma kunnen we messages lezen en versturen op de CAN bus. Wanneer we onze CAN hebben opgebouwd hebben we als eerst geprobeerd om één simpel bericht te versturen.

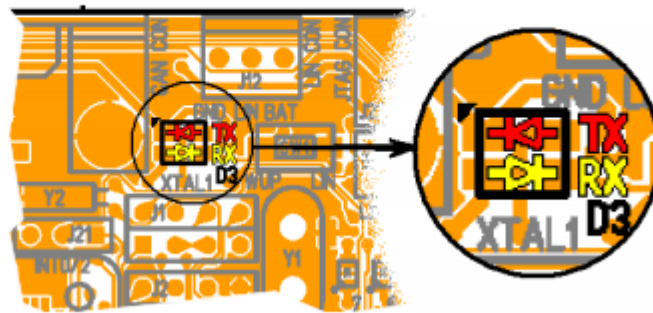
Daarna zijn we overgeschakeld om maar één bericht te receive. Wanneer beide werkte zijn we overgeschakeld om de beide te combineren met meer berichten.

Twee belangrijke indicaties waarop we kunnen baseren of er iets misgaat, of wanneer de CAN begint te versturen zijn:

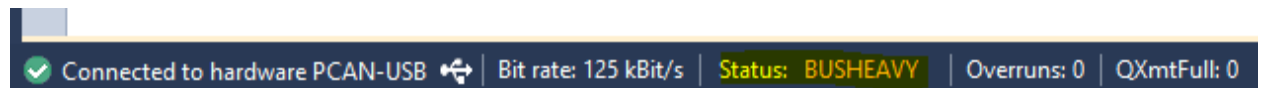
- De ledjes boven de CAN ingang

A red LED indicates a TxLIN traffic, a yellow one indicates a RxLIN traffic.

**Figure 3-24 . LIN LEDs**



- Of de CAN bus status die we kunnen lezen aan de onderkant van de PCAN-View



De CanStop & CanStart hebben we getest door eerst CanStart aan te roepen, daarna de CanStop aan te roepen en dan weer CanStart aan te roepen. Doordat dit geen errors gegeven heeft, zijn we ervan uit gegaan dat de beide functionaliteiten werkte.

De volgende afbeelding is een screenshot genomen met de PCAN:

CAN-ID	Type	Length	Data	Cycle Time	Count
002h		1	00	1709973,5	120
003h		2	00 00	327,6	712
004h		1	01	142347,8	2
001h		1	FF	327,7	800

CAN-ID	Type	Length	Data	Cycle Time	Count	Trigger	Comment
001h		1	01	✓ 1000	48392	Time	

We kunnen er vanuit gaan dat wanneer de code vergelijken met de desbetreffende verstuurde berichtjes dat de CAN driver werkt.



### 3. Model-Driven Engineering

In dit deel wordt het model opgebouwd. Het model zal bestaan uit een Cruise control in Simulink. Op basis van dit model kunnen er simulaties gemaakt worden met bijhorende testen. Uiteindelijk wanneer de requirements en testen goed zijn kan het discreet fixed-point model omgezet worden naar C code. De C code kan dan geïntegreerd worden met de drivers. Zodat de cruise control kan gebruikt worden op één van de gedistribueerde ECU's.

#### 3.1 Model

##### 3.1.1 Plant

In de plant wordt de fysieke set up van de Cruise control gemodelleerd. Hiervoor wordt er gebruik gemaakt van Newton's law. We gebruiken de volgende formule:

$$m \frac{dv}{dt} = u - bv$$

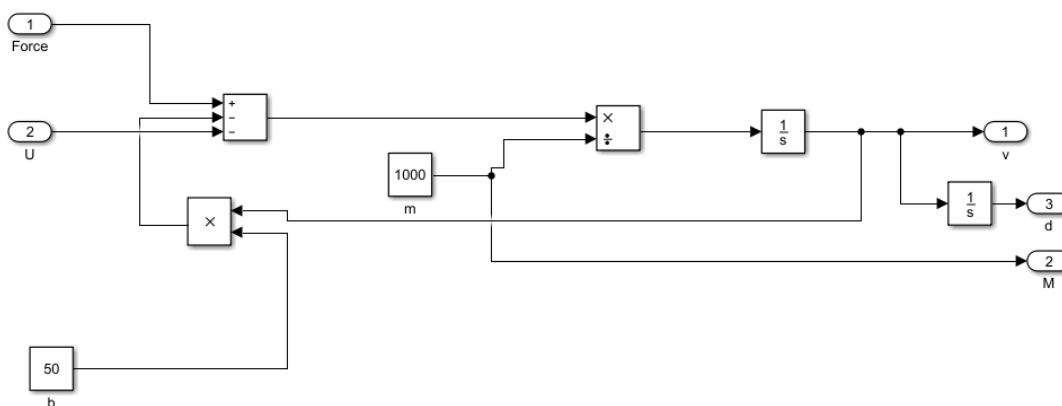
$$m = 1000kg = \text{massa}$$

$$b = \frac{50Nsec}{m}$$

$$u = 500N = \text{force from engine}$$

$$b * v = \text{friction}$$

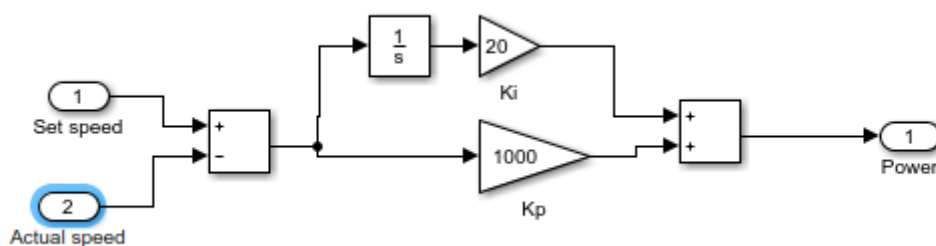
Op basis van deze formules krijgen we het volgende model. Dit model zal verder uitgebreid worden naar mate er meer requirements bekend worden.



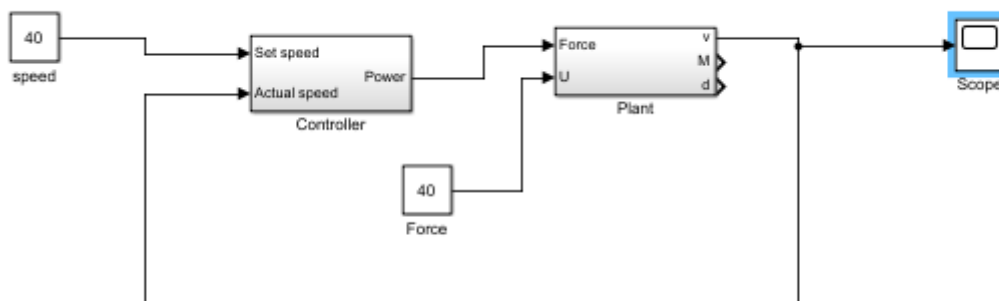
### 3.1.2 Controller

De controller zal ervoor dienen om de power van de engine te finetunen zodat er niet te veel na-ijling is. We maken gebruik van een PI-regelaar. PI staat voor Proportioneel en integrerend. Proportioneel wilt zeggen dat het verschil van de gemeten waarde en de gewenste waarde met een factor  $K_p$  versterkt wordt. Terwijl de integrerende term afhankelijk is van de fout en afhankelijk van hoelang er een fout is zal deze term meer ingrijpen. Hoe groter  $K_i$  is, hoe meer de integrerende term invloed gaat spelen.

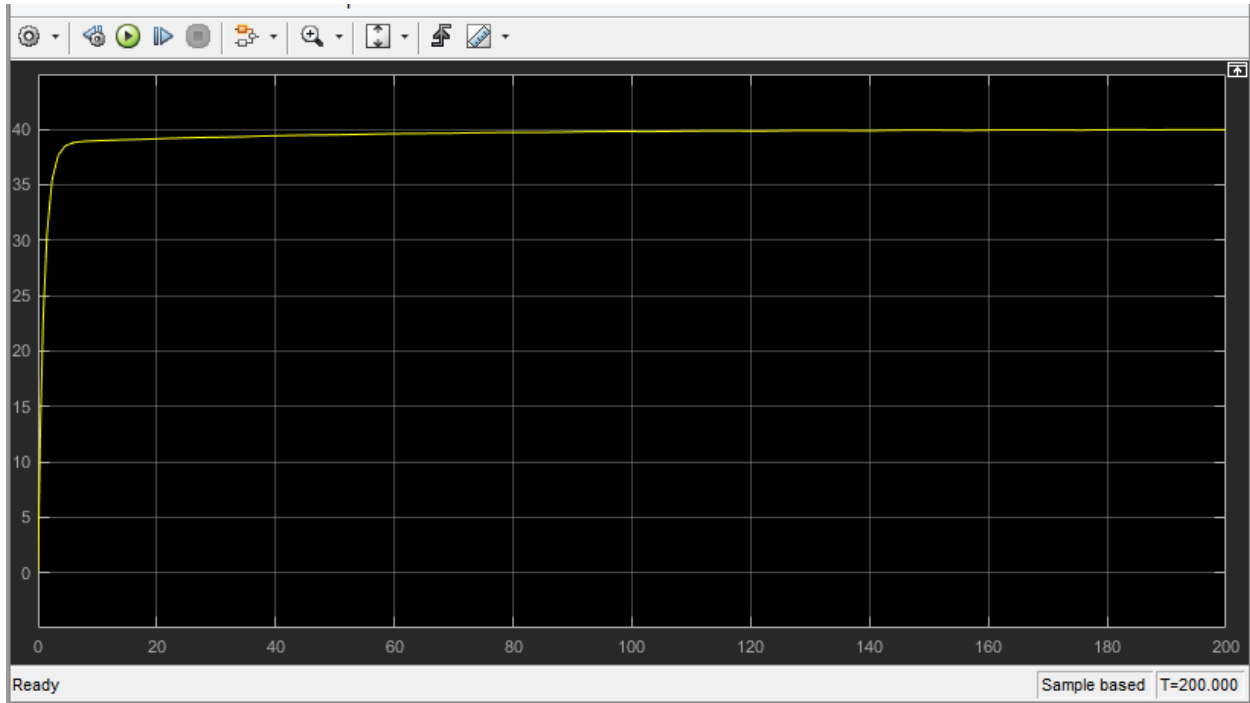
De volgende figuur toont onze PI regelaar die gebruikt wordt om het verschil tussen de gewenste snelheid en de actuele snelheid stabiel te krijgen.



Wanneer we beide modellen samenvoegen krijgen we de volgende figuur



Als we de scope bekijken krijgen we de volgende output, hieruit kunnen we concluderen dat met de gebruikte Kp & Ki er amper na-ijling is maar dat er veel tijd nodig is om een echt stabiel signaal te krijgen.

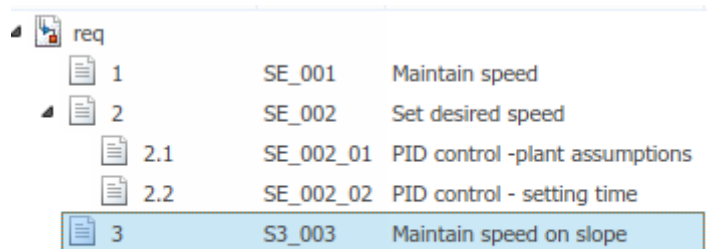


## 3.2 Requirements

Bij het maken van het model moet er rekening gehouden worden met enkele requirements die het gedrag van de cruise control bepaalt. In deze secties zal beschreven worden welke requirements er plaatsvinden en wat deze als taak hebben.

ID	Name	Description	Parent	SW system
SE_001	Maintain speed	The controller needs to be able to maintain a constant speed between 40 and 180 kmph		Controller
SE_002	Set desired speed	The driver needs the possibility to set the desired speed between 40 and 180 kmph		Controller
SE_002_01	PID control - plant assumptions	The PID controller needs to be adapted to maintain a constant speed for a car with the following assumptions: - mass = 1000 kg - friction constant = 50Nsec/m	SE_002	Controller
SE_002_02	PID control - settling time	The settling time of the controller needs to be less or equal to 10 seconds where the desired speed is within the absolute tolerance of $\pm 0.5$ kmph The targeted settling time under the assumption of controlling with a delta speed of maximum 20 kmph	SE_002	Controller

Eerst en vooral voegen we deze requirements in de requirements editor, met als gevolg krijgen we de volgende figuur. Deze requirements zullen getest worden in Simulink.

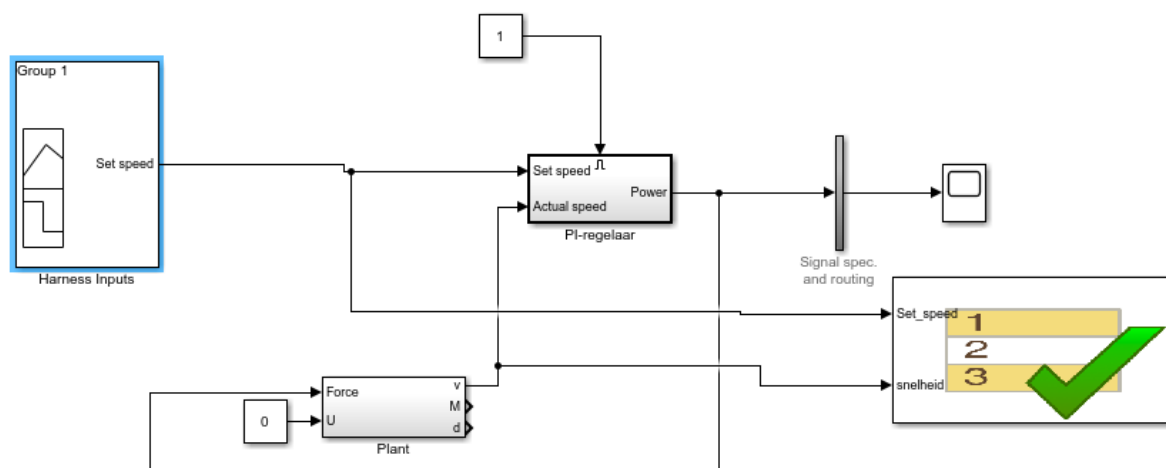


### 3.3 Maintain speed

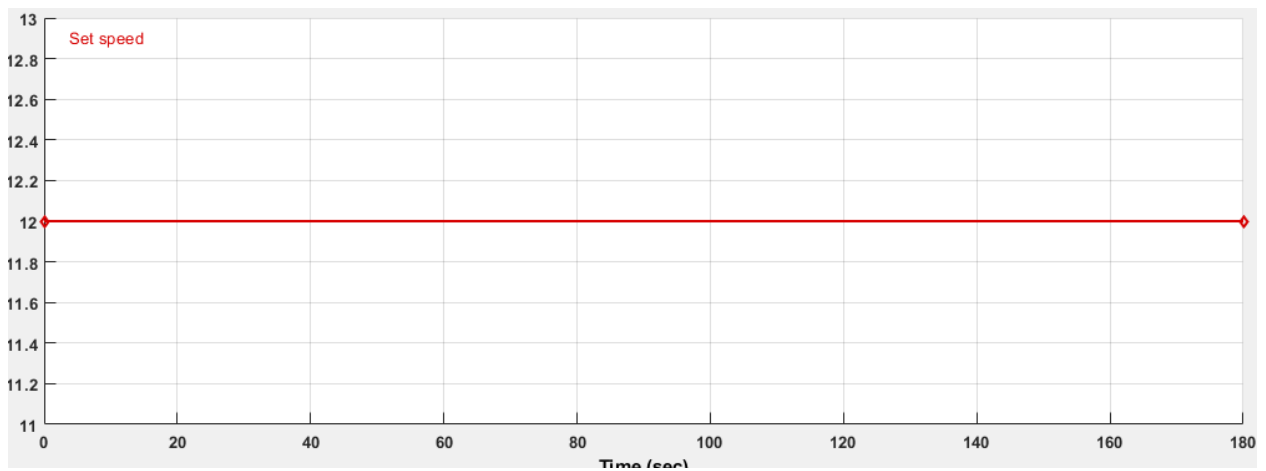
Maintain speed is de eerste requirement die geïmplementeerd zal worden. Deze requirement wilt aantonen dat de controller een constante speed moet aanhouden tussen 40 & 180 km/uur. We gaan deze requirement koppelen aan de PI-regelaar.

#### 3.3.1 Implementatie

We maken een test harness aan voor de maintain speed requirement. De test harness is gelinkt aan de PI-regelaar. We zorgen dat het een externe test harness is. Het test model ziet er als volgt uit.



We kunnen de signal builder gebruiken om de gewenste snelheid in m/s te simuleren.

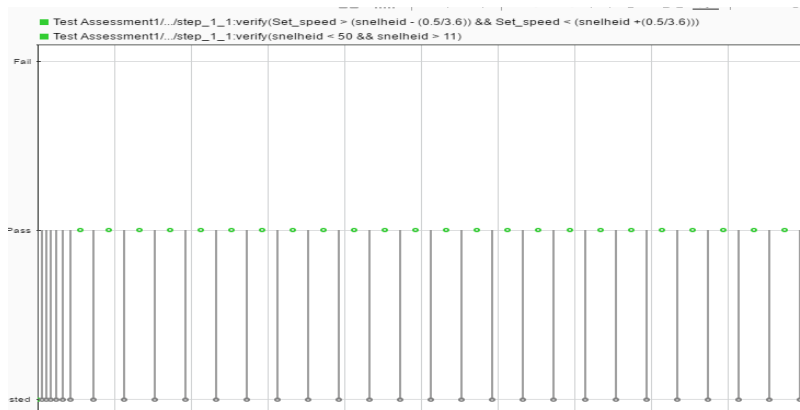


We gebruiken de volgende test code in onze test assessment blok.

Step	Transition	Next Step
<b>InitializeTest</b>  step_1_1 when t > 10 verify(Set_speed > (snelheid - (0.5/3.6)) && Set_speed < (snelheid +(0.5/3.6))); verify(snelheid < 50 && snelheid > 11)	1. true	WaitSt... ▼
<b>step_1_2</b>  WaitStabilization	1. after(1,sec)	Initializ... ▼

### 3.3.2 Resultaat

We kunnen zien dat de test geslaagd is in de volgende afbeelding.

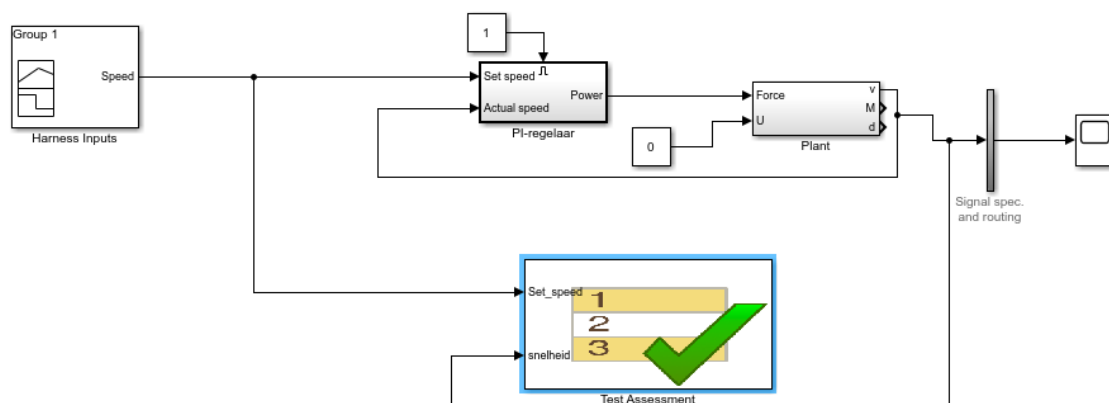


## 3.4 Set desired speed

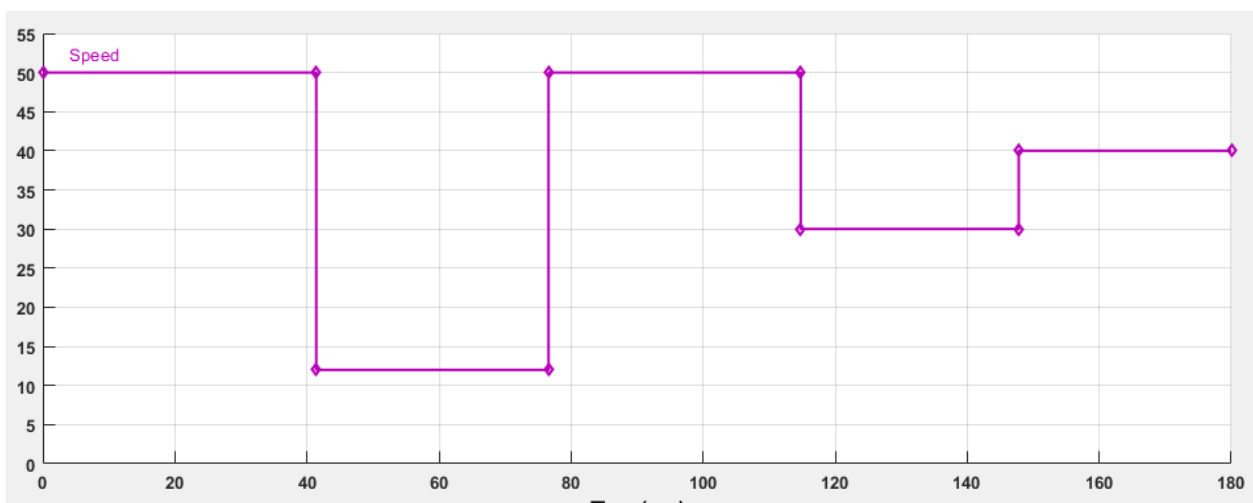
Bij deze requirement is het de bedoeling dat we een snelheid aan de PI-regelaar kunnen meegeven. Op basis daarvan moet de snelheid van de cruise control na 10 seconden binnen de grenzen van  $\pm 0.14$  m/s liggen.

### 3.4.1 Implementatie

We maken weer een externe test harness aan speciaal voor de PI-regelaar. We zorgen dat het volgende model tot stand komt.



Alweer gebruiken we een signal builder met het volgende signaal. De snelheid wordt een paar keer opnieuw ingesteld.

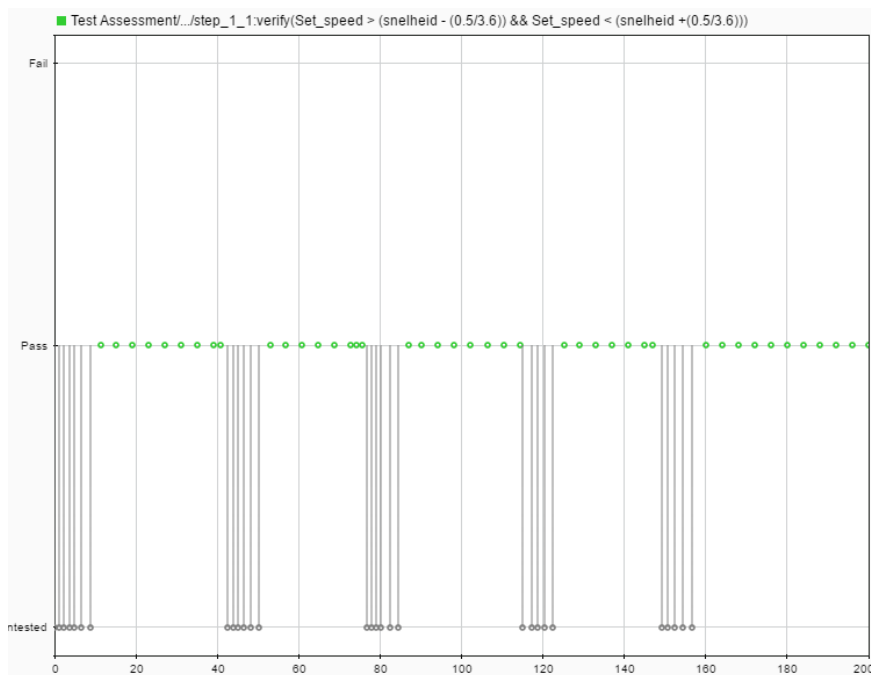


Voor te testen of de signalen correct zijn, maken we gebruik van de volgende code in onze test assement blok:

Test Sequence Editor				
Symbols	Step	Transition	Next Step	Description
<b>Input</b> 1. Set_speed 2. snelheid  <b>Output</b> speed speedtocheck  <b>Constant</b>  <b>Parameter</b>  <b>Data Store Memory</b>	<b>InitializeTest</b>  step_1_1 when t > 10 verify(Set_speed > (snelheid - (0.5/3.6)) && Set_speed < (snelheid +(0.5/3.6)));  step_1_2  WaitStabilization	1. hasChanged(Set          1. after(10,sec)	WaitSt...          Initializ...	

### 3.4.2 Resultaat

Met de volgende afbeelding kunnen we besluiten dat de testen geslaagd zijn.

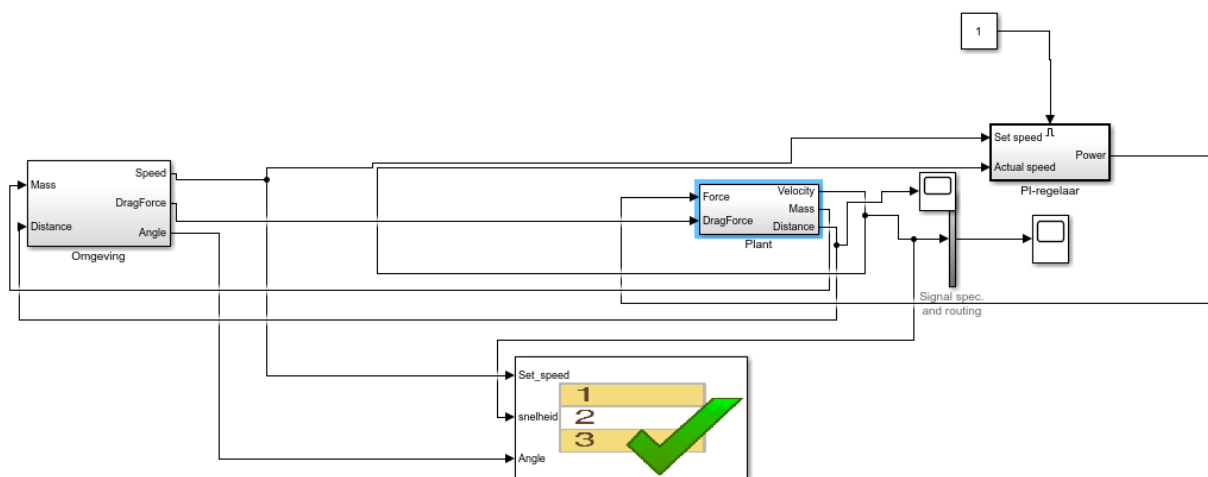


## 3.5 Maintain speed on slope

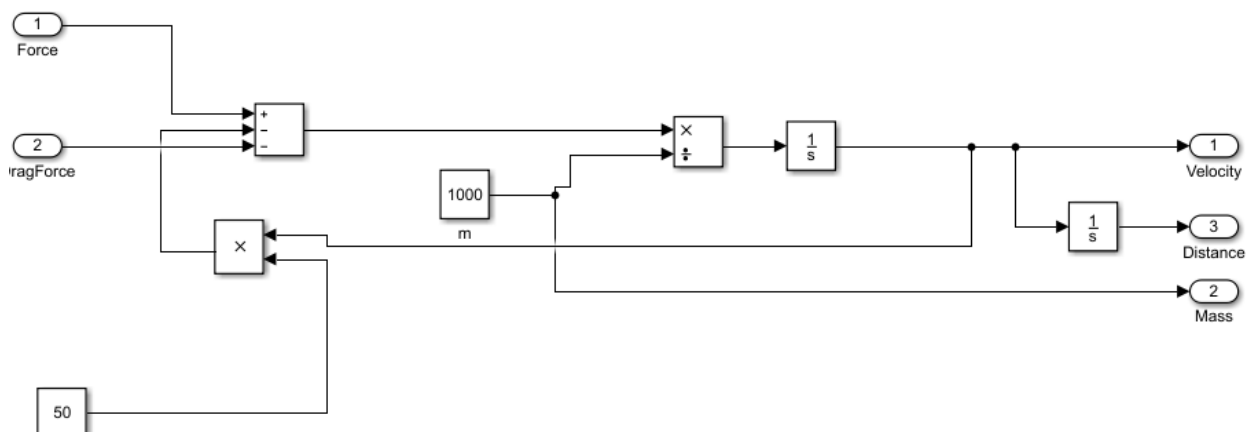
Een extra requirement voor onze cruise control is dat we rekening gaan houden met hellingen. De requirement houdt in dat we op een hoek van  $\pm 20$  graden een snelheidsverschil van  $\pm 5$  km/uur kunnen houden.

### 3.5.1 Implementatie

Weer maken we een externe test harness voor de PI-regelaar. In dit geval gaan we rekening houden met de drag force die als extra kracht inwerkt op de auto. We passen hiervoor ook de plant aan. De kracht die tegenwerkt op de auto moet afgetrokken worden van de power die de auto op dat moment geeft. Het model ziet er als volgt uit:



De vernieuwde plant ziet er als volgt uit. We moeten in dit geval ook de afstand en de massa gebruiken om inwerkende kracht te berekenen.



We hebben in dit geval ook een extra subblok gebruikt omdat de omgeving een extra parameter in rekening moeten brengen. De omgeving ziet er als volgt uit. In onze signal builder gebruiken we weer





een signaal dat zich situeert tussen 50 & 14 m/s. We maken gebruik van de volgende formule om de kracht te dragforce te berekenen.

$$F_p = m * ag * \sin(a)$$

$$ag = \text{acceleration of gravity} = \frac{9.81m}{s^2}$$

$$m = \text{mass of body}$$

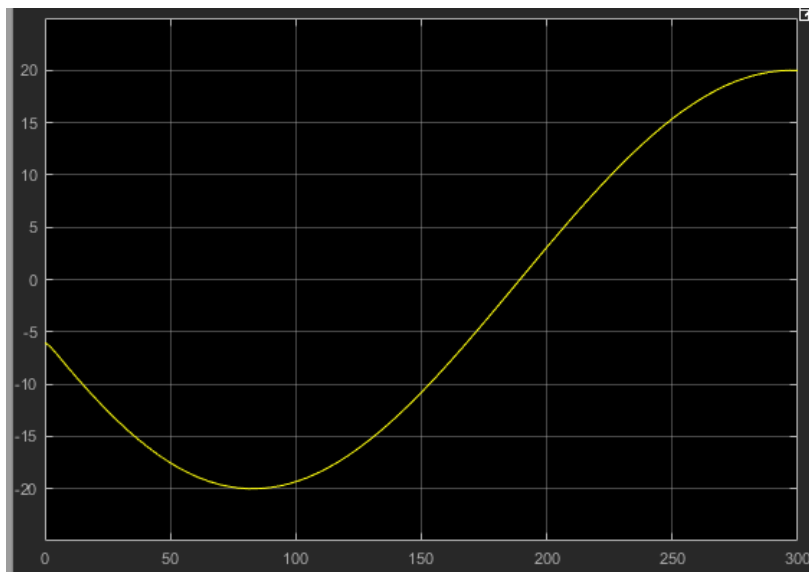
$$a = \text{elevation angle}$$

De formule wordt verwerkt in de omgeving om zo de dragforce te kunnen berekenen. Voor een berg te simuleren die -20 graden en + 20 graden bereikt, gebruiken we een 1-D T(u) blok. Op basis van de afstand krijgen we de juiste hoek. De afstand wordt berekend door een extra integraal aan de snelheid toe te voegen. Waardoor we een rechte verkrijgen.

Na wat trial and error krijgen we met de volgende configuratie een mooie berg.

Table and Breakpoints	Algorithm	Data Types
Number of table dimensions:	1	
Data specification:	Table and breakpoints	
Table data:	(20*sin([0:(60000)]/2000+60))	
Breakpoints specification:	Even spacing	
Breakpoints 1:	First point 1	Spacing 1
<a href="#">Edit table and breakpoints...</a>		

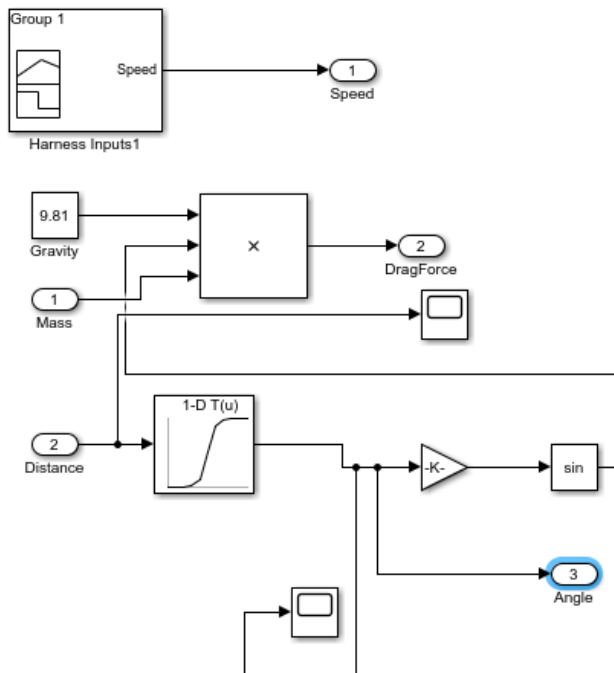
Met de scope zien we de volgende angle:



Wanneer we dit signaal willen verwerken in onze dragforce formule moeten we de sinus van de graden nemen. Omdat Simulink sinus enkel werkt in radialen, moeten we eerst nog een gain toevoegen die de graden omzet in radialen. De volgende formule wordt gebruikt:

$$rads = \frac{\pi}{180} * angle$$

Uiteindelijk bekomen we het volgende model:

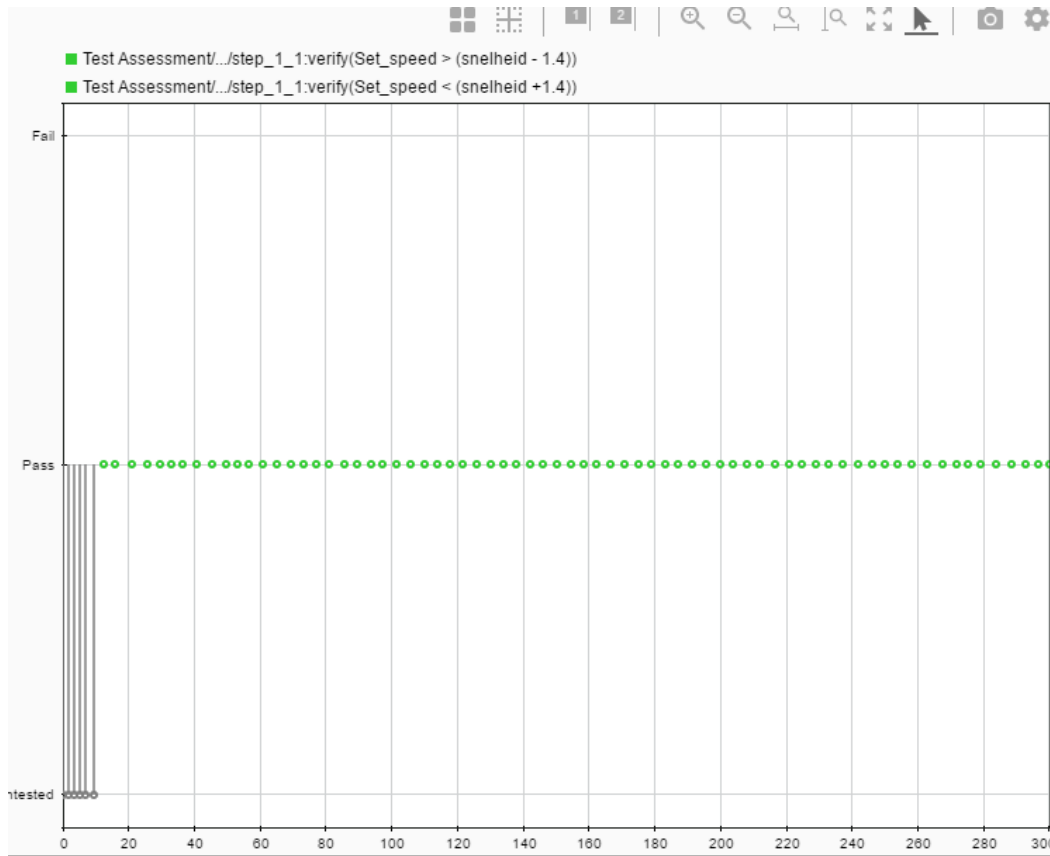


Wanneer we terug naar ons test harness gaan dan kunnen we op basis hiervan een test schrijven in de test assesment. Met de volgende code kunnen we testen of de snelheid stabiel genoeg blijft.

Step	Transition	Next Step	Description
InitializeTest	1. hasChanged(Set_speed)	WaitSt...	
step_1_1 when t > 10 && Angle > -20 && Angle < 20 verify(Set_speed > (snelheid - 1.4)); verify(Set_speed < (snelheid + 1.4));			
step_1_2			
WaitStabilize	1. after(10,sec)	Initializ...	

## 3.6 Resultaat

We kunnen er alweer van uitgaan dat het gedrag mooi stabiel blijft.



## 4. States

In deze sectie is het de bedoeling dat er een state chart gemaakt wordt. Met behulp van een state chart kunnen we makkelijk enkele states definiëren. Op basis van signalen kunnen we makkelijk van state veranderen. Elke state heeft zijn individuele eigenschappen. We kunnen onze state definiëren op basis van de use cases die in de opdracht vermeld zijn. De states en requirements van de use cases bepalen was een gezamenlijke opdracht.

### 4.1 Requirements

Na het lezen van de use case hebben we gezamenlijk in een groep de requirements opgesteld. Op basis van deze requirements hebben we de states bepaald.

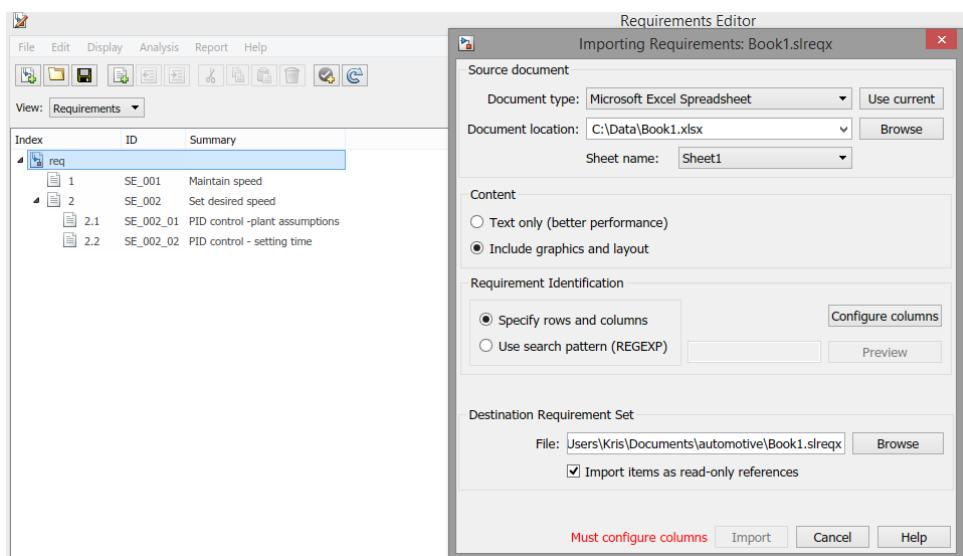
In het volgende Excel document staan de requirements opgesomd.



We kunnen de requirements op twee manieren invoegen.

#### 4.1.1 Requirements toevoegen via Excel

Wanneer we requirements willen toevoegen via Excel dan moeten we in dit geval gebruikt maken van import file in de requirement editor. Dan krijgen we de volgende afbeelding:



### 4.1.2 Resuirements manueel toevoegen

We kunnen ook de requirements manueel toevoegen zoals eerder gedaan.

Index	ID	Summary
req		
1	SE_001	Maintain speed
2	SE_002	Set desired speed
2.1	SE_002_01	PID control -plant assumptions
2.2	SE_002_02	PID control - setting time
3	S3_003	Maintain speed on slope
statechart_resuirements		
1	ST_01	Activate
1.1	ST_01_01	Set required speed
1.2	ST_01_02	Deactivate
1.3	ST_04_01	Reduce speed with break
1.3.1	ST_04_02	Resume CC after break
1.3.2	ST_04_03	Deactivate CC after break
1.4	ST_05_01	Suspend CC when transmission is out ...
1.4.1	ST_05_02	Resume CC when transmission is bac...
1.4.2	ST_05_03	Deactive CC after transmission when ...
1.5	ST_06	Maintain speed
2	ST_02	Show speed
3	ST_03_01	Accelerate with pedal
3.1	ST_03_02	Back to CC set speed

We linken al deze requirements aan de state chart.

## 4.2 Implementatie

In onze state chart maken we gebruik van drie verschillende states:

- StateOff -> dit is de default state
- StateSuspended
- StateOn

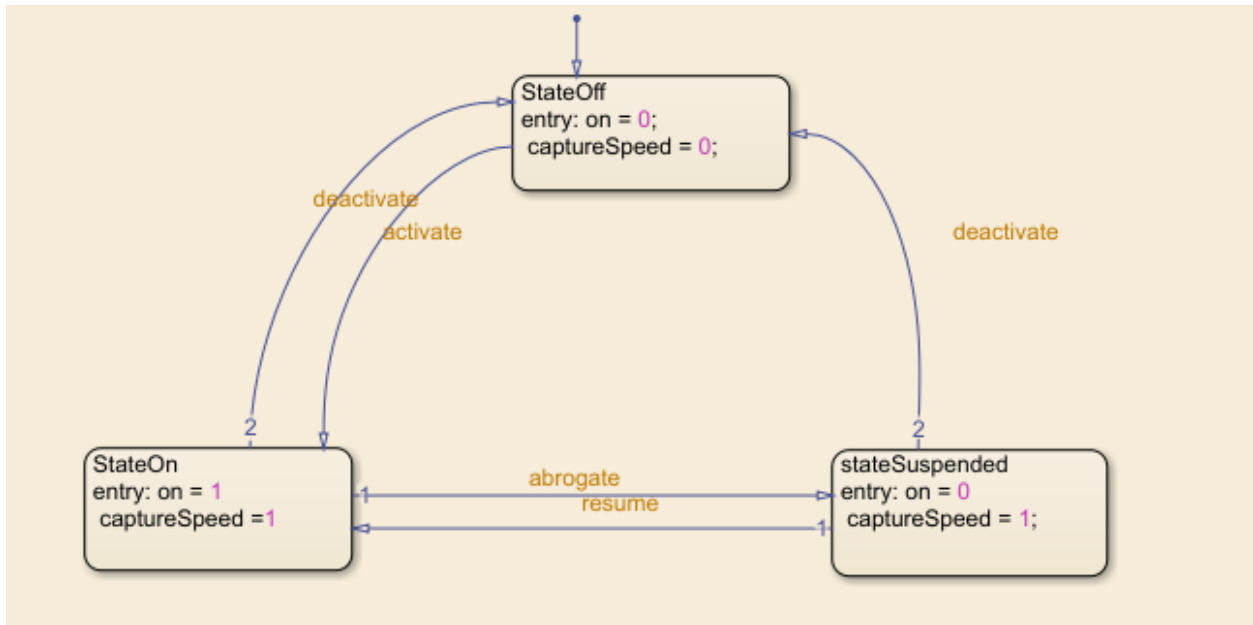
We maken ook gebruik van vier verschillende transactions om over te gaan naar een ander state:

- Deactivate
- Activate
- Abrograte
- Resume

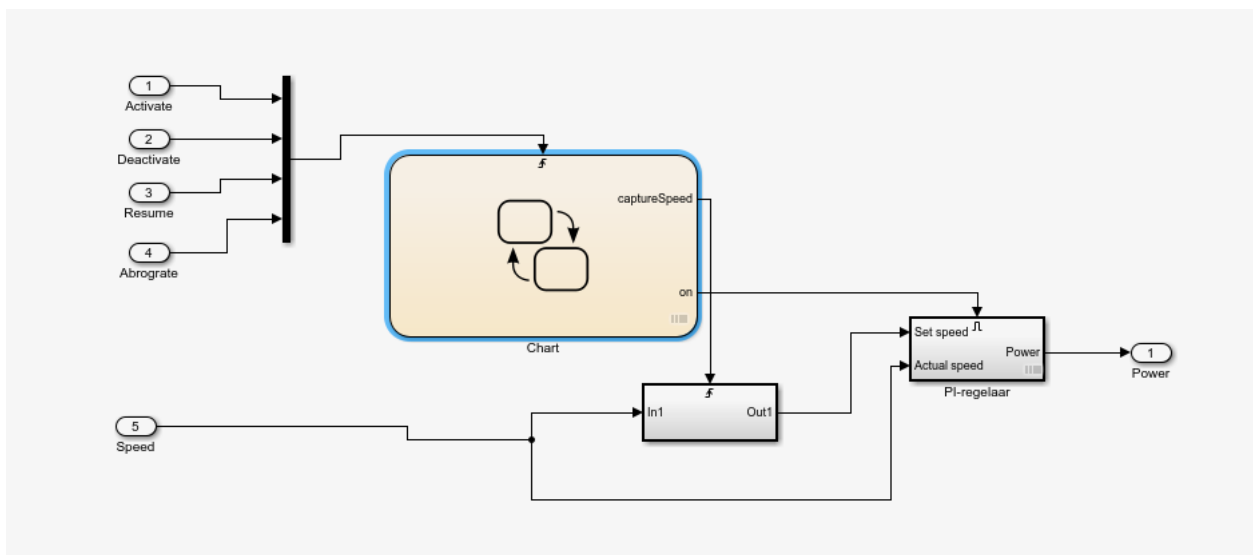
Er zijn twee variabelen die van belang zijn in onze state:

- On
- captureSpeed

Wanneer we deze elementen samenvoegen, krijgen we de volgende state chart:









Deze state chart wordt verwerkt in onze controller:



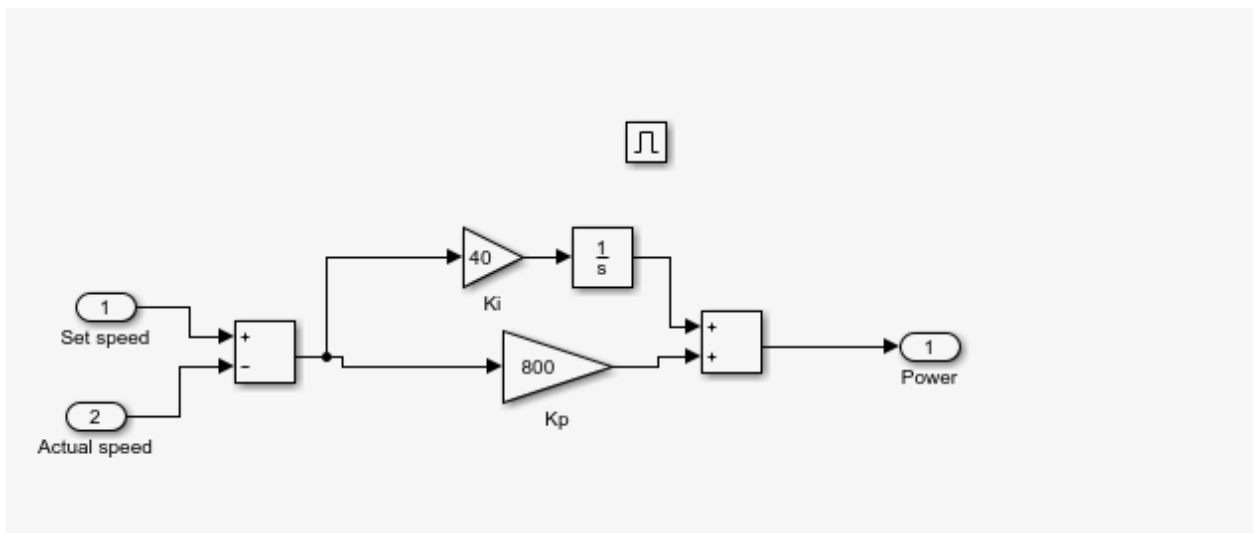
Als we het model bekijken dan is het belangrijk om te weten dat de verschillende mogelijk signalen in dezelfde volgorde als de transactions in de state chart moeten staan. De verschillende input signalen worden via een multiplexer in een vector gestoken.

Het volgende overzicht vinden we terug bij de model explorer.

Name	BlockType	Port	OutDataTypeStr	LockScale	OutMin	OutMax	PortDimensions	Unit	SampleTime	SignalType	IconDisplay
 captureSpeed		1						inherit			
 on		2						inherit			
 activate		1									
 deactivate		2									
 resume		3									
 abrogate		4									

Een andere belangrijke verandering is de triggered subsysteem. Belangrijk om te weten is dat het triggered subsysteem ervoor zorgt dat zijn laatste getriggerd value wordt bijgehouden wanneer dit systeem een pulse krijgt. Dit is een handig alternatief omdat dit eerst geprobeerd werd met een memory cell. Door de algebraïsche loop gaf dit fouten in Simulink. Met behulp van deze blok kunnen we de snelheid opslagen afhankelijk van welke state geven wordt. Hierdoor kan de juiste despised speed terug bekomen worden als de resume state gebruikt wordt.

Nog een verandering die we toebrengen in onze plant is het gebruik van een enable blok. Deze enable blok zorgt ervoor dat het systeem geactiveerd wordt bij een rising edge. De PI controller zal met andere woorden geactiveerd worden wanneer hij in de activate state is. Omdat deze state de variabel 'on' op één zet.

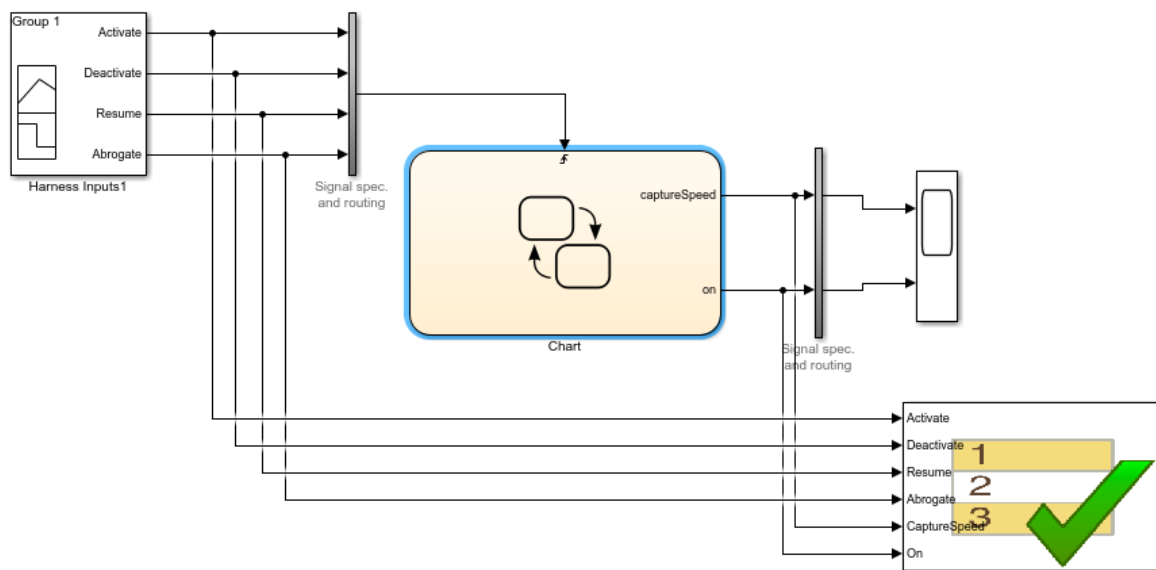




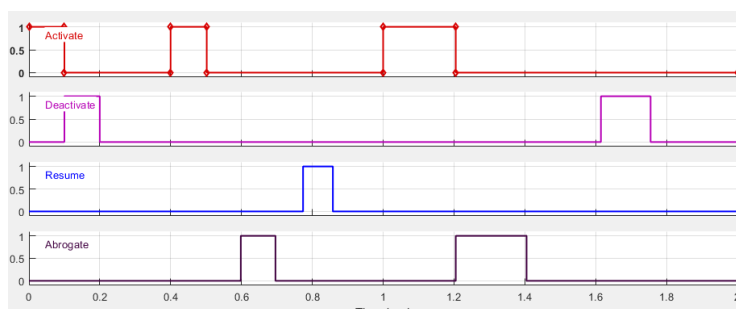
### 4.3 Testing

Net zoals we onze PI controller hebben getest, gaan we in dit geval ook de state chart testen. Op basis van de requirements gaan we na of de state chart het juiste gedrag vertoond. We maken weer gebruik van een test harness. Deze externe test harness gaat specifiek gericht zijn op de state chart.

We zorgen ervoor dat de volgende blokken gebruikt worden:



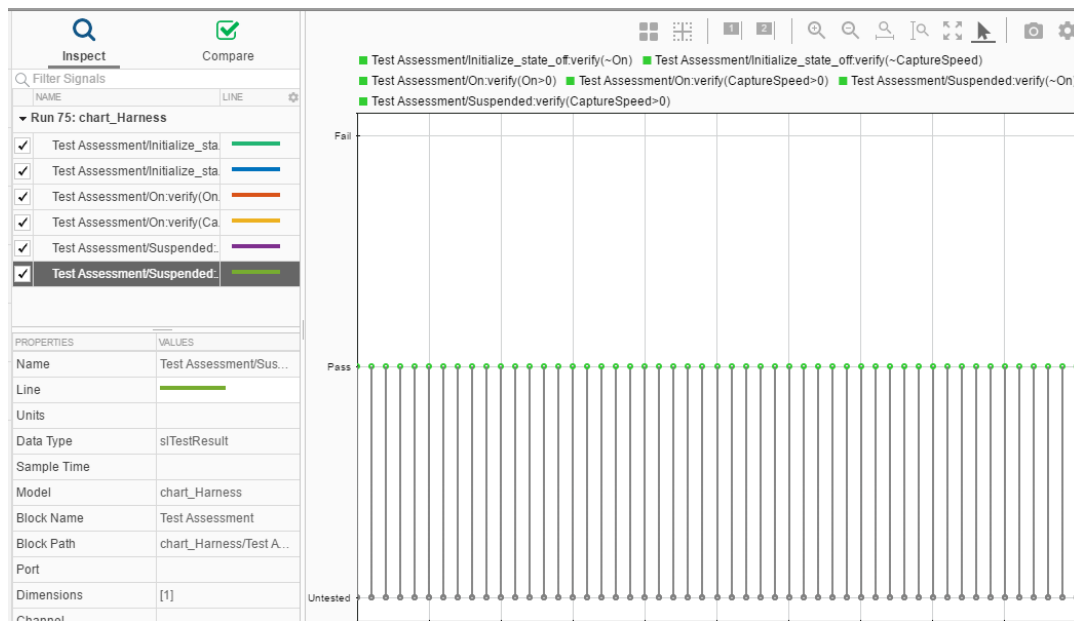
We gaan in dit geval pulsen moeten doorsturen in de signal builder



We zullen de volgende code gebruiken in onze test assessment:

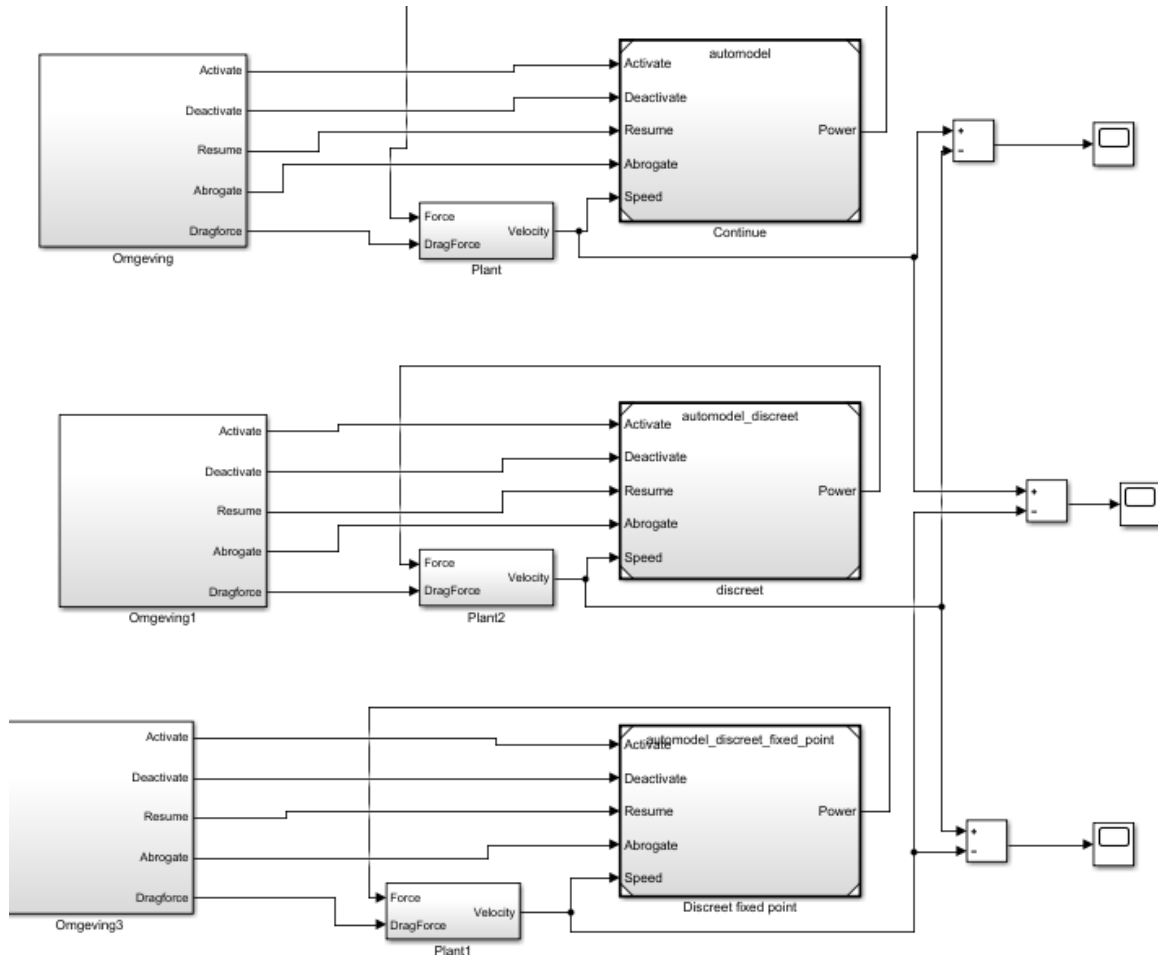
Symbols	Step	Transition	Next Step	Description
<b>Input</b> 1.  Activate 2.  Deactivate 3.  Resume 4.  Abrogate 5.  CaptureSpeed 6.  On  <b>Output</b>  <b>Local</b> <b>Constant</b> <b>Parameter</b>  <b>Data Store Memory</b>	<b>Initialize_state_off</b> verify(~On) verify(~CaptureSpeed)	1. <u>hasChangedTo(A</u>	On ▼	
	step_1_1 when t < 5			
	step_1_2			
	On verify(On>0) verify(CaptureSpeed>0)	1. <u>hasChangedTo(C</u> 2. <u>hasChangedTo(A</u>	Initializ... ▼ Suspe... ▼	
	Suspended verify(~On) verify(CaptureSpeed>0)	1. <u>hasChangedTo(C</u> 2. <u>hasChangedTo(F</u>	Initializ... ▼ On ▼	

We kunnen verifiëren dat ons state chart het juiste gedrag vertoond:



## 4.4 Equivalence testen

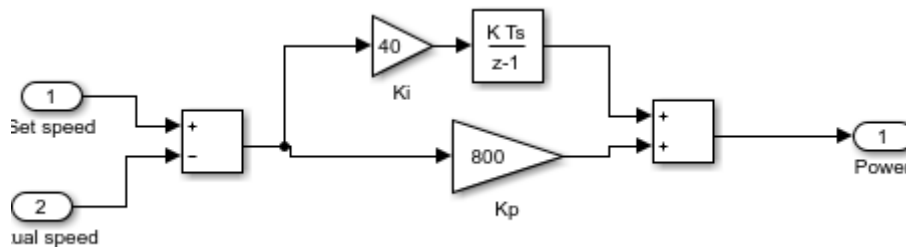
Als volgt voegen we de equivalence testen toe. Dit zijn testen die ons laten zien of het gedrag van een discreet, fixed-point discreet model niet te intens afwijken van het continu model.



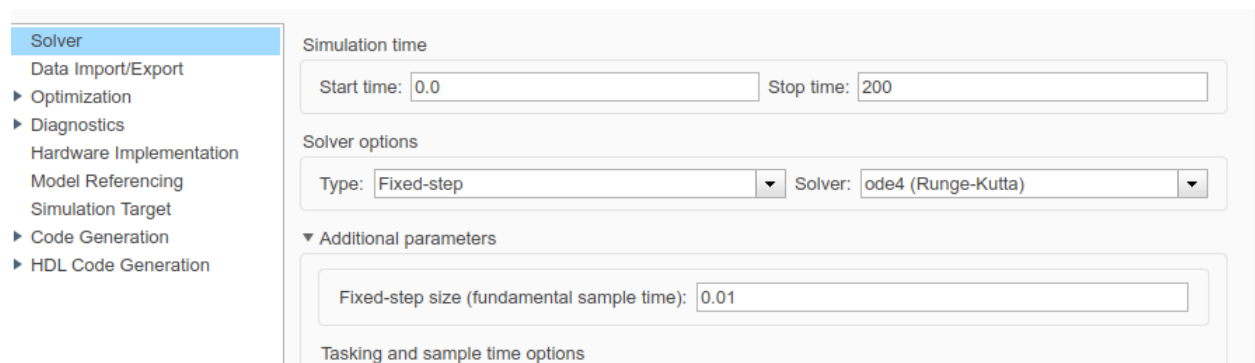
We maken gebruik van referentie blokken. Deze blokken staan gelinkt aan modellen. Het is de bedoeling dat we ons continue model twee keer kopiëren en deze verschillende modellen linken aan verschillende referentie blokken. Op deze manier kunnen we in elke referentie blok onafhankelijke aanpassingen aanbrengen. Op deze manier kunnen we ook de solvers manipuleren zodat we de verschillende blokken met elkaar kunnen vergelijken.

### 4.4.1 Discreet model

Om een model volledig discreet te maken, zullen we enkele aanpassingen aanbrengen. Ten eerste zullen we integrator blokken vervangen door tijds-discrete integrator blokken.



Ook de solver moet aangepast worden zodat er discrete waardes gegenereerd worden.



De step moet zeker klein genoeg zijn zodat er geen overbodige fluctuaties zijn, wanneer de step te groot is dan fluctueert het signaal naar oneindig groot. Hoe kleiner de step is hoe meer samples er gebruikt zullen worden waardoor het systeem accurater wordt, maar wel trager zal worden.

Wanneer we het signaal vergelijken met het continu model dan krijgen we het volgende verschil:



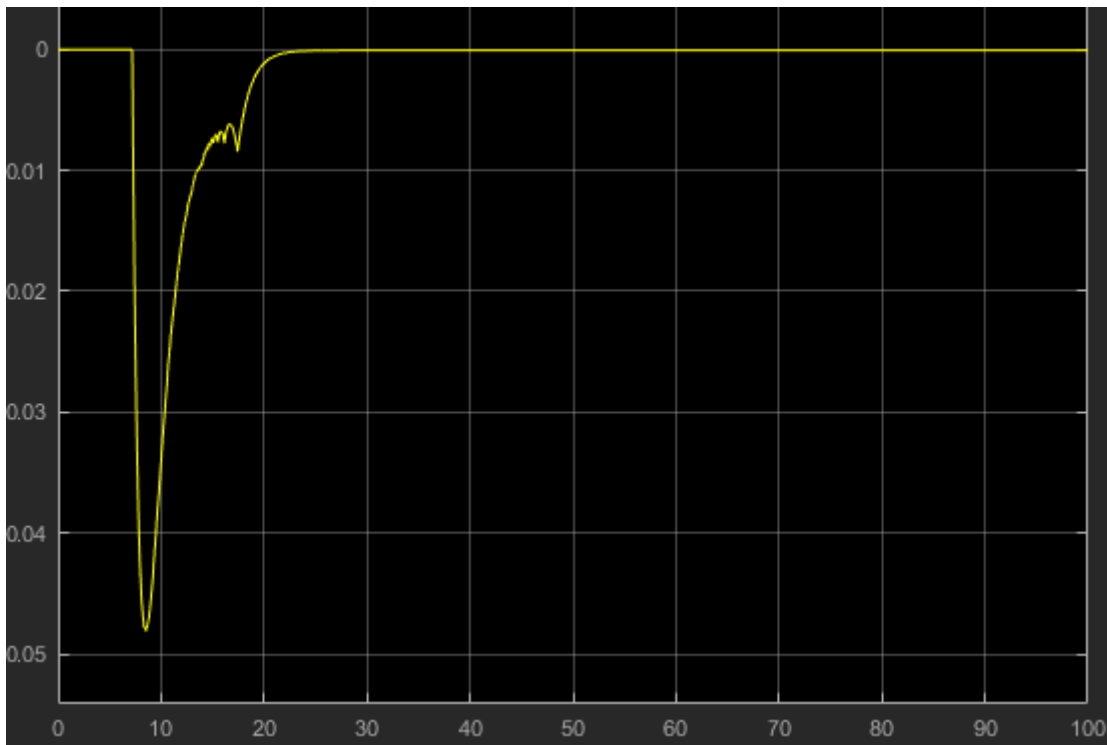
Dit verschil kan verwaarloosbaar worden.

#### 4.4.2 Discrete fixed-point

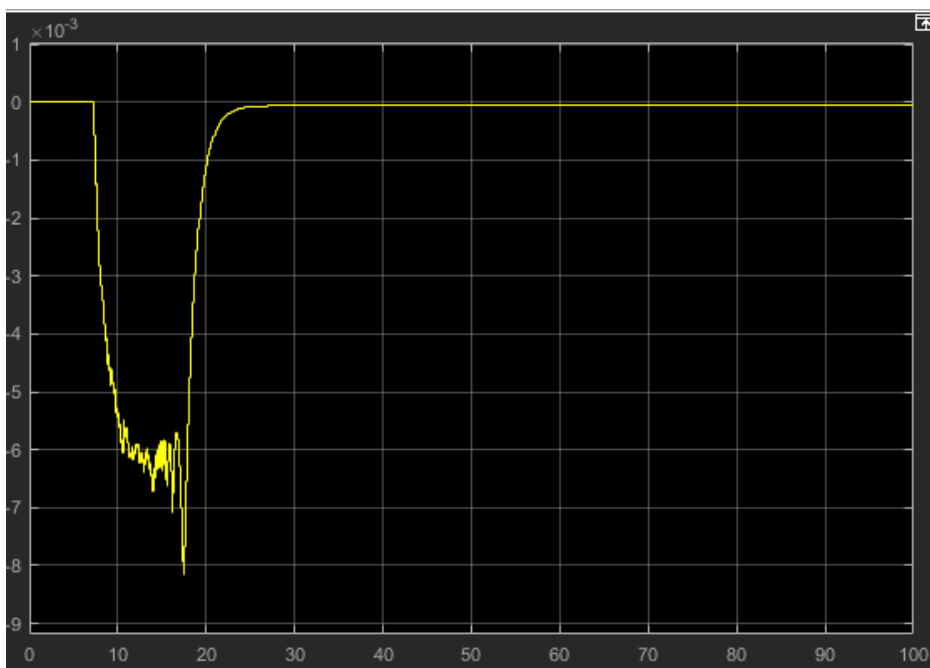
Dit model zal uiteindelijk gebruikt worden om C code te genereren. De bedoeling is dat dit model discreet is en werkt op basis van fixed points. We zorgen ervoor dat de plant overeenkomt met het discreet model zoals eerder vermeld. Nu gaan we ervoor moeten zorgen dat de signalen die gebruikt worden om te rekenen fixed-point zijn. Dit kunnen we doen door de data type van de in port 'Speed' in de controller gelijk te zetten aan fixed-point. We moeten er op letten dat de blokken die fouten geven 'inherit via backward propagatie' als data type hebben. Wat we zullen merken is dat wanneer er geen komma getal gebruikt wordt de fout groot zal zijn. Naargelang we meer komma getallen gebruiken zal de fout verkleinen. Na wat met het data type te spelen, krijgen we een redelijk nauw keurig signaal.

Data type:  >>

De volgende figuur is een vergelijking met het continue signaal. De fout is bijna hetzelfde als het discreet model.

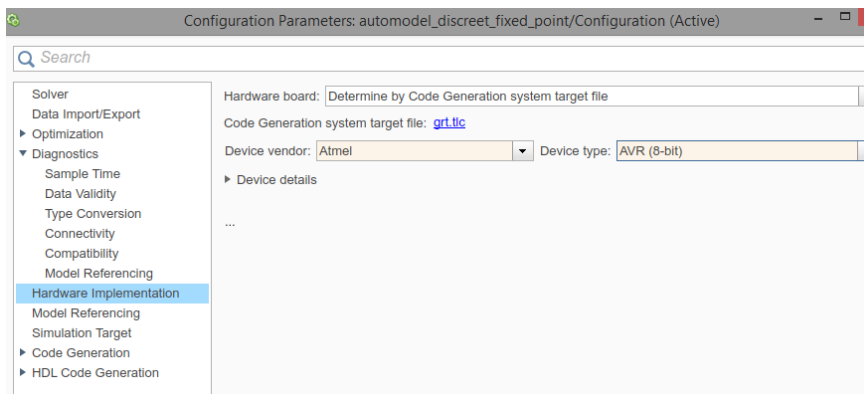


De volgende figuur is een vergelijking met het discreet signaal. De fout is zeer klein en bijna te verwaarlozen.

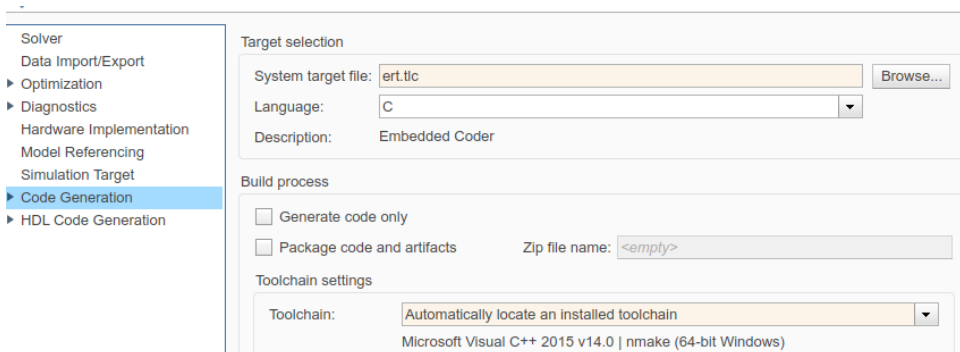


## 5. Simulink model naar C code

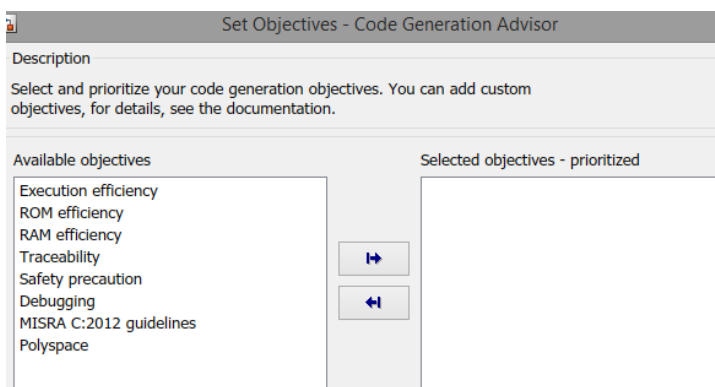
De bedoeling is nu dat we het tijds-discreet model omtoveren naar C-code zodat deze op een ECU kan draaien. Omdat we een 8 bit bordje hebben, is het de bedoeling dat we de volgende settings gebruiken:



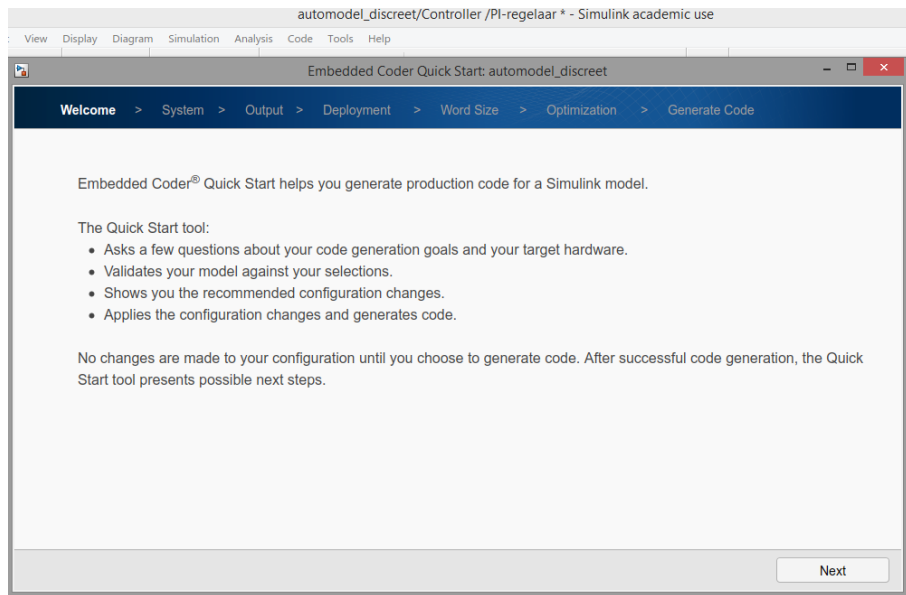
Voor de code generatie moeten we ervoor zorgen dat er ert.tlc files gegeneerd worden.



Bij set objectives is het ook nog mogelijk om bij het generen van code prioriteiten te zetten:



Daarna kunnen we start met het genereren van de code:





## 6. Integreren in een real-time OS

In deze sectie zal uitgelegd worden hoe we de Cruise control en de drivers kunnen integreren in een real-time OS. We maken gebruik van een OS genaamd OSEK (Open Systems and their Interfaces for the Electronics in Motor Vehicles). We zullen dit OS gebruiken voor de verschillende ECU's. De ECU's die wij zullen implementeren zijn bedoeld voor de wielen, het dashboard en de controller die de cruise control zal bevatten die we in het simulink model gemaakt hebben. Deze ECU zullen met elkaar communiceren over een CAN bus dat standaard vaak gebruik wordt in auto's.

### 6.1 OSEK

OSEK is dus gespecialiseerd in embedded Operating systems. Het wordt voornamelijk gebruikt in ECU's (Electronic Control Units) voor auto's. OSEK is dus een ideaal OS om te gebruiken in ons project. Voor OSEK te implementeren maken we gebruik van Trampoline. Trampoline implementeert OSEK. Trampoline maakt gebruik van OIL files die de verschillende tasks beschrijven met elk hun prioriteit die dan behandeld zullen worden door het scheduler van de real-time OS.

#### 6.1.1 Instellen van GOIL

Voor trampoline gebruiksklaar te maken voor te programmeren, moeten we GOIL eerst installeren. GOIL wordt gebruikt voor de OIL files te compileren. Een belangrijk detail is dat we ervoor zorgen dat wanneer GOIL geïnstalleerd is dat zijn PATH toegevoegd wordt aan het systeem path.

Door gebruik te maken van createProject.bat kunnen we een OSEK project aanmaken.

#### 6.1.2 Gebruik van GOIL

Nu zijn we klaar om te programmeren met OSEK. Wanneer we programmeren met OSEK moeten we rekening gaan houden dat we met taken gaan programmeren. Zodat code die het belangrijkste is de meeste prioriteit bevatten. In de OIL file, in ons geval 'firstAVR.OIL' kunnen we de verschillende tasks definiëren die gebruikt kunnen worden in de code.

We kunnen in ons OSEK gebruik maken van een periodische task of een taak die één keer uitgevoerd wordt.

Definitie van een task:

```
TASK startTask {
    PRIORITY = 5;
    AUTOSTART = TRUE {APPMODE = std;};
    ACTIVATION = 1;
    SCHEDULE = FULL;
    STACKSIZE = 256;
};
```

We zien dat we in een task de prioriteit kunnen definiëren.

Met de volgende code is het mogelijk om een task periodisch te maken.

```
ALARM periodicAl {
    COUNTER= SystemCounter;
    ACTION = ACTIVATETASK {
        TASK = periodicTask;
    };
    AUTOSTART = TRUE {
        ALARMTIME = 1;
        CYCLETIME = 1;
        APPMODE = std;
    };
};
```

Op de volgende manier kunnen we een task definiëren in de code.

```
TASK(secondTask)|
{
    //PORTA |=2;
    TerminateTask();
}
```

Het is ook mogelijk om via chainTask() een andere task te starten.

We moeten er ook voor zorgen dat onze ISR ingesteld worden in de OIL file, zodat deze ook behandeld zullen worden als taken.

```
ISR boutonISR1 {  
    CATEGORY = 2;  
    PRIORITY = 20;  
    STACKSIZE = 256;  
    SOURCE = INT1_vect;  
};
```

De source wordt dan veranderd door de INT\_vect die we normaal zouden gebruiken. We kunnen deze ISR dan aanroepen in onze code.

```
ISR(boutonISR1) {  
    funcX(0b00000100);  
    CallTerminateISR2();  
}
```

Een ISR moet ook altijd beëindigd worden. -> CallTerminatISR2().

Wanneer er wijzigingen aan gebracht zijn moeten we de 'runGoil.bat' uitvoeren zodat de verandering mee gecompileerd wordt.

## 6.2 Samenvoegen van de verschillende functionaliteiten

Nu kunnen we de verschillende functionaliteiten combineren, zodat we voor elke ECU een specifieke functionaliteit hebben, zoals eerder vermeld zullen deze ECU's samen de functionaliteit van de auto moeten vormen. Deze ECU's zullen communiceren via het CAN protocol. In deze sectie geven we een korte overview van de verschillende ECU functionaliteiten.

We kunnen de functionaliteit ook al voor een stuk afleiden uit de excel die bijgevoegd werd in de CAN-introductie sectie.

We hebben er voor gekozen om de functionaliteit van de verschillende IMU's in één project te beschrijven. Afhankelijk van een enum of een #definition in de code wordt de code voor een bepaalde ECU juist gezet.

In de 'defineHeader.h' kan de juiste ECU aangeduid worden. We hebben er voor geopteerd om gebruik te maken van een #define in plaats van ENUM zodat de precompiler ervoor zorgt dat er geen overbodige code wordt gecompileerd. Op die manier kunnen we het geheugen van de ECU besparen.

```
#ifndef DEBUG
#include "headers/rs232.h"
static FILE mystdout = FDEV_SETUP_STREAM(uart_putchar, NULL, _FDEV_SETUP_WRITE);
#endif // DEBUG

#define DASHBOARD
// #undef WHEEL
// #undef CC
```

De volgende snippet is een voorbeeld hoe we de code kunnen instellen aan de hand van een define. Afhankelijk van wat er in de defineHeader ingesteld is, kunnen we de juiste code genereren.

```
#if defined(CC)
//init CC
Subsystem0_initialize();
//init led
ledsInit();
// set a default.
rtU.DesiredSpeed1 = 80;
rtU.activate1 = TRUE;

#elif defined(WHEEL)
pwmInit();
adcInit(&ADCInterrupt);
ledsInit();

#elif defined(DASHBOARD)
ledsInit();
buttonsInit(&functionLedSet);
#endif // cc//wheel//DASHBOARD
```

We maken in ons project ook gebruik van extra functionpointers die vanuit de CAN opgeroepen kunnen worden. Op basis van de message die ontvangen wordt in de CAN zal de juiste functie voor de juiste ECU uitgevoerd worden.

```
void (*functionCANSpeed)(char value) ;
void (*functionCANButton)(char value);
void (*functionCANForce)(uint16_t force);
void (*functionCANSetSpeed)(char speed);
void (*functionCANCCOn) (char CANOn);
```

We hebben ook afgesproken om gebruik te maken van enums om de knoppen overzichtelijk te houden.

```
typedef enum BUTTON {
    /*! north button */    NORTH,
    /*! south button */   SOUTH,
    /*! west button */    WEST,
    /*! east button */    EAST,
    /*! center button */  CENTER
} t_BUTTON;
```

### 6.2.1 ECU voor de cruise control

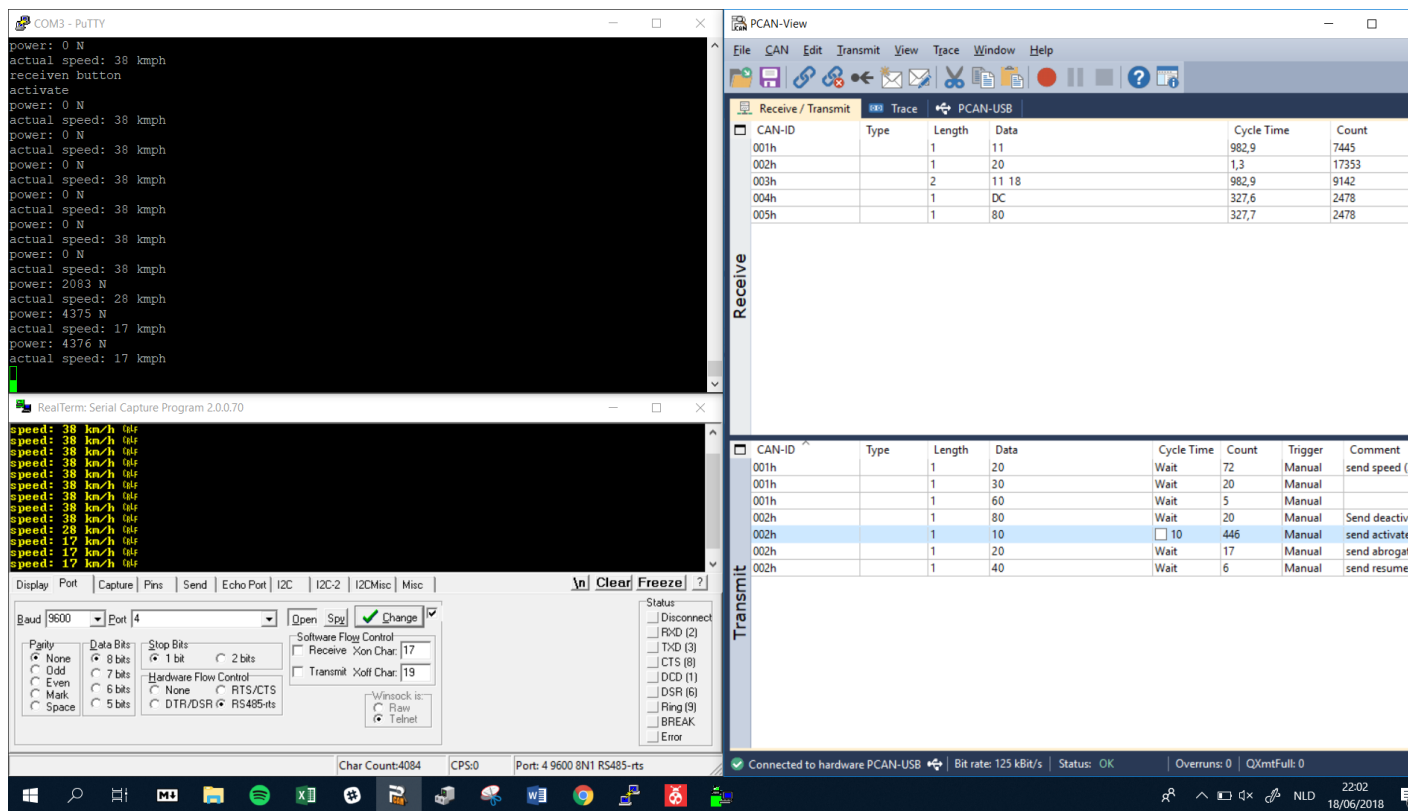
In de ECU voor de cruise control kunnen we gebruik maken van het simulink model dat we eerder gemaakt hebben. Deze ECU krijgt de actuele speed van de wielen en de desired speed mee. Deze ECU moet ook rekening houden met informatie van het dashboard, die de verschillende states kunnen doorsturen. Op basis van deze informatie kan de cruise control de power doorsturen naar de motor. Hierbij stuurt de ECU ook of dat de Cruise Control aanstaat. We kunnen deze ECU testen aan de hand van RS232 wanneer we nog geen CAN hebben. Anders kunnen we deze ECU testen door gebruikt te maken van de PCAN-View.

### 6.2.2 ECU voor de wielen

De ECU die dient voor de wielen, moet enkel informatie uit sturen op basis van de potentiometer. Deze informatie is de actuele snelheid die de auto rijdt. Deze ECU zal niks moeten ontvangen.

### 6.2.3 ECU voor het dashboard

De ECU van het dashboard zal al de informatie die verstuurd wordt door de andere ECU's ontvangen. Deze informatie kan dan aan de gebruiker getoond worden. De gebruiker kan ook doormiddel van knoppen een bericht versturen naar de cruise control. Over de verschillende states. Alweer kunnen we dit testen doormiddel van CAN of door het gebruik van RS232, samen met het gebruik van de knoppen en de ledjes. Hierdoor kunnen we dit zien als een goede integratie test van de verschillende drivers.



CONFIDENTIAL AND PROPRIETARY

© Universiteit Antwerpen, All rights reserved.