Katholieke Hogeschool Sint-Lieven
Departement Industrieel Ingenieur
Opleiding Elektronica-ICT
Afstudeerrichting elektronica
Gebroeders Desmetstraat 1, 9000 Gent

Integratie van een CAN bus in een ADPMS

Eindverhandeling ingediend tot
het behalen van de graad van
Master in Industriële Wetenschappen: Elektronica-ICT
en aangeboden door: **Stijn Wielandt**

Promotor	ing. Jean-Pierre Goemaere	Academiejaar 2010 - 2011
Copromotor	ing. Geoffrey Ottoy	
Externe promotoren	Peter Matthijs	
	Koen Puimège	

Hierbij geven de auteur(s) van dit eindwerk aan het bestuur van het departement industrieel ingenieur van KAHO Sint-Lieven de uitdrukkelijke toestemming om dit werk in bruikleen af te staan aan eender welke persoon, organisatie of firma, het ten dienste te stellen van de studenten en het geheel of gedeeltelijk te kopiëren.

Deze eindverhandeling was een examen; de tijdens de verdediging vastgestelde fouten werden niet gecorrigeerd. Het gebruik als referentie in publicaties is toegelaten na gunstig advies van de promotor van KAHO Sint-Lieven, vermeld op het titelblad.

Dankwoord

Deze masterproef werd mogelijk gemaakt dankzij KAHO Sint-Lieven en QinetiQ Space NV.

Mijn bijzondere dank gaat uit naar mijn promotoren van de hogeschool, dhr. Jean-Pierre Goemaere en dhr. Geoffrey Ottoy, die ik uitgebreid zou willen bedanken voor hun constructieve kritieken en voor het vrijmaken van hun tijd. Hierdoor kon mijn masterproef tot een goed einde gebracht worden.

Ook dhr. Peter Matthijs en dhr. Koen Puimège, mijn promotoren bij QinetiQ Space NV, wens ik te bedanken voor de ondersteuning en opvolging van mijn thesis. Dankzij hen kon mijn masterproef uitgroeien tot een unieke, verrijkende ervaring.

Bij de uitvoering van mijn masterproef kon ik bovendien rekenen op de technische kennis van de werknemers van QinetiQ Space NV. Dhr. Ronny De Vos en dhr. Dirk Steels wens ik te bedanken voor hun ondersteuning bij het schrijven van een VHDL ontwerp en de problemen die daarbij kwamen kijken. Verder ben ik ook dhr. David-Pierre Giard zeer dankbaar voor het delen van zijn kennis inzake VHDL simulaties. Zijn moeite om –eventueel meermaals– een probleem uit te leggen in een vreemde taal, werd sterk geapprecieerd. Ook dhr. Andy De Wilde en mevr. Lies Ottevaere wil ik graag bedanken voor hun inspanningen bij het opstellen van een hardware test bench. Tenslotte zou ik graag dhr. Luc De Busser bedanken voor het schrijven van een DMA driver voor Linux. Zonder deze driver zou het onmogelijk geweest zijn om het gewenste resultaat te bereiken.

Abstract

De CAN bus wordt reeds verscheidene jaren gebruikt in de industrie. In de ruimtevaart moet deze technologie zijn intrede echter nog maken. In dit toepassingsgebied zijn betrouwbaarheid en radiatiebestendigheid van groot belang. Voor deze masterproef zal de integratie getest worden van een CAN bus voor on-board communicatie in een ADPMS (Advanced Data and Power Management System), een on-board computer voor kleine satellieten. Deze computer is opgebouwd uit een set van cPCI modules met Actel RTAX FPGA's en wordt aangedreven door een LEON2 (SPARC V8 compatible) CPU met het RTEMS operating system. De opdracht omvat de integratie van een VHDL IP core in een bestaande cPCI module die ook gebruikt wordt voor de MIL1553b interface, een militaire databus voor lucht- en ruimtevaart. Dit vereist een studie van onder andere de bestaande MIL1553b interface module, de CAN bus en de gebruikte on-chip AMBA AHB bus. Om de CAN bus te implementeren in de bestaande hardware zal ook een radiatiebestendige implementatie van een CAN transceiver bestudeerd worden. Het ontwerp wordt in eerste fase uitvoerig getest en geverifieerd met behulp van ModelSim. Hiervoor zullen testscripts geschreven worden in assembler, die door een simulatieprocessor uitgevoerd worden in een VHDL testbench. Deze testbench zal naast een simulatieprocessor ook de modellen bevatten van de ontworpen CAN interface module, een geheugenmodule en enkele CAN nodes. Na de simulaties zal de hardware getest worden op een prototype van de cPCI module met behulp van een USB-to-CAN adapter en een PC. Hiervoor zal testsoftware geschreven worden in C voor de LEON2 processor.

Trefwoorden: CAN bus - ruimtevaart - radiatiebestendig - VHDL testbench

Inhoudsopgave

1 Inleiding **1**

2 CAN bus **4**

2.1 Inleiding 4
2.2 Fysieke laag 5
2.2.1 Hardware kenmerken 5
2.2.2 Signaalniveaus 6
2.2.3 Arbitratie 8
2.3 Data link laag 8
2.3.1 Codering 9
2.3.2 Foutafhandeling 9
2.3.3 Frames 10
2.4 Hogere protocols 13

3 CAN in space **15**

3.1 Fysieke laag 16
3.1.1 Ervaring met CAN bus in de ruimtevaart 16
3.1.2 Aanbevelingen voor CAN bus in de ruimtevaart 17
3.1.3 Stralingsbestendige CAN controller 18
3.1.4 Stralingsbestendige CAN transceiver 19
3.2 Data link laag 24
3.3 Hogere protocols 24

4 CAN interface module **26**

4.1 CAN IP core 26
4.2 Algemene structuur 27
4.2.1 CAN transceivers 28
4.2.2 CAN interface FPGA 28

5 VHDL test bench **33**

5.1 Algemene structuur 34
5.1.1 SRAM module 36
5.1.2 Simulatie CPU 36

5.2 CAN interface board . . . 37
5.3 CAN testnodes . . . 39
5.3.1 Functionaliteit . . . 40
5.3.2 Gebruik . . . 42
5.3.3 Ontwerp . . . 44
5.3.4 CAN testnodes test bench . . . 48
5.3.5 Resultaat CAN testnodes test bench . . . 49
5.3.6 Integratie in test bench . . . 52

6 VHDL simulaties 53
6.1 Test procedures . . . 54
6.2 Registertoegang . . . 54
6.2.1 Testscript . . . 54
6.2.2 Resultaat . . . 55
6.3 Data transfer CAN testnode - CAN interface board . . . 56
6.3.1 Testscript . . . 56
6.3.2 Resultaat . . . 57
6.4 Data transfer CAN interface board - CAN testnodes . . . 58
6.4.1 Testscript . . . 58
6.4.2 Resultaat . . . 59
6.4.3 Probleemanalyse . . . 60
6.4.4 Oplossing . . . 61
6.5 Code Coverage . . . 63

7 Hardware tests 65
7.1 VHDL synthese, place & route en programming . . . 65
7.1.1 Synthese . . . 65
7.1.2 Place & route . . . 66
7.1.3 Programming . . . 68
7.2 Selectie test board . . . 69
7.3 Opbouw test bench . . . 70
7.3.1 OBC TSIM . . . 71
7.3.2 ATB-CIM . . . 71
7.3.3 Aarding . . . 71
7.3.4 Verbindingsprocedure . . . 72
7.4 Registertoegang . . . 72
7.5 Functional tests . . . 74
7.5.1 Data reception: CAN bus A en B . . . 75
7.5.2 Data transmission: CAN bus A en B . . . 75
7.5.3 RTR reception . . . 76

7.5.4 RTR transmission . . . 76
7.5.5 Extended data frame reception . . . 77
7.5.6 Extended data frame transmission . . . 77
7.5.7 Message filtering . . . 78
7.6 Error injection tests . . . 78
7.6.1 Missing acknowledgement . . . 78
7.6.2 Receive from wrong CAN bus . . . 78
7.6.3 Buffer overrun . . . 79
7.7 Performance tests . . . 79
7.7.1 Data reception . . . 80
7.7.2 Data transmission . . . 80
7.8 Samenvatting . . . 80

8 Besluit 82

A Beschrijving van deze masterproef in de vorm van een wetenschappelijk artikel 88

B Resultaat simulaties 93
B.1 Registertoegang . . . 93
B.2 Data reception . . . 96
B.3 Data transmission . . . 98

C Poster 101

D Structuur van de bijgeleverde DVD 103

Lijst van figuren

1.1 Structuur ADPMS 2

2.1 Schematische voorstelling van een CAN bus netwerk 5
2.2 Soorten lijnterminatie 6
2.3 Voorstelling van de signaalniveau's op de differentiële CAN bus 7
2.4 Differentiële signaalniveau's op de CAN bus volgens ISO 11898 7
2.5 Structuur van standard en extended data- en remote transfer frames 12

3.1 Schematische voorstelling van een cold redundant CAN bus 18
3.2 Schematische voorstelling van een CAN bus transceiver 21
3.3 Vervangschema van de CAN bus met RS-485 transceiver 21
3.4 AC vervangschema van het weerstandsnetwerk 22

4.1 Schematische voorstelling van de CAN interface module 28
4.2 Blokschema van de GRCAN IP core van Gaisler Research 30
4.3 Gedetailleerd blokschema van de FPGA 32

5.1 Schematische voorstelling van de CAN interface FPGA test bench (Rood = zelf geschreven code, wit = overgenomen code) 35
5.2 In- en uitgangen van de CAN testnodes 43
5.3 Schema voor het ontvangen van CAN frames 45
5.4 Statendiagram van de CAN testnode 46
5.5 Schematische voorstelling van de CAN node test bench 48
5.6 Waveform bij een ontbrekende ACK bit en incorrecte RB0 bit 50
5.7 Waveform van een succesvolle reception en transmission test 51

6.1 Waveform van de data reception test 58
6.2 Waveform van het transmit probleem 61
6.3 Waveform van de AHB transfer indien de GRCAN core ingesteld wordt als default master 62
6.4 Waveform van een correcte AHB transfer 63

7.1 Lay-out van de FPGA na place & route 68
7.2 Opbouw van de hardware test bench 70

7.3 Resultaat van een PCI scan: de CAN interface module wordt gedetecteerd 73
7.4 Succesvol resultaat van de registertoegang test 74
7.5 Inhoud van de receive buffer na ontvangst van een standaard CAN data frame . . 75
7.6 Het CAN frame dat verzonden werd door de CAN interface module 76
7.7 De testkaart ontvangt een RTR frame en antwoordt hierop 76
7.8 Buffers van de GRCAN core na uitzending van een RTR frame en ontvangst van het antwoord 77
7.9 Receive buffer van de GRCAN core na ontvangst van een extended data frame . . 77
7.10 De testkaart ontvangt het extended CAN data frame zonder problemen 77
7.11 Errors indien een transmissie plaatsvindt op een inactieve bus 79

D.1 De directory structuur van de bijgeleverde DVD 103
D.2 De gebruikte directory structuur voor het ontwerp 104

Lijst van tabellen

3.1 Driver karakteristieken van een RS-485 en CAN transceiver 20
3.2 Receiver karakteristieken van een RS-485 en CAN transceiver 20
3.3 Differentiële spanning op de CAN bus met RS-485 transceiver, getoetst aan ISO 11898 24

5.1 Resultaat als verschillende nodes verschillende signalen zenden op de CAN bus . 38
5.2 Resultaat als verschillende nodes verschillende signalen zenden op de CAN bus: oplossing met zwakke signalen 39
5.3 Werking van het CAN transceiver VHDL model 39
5.4 Errordetectie in de CAN testnodes 42
5.5 Beschrijving van het tx_ide_id signaal 43
5.6 Beschrijving van het tx_dlc_data signaal 43
5.7 Informatie voor elk van de geïmplementeerde CAN nodes 52

7.1 Samenvatting van de uitgevoerde hardware tests 81

Lijst van de belangrijkste acroniemen

ADPMS	Advanced Data and Power Management System
AHB	Advanced High performance Bus
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
ATB-CIM	Automated Test Bench - CAN Interface Module
CAN	Controller Area Network
CIM	CAN interface Module
cPCI	compact Peripheral Compliant Interface
CRC	Cyclic Redundancy Check
ECSS	European Cooperation for Space Standards
EGSE	Electrical Ground Support Equipment
ESA	European Space Agency
FPGA	Field Programmable Gate Array
GRCAN	Gaisler Research CAN bus
IP	Intellectual Property
OBC TSIM	On-Board Computer Test Simulator
MPM	Main Processor Module
RTR	Remote Transfer Request
SCPU	Simulation Central Processing Unit
SEL	Single Event Latch-up
SEU	Single Event Upset
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

1 Inleiding

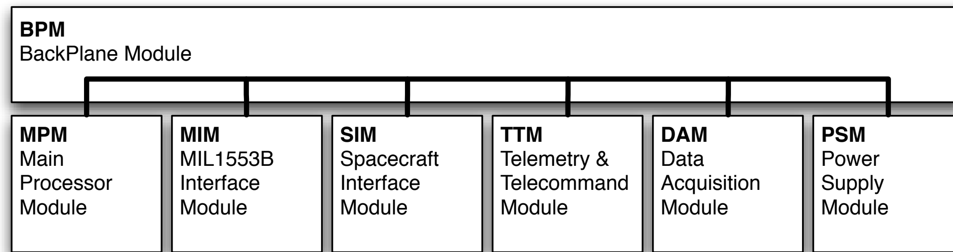
De CAN[1] bus wordt reeds verscheidene jaren gebruikt voor industriële toepassingen, vooral dan in de automobielindustrie. De laatste jaren is er echter ook interesse ontstaan om deze databus te gebruiken voor on-board communicatie in ruimtevaarttoepassingen. De CAN bus heeft immers verscheidene voordelen ten opzichte van de klassieke databussen in de ruimtevaart, zoals ESA OBDH (European Space Agency On-Board Data Handling) en MIL1553b, een militaire databus voor lucht- en ruimtevaart. De belangrijkste troeven van CAN bus zijn de lage kostprijs (dankzij de uitgebreide adoptie door de industrie), hoge betrouwbaarheid, prioritaire bus access, grote beschikbaarheid van commerciële hardware (zowel componenten als VHDL cores) en een uitgebreid en wijdverspreid aanbod van protocols, van fysische tot applicatielaag. De CAN bus verschilt op sommige vlakken echter fundamenteel van de gangbare databussen in de ruimtevaart. Zo is de CAN bus een multimaster bus die aan frame labeling doet in plaats van node labeling en bovendien werkt deze bus asynchroon. Een meer gedetailleerde uiteenzetting van het CAN protocol wordt gegeven in hoofdstuk 2 op pagina 4.

Voor deze masterproef is het de bedoeling om een CAN bus te integreren in een ADPMS[2], een onboard computer voor kleine satellieten. De ontworpen module kan gebruikt worden voor command en control van payload, zoals propulsiesystemen en sensoren. Een mogelijke toepassing voor het ontwerp is de ExoMars Descent Module, een ruimtesonde die in 2016 gelanceerd wordt en een on-board CAN bus netwerk zal hebben. De boordcomputer is opgebouwd uit cPCI kaarten die in een backplane geplugd worden, zoals te zien is in figuur 1.1. Hierin is onder andere een MIL1553b interface module te zien. Deze module bevat een FPGA waarin een VHDL IP core geïntegreerd werd voor MIL1553b ondersteuning. Het is de bedoeling dat hieraan uiteindelijk ondersteuning voor het CAN protocol toegevoegd zal worden. Dit valt echter buiten het kader van deze masterproef, maar er zal rekening mee gehouden worden dat het ontwerp van de CAN interface module nauw aanleunt bij de structuur van de bestaande MIL1553b module, zodat een latere fusie van beide ontwerpen zonder veel moeite verloopt.

Aangezien het ontwerp moet voldoen aan de strenge eisen uit de ruimtevaart is een redundante

[1] Controller Area Network

[2] Advanced Data and Power Management System



Figuur 1.1: Structuur ADPMS

CAN bus een must, dient het ontwerp uitvoerig getest te worden en moet het bovendien stralingsbestendig te zijn. Omdat stralingsbestendige CAN controllers niet of nauwelijks voorhanden zijn, is men genoodzaakt een VHDL IP core te integreren in een stralingsbestendige FPGA. Het ontwerp dient daarna uitgebreid getest te worden in zowel computersimulaties als in een prototype. Bovendien dient ook de stralingsbestendige implementatie van CAN transceivers bestudeerd te worden.

In hoofdstuk 3 op pagina 15 wordt de algemene implementatie van een CAN bus voor ruimtevaarttoepassingen bestudeerd voor de fysische laag, de data link laag en de hogere lagen. Hierin wordt ook een oplossing uitgewerkt voor de stralingsbestendige implementatie van CAN transceivers, gebaseerd op een stralingsbestendige RS-485 transceiver. Daarna wordt in hoofdstuk 4 op pagina 26 het ontwerp van de CAN interface module besproken, de cPCI kaart die uiteindelijk in het ADPMS gebruikt kan worden. Eerst wordt een studie verricht van de bestaande CAN IP cores, waarna een algemene structuur uitgewerkt wordt en een gedetailleerde beschrijving van de CAN interface FPGA gegeven wordt.

Aangezien het VHDL ontwerp uitgebreid getest dient te worden in simulaties, wordt een volledige test bench in VHDL ontworpen, zoals beschreven in hoofdstuk 5 op pagina 33. In deze VHDL test bench wordt het ontwerp van de CAN interface module via een PCI interface gekoppeld aan een simulatieprocessor (SCPU) en een geheugenmodule. Verder wordt de CAN interface module in simulatie gekoppeld aan een CAN netwerk opgebouwd uit CAN testnodes die ook zelf ontworpen worden in VHDL. Om de correcte werking van deze CAN testnodes te verifiëren, wordt nog een aparte test bench ontworpen die ook in hoofdstuk 5 beschreven wordt.

In hoofdstuk 6 op pagina 53 worden de computersimulaties van de VHDL test bench uitgewerkt. Hiervoor worden assembly testscripts geschreven voor de simulatieprocessor. Er worden drie simulaties besproken. In een eerste simulatie wordt de registertoegang tot de CAN core getest. Daarna wordt een data transfer gesimuleerd van een testnode naar de CAN interface module en tenslotte wordt een data transfer van de CAN interface module naar de testnodes gesimuleerd.

Wanneer de simulaties met succes afgerond worden, wordt een hardware implementatie getest, zoals besproken wordt in hoofdstuk hoofdstuk 7 op pagina 65. Hiervoor wordt het VHDL ontwerp van de CAN interface FPGA in een prototype van de MIL1553b interface module geprogrammeerd. Op dit prototype worden bovendien twee stralingsbestendige CAN transceivers gemonteerd, waarna de module geïntegreerd wordt in een hardware test bench. Deze test bench bestaat uit twee computers, waarvan één computer de CAN interface module bevat en de andere een commerciële CAN testkaart. Met behulp van verschillende test applicaties kan de functionaliteit van de CAN interface module uitgebreid getest worden. De tests blijven hier niet beperkt tot de drie eenvoudige tests die uitgevoerd worden in de simulaties. De hardware tests kunnen opgedeeld worden

in vier categorieën. Ten eerste wordt een registertoegang test uitgevoerd, net als bij de simulaties. Vervolgens wordt een reeks functionele tests uitgevoerd, waarin het normale gedrag van de CAN interface module getest wordt. Daarna worden zogenaamde error injection tests uitgevoerd, tests waarin het gedrag van de CAN interface module geobserveerd wordt indien fouten optreden. Tenslotte worden ook nog performance tests uitgevoerd om de correcte werking te verifiëren wanneer de CAN interface module kritisch belast wordt.

2
CAN bus

Om het vervolg van deze scriptie te kunnen volgen, is een zekere basiskennis van het CAN protocol vereist. In dit hoofdstuk zullen de belangrijkste karakteristieken van de ISO 11898 standaard[1] besproken worden, de meestgebruikte CAN standaard.

2.1 Inleiding

De CAN bus werd in 1986 ontwikkeld door Robert Bosch voor de auto industrie. De grote betrouwbaarheid van het protocol en de korte berichten die het transporteert, maken het protocol uiterst geschikt voor het uitlezen van sensors en aansturen van bijvoorbeeld motors of temperatuurregelaars. Dankzij deze eigenschappen werd het protocol in vele toepassingsgebieden geïntroduceerd, zoals de medische sector, productielijnen en domotica. De verspreiding van de CAN bus zorgt er bovendien voor dat ontwikkelingskosten voor CAN systemen zeer laag zijn geworden.

De CAN bus is een seriële multimaster broadcasting bus. Dit wil zeggen dat de transmissie over een enkel dradenpaar gebeurt en dat alle netwerknodes data mogen verzenden via de bus. Bovendien zal deze data door alle nodes ontvangen worden, wat dataconsistentie over het netwerk garandeert. Voor het verzenden van data worden dus geen nodes geadresseerd. In plaats daarvan worden de CAN frames geïdentificeerd met een identifier. Aan de hand van deze identifier beslist elke CAN node voor zichzelf of het ontvangen frame nuttige informatie bevat en of het al dan niet zal worden opgenomen. Aangezien men hier niet aan node adressering doet, zijn CAN bus systemen hot pluggable: extra nodes kunnen zonder aanpassingen in een netwerk toegevoegd worden, zelfs als het netwerk actief is.

Een andere manier om dataconsistentie te bekomen is de foutafhandeling. Elke node op de bus zal elk frame -onafhankelijk van de identifier- controleren op errors. Indien een error plaatsvindt, zal dit kenbaar gemaakt worden aan alle andere nodes op de bus, waardoor een heruitzending van het defecte frame volgt. Een andere unieke eigenschap van de CAN bus is de bitsgewijze arbitratie zonder bus arbiter. Aangezien er geen arbiter is en alle nodes master zijn, kan elke node op elk moment data op de bus zetten. Om data collision te voorkomen, zal elke node daarom controleren

of een uitgezonden bit ook effectief op de bus verschijnt. Is dit niet het geval, dan zal de node zijn transmissie stoppen. Een meer gedetailleerde beschrijving van de arbitratie volgt in de volgende paragrafen.

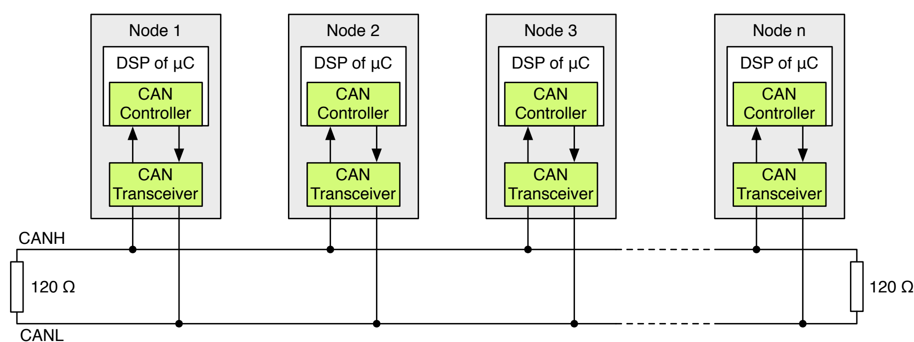
2.2 Fysieke laag

De fysieke laag is de onderste laag uit het OSI referentiestelsel en legt alle hardwarespecificaties voor het netwerk vast. Hiertoe behoren zowel het transmissiemedium als de elektrische specificaties, bijvoorbeeld de signaalniveaus. De ISO 11898 standaard legt echter niet de volledige fysische laag vast. Het gedeelte over de kabels en connectoren wordt vastgelegd in hogere protocols zoals bijvoorbeeld CANopen[2], die niet tot de ISO 11898 standaard behoren. Uiteraard dienen deze kabels en connectoren wel compatibel te zijn met de opgelegde hardwarespecificaties van ISO 11898.

2.2.1 Hardware kenmerken

De CAN bus ondersteunt transmissiesnelheden van 125 kbps tot 1 Mbps. Men dient evenwel de lengte van het CAN netwerk aan te passen aan de transmissiesnelheid, aangezien de propagatiedelay belangrijker wordt bij grotere transmissiesnelheden. Daarom varieert de maximum lengte van een CAN netwerk van 2 km tot 40 m. Als vuistregel kan men stellen dat: transmissiesnelheid (Mbps) x bus lengte (m) $\leq$ 50.

Verder moet men er rekening mee houden dat het aantal nodes op een CAN netwerk beperkt wordt tot 30. Indien men meer nodes op de bus wenst te plaatsen, dient men transceivers te gebruiken met hoog impedante ingangen, repeaters en kwalitatief betere kabels. Anders zou de zendende node teveel stroom moeten sinken of sourcen, wat kan leiden tot defecten. In figuur 2.1 wordt een schematische voorstelling van een CAN bus netwerk weergegeven.

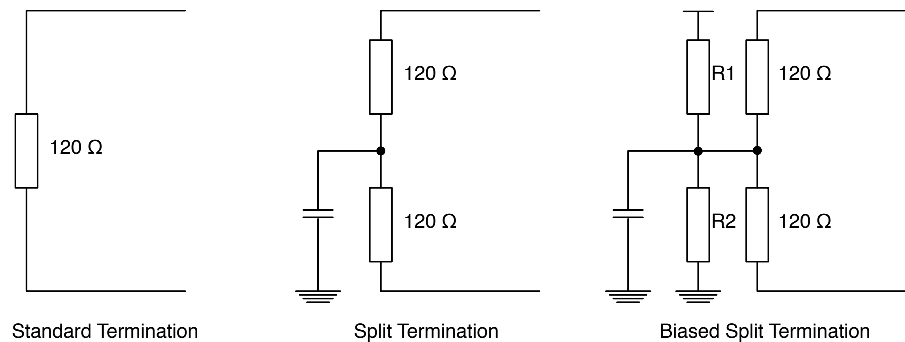


Figuur 2.1: Schematische voorstelling van een CAN bus netwerk

Voor de kabels dient men shielded of unshielded twisted pairs te gebruiken. Meer gedetailleerde specificaties worden door het hoger protocol opgegeven. Men dient er wel voor te zorgen dat de karakteristieke impedantie Z_0 120 Ω bedraagt. Men zal er dus ook op moeten toezien dat de bus aan de twee uiteinden getermineerd wordt met een 120 Ω impedantie om reflecties te voorkomen. Deze lijnterminatieweerstanden worden bovendien best niet voorzien op een node. Als een node zou uitvallen, zou de lijnterminatie ook kunnen wegvallen, wat ongewenste reflecties kan veroorzaken. In figuur 2.2 zijn drie soorten lijnterminatie te zien.

De eerste soort is de standaard 120 Ω lijnterminatie. Deze voldoet in de meeste gevallen en is dus ook de meestgebruikte. De tweede soort is de split termination. Hierbij worden twee weerstanden van 60 Ω in serie geplaatst. Het is belangrijk dat beide weerstanden gelijk zijn. De maximum tolerantie op de weerstandswaarde bedraagt dus ± 1 %. Op het koppelpunt van de twee weerstanden wordt een ontkoppelcondensator geplaatst. Deze filtert de hoogfrequente componenten op de bus, wat de emissie vermindert. Een typische waarde voor deze capaciteit bij een transmissiesnelheid van 1 Mbps is 4.7 nF. Hiervoor ligt het -3dB punt op 1.1 Mbps.

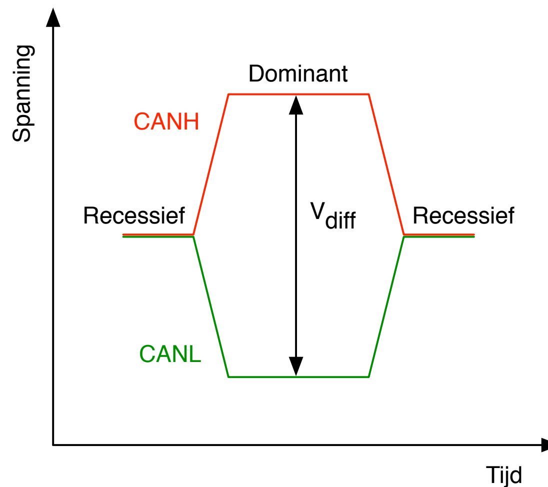
Een laatste soort lijnterminatie is de biased split termination, een soort split termination waarbij de recessieve common mode spanning constant gehouden wordt. Aangezien het niet verplicht is om alle CAN nodes met een gemeenschappelijke massa te verbinden, kunnen de verschillende nodes op een verschillend grondniveau terecht komen. Dit zorgt ervoor dat op de signaallijnen een common mode spanning ontstaat. Als men deze spanning constant houdt, zoals gebeurt bij biased split termination, wordt de emissie sterk verminderd, wat EMC problemen kan voorkomen. Om deze stabilisatie te bekomen, wordt een spanningsdeler gebruikt. Ook hier is het belangrijk dat de weerstandswaarde niet meer dan 1 % afwijkt, teneinde een spanning van $V_{DD}/2$ te bekomen over de condensator.



Figuur 2.2: Soorten lijnterminatie

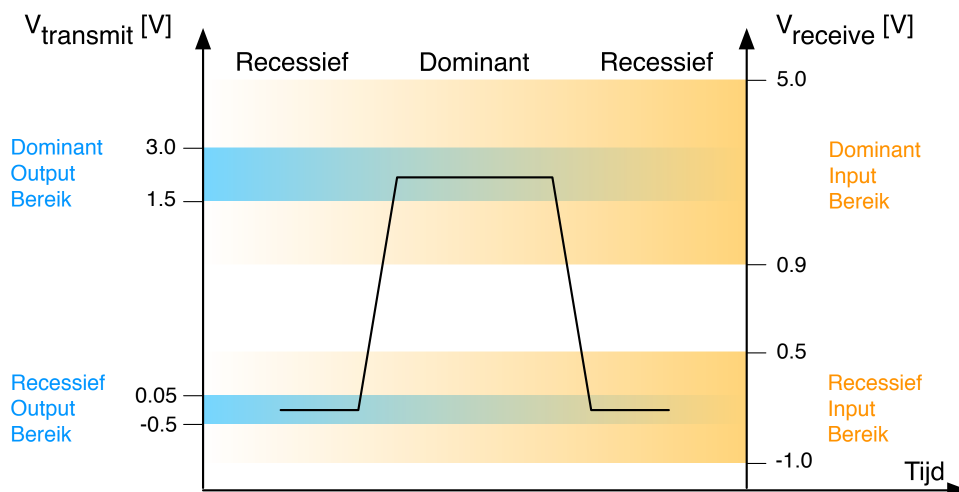
2.2.2 Signaalniveaus

Een van de redenen voor het robuuste karakter van CAN bus systemen, is het gebruik van differentiële signalen. Bits worden op de bus voorgesteld door een differentiële spanning tussen een CANH signaal en een CANL signaal. In optimale omstandigheden wordt een 1 voorgesteld door een differentiële spanning van 0 V. Dit wordt de recessieve staat genoemd, waarbij de bus niet actief aangestuurd wordt. Een 0 wordt daarentegen voorgesteld door een differentiële spanning van 2 V. In dit geval spreekt men van een dominante staat, waarbij de bus wel actief aangestuurd wordt door de transceiver. Aangezien de absolute spanningen variëren rond 2.5 V, kan het CANL signaal dan 2.5 V of 1.5 V aannemen, terwijl het CANH signaal de waarden 2.5 V of 3.5 V aanneemt. Dit wordt visueel voorgesteld in figuur 2.3.



Figuur 2.3: Voorstelling van de signaalniveau's op de differentiële CAN bus

Deze ideale omstandigheden kunnen uiteraard niet altijd bewerkstelligd worden aangezien het signaal onderhevig is aan attenuatie. Hier houdt de ISO 11898 standaard dan ook rekening mee door onder- en bovengrenzen te definiëren voor zowel de ingangs- als uitgangssignalen. Een recessieve staat moet aangestuurd worden met een differentiële spanning van -0.5 V tot 0.05 V, terwijl de dominante staat aangestuurd wordt met een differentiële spanning tussen 1.5 V en 3 V. Aan de ingang van een transceiver vergroten deze marges echter: een recessieve staat wordt gedetecteerd als het differentiële signaalniveau zich bevindt tussen -1.0 V en 0.5 V, een dominante input wordt gekenmerkt door een signaalniveau tussen 0.9 V en 5.0 V. Deze eigenschappen kunnen teruggevonden worden in figuur 2.4.



Figuur 2.4: Differentiële signaalniveau's op de CAN bus volgens ISO 11898

Men dient nog op te merken dat de nodes niet verbonden moeten zijn met eenzelfde massa. Dit zorgt ervoor dat nodes onderling verschillende massapunten kunnen hebben. Hierdoor ontstaat een ground offset, ook wel common mode bus voltage genaamd. Dit is het spanningsniveau op de CANL en CANH lijnen als de bus zich in recessieve staat bevindt. De ISO 11898 standaard legt

voor deze ground offset een marge vast van -2.0 V tot 7.0 V.

2.2.3 Arbitratie

De CAN bus maakt gebruik van non-destructieve bitsgewijze arbitratie zonder arbiter. Dit wil zeggen dat elke node data over de bus kan zenden zonder dat ze hiervoor toestemming moet vragen. Bovendien zal nooit een heruitzending van het winnende bericht nodig zijn ten gevolge van de arbitratie. We beschouwen volgend voorbeeld, waarbij twee nodes tegelijkertijd data op de bus zenden. Uiteraard kan dit voorbeeld uitgebreid worden voor een onbeperkt aantal nodes. Als de twee nodes een recessieve bit uitzenden, zal geen van beide nodes de bus actief aansturen. De bus zal dus een recessieve staat aannemen en geen van beide nodes merkt iets van de andere node. Als beide nodes een dominante bit uitzenden, zal de bus de dominante staat aannemen aangezien ze actief aangestuurd wordt door beide nodes. Ook hier treedt dus geen enkel probleem op en kunnen beide nodes de transmissie voortzetten. Als daarentegen door de ene node een recessieve bit op de bus gezet wordt, terwijl de andere node een dominante bit op de bus zet, dan zal er een probleem optreden. De bus zal immers actief aangestuurd worden door de tweede node en bijgevolg een dominante staat aannemen. De node die de recessieve bit uitzond zal de dominante staat op de bus waarnemen en bijgevolg de transmissie stoppen. Ze zal direct overschakelen naar ontvangstmodus en het desbetreffende bericht verder ontvangen. Het bericht zou immers voor de verliezende node bestemd kunnen zijn. De winnende node daarentegen merkt niets van de arbitratie en zet de transmissie gewoon verder. Het komt er dus op neer dat de node die het langst dominante bits uitzendt, de arbitratie zal winnen. Daarom zal het begin van elk CAN frame een identifier bevatten die een maat is voor de prioriteit van het desbetreffend bericht. Hierop wordt dieper ingegaan in paragraaf 2.3.

Een andere belangrijke factor die een rol speelt bij de arbitratie, is het sampling tijdstip. Elke node zal immers, na het uitzenden van een bit, de staat van de bus controleren, om eventueel de transmissie te stoppen als de arbitratie verloren werd (als de uitgestuurde bit niet gelijk is aan de ingelezen bit). Het tijdstip waarop dit samplen gebeurt, wordt bepaald door de synchronisatie en de propagation delay van het systeem. Men mag immers de staat van de bus pas controleren als men er zeker van is dat alle signalen, ook uit de verste nodes van het netwerk, aangekomen zijn. De ISO 11898 standaard legt het sampling tijdstip vast op $t_{75\%}$, op 75 % van de bitlengte. De propagation delay van het systeem kan berekend worden volgens: $t_{prop} = 2 \cdot [t_{bus} + t_{transceiver,out} + t_{transceiver,in}]$. Hieruit blijkt de invloed van de lengte van de bus. Aangezien de propagation delay bij grote transmissiesnelheden klein moet blijven, zal t_{bus} beperkt moeten worden. De buslengte zal dus beperkt worden, zoals beschreven werd in paragraaf 2.2.1 op pagina 5. De vermenigvuldiging met 2 wordt uitgevoerd omdat een bit soms beantwoord wordt door een ontvangende node (binnen dezelfde bittijd). Een voorbeeld hiervan is de acknowledgement bit, die in paragraaf 2.3 aan bod komt. Het signaal moet zich in dit geval verplaatsen naar het andere uiteinde van de bus en terug.

2.3 Data link laag

De data link laag bestaat uit het transfer protocol en is in feite de kern van het CAN protocol. Ze vervult functies als het vormen van frames, arbitratie, foutcontrole, -signalisatie en -beperking, acknowledgements als bevestiging van al dan niet goed ontvangen frames, het starten van transmissie en gedeeltelijke controle van bit timing en transfer rate. Het transfer protocol kan zowel in hardware als software gerealiseerd worden. In de praktijk zal het echter bijna altijd in hardware in

de vorm van een CAN controller geïntegreerd worden.

2.3.1 Codering

Het CAN protocol maakt geen gebruik van Manchester codering, wat een belangrijk voordeel is voor de benodigde bandbreedte van de bus. Anders zou deze immers dubbel zo groot moeten zijn. Een nieuw probleem dat zich dan echter wel vormt, is de synchronisatie. De manchester codering zorgt voor een groot aantal flanken in de transmissie, waarop gesynchroniseerd kan worden. Als op de CAN bus echter verschillende opeenvolgende bits gelijk zijn, zullen geen flanken voorkomen waarop getriggerd kan worden. Daarom past men bij het CAN protocol bit stuffing toe: als vijf opeenvolgende bits in een frame identiek zijn, zal een extra complementaire bit ingevoegd worden teneinde de synchronisatie correct te laten verlopen. Men dient hierbij op te merken dat een bit stuffing bit ook mee kan tellen als een van de vijf identieke bits. Dus als een recessieve bit stuffing bit gevolgd wordt door vier recessieve frame bits, zal hierna een dominante bit stuffing bit volgen. Bij de ontvanger worden de bit stuffing bits herkend en weggefilterd. Bit stuffing zal enkel plaats vinden in data en remote transfer request frames, die in paragraaf 2.3.3 op de pagina hierna verder besproken zullen worden.

2.3.2 Foutafhandeling

Detectie

Het CAN protocol neemt verschillende functies op in verband met de foutafhandeling. In eerste instantie zal aan foutdetectie gedaan worden. Alle bits die op de bus gezet worden, zullen ook gemonitord worden. Zo kan men vijf verschillende soorten errors detecteren. Bit errors worden gedetecteerd door de zendende node en komen voor wanneer een verzonden bit niet op de bus verschijnt. Uiteraard worden bit errors niet gedetecteerd tijdens arbitratie, acknowledgement, error frames en overload frames. In deze gevallen kunnen (of moeten) recessieve bits op de bus overschreven worden door dominante bits, wat verduidelijkt wordt in paragraaf 2.3.3 op de volgende pagina. Form errors komen voor als een bit in een frame niet overeenkomt met een waarde die vastgelegd is in het CAN protocol. Stuffing errors worden dan weer gedetecteerd als in een data of remote transfer request frame minstens 6 identieke opeenvolgende bits gedetecteerd worden. CRC errors komen vanzelfsprekend voor als een CRC check van een data of remote transfer request frame een negatief resultaat geeft. Als bij de transmissie van een frame geen acknowledgement verkregen wordt, zal een acknowledgement error gegenereerd worden.

Beperking

Het CAN protocol zorgt ervoor dat een onderscheid gemaakt wordt tussen het kort en langdurig falen van een CAN node. Op deze manier kan de invloed van een node aangepast worden volgens het aantal fouten dat ze genereert of detecteert, waardoor het aantal fouten op de bus beperkt kan worden. Het CAN protocol voorziet hiervoor twee tellers: een transmit error counter en een receive error counter. Deze worden geïncrementeerd met een bepaalde gewichtsfactor volgens het aantal fouten dat optreedt bij de desbetreffende node. Als een van deze tellers groter wordt dan 255, zal de node in 'bus off' mode gezet worden, ze mag dan niet meer deelnemen aan transfers. Indien een van de tellers groter is dan 127, wordt de bus in 'error passive' mode gezet, een mode die de invloed van de node op de CAN bus zal verkleinen maar niet uitschakelen. Dit zal vooral

invloed hebben op error frames, wat besproken zal worden in paragraaf 2.3.3. Normaal gezien zullen beide tellers kleiner zijn dan 127 en zal de bus zich in 'error active' mode bevinden. Ze heeft dan sneller toegang tot de bus en bijgevolg een grotere invloed op vlak van foutdetectie of -signalisatie.

Signalisatie

Als een fout gedetecteerd wordt door een node, zal dit meegedeeld worden aan alle andere nodes op de bus, waardoor eventueel een heruitzending kan volgen. Deze signalisatie gebeurt met behulp van passieve of actieve error frames, conform de mode waarin de signaliserende node zich bevindt. De error frames zullen besproken worden in paragraaf 2.3.3.

2.3.3 Frames

Het CAN protocol maakt gebruik van 4 soorten frames. De belangrijkste zijn de data- en remote transfer frames. Verder bestaan er nog error- en overload frames.

Data- en remote transfer frames

Een data frame bestaat uit verschillende bitvelden, zoals weergegeven in figuur 2.5. Al deze velden hebben een vaste structuur en dus vaste lengte, met uitzondering van het data veld, dat 0 tot 8 bytes kan innemen.

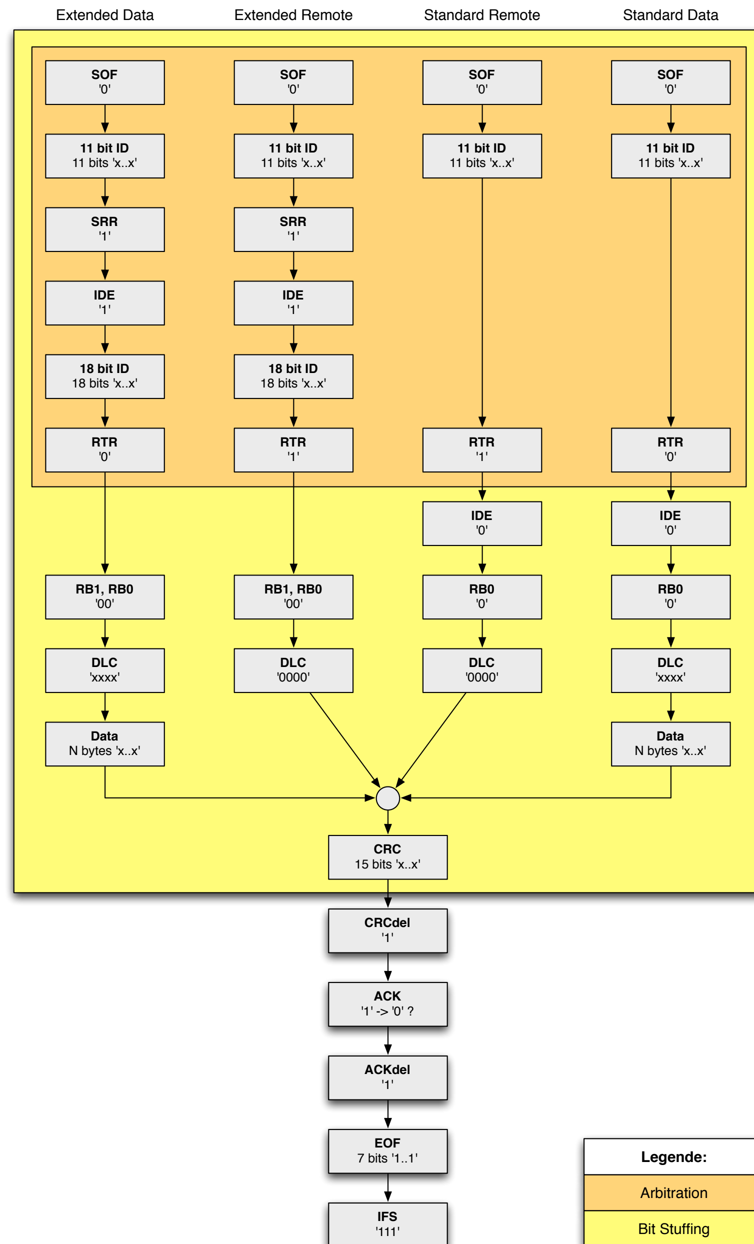
Een frame start altijd met een Start Of Frame (SOF) bit. Deze dominante bit geeft de start van een transmissie aan. Hierna volgt een arbitratieveld dat op zijn minst bestaat uit een identifier en een Remote Transfer Request (RTR) bit. Bij extended frames zal de identifier 29 bits innemen, waar dit slechts 11 bits zijn bij standaard frames. In beide gevallen mogen de 7 most significant bits niet allemaal recessief zijn. Of het al dan niet om een extended frame gaat, wordt aangegeven door de Identifier Extended (IDE) bit. Deze IDE bit zorgt er tevens voor dat standaard CAN frames de arbitratie altijd zullen winnen van extended CAN frames.Verder duidt de RTR bit aan of het om een dataframe of een RTR-frame gaat. In een dataframe kunnen 0 tot 8 bytes data meegestuurd worden. Een RTR frame daarentegen bevat geen data en wordt uitgezonden door een node om data op te vragen. Het soort data dat opgevraagd wordt, wordt geïdentificeerd met de (al dan niet extended) identifier. Stel dat een node (toevallig) terzelfdertijd de gevraagde data uitzendt, dan zal de identifier gelijk zijn aan die van het RTR frame. Toch geeft dit geen arbitratieproblemen dankzij het verschil in RTR bit. Het RTR frame zal de arbitratie namelijk verliezen door de recessieve RTR bit, waardoor de gevraagde data direct ontvangen kan worden. Dit zorgt voor een efficiente transmissie van data, die heruitzending in dit geval overbodig maakt. Bij de extended frames worden de eerste 11 bits van de identifier gevolgd door een Substitute Remote Request (SRR) bit, die altijd recessief is.

Na het arbitratieveld volgen nog een of twee gereserveerde bits (RB, Reserved Bit), die per definitie dominant zijn. Hierna volgt de Data Length Code (DLC), een 4 bits veld dat een getal van 0 tot 8 vormt. Dit getal geeft de lengte van het data veld (in bytes) weer. Na het data veld, dat 0 tot 8 bytes lang is, volgt een CRC veld dat bestaat uit een 15 bits CRC sequentie, gevolgd door een recessieve bit, de CRC delimiter. De CRC sequentie wordt berekend door een bitsequentie te delen door de volgende 16 bits polynoom: 1100 0101 1001 1001, vaak voorgesteld als: $X^{15}+X^{14}+X^{10}+X^{8}+X^{7}+X^{4}+X^{3}+1$. De bitsequentie die gedeeld wordt, bestaat uit alle

voorgaande bitvelden van SOF tot dataveld, verlengd met 15 dominante (0) bits. De bit stuffing bits worden niet in beschouwing genomen voor de CRC berekening. De rest van de deling kan maximaal 15 bits lang zijn. Dit is de CRC sequentie. Als een ontvangende node alle bitvelden van SOF tot en met CRC sequentie deelt door dezelfde 16 bits polynoom, dient dit een 0 als rest te geven. Deze CRC sequentie garandeert dat elke verandering van maximaal 5 willekeurige bits in het frame gedetecteerd zal worden. Wat ook belangrijk is om te vermelden, is dat alle bitvelden van SOF tot en met de CRC sequentie (dus zonder de CRC delimiter) onderworpen zijn aan de regels van bit stuffing. Voor de daaropvolgende velden wordt geen bit stuffing meer toegepast.

Na het CRC veld volgt nog een Acknowledgement (ACK) veld dat bestaat uit een ACK slot bit en een ACK delimiter bit, die beiden als recessieve bits uitgezonden worden. Als een ontvangende node het bericht correct ontvangen heeft, zal ze het ACK slot echter invullen met een dominante bit. Het komt er dus op neer dat het frame correct aangekomen is als de zender een recessief ACK slot wegschreef, maar een dominant ACK slot op de bus ziet. Als het ACK slot recessief blijft, is er een fout opgetreden bij de transmissie.

Tenslotte wordt het frame afgesloten met 7 recessieve bits, het End Of Frame (EOF), gevolgd door ten minste 3 recessieve bits Inter Frame Space (IFS). De IFS kan ook langer zijn als de bus idle wordt (als er geen nieuwe frames op de bus komen), er kan dan een willekeurig aantal recessieve bits op de bus komen.



Figuur 2.5: Structuur van standard en extended data- en remote transfer frames

Error en overload frames

Error en overload frames hebben beiden een eenvoudige, identieke structuur, bestaande uit een 6 bits vlaggenveld, gevolgd door een delimiter van 8 recessieve bits. Het verschil ligt in het gebruik van beide frames. Overload frames worden verzonden als een node de toevloed van data niet kan volgen of als een dominante bit gedetecteerd wordt in de eerste drie bits van de IFS, ook de intermission genoemd. Overload frames worden bijgevolg verzonden *buiten* het data of RTR frame. Hun doel is een delay te introduceren tussen opeenvolgende data of RTR frames. Voor overload frames zal het vlaggenveld altijd bestaan uit 6 dominante bits.

Een error frame wordt verzonden als een node een error in een ontvangen data of RTR frame detecteert. Een error frame wordt dus verzonden *tijdens* een transmissie. Voor error active nodes zal het vlaggenveld uit 6 dominante bits bestaan, net als bij overload frames. Hierdoor zal het bericht waarin een error gedetecteerd werd, vernietigd worden omdat het niet meer voldoet aan de bit stuffing regels. Dit zorgt er dan voor dat alle andere nodes een bit stuffing error zien, waardoor deze ook allemaal een error frame op de bus zullen zetten. Het aantal opeenvolgende dominante bits op de bus kan daardoor oplopen tot 12 (6 door de node die de error detecteerde en 6 door de andere nodes die reageren op het error frame). De zendende node zal de error frames detecteren, waardoor later eventueel een heruitzending kan plaatsvinden. Error passive nodes hebben een vlaggenveld dat bestaat uit 6 recessieve bits, waardoor hun invloed op de bus geminimaliseerd wordt. De recessieve bits kunnen immers overschreven worden door dominante bits.

2.4 Hogere protocols

De protocols voor de hogere lagen zijn niet vastgelegd in de ISO 11898 standaard. Afhankelijk van het toepassingsgebied zijn hier verschillende protocols voor vastgelegd zoals CANopen, DeviceNet en J1939. De meestgebruikte standaard is CANopen (EN 50325) die ontwikkeld is door de CiA (CAN in Automation) groep. Het protocol omvat een applicatielaag, communicatieprofielen, richtlijnen voor kabels en connectoren en een beschrijving van gebruikte SI-eenheden en prefixen. De standaard is tot stand gekomen om ontwikkelaars de mogelijkheid te bieden CAN apparatuur te ontwikkelen zonder dat de details van het CAN protocol (ISO 11898) gekend moeten zijn. Aangezien voor deze thesis geen software drivers geschreven moeten worden, zal de beschrijving van CANopen beperkt worden.

Het CANopen protocol beschrijft dat elke node een object dictionary bevat, een lijst met objecten die uitgewisseld kunnen worden op het netwerk. Deze communicatie objecten (COB) kunnen gespreid worden over verschillende CAN frames met eventueel vooraf vastgelegde identifiers. Er zijn verschillende soorten communicatie objecten. De service data objects (SDO) zorgen voor client-server verbindingen tussen twee nodes en hebben een onbeperkte frame-grootte. Process data objects (PDO) kunnen zowel synchroon (cyclisch en acyclisch) als asynchroon verlopen. Dit zijn korte berichten met hoge prioriteit die gebroadcasted worden en waarvoor geen confirmation wordt gestuurd. Synchronisatie objecten (SYNC) worden cyclisch op de bus gezet en maken het verzenden van synchrone process data objects mogelijk. Met network management objects (NMT) kan het gedrag van zogenaamde NMT slave devices gecontroleerd worden. In de software kan dus wel een onderscheid gemaakt worden tussen master en slave nodes, wat niet mogelijk is in hardware. De NMT objecten zorgen ervoor dat nodes kunnen deelnemen aan gedeelde toepassingen, ze controleren de errorstatus van de nodes en ze zorgen voor transfers van configuratiedata over het CAN netwerk. Verder zullen de NMT objecten gebruikt worden voor node en life guarding. Bij node guarding houdt de NMT master een lijst bij van actieve nodes. Hiertoe worden op regelmatige tijdstippen NMT objecten uitgestuurd naar de slaves. In het omgekeerde geval controleren de NMT slaves of de master nog actief is, men spreekt dan van life guarding. Hierbij controleren de nodes of er aan node guarding gedaan wordt door de master. Ter vervanging van het node guarding protocol kan men ook gebruik maken van het heartbeat protocol, dat gebruik maakt van zogenaamde heartbeat signals om de activiteit van nodes te bepalen. Emergency objects (EMCY) zijn berichten die met hoge prioriteit door een producer naar meerdere consumers gestuurd worden in geval van een device failure. Voor de distributie van tijdsinformatie in het CAN netwerk kan men gebruik maken van twee methoden. Time stamp objects geven in een 48 bit getal de

verstreken tijd sinds 1 januari 1984 weer in milliseconden. Als deze verdeling niet nauwkeurig genoeg is, kan het CANopen high resolution time stamp protocol gebruikt worden. Dit zorgt voor een nauwkeurigheid van 1 microseconde door een high resolution time stamp te versturen in een process data object.

3
CAN in space

De laatste jaren is de interesse in de CAN bus voor on-board communicatie in ruimtevaarttoepassingen toegenomen, met het oog op de vervanging van de huidige MIL1553b of ESA OBDH[1] bussen die single master zijn, grote kosten met zich meebrengen en een hoog vermogenverbruik kennen. De CAN bus biedt voordelen zoals de multimaster opbouw van het netwerk, wat de CAN bus flexibel maakt en daarom ook uiterst geschikt voor payload systemen van bijvoorbeeld satellieten. Verder is de implementatie van de CAN bus relatief goedkoop dankzij het grote aantal ontwikkelingstools dat beschikbaar is. Bovendien verzekeren technieken zoals bit checks, bit stuffing, CRC, enz. een zeer grote betrouwbaarheid. Deze betrouwbaarheid werd reeds uitgebreid bewezen in aardse toepassingen zoals auto's en de industrie. Bovendien wordt de CAN bus ook op aarde gebruikt in veeleisende en ruwe omgevingen, die gelijkaardig zijn aan de ruimte. Toch kunnen de standaard CAN systemen niet overgenomen worden voor ruimtevaarttoepassingen, aangezien de bestaande elektronische componenten voor CAN bus niet stralingsbestendig zijn en aangezien de huidige CAN standaard geen ondersteuning biedt voor hardware redundantie.

Deze ontwikkelingen hebben ertoe geleid dat een *ESA-Industry Working Group on CAN bus* opgericht werd, een onderdeel van ESTEC (European Space Research and Technology Centre). Deze instantie onderzoekt het gebruik van CAN bus voor ruimtevaarttoepassingen, zodat een ECSS (European Cooperation for Space Standardisation) standaard opgesteld kan worden. Hiervoor werd de beperkte ervaring met CAN bus systemen in de ruimte in beschouwing genomen, net als de ervaring met aardse toepassingen. Tot op heden is er echter nog geen standaard voorhanden, daarom worden de specificaties gebaseerd op een draft standaard van het ECSS[6]. Deze standaard beschrijft de protocols en services die gebruikt worden bovenop de ISO 11898 standaard om te voldoen aan de specifieke eisen van de ruimtevaart, en hoe deze dan geïmplementeerd kunnen worden. Belangrijk is dat deze ECSS standaard geen aanpassingen aan de bestaande ISO 11898 standaard aanbrengt, maar enkel uitbreidende protocols voorziet.

De CAN bus in ruimtevaarttoepassingen zal vooral gebruikt worden voor het synchroon en realtime ontvangen van data, afkomstig van sensoren. Ook zal de bus gebruikt worden om eveneens

[1] Tot nu toe werd vooral gebruik gemaakt van de MIL1553b bus[3] (een militaire databus voor lucht- en ruimtevaart), of de ESA OBDH[4] (European Space Agency On-Board Data Handling) bus in satellieten[5].

synchroon en real-time commando's te verzenden naar actuators. Verder kan de bus gebruikt worden om grotere datapakketen asynchroon uit te wisselen tussen nodes. Zeker in de toekomst, als meer intelligente nodes op CAN systemen aangesloten worden, zullen grotere datatransfers plaatsvinden. Tenslotte zal de CAN bus ook gebruikt worden voor een nauwkeurige tijdsdistributie over het CAN netwerk.

3.1 Fysieke laag

Voor de implementatie van elektronische systemen in ruimtevaarttoepassingen dient men rekening te houden met de hoog energetische straling die men terugvindt in de ruimte. Hiertoe behoort kosmische straling, die bestaat uit protonen, alfastraling (He^{+}), zware ionen, x-straling en gammastraling. De meeste schadelijke deeltjes hiervan hebben een energie van 0.1 GeV tot 20 GeV. Daarnaast treft men ook de zogenaamde solar particle events aan, protonen, zware ionen en x-straling met een energie van enkele GeV, die uitgezonden worden door de zon bij zonnestormen. Verder vindt men nog elektronen en protonen met een energie van 10 MeV tot 100 MeV terug in het geomagnetisch veld, de zogenaamde Van Allen stralingsgordels.

Een inslag van een hoog energetisch deeltje kan duizenden elektronen losslaan uit een kristalrooster en de kristalstructuur veranderen, wat desastreuze effecten kan hebben op elektronische schakelingen in de vorm van ruis en spikes. In digitale schakelingen kunnen zo single-event effects (SEE) optreden, waarvan de belangrijkste single-event upsets (SEU) en single-event latchups (SEL) zijn. Een single-event upset treedt op als een bit tijdelijk omkeert ten gevolge van een invallend ion. Een effect dat geen permanente schade toebrengt aan een halfgeleider. Een single-event latchup daarentegen, zorgt ervoor dat een thyristorstructuur (PNPN) gevormd wordt door een invallend ion. Hoewel dit effect in principe omkeerbaar is door de voedingsspanning opnieuw aan te leggen, kan reeds onomkeerbare schade toegebracht zijn aan de halfgeleider door grote stromen in de halfgeleiderstructuur.

Deze effecten hebben aanleiding gegeven tot de ontwikkeling van stralingsbestendige elektronische componenten. Ze zijn bestemd voor gebruik in de ruimte, vliegtuigen op grote hoogte, deeltjesversnellers, nucleaire reactors en militaire toepassingen. Deze componenten zijn meestal bestand tegen een stralingsdosis van enkele honderden krad[2]. Aangezien de ontwikkeling en het testen van dergelijke componenten vrij traag verloopt, duurt het meestal enkele jaren vooraleer stralingsbestendige componenten beschikbaar zijn voor een nieuwe technologie. Om deze reden zijn er nagenoeg geen stralingsbestendige CAN controllers of transceivers op de markt.

3.1.1 Ervaring met CAN bus in de ruimtevaart

Tot op vandaag is er zeer weinig ervaring met CAN bus systemen in de ruimte. De onderzoeksgroep "Surrey Satellite Technologies Ltd"(SSTL) van de University of Surrey implementeerde de CAN bus reeds van 1996 tot 2004 in zijn satellieten[7], net als het Pakistaanse ruimtevaart agentschap[8]. Hiervoor werden echter COTS (Commercial Off-The-Shelf) componenten gebruikt, wat over het algemeen de mogelijkheid biedt om nieuwe technologieën snel te integreren in satellieten. Deze componenten kunnen maximaal 10 krad verdragen, wat hen enkel geschikt

[2]De rad is een eenheid die vooral nog gebruikt wordt voor stralingsbestendige elektronica, hoewel het gebruik ervan ontmoedigd wordt door het NIST (U.S. National Institute of Standards and Technology). 100 krad komt overeen met een stralingsdosis van 1000 Sievert, 1000 keer hoger dan de hoogste stralingspiek die gemeten werd bij de kernramp in het Japanse Fukushima in 2011.

maakt voor low orbit toepassingen[9]. Zo werden de desbetreffende satellieten gebruikt op 600 km tot 1000 km hoogte, waar de bescherming door de atmosfeer nog vrij groot is en de stralingsdosis 1krad/jaar bedraagt voor de componenten, afgeschermd met 5 mm aluminium. Bij het gebruik van de satellieten werd ongeveer 1 single-event upset (SEU) per MB SRAM waargenomen per dag, wat gecorrigeerd kan worden met behulp van EDAC (Error Detection And Correction). In de tientallen jaren dat de satellieten in orbit waren, werden 4 single-event latchups (SEL) gedetecteerd. Deze fouten werden opgevangen door zogenaamde "cold redundant modules" te voorzien, modules die inactief blijven tot wanneer ze de werking van een defecte module moeten overnemen[10].

Sinds 2004 bleek echter een meer betrouwbare, stralingsbestendige implementatie aangewezen. Hiervoor bleken stralingsbestendige componenten tot 100 krad nodig te zijn, die de componentenkost met een factor 1500 zouden verhogen.

Voor de implementatie van een CAN bus blijkt het gebruik van COTS componenten niet aangewezen, aangezien de module mogelijks gebruikt zal worden voor deep space toepassingen zoals de ExoMars missie, waar stralingsniveaus veel hoger kunnen oplopen[11]. Bovendien speelt in dit soort missies het gewicht een zeer grote rol, waardoor dikke afschermingen niet mogelijk zijn en ook cold redundant modules vermeden moeten worden. Uiteraard is voor deze missies de betrouwbaarheid van zeer groot belang, waardoor enkel een stralingsbestendige oplossing aanvaardbaar is.

3.1.2 Aanbevelingen voor CAN bus in de ruimtevaart

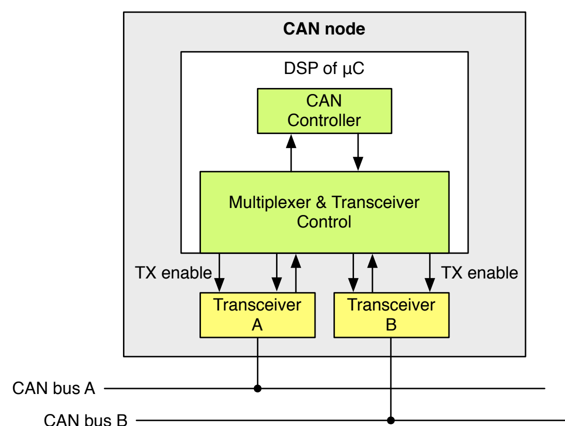
Voor de verbinding van CAN nodes wordt een lineaire multi-drop structuur of daisy chain aanbevolen. Voor de multi-drop structuur zal men moeten werken met stubs op een hoofdlijn, waar bij daisy chain topologie alle nodes doorgelust worden, waardoor elke node een dubbele connector dient te voorzien. Verder dient een CAN bus netwerk minstens 64 nodes aan te kunnen. Voor de interconnectie worden afgeschermde kabels aangeraden waarvan de afscherming aan de ground van elke node wordt aangesloten, de connectors kunnen vrij gekozen worden volgens wat best past in de toepassing. Een belangrijke eis is het plaatsen van lijnterminatie weerstanden buiten de nodes, dus in de kabel of connector verwerkt.

Voor de transceivers dienen de elektrische karakteristieken van de ISO 11898 standaard gevolgd te worden. Men kan gebruik maken van commerciële transceivers, die in principe betere eigenschappen hebben op vlak van EMC en overspanning. Bovendien zorgt deze implementatie voor een lagere kost en ontwikkelingstijd. Men moet er echter zeker van zijn dat de gebruikte transceiver bestand is tegen single-event latchups (SEL). De Philips TJA1050 CAN transceiver kan voldoen voor gebruik in ruimtevaarttoepassingen, maar aangezien Philips geen details vrijgeeft over het productieproces, dienen de transceivers eerst zelf getest te worden op stralingsbestendigheid[12]. Indien men echter wil voldoen aan de formele eisen van ESA, dient men een volledig radiatiebestendige oplossing te implementeren. Dit kan men bereiken door een volledige transceiver op te bouwen uit discrete componenten (weerstanden en transistoren), wat meestal een hoge ontwikkelingskost en extra gewicht met zich meebrengt, ofwel door standaard space-qualified RS-485 transceivers te gebruiken op een niet-conventionele manier (door het transmit signaal te verbinden met de enable klem)[12].

Verder kan men galvanische scheiding overwegen om ground loops te elimineren, om ruisgevoeligheid te verkleinen en om de schakeling te beschermen bij defecten of statische ontladingen. Deze galvanische scheiding zal dan plaatsvinden tussen de CAN controller en CAN transceiver. Men dient op te merken dat deze scheiding niet conform de ISO 11898 standaard is en dus be-

ter vermeden kan worden als compatibiliteit nagestreefd wordt. Galvanische scheiding brengt verder ook een significante toename in kostprijs en gewicht met zich mee, bovendien zijn de optische detectors gevoelig aan straling, waardoor single-event latchups en single-event upsets kunnen voorkomen. Het falen van een optocoupler kan bovendien de werking van een gehele CAN bus belemmeren. Een andere manier van galvanisch scheiden is het gebruik van optische vezels. Hiermee worden invloeden van magnetische en elektrische interferentie voorkomen, wordt het gewicht gereduceerd, en kan in principe de data rate opgedreven worden. Men dient ook hier op te merken dat deze aanpassing niet conform de ISO 11898 standaard is en dat de werking van optische vezels in deep space nog niet volledig bekend is. Een laatste manier van galvanische scheiding is het gebruik van transformators, maar aangezien het CAN protocol geen return-to-zero protocol is, kunnen lange reeksen ‘0’ of ‘1’ ervoor zorgen dat een magnetische flux in de transformator wordt opgebouwd, waardoor deze beschadigd kan raken. Om dit probleem op te lossen zijn aanpassingen nodig aan het CAN protocol, dit is dan ook de reden waarom een galvanische scheiding met behulp van transformators sterk afgeraden wordt.

In de ruimtevaart wordt het algemeen aangeraden om redundantie te voorzien[10]. Om deze reden kan een redundante CAN bus voorzien worden om bijvoorbeeld fouten van kabels en connectors uit te sluiten. Hiervoor is een vorm van redundancy management nodig, waarbij een zogenaamde redundancy master de actieve bus kiest. Dit wordt echter gestuurd vanuit een hoger protocol (in dit geval CANopen). Voor de fysieke implementatie heeft men keuze uit twee topologieën: selective bus access of cold redundancy, en parallelle bus access of hot redundancy. Over het algemeen wordt selective bus access aanbevolen, waarbij één CAN controller twee transceivers met twee fysieke CAN bussen aanstuurt. Hiervoor dient men een mechanisme te voorzien voor busselectie tussen de CAN controller en de twee transceivers. Het is dus enkel mogelijk om één bus tegelijkertijd te gebruiken. Bij parallelle bus access wordt het gehele CAN netwerk redundant opgebouwd, dus ook de CAN controllers. Hierbij is het in principe mogelijk om de twee bussen tegelijkertijd te gebruiken. Een schematische voorstelling van de selective bus access vindt men terug in figuur 3.1.



Figuur 3.1: Schematische voorstelling van een cold redundant CAN bus

3.1.3 Stralingsbestendige CAN controller

Momenteel is slechts één stralingsbestendige CAN transceiver op de markt: de AT7908E van Atmel[13], die ontworpen werd door Aurelia Microelettronica S.p.A., een Italiaans ruimtevaartbedrijf. Deze stand-alone CAN controller biedt echter geen ondersteuning voor een redundante

CAN bus en kan niet aangesloten worden op de PCI of AHB bus die gebruikt wordt in het ADPMS. Daarom blijft een implementatie van een CAN controller in een FPGA met behulp van VHDL IP cores als enige optie over. Voor de FPGA wordt een RTAX2000S/SL type van Actel gebruikt, een type dat geoptimaliseerd werd voor gebruik in de ruimte, met extra laag vermogenverbruik en stralingsbestendigheid tot 300 krad. Bovendien wordt gebruik gemaakt van non volatile antifuse technologie, die zorgt voor een robuust ontwerp, dat evenwel niet herprogrammeerbaar is. Tot op vandaag, na meer dan 1.8 miljoen uur testen, werden nog steeds geen antifuse fouten waargenomen.

3.1.4 Stralingsbestendige CAN transceiver

Ook het gebruik van een stralingsbestendige CAN transceiver blijkt niet eenvoudig. Ook hiervan is slechts één model op de markt, ook een model van Aurelia Microelettronica S.p.A. (CANTRAN [14]). De transceiver blijkt echter niet bruikbaar aangezien het om een relatief grote, zware DIL28 package gaat, een package die onbruikbaar is voor missies waarbij gewicht een grote rol speelt, zeker als ook een redundante CAN bus geïmplementeerd wordt, waardoor twee transceivers noodzakelijk zouden zijn. Bovendien verkeert de transceiver nog in prototype stadium en zou geparticipeerd moeten worden in de verdere ontwikkeling, wat de transceiver volledig onbruikbaar maakt. Hoewel het gebruik van commerciële CAN transceivers aangeraden wordt, is dit in deze toepassing niet mogelijk aangezien voldaan moet worden aan de normen van ESA, zeker aangezien het ontwerp mogelijks gebruikt zal worden in deep space, waar veel grotere stralingsniveaus heersen. De opbouw van een CAN transceiver uit discrete componenten zou teveel gewicht en kosten met zich meebrengen, daarom werd gekozen voor een implementatie met stralingsbestendige RS-485 transceivers, zoals aangeraden wordt door ECSS[6] en Omnisys[12].

CAN transceiver met RS-485 transceiver

Figuur 3.2 op pagina 21 toont het schema van een CAN transceiver met een stralingsbestendige RS-485[15] transceiver (DS16F95W van National Semiconductor). Dit schema werd niet volledig zelf ontworpen. Het gaat om een bestaand ontwerp, waarvan de werking gecontroleerd werd en waarvan de weerstandswaarden nog berekend moesten worden. Om de correcte werking te begrijpen, is het aangewezen om de werking van een CAN transceiver en een RS-485 transceiver te vergelijken met elkaar.

In de driver karakteristieken, die men kan waarnemen in tabel 3.1 op de pagina hierna, kan men opmerken dat uitgang *A* van een RS-485 transceiver gebruikt kan worden als *CANH* klem en uitgang *B* als *CANL* klem. In dit geval moet men verzekeren dat de de driver input (*DI*) klem van de RS-485 transceiver gegarandeerd hoog (H) is. De enable klem van de RS-485 transceiver kan dan gebruikt worden als CAN driver input. Men dient er wel op te letten dat het CAN driver input signaal nog geïnverteert moet worden!

In het schema wordt deze aansluiting verwezenlijkt door de driver input (*DI*) klem van de RS-485 transceiver aan de voedingsspanning te koppelen en de drive enable (*DE*) klem van de RS-485 transceiver aan het CAN drive (*CAN_D*) signaal te koppelen. De inversie van het *CAN_D* signaal dient dus door de CAN controller (de FPGA) te gebeuren! Deze situatie is niet 100 % compatibel met de ISO 11898 standaard, maar aangezien de controller en transceiver zich op dezelfde module bevinden, zal dit geen incompatibiliteit met andere ISO 11898 nodes veroorzaken.

Verder wordt in het driver circuit nog een hoogdoorlaatfilter waargenomen in de vorm van een RC kring met diode. Deze schakeling zorgt ervoor dat een defecte CAN controller die continu een dominante staat aanneemt, de CAN bus niet domineert en bijgevolg een degelijke werking zou verhinderen. De CAN controller wordt in dit geval van de transceiver ontkoppeld.

Tabel 3.1: Driver karakteristieken van een RS-485 en CAN transceiver

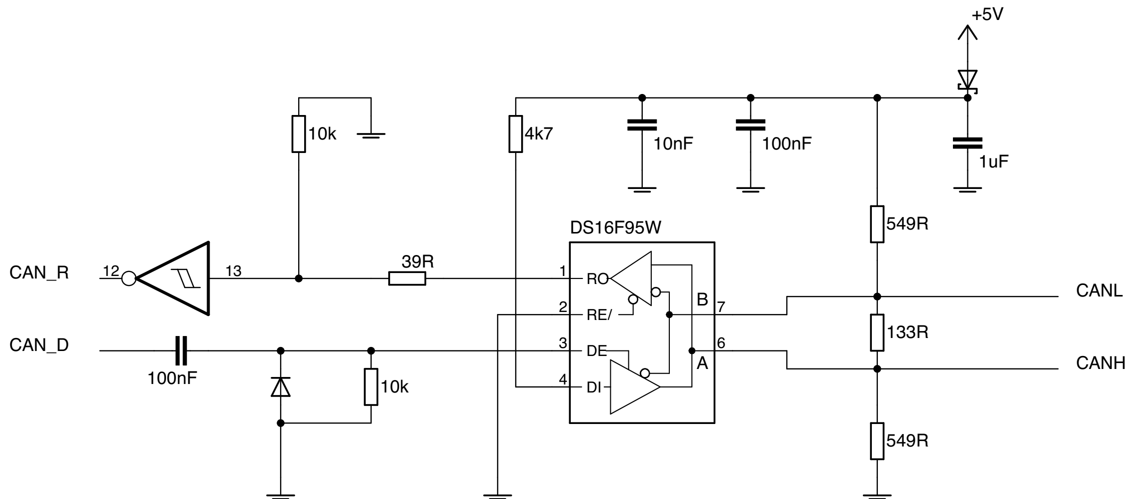
RS-485				CAN bus			
Driver Input	Enable	outputs		Driver Input	Outputs		
DI	*DE*	*A*	*B*	*D*	*CANH*	*CANL*	Bus State
H	H	H	L	L	H	L	Dominant
L	H	L	H				
X	L	Z	Z	H	Z	Z	Recessief

In tabel 3.2 kan men de receiver karakteristieken van beide transceivers vergelijken. Hieruit blijkt dat de RS-485 inputs A en B ook gebruikt kunnen worden voor respectievelijk het $CANH$ en $CANL$ signaal. Er zal evenwel een conversie moeten plaatsvinden die de differentiële spanningen omzet naar de gewenste niveaus. Dit wordt verder besproken. Ook in dit ontwerp dient men erop te letten dat het receive output (RO) signaal nog geïnverteerd dient te worden alvorens het te gebruiken als CAN receive (CAN_R) signaal.

Tabel 3.2: Receiver karakteristieken van een RS-485 en CAN transceiver

RS-485			CAN bus			
Differentiële Input	Enable	output	Inputs			Output
$V_{ID} = A - B$	$\overline{RE}$	RO	Bus State	$V_{ID} = V_{CANH} - V_{CANL}$	D	R
$\geq$ 0.2 V	L	H	Dominant	$\geq$0.9 V	X	L
			X	X	L	L
$\leq$ -0.2 V	L	L	Recessief	$\leq$0.5 V	H	H
			X	X	H	H
0.2 V $> V_{ID} >$ -0.2 V	L	?	?	0.5V $< V_{ID} <$ 0.9 V	H	?
X	H	Z				

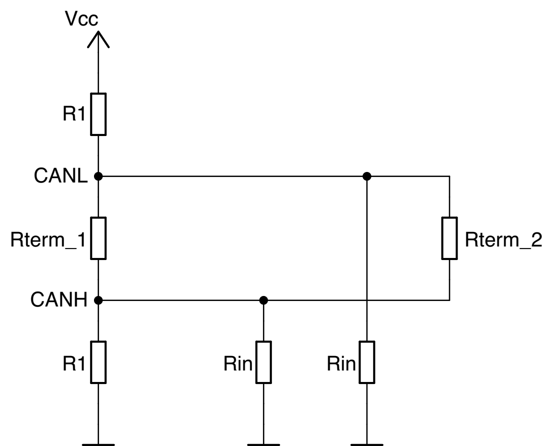
In het elektronisch schema ziet men dat de receive enable ($\overline{RE}$) klem van de RS-485 transceiver aan de massa hangt. In dit geval wordt de receiver dus sowieso enabled, een keuze om consistentie te creëren aangezien de driver functie ook niet disabled kan worden. Deze keer werd wel een discrete invertor gebruikt om de receive output (RO) van de RS-485 transceiver als CAN receive (CAN_R) signaal te gebruiken. Deze invertor dient louter voor logische niveauconversie. De RS-485 transceiver zal immers bij hoog niveau een 5 V signaal uitsturen, terwijl de CAN controller (de FPGA) met niveau's van 3.3 V werkt. De inversie die gaandeweg gebeurt, is dan een handige nevenwerking die ervoor zorgt dat het signaal ISO 11898 compatibel is. De 10 k weerstand die in de receive kring is aangesloten, dient enkel om een logisch signaal aan de controller aan te sluiten als de PCB niet volledig bestukt zou worden. De 39 Ω weerstand zorgt voor stroombeperking.



Figuur 3.2: Schematische voorstelling van een CAN bus transceiver

Tenslotte kan men nog een netwerk van 3 weerstanden opmerken aan de zijde van het CAN netwerk. Deze weerstanden verzekeren signaalniveaus conform de ISO 11898 standaard voor het zenden en ontvangen van signalen, alsook een correcte afsluiting van de CAN bus met de karakteristieke impedantie van 120 Ω. Hier werd gebruik gemaakt van standaard lijnterminatie, en dus niet van (biased) split terminatie, zoals beschreven in hoofdstuk 2 op pagina 4.

In figuur 3.3 wordt het vervangschema van de CAN bus met RS-485 transceiver gegeven. Hierin vertegenwoordigt R_{term_2} de normale afsluitweerstand van de CAN bus, die 120 Ω bedraagt. R_{term_1} is de afsluitweerstand die berekend zal worden. Ook de weerstandswaarde van R_1 zal berekend worden. De weerstanden R_{in} vertegenwoordigen de ingangsimpedantie van de RS-485 transceiver.



Figuur 3.3: Vervangschema van de CAN bus met RS-485 transceiver

In wat volgt, zal een berekening[3] gemaakt worden van de weerstanden R_{term_1} en R_1, aan de hand van het vervangschema uit figuur 3.3.

[3]De berekeningen werden gebaseerd op [16] (en gecorrigeerd)

Ten eerste kan een vergelijking opgesteld worden van de stromen in knooppunt $CANL$ en $CANH$:

$$\frac{V_{cc} - V_{CANL}}{R_1} = \frac{V_{CANL} - V_{CANH}}{R_{term_1}} + \frac{V_{CANL} - V_{CANH}}{R_{term_2}} + \frac{V_{CANL}}{R_{in}} \tag{3.1}$$

$$\frac{V_{CANH}}{R_1} + \frac{V_{CANH}}{R_{in}} = \frac{V_{CANL} - V_{CANH}}{R_{term_1}} + \frac{V_{CANL} - V_{CANH}}{R_{term_2}} \tag{3.2}$$

Als men $(V_{CANL} - V_{CANH})$ gelijk stelt aan V_{CAN}, bekomt men na aftrekking van vergelijking 3.1 en 3.2:

$$V_{CAN} = R_{in} \cdot \left(\frac{V_{cc}}{R_1} - \left(\frac{V_{CANL}}{R_1} + \frac{V_{CANH}}{R_1} \right) - 2 \cdot V_{CAN} \cdot \left(\frac{1}{R_{term_1}} + \frac{1}{R_{term_2}} \right) \right) \tag{3.3}$$

Een verdere vereenvoudiging geeft de differentiële spanning tussen de knooppunten CANL en CANH:

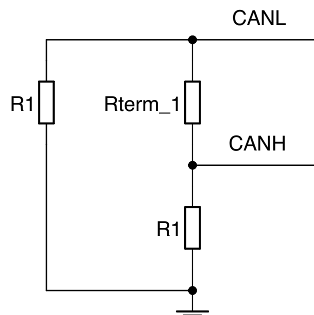
$$V_{CAN} = \frac{V_{cc}}{\frac{1}{R_1} + \frac{1}{R_{in}} + 2 \cdot \left(\frac{1}{R_{term_1}} + \frac{1}{R_{term_2}} \right)} \tag{3.4}$$

De RS-485 standaard[15] beschrijft een minimumbelasting van 375 Ω voor zowel klem A als B. Men kan dus besluiten dat de weerstandswaarden R_1 en R_{in} moeten voldoen aan vergelijking 3.5, aangezien zij parallel op deze uitgangen aangesloten zijn.

$$\frac{1}{R_1} + \frac{1}{R_{in}} \geq \frac{1}{375} \tag{3.5}$$

Bovendien wordt het weerstandsnetwerk op AC niveau gezien zoals weergegeven in figuur 3.4. Om een 120 Ω aanpassing te creëren, dienen de weerstandswaarden dus ook te voldoen aan vergelijking 3.6.

$$\frac{1}{2 \cdot R_1} + \frac{1}{R_{term_1}} = \frac{1}{120} \tag{3.6}$$



Figuur 3.4: AC vervangschema van het weerstandsnetwerk

Als men vergelijking 3.5 en 3.6 substitueert in vergelijking 3.4, samen met de kennis van $R_{term_2} =$ 120 Ω, bekomt men vergelijking 3.7 op de volgende pagina, die toelaat de weerstandswaarde van R_1 te berekenen.

$$R_1 = \left(\frac{V_{cc}}{V_{CAN}} + 1\right) \cdot \frac{250}{9} \tag{3.7}$$

In het schema van figuur 3.2 op pagina 21 bemerkt men een schottky diode aan de ingang die onder deze omstandigheden een spanningsval van ongeveer 0.3 V introduceert. De voedingsspanning V_{cc} bedraagt hier dus 4.7 V. Voor de differentiële spanning op de CAN bus dient men de minimum input van 0.2 V voor de RS-485 receiver in rekening te brengen, plus een ruismarge van 50 mV. V_{CAN} bedraagt dus 250 mV. Als deze gegevens ingevuld worden in vergelijking 3.7, bekomt men dat $R_1 = 550\ \Omega$. Deze waarde kan gerealiseerd worden met een 549 Ω weerstand uit de E-96 reeks.

De weerstandswaarde van R_{term_1} kan dan berekend worden met behulp van vergelijking 3.6 op de pagina hiervoor:

$$R_{term_1} = \frac{1}{\frac{1}{2\cdot549} + \frac{1}{120}} = 134.72\ \Omega \tag{3.8}$$

Deze waarde kan best benaderd worden met de E-96 waarde van 133 Ω.

Tenslotte kan men alle ingangs- en uitgangsspanningen van de RS-485 transceiver berekenen met de gevonden weerstandswaarden, om zo te controleren of de uitgewerkte oplossing voldoet aan de ISO 11898 standaard.

In het eerste geval wordt de transmissie van RS-485 transceiver naar CAN transceiver gecontroleerd. Als de RS-485 een dominante staat uitzendt op de bus, wordt actief een differentiële spanning gecreëerd tussen de klemmen A en B van de transceiver. Deze spanning is vastgelegd in de RS-485 standaard en bedraagt 1.5 V tot 2.9 V. De ISO 11898 standaard legt een bereik op van 1.5 V tot 3.0 V. Een dominant signaal, afkomstig van de RS-485 transceiver valt dus binnen de ISO 11898 normen.

Als de RS-485 transceiver een recessieve staat uitzendt, neemt deze een hoog impedante staat aan, waardoor het weerstandsnetwerk zorgt voor een instelling van het spanningsniveau op de CAN bus. Deze differentiële spanning kan berekend worden als:

$$\begin{aligned} V_{CANH} - V_{CANL} &= -V_{R_{term_1}} \\ &= -V_{cc} \cdot \frac{\left(\frac{1}{R_{term_1}} \cdot \frac{1}{120}\right)^{-1}}{\left(\frac{1}{R_{term_1}} \cdot \frac{1}{120}\right)^{-1} + R_1 + R_1} \\ &= -4.7 \cdot \frac{\left(\frac{1}{133} \cdot \frac{1}{120}\right)^{-1}}{\left(\frac{1}{133} \cdot \frac{1}{120}\right)^{-1} + 2 \cdot 549} = -258\ mV \end{aligned} \tag{3.9}$$

Ook deze spanning valt binnen de ISO 11898 norm, die stelt dat een recessieve uitgang op de CAN bus aangestuurd moet worden met een differentiële spanning van -0.5 V tot 0.05 V.

Vervolgens wordt onderzocht of ook in de omgekeerde richting voldaan wordt aan de ISO 11898 standaard. Er wordt dus nagegaan of de schakeling met de RS-485 transceiver ook correcte receiver eigenschappen heeft. Een dominant signaal aan een CAN receiver heeft volgens de ISO 11898

standaard een bereik van 0.9 V tot 5.0 V. De RS-485 transceiver beschouwt elk signaal groter dan 0.2 V als dominant. De werking voldoet hier dus niet volledig aan de ISO 11898 standaard, maar dit zal geen enkele functionele beperking met zich meebrengen, waardoor dit niet als een probleem beschouwd moet worden.

Als een CAN transceiver een recessief signaal op de bus zendt, zal dit gebeuren door een hoog impedante staat aan te nemen. In dit geval zal dus dezelfde situatie ontstaan als wanneer de RS-485 transceiver een recessief signaal uitzendt. Alle transceivers zullen zich in hoog impedante staat bevinden, waardoor het weerstandsnetwerk het differentiëel spanningsniveau op de bus bepaalt. Dit niveau zal weerom -258 mV bedragen, zoals berekend werd met formule 3.9 op de vorige pagina, wat zich binnen het bereik van -1.0 V tot 0.5 V bevindt, dat door de ISO 11898 standaard wordt opgelegd.

Men kan dus besluiten dat de RS-485 transceiver in de voorgestelde schakeling volledig functioneel is als CAN transceiver, hoewel de ISO 11898 norm op één punt niet gehaald werd. Voor de volledigheid worden de berekende spanningsniveaus samengevat in tabel 3.3.

Tabel 3.3: Differentiële spanning op de CAN bus met RS-485 transceiver, getoetst aan ISO 11898

	RS-485 to CAN (drive)			CAN to RS-485 (receive)		
	Min_{ISO}	$RS-485$	Max_{ISO}	Min_{ISO}	$RS-485$	Max_{ISO}
Dominant	1.5 V	1.5 V - 2.9 V	3 V	0.9 V	0.2 V	5.0 V
Recessief	-0.5 V	-0.258 V	0.05 V	-1.0 V	-0.258 V	0.5 V

3.2 Data link laag

Net als de fysieke laag, dient ook de data link laag zoveel mogelijk overeen te stemmen met de ISO 11898 standaard die werd uitgelegd in paragraaf 2.3 op pagina 8. Daarom wordt hiervan de data link laag volledig overgenomen en wordt aangeraden om read access te voorzien voor de error counters, zodat de kwaliteit van de CAN bus gecontroleerd kan worden.

3.3 Hogere protocols

Voor ruimtevaarttoepassingen heeft ESTEC (European Space Research and Technology Centre) het CANopen protocol geselecteerd als hoger protocol, een protocol dat besproken werd in paragraaf 2.4 op pagina 13. Dit protocol wordt ook voor commerciële toepassingen meest gebruikt en werd gekozen omdat het het meest geavanceerde protocol is dat bovendien zeer flexibel is en veel functies ondersteunt.

Opvallend is dat dit protocol voor ruimtevaarttoepassingen veelal geïntegreerd wordt als VHDL IP core, voorbeelden hiervan zijn de ESA CANopy core en de CCIPC CANopen core. Een softwarematige integratie van het CANopen protocol biedt vooral voordelen in commerciële toepassingen waar massaproductie en het gebruik van goedkope (micro)processors belangrijk zijn. Voor ruimtevaarttoepassingen is het aantal beschikbare (micro)processors echter beperkt, en is er interesse in eenvoudige CPU-loze ontwerpen, daarom kan een hardwarematige implementatie hier aangewezen zijn. Een implementatie van het CANopen protocol in de vorm van een VHDL IP core biedt in het kader van dit project echter geen grote voordelen aangezien hier een implementatie in

een boordcomputer met relatief krachtige CPU wordt uitgewerkt.

Net als bij de ISO 11898 standaard wordt ook bij het CANopen protocol getracht om geen modificaties aan de bestaande standaard toe te brengen. Voor ruimtevaarttoepassingen blijven evenwel nog enkele specifieke eisen gelden, wat resulteert in extra toegevoegde functies, die naast het bestaande CANopen protocol gebruikt kunnen worden.

Voor de ruimtevaart is ondersteuning voor service data objects (SDO) een verplichting, net als de network management objects (NMT) en een object dictionary. De network management objects bieden voorzieningen voor het starten, resetten en stoppen van een node, en voor het monitoren van de status. Bij ruimtevaarttoepassingen krijgen de NMT objecten echter ook een functie in het redundancy management. Het redundancy management wordt immers bestuurd door het hoger protocol, in dit geval dus CANopen. Één node zal de status van redundancy master (de NMT master) toegewezen krijgen, een master die heartbeats stuurt op de actieve bus. De nodes zullen deze periodieke heartbeats waarnemen op één van de twee bussen, de bus waarop het dataverkeer dan zal plaatsvinden. Als gedurende een bepaalde periode geen heartbeats ontvangen worden op de actieve bus, zal een buswissel plaatsvinden.

Een andere extra voorziening voor de ruimtevaart is de large data unit transfer (LDUT). Dit soort van objecten kan naast de SDO's bestaan in een CANopen netwerk en is vooral geschikt voor grote datatransfers aangezien de overhead beperkt wordt. Het gebruik hiervan is echter niet verplicht.

Verder zijn er nog aparte voorzieningen voor tijdsdistributie in ruimtetoestellen. Deze kan op twee manieren gebeuren die niet gecombineerd mogen worden. Ten eerste zijn er de spacecraft elapsed time objects (SCET), objecten die de zogenaamde CUC tijdsverdeling voorzien, zoals die ook gebeurt binnen de satellieten van QinetiQ Space. Deze CUC (CCSDS[4] Unsegmented Time Code) bevat een binaire teller tot 2^{32} seconden en subseconden tot 2^{-8}, 2^{-16} of 2^{-24}. In een later stadium zal hiervoor dus een voorziening geïntegreerd moeten worden in de software van het ADPMS. Dit valt echter buiten het opzet van dit project. Een ander soort objecten voor tijdsverdeling zijn de universal time coordinated (UTC) objecten. Deze bevatten een tijdscode, bestaande uit dagen, milliseconden en submilliseconden. Indien extra hoge nauwkeurigheid vereist is, kunnen deze tijdsobjecten gecombineerd worden met het high resolution time distribution protocol dat standaard aanwezig is in het CANopen protocol.

[4]Consultative Committee for Space Data Systems, een internationaal comité voor standaardisatie in de ruimtevaart.

4
CAN interface module

Voor dit project wordt onderzocht of een CAN bus geïntegreerd kan worden in het ADPMS (Advanced Data and Power Management System) van QinetiQ Space, een boordcomputer die opgebouwd is uit een aantal compactPCI kaarten die op een passieve backplane geplugd worden. De belangrijkste modules zijn de main processor module (MPM), telemetry and telecommand module (TTM), spacecraft interface module (SIM), data acquisition module (DAM), MIL1553b interface module (MIM) en de power supply module (PSM). Alle functies worden geïmplementeerd met behulp van ASICs en FPGA's. De Main Processor Module bevat geheugen, interfaces voor de controle van andere cPCI kaarten en een LEON-2FT (Sparc V8 compatible) CPU, hierop draait het RTEMS[1] operating system. Voor de communicatie van modules onderling wordt het PCI protocol gebruikt, voor on-chip communicatie van IP cores wordt de AMBA AHB[2] bus gebruikt. Voor de integratie van de CAN bus wordt gebruik gemaakt van de MIL1553b interface module. Aanpassingen in de hardware van deze kaart zullen de CAN bus functionaliteit voorzien. Er zullen met andere woorden een CAN controller en transceivers voorzien worden op het bestaande PCB ontwerp. Het is de bedoeling dat het uiteindelijke ontwerp van de CAN interface module verenigbaar is met de MIL1553b interface module, zodat één module ontstaat die twee communicatieprotocols ondersteunt. Dit valt echter buiten het opzet van dit project, maar bij het ontwerp zal getracht worden om zo dicht mogelijk bij de structuur van de bestaande MIL1553b interface module aan te leunen. Zo kan een latere fusie van beide ontwerpen eenvoudig verlopen en kunnen test benches en test scripts overgenomen worden.

4.1 CAN IP core

Zoals reeds werd aangehaald in paragraaf 3.1.3 op pagina 18, zal voor de integratie van een CAN controller een VHDL IP core vereist zijn. Het is echter belangrijk dat deze voldoet aan de eisen die gesteld worden door de ruimtevaartindustrie. Zo is ondersteuning voor een (cold) redundant CAN

[1]Real-Time Executive for Multiprocessor Systems, een open source real-time operating system voor embedded systems.

[2]Advanced Mictrocontroller Bus Architecture, Advanced High performance Bus

bus vereist, met het nodige mechanisme voor busselectie. Bovendien moet de IP core geïntegreerd kunnen worden in de bestaande structuur van het ADPMS en later zelfs in de MIL1553b interface module. Een interface met AMBA AHB bus is dus gewenst, evenals geheugentoegang via direct memory access (DMA).

In een studie van de bestaande VHDL IP cores voor CAN bus werden verschillende modellen gevonden, waarvan de meesten echter niet voldeden aan de vereisten. De ESA HurriCANe core bleek niet geschikt voor dit project aangezien voorzieningen voor een redundante CAN bus ontbreken, net als ondersteuning voor DMA. Een implementatie van deze IP core zou dus teveel extra werk met zich meebrengen. Vervolgens konden nog drie CAN IP cores uitgesloten worden aangezien zij geen voorzieningen hebben voor de on-chip AMBA AHB bus: de Xilinx CAN core, de Hitech Global CAN core en de Actel CAST CAN core.

Het bedrijf Gaisler Research, dat tevens de MIL1553b core levert aan QinetiQ Space, biedt drie CAN cores aan. Een eerste core, de CAN_OC core biedt geen ondersteuning voor DMA en bus redundancy. Bovendien voorziet deze core geen heruitzending als de arbitratie verloren werd op de CAN bus.

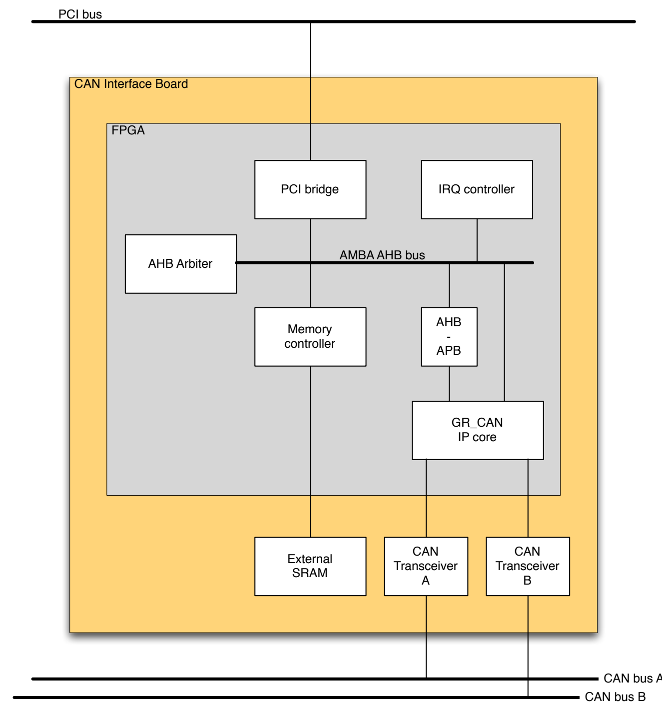
De tweede CAN core van Gaisler Research, GRCAN, is uitermate geschikt voor het project. Hij voorziet een on-chip interface met de AMBA AHB en APB[3] bus, ondersteunt een redundante CAN bus en maakt gebruik van DMA voor toegang tot circulaire buffers die zich buiten de IP core bevinden. Verder biedt de core ondersteuning voor zowel standaard als extended frames en bitrates van 20 kbps tot 1 Mbps, waardoor hij voldoet aan de draft standaard van ECSS, die kort werd toegelicht in hoofdstuk 3 op pagina 15. Een ander voordeel is dat Gaisler Research software drivers voor de GRCAN core aanbiedt voor het RTEMS operating system, wat later de softwarematige implementatie (die evenwel buiten het kader van dit project valt) kan vereenvoudigen. Het enige nadeel van deze CAN core is het gebruik van de AMBA APB bus voor het lezen en schrijven van configuratie-, controle- en statusregisters. De APB bus wordt normaal gezien niet gebruikt als on-chip bus bij QinetiQ Space, daarom zal een conversie van APB naar AHB bus nodig zijn.

Tenslotte biedt Gaisler Research ook nog de GRHCAN IP core aan, een CAN IP core die dezelfde voorzieningen biedt als de GRCAN core, maar de ESA HurriCANe core bevat. Deze core is enkel beschikbaar voor ESA projecten. Uiteindelijk werd besloten om de GRCAN IP core te gebruiken voor integratie in de CAN interface module.

4.2 Algemene structuur

In figuur 4.1 op de volgende pagina wordt de algemene structuur van de CAN interface module schematisch voorgesteld. Hierop is duidelijk dat de cPCI module gebruik maakt van een FPGA als CAN controller, en twee CAN transceivers voor de dubbele CAN bus. Verder bemerkt men een extern SRAM geheugen op de CAN interface module, dit geheugen werd overgenomen van de MIL1553b interface module. Daar wordt het gebruikt voor debug doeleinden zoals het opslaan van ontvangen data. Het geheugen zal dan ook enkel bestukt worden op prototypes en niet op de uiteindelijke “flight module”. Hoewel het geheugen niet nodig is op het CAN interface board, wordt toch een memory controller geïntegreerd in de FPGA om zoveel mogelijk overeenkomsten met het MIL1553b interface board te bekomen.

[3] Advanced Peripheral Bus



Figuur 4.1: Schematische voorstelling van de CAN interface module

4.2.1 CAN transceivers

De CAN transceivers die op het CAN interface board geïntegreerd worden, worden gebouwd met stralingsbestendige RS-485 transceivers, zoals beschreven werd in paragraaf 3.1.4 op pagina 19. De voorzieningen voor deze hardware werden al vastgelegd in het PCB ontwerp van de MIL1553b interface module.

4.2.2 CAN interface FPGA

Het belangrijkste onderdeel van de CAN interface module is zonder twijfel de FPGA. Deze bevat verschillende VHDL IP cores en een on-chip AMBA AHB bus om de nodige functionaliteit te voorzien. Een gedetailleerd blokdiagram van de FPGA met de geïntegreerde IP cores is te zien in figuur 4.3 op pagina 32. Het ontwerp werd gebaseerd op het ontwerp van de MIL1553b interface module, waarin de MIL1553b IP core vervangen werd door de GRCAN IP core. Aangezien de IP cores niet zelf geschreven werden en omwille van intellectuele eigendom, zal enkel een korte beschrijving gegeven worden van hun functies, samen met de eigenschappen die belangrijk zijn in het verdere verloop van dit project.

VHDL IP cores

PCI2AHB bridge De PCI2AHB bridge (versie 5.8) van QinetiQ Space vormt een verbinding tussen de on-chip AMBA AHB bus en de externe PCI bus. De core gedraagt zich als PCI master en AHB slave, of PCI target en AHB master. In PCI target mode worden alle binnenkomende

PCI read/write cycles omgezet naar AHB master read/write cycles en in PCI master mode worden AHB slave read/write cycles omgezet naar PCI read/write cycles.

Verder biedt de core ook nog de mogelijkheid om interrupts van een externe interrupt controller door te geven op de PCI bus.

AHBARB De AHBARB core (versie 1.3) van QinetiQ Space wordt gebruikt als AHB arbiter. Het AHB protocol[17] legt de rol van arbiter vast als een eenheid die ervoor zorgt dat slechts één master tegelijkertijd toegang heeft tot de bus. Indien verschillende masters toegang tot de bus vragen (request), zal de arbiter bepalen welke de hoogste prioriteit heeft en bijgevolg toegang krijgt (grant) tot de bus.

GRCAN De GRCAN core van Gaisler Research (versie 13) werd reeds toegelicht in paragraaf 4.1 op pagina 26. De core wordt aangesloten als master op de AHB bus voor het lezen of schrijven van CAN berichten in een geheugen, en als slave op een APB bus voor configuratie, controle en status handling van de core. Aangezien hier echter geen APB bus gebruikt wordt, zal een aparte IP core gebruikt worden voor de omzetting van AHB naar APB en omgekeerd.

De CAN berichten worden opgeslagen in circulaire buffers (een send en receive buffer) die zich in een extern geheugen (op het MPM[4] board) bevinden en waarvan de grootte vrij gekozen kan worden tot 1 MB. Een read pointer wijst naar het laatst gelezen CAN bericht, terwijl een write pointer wijst naar het laatst geschreven bericht in de buffer. Een verschil tussen beide pointers wijst er dus op dat berichten verzonden moeten worden of dat berichten ontvangen werden.

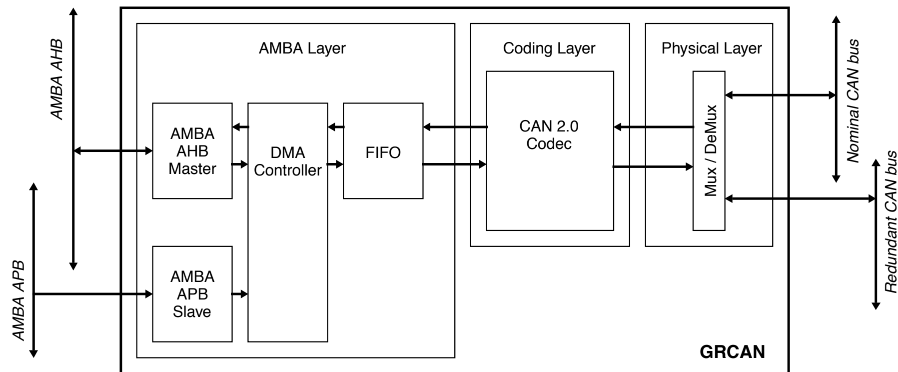
De core voorziet twee CAN bussen met een redundancy (de)multiplexer voor selective bus access. Aangezien geen hoger protocol geïntegreerd werd in de core, dient de busselectie te gebeuren vanuit de software via een registerbit.

Verder bevat de core verschillende indicators die de toestand van de core weergeven, zoals een transmitter active indicator, bus-off condition indicator, error-passive condition indicator[5] en over-run indicator[6]. Bovendien zijn de CAN error counters toegankelijk via een register, waardoor de core ook voldoet aan de ECSS aanbevelingen voor de CAN datalink laag, zoals beschreven in 3.2 op pagina 24. Verder is er een interrupt register waarin alle soorten errors (vb. AHB errors, transmit en receive errors) en gebeurtenissen (vb. transmission/reception successful) worden aangekondigd.

[4]Main Processor Module

[5]Voorzieningen voor foutbeperking, zoals beschreven in paragraaf 2.3.2 op pagina 9

[6]Over-run: wanneer meer berichten ontvangen worden dan de receive buffer aankan



Figuur 4.2: Blokschema van de GRCAN IP core van Gaisler Research

AHB2APB_REGS De AHB2APB_REGS core (versie 1.1) van QinetiQ Space zorgt voor de vertaling van berichten tussen de AHB en APB bus van de CAN core. Deze core gedraagt zich als slave op de AHB bus en als master op de APB bus.

AHB2IRQC De AHB2IRQC core zorgt ervoor dat alle interrupts van alle IP cores gecentraliseerd worden in één interruptlijn, die dan doorgegeven wordt aan de PCI2AHB core. Op deze manier kunnen alle gebeurtenissen op de CAN interface module opgevolgd worden door de Main Processor Module, via de PCI bus. In dit geval werden slechts drie interrupts gebruikt: één interruptsignaal van de GRCAN core en twee van de SRAM Memory Controller.

AHB2SMC De AHB2SMC core (versie 1.7) van QinetiQ Space verzorgt een interface tussen de AMBA AHB bus en een extern SRAM geheugen dat zich evenwel op dezelfde PCB, dicht bij de FPGA bevindt. Deze core wordt vooral geïmplementeerd om overeenkomsten met de MIL1553b interface module te behouden. Eventueel kan het geheugen in een later stadium wel nog gebruikt worden voor testdoeleinden. De core gedraagt zich als twee slaves op de AHB bus, één voor registertoegang tot de core en één voor de datatransfers. De core bevat verder ook voorzieningen voor foutcontrole en correctie. Deze functies worden echter niet benut voor dit project, waardoor een bespreking dus ook niet relevant is.

Communicatie protocols

In de CAN interface module worden verschillende protocols geïmplementeerd.

Ten eerste wordt een PCI interface voorzien die zorgt voor de interconnectie van de CAN interface module met de Main Processor Module. Deze interface werkt met een 33Mhz kloksignaal. Deze frequentie wordt ook aangehouden voor de on-chip AMBA AHB en APB bus. Deze bussen kan men echter nog verder opsplitsen. De VHDL IP cores van QinetiQ Space maken immers gebruik van de ESA AHB en APB VHDL packages. De IP core van Gaisler Research daarentegen, maakt gebruik van een eigen AHB en APB VHDL package. Deze twee AHB bussen zijn niet 100 % compatibel omdat verschillende testsignalen van de ESA AHB bus ontbreken bij de Gaisler Research AHB bus, en omdat de overige signalen anders benoemd werden. Als de functies van de aanwezige signalen bestudeerd worden, kan men beide AHB bussen toch correct verbinden, zodat de GRCAN core toch volledig functioneel is. Verder dient men er rekening mee te houden dat

elk AHB bus systeem een default master moet bevatten. Deze default master krijgt toegang tot de AHB bus wanneer de andere AHB masters geen AHB toegang vragen aan de arbiter. Voor de on-chip AHB bus van de CAN interface FPGA werd de PCI2AHB core gekozen als default master, een keuze die ervoor zorgt dat de CAN interface module ten allen tijde toegankelijk is via de PCI bus, ook indien andere IP cores in de FPGA incorrect zouden werken. Het is ook belangrijk te weten dat alle IP cores werken met een big endian systeem, waardoor de PCI bus en ook de AHB bus volledig met een big endian systeem gebruikt kunnen worden.

De GRCAN core voorziet een dubbele CAN interface die een drive signaal vanuit de GRCAN core naar de transceivers stuurt (*cano[1::0]* in figuur 4.3 op de volgende pagina). Dit signaal wordt echter eerst nog geïnverteerd aangezien hier gewerkt wordt met een RS-485 transceiver, zoals beschreven werd in paragraaf 3.1.4 op pagina 19. Het receive signaal dat van de transceiver naar de GRCAN core wordt gestuurd, dient dan weer niet geïnverteerd te worden. Verder dient men nog op te merken dat een transceiver enable signaal voorzien is in de GRCAN core, maar dit wordt niet naar buiten gebracht aangezien de huidige implementatie van CAN transceivers niet enabled of disabled kunnen worden. In tegenstelling tot andere IP cores die een communicatieprotocol integreren (zoals bijvoorbeeld de MIL1553b core van Gaisler Research), heeft de GRCAN core geen extern kloksignaal nodig. Dit wordt immers afgeleid uit het kloksignaal van de AHB bus, dat op zijn beurt werd afgeleid uit het PCI kloksignaal. Om de gewenste bitsnelheid te bekomen op de CAN bus, worden via software een prescaler en twee fasesegmenten ingesteld in een configuratieregister van de GRCAN core. Deze gegevens worden dan gebruikt om het AHB kloksignaal verder te delen tot men bijvoorbeeld 1 Mbps bekomt.

Verder bevat het ontwerp een SRAM interface voor het aansluiten van één of twee SRAM modules. Ook werden in het ontwerp al poorten voorzien voor de MIL1553b bus en de SpaceWire[7] bus. Deze poorten zijn echter nog niet aangesloten en werden enkel voorzien om een toekomstige fusie van verschillende ontwerpen te vereenvoudigen.

Extra logica

Naast de interconnectie van de IP cores werd ook nog een beperkte hoeveelheid logica geïmplementeerd. Zo werd een asynchroon power-on-reset signaal (*por_n*) dat afkomstig is van de Main Processor Module gecombineerd met een asynchroon PCI reset signaal (*pci_rstn*). Dit werd daarna gesynchroniseerd met het kloksignaal (*sys_clk_i*), waarna het als global buffer (*sys_rst_n_i*) gedeclareerd werd in de FPGA. Op eenzelfde manier werd ook het PCI kloksignaal (*pci_clk*) als global buffer (*sys_clk_i*) gedeclareerd om timing delays zoveel mogelijk te beperken.

Ook werd een teller ingebouwd die pulsen van 1 kHz (*tick_1ms_i*) genereert voor de scrubber van de AHB2SMC core, een functie voor het periodiek opsporen van fouten.

[7] SpaceWire is een door ESA ontwikkelde standaard voor datanetwerken in de ruimte. Hiermee kunnen bitrates tot 400 Mbps behaald worden, 400 keer meer dan MIL1553b of CAN bus.

Figuur 4.3: Gedetailleerd blokschema van de FPGA

5
VHDL test bench

Bij het ontwerpen van componenten voor de ruimtevaart gaan de grootste inspanningen zonder twijfel naar tests. De exacte procedure voor het ontwerpen en testen van FPGA's wordt beschreven in een ECSS document[18]. Deze procedure wordt in grote lijnen gevolgd voor het ontwerpen en testen van de CAN interface module, maar op sommige punten is het naleven van deze richtlijnen onmogelijk. In normale omstandigheden wordt immers met een heel ingenieursteam gedurende maanden gewerkt aan de ontwikkeling van een cPCI module. Het testen kan dan door aparte test ingenieurs gebeuren, zoals de ECSS procedure voorschrijft. Hier zal vooral het algemene proces van de productontwikkeling gevolgd worden, zoals het beschreven wordt in de ECSS procedure. Deze geeft aan dat elke IP core die geïntegreerd werd in een ontwerp, op zichzelf getest moet worden in een aparte VHDL test bench. Deze test bench dient volledige geautomatiseerd te zijn, een menselijke interpretatie of interventie is dus niet toegestaan. Bij het uitvoeren van de test bench dient een automatisch testrapport gegenereerd te worden, waarin aangegeven wordt of de uitgevoerde tests goed doorstaan waren. Aangezien hier gewerkt werd met IP cores van QinetiQ Space die reeds geïntegreerd werden in andere projecten, werd deze stap vroeger al uitgevoerd. Ook de GRCAN IP core van Gaisler werd met een aparte test bench meegeleverd en zal dus niet meer afzonderlijk getest moeten worden.

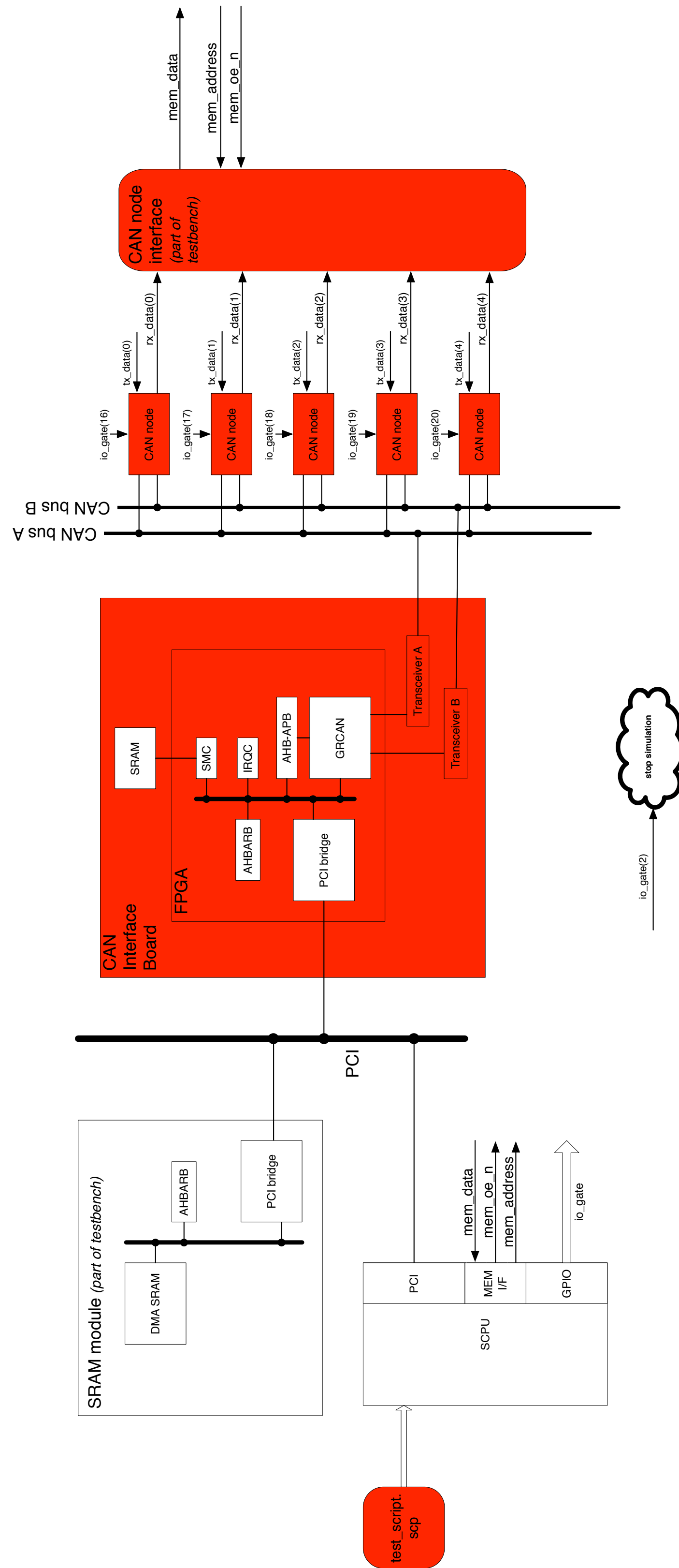
Vervolgens is het belangrijk dat de volledige functionaliteit van het ontwerp getest wordt in een algemene VHDL test bench. Deze test bench mag enkel componenten bevatten die reeds eerder geverifieerd werden. Nieuwe code mag dus niet toegevoegd worden in de algemene test bench voordat ze eerst getest werd in een afzonderlijke test bench.

In dit hoofdstuk wordt de opbouw van de algemene VHDL test bench besproken. Deze test bench wordt dan gesimuleerd met behulp van het softwarepakket ModelSim, waarvoor compilatie- en simulatiescripts geschreven worden om het geheel te automatiseren en om herhaalbaarheid van de tests te waarborgen.

5.1 Algemene structuur

Om de volledige werking van de CAN interface FPGA na te gaan, wordt hierrond een volledige VHDL test bench gebouwd. Deze test bench moet toelaten om elke functie van het FPGA ontwerp te simuleren in een omgeving die de uiteindelijke omgeving waarin de FPGA geïmplementeerd zal worden, voldoende benadert. Bovendien moet elk signaal zowel binnen als buiten de FPGA gemonitord kunnen worden tijdens de simulaties. Concreet wil dit zeggen dat het FPGA ontwerp in de VHDL test bench geïmplementeerd wordt in een VHDL model van de volledige CAN interface module. De CAN interface module wordt dan op haar beurt aangesloten op een PCI bus en op twee CAN netwerken. De CAN netwerken bevatten testnodes die CAN berichten kunnen zenden en ontvangen. Deze nodes worden volledig zelf gemodelleerd en worden uitgebreid besproken in paragraaf 5.3 op pagina 39. Tenslotte worden op de PCI bus simulatiemodellen van een CPU en een geheugenmodule voor DMA aangesloten.

Het ontwerp van deze test bench wordt gedeeltelijk gebaseerd op een test bench voor de MIL1553b interface FPGA. In figuur 5.1 op de pagina hierna wordt de volledige structuur van de VHDL test bench weergegeven. De onderdelen die zelf geschreven werden, worden in het rood weergegeven. De witte onderdelen zijn VHDL IP cores of modellen die uit de MIL1553b test bench overgenomen kon worden.



Figuur 5.1: Schematische voorstelling van de CAN interface FPGA test bench (Rood = zelf geschreven code, wit = overgenomen code)

5.1.1 SRAM module

Aangezien geen VHDL model ter beschikking is van een SRAM geheugen dat rechtstreeks op de PCI bus aangesloten kan worden, wordt hier gewerkt met een SRAM model met AHB interface. In de test bench wordt dus een module gecreëerd die bestaat uit het SRAM model, een AMBA AHB bus met AHB arbiter en een PCI2AHB bridge. Voor de AHB arbiter en de PCI2AHB bridge worden dezelfde IP cores van QinetiQ Space gebruikt als in de CAN interface FPGA, zoals beschreven werd in paragraaf 4.2.2 op pagina 28. De gebruikte AHB bus is dus ook hier gebaseerd op de ESA AHB VHDL package. Er zijn twee masters en twee slaves op de AHB bus, telkens één voor het SRAM model en één voor de PCI2AHB core. Ook hier is de PCI2AHB core de default master op de AHB bus.

Het gebruikte SRAM model is het SRAM simulation model (versie 2.2) van QinetiQ Space. Dit model kan enkel gebruikt worden in simulaties en is dus niet bruikbaar voor integratie in een FPGA. Het model gedraagt zich als SRAM geheugen op de AHB bus, maar biedt ook specifieke voorzieningen binnen de test bench. Zo kan de geheugeninhoud hexadecimaal ingelezen of weggeschreven worden naar een tekstbestand. Ook andere instellingen, waarop hier niet verder wordt ingegaan, zijn instelbaar met behulp van generics.

DMA read/write access

Het SRAM model laat toe dat via DMA toegang verkregen wordt tot de geheugeninhoud. Als een DMA read of write access plaatsvindt vanuit de GRCAN core in de CAN interface FPGA, zal de GRCAN AHB master dus een aanvraag voor de transfer sturen op de on-chip AHB bus. Indien de transfer (request) toegelaten wordt (grant) door de AHB arbiter van de CAN interface FPGA, wordt een transfer opgezet tussen de GRCAN AHB master en de PCI2AHB bridge AHB slave. De transfer wordt dan voortgezet op de PCI bus naar de PCI2AHB bridge van de SRAM module. Indien een read access plaatsvindt, zal de opgevraagde data niet direct beschikbaar zijn in de PCI2AHB bridge van de CAN interface module. Er zullen dan wait states geïntroduceerd worden op de on-chip AHB bus. Wanneer de PCI2AHB bridge van de SRAM module een read of write access ontvangt op de PCI bus, zal deze voortgezet worden door de AHB master van de PCI2AHB bridge. Nadat de transfer toegestaan werd door de AHB arbiter van de SRAM module, kan verbinding gemaakt worden met de AHB slave van het SRAM simulatiemodel. De gewenste data kan dan opgehaald of weggeschreven worden. Indien data opgehaald wordt, zal deze dus terecht komen in de PCI2AHB bridge van de SRAM module. Via de PCI bus komt de data dan ook terecht in de PCI2AHB bridge van de CAN interface module, die dan stopt met het uitzenden van wait states op de AHB bus en de gevraagde data via de AHB bus doorstuurt naar de GRCAN core.

Voor een DMA transfer zal de data dus eerst over een AHB bus verlopen, daarna over een PCI bus en uiteindelijk weer over een AHB bus.

5.1.2 Simulatie CPU

Om de CAN interface FPGA volledig te kunnen testen, zal data geschreven en gelezen moeten worden in registers en geheugens. Bovendien moet deze data geïnterpreteerd worden om vervolgens de gepaste acties te ondernemen. Zo kan de test bench volledig geautomatiseerd worden. Daarom wordt een zogenaamde simulatie CPU (SCPU) geïntegreerd, een VHDL model van een CPU dat enkel bestemd is voor simulaties en dus niet geschikt is voor synthese. Het model dat

hier gebruikt werd is de s_processor (versie 6.1) van QinetiQ Space. De SCPU ondersteunt verschillende interfaces voor communicatie, hier zullen enkel de gebruikte functies kort besproken worden. De instructies voor de SCPU worden opgehaald uit een bestand dat assembly code bevat, een zogenaamd testscript. Op de uitvoering van de testscripts wordt dieper ingegaan in hoofdstuk 6 op pagina 53. De code in de testscripts wordt sequentieel uitgevoerd, maar kan jumps bevatten. Net als een normale CPU, heeft deze SCPU een volledige instructieset aan boord met transfer instructies, wiskundige en logische instructies, I/O instructies, spronginstructies en instructies voor de configuratie van de processor.

In eerste instantie wordt de SCPU verbonden met de PCI bus. Hierdoor kan data gelezen en geschreven worden in de registers van de IP cores in de CAN interface FPGA, of in de SRAM module.

Ook de memory bus van de SCPU wordt benut. Deze is afgeleid van de LEON processor[1] memory bus, die op zijn beurt gebaseerd is op een 32 bit Intel memory bus. De bus bestaat uit een aantal uitgaande adreslijnen, die aangeven van welk adres data gelezen of geschreven moet worden. Vervolgens zijn er 32 datalijnen, die zowel als ingang als uitgang kunnen dienen, afhankelijk van de aard van de geheugentoegang (lezen of schrijven). Tenslotte zijn er nog enkele controlesignalen die het lezen en schrijven van geheugens controleren. In de huidige test bench wordt enkel de leesfunctionaliteit benut, deze wordt verder toegelicht in paragraaf 5.3.6 op pagina 52.

Verder worden ook de I/O functies van de SCPU benut. Het aantal I/O poorten werd in deze test bench vastgelegd op 32. Deze poorten kunnen gebruikt worden om vanuit de software signalen hoog of laag te zetten. Zo wordt het signaal van io_gate(2) gebruikt om de uitvoering van de test bench te stoppen. Op deze manier kan vanuit de software (het testscript) de test bench gestopt worden als alle instructies uitgevoerd zijn. Ook worden vijf aparte I/O signalen gebruikt om de CAN nodes in de test bench afzonderlijk te enablen. Hierop wordt dieper ingegaan in paragraaf 5.3 op pagina 39.

5.2 CAN interface board

Om de CAN interface FPGA te testen in de test bench, zal het ontwerp geïntegreerd moeten worden in een model van het CAN interface board. Dit is een model dat alle hardware van de CAN interface module beschrijft. Het gaat dus om de PCI interface, de dubbele CAN interface met de transceivers, het SRAM geheugen en uiteraard het FPGA ontwerp. Ook alle ongebruikte interfaces van de FPGA, zoals de MIL1553b interface en de SpaceWire interface, zullen aangesloten worden op het CAN interface board. Deze signalen zullen echter niet verder gebruikt worden.

PCI interface

Aangezien het CAN interface board voorzien is van een PCI interface, kan deze direct verbonden worden met de PCI interface van de FPGA. Hier zijn immers geen tussenliggende componenten zoals transceivers nodig, waardoor de FPGA dus rechtstreeks aangesloten kan worden op de PCI bus van de test bench.

[1] De LEON processor wordt in het ADPMS gebruikt op de Main Processor Module

SRAM geheugen

Voor het SRAM geheugen op het CAN interface board kan niet het SRAM simulation model gebruikt worden dat beschreven werd in paragraaf 5.1.1 op pagina 36. Dat model wordt immers aangesloten op een AHB interface, terwijl het geheugen hier aangesloten moet worden op de SRAM interface van de FPGA. Daarom wordt hier gebruik gemaakt van het k6r4016v1c geheugenmodel (versie 1.5), een open source geheugenmodel van de University of Hamburg dat aangepast werd door QinetiQ Space. Net als het SRAM simulation model van QinetiQ Space, kan ook dit model geconfigureerd worden met behulp van VHDL generics en kan het een geheugeninhoud ophalen of wegschrijven naar een hexadecimaal tekstbestand. Hierop zal niet verder ingegaan worden, aangezien dit SRAM geheugen verder niet gebruikt zal worden in de tests. Het werd slechts toegevoegd om de test bench te vervolledigen en om extra functionaliteit toe te voegen voor eventuele tests in een later stadium.

CAN transceivers

Tenslotte dient men ook de CAN transceivers te modelleren in VHDL, zodat de drive en receive signalen van de CAN interface FPGA omgezet worden naar geldige CAN bus signalen in de test bench. De geïmplementeerde CAN transceivers werden gebaseerd op RS-485 transceivers en hebben daarom een werking die licht afwijkt van een normale CAN transceiver werking. Het drive signaal van de geïmplementeerde transceivers werkt immers invers, zoals beschreven werd in paragraaf 3.1.4 op pagina 19. Bovendien beschikken deze transceivers niet over een enable signaal.

Als men de werking van een CAN bus analyseert, ontdekt men al gauw dat een beschrijving in digitale logica een probleem oplevert. In het geval van een dominant signaal zal de bus immers actief aangestuurd worden, waardoor alle recessieve signalen overschreven worden. Een recessief signaal daarentegen, zal wel zorgen voor een signaal op de CAN bus, maar dit kan overschreven worden door dominante signalen. Het is dus niet mogelijk om dit probleem dat veroorzaakt wordt door de analoge werking CAN transceivers, op te lossen met enkel hoge en lage digitale signalen. Het probleem wordt duidelijk gemaakt in tabel 5.1.

Tabel 5.1: Resultaat als verschillende nodes verschillende signalen zenden op de CAN bus

CAN node		andere CAN nodes		CAN bus resultaat		
CANH	CANL	CANH	CANL	CANH	CANL	Bus State
Z	Z	Z	Z	0	1	Recessief
Z	Z	1	0	1	0	Dominant
1	0	Z	Z	1	0	Dominant
1	0	1	0	1	0	Dominant

Het probleem kan echter opgelost worden aangezien VHDL ondersteuning biedt voor zogenaamde sterke en zwakke signalen. Zo zijn sterke signalen (‘0’ en ‘1’) niet overschrijfbaar, terwijl zwakke signalen (‘L’ en ‘H’) dat wel zijn. Ter verduidelijking: als op een bus tegelijkertijd een ‘H’ en ‘0’ signaal gestuurd wordt door twee verschillende componenten, zal het resultaat ‘0’ zijn. Op dezelfde manier levert een ‘L’ en een ‘1’ uiteindelijk ‘1’ op. Als men deze functionaliteit benut, wordt tabel 5.1 omgevormd tot tabel 5.2 op de volgende pagina

Tabel 5.2: Resultaat als verschillende nodes verschillende signalen zenden op de CAN bus: oplossing met zwakke signalen

CAN node		andere CAN nodes		CAN bus resultaat		
CANH	CANL	CANH	CANL	CANH	CANL	Bus State
L	H	L	H	L	H	Recessief
L	H	1	0	1	0	Dominant
1	0	L	H	1	0	Dominant
1	0	1	0	1	0	Dominant

Op deze manier kan de werking van de CAN transceivers vrij eenvoudig beschreven worden in VHDL. Het resultaat wordt voorgesteld in tabel 5.3. Hierin wordt ook de inverse werking van het drive signaal, veroorzaakt door het gebruik van een RS-485 transceiver, in rekening gebracht.

Tabel 5.3: Werking van het CAN transceiver VHDL model

Transmit				Receive			
CAN drive	CANH	CANL	Bus State	CAN receive	CANH	CANL	Bus State
0	L	H	recessief	0	1	0	dominant
1	1	0	dominant	1	L	H	recessief

De uiteindelijke VHDL code voor de modellering van de twee CAN transceivers wordt weergegeven op deze pagina. Het *can_x_d_i* signaal is het drive signaal, afkomstig van de FPGA. Het *can_x_r_b_n_i* signaal is het receive signaal van de transceiver dat naar de FPGA gestuurd wordt. De *can_x_h* en *can_x_l* signalen worden rechtstreeks aan de CAN bus gekoppeld.

De ECSS aanbevelingen voor het ontwikkelen en testen van ASICs en FPGA's geven aan dat elk onderdeel van de test bench eerst in een afzonderlijke test bench getest moet worden. In dit geval is dit echter niet nodig aangezien het maar om enkele regels code gaat.

```
can_a_h       <= '1' WHEN (can_a_d_i = '1') ELSE 'L';
can_a_l       <= '0' WHEN (can_a_d_i = '1') ELSE 'H';
can_a_r_b_n_i <= '0' WHEN ((can_a_h = '1') AND (can_a_l = '0')) ELSE '1';

can_b_h       <= '1' WHEN (can_b_d_i = '1') ELSE 'L';
can_b_l       <= '0' WHEN (can_b_d_i = '1') ELSE 'H';
can_b_r_b_n_i <= '0' WHEN ((can_b_h = '1') AND (can_b_l = '0')) ELSE '1';
```

5.3 CAN testnodes

Een laatste belangrijk onderdeel van de test bench is het dubbele CAN bus netwerk. Dit netwerk wordt gebruikt om een transmissie op te starten met de GRCAN core, of om een reception te testen met de GRCAN core. Om deze tests uit te voeren, moeten andere CAN nodes op het netwerk geplaatst worden, de zogenaamde testnodes. De simulatiemodellen van deze nodes zijn echter nergens ter beschikking en zullen bijgevolg zelf geschreven en getest worden alvorens ze te integreren in de test bench.

5.3.1 Functionaliteit

De testnodes bieden de mogelijkheid om volledige CAN transfers uit te voeren in de simulatie. De modellen zijn dan ook enkel bruikbaar voor simulaties en niet voor synthese. De transfers kunnen uitgevoerd worden op twee verschillende CAN bussen, waartussen men kan kiezen met behulp van een selectiesignaal. Indien een CAN frame ontvangen wordt, zal vanzelf gedetecteerd worden op welke bus dit gebeurt. Deze bus zal dan actief worden.

De functies die geïmplementeerd werden, voldoen niet allemaal aan de ISO 11898 standaard. Dit is echter ook niet de bedoeling aangezien de nodes enkel voor testdoeleinden in simulaties gebruikt zullen worden. In de nodes werd een fysieke laag en een data link laag geïmplementeerd. Een hoger protocol zoals het CANOpen protocol is hier niet nodig aangezien de GRCAN core hier zelf geen ondersteuning voor biedt.

Fysieke laag

Aangezien het model enkel gebruikt zal worden in logische simulaties, zullen verschillende eisen van een reëel CAN systeem wegvallen. Zo dient men geen rekening te houden met de lengte van het netwerk, connectors, de belasting van het netwerk, lijnterminatie of attenuatie. Ook de propagation delay kan in logische simulaties verwaarloosd worden. Daarom kan gewerkt worden met sampling tijdstippen op 50 % van de bittijd (500 µs na een flank bij 1 Mbps transmissie).

Ook met signaalniveaus hoeft men weinig rekening te houden in de simulaties, het volstaat te werken met sterke en zwakke signalen (‘0’, ‘1’, ‘L’, ‘H’), zoals beschreven werd in paragraaf 5.2 op pagina 38. Op deze manier zal de arbitratie automatisch gebeuren, recessieve signalen (‘L’, ‘H’) kunnen immers overschreven worden door dominante signalen (‘0’, ‘1’). De nodes omvatten dus ook een transceiver gedeelte dat de bitstream omzet naar geldige CAN bus signalen. Indien een node de arbitratie verliest, zal ze in back-off status gaan. Dit wil zeggen dat ze enkel nog een recessief signaal zal uitzenden en dat alle reeds verzonden data onthouden wordt en aangevuld wordt met de nieuwe ontvangen data.

De CAN testnodes zijn ontworpen om data te verzenden en ontvangen aan een bitrate van 1 Mbps, andere bitrates worden dus niet ondersteund. Het veranderen of zelfs instelbaar maken van de bitrate is vrij eenvoudig, maar hier is voorlopig geen noodzaak aan in de test bench.

Data link laag

Voor de tests van de GRCAN core in de test bench is een integratie van een data link laag in de testnodes niet strikt noodzakelijk. Als de fysieke laag voorzien wordt, kan men in principe handmatig CAN frames samenstellen, CRC berekeningen uitvoeren en errors opzoeken. Deze aanpak zorgt echter voor een tijdrovende testprocedure die bovendien geen absolute zekerheid biedt aangezien hier tussenkomst van de gebruiker nodig is. Deze aanpak is bovendien niet toegestaan volgens de ECSS richtlijnen voor het ontwerpen en testen van ASICs en FPGA’s. Aangezien de test bench volledig geautomatiseerd moet zijn en herhaalbaarheid van de tests gewaarborgd moet worden, zal dus toch een data link laag geïmplementeerd worden.

Codering De geïmplementeerde data link laag voorziet de codering van CAN frames, rekening houdend met de regels voor bit stuffing in data en remote transfer request (RTR) frames. De testnodes zullen er dus voor zorgen dat na de verzending van 5 identieke bits een inverse bit

wordt doorgestuurd. Deze inverse bit bevat dus geen informatie en dient enkel voor synchronisatie doeleinden, zoals beschreven werd in paragraaf 2.3 op pagina 8. Men moet er rekening mee houden dat een bit stuffing bit ook kan meetellen als één van de vijf identieke bits. Uiteraard kan een testnode ook bit stuffing bits detecteren bij ontvangst, wat noodzakelijk is aangezien men voor de berekening van de CRC code geen bit stuffing bits in rekening mag brengen.

Foutafhandeling De testnodes voorzien ook een zekere vorm van foutafhandeling, die enigszins afwijkt van de foutafhandeling die wordt toegepast in normale CAN controllers. In de testnodes werd de error detectie zeer ver doorgevoerd. Zo wordt een ontvangen frame gecontroleerd op bit errors, er wordt dus gecontroleerd of een verzonden bit effectief op de bus verschijnt. Deze controle wordt uiteraard niet uitgevoerd tijdens arbitratie of acknowledgement, momenten waarop het de bedoeling is dat uitgezonden recessieve bits overschreven worden met dominante bits door een andere node. Verder wordt er gecontroleerd op form errors, een foutsoort die optreedt wanneer vastgelegde bits uit het CAN protocol (zoals bijvoorbeeld reserved bits) verkeerdelijk op de bus komen. Ook stuffing errors worden gedetecteerd, errors die voorkomen als meer dan 5 identieke opeenvolgende bits voorkomen binnen een CAN frame. Uiteraard worden deze errors niet gedetecteerd in interframe space (IFS), die in theorie oneindig veel recessieve bits kan bevatten. Een ander soort errors dat gedetecteerd wordt, zijn CRC errors die gegenereerd worden als een berekende CRC code niet overeen komt met de CRC code uit het CAN frame. Uiteraard worden ook acknowledgement errors gedetecteerd, errors die aanduiden dat geen acknowledgement bit ontvangen werd op het verwachte tijdstip binnen het CAN frame. Tenslotte werden in de testnodes extra foutmeldingen geïmplementeerd die duiden op een foutieve invoer van data. Op deze manier worden fouten gesignaleerd nog voor men aan de transmissie begint.

In tegenstelling tot klassieke CAN controllers, werden hier geen transmit en receive error counters geïntegreerd. Deze tellers worden immers geïmplementeerd voor foutbeperking. In het geval van de CAN testnodes is het vrij nutteloos om fouten te tellen, om zo eventueel een 'error passive' of 'bus off' staat aan te nemen. Het is immers de bedoeling dat geen enkele fout gedetecteerd wordt.

In de testnodes werd verder geen ondersteuning geïntegreerd voor error en overload frames. Deze frames worden normaal gezien gebruikt voor foutsignalisatie, maar in het geval van de testnodes, kunnen gedetecteerde errors direct gedetailleerd gesignaleerd worden in de test bench. Men dient er evenwel op te letten dat enkel de eerste error melding relevant is om de oorzaak van een fout te achterhalen. Indien één error gedetecteerd werd, zullen mogelijks andere error meldingen volgen aangezien de vorm van het CAN frame niet meer voldoet aan de verwachtingen. Als bijvoorbeeld één bit ontbreekt, kan dit direct aangeduid worden met een form error. Hierop zullen echter vrijwel zeker stuffing errors, CRC errors en acknowledgement errors gedetecteerd worden. Enkel de form error is dan relevant om de oorzaak van de error te bepalen. Men kan ook nog opmerken dat de methode van error signalisatie zeer gedetailleerd gebeurt. Zo wordt telkens de foutsoort vermeld, samen met de plaats waar de fout optreedt. De geïmplementeerde foutsignalisaties worden weergegeven in tabel 5.4 op de pagina hierna.

Tabel 5.4: Errordetectie in de CAN testnodes

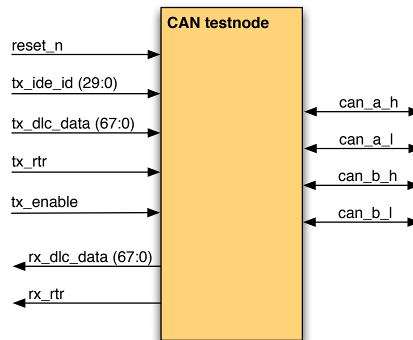
Bitstuffing Error: Missing bitstuffing bit during arbitration backoff
Bitstuffing Error: Bitstuffing bit got overwritten during arbitration backoff
Bitstuffing Error: Illegal bit stuffing during arbitration: v_bitcount>4
Bitstuffing Error: Bitstuffing bit got overwritten during standard transmission
Bitstuffing Error: Illegal bit stuffing during transmission: v_bitcount>4
Bitstuffing Error: Missing bitstuffing bit after IDE reception
Bitstuffing Error: Missing bitstuffing bit during RB0/DLC reception
Bitstuffing Error: Missing bitstuffing bit after IDE reception
Bitstuffing Error: Missing bitstuffing bit during RB0/DLC reception
Bitstuffing Error: Missing bitstuffing bit during CRC reception”
Form Error: Illegal bit detected during transmission
Form Error: Illegal bit detected during transmission of CRCdel
Form Error: Illegal RB0 bit
Form Error: Illegal bit detected during transmission of ACKdel, EOF, IFS
Form Error: Illegal CRCdel
Form Error: Illegal ACKdel/EOF/IFS
CRC error: error during reception
Acknowledgement Error: No acknowledgement
Error: Illegal data length received
Error: Extended frame transmission not supported
Error: Extended frame reception not supported
Error: Illegal RTR/SRR/IDE combination

Frames De testnodes werden ontworpen om de functionaliteit van een externe node (in dit geval de GRCAN core) te testen. De belangrijkste tests zijn dus het zenden en ontvangen van standaard data en RTR frames. Deze functionaliteit werd dan ook geïntegreerd in de testnodes. Error en overload frames worden niet ondersteund aangezien de signalisatie rechtstreeks in de test bench gebeurt aan de hand van error meldingen. Bovendien ondersteunt de GRCAN core zelf geen overload frames. Ook het zenden en ontvangen van extended CAN frames wordt niet ondersteund. Voor deze beperking werd bewust gekozen omwille van het beschikbare tijdsbestek waarin gewerkt moest worden. In het ontwerp werd evenwel rekening gehouden met het bestaan van extended CAN frames. De frames worden wel herkend door de testnodes, maar kunnen niet verder verwerkt worden voor verzending of ontvangst. Dit wordt aangegeven met behulp van een foutmelding indien nodig. Zoals duidelijk wordt in paragraaf 5.3.3 op pagina 44, werd de integratie van extended remote frames wel al in de planning opgenomen, zodat de ondersteuning hiervoor in de toekomst eenvoudig geïntegreerd kan worden.

5.3.2 Gebruik

De CAN testnodes werken volledig autonoom dankzij de integratie van een data link laag. Dit wil zeggen dat enkel data aangeboden moet worden, de verdere verwerking tot CAN frames gebeurt

door de testnodes. Voor de aan- en afvoer van data worden de nodes voorzien van verschillende in- en uitgangen, zoals weergegeven in figuur 5.2.



Figuur 5.2: In- en uitgangen van de CAN testnodes

Ten eerste is er een *reset_n* signaal dat voorlopig geen nut heeft. Het werd geïntegreerd om in de toekomst een eventuele reset van het model te kunnen doorvoeren, maar in de huidige omstandigheden is dit een overbodige functie.

Vervolgens is er een *tx_ide_id* signaal dat bestaat uit 30 bits. Dit signaal bevat de identifier van het CAN frame dat men wenst te verzenden. Bit 29 duidt aan of het om een standaard identifier gaat (aangeduid met '0'), of een extended identifier ('1'). Men dient evenwel op te merken dat ondersteuning voor extended identifiers nog voorzien moet worden, zoals uitgelegd werd in paragraaf 5.3.1 op de vorige pagina. De volledige structuur van het signaal wordt visueel weergegeven in tabel 5.5.

Tabel 5.5: Beschrijving van het tx_ide_id signaal

Bit:	29	28..18	17..0
Inhoud:	Std/Ext ID selectiebit	Std ID of begin ext ID	Vervolg ext ID

Het *tx_dlc_data* signaal bevat 68 bits, waarvan de eerste vier de data length code (DLC) bevatten, dit is een binaire code die aanduidt hoeveel bytes data er opgenomen moeten worden in het frame (0 tot 8). De daaropvolgende bytes bevatten de effectieve data, zoals duidelijk is in tabel 5.6

Tabel 5.6: Beschrijving van het tx_dlc_data signaal

Bit:	67..64	63..56	55..48	47..40	39..32	31..24	23..16	15..8	7..0
Inhoud:	DLC	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8

Met het *tx_rtr* signaal kan de gebruiker aangeven of het frame dat verzonden zal worden een data frame of RTR frame is.

Het *tx_enable* signaal wordt gebruikt om een transfer te starten. Men dient er dus voor te zorgen dat eerst de *tx_ide_id*, *tx_dlc_data* en *tx_rtr* signalen correct ingesteld zijn, alvorens men *tx_enable* hoog zet.

Verder zijn er twee output signalen die informatie teruggeven over een ontvangen CAN frame. Het

rx_dlc_data signaal geeft de data length code en de data weer die ontvangen werden, de structuur van dit signaal is identiek aan die van het *tx_dlc_data* signaal. Het *rx_rtr* signaal geeft aan of het ontvangen frame een data frame of remote transfer request frame was, net zoals het *tx_rtr* signaal. De *can_x_h* en *can_x_l* signalen worden aan de twee CAN bussen aangesloten.

5.3.3 Ontwerp

Fysieke laag

Ook in de CAN nodes werden twee CAN transceivers ingebouwd. Deze verschillen licht van de transceivers die ingebouwd werden in het model van de CAN interface module, beschreven op pagina 38. In de CAN testnodes werd immers een *transceiver enable* (*bus_x_en_i*) signaal ingevoerd, waarmee één van de twee CAN bussen geselecteerd kan worden voor transmissie. Hierdoor kan ook gewerkt worden met één drive signaal (*tx_stream_i*), aangezien nooit over twee bussen tegelijkertijd gestuurd moet worden. Overigens is de werking nagenoeg identiek aan die van de transceiver die geïntegreerd werd in de CAN interface module. Voor het receiver gedeelte volstaat het om alle zwakke signalen (‘L’ en ‘H’) om te zetten naar sterke signalen (‘0’ en ‘1’). Dit wordt duidelijk in de code. Als uiteindelijk receive signaal kan het *to_x01_can_x_l* signaal gebruikt worden.

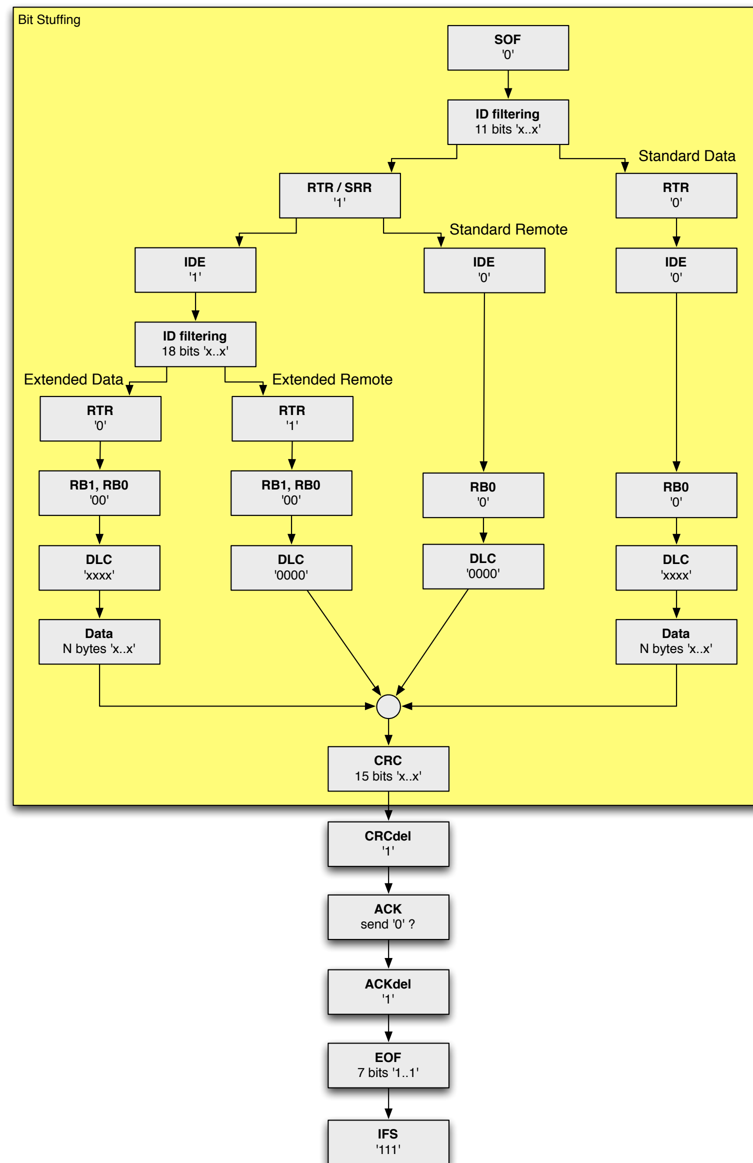
```
--transmitter gedeelte:
can_a_l <= '0'      WHEN (bus_a_en_i = '1' AND tx_stream_i = '0') ELSE 'H';
can_a_h <= '1'      WHEN (bus_a_en_i = '1' AND tx_stream_i = '0') ELSE 'L';

can_b_l <= '0'      WHEN (bus_b_en_i = '1' AND tx_stream_i = '0') ELSE 'H';
can_b_h <= '1'      WHEN (bus_b_en_i = '1' AND tx_stream_i = '0') ELSE 'L';

--receiver gedeelte:
to_x01_can_a_h <= To_X01(can_a_h);
to_x01_can_a_l <= To_X01(can_a_l);
to_x01_can_b_h <= To_X01(can_b_h);
to_x01_can_b_l <= To_X01(can_b_l);
```

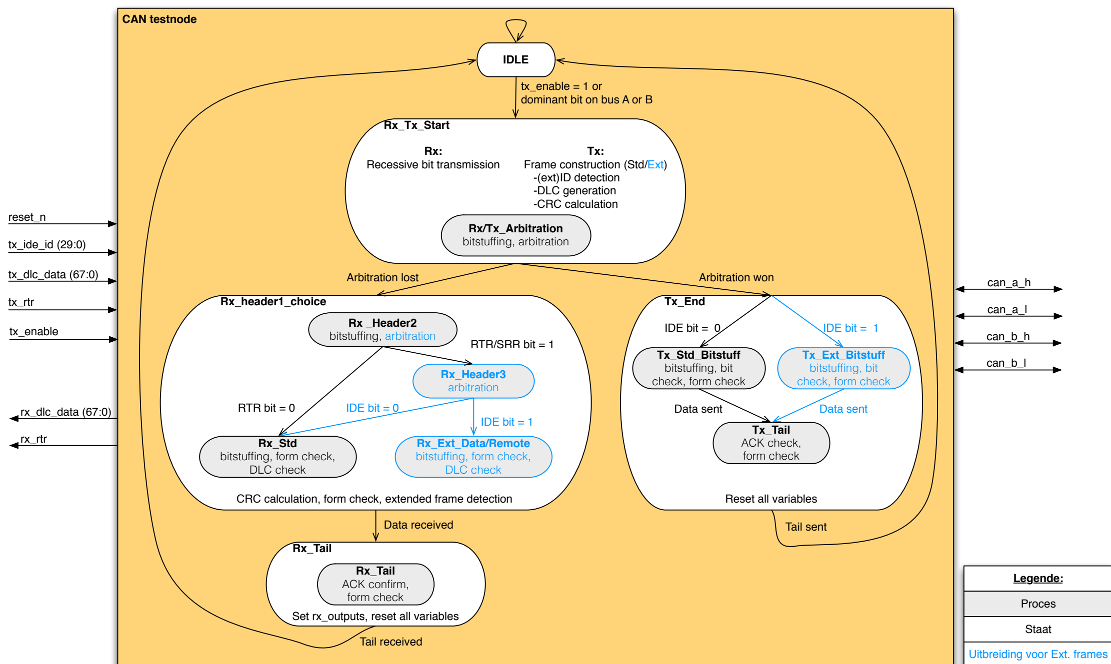
Data link laag

Om de data link laag te implementeren in VHDL is een veel complexer ontwerp nodig dan voor de transceivers. Deze complexiteit wordt veroorzaakt doordat de volledige frames samengesteld of ontcijferd moeten worden volgens de structuur van het CAN protocol en omdat een zeer intensieve foutdetectie doorgevoerd wordt. Bovendien zorgt de variabele lengte van de CAN frames, veroorzaakt door bit stuffing en een variabel aantal data bytes, ervoor dat geen statische structuur vastgelegd kan worden in het ontwerp, zoals dat wel het geval is in sommige andere protocols (vb. MIL1553b). Daarom werden de verschillen en gelijkenissen tussen de verschillende soorten CAN frames nauwkeurig geanalyseerd en uiteengezet in twee schema’s. Het eerste schema kan gebruikt worden voor het versturen van gegevens en wordt weergegeven in figuur 2.5 op pagina 12, hier worden de gelijkenissen samengevat naarmate men het einde van het frame nadert. Er wordt ook aangeduid waar bit stuffing en arbitratie plaatsvindt. Een tweede schema kan gebruikt worden voor het ontvangen van gegevens, aangezien stapsgewijs een opsplitsing wordt gemaakt, waardoor uiteindelijk het type frame bepaald wordt. Dit schema wordt weergegeven in figuur 5.3 op de pagina hierna.



Figuur 5.3: Schema voor het ontvangen van CAN frames

Om de verschillende opeenvolgende stappen voor het verzenden of ontvangen van CAN frames te integreren in één ontwerp, werd een methode uitgewerkt met behulp van een statenmachine. Hierin bevat elke staat een zekere hoeveelheid logica voor de keuze van faseovergangen, maar vanuit elke staat worden ook één of meerdere processen opgeroepen die het verzenden of ontvangen van data behandelen. Dit kan men waarnemen in figuur 5.4 op de volgende pagina. De functies die in het blauw weergegeven worden, dienen voor de behandeling van extended frames. Deze functies werden dus wel al gepland, maar dienen nog geïmplementeerd te worden in het VHDL ontwerp van de CAN nodes indien dit later noodzakelijk zou blijken.



Figuur 5.4: Statendiagram van de CAN testnode

Idle Bij het opstarten zal de statenmachine zich standaard in de *Idle* staat bevinden. In deze staat worden beide CAN bussen gecontroleerd op activiteit en wordt gewacht op een eventueel actief *tx_enable* signaal. Indien een signaal gedetecteerd wordt op één van de CAN bussen, wordt deze bus vastgelegd voor de hierop volgende acties zoals het zenden en ontvangen van bits. Indien het *tx_enable* signaal actief wordt, zal bus A geselecteerd worden voor de CAN transfers. Deze keuze is evenwel aanpasbaar door de interne variabele *v_bus_sel* op '1' te zetten, zodat bus B geselecteerd wordt. Indien nodig, kan deze variabele eenvoudig naar een uitgangspoort gebracht worden, zodat de actieve bus via een extern signaal gekozen kan worden. Nadat de actieve bus vastgelegd werd, wordt de volgende staat geselecteerd, de *Rx_Tx_Start* staat.

Rx_Tx_Start Elke transmissie of ontvangst van een frame begint bij de *Rx_Tx_Start* staat. In eerste instantie wordt hier het volledige CAN frame samengesteld. Indien een transmissie voorbereid wordt, worden alle identifier-, RTR-, IDE-, RB0- en DLC-bits vastgelegd, zoals weergegeven in tabel 2.5 op pagina 12. Ondertussen wordt ook gecontroleerd of het om een extended frame gaat. Indien dit het geval is, zal een error gegenereerd worden aangezien hiervoor (nog) geen ondersteuning ingebouwd werd. Tot aan het DLC veld kan men een statische vorm van het frame aanhouden. Hierna volgt echter het data veld, waarvan de lengte afhangt van de data length code (DLC). Het einde van het data veld is dus dynamisch, alsook het begin van het CRC veld. Na de bepaling van het data veld wordt een 15 bit CRC code berekend en ingevoegd in het CAN frame. Hoe deze berekeningen verlopen, wordt uitgelegd op de volgende pagina. De bits die volgen op het CRC veld zijn voor elke transmission identiek en dienen dus ook niet vooraf vastgelegd te worden in het CAN frame. Om een data reception voor te bereiden, wordt het CAN frame opgevuld met recessieve bits. Het verzenden van een dergelijk frame zal altijd resulteren in het verliezen van de arbitration, waardoor automatisch naar de ontvangstmodus wordt overgeschakeld. Na het

vastleggen van de CAN frames, wordt de *std_arbit* procedure uitgevoerd.

In de *std_arbit* procedure wordt het arbitratieveld uitgezonden. Indien hierbij de arbitratie verloren wordt, zal een back-off status aangenomen worden. Er worden dan enkel nog recessieve signalen uitgezonden, terwijl data ontvangen wordt. Uiteraard dient in dit gedeelte ook bit stuffing te gebeuren. In de verzonden bit stream zullen bit stuffing bits tussengevoegd worden en in de ontvangen bit stream zal een controle op bit stuffing plaatsvinden.

Tx_End Als de arbitratie gewonnen werd, kan men de transmissie voortzetten in de *Tx_End* staat. Voorlopig kan hierin enkel een standaard data of remote transfer request (RTR) frame uitgezonden worden met behulp van het *Tx_Std_Bitstuff* proces. Dit proces zorgt ervoor dat het standaard frame uitgezonden wordt tot net voor de CRC delimiter bit. In deze regio zal dus bit stuffing plaatsvinden. Verder worden er tijdens de transmissie bit checks en form checks uitgevoerd. Als de data verzonden is, wordt de transmissie voortgezet in een nieuwe procedure: *Tx_Tail*. Deze procedure zorgt dan voor de transmissie van het einde van het CAN frame. Hier zal dus geen bit stuffing meer plaatsvinden. Er zal een acknowledgement check en een form check uitgevoerd worden. Als deze procedure afgerond wordt, is het volledige CAN frame verzonden. Alle variabelen worden dan gereset en er wordt teruggekeerd naar de *Idle* staat.

Rx_header1_choice Indien de arbitratie in de *Rx_Tx_Start* staat verloren werd, zal de CAN node overgaan naar de *Rx_header1_choice* staat. In deze staat zal enkel nog data ontvangen worden. Met behulp van een eerste procedure, *Rx_Header2*, kan het onderscheid gemaakt worden tussen een standaard en extended CAN frame. Indien een extended frame gedetecteerd wordt, wordt weerom een foutmelding gegenereerd. Indien het om een standaard frame gaat, wordt de ontvangst voortgezet in de *Rx_Std* procedure. Hier wordt het CAN frame ontvangen tot en met het CRC veld. Intussen wordt het frame gecontroleerd op bitstuffing errors, form errors en een DLC error. Na de *Rx_Std* procedure wordt een CRC berekening uitgevoerd om het ontvangen CAN frame te controleren. Tenslotte wordt de ontvangst afgerond in de *Rx_Tail* staat.

Rx_Tail In de *Rx_Tail* staat wordt een *Rx_Tail* procedure uitgevoerd. Hierin wordt de acknowledgement bit geset als antwoord op het ontvangen CAN frame. Bovendien wordt nog een form check uitgevoerd. Na de procedure worden alle gebruikte variabelen gereset en wordt de ontvangen data op de *rx_* uitgangen gezet. Daarna keert de node terug naar de *Idle* staat.

CRC berekening Voor de CRC berekening wordt gewerkt met het CAN frame zonder bit stuffing bits. De CRC code kan beschouwd worden als de rest bij een deling van een bitstream door een vaste bitsequentie, de polynoom. De polynoom voor het CAN protocol is '100010110011001'. Als een CRC code berekend wordt voor verzending, zal deze berekend worden op basis van de bits voor het CRC veld, aangevuld met 15 nullen. Het resultaat van deze berekening kan dan rechtstreeks ingevuld worden in het CRC veld. Voor een controle van de CRC code na ontvangst, wordt de CRC berekening uitgevoerd op alle ontvangen velden van het CAN frame tot en met het CRC veld. De CRC code die nu bekomen wordt, moet 0 zijn. Indien dit niet het geval is, zal een CRC foutmelding optreden.

Voor de CRC berekening werd gesteund op een tip in de Bosch CAN specification[19]. Hier wordt gesuggereerd om een linksschuivend schuifregister te implementeren waarin een voorlopig CRC resultaat wordt opgeslagen. De inhoud van dit register wordt onder bepaalde voorwaarden

onderworpen aan een exclusive OR, waardoor uiteindelijk het CRC resultaat bekomen wordt. De exacte werking wordt weergegeven in de VHDL code op deze pagina.

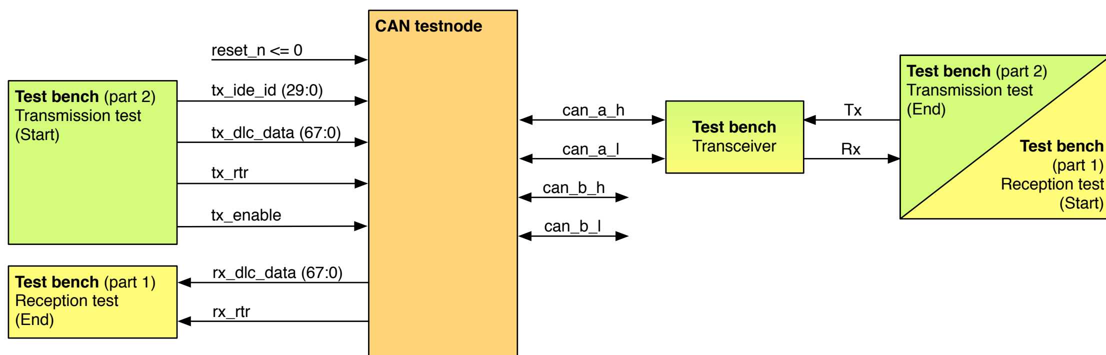
Deze methode met logische componenten vereist minder rekenkracht dan een louter wiskundige deling aangezien nu meer op hardware niveau gewerkt wordt.

```
v_crc := (OTHERS =>'0');
FOR I IN 131 DOWNTO (112 - v_data_length) LOOP
   v_crc(15 DOWNTO 1) := v_crc(14 DOWNTO 0);
   v_crc(0) := '0';
   IF ((v_crc(15) XOR v_tx_data(I)) = '1') THEN
       v_crc(14 DOWNTO 0) := v_crc(14 DOWNTO 0) XOR "100010110011001";
   END IF;
END LOOP;
v_tx_data((112 - v_data_length - 1) DOWNTO (112 - v_data_length - 15))
                                                    := v_crc(14 DOWNTO 0);
```

5.3.4 CAN testnodes test bench

Aangezien het ontwerp van de CAN testnodes vrij complex is, eist de ECSS richtlijn voor het ontwerpen en testen van ASICs en FPGA's dat het ontwerp eerst in een aparte test bench getest wordt, alvorens het te integreren in de algemene test bench voor de GRCAN core. Daarom werd ook hiervoor eerst een VHDL test bench geschreven die het zenden en ontvangen van CAN frames met de CAN testnodes moet verifiëren. Omwille van het korte tijdsbestek, worden ook hier enkel de belangrijkste functies getest, namelijk het zenden en ontvangen van CAN data frames op de nominale CAN bus (bus A). Indien deze tests positief zijn, is het overgrote deel van de werking geverifieerd. Aangezien hier een volledige test bench opgebouwd wordt, is het testen van bijkomende functies vrij eenvoudig in de bestaande structuur.

In figuur 5.5 wordt de structuur van de CAN node test bench weergegeven. Hierin wordt duidelijk dat ook deze een CAN transceiver bevat. De transceiver wordt hier gebruikt als verbinding tussen de CAN bus van de testnode en de CAN bus van de test bench.



Figuur 5.5: Schematische voorstelling van de CAN node test bench

De test bench bestaat uit twee delen die sequentieel worden uitgevoerd. In een eerste gedeelte wordt een data frame reception gesimuleerd. Dit wil zeggen dat de test bench een standaard CAN data frame op de CAN bus uitzendt. De CAN testnode zal dit frame ontvangen en als de transfer foutloos beëindigd is, worden de gepaste signalen normaal gezien op de *rx_* klemmen gezet. Deze uitgangen zullen dan weer gecontroleerd worden door de test bench.

Voor het zenden van het CAN frame, maakt de test bench gebruik van de *std_arbit*, *tx_std_bitstuff* en *tx_tail* procedures die beschreven werden in de CAN testnodes. Het data frame wordt hier echter volledig handmatig gedefinieerd alsook de CRC code. Hier wordt de statenmachine dus niet gebruikt, men kan dus met zekerheid testen of een geldig CAN frame herkend wordt.

In een tweede deel wordt een data frame transmission gesimuleerd. Hier biedt de test bench de correcte signalen aan, waaruit de CAN testnode een frame kan samenstellen en verzenden op de CAN bus. Dit frame wordt dan weer ontvangen door de testnode test bench, die de ontvangen data dan controleert op eventuele fouten. Voor het verzenden worden de *tx_* signalen aangestuurd door de test bench. Als de CAN testnode het frame begint te verzenden op de CAN bus, wordt het weer opgevangen door de test bench met behulp van de *std_arbit*, *rx_header2*, *rx_std* en *rx_tail* procedures van de CAN testnode. Het komt er dus op neer dat de data reception getest wordt met behulp van procedures voor data transmission en dat de data transmission getest wordt met behulp van procedures van data reception.

Indien tijdens deze tests fouten optreden, zal de simulatie niet stilvallen, maar de foutmeldingen zullen wel gemeld worden door de simulator. In de test bench voor de CAN test nodes waren foutmeldingen slechts een hulpmiddel bij het debuggen van de code. Een foutmelding kon er in dit stadium immers op wijzen dat er ofwel een fout optrad tijdens de transmissie, ofwel tijdens de foutcontrole. De foutmeldingen mogen pas als geldig beschouwd worden als de CAN testnodes volledig getest en gedebugged zijn. In de simulator wordt een bevestiging gegenereerd als de transmission en reception test correct afgerond worden. Deze melding verschijnt als error omdat dit de enige manier is om een boodschap weer te geven in de simulator. Uiteraard dient deze melding niet als error beschouwd te worden.

5.3.5 Resultaat CAN testnodes test bench

Voor de uitvoering van de simulaties werd gebruik gemaakt van het softwarepakket ModelSim 6.5b. Dit pakket laat toe de geschreven VHDL code te testen en alle signalen te volgen. Bovendien kan het pakket *.tcl* scripts interpreteren, wat nodig is om de automatisering en herhaalbaarheid te bereiken die opgelegd werden door de ECSS richtlijnen voor het ontwerp en testen van ASICs en FPGA's.

In een eerste *.tcl* script wordt het ontwerp van de testnode en de testnode test bench gecompileerd. Een tweede script zorgt ervoor dat de simulatie geïnitialiseerd wordt. De simulatie kan gestart worden door de twee testscripts uit te voeren. Hiertoe geeft men de *do build.do* en *do sim.do* commando's in de command line interface in. Met behulp van het *run -a* commando kan vervolgens de test bench uitgevoerd worden.

Simulatie met errors

Bij het debuggen werd duidelijk dat alle foutmeldingen correct functioneren. Fouten in de code leidden er immers toe dat de foutmeldingen opgeroepen werden op de gepaste momenten. Het debuggen van de code heeft ertoe geleid dat er geen error messages meer getoond worden. Toch wordt hier ter illustratie het resultaat van een simulatie weergegeven bij een ontbrekende acknowledgement bit (acknowledgement error) in zowel reception als transmission tests, en een verkeerd uitgezonden reserved bit (form error) bij de transmission test. In het transcript op de pagina hierna worden de foutmeldingen weergegeven. Voor de transmission test wordt een acknowledgement error gegenereerd na 50.5 μs. Toch wordt de data reception als succesvol beschouwd aangezien

alle data correct ontvangen werd. In de transmission test werd echter een form error geïnjecteerd onder de vorm van een verkeerde reserved bit, die na 88 µs gedetecteerd wordt in de simulatie. Deze error veroorzaakt wel een corrupt CAN frame, waardoor de data transmission test wordt afgesloten met de melding dat de ontvangen data incorrect is. Aangezien beide tests niet met succes afgerond werden, wordt dit ook meegedeeld bij het beëindigen van de test bench.

```
# ** Error: can_node_model: error: No acknowledgement
#    Time: 50500 ns  Iteration: 0  Instance: /tb_can_node_model/b_local_signals
# ** Error: Data reception successful
#    Time: 63 us  Iteration: 0  Instance: /tb_can_node_model/b_local_signals
# ** Error: can_node_model: Form Error: Illegal RB0 bit
#    Time: 88 us  Iteration: 0  Instance: /tb_can_node_model/b_local_signals
# ** Error: can_node_model: error: No acknowledgement
#    Time: 112501 ns  Iteration: 0  Instance: /tb_can_node_model/i_can_node_model
# ** Error: The received data on bus A is incorrect.
#    Time: 124 us  Iteration: 0  Instance: /tb_can_node_model/b_local_signals
# ** Failure: Ending of the testbench with ERRORS!!!.
#    Time: 134 us  Iteration: 0  File: ../tbench/tb_can_node_model.vhd
```

In figuur 5.6 wordt de bijhorende waveform weergegeven. De gedetecteerde fouten worden hier aangeduid op de overeenkomstige tijdstippen. De CAN bus signalen worden weergegeven als *can_a_h* en *can_a_l*.

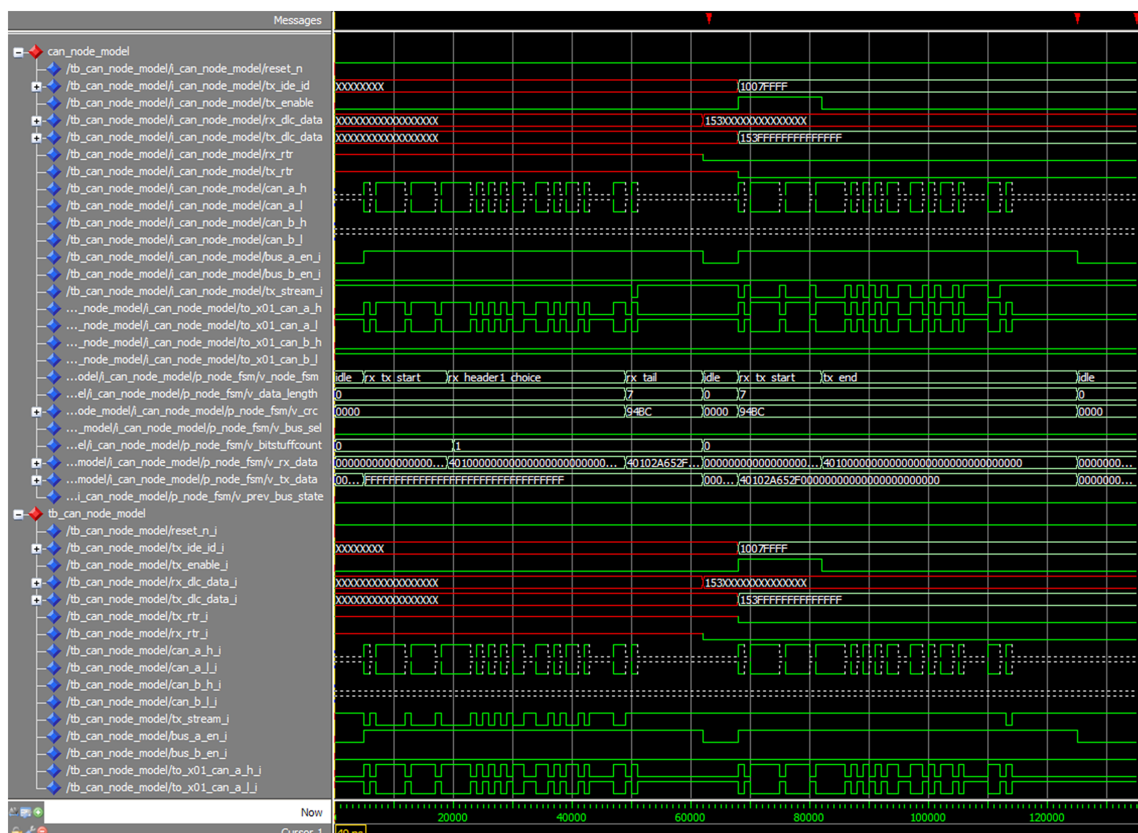
Figuur 5.6: Waveform bij een ontbrekende ACK bit en incorrecte RB0 bit

Succesvolle simulatie

Na het debuggen werd een simulatie bekomen waarin geen fouten meer gedetecteerd werden. Er werd dus bewezen dat het zenden en ontvangen van standaard data frames foutloos gebeurt met de CAN testnodes. Het resultaat van deze simulatie wordt weergegeven in het transcript op deze pagina. Hierin wordt de succesvolle afloop van de simulatie bevestigd.

```
# ** Error: Data reception successful
#    Time: 63 us  Iteration: 0  Instance: /tb_can_node_model/b_local_signals
# ** Error: Data transmission successful
#    Time: 125 us  Iteration: 0  Instance: /tb_can_node_model/b_local_signals
# ** Failure: Ending of the testbench with NO error(s).
#    Time: 135 us  Iteration: 0  File: ../tbench/tb_can_node_model.vhd
```

Figuur 5.7 toont de bijhorende waveform. De bovenste signalen zijn interne signalen uit de CAN testnode, terwijl de onderste signalen afkomstig zijn van de test bench. Het *v_node_fsm* signaal geeft de statenovergangen van de testnode aan. *v_rx_data* en *v_tx_data* geven respectievelijk het ontvangen en te verzenden CAN frame weer (zonder bit stuffing). Na 50 µs bemerkt men in het *i_can_node_model/tx_stream_i* signaal de acknowledgement bit, die uitgezonden wordt door de testnode als antwoord op een correct ontvangen CAN frame. De reception test wordt afgerond na 63 µs, waarna de transmission test start. Deze wordt dan afgerond na 125 µs.



Figuur 5.7: Waveform van een succesvolle reception en transmission test

5.3.6 Integratie in test bench

In de algemene test bench van de GRCAN core wordt een array van 5 testnodes geïntegreerd. De CAN interfaces van deze nodes worden aangesloten op de dubbele CAN bus van de test bench. De *tx_enable* klem van elke testnode wordt verbonden met een output signaal (*io_gate*) van de SCPU. Op deze manier kan de transmissie van een signaal gestuurd worden vanuit software. De overige interfaces voor het verzenden van data worden niet verbonden met een andere component uit de test bench. In de plaats daarvan worden rechtstreeks datasignalen gekoppeld aan de *tx_* signalen. Elke node krijgt andere informatie mee, zodat toch verschillende CAN frames verzonden kunnen worden. Eventueel kunnen verschillende nodes tegelijkertijd geactiveerd worden, waardoor men arbitration kan testen. In tabel 5.7 wordt de informatie voor elk van de vijf nodes weergegeven. Bij de keuze van de identifiers en data werd getracht een zo veelzijdig mogelijke implementatie te bekomen. Daarom werden grote verschillen voorzien in identifiers, lengte van de frames, aantal opeenvolgende identieke bits en soorten frames (RTR of data). Hierdoor kunnen functies als arbitration, bit stuffing, CRC calculation en form checks gecontroleerd worden.

Tabel 5.7: Informatie voor elk van de geïmplementeerde CAN nodes

Node nr.	ID (Std.)	DLC	Data (hex)	RTR bit
0	00000000011	1	53	0
1	01010101010	2	5353	0
2	00110011001	3	530053	0
3	00010000000	8	53FF00FF5300FF00	0
4	00001111111	0	-	1

Aangezien de test bench volledig geautomatiseerd moet worden, moeten de *rx_* outputs automatisch gecontroleerd kunnen worden. Hiertoe is een verbinding met de SCPU noodzakelijk. Deze verbinding wordt gemaakt via de aanwezige memory bus. Voor elke node worden de *rx_dlc_data* en *rx_rtr* signalen gemapped op de respectievelijke geheugenadressen 0x00000, 0x10000, 0x20000, 0x30000 en 0x40000. Aangezien de signalen bestaan uit 69 lijnen en de geheugenbus slechts 32 lijnen bevat, zal alle data telkens gespreid worden over 3 woorden, zoals beschreven wordt in de code op deze pagina. Uiteindelijk kan de nodige data dan via het mem_read commando ingelezen worden in de software.

```
node_select_i     <= conv_integer(mem_address_i(20 DOWNTO 16));
mem_data_i <= (node_rx_dlc_data_i(node_select_i)(67 DOWNTO 36))
        WHEN ((mem_oe_n_i = '0') AND (mem_address_i(31 DOWNTO 24) = x"10")
                AND (mem_address_i(1 DOWNTO 0) = "00"))
        ELSE (OTHERS => 'Z');
mem_data_i <= (node_rx_dlc_data_i(node_select_i)(35 DOWNTO 4))
        WHEN ((mem_oe_n_i = '0') AND (mem_address_i(31 DOWNTO 24) = x"10")
                AND (mem_address_i(1 DOWNTO 0) = "01"))
        ELSE (OTHERS => 'Z');
mem_data_i <= (node_rx_dlc_data_i(node_select_i)(3 DOWNTO 0)
        & conv_std_logic_vector(0, 28))
        WHEN ((mem_oe_n_i = '0') AND (mem_address_i(31 DOWNTO 24) = x"10")
                AND (mem_address_i(1 DOWNTO 0) = "10"))
        ELSE (OTHERS => 'Z');
mem_data_i <= (node_rx_rtr_i(node_select_i) & conv_std_logic_vector(0, 31))
        WHEN ((mem_oe_n_i = '0') AND (mem_address_i(31 DOWNTO 24) = x"10")
                AND (mem_address_i(1 DOWNTO 0) = "11"))
        ELSE (OTHERS => 'Z');
```

6
VHDL simulaties

In hoofdstuk 5 werd beschreven hoe een algemene test bench opgebouwd werd voor het testen van het CAN interface FPGA ontwerp. Voor het opbouwen van de test bench voor simulaties werd gesteund op de richtlijnen van het ECSS omtrent het ontwikkelen en testen van ASICs en FPGA's[18]. Zo werd ervoor gezorgd dat alle componenten van de test bench reeds getest waren of getest werden op core niveau. Ook voor het testen op module niveau worden richtlijnen opgegeven. Deze richtlijnen worden zo goed mogelijk gevolgd, maar ook hier zal een volledige uitvoering onmogelijk zijn gezien het grote aantal tests dat uitgevoerd zou moeten worden. In normale omstandigheden worden immers functionele tests, error injection tests en performance tests uitgevoerd. In functionele tests wordt de werking getest van de gebruikte interfaces (AHB, APB en PCI), registertoegang, data processing van de core en data flow. Bij error injection tests wordt het gedrag van de module getest als er fouten optreden en bij performance tests wordt het gedrag geobserveerd indien een functie kritisch belast wordt. Hoewel de geconstrueerde VHDL test bench zich leent tot de uitvoering van deze tests, zullen voor dit project slechts enkele functionele tests uitgevoerd worden in de simulaties om de normale werking van de GRCAN core de testen. Het hoofddoel van dit project is immers te onderzoeken hoe een CAN bus geïntegreerd kan worden, geen complete productontwikkeling. Daarom zullen er drie tests uitgevoerd worden. Ten eerste wordt registertoegang getest, daarna wordt een data reception met de GRCAN core getest en tenslotte wordt een data transmission met de GRCAN core getest. Voor deze tests zullen zelfverifiërende test scripts opgesteld worden. Deze scripts kunnen zelf het resultaat bepalen van een simulatie, zodat geen interactie door de gebruiker hoeft plaats te vinden, zoals het ECSS vastlegt. Bovendien worden ook alle resultaten van de tests automatisch gelogd en opgeslagen. De simulaties worden enkel uitgevoerd op RTL[1] niveau. Tests op netlist niveau zijn pas nodig indien het prototype stadium voorbij is en zullen dus niet uitgevoerd worden voor dit project.

[1]Register Transfer Level, het niveau van de hardware description language (in dit geval VHDL)

6.1 Test procedures

Om de uitvoering van de test bench te automatiseren, wordt gebruik gemaakt van *.tcl* scripts voor het compileren en simuleren. Het script voor compilatie kan gedeeltelijk overgenomen worden van het compilatiescript van de MIL1553b test bench. Het script zorgt ervoor dat voor elke geïmplementeerde VHDL IP core een compilatiescript opgeroepen wordt. Voor de IP cores van QinetiQ Space waren deze scripts reeds geschreven, maar voor de GRCAN core en voor de CAN testnodes dienden de compilatiescripts zelf geschreven te worden.

De uiteindelijk compilatie van de test bench bleek niet mogelijk met ModelSim 6.5b. Het gebruik van ModelSim 5.7f bleek dit probleem echter op te lossen. Men dient er wel op te letten dat compilatiebestanden niet uitwisselbaar zijn tussen verschillende versies van ModelSim. Na een overstap is het dus nodig dat alle reeds aanwezige compilatiebestanden verwijderd worden.

Na de compilatie dient de simulatie gestart te worden. Hiervoor werd eveneens een *.tcl* script gebruikt, gebaseerd op het simulatiescript voor de MIL1553b test bench. Het script zorgt er normaal gezien voor dat een gebruiker de gewenste test (registertoegang, data reception of data transmission) uit een lijst kan selecteren met behulp van de command line interface van ModelSim. Een bug in de ModelSim 5.7f software zorgt er echter voor dat deze interface niet werkt. Bijgevolg is een manuele aanpassing in de *.tcl* code van het simulatiescript nodig indien de gebruiker een andere testroutine wil uitvoeren.

Zoals reeds in hoofdstuk 5 duidelijk werd gemaakt, bevat de test bench een simulatie CPU die test scripts uitvoert. Deze test scripts worden geschreven met behulp van assembly instructies die sequentieel uitgevoerd worden. De inhoud van deze scripts wordt in de volgende paragrafen beschreven. Omwille van intellectuele eigendommen worden hier geen letterlijke commando's van de SCPU meegegeven. Een verduidelijkende uitleg en de transcripts van de uitgevoerde tests moeten echter ruimschoots volstaan om een duidelijk inzicht te creëren in het verloop van de tests.

6.2 Registertoegang

Een eerste test die moet aanduiden dat de GRCAN core correct geïntegreerd werd, is de registertoegang test. Hierin wordt geprobeerd om met behulp van de simulatie CPU toegang te krijgen tot de registers van de GRCAN core. De communicatie zal dus verlopen van de PCI bus naar de on-chip AHB bus. Aangezien de GRCAN core gebruik maakt van een APB bus voor registertoegang, wordt ook deze bus betrokken bij de test. Voor de adressen en inhoud van de registers werd de GRCAN datasheet[20] geraadpleegd.

6.2.1 Testscript

In eerste instantie wordt een initialisatie van de volledige test bench uitgevoerd. Dit wil zeggen dat alle IP cores geconfigureerd worden zodat de test bench correct kan functioneren. Één van de belangrijkste instellingen is wellicht de configuratie van de PCI2AHB bridge van de CAN interface module. Deze bridge dient zo geconfigureerd te worden dat ze zich als master kan gedragen op de PCI bus. Dit is nodig om bijvoorbeeld het geheugen te benaderen via DMA. Verder worden ook de andere cores van de test bench geïnitialiseerd, deze instellingen konden echter vooral overgenomen worden uit het MIL1553b project, een bespreking hiervan is dus overbodig.

Voor de test worden alle default waarden van de GRCAN core registers gecontroleerd. De waarden

worden dus via een read-commando uitgelezen en door de SCPU gecontroleerd. Indien de uitgelezen waarde niet overeenkomt met de verwachte waarde, zal een foutroutine uitgevoerd worden. Deze routine toont dan een foutmelding ("— FAILURE —") in ModelSim en incrementeert een error counter. Als de simulatie beëindigd wordt, zal de waarde van deze error counter gebruikt worden om te beslissen of er al dan niet errors zijn opgetreden tijdens de uitvoering van de simulatie. Dit wordt dan ook vermeld in de simulator, zodat in één oogopslag duidelijk is of de test succesvol uitgevoerd werd. De simulator kan dus afsluiten met de melding "— FAIL: 1 or more tests have failed criteria. —" of "— PASS: All tests successful. —".

Na de controle van de default waarden, worden de write-functies van de registers gecontroleerd. Dit wil zeggen dat elke beschrijfbare bit in de registers beschreven wordt, waarna de registerinhoud weer uitgelezen en gecontroleerd wordt door de SCPU. Bij de write tests worden ook de onbeschrijfbare bits beschreven, als deze bits dan weer ingelezen worden, mag hun waarde niet veranderd zijn. Verder wordt ook de werking van masker registers getest met het script voor registertoegang. Ook automatische clears worden gecontroleerd, zo zal bijvoorbeeld het pending interrupt register (PIR) automatisch gecleared worden als het uitgelezen wordt. Als het tweemaal na elkaar uitgelezen wordt, zal de tweede keer dus een gecleared register uitgelezen moeten worden. Dit wordt gedemonstreerd in een gedeelte van het transcript dat weergegeven werd op deze pagina. In een eerste kolom wordt het tijdstip weergegeven (in *ps*), de volgende kolom duidt de actie aan (vb. read of write). Daarna wordt de uitgelezen data weergegeven (vb. D[0xFFFFFFFF]), gevolgd door het adres waarvan gelezen werd(vb. A[0xB000010C]). De laatste kolom geeft het lijnnummer weer van het commando in het test script.

```
# <68145000 ps> <SCPU ><68145000 ps>  --------------------------------------------------------------------
# <68145000 ps> <SCPU ><68145000 ps>  -- Check Pending Interrupt register for write/read access.
# <68145000 ps> <SCPU ><68145000 ps>  --------------------------------------------------------------------
# <68775000 ps> <SCPU ><68775000 ps> write D[0xFFFFFFFF]; A[0xB000010C] (0)[587]
# <69435000 ps> <SCPU ><69435000 ps> read D[0x0001FFFF]; A[0xB000010C] (0)[588]
# <70095000 ps> <SCPU ><70095000 ps> read D[0x00000000]; A[0xB000010C] (0)[590]
```

Bij de uitvoering van de registertoegang test dient men er rekening mee te houden dat het schrijven van bepaalde registers invloed kan hebben op het gedrag van andere registers. Een eenvoudig voorbeeld zijn de masker registers, die zorgen dat een uitgelezen waarde gedeeltelijk gemaskeerd wordt. Bij het schrijven van data in registers dient men dus rekening te houden met de gevolgen die teweeg gebracht worden. Ten slotte wordt ook nog getest hoe de core reageert als een onbestaand register benaderd wordt. Hiertoe wordt data geschreven naar een adres waarop geen register werd vastgelegd. Indien dit adres daarna weer wordt uitgelezen, mag het nog steeds geen data bevatten (enkel nullen).

6.2.2 Resultaat

Het volledige transcript van de registertoegang test wordt weergegeven in bijlage B.1 op pagina 93. Hierin is duidelijk dat elk register wordt gecontroleerd op read en/of write access. Het doel van elke test wordt meegegeven in commentaar. Aangezien nergens een foutmelding optreedt, is het besluit dat alle tests succesvol waren correct.

Men kan uit dit positieve testresultaat dus afleiden dat de GRCAN core correct werd aangesloten op de APB bus. De AHB aansluiting werd met deze test nog niet getest aangezien deze enkel gebruikt wordt voor geheugentoegang, een functie die hier nog niet getest werd. Bovendien wil dit ook zeggen dat de AHB2APB_regs core correct functioneert, net als de AHB arbiter en PCI2AHB bridge van de CAN interface FPGA. Aangezien de tests door de SCPU uitgevoerd worden over de PCI bus, kan men ook besluiten dat de SCPU correct aangesloten werd op het CAN interface

board, dat op zijn beurt correct verbinding maakt met de CAN interface FPGA.

6.3 Data transfer CAN testnode - CAN interface board

In een tweede simulatie wordt getest of de GRCAN core correct CAN frames ontvangt en wegschrijft naar een ontvangstbuffer in een extern geheugen. Ook deze simulatie wordt gestuurd via een test script dat uitgevoerd wordt door de SCPU. Deze zorgt ervoor dat één van de CAN testnodes een CAN frame verstuurt op CAN bus A. Dit bericht moet dan ontvangen worden door de GRCAN core, die de inhoud ervan opslaat in de externe SRAM van de test bench. Voor deze test wordt dus gebruik gemaakt van de PCI interface voor de SCPU commando's en DMA data transfers. Ook de AHB en APB aansluitingen van de GRCAN core worden gebruikt, net als CAN bus A in de test bench en de I/O interface van de SCPU voor het aansturen van de CAN testnodes.

6.3.1 Testscript

Ook in dit script wordt gesteund op de initalisaties uit het MIL1553b project voor het configureren van de QinetiQ IP cores in de test bench. De belangrijkste zijn weerom de instelling van de PCI2AHB cores van zowel de SRAM module als de CAN interface FPGA. Deze moeten geconfigureerd worden als PCI master om transfers op de PCI bus te kunnen opstarten. Na deze initialisaties wordt de GRCAN core geconfigureerd met behulp van het configuratieregister. Dit 32 bit register bevat onder andere een scaler, twee fasesegmenten (*PS1* en *PS2*) en twee bits voor het instellen van de baudrate (*BPR*). Deze worden gebruikt voor het instellen van de bitrate op de CAN bus, afhankelijk van de 33 MHz systeemklok. De berekening van de bitrate gebeurt volgens vergelijking 6.1.

$$\text{bitrate} = \frac{\text{systeemklok}}{(scaler+1)\cdot 2^{BPR}\cdot(1+PS1+1+PS2)} \quad (6.1)$$

De variabelen dienen bovendien te voldoen aan enkele voorwaarden: $PS1_{[1..15]} > PS2_{[2..8]} \geq 4$, $scaler = [0..255]$ en $BPR = [0..3]$. Hier zal een bitrate van 1 Mbps ingesteld worden aangezien enkel deze bitrate ondersteund wordt door de CAN testnodes. Dit is bovendien de hoogst instelbare bitrate, die waarschijnlijk ook gebruikt zal worden in de uiteindelijke implementatie. Indien tests bij deze bitrate goed verlopen, zullen ook de lagere bitrates normaal gezien geen problemen opleveren. Om de bitrate van 1 Mbps te bekomen, werden volgende waarden gekozen: $scaler = 2$, $PS1 = 5$, $PS2 = 4$ en $BPR = 0$. De correcte uitkomst wordt verduidelijkt in vergelijking 6.2

$$\text{bitrate} = \frac{33\text{ MHz}}{(2+1)\cdot 2^{0}\cdot(1+5+1+4)} = 1\text{ Mbps} \quad (6.2)$$

Verder wordt in het configuratieregister de 'silent' mode uitgeschakeld, waardoor de GRCAN core ook zelf kan zenden op de CAN bus. Ook de actieve bus wordt geselecteerd (bus A) met behulp van een selectiebit. Er zijn verder nog *enable* bits ter beschikking voor het (de)activeren van transceivers. Deze functie wordt echter niet ondersteund door de geïmplementeerde transceivers, waardoor de waarde van de *enable* bits geen belang heeft.

Om deze configuratie in te stellen, wordt de waarde '0x02544006' in het configuratieregister geschreven. Deze waarde zal ook in de volgende tests gebruikt worden.

Na de instelling van de GRCAN core kan de core geactiveerd worden met behulp van een controle-register. Daarna wordt de receive buffer in het externe geheugen geconfigureerd. Hiervoor wordt het startadres en de grootte van de buffer vastgelegd in de GRCAN core registers. Het masker voor het filteren van CAN identifiers wordt zo ingesteld dat alle berichten geaccepteerd worden. Vervolgens wordt de receive functionaliteit van de core geactiveerd, na 10 bittijden (hier 10 µs) is de core dan klaar om CAN frames te ontvangen en weg te schrijven naar het geheugen.

Op dit moment kan een CAN testnode geactiveerd worden, waardoor een CAN frame wordt uitgezonden op de CAN bus. Dit gebeurt door gedurende een korte tijd het *tx_enable* signaal van testnode 0 te activeren met de I/O interface van de SCPU. Vanaf dit moment wordt continu de inhoud van de externe SRAM module gecontroleerd met behulp van een lus. Deze lus wordt onderbroken wanneer een gedeelte van de uitgezonden data in het geheugen terecht komt. Na een korte wachttijd wordt dan gecontroleerd of alle andere testnodes ook het uitgezonden frame ontvangen hebben. Dit gebeurt via de memory bus van de SCPU. Daarna wordt het volledige data frame, dat bestaat uit 4 woorden, gecontroleerd in de externe SRAM module. Om de test te beëindigen, wordt de inhoud van verschillende GRCAN registers gecontroleerd om de status van de core te controleren. Zo wordt het statusregister en interrupt register gecontroleerd, net als het receive channel interrupt register en de read en write pointers van de receive buffer. Deze pointers geven aan welk CAN frame laatst in het geheugen gelezen en geschreven werd. Indien een nieuw CAN frame ontvangen is, moet het verschil tussen deze pointers dus 1 bedragen.

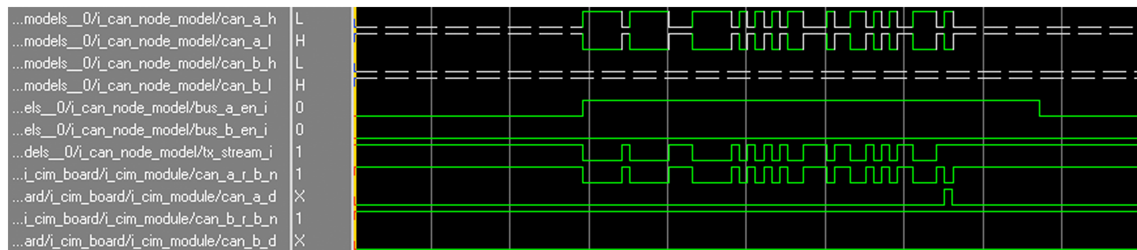
Ook voor dit testscript werd gebruik gemaakt van dezelfde foutroutines als bij de de registertoegang test. Op het einde van de test kan dus in één oogopslag gezien worden of de test succesvol was.

6.3.2 Resultaat

Het transcript van deze test wordt weergegeven in bijlage B.2 op pagina 96. Men kan hierop zien dat elke stap die uitgelegd werd in paragraaf 6.4.1 op de volgende pagina succesvol uitgevoerd werd, de foutroutine werd immers geen enkele keer opgeroepen. De verstuurde data kwam dus terecht bij alle nodes, ook de CAN interface module. Deze test bewijst dus dat alle geïmplementeerde nodes correct aangesloten werden op de CAN bus. Bovendien schreef de CAN interface module de ontvangen data met succes weg in een externe SRAM module. Dit bewijst dus dat de AHB interface van de GRCAN core correct werkt voor DMA write cycles, een functie die nog niet getest werd in de registertoegang test.

Na de transfer worden de GRCAN registers gecontroleerd. Bij deze controle bemerkt men dat het statusregister onveranderd bleef, en dus geen fouten aangeeft. In het interruptregister kan men wel een verandering waarnemen, een bit markeert nu ‘Successful reception of message’. Verder werd de write pointer van de receive buffer één plaats opgeschoven, wat erop duidt dat één CAN message in de buffer geschreven werd. De andere registers die gecontroleerd werden, bleven onveranderd, wat erop wijst dat er verder geen errors optraden.

Ter illustratie wordt in figuur 6.1 op de pagina hierna een waveform van de uitgevoerde test weergegeven. De bovenste vier signalen stellen de lijnen van respectievelijk CAN bus A en B voor. Het *i_can_node_model/tx_stream_i* signaal is het signaal dat door de CAN testnode wordt uitgezonden op CAN bus A. Dit wordt ontvangen door de CAN interface module, het ontvangen signaal is *i_cim_model/can_a_r_b_n*. Men kan opmerken dat de CAN interface module een acknowledgement stuurt op CAN bus A. Deze ene bit is duidelijk te zien in het drive signaal van de CAN interface module, *i_cim_model/can_a_d*. CAN bus B wordt in deze test duidelijk niet gebruikt.



Figuur 6.1: Waveform van de data reception test

6.4 Data transfer CAN interface board - CAN testnodes

De laatste test om de functionaliteit van de GRCAN core na te gaan, is een data transmission test. Hiervoor zal de GRCAN core in de test bench zelf een CAN frame ophalen uit een transmit buffer in extern geheugen. Dit frame wordt vervolgens verzonden op CAN bus A naar de CAN testnodes. Uiteindelijk wordt de ontvangen data van de testnodes dan gecontroleerd. Ook voor deze test wordt de SCPU gebruikt in combinatie met een testscript. Net als bij de reception test, worden hier de PCI, AHB, APB en CAN interfaces benut.

6.4.1 Testscript

In dit script worden de algemene instellingen van de test bench aangevuld met een activatie van de interrupt controller van de SCPU. In dit testscript kan dus ook een interrupt service routine geschreven worden, waarop verder wordt ingegaan.

Na de algemene instellingen, wordt de GRCAN core geconfigureerd op dezelfde manier als bij de reception tests. Er zal immers weer gebruik gemaakt worden van dezelfde CAN bus bij eenzelfde bitrate van 1 Mbps. Daarna kan de core geactiveerd worden met zijn controleregister. In de volgende stap wordt het CAN frame klaargezet in het externe SRAM geheugen. Hiertoe worden met behulp van de SCPU 4 woorden in het geheugen geschreven. Deze vier woorden worden ingevuld zoals de GRCAN datasheet [20] aangeeft. Dit wil zeggen dat een veld voorbehouden werd voor de identifiers, voor de data length code, voor de 8 bytes data, enz. De core zorgt dus zelf voor de uiteindelijke constructie van de CAN frames, net zoals de CAN testnodes. Dit hoeft dus niet in de software te gebeuren.

Vervolgens kan het startadres en de grootte van de transmit buffer ingesteld worden in de gepaste GRCAN registers. Hier is het ook belangrijk dat de read en write pointers gelijk worden ingesteld. Een verschil in read en write pointer zal er immers voor zorgen dat een actieve GRCAN core direct tot transmissie overgaat. Na de instellingen van de transmit buffer kan het transmit gedeelte van de core geactiveerd worden met behulp van het transmission control register. Vervolgens wordt het interrupt mask register zo geset dat geen enkele interrupt gemaskeerd wordt, een maatregel die nodig is voor de interrupt service routine die geïmplementeerd wordt. Uiteindelijk wordt dan de write pointer van de transmit buffer één plaats opgeschoven. Hierdoor wordt een verschil gecreëerd tussen read en write pointer, waardoor het frame dat zich tussen beide pointers bevindt, verzonden wordt. Na de transmissie zullen beide pointers weer gelijk zijn.

In normale omstandigheden zal een interrupt optreden als het bericht succesvol verzonden werd. Daarom wordt een interrupt service routine geschreven, een routine die bij het optreden van een interrupt het interrupt register van de CAN core zal inlezen en resetten. Daarna wordt het inter-

ruptsignaal gereset om te voorkomen dat de interrupt service routine nogmaals opgeroepen zou worden.

Nadat de transmissie gestart werd, wordt in een lus gecontroleerd of de verzonden data beschikbaar is aan de uitgangen van een CAN node. Zolang de data niet beschikbaar is, wordt deze lus doorlopen. Daarna wordt de data gecontroleerd in elke CAN node met behulp van de memory bus.

Tenslotte worden nog enkele registers gecontroleerd om de status van de GRCAN core te controleren. Het status register mag geen foutaanwijzingen bevatten en het interrupt register zou ook leeg moeten zijn, aangezien interrupts afgehandeld en gecleared worden door de interrupt service routine. Ook de waarden van de read en write pointers worden gecontroleerd, deze zouden aan het einde van de test weer gelijk moeten zijn.

6.4.2 Resultaat

Voor deze test werd geen positief resultaat verkregen. Bovendien was dit probleem niet eenvoudig op te lossen door debugging of kleine correcties in de VHDL code of test scripts, zoals dat wel het geval was voor de vorige test scripts. Het einde van deze test wordt weergegeven in het transcript op deze pagina. Dit einde werd bereikt door een limiet te zetten op de lus die normaal gezien wacht tot de data beschikbaar is in de testnodes. Zoals duidelijk te zien is aan de simulatietijden in de eerste kolom, werd zeer lang gewacht op data die er uiteindelijk niet kwam.

```
# <401145000 ps> <SCPU ><401145000 ps>  -- check CAN node 0
# <401235000 ps> <SCPU ><401235000 ps> read D[0xUUUUUUUU]; A[0x10000000] (0)[475]
# <401235000 ps> <SCPU ><401235000 ps>  --- FAILURE ---
# <401235000 ps> <SCPU ><401235000 ps>  -- check CAN node 1
# <401325000 ps> <SCPU ><401325000 ps> read D[0xUUUUUUUU]; A[0x10010000] (0)[480]
# <401325000 ps> <SCPU ><401325000 ps>  --- FAILURE ---
# <401325000 ps> <SCPU ><401325000 ps>  -- check CAN node 2
# <401415000 ps> <SCPU ><401415000 ps> read D[0xUUUUUUUU]; A[0x10020000] (0)[485]
# <401415000 ps> <SCPU ><401415000 ps>  --- FAILURE ---
# <401415000 ps> <SCPU ><401415000 ps>  -- check CAN node 3
# <401505000 ps> <SCPU ><401505000 ps> read D[0xUUUUUUUU]; A[0x10030000] (0)[490]
# <401505000 ps> <SCPU ><401505000 ps>  --- FAILURE ---
# <401505000 ps> <SCPU ><401505000 ps>  -- check CAN node 4
# <401595000 ps> <SCPU ><401595000 ps> read D[0xUUUUUUUU]; A[0x10040000] (0)[495]
# <401595000 ps> <SCPU ><401595000 ps>  --- FAILURE ---
# <401595000 ps> <SCPU ><401595000 ps>  ----------------------------------------------------------------
# <401595000 ps> <SCPU ><401595000 ps>  -- check CAN registers
# <401595000 ps> <SCPU ><401595000 ps>  ----------------------------------------------------------------
# <401595000 ps> <SCPU ><401595000 ps>  -- check Status register
# <402225000 ps> <SCPU ><402225000 ps> read D[0x00000000]; A[0xB0000004] (0)[506]
# <402225000 ps> <SCPU ><402225000 ps>  -- check Tx Channel interrupt register
# <402855000 ps> <SCPU ><402855000 ps> read D[0x00000000]; A[0xB0000214] (0)[512]
# <402855000 ps> <SCPU ><402855000 ps>  -- check Tx Write register (should be 0x00000010)
# <403485000 ps> <SCPU ><403485000 ps> read D[0x00000010]; A[0xB000020C] (0)[518]
# <403485000 ps> <SCPU ><403485000 ps>  -- check Tx Read register   (should be 0x00000010)
# <404115000 ps> <SCPU ><404115000 ps> read D[0x00000000]; A[0xB0000210] (0)[524]
# <404115000 ps> <SCPU ><404115000 ps>  --- FAILURE ---
# <404115000 ps> <SCPU ><404115000 ps>  -- check Tx channel CTRL register (should be 0x00000001: enabled, not ongoing)
# <404745000 ps> <SCPU ><404745000 ps> read D[0x00000003]; A[0xB0000200] (0)[530]
# <404745000 ps> <SCPU ><404745000 ps>  --- FAILURE ---
# <404745000 ps> <SCPU ><404745000 ps>
# <404745000 ps> <SCPU ><404745000 ps>  --- FAIL: 1 or more tests have failed criteria. ---
```

Zoals men kan zien in het transcript, ontvangen de testnodes geen data. Bovendien geven de testnodes zelf geen foutmeldingen uit tabel 5.4 op pagina 42, wat er wellicht op wijst dat de transmissie nooit gestart werd. Verder merkt men op dat de read pointer van de transmit buffer niet opgeschoven is, een teken dat de transmissie nog niet voltooid werd. Het transmit control register toont dat het transmit kanaal enabled is, zoals verwacht, maar ook dat een transmission ongoing is. Dit laatste is zeer vreemd aangezien er blijkbaar niets gebeurt op de CAN bus, bovendien werd er gewacht gedurende tientallen seconden, terwijl de transmissie van een CAN frame nog geen milliseconde duurt bij de ingestelde bitrate. Wat het probleem volledig onverklaarbaar maakt, is dat het status register en de interrupt registers geen enkel probleem signaleren. Een grondige analyse van het probleem is dus aangewezen.

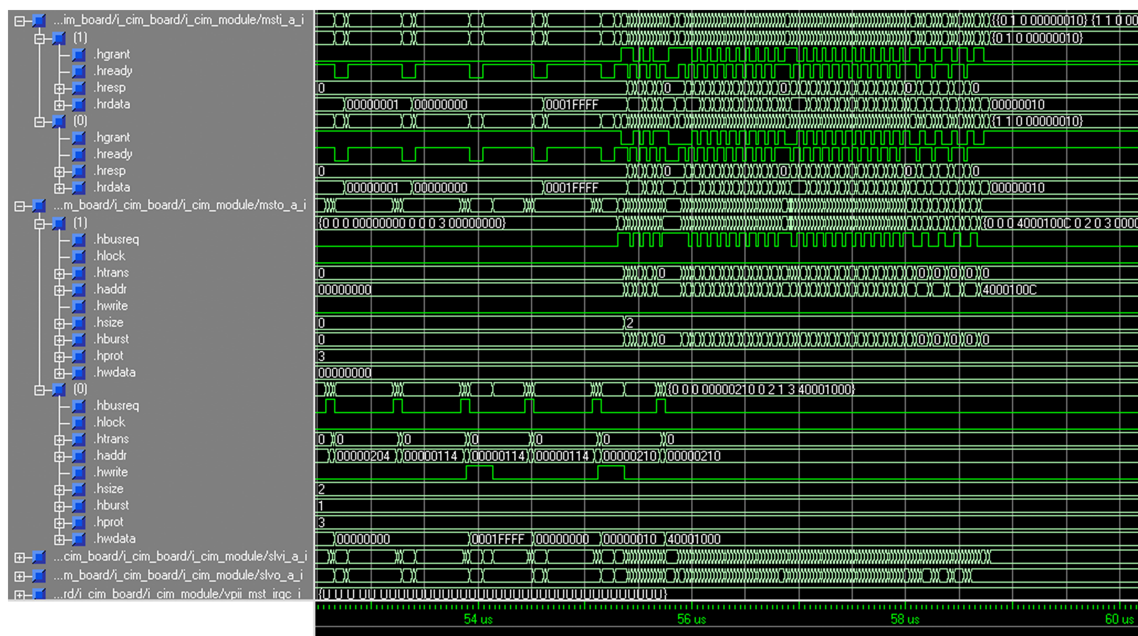
6.4.3 Probleemanalyse

In eerste instantie werd de test procedure gecontroleerd. Aangezien de datasheet van de GRCAN core hier geen volledige duidelijkheid schept, zijn verschillende interpretaties mogelijk. Tal van procedures werden uitgeprobeerd, zonder resultaat.

Daarom werd de data flow in de gehele test bench geanalyseerd. De commando's van de SCPU werden getraceerd op de PCI, AHB en APB interface. Hier werden echter geen anomalieën gevonden. Daarom werd ook de data flow van externe SRAM module naar de GRCAN core geanalyseerd over de PCI en AHB bus. De informatie die opgevraagd werd door de GRCAN core werd door de SRAM module ter beschikking gesteld, dus ook hier kwam geen probleem naar boven.

Ook werden alle VHDL ontwerpen gedetailleerd nagekeken en bestudeerd, van FPGA ontwerp tot test bench ontwerp. Hier werden verscheidene aanpassingen aangebracht, weerom zonder resultaat. Ook de endianness werd gecontroleerd doorheen het hele ontwerp, maar ook hier lag de fout niet. Daarom werd de hulp ingeroepen van QinetiQ Space en Gaisler Research, het bedrijf dat de GRCAN core ontworpen had. Desondanks werd er geen oplossing gevonden, er werd gesuggereerd om de testprocedures na te kijken en eventueel aan te passen.

Aangezien het probleem niet opgelost raakte, werden alle controles opnieuw en nog grondiger uitgevoerd. Zo werd uiteindelijk een probleem ontdekt met de AHB interface van de GRCAN core. De GRCAN core gebruikt de AHB interface voor het ophalen van een CAN frame in het externe SRAM geheugen. Aangezien de data verspreid wordt over 4 woorden, dienen deze woorden achtereenvolgens ingelezen te worden. De opgevraagde data zal echter niet direct beschikbaar zijn voor de CAN interface FPGA aangezien deze via PCI opgevraagd moet worden in een extern geheugen. Daarom zal een groot aantal retries geïntroduceerd worden op de AHB bus, waardoor de GRCAN core uiteindelijk stopt met het opvragen van data nadat het derde woord uit de transmit buffer ontvangen werd. Het vierde woord komt dus nooit aan in de GRCAN core. Deze situatie wordt visueel voorgesteld in de waveform van figuur 6.2 op de volgende pagina. Hierop worden de AHB master inputs en outputs weergegeven. Master 0 is de PCI2AHB bridge. Omdat deze de default master is op de AHB bus, krijgt hij zeer veel toegang tot de AHB bus (grant), zelfs al wordt er geen enkel bus request gestuurd naar de arbiter. Master 1 daarentegen, de GRCAN core, verliest steeds opnieuw de toegang tot de AHB bus, waardoor telkens een nieuw request gestuurd moet worden. Uiteindelijk wordt de transmissie dan stopgezet door de GRCAN core.

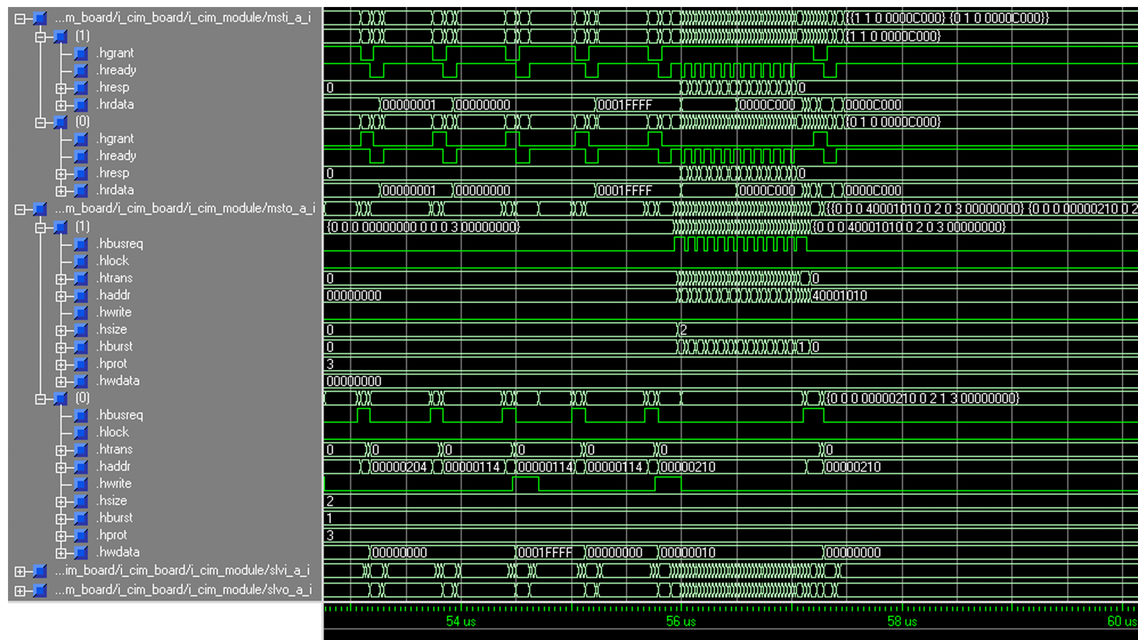


Figuur 6.2: Waveform van het transmit probleem

6.4.4 Oplossing

In eerste instantie werd getracht het probleem op te lossen door een update van alle QinetiQ IP cores in het VHDL ontwerp. Vooral de PCI2AHB core en AHBARB core hadden intussen belangrijke updates ondergaan, waardoor het probleem misschien opgelost zou kunnen worden door de cores te vernieuwen. Een vernieuwing van de cores bracht echter geen oplossing.

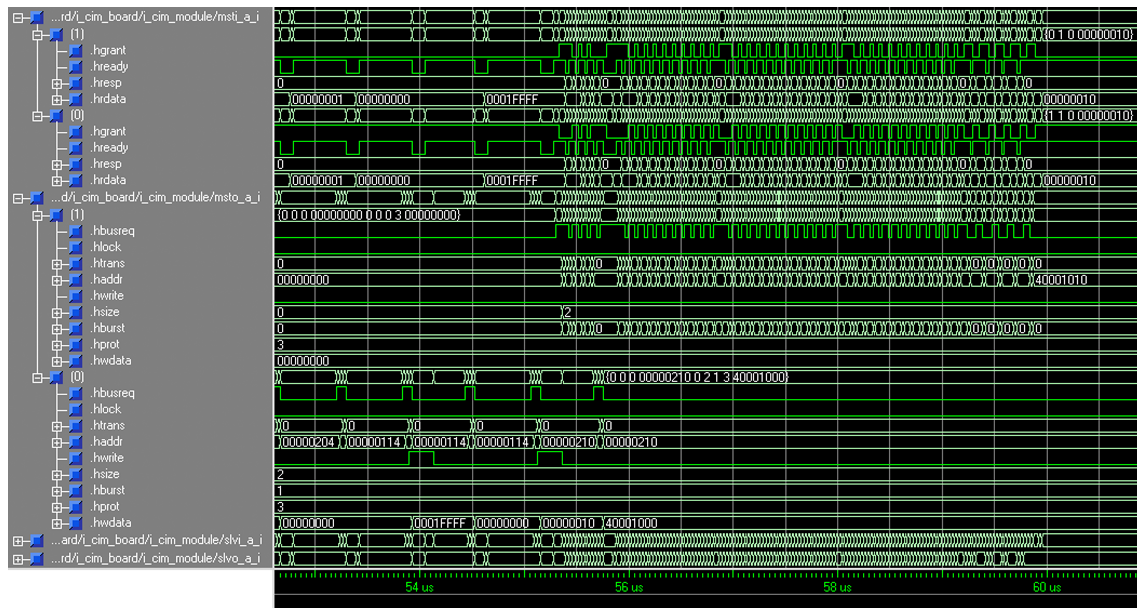
Daarom werd besloten om de default AHB master in het ontwerp te wijzigen. In plaats van de PCI2AHB core zou nu de GRCAN core ingesteld worden als default master op de AHB bus. Deze keuze is zeer riskant en zeker niet aanvaardbaar in ontwerpen waarin betrouwbaarheid een grote rol speelt. Ze zorgt er immers voor dat PCI toegang tot de CAN interface module nu niet meer 100 % gegarandeerd kan worden. Toch werd deze keuze gemaakt om de exacte oorzaak van de fout te kunnen bepalen en eventueel andere tests te kunnen verderzetten. Het resultaat van deze wijziging was positief. Aangezien de GRCAN core nu default master is, is het niet meer nodig om telkens opnieuw bus requests te sturen naar de AHB arbiter. De GRCAN core zal nu immers continu toegang hebben tot de AHB bus indien geen andere cores toegang vragen. Figuur 6.3 op de pagina hierna maakt duidelijk dat het ophalen van data via de AHB bus nu veel sneller verloopt en dat bovendien nu wel 4 woorden opgehaald kunnen worden uit het externe geheugen.



Figuur 6.3: Waveform van de AHB transfer indien de GRCAN core ingesteld wordt als default master

Hoewel het probleem nu opgelost lijkt, dient deze onbetrouwbare toestand vermeden te worden. Bovendien is het onveilig dat de GRCAN core zonder enige melding in interrupt of status registers het ophalen van data kan stopzetten, waardoor de werking van de core blokkeert. Deze bevindingen, samen met alle testresultaten werden dan ook doorgestuurd naar Gaisler Research, het bedrijf dat de GRCAN core ontwierp. Het bedrijf concludeerde hieruit dat een bug in de AHB interface van de GRCAN core de oorzaak van het probleem was. Daarom werd een nieuwe versie van de GRCAN core uitgebracht, waarbij de AHB bug verholpen was. Het zelfstandig onderzoeken en oplossen van deze bug was hier niet mogelijk, aangezien gewerkt werd met een netlist versie van de GRCAN core.

Nadat de nieuwe GRCAN core (nu versie 13 in plaats van versie 10) geïntegreerd werd, werden alle tests opnieuw uitgevoerd met succes. In figuur 6.4 op de volgende pagina wordt de correcte AHB transfer weergegeven. Het aantal retries ligt nog steeds hoog, wat normaal is aangezien de gevraagde data niet direct beschikbaar is. Nu zal de transfer echter niet gestopt worden, waardoor alle data toekomt in de GRCAN core en dus een CAN transmissie opgestart kan worden.



Figuur 6.4: Waveform van een correcte AHB transfer

In bijlage B.3 op pagina 98 wordt het transcript van de transmission test weergegeven. Hieruit blijkt dat de test succesvol verloopt. Na de transmissie wordt de interrupt service routine opgeroepen, waarin een succesvolle transmissie van een CAN frame gemeld wordt, evenals een succesvolle transmissie van alle CAN frames tussen read en write pointer. Enkele ogenblikken na de interrupt service routine, is de data beschikbaar aan alle CAN testnodes. Een laatste controle van de registers wijst erop dat de transmissie afgerond is en dat read en write pointers gelijk gekomen zijn, zoals verwacht.

6.5 Code Coverage

De testen die uitgevoerd werden, tonen aan dat de GRCAN core waarschijnlijk correct geïmplementeerd werd en dat de belangrijkste functies van de core correct werken. Volgens de richtlijnen van het ECSS zouden veel meer tests uitgevoerd moeten worden, wat buiten het kader van dit project valt. Het kan echter wel interessant zijn om te controleren in hoeverre het ontwerp getest werd. Hiervoor wordt statement en branch coverage getest in ModelSim. Bij statement coverage wordt onderzocht hoeveel statements van de VHDL code minstens eenmaal uitgevoerd werden. Dit aantal kan dan procentueel uitgedrukt worden ten opzichte van het totale aantal statements. Bij branch coverage wordt getest in hoeverre alle keuzemogelijkheden in de code doorlopen werden. Ook dit kan men procentueel uitdrukken ten opzichte van het totale aantal keuzemogelijkheden.

Voor de drie uitgevoerde simulaties werd telkens ook een code coverage test uitgevoerd met behulp van ModelSim. De resultaten van deze tests werden vervolgens samengevoegd, zodat een compleet beeld verkregen werd van de efficiëntie van de tests. Voor deze resultaten is enkel code coverage op FPGA niveau van belang, dit is immers het ontwerp dat getest wordt. Alle andere elementen van de test bench mogen hier dus niet meegerekend worden in de code coverage resultaten.

ECSS legt een minimum branch coverage op van 85 % voor het testen van VHDL ontwerpen,

voor statement coverage ligt het minimum zelfs op 90 %. Resultaten van 100 % zijn niet haalbaar aangezien sommige delen in de VHDL code gewoon niet bereikbaar zijn (bijvoorbeeld ‘others’ statements).

Uiteindelijk werd met de uitgevoerde simulaties een statement coverage bereikt van 79.9 % en een branch coverage van 44.6 %. Het is dus duidelijk dat verdere tests (onder de vorm van functionele tests, error injection tests en performance tests) uitgevoerd moeten worden om de ECSS normen te behalen. Men kan echter wel besluiten dat al een groot deel van de code met succes getest werd met de huidige tests.

7 Hardware tests

Nadat het ontwerp van de CAN interface FPGA getest werd in de simulaties die beschreven werden in hoofdstuk 6 op pagina 53, wordt het getest in een prototype van de MIL1553b interface module. Het gaat hier dus om hardware tests waarvoor de ontworpen VHDL test bench niet meer gebruikt wordt. Voor het uitvoeren van deze tests moet het ontwerp geïmplementeerd worden in een FPGA, waarvoor eerst synthese en daarna place & route nodig is. De FPGA kan dan geïntegreerd worden in het prototype van de MIL1553b interface module, die op haar beurt geïntegreerd wordt in een hardware test bench. De FPGA die gebruikt wordt in de flight module[1] is een Actel RTAX2000S/SL, zoals beschreven werd in paragraaf 3.1.3 op pagina 18. Dit type is echter niet aangewezen voor gebruik in prototypes gezien de zeer hoge kostprijs en het gebruik van antifuse technologie, waardoor de FPGA niet herprogrammeerbaar is. Daarom wordt in het prototype gebruik gemaakt van de Actel ProASIC3/E A3PE3000. Dit is een gelijkaardige FPGA, die evenwel gebruik maakt van Flash technologie en niet stralingsbestendig is. Deze FPGA is dus wel herprogrammeerbaar en heeft bovendien een kostprijs die ongeveer 100 keer lager ligt.

Voor het testen worden zowel functionele tests, error injection tests als performance tests uitgevoerd. De FPGA wordt dus intensiever getest op hardware niveau dan in de simulaties.

7.1 VHDL synthese, place & route en programming

7.1.1 Synthese

In de eerste stap voor het integreren van de VHDL code in de FPGA, moet de VHDL code gesynthetiseerd worden. Bij de synthese van VHDL code wordt het ontwerp eerst geoptimaliseerd en vervolgens omgezet naar FPGA logica, een zogenaamde netlist.

Voor dit project werd gebruik gemaakt van het Synplify softwarepakket van Synopsis. Net als bij de simulaties werden ook hier *.tcl* scripts gebruikt om de synthese te automatiseren. Een centraal script zorgt ervoor dat voor elke geïntegreerde IP core een apart *.tcl* script opgeroepen

[1] De module die uiteindelijk gebruikt wordt voor een missie

wordt dat zorgt voor de compilatie van de desbetreffende core. Verder wordt in het centrale script de gebruikte FPGA geselecteerd, alsook verschillende opties voor de synthese. Het centrale *.tcl* script kon gedeeltelijk overgenomen worden uit het MIL1553b project. Ook de verschillende *.tcl* scripts voor de compilatie van de QinetiQ Space IP cores waren reeds beschikbaar. Enkel het *.tcl* script voor de GRCAN core werd dus volledig zelf geschreven. Men dient hierbij op te merken dat dit compilatiescript niet gelijk is aan het compilatiescript voor de simulaties. Verschillende onderdelen die nodig waren voor de simulaties zijn immers niet synthetiseerbaar. Tenslotte werd ook een bestand aangemaakt met de timing constraints van het ontwerp. Dit vereiste echter weinig aanpassingen in vergelijking met het MIL1553b project.

Resultaat

In eerste instantie werd de synthese uitgevoerd met behulp van Synplify versie 9.6.1. Deze synthese resulteerde echter telkens in 43 (van de 73) constraints errors. Aangezien dit probleem niet opgelost raakte, werd gebruik gemaakt van Synplify versie 2009-03A-2. Hiermee kon de synthese doorlopen worden zonder errors. Er werden extra constraints, syntax en synthese controles uitgevoerd die telkens succesvol waren. Er werden wel 2926 warnings gegenereerd, maar na controle werd besloten dat deze genegeerd konden worden.

7.1.2 Place & route

Na de synthese dient een place & route uitgevoerd te worden van de netlist die geproduceerd werd door Synplify. Dit wil zeggen dat het ontwerp omgevormd wordt tot een bestand dat in de FPGA geprogrammeerd kan worden. Hiervoor wordt Actel Designer versie 8.6 gebruikt. In eerste instantie werd geprobeerd de place & route te automatiseren met behulp van een *.tcl* script, maar gezien de inconsistente syntax die gehanteerd wordt door Actel Designer, was dit niet mogelijk. Daarom werd de place & route manueel uitgevoerd. Hiervoor werden physical en timing constraints geïmporteerd. De physical constraints moesten integraal van het MIL1553b project overgenomen worden. Deze bevatten immers de pin connecties, die identiek moeten zijn aangezien het MIL1553b interface board prototype gebruikt wordt. De timing constraints werden eveneens overgenomen, maar hier werden wel verschillende aanpassingen doorgevoerd. Verder werd bij het configureren van de place & route het type FPGA, de IO standaard (LVTTL) en de junctietemperatuur (0 °C tot 70 °C) ingesteld.

In een eerste stap wordt het ontwerp gecompileerd. Hier wordt de netlist, gegenereerd met Synplify, geïnterpreteerd en omgezet naar logica die aanwezig is in de gebruikte FPGA. In de volgende stap kan het gecompileerde ontwerp dan volledig gemapped worden op de FPGA, de zogenaamde place & route. Vervolgens kan men alle eigenschappen van deze place & route controleren en eventueel aanpassen. Verder wordt nog een zogenaamde back-annotation uitgevoerd, een proces dat timing data uit het uiteindelijke resultaat zal afleiden, zodat een simulatie van de timing uitgevoerd kan worden. Ten slotte wordt een programmeerbestand gegenereerd, een STAPL bestand dat gebruikt kan worden om uiteindelijk de FPGA te programmeren. Dit zal echter weer met andere software gebeuren.

Resultaat

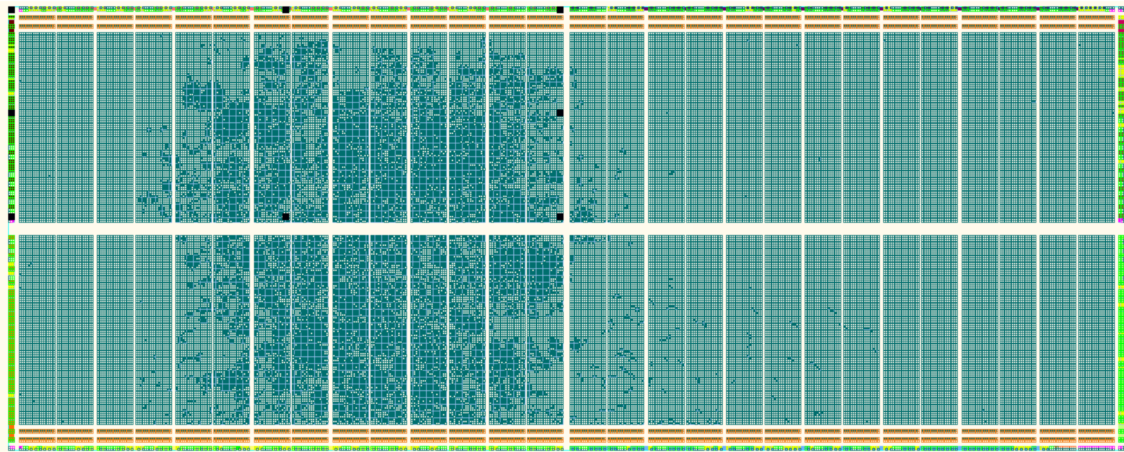
Na compilatie werd duidelijk dat 19.36 % van de 3 miljoen poorten op systeemniveau gebruikt zouden worden. De uiteindelijke FPGA voor de flight module bevat slechts 2 miljoen poorten,

maar dit is nog ruimschoots voldoende voor het ontwerp. Een gedetailleerd overzicht leert ons dat 165 van de 620 IO pinnen gebruikt worden. Verder worden alle 6 chip global networks gebruikt, evenals 2 van de 12 quadrant global networks. Deze global networks worden gebruikt voor een efficiënte verdeling van kloksignalen dankzij zeer korte stijgtijden.

De place & route gaf vrij veel problemen. Het bleek niet mogelijk te zijn om het ontwerp te routen binnen de FPGA, rekening houdend met de timing constraints. In ieder geval moest gekozen worden voor een timing driven routing om zoveel mogelijk te voldoen aan de timing delays. Ook de opties 'high effort layout', 'sequential optimization' en repair minimum delay violations' werden benut, ook al zorgt dit voor een veel intensiever rekenproces. Aangezien deze opties nog steeds niet tot een positief resultaat konden leiden, werden alle andere opties uitgetest. Zo werd het ontwerp gerouted in verschillende stappen, waarbij elke nieuwe stap de fouten van de vorige stap tracht te verbeteren. Na meer dan 100 pogingen kon geen resultaat verkregen worden dat voldeed aan de timing constraints. Uiteindelijk werd dan besloten om het beste resultaat te gebruiken voor verdere integratie. De timing simulaties houden immers rekening met een worst case scenario, waardoor de voorspellingen veel negatiever uitvallen dan de realiteit. Het uiteindelijke ontwerp bleek een signaal te bevatten dat maximaal 2.8 ns kan afwijken van de maximale delay. Van de minimale delay kan maximaal 0.9 ns afgeweken worden. Deze afwijkingen kwamen enkel voor bij signalen van de PCI2AHB core, een core die ook in andere projecten reeds moeilijk te routen bleek. De GRCAN core bleek echter geen enkel probleem te geven bij het routen. Om nog betere resultaten te bekomen, kan gebruik gemaakt worden van Actel Designer versie 9.1 SP1A. Deze nieuwere versie bewees reeds in andere projecten over betere routing capaciteiten te beschikken. Ook kan de variatie van de junctietemperatuur verkleind worden, waardoor de place & route minder rekenintensief zal zijn en men bijgevolg sneller tot betere resultaten kan komen. Deze aanpassing van de junctietemperatuur in de configuratie dient echter wel verantwoord te zijn.

Na de place & route kunnen verschillende eigenschappen van de FPGA gevisualiseerd of gesimuleerd worden. Zo wordt in figuur 7.1 op de volgende pagina de volledige lay-out van de FPGA weergegeven, waarin alle gebruike cellen ingekleurd werden. Hierop is te zien dat een relatief groot deel van het silicium oppervlak (ongeveer 80 %) onbenut blijft, waardoor de integratie van een MIL1553b IP core in de toekomst dus mogelijk blijft. Een dergelijke IP core van Gaisler Research zou immers slechts 14 % extra oppervlakte innemen.

Ook het vermogenverbruik van de FPGA werd gesimuleerd. Het maximale verbruik dat zo gevonden werd, bedraagt slechts 95.157 mW. Dit bestaat uit een statisch (en dus minimaal) vermogen van 38.940 mW en een dynamisch vermogen van 56.217 mW. Hiervan wordt slechts 2.949 mW verbruikt door de GRCAN core. Het statische verbruik van deze core bedraagt zelfs 0 mW. Een toevoeging van de GRCAN core in de bestaande MIL1553b interface module kan dus geen grote toename in vermogenverbruik veroorzaken.



Figuur 7.1: Lay-out van de FPGA na place & route

Ook werd er een rapport aangemaakt van alle pinconfiguraties van de FPGA. Dit rapport werd zeer nauwkeurig gecontroleerd om te verzekeren dat het geïmplementeerde ontwerp volledig compatibel is met het MIL1553b interface board prototype.

Na al deze controles werd uiteindelijk een STAPL programmeerbestand aangemaakt. Dit verliep zonder problemen.

7.1.3 Programming

Voor het programmeren van de FPGA werd gebruik gemaakt van de Actel FlashPro software en een USB programmer. Deze USB programmer kon rechtstreeks aangesloten worden op het prototype van de MIL1553b interface module, die op haar beurt in een lege cPCI backplane geplugd werd. Voor het programmeren werd een gespecialiseerde voeding aangesloten voor monitoring en beperking van alle stromen en spanningen. Dit zorgt ervoor dat eventuele kortsluitstromen en dus ook schade aan de hardware beperkt worden.

Resultaat

Het programmeren bestond uit drie stappen die probleemloos verliepen. Eerst werd de FPGA gewist, waarna hij geprogrammeerd en vervolgens geverifieerd werd. Tijdens het programmeren werden stromen gemeten tot 11.8 mA bij 5 V en 99.2 mA bij 3.4 V. Dit is een lichte verhoging ten opzichte van de waarden die gemeten werden bij normale werking, namelijk 9.0 mA bij 5 V en 72.0 mA bij 3.4 V. Verder werd ook de zogenaamde 'inrush' stroom gecontroleerd nadat de FPGA geprogrammeerd was, het verloop van de stroom bij het opstarten van de cPCI module. Het gemeten verloop kende geen onregelmatigheden en de spanning werd lineair opgebouwd volgens de verwachtingen. Deze elektrische tests zijn de enige die uitgevoerd werden. Gezien de hoge kostprijs van het prototype en de strenge procedures die verbonden zijn aan het uitmeten hiervan, was het niet mogelijk om uitgebreidere tests uit te voeren. Bovendien vormt dit niet de essentie van het project.

7.2 Selectie test board

Voor de uiteindelijke hardware tests is een commerciële CAN node nodig die enerzijds aangesloten kan worden op de CAN interface module (het MIL1553b interface module prototype) en anderzijds op een testsysteem, een PC met CAN test software. Er werden verschillende oplossingen van verschillende fabrikanten bestudeerd. De nadruk lag hier op USB to CAN adapters, aangezien deze probleemloos op een willekeurige PC aangesloten kunnen worden. Tegelijkertijd werd getracht om een oplossing te vinden die USB drivers voor zowel Windows als Linux bevat, zodat een eventuele integratie in de testsystemen van QinetiQ Space probleemloos kan verlopen. De USB-2-X adapter van Trinamic bleek op het eerste zicht een degelijke oplossing aangezien de adapter meerdere protocols ondersteunt, waaronder het CAN protocol. Bovendien is het mogelijk om hiervoor zelf software te ontwikkelen in C++, wat een zekere flexibiliteit van de testsoftware garandeert. Verder onderzoek wees echter uit dat de meegeleverde software enkel werkt onder windows (tot versie XP), bovendien was de documentatie zeer beperkt en uit een test bleek de software niet stabiel te werken.

De Imfsoft USB-CAN adapter bleek uitgebreide monitor software te bevatten, die evenwel enkel beschikbaar is voor Windows en sterk verouderd is. Deze software bleek uit tests bovendien zeer onstabiel. Ook deze optie valt dus niet te verkiezen.

Verder werd een USB-CAN adapter bestudeerd van Sys Tec, een volledig functionele adapter met zowel Linux als Windows drivers. Bovendien wordt een API meegeleverd voor verschillende programmeertalen, zoals C#, C++, java, python, Visual C++,... Verder wordt ook monitor software meegeleverd voor Windows. Uit nader onderzoek bleek echter dat de software door Peak System ontwikkeld werd, terwijl de hardware ontwikkeld werd door Sys Tec. Dit hoeft op zich geen probleem te zijn, maar de kans op problemen wordt hierdoor wel groter.

Een laatste USB-CAN adapter die gevonden werd, is de PCAN-USB adapter van Peak System. Deze adapter wordt ondersteund door drivers voor Linux en Windows XP tot Windows 7. De software is dus niet verouderd en wordt ook hier vergezeld van een API in dezelfde programmeertalen als de adapter van Sys Tec. Bovendien wordt deze adapter voorzien van de beste documentatie en kon de meegeleverde monitor software voor Windows op voorhand getest worden. Hieruit bleek dat de software stabiel werkt en dat alle basisfuncties van CAN bus ondersteund worden, waardoor de adapter zou voldoen voor alle basistests. Een extra voordeel is dat de hardware en software door dezelfde fabrikant ontwikkeld werden, waardoor deze dus optimaal op elkaar afgestemd konden worden en de kans op problemen verkleint.

Naast deze USB adapters werd ook gezocht naar systemen die optimaal geïntegreerd kunnen worden in de EGSE[2] van QinetiQ Space. Er werd dus gezocht naar compactPCI kaarten voor CAN bus, eventueel speciaal ontworpen voor testdoeleinden.

Zo werd de ARINC-825 PMC[3] kaart gevonden van AIM, een fabrikant van testsystemen voor lucht- en ruimtevaart. De ARINC-825 interface is de benaming voor CAN bus in lucht- en ruimtevaarttoepassingen. De kaart kan met behulp van een PMC-cPCI adapter geïntegreerd worden in de EGSE van QinetiQ Space. Er wordt een zeer compleet testsysteem met analyser meegeleverd dat uitgebreide tests mogelijk maakt. De hoge kostprijs maakt dit systeem echter onbruikbaar voor dit project, maar eventueel kan deze kaart gebruikt worden voor tests in een later stadium.

Tenslotte werd ook een cPCI module gevonden van Peak System. Deze kaart is optimaal integreerbaar in de testsystemen van QinetiQ Space en biedt bovendien alle voordelen die vermeld

[2] Electrical Ground Support Equipment, apparatuur voor het ontwikkelen en testen van ruimtevaarttoepassingen

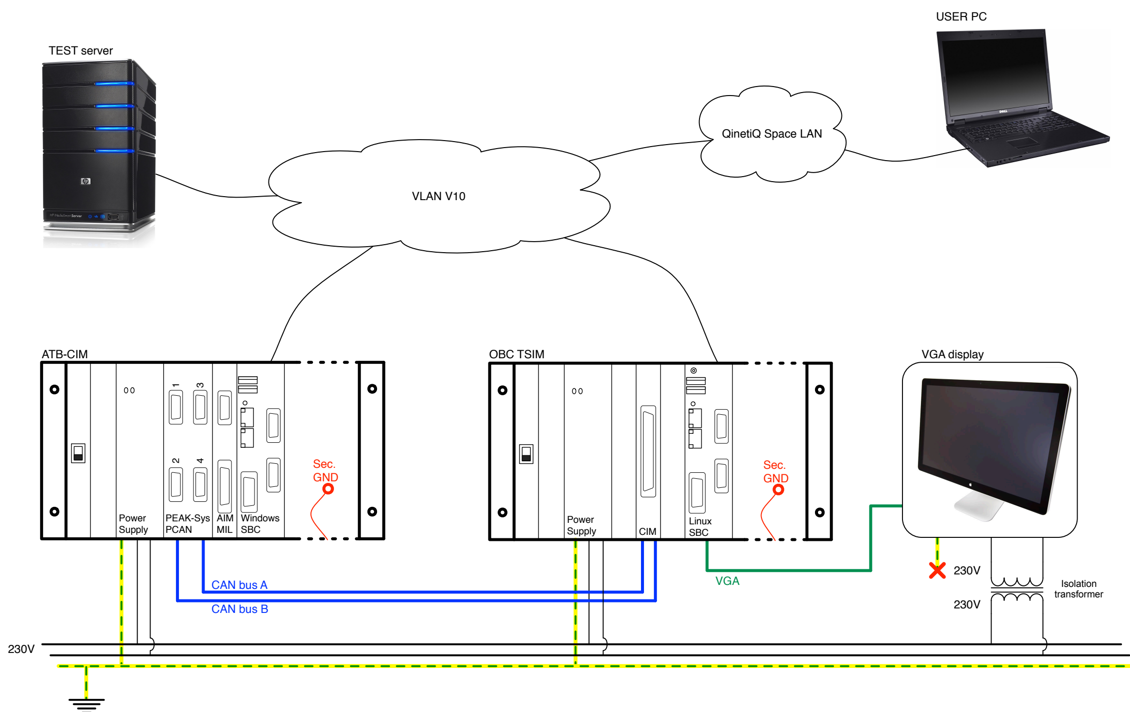
[3] PCI Mezzanine Card, een kaart die gebruik maakt van een aangepaste PCI interface

werden bij de PCAN-USB adapter. De kaart werkt immers met dezelfde software. Een bijkomend voordeel is dat deze kaart twee tot vier CAN bussen integreert, wat het testen van de redundante bus mogelijk maakt.

Uiteindelijk bleek deze laatste kaart dus ook meest geschikt voor dit project. Deze kaart werd dan ook gekozen en geïntegreerd in de hardware testopstelling.

7.3 Opbouw test bench

Voor het testen van de CAN interface module werd een volledige test bench opgesteld, zoals weergegeven in figuur 7.2.



Figuur 7.2: Opbouw van de hardware test bench

De test bench wordt opgebouwd uit een computer die de CAN testkaart bevat, de zogenaamde OBC TSIM[4], en een computer die het prototype van de CAN interface module bevat, de zogenaamde ATB-CIM[5]. Beide systemen worden met elkaar verbonden via de dubbele CAN bussen, zodat CAN bus communicatie getest kan worden. Verder worden de computers verbonden met een lokaal beveiligd VLAN V10 netwerk. Binnen dit netwerk bevindt zich ook een server voor testdoeleinden, waarop ingelogd kan worden vanaf het externe, minder beveiligde QinetiQ Space LAN netwerk.

[4]On-Board Computer Test Simulator, een computer die de werking van het ADPMS kan simuleren

[5]Automated Test Bench - CAN Interface Module, een computer met voorzieningen voor het testen van de CAN interface module

7.3.1 OBC TSIM

De OBC TSIM systeem is een systeem dat sterk lijkt op het ADPMS. Het bestaat uit een compactPCI backplane waarop verschillende modules ingeplugd kunnen worden. In dit geval werd een power supply module ingeplugd, de CAN interface module (CIM) en een commerciële single board computer. Er wordt dus geen echt MPM[6] board gebruikt met LEON processor, maar een commercieel alternatief met een Intel processor. Men dient er dus op te letten in de hardware tests dat gewerkt wordt met een little endian processor in plaats van de gebruikelijke big endian CPU. Op de single board computer werd een VGA display en toetsenbord aangesloten om ook lokale input en output van de command line interface mogelijk te maken. De single board computer draait met een Linux besturingssysteem gebruikt met DMA driver. Deze DMA driver werd specifiek voor dit project ontworpen door QinetiQ Space, zodat de GRCAN core berichten kan lezen en schrijven in een extern geheugen met behulp van DMA. De DMA driver biedt een 2 KB transmit buffer, gevolgd door een 2 KB receive buffer. De inhoud van deze buffers kan gelezen en aangepast worden vanuit de command line interface in Linux.

7.3.2 ATB-CIM

Het ATB-CIM systeem is een testsysteem dat eveneens opgebouwd werd op een compactPCI backplane. Hierop werd een power supply voorzien, de PCAN testkaart die beschreven werd in paragraaf 7.2 op pagina 69 en een single board computer. Verder was ook een testkaart aanwezig voor de MIL1553b interface, een kaart die niet gebruikt wordt voor CAN tests. De PCAN testkaart werd met 2 van de 4 CAN bussen verbonden met de dubbele CAN interface van de OBC TSIM. Hiervoor werd een kabel met gepaste connectoren en lijnterminatie voorzien door QinetiQ Space.

De single board computer die hier gebruikt werd, is identiek aan de computer die gebruikt werd in de OBC TSIM. Deze computer werd echter niet voorzien van een beeldscherm en draait bovendien op Windows XP, waardoor het mogelijk is om de PCAN monitorsoftware van Peak System te gebruiken.

7.3.3 Aarding

Bij de opstelling van de test bench worden beide testcomputers met elkaar verbonden via twee CAN bussen. Aangezien dit differentiële bussen zijn, wordt echter geen ground verbinding gecreëerd. Daarom dient men zelf voorzieningen te treffen zodat de twee testcomputers zich op eenzelfde grondpotentiaal bevinden, anders zouden de absolute potentialen op de CAN bus te groot kunnen worden voor één van beide testcomputers. Aangezien ook aardingslussen vermeden moeten worden, dient men elke mogelijke ground connectie te controleren.

De USB apparatuur die aangesloten wordt (memory stick, muis en toetsenbord) voorziet geen enkele connectie met de aarde en kan dus ook geëlimineerd worden als oorzaak van aardingslussen. De ethernet verbindingen van beide computers met het VLAN V10 netwerk voorzien ook geen ground connecties en moeten dus ook verder bestudeerd worden. Ook de CAN bussen zorgen niet voor ground connecties aangezien het om differentiële bussen gaat. De VGA connector heeft echter wel een shielding die verbonden wordt met de ground van het VGA display. Deze ground is op haar beurt verbonden met het chassis van het beeldscherm, dat geaard is in normale omstandigheden. Aangezien de twee testcomputers op een normale manier geaard worden, zorgt

[6] Main Processor Module

de VGA connectie dus voor een aardlus. Om veiligheidsredenen mag de aarding van het beeldscherm niet zomaar onderbroken worden, daarom werd een scheidingstransformator gebruikt voor het beeldscherm, zodat aarding overbodig wordt.

De twee testcomputers worden op een normale manier geaard op eenzelfde aardverbinding, waardoor de chassis van de twee computers zich op een gelijk potentiaal bevinden. Deze primaire ground wordt echter niet verbonden door het compactPCI backplane, waardoor de cPCI modules zonder aanpassing nog steeds een zwevend grondpotentiaal hebben. Daarom wordt nog een extra connectie voorzien van de secundaire ground op het backplane naar het chassis van de computers, zoals te zien is in figuur 7.2 op pagina 70.

7.3.4 Verbindingsprocedure

Om de tests uit te voeren werd gebruik gemaakt van een PC in het local area network (LAN) van QinetiQ Space. Hiermee werd dan vanop afstand controle overgenomen van de twee testcomputers. De ATB-CIM werd gecontroleerd via de 'Remote Desktop Connection' software van Microsoft. Hiermee was het dus mogelijk om de PCAN monitorsoftware te bedienen vanop afstand.

Om de OBC TSIM te controleren wordt een andere methode toegepast. Ten eerste dient men in te loggen op een Linux test server die zich in het beveiligde VLAN V10 netwerk bevindt. Hiervoor wordt het open source programma Putty gebruikt voor het opzetten van de verbinding. Om ook grafische interfaces op afstand te kunnen benaderen wordt gebruik gemaakt van Xming, een zogenaamde X-window server. Men is nu in staat om op de test server de ADPMS PCI client software van QinetiQ Space op te starten en te bedienen vanop afstand. De PCI client software zorgt ervoor dat alle registers van de cPCI modules (zoals de CAN interface module) in de OBC TSIM gelezen en geschreven kunnen worden met behulp van een grafische interface. Het is hier dus niet mogelijk om de tests te automatiseren aangezien de gebruikte software geen scripts ondersteunt. In een later stadium is automatisatie evenwel vereist, maar hiervoor zijn aanpassingen nodig in de ADPMS PCI client software van QinetiQ Space.

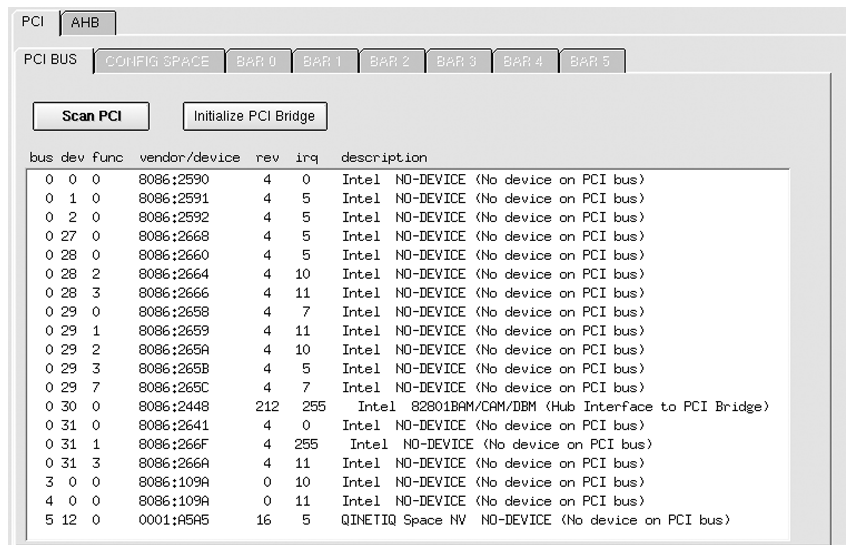
Voor de tests zal men echter ook toegang moeten krijgen tot de command line interface van de OBC TSIM, zodat de transmit en receive buffers gelezen en geschreven kunnen worden. Daarom wordt nogmaals ingelogd op de test server, die nu echter op zijn beurt via het telnet protocol verbinding maakt met de OBC TSIM. Zo heeft de gebruiker toegang tot de command line interface van de OBC TSIM via een test server. Ook hier is automatisatie niet mogelijk aangezien de DMA driver enkel manuele invoer via de command line ondersteunt. Ook deze dient dus in een later stadium nog aangepast te worden, zodat het gebruik van bijvoorbeeld *.tcl* scripts mogelijk is.

7.4 Registertoegang

Net als bij de simulaties in paragraaf 6.2 op pagina 54, wordt ook hier begonnen met een registertoegang test. In deze test wordt nagegaan of de CAN interface module ten eerste herkend wordt in de OBC TSIM computer. Vervolgens wordt getest of alle registers van de GRCAN core gelezen en/of beschreven kunnen worden, zoals ook gebeurde bij de simulaties. Voor al deze tests wordt de ADPMS PCI client software op de test server gebruikt. Hiervoor werd eerst een configuratiebestand aangemaakt waarin elk registeradres gelinkt werd met de overeenkomstige naam van het register en waarin de read/write functionaliteit werd vastgelegd.

Resultaat

Zoals men kan merken in figuur 7.3, wordt de kaart correct gedetecteerd tijdens een PCI scan van de OBC TSIM met behulp van de ADPMS PCI client software. De laatste module die gedetecteerd wordt, heeft de correcte vendor ID (0001), waardoor de kaart herkend wordt als QinetiQ Space kaart. De device ID (A5A5) die gedetecteerd wordt, komt ook overeen met de instelling die in de CAN interface FPGA geprogrammeerd werd. Deze device ID werd nog niet toegevoegd in de ADPMS PCI client configuratie, waardoor de naam van deze kaart nog niet weergegeven kan worden, daarom wordt ze weergegeven als 'no device'. Men kan dus besluiten dat de PCI bridge op het eerste zicht correct geïntegreerd werd in de CAN interface FPGA.



Figuur 7.3: Resultaat van een PCI scan: de CAN interface module wordt gedetecteerd

Vervolgens werd getest of alle geïntegreerde IP cores gedetecteerd werden. Ook deze test was succesvol, waardoor tenslotte alle read/write functionaliteit van de GRCAN core getest kon worden. In eerste instantie werden alle initiële waarden gecontroleerd. Vervolgens werden in de registers waarden geschreven die direct weer uitgelezen werden ter controle. Ook hier werden ongebruikte bits in de registers beschreven, waarvan de waarde onveranderd moet blijven, net zoals bij de simulaties. Verder werden ook automatische clears en masker registers getest. In de bestaande configuratie was het niet mogelijk om een onbestaand register te benaderen, een test die wel uitgevoerd werd in de simulaties en hier dus niet meer echt noodzakelijk is. In figuur 7.4 op de volgende pagina worden de resultaten van de registertoegang test opgelijst, waaruit blijkt dat alle tests met succes afgerond werden. Dit wijst ten eerste op een correcte synthese, place & route en programmatie van de CAN interface FPGA, maar ook op een correcte implementatie van de CAN interface module in de test bench.

Address	Bits	reg name	write value	write	read	read value	read value (binary)
0x00	32	Configuration Register (CanCONF)	0	WRITE	READ	FFFF731F	1111.1111.1111.1111.
0x04	32	Status Register (CanSTAT)	0	WRITE	READ	00000000	0000.0000.0000.0000.
0x08	32	Control Register (CanCTRL)	0	WRITE	READ	00000000	0000.0000.0000.0000.
0x18	32	SYNC Mask Filter Register (CanMASK)	0	WRITE	READ	00000000	0000.0000.0000.0000.
0x1C	32	SYNC Code Filter Register (CanCODE)	FFFFFFFF	WRITE	READ	1FFFFFFF	0001.1111.1111.1111.
0x100	32	Pending Interrupt Masked Status Register (CanPIMSR)	0	WRITE	READ	00000000	0000.0000.0000.0000.
0x104	32	Pending Interrupt Masked Register (CanPIMR)	0	WRITE	READ	00000000	0000.0000.0000.0000.
0x108	32	Pending Interrupt Status Register (CanPISR)	0	WRITE	READ	0001FFFF	0000.0000.0000.0001.
0x10C	32	Pending Interrupt Register (CanPIR)	FFFFFFFF	WRITE	READ	00000000	0000.0000.0000.0000.
0x110	32	Interrupt Mask Register (CanIMR)	FFFFFFFF	WRITE	READ	0001FFFF	0000.0000.0000.0001.
0x114	32	Pending Interrupt Clear Register (CanPICR)	FFFFFFFF	WRITE	READ		
0x200	32	Transmit Channel Control Register (CanTxCTRL)	FFFFFFF1	WRITE	READ	00000001	0000.0000.0000.0000.
0x204	32	Transmit Channel Address Register (CanTxADDR)	FFFFFFFF	WRITE	READ	FFFFFC00	1111.1111.1111.1111.
0x208	32	Transmit Channel Size Register (CanTxSIZE)	FFFFFFFF	WRITE	READ	001FFFC0	0000.0000.0001.1111.
0x20C	32	Transmit Channel Write Register (CanTxWR)	FFFFFFFF	WRITE	READ	000FFFF0	0000.0000.0000.1111.
0x210	32	Transmit Channel Read Register (CanTxRD)	FFFFFFFF	WRITE	READ	000FFFF0	0000.0000.0000.1111.
0x214	32	Transmit Channel Interrupt Register (CanTxIRQ)	FFFFFFFF	WRITE	READ	000FFFF0	0000.0000.0000.1111.
0x300	32	Receive Channel Control Register (CanRxCTR)	FFFFFFFF	WRITE	READ	00000001	0000.0000.0000.0000.
0x304	32	Receive Channel Address Register (CanRxADDR)	FFFFFFFF	WRITE	READ	FFFFFC00	1111.1111.1111.1111.
0x308	32	Receive Channel Size Register (CanRxSIZE)	FFFFFFFF	WRITE	READ	001FFFC0	0000.0000.0001.1111.
0x30C	32	Receive Channel Write Register (CanRxWR)	FFFFFFFF	WRITE	READ	000FFFF0	0000.0000.0000.1111.
0x310	32	Receive Channel Read Register (CanRxRD)	FFFFFFFF	WRITE	READ	000FFFF0	0000.0000.0000.1111.
0x314	32	Receive Channel Interrupt Register (CanRxIRQ)	FFFFFFFF	WRITE	READ	000FFFF0	0000.0000.0000.1111.
0x318	32	Receive Channel Mask Register (CanRxMASK)	E0000000	WRITE	READ	00000000	0000.0000.0000.0000.

Figuur 7.4: Succesvol resultaat van de registertoegang test

7.5 Functional tests

Bij de functionele tests worden zoveel mogelijk functies van de GRCAN core getest. Voor dit project diende enkel een data reception en transmission getest te worden als bewijs van een correcte integratie. Toch konden nog extra tests uitgevoerd worden waardoor het bewijs nog beter onderbouwd werd.

Voor de functionele tests worden verschillende soorten CAN frames verzonden en ontvangen. De algemene configuratie van de GRCAN core voor elk van deze tests is dus nagenoeg identiek en kan dus reeds op voorhand besproken worden.

Om te beginnen wordt de PCI bridge ingesteld als PCI master, op deze manier kan de CAN interface module PCI transfers starten, wat nodig is voor het lezen en schrijven van data in de DMA buffers. Vervolgens wordt de GRCAN core geconfigureerd op de manier die beschreven werd in paragraaf 6.4.1 op pagina 58. De bitrate wordt dus ingesteld op 1 Mbps en de actieve CAN bus wordt geselecteerd. Daarna wordt de GRCAN core geactiveerd met behulp van een controle register. De receive instellingen worden dan geconfigureerd, waaronder het beginadres van de receive buffer, dat opgevraagd kan worden met de DMA driver. Dit adres wordt bepaald bij het opstarten van de computer en is bijgevolg variabel. Dit zorgt ervoor dat de tests niet geautomatiseerd kunnen worden en dus niet herhaalbaar zijn zonder tussenkomst van de gebruiker. Verder wordt een acceptance mask ingesteld, een masker dat bepaalt welke berichten geaccepteerd worden, aan de hand van hun identifier. Vervolgens worden de transmission instellingen geconfigureerd, ook hier wordt de grootte en het beginadres vastgelegd van de transmission buffer, alsook de transmit read en write pointers. Nadat de transmit en receive kanalen geactiveerd zijn, is de GRCAN core klaar voor gebruik.

Verder dient men ook de PCAN software op de ATB-CIM te configureren. Hier wordt één van de vier CAN bussen geactiveerd, de bitrate wordt ingesteld op 1 Mbps en eventueel wordt een CAN frame samengesteld voor transmissie. Al deze instellingen gebeuren manueel in de grafische interface van de PCAN software.

7.5.1 Data reception: CAN bus A en B

De eerste functionele test die uitgevoerd werd, is een data reception test. Voor deze test wordt een standaard CAN data frame samengesteld in de ATB-CIM met behulp van de PCAN software. Nadat de GRCAN core geïnitialiseerd is, verzendt de ATB-CIM het CAN frame op CAN bus A. Nadat de verzending met succes afgerond wordt, wordt gecontroleerd of het verzonden frame correct aangekomen is bij de GRCAN core. Hiervoor wordt het volledige CAN frame dat uit 4 woorden bestaat, gecontroleerd in de receive buffer. Aangezien hier met een Intel processor gewerkt wordt, zal de volgorde van de bytes in elk woord omgekeerd zijn omdat hier gewerkt wordt met het little endian formaat. Om de test af te sluiten worden de status- en interruptregisters van de GRCAN core gecontroleerd. Hier worden geen errors verwacht, maar wel een aanduiding voor een succesvolle ontvangst van een CAN frame. Bovendien dient de identifier match aangeduid te worden. Ook wordt de write pointer gecontroleerd, die nu met 1 verhoogd moet zijn.

Aan al deze verwachtingen werd voldaan, waardoor de test met succes afgerond werd. Het ontvangen frame in de receive buffer wordt weergegeven in figuur 7.5.Bovendien werd de gehele test ook herhaald op de redundante CAN bus (CAN bus B). Ook hier verliep de test succesvol. Men kan dus besluiten dat het ontvangen van data frames op beide CAN bussen correct verloopt. De ontworpen CAN controller en de CAN transceivers functioneren dus volgens de verwachtingen.

```
DMA receive buffer:
  cpu_addr =   0xD98E4800
  buffer size = 2048 bytes
  dma_handle = 0x198E4800
  bus_to_virt(dma_handle) = 0xD98E4800
  data : 00 00 0C 00 00 00 00 10 00 00 00 53 00 00 00 00
```

Figuur 7.5: Inhoud van de receive buffer na ontvangst van een standaard CAN data frame

7.5.2 Data transmission: CAN bus A en B

De tweede belangrijke test is een data transmission test. Hierin wordt probeerd om met behulp van de CAN interface module een standaard CAN data frame op de bus te zetten, dat dan ontvangen wordt door de PCAN testkaart. Hiertoe wordt na de initialisatie van de GRCAN core een standaard CAN data frame geschreven in de transmit buffer met behulp van de DMA driver. Ook hier dient men te letten op de little endianness van de Intel processor. Nadat het frame in de transmit buffer geschreven werd, wordt de PCAN software in 'record' mode gezet, waardoor alle activiteit op het CAN netwerk opgeslagen wordt. Daarna kan de write pointer van de transmit buffer geïncrementeerd worden, waardoor de transmissie automatisch start. Het ontvangen CAN frame wordt dan gecontroleerd met de PCAN software en ook hier worden de status- en interruptregisters gecontroleerd op eventuele errors. De transmit read en write pointers moeten na de transmissie weer gelijk staan.

Ook deze test verliep succesvol omdat geen errors gedetecteerd werden en omdat het CAN frame correct aankwam in de PCAN testkaart, zoals weergegeven wordt in figuur 7.6 op de pagina hierna. Een identieke test werd ook met succes uitgevoerd op de redundante CAN bus. Men kan dus besluiten dat de basisfunctionaliteit van de CAN interface module voldoet aan de vereisten.

Recording...	23,0888 s	0,00 %	Ring Buffer	Rx: 1	Tx: 0	Errors: 0

Time	Type	ID	DLC	Data
23,0888	Data	003	1	53

Connected to PCAN-PCI (1 MBit/s) Overruns: 0 QXmtFull: 0

Figuur 7.6: Het CAN frame dat verzonden werd door de CAN interface module

7.5.3 RTR reception

De volgende functionele test die uitgevoerd werd, omvat de ontvanst van een remote transfer request. Deze aanvraag voor data wordt gesignaleerd door de GRCAN core, het antwoord zelf moet via een hoger protocol in software gegenereerd en verzonden worden, dit maakt dus geen deel uit van de test. Deze test loopt grotendeels zoals de data reception test die beschreven werd in paragraaf 7.5.1 op de vorige pagina. In deze test verzendt de PCAN testkaart echter een standaard RTR frame op bus A. Uiteindelijk wordt de receive buffer van de GRCAN core gecontroleerd, net als de interrupt- en statusregisters.

Deze test werd succesvol afgerond, net als de vorige tests.

7.5.4 RTR transmission

Vervolgens werd een RTR transmission test uitgevoerd, waarbij de GRCAN core een standaard RTR frame verzendt naar de PCAN testkaart. Deze test verliep grotendeels zoals de data transmission test, beschreven in paragraaf 7.5.2 op de pagina hiervoor. De test verschilt echter wel op een belangrijk punt. De PCAN software integreert namelijk ook gedeeltelijk een hoger protocol. Hierdoor zal de PCAN testkaart ook direct een antwoord op het RTR frame proberen uitzenden indien het beschikbaar is in de PCAN transmit buffer. Dit wordt duidelijk gemaakt in figuur 7.7. Hierop kan men meteen zien dat het RTR frame zonder problemen aankomt in de PCAN testkaart.

Recording...	32,6931 s	0,00 %	Ring Buffer	Rx: 1	Tx: 1	Errors: 0

Time	Type	ID	DLC	Data
32,6671	RTR	003	0	
32,6931	Data	003	1	53

Connected to PCAN-PCI (1 MBit/s) Overruns: 0 QXmtFull: 0

Figuur 7.7: De testkaart ontvangt een RTR frame en antwoordt hierop

Vervolgens kan men controleren of het antwoord correct aankomt bij de CAN interface module. Hiervoor wordt de receive buffer gecontroleerd. Figuur 7.8 op de pagina hierna geeft de transmit en receive buffers van de CAN interface module weer. De transmit buffer bevat nog steeds het RTR frame met identifier ‘003’, terwijl de receive buffer het antwoord van de PCAN testkaart bevat, met eveneens de correcte identifier ‘003’.

```
Linux kernel module cim
DMA transmit buffer:
  cpu_addr =   0xD907E000
  buffer size = 2048 bytes
  dma_handle = 0x1907E000
  bus_to_virt(dma_handle) = 0xD907E000
  data : 00 00 0C 40 00 00 00 00 00 00 00 00 00 00 00 00
DMA receive buffer:
  cpu_addr =   0xD907E800
  buffer size = 2048 bytes
  dma_handle = 0x1907E800
  bus_to_virt(dma_handle) = 0xD907E800
  data : 00 00 0C 00 00 00 00 10 00 00 00 53 00 00 00 00
```

Figuur 7.8: Buffers van de GRCAN core na uitzending van een RTR frame en ontvangst van het antwoord

7.5.5 Extended data frame reception

Aangezien het gebruik van extended CAN frames niet ondersteund werd in de simulaties, werd deze functionaliteit wel in de hardware getest. Hiertoe werd een identieke procedure gevolgd zoals beschreven in paragraaf 7.5.1 op pagina 75. Deze keer werd echter een extended CAN data frame uitgezonden. De identifier bestaat dus uit 29 bits in plaats van 11 bits en de structuur van het CAN frame verschilt licht van standaard CAN frames, zoals reeds toegelicht werd in paragraaf 2.3.3 op pagina 10. Het resultaat van deze test is te zien in figuur 7.9, waar de receive buffer van de GRCAN core weergegeven wordt. Hier werd gewerkt met een geüpdatete versie van de DMA driver die de little endianness automatisch omzet naar big endianness om de leesbaarheid te verhogen. Men kan hierin de identifier '000C0003', de data length code '1' en de data byte '53' terugvinden, zoals verwacht. Ook deze test verliep dus met succes.

```
DMA receive buffer:
  cpu_addr =   0xD907A800
  buffer size = 2048 bytes
  dma_handle = 0x1907A800
  bus_to_virt(dma_handle) = 0xD907A800
0000:  80 0C 00 03  10 00 00 00  53 00 00 00  00 00 00 00
```

Figuur 7.9: Receive buffer van de GRCAN core na ontvangst van een extended data frame

7.5.6 Extended data frame transmission

Ook de transmissie van extended data frames werd getest. Hiervoor werd de werkwijze uit paragraaf 7.5.2 op pagina 75 overgenomen. In figuur 7.10 ziet men dat het verzonden extended data frame correct toekomt in de PCAN testkaart. De test verliep dus succesvol.

Stopped | 17,7341 s | 0,00 % | Ring Buffer | Rx: 1 | Tx: 0 | Errors: 0

Time	Type	ID	DLC	Data
17,7341	Data	000C0003	1	53

Connected to PCAN-PCI (1 MBit/s) | Overruns: 0 | QXmtFull: 0

Figuur 7.10: De testkaart ontvangt het extended CAN data frame zonder problemen

7.5.7 Message filtering

Een laatste functionele test die uitgevoerd werd, is een message filtering test. Hierbij wordt een standaard CAN data frame uitgezonden door de PCAN testkaart. Dit data frame moet echter weggefilterd worden door de CAN interface module, aangezien de identifier niet overeen stemt met het acceptance mask dat opgegeven werd tijdens de initialisatie van de GRCAN core. Het frame mag dus niet terecht komen in de GRCAN receive buffer en het interruptregister moeten aanduiden dat een ontvangen frame weggefilterd werd ('Message filtered away during reception'). Verder mogen uiteraard geen errors optreden en mogen de read en write pointers niet veranderen tijdens deze test. Aan al deze voorwaarden werd voldaan, waardoor men kan besluiten dat ook deze test succesvol was.

7.6 Error injection tests

Om de werking van de CAN interface module volledig te testen, is het ook nodig om het gedrag te observeren in abnormale omstandigheden. Daarom worden zogenaamde 'error injection tests' uitgevoerd. Met de gebruikte PCAN kaart was het niet mogelijk om uitgebreide error injection tests uit te voeren, hiervoor zijn meer gespecialiseerde kaarten nodig waarmee bijvoorbeeld bit stuffing errors of CRC errors geïnjecteerd kunnen worden. Bovendien is dit ook niet de bedoeling van dit project.

7.6.1 Missing acknowledgement

In eerste instantie werd een missing acknowledgement test uitgevoerd, een test die overeenkomt met de data transmission test uit paragraaf 7.5.2 op pagina 75. Hier wordt echter de verbinding met de PCAN testkaart verbroken, waardoor geen acknowledgement gegenereerd kan worden.

Bij de controle van de status- en interruptregisters van de GRCAN core zouden nu wel errors moeten verschijnen. Bij de uitvoering van de test merkt men op dat 128 transmission errors opgetreden zijn en dat de GRCAN core zich in error passive mode bevindt. De read- en write pointers zijn wel gelijk gekomen, maar de transmissie wordt nog aangeduid als 'ongoing' aangezien het frame nog nergens correct aangekomen is.

Men kan dus besluiten dat de GRCAN core zich gedraagt volgens de verwachtingen. Aangezien de core in error passive mode wordt geplaatst vanaf 128 errors, voldoet hij aan de regels van foutbeperking die deel uitmaken van de data link laag, zoals besproken in paragraaf 2.3.2 op pagina 9.

7.6.2 Receive from wrong CAN bus

In een andere error injection test wordt het gedrag van de GRCAN core geobserveerd indien een standaard CAN data frame over de inactieve CAN bus verzonden wordt. Hiertoe wordt CAN bus A geactiveerd in de GRCAN core, terwijl de PCAN testkaart een data frame probeert te verzenden over CAN bus B.

De GRCAN core reageert hier volledig niet op en zendt dus enkel recessieve staten uit op CAN bus B. Dit zorgt er dus ook voor dat geen enkel register of buffer aangepast wordt tegen het einde van de test.

De PCAN testkaart ondervindt wel degelijk hinder van de inactieve bus aangezien recessieve staten uitgezonden worden, wat overeen komt met error frames en een een gebrek aan acknowledgements. Dit veroorzaakt zeer veel errors en en zware belasting van de inactieve bus, zoals weergegeven in figuur 7.11.

Dit gedrag is niet problematisch aangezien het de werking van de CAN interface module niet beïnvloedt.

Stopped | 5,4907 s | 43,60 % | Ring Buffer | Rx: 0 | Tx: 1 | Errors: 43604

Time	Type	ID	DLC	Data
5,4902	Error			Other Error, Tx, Acknowledge slot, RxErr=0, TxErr=128
5,4903	Error			Other Error, Tx, Acknowledge slot, RxErr=0, TxErr=128
5,4904	Error			Other Error, Tx, Acknowledge slot, RxErr=0, TxErr=128
5,4904	Error			Other Error, Tx, Acknowledge slot, RxErr=0, TxErr=128
5,4905	Error			Other Error, Tx, Acknowledge slot, RxErr=0, TxErr=128
5,4906	Error			Other Error, Tx, Acknowledge slot, RxErr=0, TxErr=128
5,4907	Error			Other Error, Tx, Acknowledge slot, RxErr=0, TxErr=128

Connected to PCAN-PCI (1 MBit/s) | Overruns: 0 | QXmtFull: 0 | BUSHEAVY

Figuur 7.11: Errors indien een transmissie plaatsvindt op een inactieve bus

7.6.3 Buffer overrun

De laatste error injection test die uitgevoerd werd, is een buffer overrun test. Hier wordt het gedrag van de GRCAN core getest wanneer de receive write pointer de read pointer inhaalt. Dit wordt bereikt door een hoeveelheid CAN frames naar de GRCAN core te sturen die groter is dan de capaciteit van de receive buffer. Als de read pointer niet verplaatst wordt, zal de write pointer deze inhalen, waardoor een buffer overrun plaatsvindt. In praktijk werd dit uitgevoerd door 127 standaard CAN data frames te versturen naar de GRCAN core die zelf een receive buffer heeft voor slechts 63 CAN frames.

Aan het einde van de test stelt men vast dat de eerste 63 CAN frames zich in de buffer bevinden en dat de write pointer wijst naar het laatste CAN frame in de buffer. De overige CAN frames zijn echter verloren. In de status- en interruptregisters wordt een overrun error gesignaleerd, net als een ‘successful reception of all messages possible to store in buffer’.

7.7 Performance tests

Als de normale functionaliteit van de GRCAN core bewezen werd, kan men nog steeds niet met 100 % zekerheid een correcte werking garanderen. Hiervoor moeten immers nog performance tests uitgevoerd worden, tests die de functies van de GRCAN core kritisch belasten. In de praktijk komt dit erop neer dat een grote hoeveelheid data in één keer overgezet wordt van of naar de GRCAN core.

Met de huidige software kunnen slechts beperkte performance tests uitgevoerd worden. De geimplementeerde transmit en receive buffers van de DMA driver hebben immers een maximale grootte van 2 KB, terwijl de GRCAN core buffers tot 1 MB ondersteunt. Bovendien zijn de testroutines niet te automatiseren. Het is dus niet mogelijk om in zeer korte tijd buffers weer op te vullen of read of write pointers te verplaatsen. Daarom zal in de hierop volgende tests optimaal gebruik gemaakt worden van de bestaande buffers, zodat toch reeds een beeld gevormd kan worden van de betrouwbaarheid van de GRCAN core.

7.7.1 Data reception

Om te beginnen werd een data reception performance test uitgevoerd, een test waarbij de PCAN testkaart een hoeveelheid data naar de GRCAN core zendt waarmee de volledige receive buffer gevuld wordt. In de praktijk komt dit neer op 127 standaard CAN frames om 2 KB te vullen (4 woorden moeten leeg blijven, vandaar 127 in plaats van 128 frames). Hiervoor werd een transfer list aangemaakt met behulp van de PCAN software, een lijst met 127 CAN messages. Bij het aanmaken van deze lijst werden verschillende patronen geïntegreerd om het gedrag te kunnen observeren indien een groot aantal opeenvolgende nullen, enen of afwisselende bits in de buffer geschreven wordt. Bovendien werd de maximale capaciteit van de CAN frames (8 bytes) benut in elk frame.

Na de ontvangst van alle frames werd de volledige receive buffer gecontroleerd op afwijkingen. Er werden echter geen onregelmatigheden vastgesteld. Ook de status- en interruptregisters wezen een succesvolle ontvangst aan. Bovendien werd een 'successful reception of all messages possible to store in buffer' aangeduid door het interruptregister.

Ook voor relatief grote hoeveelheden data op korte termijn lijkt de GRCAN core dus correct data te ontvangen en weg te schrijven naar een extern geheugen.

7.7.2 Data transmission

Tenslotte werd nog een data transmission performance test uitgevoerd, waarbij de volledige transmit buffer van de GRCAN core opgevuld wordt met 127 standaard CAN frames, om deze vervolgens in één keer naar de PCAN testkaart te sturen. Deze keer werd echter geen uitgebreid patroon voorzien in de CAN data. Er werden vooral nullen verzonden, aangezien de transmit buffer manueel ingevuld moest worden, met één commando per byte. Aan het einde van de test bleek alle data correct aangekomen in de PCAN testkaart. Verder gaven de status- en interruptregisters van de GRCAN core geen onregelmatigheden aan, waardoor ook deze test als succesvol beschouwd kan worden.

Bovendien kon aan de hand van deze test ook een data transfer rate berekend worden. De 127 CAN frames bevatten elk 8 bytes data en konden verzonden worden in 16 ms. De data transfer rate die uitgerekend wordt in formule 7.1 bedraagt 496 kbps. Gemiddeld zal de data transfer rate echter iets hoger liggen, afhankelijk van de identifier en de data die verzonden wordt, aangezien voor deze test vooral nullen uitgezonden werden. De bit stuffing bits zorgen er dan voor dat relatief veel overhead ontstaat. Algemeen kan men aannemen dat de data transfer rate ongeveer de helft bedraagt van de bitrate.

$$\text{Data transfer rate} = \frac{8 \cdot 8 \cdot 127}{1024 \cdot 0.016} = 496 \text{ kbps} \tag{7.1}$$

7.8 Samenvatting

Men kan dus besluiten dat alle hardware tests met succes afgerond werden. Het aantal tests dat uitgevoerd werd, lag veel hoger dan het vereiste aantal voor dit project. Men kan nu dus relatief zeker zijn dat het ontwerp van CAN interface FPGA en de CAN transceivers correct werkt. In tabel 7.1 op de pagina hierna wordt een samenvattend overzicht gegeven van de geteste functies. Hierin is duidelijk dat niet elke test uitgevoerd werd voor zowel standaard als extended frames en

dat niet elke test op de redundante CAN bus uitgevoerd werd. Dit kan echter op een eenvoudige manier gebeuren in een later stadium, als testroutines geautomatiseerd kunnen worden.

Voor de exacte testroutines en voor uitgebreide resultaten werd een extra DVD toegevoegd in bijlage D op pagina 103.

Tabel 7.1: Samenvatting van de uitgevoerde hardware tests

Test	Std./Ext.	Bus	PASS/FAIL
Registertoegang	-	-	PASS
Data reception	Std. & Ext.	A & B	PASS
Data transmission	Std. & Ext.	A & B	PASS
RTR reception	Std.	A	PASS
RTR transmission	Std.	A	PASS
Message filtering	Std.	A	PASS
Missing acknowledgement	Std.	A	PASS
Wrong bus reception	-	-	PASS
Buffer overrun	Std.	A	PASS
Performance data reception	Std.	A	PASS
Performance data transmission	Std.	A	PASS

8 Besluit

Voor deze masterproef werd onderzocht hoe een CAN bus geïntegreerd kan worden in een ADPMS, een boordcomputer voor kleine satellieten. Hiervoor werden in eerste instantie de fysische laag, de data link laag en de hogere protocols van de CAN bus bestudeerd. Deze karakteristieken werden dan getoetst aan de vereisten voor ruimtevaarttoepassingen. Hieruit kwam de noodzaak aan stralingsbestendige CAN controllers en CAN transceivers naar voor. De controllers kunnen bekomen worden door VHDL IP cores te integreren in stralingsbestendige FPGA's, terwijl de CAN transceivers best geïmplementeerd worden met behulp van stralingsbestendige RS-485 transceivers. Men dient bovendien te zorgen voor een redundante CAN bus, waarvoor een schakelmechanisme en een vorm van redundancy management vereist is. Het schakelmechanisme dient geïmplementeerd te worden in hardware, maar het redundancy management wordt voorzien in het hoger protocol, dat voor ruimtevaarttoepassingen best gebaseerd wordt op het CANopen protocol. In dit hoger protocol kunnen bovendien aanvullende diensten geïmplementeerd worden voor het verzenden van grote hoeveelheden data en voor tijdsdistributie binnen satellieten. Een diepere uitwerking van deze hogere protocols maakte echter geen deel uit van de opdracht.

Voor de integratie van de CAN bus in een ADPMS werd gebruik gemaakt van een bestaande MIL1553b interface module, een cPCI module die op een backplane geplugd kan worden. Aanpassingen in de hardware van deze kaart zorgen ervoor dat ze gebruikt kan worden als CAN interface module. Hiervoor worden stralingsbestendige CAN transceivers gebruikt, net als een FPGA ontwerp van een CAN controller. Aangezien het de bedoeling is dat in een later stadium zowel de CAN interface als de MIL1553b interface geïntegreerd kunnen worden in eenzelfde module, werd gekozen voor een FPGA ontwerp dat nauw aanleunt bij het FPGA ontwerp van de MIL1553b interface module. Er werd daarom gebruik gemaakt van verschillende IP cores van QinetiQ Space, verbonden via een on-chip AHB bus. Voor de integratie van de CAN bus werd de GRCAN core van Gaisler Research als meest geschikt beschouwd. Deze IP core werd dan ook geïntegreerd in het ontwerp.

Vervolgens werd een volledige test bench geschreven in VHDL, een structuur die toelaat om alle

functies van het FPGA ontwerp te simuleren. In deze test bench werd een simulatie CPU en een extern geheugen geïntegreerd, samen met een model van de CAN interface module en verschillende CAN testnodes. Op deze manier werd dus eigenlijk een volledige testopstelling beschreven in VHDL. De testnodes werden volledig zelf gemodelleerd en getest in een aparte test bench alvorens ze te integreren in de algemene test bench. Ze zijn in staat om standaard CAN frames te zenden en ontvangen in de simulaties en voorzien bovendien een doorgedreven vorm van foutcontrole en -signalisatie.

Na de constructie van de VHDL test bench werden assembler simulatiescripts geschreven voor de simulatie CPU. Met deze testscripts werden registertoegang en data transfer tests uitgevoerd, goed om de basisfunctionaliteit van de CAN interface FPGA te testen. Bij de uitvoering van de testscripts werd een bug ontdekt in de AHB master van de GRCAN core van Gaisler Research. Na communicatie met Gaisler Research werd een nieuwe versie van deze IP core uitgebracht, waardoor alle simulaties met succes afgerond konden worden. Met deze werkwijze werd een volledig geautomatiseerd testsysteem opgebouwd, waardoor herhaalbaarheid van de tests verzekerd werd. Bovendien werd elk onderdeel van de test bench zowel apart getest als in het geheel. Op deze manier werd de testprocedure uitgevoerd volgens de richtlijnen van het ECSS.

In een laatste stap werd het FPGA ontwerp geïntegreerd in een prototype van de MIL1553b interface module voor het uitvoeren van hardware tests. Hiervoor werd opeenvolgend een synthese, place & route en programmering van het FPGA ontwerp uitgevoerd, waardoor de MIL1553b interface module omgevormd werd tot CAN interface module. De hierop volgende elektrische tests waren positief, waardoor de CAN interface module geïntegreerd kon worden in een hardware testopstelling. Deze testopstelling bestond uit twee computers die vanop afstand bediend werden. De ene bevatte een commerciële testkaart terwijl de andere de CAN interface module bevatte. Voor het uitvoeren van de hardware tests werd gebruik gemaakt van commerciële CAN testsoftware en testsoftware van QinetiQ Space. Met deze applicaties is het echter niet mogelijk om de tests te automatiseren met behulp van test scripts, waardoor hier niet de herhaalbaarheid bereikt kon worden die het ECSS voorschrijft. Het hoofddoel van de hardware tests werd echter wel behaald, er werd in de hardware testopstelling namelijk een eenvoudige CAN transfer opgezet in beide richtingen. De tests konden echter significant uitgebreid worden tot een hele reeks functionele tests, error injection tests en performance tests die allen met succes afgerond werden.

Men kan dus besluiten dat het doel van deze masterproef ruimschoots behaald werd. De integratie van een CAN bus in ruimtevaarttoepassingen werd bestudeerd en in de praktijk getest met behulp van de GRCAN core van Gaisler Research. Hiervoor werd een simulatie test bench geconstrueerd die ook voor toekomstige tests nog gebruikt kan worden. Ook de CAN testnodes die hiervoor ontworpen werden, zijn geschikt voor verdere tests. Eventueel kan hierin nog ondersteuning voor extended CAN frames geïntegreerd worden, zodat nog uitgebreidere tests uitgevoerd kunnen worden. Een code coverage test toonde aan dat in de simulaties slechts 79.9 % van de statements en 44.6 % van de branches getest werden, wat bewijst dat verdere functionele tests, error injection tests en performance tests nodig zijn in simulaties voordat het ontwerp echt klaar is voor gebruik. Deze uitgebreide tests konden echter wel grotendeels uitgevoerd worden in een hardware testopstelling, waardoor bewezen werd dat de GRCAN core geschikt is voor integratie in een ADPMS.

Voor toekomstige hardware tests is het noodzakelijk dat de ADPMS PCI client software en DMA driver van QinetiQ Space aangepast worden om automatisering en herhaalbaarheid te bekomen,

zoals het ECSS voorschrijft. Hiervoor is ondersteuning voor bijvoorbeeld *.tcl* scripts vereist in plaats van een manuele bediening. Bovendien is het dan beter om de ARINC-825 kaart van AIM te gebruiken als testkaart, zodat toekomstige hardware tests nog dieper uitgewerkt kunnen worden. In een laatste fase van de hardware tests dient het ontwerp getest te worden in een ADPMS. Er wordt dan gebruik gemaakt van de LEON-2FT CPU en het RTEMS operating system. Voor dit operating system moeten dan ook de geschikte software drivers ontworpen worden, waarin het hoger protocol (een uitgebreide versie van CANopen) geïntegreerd wordt. Het prototype stadium is voorbij als al deze tests met succes afgerond worden. Er kan dan een flight module ontworpen worden die, na de nodige tests, klaar is voor een ruimtemissie.

Referenties

[1] "ISO 11898-1:2003 - Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling," 2003.

[2] "BS EN 50325-4: Industrial communication subsystem based on ISO 11898 (CAN) for controller-device interfaces. CANopen," 2002.

[3] "MIL-STD-1553b: Aircraft internal time division command/response multiplex data bus," Sept. 1979.

[4] "ESA OBDH TTC-B-01: Spacecraft Data Handling Interface Standards," Sept. 1978.

[5] F. Tortosa Lopez, C. Plummer, P. Roos, S. L., and B. Storni, "The can bus in spacecraft on board applications," in *Data systems in aerospace, 2004. Proceedings.*, July 2004.

[6] Anon., "Recommendations for CAN bus in spacecraft onboard applications, Draft 2.1," tech. rep., ECSS, 2005.

[7] A. Woodroffe and P. Madle, "Application and experience of can as a low cost obdh bus system," in *Data systems in aerospace, 2004. Proceedings.*, July 2004.

[8] M. Khurram and S. Zaidi, "Can as a spacecraft communication bus in leo satellite mission," in *Recent Advances in Space Technologies, 2005. RAST 2005. Proceedings of 2nd International Conference on*, pp. 432 – 437, june 2005.

[9] A. Tai, S. Chau, and L. Alkalai, "Cots-based fault tolerance in deep space: Qualitative and quantitative analyses of a bus network architecture," in *High-Assurance Systems Engineering, 1999. Proceedings. 4th IEEE International Symposium on*, pp. 97 –104, 1999.

[10] J. Yang, T. Zhang, J. Song, H. Sun, G. Shi, and Y. Chen, "Redundant design of a can bus testing and communication system for space robot arm," in *Control, Automation, Robotics and Vision, 2008. ICARCV 2008. 10th International Conference on*, pp. 1894 –1898, dec. 2008.

[11] S. Chau, "Experience of using cots components for deep space missions," in *High-Assurance Systems Engineering, 1999. Proceedings. 4th IEEE International Symposium on*, p. 116, 1999.

[12] A. Dr. Emrich, "CAN application in avionics," tech. rep., Omnisys Instruments, ESTEC, 2001.

[13] Anon., "AT7908E: CAN Controller for Space Application," tech. rep., Atmel, 2004.

[14] Anon., "CANTRAN, CAN bus transceiver for space applications, irradiation test report document," tech. rep., Aurelia Microelettronica S.p.A., 2003.

[15] "ANSI/TIA 485-A1998, Electrical Characteristics of Generators and Receivers for Use in Balanced Digital Multipoint Systems," Mar. 2003.

[16] T. Kugelstadt, "RS-485: Passive failsafe for an idle bus," tech. rep., Texas Instruments, 2009.

[17] ARM Limited, *AMBA Specification v2.0*, 1999.

[18] E. W. Group, "Space product assurance: ASIC and FPGA development," tech. rep., ESA-ESTEC, 2008.

[19] R. B. GmbH, "CAN specification Version 2.0," tech. rep., Bosch, 1991.

[20] A. Gaisler, *GRLIB IP Core User Manual*, 2010.

Bibliografie

Anon. Industrial Automation Using the CAN BUS Platform White Paper. Technical report, Texas Instruments, 2003.

Anon. CAN message frames. Technical report, Microchip, 2005.

Anon. Controller Area Network. http://www.can-cia.org/, WWW.

Anon. Radiation hardening. http://en.wikipedia.org/wiki/Radiation_hardening, WWW.

S. Corrigan. Controller Area Network Physical Layer Requirements. Technical report, Texas Instruments, 2008.

S. Corrigan. Introduction to the Controller Area Network (CAN). Technical report, Texas Instruments, 2008.

R. De Vos. ADPMS s_processor VHDL beh design document. Technical report, QinetiQ Space, 2010.

R. De Vos. IXV OBC MIM FPGA verification plan. Technical report, QinetiQ Space, 2010.

R. De Vos. IXV OBC MIM FPGA verification report. Technical report, QinetiQ Space, 2010.

ECSS_Q_ST_60_02 Working Group. Space product assurance: ASIC and FPGA development. Technical report, ESA-ESTEC, 2008.

Klaus Janschek and Annerose Braune. Application of industrial can bus technology for leo-satellites. *Acta Astronautica*, 46(2-6):313 – 317, 2000. 2nd IAA International Symposium on Small Satellites for Earth Observation.

L. Ottevaere. IXV OBC MIM FPGA development plan. Technical report, QinetiQ Space, 2010.

C. Plummer, P. Roos, and Stagnaro L. Can bus as a spacecraft onboard bus. In *Data systems in aerospace, 2003. Proceedings.*, pages 51 – 62, June 2003.

P. Richards. A CAN Physical Layer Discussion. Technical report, Microchip, 2002.

J. Scarpulla and A. Yarbrough. What could go wrong? The effects of ionizing radiation on space electronics. http://www.aero.org/publications/crosslink/summer2003/03.html, 2003.

A. Woodroffe and P. Madle. Application and experience of CAN as a low cost OBDH bus system. Technical report, Surrey Sattellite Technologies LTD, University of Surrey, 2004.

Beschrijving van deze masterproef in de vorm van een wetenschappelijk artikel

Integration of a CAN bus in an Onboard Computer for Space Applications

Stijn Wielandt
Katholieke Hogeschool Sint-Lieven, Department of Industrial Engineering
Gebroeders Desmetstraat 1, 9000 Ghent, Belgium
stijn.wielandt@kahosl.be

***Abstract*—The integration of CAN bus networks in space applications has gained interest over the past years because of its high reliability and cost effectiveness. The bus has already been integrated in some low earth orbit satellites with commercial off-the-shelf components, but the harsh environment in deep space demands a more reliable solution. That is why a radiation hardened design of a CAN transceiver and a CAN controller is being investigated. The transceiver is constructed with a radhard RS-485 transceiver and the CAN controller is implemented in a radhard FPGA by means of VHDL IP cores. For this paper, a CAN IP core from Gaisler Research is selected and implemented in an FPGA. Eventually, this design was successfully tested in simulations of a VHDL test bench as well as in a hardware test bench. These tests proved the fitness of the CAN controller for use in deep space applications.**

I. Introduction

OVER the past two decades, CAN bus[1] has been widely adopted in automotive industry and automation. In the last couple of years there has been a growing interest in this bus for space applications. The low development cost, reliability, priority based bus access and presence of commercial hardware make CAN bus a good alternative to classic spacecraft communication busses, such as MIL1553b[2] and ESA OBDH[3]. The bus has already been used in recent *low earth orbit* (LEO) missions, but an implementation for deep space applications has never been established. In this context reliability and radiation hardness are major concerns.

This paper describes the implementation of a CAN bus in an *advanced data and power management system* (ADPMS), an onboard computer for small satellites. A possible approach involves the use of commercial off-the-shelf (COTS) components, an approach that has been adopted in LEO satellites. This method is without doubt the most cost effective, but doesn't meet all requirements such as fault tolerance and reliability. Another method that is more expensive implies the use of space qualified CAN components. However, these components are still in development and not ready for deep space applications. The third option includes the design of custom built space qualified components. Therefore, a radiation hardened implementation of a CAN transceiver needs to be designed, as well as a radhard CAN controller. Because this last option is the only one that meets the strict demands of deep space missions, it was studied, elaborated and tested in detail. Therefor a CAN controller was designed in VHDL for implementation in a radiation hardened FPGA. This FPGA drives a cPCI module which can be plugged into an ADPMS. The design was tested in simulations as well as in a hardware test bench.

This paper is organized as follows. First a general description of the CAN protocol is given, together with the space requirements such as fault tolerance and reliability. Afterwards the possible solutions are considered, ranging from COTS components to custom built components. This last option is completely elaborated. A CAN transceiver is put together with a radhard RS-485 transceiver and a CAN controller is designed with the help of IP cores and an FPGA.

Next (Section III), the design is tested by means of computer simulations of a VHDL test bench. After these tests, the design can be tested in a hardware test bench. The tests are executed as much as possible in accordance with the ASIC and FPGA development guidelines of the European Cooperation for Space Standardization (ECSS)[4]. This enables us to assess the success of this project (Section IV), in order to facilitate future implementations.

II. Theory

A. CAN bus in space

The CAN bus was developed in the late 80's by Robert Bosch GmbH[5] for use in the automobile industry and was subsequently standardized as ISO 11898. This protocol only defines a part of the physical layer and the data link layer. The application layer can be implemented in various ways, however CANopen[6] has been pointed to be the most appropriate protocol in space applications, according to ECSS[7]. The high reliability of the CAN bus, together with reduced costs and a multimaster structure, make this bus an ideal medium for onboard communication in space applications.

In spite of the proven reliability in terrestrial applications in harsh environments, a CAN bus implementation according to the ISO 11898 does not meet space standards. This is why ECSS published it's recommendations for implementing a CAN bus in space systems[7]. The CAN bus has already been implemented in several LEO satellites, which operate at an altitude of 100 km to 1000 km. At these altitudes, the radiation levels are still relatively low and the environmental characteristics such as temperature and EMC are comparable with the situation in a car's engine block. This is why COTS components are usable in LEO missions[8][9][10]. Since the

use of COTS components for CAN bus implementations in LEO missions, no failures have been encountered. However, a redundant bus should be foreseen[11]. In the case of a hot redundant bus, both CAN busses actively participate in transfers. A cold redundant bus is easier to implement but can not recover lost messages, since only one bus is active at a time. The implementation of a redundant bus requires a form of redundancy management, an option that is not foreseen in the ISO 11898 or CANopen standard.

In deep space applications, the use of COTS components is not appropriate because of the high temperature range and radiation levels. They don't provide the fault tolerance that is needed because it would raise development and production costs[12]. Besides, these components don't offer the long term survivability that is required for deep space missions. In order to resolve these issues while maintaining low development costs, COTS IPs can be transferred to space qualified ASICs[13].

It's also possible to use space qualified instead of COTS components, but in this domain the choice of CAN components is very limited. There is only one radhard CAN controller available, the AT7908E from Atmel. But this controller doesn't support redundant CAN busses and it's not compatible with the onboard PCI bus of the ADPMS, so the only solution that is left, is the implementation of a CAN controller in a radhard FPGA, the Actel RTAX2000S/SL.

The CAN transceiver can be implemented in various ways. The most logical option involves the use of CANTRAN, a radiation hardened CAN controller from Aurelia Microelettronica. But since this design is currently only available in DIL28 package, it's not usable for deep space missions because of its size and weight. A CAN transceiver built of discrete components also isn't usable because of these reasons. That's why an implementation with a radiation hardened RS-485 transceiver[14] is the best option.

B. Hardware implementation

1) CAN transceiver: The CAN transceiver is based on a radhard RS-485 transceiver with the *drive enable* input connected to the CAN drive output from the CAN controller, as described in [14]. At the CAN bus side of the transceiver, a biasing network is added in order to terminate the network with the characteristic impedance (120 Ω) and to regulate voltage levels on the CAN bus in undriven states.

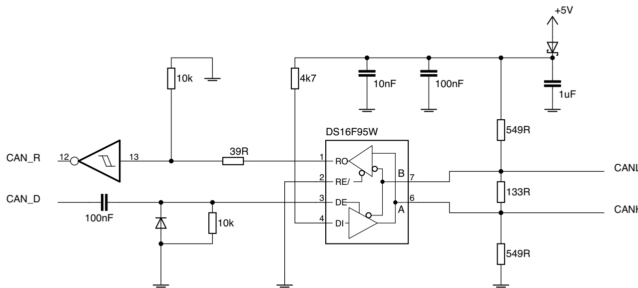


Fig. 1. Radhard CAN transceiver

The values of the resistors in the biasing network are calculated as follows. At first the differential voltage on the CAN bus is calculated in equation 1.

$$V_{CAN_diff} = \frac{V_{cc}}{\frac{1}{R_1} + \frac{1}{R_{in}} + 2 \cdot \left(\frac{1}{R_{term_1}} + \frac{1}{R_{term_2}} \right)} \tag{1}$$

The RS-485 standard[15] demands a minimum load of 375 Ω on both outputs of the transceiver, which is represented by equation 2.

$$\frac{1}{R_1} + \frac{1}{R_{in}} \geq \frac{1}{375} \tag{2}$$

The input impedance of the transceiver is AC coupled and fixed at 120 Ω, the characteristic impedance of the CAN bus network. This is represented by equation 3.

$$\frac{1}{2 \cdot R_1} + \frac{1}{R_{term_1}} = \frac{1}{120} \tag{3}$$

A substitution of equation 2 and 3 in equation 1, together with a 120 Ω value of R_{term_2}, yields equation 4.

$$R_1 = \left(\frac{V_{cc}}{V_{CAN_diff}} + 1 \right) \cdot \frac{250}{9} \tag{4}$$

This equation enables the calculation of the bias resistors. The minimum differential input voltage of the RS-485 transceiver is 200 mV. Together with a noise margin of 50 mV and a V_{cc} voltage of 4.7 V, this equation results in bias resistors of 550 Ω, practically realizable with 549 Ω resistors.

The value of the termination resistor can then be calculated with equation 3:

$$R_{term_1} = \frac{1}{\frac{1}{2 \cdot 549} + \frac{1}{120}} = 134.72\ \Omega \tag{5}$$

This value can be realized with a 133 Ω resistor.

2) CAN controller: As mentioned in section II-A, the CAN controller is implemented in a radhard FPGA with the help of VHDL IP cores, interconnected with an on-chip AMBA AHB bus. The most important IP cores are the PCI bridge from QinetiQ Space NV for communication with the ADPMS and the CAN core to implement CAN bus functionality. From all available CAN cores, the GRCAN IP core from Gaisler Research seemed the best fit for integration in the ADPMS. Unlike other CAN cores, this core provides an AHB interface, direct memory access (DMA) and a cold redundant CAN bus. The only disadvantage is the use of an APB bus, which can be solved with an APB to AHB translate core.

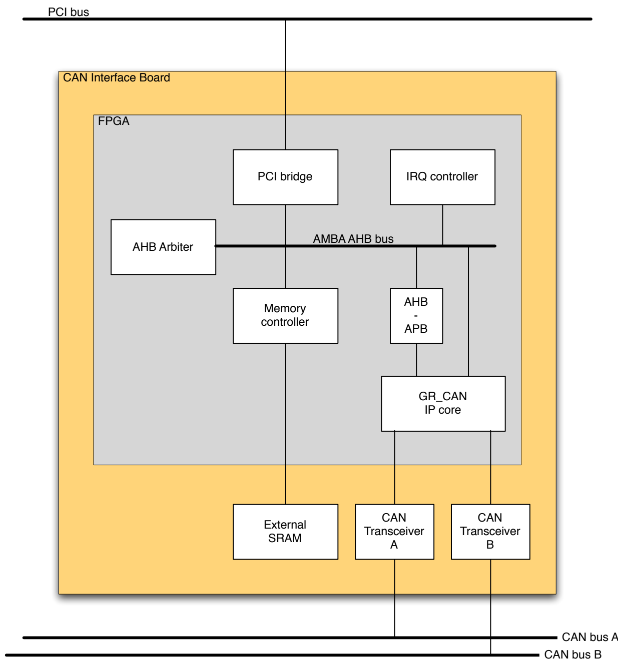


Fig. 2. CAN interface module with radhard CAN controller

III. Experiments

One of the most important steps in the development of space applications, is testing. All tests are being executed as much as possible in accordance with [4]. This means that the complete design is first tested in simulations and afterwards in a hardware test bench.

A. Simulations

At first, a complete test bench is created in VHDL code. Simulations of this test bench have to prove the correct operation of the FPGA design. The VHDL test bench is built of several simulation models, such as a simulation CPU (SCPU), an SRAM memory, some CAN test nodes and a model of the CAN interface board, as can be seen in figure 3. These models are only intended for simulation purposes and contain no synthesizable code. The SCPU from QinetiQ Space NV enables us to execute read and write commands in registers or memory. It is connected with other simulation models through a PCI bus. The executable assembler commands are loaded from a test script at the start of a simulation. The SRAM model from QinetiQ Space NV is meant for the setup of receive and transmit buffers of the CAN controller. The design of the CAN controller –which is being tested– is implemented in a simulation model of the CAN interface cPCI module that also contains simulation models of two CAN transceivers. These transceivers connect the CAN interface module with two CAN networks in the test bench, a nominal and a redundant one. The CAN networks are connected to several CAN test nodes, simulation models that can send CAN frames or check received frames on one of the CAN busses. The test nodes were constructed with a high level of autonomy, in order to enable automatic CAN frame construction and control of received frames.

As demanded in [4], the operation of all simulation models is tested in a separate test bench before implementing them in the general VHDL test bench of the CAN controller.

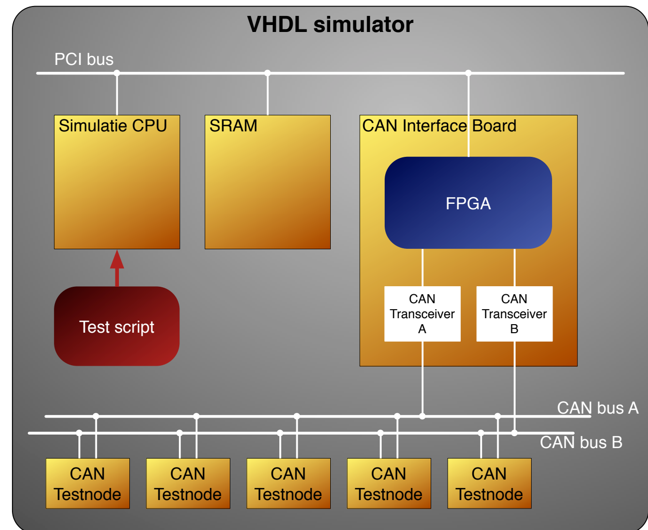


Fig. 3. VHDL test bench

After writing a test bench, the basic functionality of the CAN controller is tested with assembler test scripts for the SCPU. A first test proves the correct read and write capabilities of every register of the CAN IP core. The subsequent data reception test was also successful, which proves that CAN messages can be received correctly and stored in a buffer in external memory. The last test however revealed a problem with the CAN IP core from Gaisler Research. A data transmission could not be initiated by the CAN IP core because of a bug in the AHB part of the core. A bug fix in version 13 of this IP core solved the issue, resulting in a successful data transmission test. A code coverage test showed that 79.9 % of the statements and 44.6 % of the branches were covered during the simulations. These results are far below the requirements of [4], which demand a coverage of respectively 90 % and 85 %. As a consequence, more testing will have to be performed before a final implementation for space applications can be manufactured, but a proof of concept has been delivered.

B. Hardware tests

After successful simulations, the design can be tested thoroughly in a hardware test bench. Therefor the design of the CAN controller is programmed in an FPGA, which is implemented in a prototype of the CAN interface module. Apart from that, the radiation hardened CAN transceivers are implemented in the same prototype. Thereafter, the prototype is implemented in a test system built on a cPCI backplane. This test system is connected to another test system through a double CAN interface. The second test system contains a commercial CAN test card and runs commercial monitor and test software. This setup is visualized in figure 4.

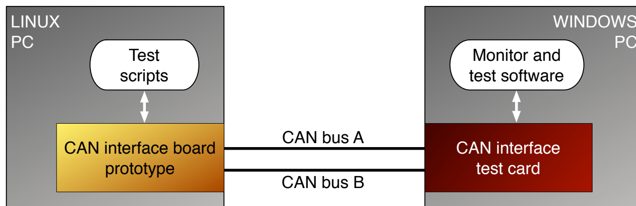


Fig. 4. Hardware test bench

The hardware tests can be divided into three categories. The first tests verify the functional behaviour of the CAN controller. These functional tests include a register access test and reception and transmission tests for different sorts of CAN frames, such as standard and extended remote transfer request (RTR) and data frames. The difference between standard and extended frames lies in the length of the identifier field of the CAN frame. RTR frames are transmitted as a request for data, whereas a data frame contains 0 to 8 bytes of information.

Consequently, several error injection tests are performed in order to verify the behaviour of the CAN controller in abnormal circumstances. First the behaviour of the controller is observed when no acknowledgement is being generated by the receiving node. In the next test a frame reception on the inactive (redundant) bus is tested and finally a buffer overrun test is executed. In this last test, an amount of data larger than the receive buffer is transmitted to the CAN controller.

Finally the design of the CAN controller is put through two performance tests. These tests verify the operation of the CAN controller in case an amount of data equal to the size of the receive or transmit buffer is being received or transmitted at the maximum bitrate of 1 Mbps.

All hardware tests finished successfully. The functional tests proved the correct operation of the CAN controller in normal circumstances, whereas the error injection tests proved a correct error handling. Also the performance tests didn't result in any anomalies and a data transmission rate of 496 kbps could be measured, a number that can deviate slightly, depending on the data content.

IV. Conclusion and Future Work

An implementation of a CAN bus for space applications can be achieved in different ways, but for deep space applications only one option is appropriate. Since COTS components don't offer the required reliability and since space qualified CAN components are still to be finalized, the design of custom components is the only option left.

The CAN transceiver can be constructed with an RS-485 transceiver and a biasing network. The CAN controller can be programmed into a radhard FPGA with the help of VHDL IP cores.

As a test, the GRCAN IP core from Gaisler Research was integrated in a CAN controller design. This design was first tested in simulations and consequently in a hardware test bench. The tests revealed a bug in the GRCAN IP core, but as of version 13 of the core, all test requirements were met. This proves the GRCAN core's feasibility for use in space applications, however further testing needs to be performed, particularly in simulations. These tests only covered 79.9 % of the statements and 44.6 % of the branches instead of the required 90 % and 85 %.

Acknowledgment

This research was made possible thanks to the structural and intellectual supplies of QinetiQ Space NV.

References

[1] "ISO 11898-1:2003 - Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling," 2003.
[2] "MIL-STD-1553b: Aircraft internal time division command/response multiplex data bus," Sep. 1979.
[3] "ESA OBDH TTC-B-01: Spacecraft Data Handling Interface Standards," Sep. 1978.
[4] E. W. Group, "Space product assurance: ASIC and FPGA development," ESA-ESTEC, Tech. Rep., 2008.
[5] R. Bosch, "CAN specification version 2.0," Sep. 1991.
[6] "BS EN 50325-4: Industrial communication subsystem based on ISO 11898 (CAN) for controller-device interfaces. CANopen," 2002.
[7] Onboard bus LAN working group, "Recommendations for CAN bus in spacecraft onboard applications, Draft 2.1," ECSS, Tech. Rep., 2005.
[8] A. Woodroffe and P. Madle, "Application and experience of can as a low cost obdh bus system," in *Data systems in aerospace, 2004. Proceedings.*, July 2004.
[9] K. Janschek and A. Braune, "Application of industrial can bus technology for leo-satellites," *Acta Astronautica*, vol. 46, no. 2-6, pp. 313 – 317, 2000, 2nd IAA International Symposium on Small Satellites for Earth Observation.
[10] M. Khurram and S. Zaidi, "Can as a spacecraft communication bus in leo satellite mission," in *Recent Advances in Space Technologies, 2005. RAST 2005. Proceedings of 2nd International Conference on*, june 2005, pp. 432 – 437.
[11] J. Yang, T. Zhang, J. Song, H. Sun, G. Shi, and Y. Chen, "Redundant design of a can bus testing and communication system for space robot arm," in *Control, Automation, Robotics and Vision, 2008. ICARCV 2008. 10th International Conference on*, dec. 2008, pp. 1894 –1898.
[12] A. Tai, S. Chau, and L. Alkalai, "Cots-based fault tolerance in deep space: Qualitative and quantitative analyses of a bus network architecture," in *High-Assurance Systems Engineering, 1999. Proceedings. 4th IEEE International Symposium on*, 1999, pp. 97 –104.
[13] S. Chau, "Experience of using cots components for deep space missions," in *High-Assurance Systems Engineering, 1999. Proceedings. 4th IEEE International Symposium on*, 1999, p. 116.
[14] A. Dr. Emrich, "CAN application in avionics," Omnisys Instruments, ESTEC, Tech. Rep., 2001.
[15] "ANSI/TIA 485-A1998, Electrical Characteristics of Generators and Receivers for Use in Balanced Digital Multipoint Systems," Mar. 2003.
[16] C. Plummer, P. Roos, and S. L., "Can bus as a spacecraft onboard bus," in *Data systems in aerospace, 2003. Proceedings.*, June 2003, pp. 51 – 62.
[17] F. Tortosa Lopez, C. Plummer, P. Roos, S. L., and B. Storni, "The can bus in spacecraft on board applications," in *Data systems in aerospace, 2004. Proceedings.*, July 2004.

B

Resultaat simulaties

B.1 Registertoegang

```
# Initializing SRAM with zero ...Initializing SRAM with zero ...<0 ps> <SCPU ><0 ps>
# <0 ps> <SCPU ><0 ps>
# <0 ps> <SCPU ><0 ps> **************************
# <0 ps> <SCPU ><0 ps> **                      **
# <0 ps> <SCPU ><0 ps> **     S_PROCESSOR:     **
# <0 ps> <SCPU ><0 ps> **                      **
# <0 ps> <SCPU ><0 ps> ** - Version: 6.0       **
# <0 ps> <SCPU ><0 ps> ** - Author:  R. De Vos **
# <0 ps> <SCPU ><0 ps> ** - Date:    20.05.2010 **
# <0 ps> <SCPU ><0 ps> **                      **
# <0 ps> <SCPU ><0 ps> **************************
# <0 ps> <SCPU ><0 ps>
# <0 ps> <SCPU ><0 ps>
# grcan0: GR CAN Controller rev 0, irq 0
# <525000 ps> <SCPU ><525000 ps> ---- TEST STARTED ----
# <525000 ps> <SCPU ><525000 ps>
# <525000 ps> <SCPU ><525000 ps>  cim0100_register_access_test.scp
# <525000 ps> <SCPU ><525000 ps>
# <3525000 ps> <SCPU ><3525000 ps> ---------------------------------------------
# <3525000 ps> <SCPU ><3525000 ps> -- Configure the defaults of the cim module. --
# <3525000 ps> <SCPU ><3525000 ps> ---------------------------------------------
# ...
# <26625000 ps> <SCPU ><26625000 ps>  Initialise command register of CAN PCI-bridge.
# <26625000 ps> <SCPU ><26625000 ps>    COMMAND = 0x0146
# <26625000 ps> <SCPU ><26625000 ps>
# <27015000 ps> <SCPU ><27015000 ps> write D[0x00000146]; A[0x01000004] (0)[297]
# <27405000 ps> <SCPU ><27405000 ps> read D[0x02000146]; A[0x01000004] (0)[298]
# <27405000 ps> <SCPU ><27405000 ps>
# <27405000 ps> <SCPU ><27405000 ps> ------------------------------------------------------------
# <27405000 ps> <SCPU ><27405000 ps> -- Begin CAN IP core register access tests
# <27405000 ps> <SCPU ><27405000 ps> ------------------------------------------------------------
# <27405000 ps> <SCPU ><27405000 ps>
# <27405000 ps> <SCPU ><27405000 ps> ------------------------------------------------------------
# <27405000 ps> <SCPU ><27405000 ps> -- Check the control register for write/read access.
# <27405000 ps> <SCPU ><27405000 ps> ------------------------------------------------------------
# <28065000 ps> <SCPU ><28065000 ps> read D[0x00000000]; A[0xB0000008] (0)[315]
# <28695000 ps> <SCPU ><28695000 ps> write D[0xFFFFFFFD]; A[0xB0000008] (0)[317]
# <29355000 ps> <SCPU ><29355000 ps> read D[0x00000001]; A[0xB0000008] (0)[318]
# <29985000 ps> <SCPU ><29985000 ps> write D[0x0000000F]; A[0xB0000008] (0)[320]
# <30645000 ps> <SCPU ><30645000 ps> read D[0x00000000]; A[0xB0000008] (0)[321]
# <30645000 ps> <SCPU ><30645000 ps> ------------------------------------------------------------
# <30645000 ps> <SCPU ><30645000 ps> -- Check configuration register for  write/read access.
# <30645000 ps> <SCPU ><30645000 ps> ------------------------------------------------------------
# <31305000 ps> <SCPU ><31305000 ps> read D[0x00000000]; A[0xB0000000] (0)[360]
# <31935000 ps> <SCPU ><31935000 ps> write D[0xFFFFFFFF]; A[0xB0000000] (0)[363]
# <32595000 ps> <SCPU ><32595000 ps> read D[0xFFFF731F]; A[0xB0000000] (0)[364]
# <32595000 ps> <SCPU ><32595000 ps> ------------------------------------------------------------
# <32595000 ps> <SCPU ><32595000 ps> -- Check Status register for read access.
# <32595000 ps> <SCPU ><32595000 ps> ------------------------------------------------------------
# <33255000 ps> <SCPU ><33255000 ps> read D[0x00000000]; A[0xB0000004] (0)[372]
# <33255000 ps> <SCPU ><33255000 ps> ------------------------------------------------------------
```

```
# <33255000 ps> <SCPU > <33255000 ps>  -- Check SYNC Mask Filter register for  write/read access.
# <33255000 ps> <SCPU > <33255000 ps>  ----------------------------------------------------------------
# <33915000 ps> <SCPU > <33915000 ps> read D[0x1FFFFFFF]; A[0xB0000018] (0)[380]
# <34545000 ps> <SCPU > <34545000 ps> write D[0xE0000000]; A[0xB0000018] (0)[383]
# <35205000 ps> <SCPU > <35205000 ps> read D[0x00000000]; A[0xB0000018] (0)[384]
# <35205000 ps> <SCPU > <35205000 ps>  ----------------------------------------------------------------
# <35205000 ps> <SCPU > <35205000 ps>  -- Check SYNC Code Filter register for  write/read access.
# <35205000 ps> <SCPU > <35205000 ps>  ----------------------------------------------------------------
# <35865000 ps> <SCPU > <35865000 ps> read D[0x00000000]; A[0xB000001C] (0)[392]
# <36495000 ps> <SCPU > <36495000 ps> write D[0xFFFFFFFF]; A[0xB000001C] (0)[395]
# <37155000 ps> <SCPU > <37155000 ps> read D[0x1FFFFFFF]; A[0xB000001C] (0)[396]
# <37155000 ps> <SCPU > <37155000 ps>  ----------------------------------------------------------------
# <37155000 ps> <SCPU > <37155000 ps>  -- Check Transmit Channel Control register for  write/read access.
# <37155000 ps> <SCPU > <37155000 ps>  ----------------------------------------------------------------
# <37815000 ps> <SCPU > <37815000 ps> read D[0x00000000]; A[0xB0000200] (0)[404]
# <38445000 ps> <SCPU > <38445000 ps> write D[0xFFFFFFF4]; A[0xB0000200] (0)[407]
# <39105000 ps> <SCPU > <39105000 ps> read D[0x00000004]; A[0xB0000200] (0)[408]
# <39735000 ps> <SCPU > <39735000 ps> write D[0xFFFFFFF1]; A[0xB0000200] (0)[410]
# <40395000 ps> <SCPU > <40395000 ps> read D[0x00000001]; A[0xB0000200] (0)[411]
# <40395000 ps> <SCPU > <40395000 ps>  ----------------------------------------------------------------
# <40395000 ps> <SCPU > <40395000 ps>  -- Check Transmit Channel Address register for  write/read access.
# <40395000 ps> <SCPU > <40395000 ps>  ----------------------------------------------------------------
# <41055000 ps> <SCPU > <41055000 ps> read D[0x00000000]; A[0xB0000204] (0)[419]
# <41685000 ps> <SCPU > <41685000 ps> write D[0xFFFFFFFF]; A[0xB0000204] (0)[422]
# <42345000 ps> <SCPU > <42345000 ps> read D[0xFFFFFC00]; A[0xB0000204] (0)[423]
# <42345000 ps> <SCPU > <42345000 ps>  ----------------------------------------------------------------
# <42345000 ps> <SCPU > <42345000 ps>  -- Check Transmit Channel Size register for  write/read access.
# <42345000 ps> <SCPU > <42345000 ps>  ----------------------------------------------------------------
# <43005000 ps> <SCPU > <43005000 ps> read D[0x00000000]; A[0xB0000208] (0)[431]
# <43635000 ps> <SCPU > <43635000 ps> write D[0xFFFFFFFF]; A[0xB0000208] (0)[434]
# <44295000 ps> <SCPU > <44295000 ps> read D[0x001FFFC0]; A[0xB0000208] (0)[435]
# <44295000 ps> <SCPU > <44295000 ps>  ----------------------------------------------------------------
# <44295000 ps> <SCPU > <44295000 ps>  -- Check Transmit Channel Write register for  write/read access.
# <44295000 ps> <SCPU > <44295000 ps>  ----------------------------------------------------------------
# <44955000 ps> <SCPU > <44955000 ps> read D[0x00000000]; A[0xB000020C] (0)[443]
# <45585000 ps> <SCPU > <45585000 ps> write D[0xFFFFFFFF]; A[0xB000020C] (0)[446]
# <46695000 ps> <SCPU > <46695000 ps> read D[0x000FFFF0]; A[0xB000020C] (0)[447]
# <46695000 ps> <SCPU > <46695000 ps>  ----------------------------------------------------------------
# <46695000 ps> <SCPU > <46695000 ps>  -- Check Transmit Channel Read register for  write/read access.
# <46695000 ps> <SCPU > <46695000 ps>  ----------------------------------------------------------------
# <47355000 ps> <SCPU > <47355000 ps> read D[0x00000000]; A[0xB0000210] (0)[455]
# <47985000 ps> <SCPU > <47985000 ps> write D[0xFFFFFFFF]; A[0xB0000210] (0)[458]
# <48645000 ps> <SCPU > <48645000 ps> read D[0x000FFFF0]; A[0xB0000210] (0)[459]
# <48645000 ps> <SCPU > <48645000 ps>  ----------------------------------------------------------------
# <48645000 ps> <SCPU > <48645000 ps>  -- Check Transmit Channel Interrupt register for  write/read access.
# <48645000 ps> <SCPU > <48645000 ps>  ----------------------------------------------------------------
# <49305000 ps> <SCPU > <49305000 ps> read D[0x00000000]; A[0xB0000214] (0)[467]
# <49935000 ps> <SCPU > <49935000 ps> write D[0xFFFFFFFF]; A[0xB0000214] (0)[470]
# <50595000 ps> <SCPU > <50595000 ps> read D[0x000FFFF0]; A[0xB0000214] (0)[471]
# <50595000 ps> <SCPU > <50595000 ps>  ----------------------------------------------------------------
# <50595000 ps> <SCPU > <50595000 ps>  -- Check Receive Channel Control register for  write/read access.
# <50595000 ps> <SCPU > <50595000 ps>  ----------------------------------------------------------------
# <51255000 ps> <SCPU > <51255000 ps> read D[0x00000000]; A[0xB0000300] (0)[479]
# <51885000 ps> <SCPU > <51885000 ps> write D[0xFFFFFFFF]; A[0xB0000300] (0)[482]
# <52545000 ps> <SCPU > <52545000 ps> read D[0x00000001]; A[0xB0000300] (0)[483]
# <52545000 ps> <SCPU > <52545000 ps>  ----------------------------------------------------------------
# <52545000 ps> <SCPU > <52545000 ps>  -- Check Receive Channel Address register for  write/read access.
# <52545000 ps> <SCPU > <52545000 ps>  ----------------------------------------------------------------
# <53205000 ps> <SCPU > <53205000 ps> read D[0x00000000]; A[0xB0000304] (0)[491]
# <53835000 ps> <SCPU > <53835000 ps> write D[0xFFFFFFFF]; A[0xB0000304] (0)[494]
# <54495000 ps> <SCPU > <54495000 ps> read D[0xFFFFFC00]; A[0xB0000304] (0)[495]
# <54495000 ps> <SCPU > <54495000 ps>  ----------------------------------------------------------------
# <54495000 ps> <SCPU > <54495000 ps>  -- Check Receive Channel Size register for  write/read access.
# <54495000 ps> <SCPU > <54495000 ps>  ----------------------------------------------------------------
# <55155000 ps> <SCPU > <55155000 ps> read D[0x00000000]; A[0xB0000308] (0)[503]
# <55785000 ps> <SCPU > <55785000 ps> write D[0xFFFFFFFF]; A[0xB0000308] (0)[506]
# <56445000 ps> <SCPU > <56445000 ps> read D[0x001FFFC0]; A[0xB0000308] (0)[507]
# <56445000 ps> <SCPU > <56445000 ps>  ----------------------------------------------------------------
# <56445000 ps> <SCPU > <56445000 ps>  -- Check Receive Channel Write register for  write/read access.
# <56445000 ps> <SCPU > <56445000 ps>  ----------------------------------------------------------------
# <57105000 ps> <SCPU > <57105000 ps> read D[0x00000000]; A[0xB000030C] (0)[515]
# <57735000 ps> <SCPU > <57735000 ps> write D[0xFFFFFFFF]; A[0xB000030C] (0)[518]
# <58395000 ps> <SCPU > <58395000 ps> read D[0x000FFFF0]; A[0xB000030C] (0)[519]
# <58395000 ps> <SCPU > <58395000 ps>  ----------------------------------------------------------------
# <58395000 ps> <SCPU > <58395000 ps>  -- Check Receive Channel Read register for  write/read access.
# <58395000 ps> <SCPU > <58395000 ps>  ----------------------------------------------------------------
# <59055000 ps> <SCPU > <59055000 ps> read D[0x00000000]; A[0xB0000310] (0)[527]
# <59685000 ps> <SCPU > <59685000 ps> write D[0xFFFFFFFF]; A[0xB0000310] (0)[530]
# <60345000 ps> <SCPU > <60345000 ps> read D[0x000FFFF0]; A[0xB0000310] (0)[531]
# <60345000 ps> <SCPU > <60345000 ps>  ----------------------------------------------------------------
# <60345000 ps> <SCPU > <60345000 ps>  -- Check Receive Channel Interrupt register for  write/read access.
# <60345000 ps> <SCPU > <60345000 ps>  ----------------------------------------------------------------
# <61005000 ps> <SCPU > <61005000 ps> read D[0x00000000]; A[0xB0000314] (0)[539]
# <61635000 ps> <SCPU > <61635000 ps> write D[0xFFFFFFFF]; A[0xB0000314] (0)[542]
# <62295000 ps> <SCPU > <62295000 ps> read D[0x000FFFF0]; A[0xB0000314] (0)[543]
# <62295000 ps> <SCPU > <62295000 ps>  ----------------------------------------------------------------
# <62295000 ps> <SCPU > <62295000 ps>  -- Check Receive Channel Mask register for  write/read access.
# <62295000 ps> <SCPU > <62295000 ps>  ----------------------------------------------------------------
# <62955000 ps> <SCPU > <62955000 ps> read D[0x1FFFFFFF]; A[0xB0000318] (0)[551]
# <63585000 ps> <SCPU > <63585000 ps> write D[0xE0000000]; A[0xB0000318] (0)[554]
# <64245000 ps> <SCPU > <64245000 ps> read D[0x00000000]; A[0xB0000318] (0)[555]
```

```
# <64245000 ps> <SCPU ><64245000 ps> ------------------------------------------------------------
# <64245000 ps> <SCPU ><64245000 ps> -- Check Receive Channel Code register for write/read access.
# <64245000 ps> <SCPU ><64245000 ps> ------------------------------------------------------------
# <64905000 ps> <SCPU ><64905000 ps> read D[0x00000000]; A[0xB000031C] (0)[563]
# <65535000 ps> <SCPU ><65535000 ps> write D[0xFFFFFFFF]; A[0xB000031C] (0)[566]
# <66195000 ps> <SCPU ><66195000 ps> read D[0x1FFFFFFF]; A[0xB000031C] (0)[567]
# <66195000 ps> <SCPU ><66195000 ps> ------------------------------------------------------------
# <66195000 ps> <SCPU ><66195000 ps> -- Check Interrupt Mask register for write/read access.
# <66195000 ps> <SCPU ><66195000 ps> ------------------------------------------------------------
# <66855000 ps> <SCPU ><66855000 ps> read D[0x00000000]; A[0xB0000110] (0)[575]
# <67485000 ps> <SCPU ><67485000 ps> write D[0xFFFFFFFF]; A[0xB0000110] (0)[578]
# <68145000 ps> <SCPU ><68145000 ps> read D[0x0001FFFF]; A[0xB0000110] (0)[579]
# <68145000 ps> <SCPU ><68145000 ps> ------------------------------------------------------------
# <68145000 ps> <SCPU ><68145000 ps> -- Check Pending Interrupt register for write/read access.
# <68145000 ps> <SCPU ><68145000 ps> ------------------------------------------------------------
# <68775000 ps> <SCPU ><68775000 ps> write D[0xFFFFFFFF]; A[0xB000010C] (0)[587]
# <69435000 ps> <SCPU ><69435000 ps> read D[0x0001FFFF]; A[0xB000010C] (0)[588]
# <70095000 ps> <SCPU ><70095000 ps> read D[0x00000000]; A[0xB000010C] (0)[590]
# <70095000 ps> <SCPU ><70095000 ps> ------------------------------------------------------------
# <70095000 ps> <SCPU ><70095000 ps> -- Check Pending Interrupt Masked register for read access.
# <70095000 ps> <SCPU ><70095000 ps> ------------------------------------------------------------
# <70725000 ps> <SCPU ><70725000 ps> write D[0xFFFFFFFF]; A[0xB000010C] (0)[598]
# <71385000 ps> <SCPU ><71385000 ps> read D[0x0001FFFF]; A[0xB0000104] (0)[601]
# <72045000 ps> <SCPU ><72045000 ps> read D[0x00000000]; A[0xB0000104] (0)[603]
# <72045000 ps> <SCPU ><72045000 ps> ------------------------------------------------------------
# <72045000 ps> <SCPU ><72045000 ps> -- Check Pending Interrupt Masked Status register for read access.
# <72045000 ps> <SCPU ><72045000 ps> ------------------------------------------------------------
# <72675000 ps> <SCPU ><72675000 ps> write D[0xFFFFFFFF]; A[0xB000010C] (0)[611]
# <73335000 ps> <SCPU ><73335000 ps> read D[0x0001FFFF]; A[0xB0000100] (0)[614]
# <73995000 ps> <SCPU ><73995000 ps> read D[0x0001FFFF]; A[0xB0000100] (0)[616]
# <73995000 ps> <SCPU ><73995000 ps> ------------------------------------------------------------
# <73995000 ps> <SCPU ><73995000 ps> -- Check Pending Interrupt Status register for read access.
# <73995000 ps> <SCPU ><73995000 ps> ------------------------------------------------------------
# <74655000 ps> <SCPU ><74655000 ps> read D[0x0001FFFF]; A[0xB0000108] (0)[624]
# <75315000 ps> <SCPU ><75315000 ps> read D[0x0001FFFF]; A[0xB0000108] (0)[626]
# <75315000 ps> <SCPU ><75315000 ps> ------------------------------------------------------------
# <75315000 ps> <SCPU ><75315000 ps> -- Check Pending Interrupt Clear register for write access.
# <75315000 ps> <SCPU ><75315000 ps> ------------------------------------------------------------
# <75945000 ps> <SCPU ><75945000 ps> write D[0xFFFFFFFF]; A[0xB0000114] (0)[634]
# <76605000 ps> <SCPU ><76605000 ps> read D[0x00000000]; A[0xB0000100] (0)[637]
# <76605000 ps> <SCPU ><76605000 ps> ------------------------------------------------------------
# <76605000 ps> <SCPU ><76605000 ps> -- Write to a non existing register.
# <76605000 ps> <SCPU ><76605000 ps> ------------------------------------------------------------
# <77235000 ps> <SCPU ><77235000 ps> write D[0x5555AAAA]; A[0xB0000218] (0)[663]
# <77895000 ps> <SCPU ><77895000 ps> read D[0x00000000]; A[0xB0000218] (0)[664]
# <77895000 ps> <SCPU ><77895000 ps>
# <77895000 ps> <SCPU ><77895000 ps> --- PASS: All tests successful. --
# <77895000 ps> <SCPU ><77895000 ps>
# <77895000 ps> <SCPU ><77895000 ps>
# <77895000 ps> <SCPU ><77895000 ps> --- TEST COMPLETED ---
# <77895000 ps> <SCPU ><77895000 ps>
# <77895000 ps> <SCPU ><77895000 ps> write D[0x00003F04] to IO value register[0].[694]
# <77895000 ps> <SCPU ><77895000 ps> end[695]
# <77905000 ps> ------------------------------------------------------------
# <77905000 ps>                   -- Testbench ended --
# <77905000 ps> ------------------------------------------------------------
# ** Failure: Assertion violation.
#    Time: 77905 ns  Iteration: 0  Process: /tb_cim_board/p_simulation File: ../tbench/tb_cim_board.vhd
# Break at ../tbench/tb_cim_board.vhd line 920
```

B.2 Data reception

```
# Initializing SRAM with zero ... Initializing SRAM with zero ...<0 ps> <SCPU ><0 ps>
# <0 ps> <SCPU ><0 ps>
# <0 ps> <SCPU ><0 ps> **************************
# <0 ps> <SCPU ><0 ps> **                      **
# <0 ps> <SCPU ><0 ps> **     S_PROCESSOR:     **
# <0 ps> <SCPU ><0 ps> **                      **
# <0 ps> <SCPU ><0 ps> ** - Version: 6.0       **
# <0 ps> <SCPU ><0 ps> ** - Author:  R. De Vos  **
# <0 ps> <SCPU ><0 ps> ** - Date:    20.05.2010 **
# <0 ps> <SCPU ><0 ps> **                      **
# <0 ps> <SCPU ><0 ps> **************************
# <0 ps> <SCPU ><0 ps>
# <0 ps> <SCPU ><0 ps>
# grcan0: GR CAN Controller rev 0, irq 0
# <525000 ps> <SCPU ><525000 ps> --- TEST STARTED ---
# <525000 ps> <SCPU ><525000 ps>
# <525000 ps> <SCPU ><525000 ps>  cim0101_board_to_node_transfer_test.scp
# <525000 ps> <SCPU ><525000 ps>
# <3525000 ps> <SCPU ><3525000 ps> ---------------------------------------------
# <3525000 ps> <SCPU ><3525000 ps> -- Configure the defaults of the cim module. --
# <3525000 ps> <SCPU ><3525000 ps> ---------------------------------------------
# <3525000 ps> <SCPU ><3525000 ps> -- pio(16) = node_tx_enable_i(0). --
# <3525000 ps> <SCPU ><3525000 ps> -- pio(17) = node_tx_enable_i(1). --
# <3525000 ps> <SCPU ><3525000 ps> -- pio(18) = node_tx_enable_i(2). --
# <3525000 ps> <SCPU ><3525000 ps> -- pio(19) = node_tx_enable_i(3). --
# <3525000 ps> <SCPU ><3525000 ps> -- pio(20) = node_tx_enable_i(4). --
# ...
# <23895000 ps> <SCPU ><23895000 ps>  Initialise command register of CAN PCI-bridge.
# <23895000 ps> <SCPU ><23895000 ps>    COMMAND = 0x0146
# <23895000 ps> <SCPU ><23895000 ps>
# <24285000 ps> <SCPU ><24285000 ps> write D[0x00000146]; A[0x01000004] (0)[291]
# <24675000 ps> <SCPU ><24675000 ps> read D[0x02000146]; A[0x01000004] (0)[292]
# <24675000 ps> <SCPU ><24675000 ps>
# <24675000 ps> <SCPU ><24675000 ps>  Initialise command register of TB PCI-bridge.
# <24675000 ps> <SCPU ><24675000 ps>    COMMAND = 0x0146
# <24675000 ps> <SCPU ><24675000 ps>
# <25065000 ps> <SCPU ><25065000 ps> write D[0x00000146]; A[0x02000004] (0)[302]
# <25455000 ps> <SCPU ><25455000 ps> read D[0x02000146]; A[0x02000004] (0)[303]
# ...
# <26565000 ps> <SCPU ><26565000 ps>
# <26565000 ps> <SCPU ><26565000 ps> ---------------------------------------------
# <26565000 ps> <SCPU ><26565000 ps> -- Begin CAN IP core reception tests
# <26565000 ps> <SCPU ><26565000 ps> ---------------------------------------------
# <26565000 ps> <SCPU ><26565000 ps>
# <26565000 ps> <SCPU ><26565000 ps> ---------------------------------------------
# <26565000 ps> <SCPU ><26565000 ps> -- configure CAN Configuration register
# <26565000 ps> <SCPU ><26565000 ps> ---------------------------------------------
# <27195000 ps> <SCPU ><27195000 ps> write D[0x02544006]; A[0xB0000000] (0)[340]
# <27855000 ps> <SCPU ><27855000 ps> read D[0x02544006]; A[0xB0000000] (0)[341]
# <27855000 ps> <SCPU ><27855000 ps> ---------------------------------------------
# <27855000 ps> <SCPU ><27855000 ps> -- configure CAN Control register
# <27855000 ps> <SCPU ><27855000 ps> ---------------------------------------------
# <28485000 ps> <SCPU ><28485000 ps> write D[0x00000001]; A[0xB0000008] (0)[349]
# <29145000 ps> <SCPU ><29145000 ps> read D[0x00000001]; A[0xB0000008] (0)[350]
# <29145000 ps> <SCPU ><29145000 ps> ---------------------------------------------
# <29145000 ps> <SCPU ><29145000 ps> -- configure CanRxADDR register
# <29145000 ps> <SCPU ><29145000 ps> ---------------------------------------------
# <29775000 ps> <SCPU ><29775000 ps> write D[0x40000000]; A[0xB0000304] (0)[358]
# <30435000 ps> <SCPU ><30435000 ps> read D[0x40000000]; A[0xB0000304] (0)[359]
# <30435000 ps> <SCPU ><30435000 ps> ---------------------------------------------
# <30435000 ps> <SCPU ><30435000 ps> -- configure CanRxSIZE register
# <30435000 ps> <SCPU ><30435000 ps> ---------------------------------------------
# <31065000 ps> <SCPU ><31065000 ps> write D[0x00000400]; A[0xB0000308] (0)[368]
# <31725000 ps> <SCPU ><31725000 ps> read D[0x00000400]; A[0xB0000308] (0)[369]
# <31725000 ps> <SCPU ><31725000 ps> ---------------------------------------------
# <31725000 ps> <SCPU ><31725000 ps> -- configure CanRxMASK register
# <31725000 ps> <SCPU ><31725000 ps> ---------------------------------------------
# <32355000 ps> <SCPU ><32355000 ps> write D[0x00000000]; A[0xB0000318] (0)[377]
# <33015000 ps> <SCPU ><33015000 ps> read D[0x00000000]; A[0xB0000318] (0)[378]
# <33015000 ps> <SCPU ><33015000 ps> ---------------------------------------------
# <33015000 ps> <SCPU ><33015000 ps> -- configure CanRxCTRL register
# <33015000 ps> <SCPU ><33015000 ps> ---------------------------------------------
# <33645000 ps> <SCPU ><33645000 ps> write D[0x00000001]; A[0xB0000300] (0)[386]
# <34305000 ps> <SCPU ><34305000 ps> read D[0x00000001]; A[0xB0000300] (0)[387]
# <34305000 ps> <SCPU ><34305000 ps> ---------------------------------------------
# <34305000 ps> <SCPU ><34305000 ps> -- Start transmission from node 0
# <34305000 ps> <SCPU ><34305000 ps> ---------------------------------------------
# <49575000 ps> <SCPU ><49575000 ps> write D[0x00013F00] to IO value register[0].[400]
# <50265000 ps> <SCPU ><50265000 ps> read D[0x00000000]; A[0xB0000004] (0)[403]
# <50925000 ps> <SCPU ><50925000 ps> read D[0x00000000]; A[0xB0000004] (0)[404]
# <50925000 ps> <SCPU ><50925000 ps> write D[0x00003F00] to IO value register[0].[405]
# <51435000 ps> <SCPU ><51435000 ps> read D[0x00000002]; A[0x40000008] (0)[410]
# <51945000 ps> <SCPU ><51945000 ps> read D[0x00000002]; A[0x40000008] (0)[410]
# <52455000 ps> <SCPU ><52455000 ps> read D[0x00000002]; A[0x40000008] (0)[410]
# <52965000 ps> <SCPU ><52965000 ps> read D[0x00000002]; A[0x40000008] (0)[410]
# <53475000 ps> <SCPU ><53475000 ps> read D[0x00000002]; A[0x40000008] (0)[410]
# <53985000 ps> <SCPU ><53985000 ps> read D[0x00000002]; A[0x40000008] (0)[410]
# <54495000 ps> <SCPU ><54495000 ps> read D[0x00000002]; A[0x40000008] (0)[410]
```

```
# ...
# <101415000 ps> <SCPU ><101415000 ps> read D[0x00000002]; A[0x40000008] (0)[410]
# <101925000 ps> <SCPU ><101925000 ps> read D[0x00000002]; A[0x40000008] (0)[410]
# <102435000 ps> <SCPU ><102435000 ps> read D[0x00000002]; A[0x40000008] (0)[410]
# <102945000 ps> <SCPU ><102945000 ps> read D[0x00000002]; A[0x40000008] (0)[410]
# <103455000 ps> <SCPU ><103455000 ps> read D[0x00000002]; A[0x40000008] (0)[410]
# <103965000 ps> <SCPU ><103965000 ps> read D[0x00000002]; A[0x40000008] (0)[410]
# <104475000 ps> <SCPU ><104475000 ps> read D[0x00000002]; A[0x40000008] (0)[410]
# <105735000 ps> <SCPU ><105735000 ps> read D[0x53000000]; A[0x40000008] (0)[410]
# <113475000 ps> <SCPU ><113475000 ps> read D[0xUUUUUUUU]; A[0x10000000] (0)[415]
# <113565000 ps> <SCPU ><113565000 ps> read D[0x153UUUUU]; A[0x10010000] (0)[416]
# <113655000 ps> <SCPU ><113655000 ps> read D[0x153UUUUU]; A[0x10020000] (0)[417]
# <113745000 ps> <SCPU ><113745000 ps> read D[0x153UUUUU]; A[0x10030000] (0)[418]
# <113835000 ps> <SCPU ><113835000 ps> read D[0x153UUUUU]; A[0x10040000] (0)[419]
# <113835000 ps> <SCPU ><113835000 ps> ----------------------------------------------------
# <113835000 ps> <SCPU ><113835000 ps> -- check received CAN frame in memory
# <113835000 ps> <SCPU ><113835000 ps> ----------------------------------------------------
# <114345000 ps> <SCPU ><114345000 ps> read D[0x000C0000]; A[0x40000000] (0)[424]
# <114855000 ps> <SCPU ><114855000 ps> read D[0x10000000]; A[0x40000004] (0)[426]
# <115365000 ps> <SCPU ><115365000 ps> read D[0x53000000]; A[0x40000008] (0)[428]
# <115875000 ps> <SCPU ><115875000 ps> read D[0x00000000]; A[0x4000000C] (0)[430]
# <115875000 ps> <SCPU ><115875000 ps> ----------------------------------------------------
# <115875000 ps> <SCPU ><115875000 ps> -- check CAN registers
# <115875000 ps> <SCPU ><115875000 ps> ----------------------------------------------------
# <116535000 ps> <SCPU ><116535000 ps> read D[0x00000000]; A[0xB0000004] (0)[438]
# <117195000 ps> <SCPU ><117195000 ps> read D[0x00000200]; A[0xB000010C] (0)[443]
# <117855000 ps> <SCPU ><117855000 ps> read D[0x00000000]; A[0xB0000004] (0)[448]
# <118515000 ps> <SCPU ><118515000 ps> read D[0x00000000]; A[0xB000010C] (0)[453]
# <119175000 ps> <SCPU ><119175000 ps> read D[0x00000000]; A[0xB0000314] (0)[458]
# <119835000 ps> <SCPU ><119835000 ps> read D[0x00000010]; A[0xB000030C] (0)[463]
# <120495000 ps> <SCPU ><120495000 ps> read D[0x00000000]; A[0xB0000310] (0)[468]
# <120495000 ps> <SCPU ><120495000 ps> ----------------------------------------------------
# <120495000 ps> <SCPU ><120495000 ps> -- check CanRxSIZE register
# <120495000 ps> <SCPU ><120495000 ps> ----------------------------------------------------
# <121155000 ps> <SCPU ><121155000 ps> read D[0x00000400]; A[0xB0000308] (0)[476]
# <121155000 ps> <SCPU ><121155000 ps>
# <121155000 ps> <SCPU ><121155000 ps> --- PASS: All tests successful. --
# <121155000 ps> <SCPU ><121155000 ps>
# <121155000 ps> <SCPU ><121155000 ps>
# <121155000 ps> <SCPU ><121155000 ps> --- TEST COMPLETED ---
# <121155000 ps> <SCPU ><121155000 ps>
# <121155000 ps> <SCPU ><121155000 ps> write D[0x00003F04] to IO value register[0].[508]
# <121155000 ps> <SCPU ><121155000 ps> end[509]
# <121165000 ps> ------------------------------------------------------------------
# <121165000 ps>                 -- Testbench ended --
# <121165000 ps> ------------------------------------------------------------------
# ** Failure: Assertion violation.
#    Time: 121165 ns  Iteration: 0  Process: /tb_cim_board/p_simulation File: ../tbench/tb_cim_board.vhd
# Break at ../tbench/tb_cim_board.vhd line 920
```

B.3 Data transmission

```
# Initializing SRAM with zero ...Initializing SRAM with zero ...<0 ps> <SCPU ><0 ps>
# <0 ps> <SCPU ><0 ps>
# <0 ps> <SCPU ><0 ps> **************************
# <0 ps> <SCPU ><0 ps> **    **
# <0 ps> <SCPU ><0 ps> **      S_PROCESSOR:        **
# <0 ps> <SCPU ><0 ps> **    **
# <0 ps> <SCPU ><0 ps> ** - Version: 6.0          **
# <0 ps> <SCPU ><0 ps> ** - Author:  R. De Vos    **
# <0 ps> <SCPU ><0 ps> ** - Date:     20.05.2010 **
# <0 ps> <SCPU ><0 ps> **    **
# <0 ps> <SCPU ><0 ps> **************************
# <0 ps> <SCPU ><0 ps>
# <0 ps> <SCPU ><0 ps>
# grcan0: GR CAN Controller rev 0, irq 0
# <525000 ps> <SCPU ><525000 ps>  --- TEST STARTED ---
# <525000 ps> <SCPU ><525000 ps>
# <525000 ps> <SCPU ><525000 ps>   cim0101_board_to_node_transfer_test.scp
# <525000 ps> <SCPU ><525000 ps>
# <3525000 ps> <SCPU ><3525000 ps>  ----------------------------------------------
# <3525000 ps> <SCPU ><3525000 ps>  -- Configure the defaults of the cim module. --
# <3525000 ps> <SCPU ><3525000 ps>  ----------------------------------------------
# <3525000 ps> <SCPU ><3525000 ps>  -- pio(16) = node_tx_enable_i(0). --
# <3525000 ps> <SCPU ><3525000 ps>  -- pio(17) = node_tx_enable_i(1). --
# <3525000 ps> <SCPU ><3525000 ps>  -- pio(18) = node_tx_enable_i(2). --
# <3525000 ps> <SCPU ><3525000 ps>  -- pio(19) = node_tx_enable_i(3). --
# <3525000 ps> <SCPU ><3525000 ps>  -- pio(20) = node_tx_enable_i(4). --
# ...
# <23895000 ps> <SCPU ><23895000 ps>
# <23895000 ps> <SCPU ><23895000 ps>   Initialise command register of CAN PCI-bridge.
# <23895000 ps> <SCPU ><23895000 ps>     COMMAND = 0x0146
# <23895000 ps> <SCPU ><23895000 ps>
# <24285000 ps> <SCPU ><24285000 ps> write D[0x00000146]; A[0x01000004] (0)[291]
# <24675000 ps> <SCPU ><24675000 ps> read D[0x02000146]; A[0x01000004] (0)[292]
# <24675000 ps> <SCPU ><24675000 ps>
# <24675000 ps> <SCPU ><24675000 ps>   Initialise command register of TB PCI-bridge.
# <24675000 ps> <SCPU ><24675000 ps>     COMMAND = 0x0146
# <24675000 ps> <SCPU ><24675000 ps>
# <25065000 ps> <SCPU ><25065000 ps> write D[0x00000146]; A[0x02000004] (0)[302]
# <25455000 ps> <SCPU ><25455000 ps> read D[0x02000146]; A[0x02000004] (0)[303]
# <25455000 ps> <SCPU ><25455000 ps>
# <25455000 ps> <SCPU ><25455000 ps>  -- Configure the IRQ controler of the SCPU. --
# <25455000 ps> <SCPU ><25455000 ps>
# <25455000 ps> <SCPU ><25455000 ps> write D[0xFFFFFFFE] to IRQ mask register.[310]
# <25455000 ps> <SCPU ><25455000 ps> write D[0x00000001] to IRQ configuration register[0].[311]
# <25485000 ps> <SCPU ><25485000 ps>
# <25485000 ps> <SCPU ><25485000 ps>  -- configure + enable the IRQ controler.--
# <25485000 ps> <SCPU ><25485000 ps>
# <26025000 ps> <SCPU ><26025000 ps> write D[0x00000001]; A[0xB3000004] (0)[321]
# <26565000 ps> <SCPU ><26565000 ps> write D[0x00000001]; A[0xB3000000] (0)[324]
# <26565000 ps> <SCPU ><26565000 ps>
# <26565000 ps> <SCPU ><26565000 ps>  ------------------------------------------------------------
# <26565000 ps> <SCPU ><26565000 ps>  -- Begin CAN IP core Transmission tests
# <26565000 ps> <SCPU ><26565000 ps>  ------------------------------------------------------------
# <26565000 ps> <SCPU ><26565000 ps>
# <26565000 ps> <SCPU ><26565000 ps>  ------------------------------------------------------------
# <26565000 ps> <SCPU ><26565000 ps>  -- configure CAN Configuration register
# <26565000 ps> <SCPU ><26565000 ps>  ------------------------------------------------------------
# <27195000 ps> <SCPU ><27195000 ps> write D[0x02544002]; A[0xB0000000] (0)[347]
# <27855000 ps> <SCPU ><27855000 ps> read D[0x02544002]; A[0xB0000000] (0)[348]
# <27855000 ps> <SCPU ><27855000 ps>  ------------------------------------------------------------
# <27855000 ps> <SCPU ><27855000 ps>  -- configure CAN Control register
# <27855000 ps> <SCPU ><27855000 ps>  ------------------------------------------------------------
# <28485000 ps> <SCPU ><28485000 ps> write D[0x00000001]; A[0xB0000008] (0)[356]
# <29145000 ps> <SCPU ><29145000 ps> read D[0x00000001]; A[0xB0000008] (0)[357]
# <29145000 ps> <SCPU ><29145000 ps>  ------------------------------------------------------------
# <29145000 ps> <SCPU ><29145000 ps>  -- Prepare CAN data frame for transmission
# <29145000 ps> <SCPU ><29145000 ps>  ------------------------------------------------------------
# <29625000 ps> <SCPU ><29625000 ps> write D[0x0000C000]; A[0x40001000] (0)[365]
# <30105000 ps> <SCPU ><30105000 ps> write D[0x10000000]; A[0x40001004] (0)[367]
# <30585000 ps> <SCPU ><30585000 ps> write D[0x53000000]; A[0x40001008] (0)[369]
# <31065000 ps> <SCPU ><31065000 ps> write D[0x00000000]; A[0x4000100C] (0)[371]
# <31065000 ps> <SCPU ><31065000 ps>  ------------------------------------------------------------
# <31065000 ps> <SCPU ><31065000 ps>  -- check CanSTAT register
# <31065000 ps> <SCPU ><31065000 ps>  ------------------------------------------------------------
# <31725000 ps> <SCPU ><31725000 ps> read D[0x00000000]; A[0xB0000004] (0)[378]
# <46995000 ps> <SCPU ><46995000 ps>  ------------------------------------------------------------
# <46995000 ps> <SCPU ><46995000 ps>  -- configure CanTxADDR register
# <46995000 ps> <SCPU ><46995000 ps>  ------------------------------------------------------------
# <47625000 ps> <SCPU ><47625000 ps> write D[0x40001000]; A[0xB0000204] (0)[393]
# <48285000 ps> <SCPU ><48285000 ps> read D[0x40001000]; A[0xB0000204] (0)[394]
# <48285000 ps> <SCPU ><48285000 ps>  ------------------------------------------------------------
# <48285000 ps> <SCPU ><48285000 ps>  -- configure CanTxSIZE register
# <48285000 ps> <SCPU ><48285000 ps>  ------------------------------------------------------------
# <48915000 ps> <SCPU ><48915000 ps> write D[0x00000100]; A[0xB0000208] (0)[402]
# <49575000 ps> <SCPU ><49575000 ps> read D[0x00000100]; A[0xB0000208] (0)[403]
# <49575000 ps> <SCPU ><49575000 ps>  ------------------------------------------------------------
# <49575000 ps> <SCPU ><49575000 ps>  -- configure CanTxWRITE register
# <49575000 ps> <SCPU ><49575000 ps>  ------------------------------------------------------------
```

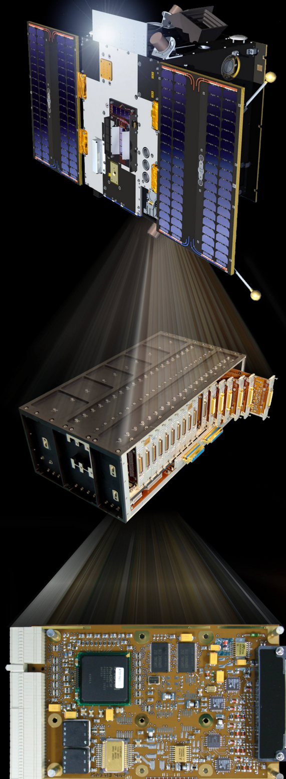
```
# <50205000 ps> <SCPU > <50205000 ps> write D[0x00000000]; A[0xB000020C] (0)[411]
# <50865000 ps> <SCPU > <50865000 ps> read D[0x00000000]; A[0xB000020C] (0)[412]
# <50865000 ps> <SCPU > <50865000 ps> ------------------------------------------------------------
# <50865000 ps> <SCPU > <50865000 ps> -- configure CanTxREAD register
# <50865000 ps> <SCPU > <50865000 ps> ------------------------------------------------------------
# <51495000 ps> <SCPU > <51495000 ps> write D[0x00000000]; A[0xB0000210] (0)[420]
# <52155000 ps> <SCPU > <52155000 ps> read D[0x00000000]; A[0xB0000210] (0)[421]
# <52155000 ps> <SCPU > <52155000 ps> ------------------------------------------------------------
# <52155000 ps> <SCPU > <52155000 ps> -- configure CanTxCTRL register
# <52155000 ps> <SCPU > <52155000 ps> ------------------------------------------------------------
# <52785000 ps> <SCPU > <52785000 ps> write D[0x00000001]; A[0xB0000200] (0)[429]
# <53445000 ps> <SCPU > <53445000 ps> read D[0x00000001]; A[0xB0000200] (0)[430]
# <53445000 ps> <SCPU > <53445000 ps> ------------------------------------------------------------
# <53445000 ps> <SCPU > <53445000 ps> -- check CanIMR register
# <53445000 ps> <SCPU > <53445000 ps> ------------------------------------------------------------
# <54105000 ps> <SCPU > <54105000 ps> read D[0x00000000]; A[0xB0000110] (0)[438]
# <54105000 ps> <SCPU > <54105000 ps> ------------------------------------------------------------
# <54105000 ps> <SCPU > <54105000 ps> -- configure CanIMR register
# <54105000 ps> <SCPU > <54105000 ps> ------------------------------------------------------------
# <54735000 ps> <SCPU > <54735000 ps> write D[0x0001FFFF]; A[0xB0000110] (0)[446]
# <55395000 ps> <SCPU > <55395000 ps> read D[0x0001FFFF]; A[0xB0000110] (0)[447]
# <55395000 ps> <SCPU > <55395000 ps> ------------------------------------------------------------
# <55395000 ps> <SCPU > <55395000 ps> -- configure CanTxWRITE register
# <55395000 ps> <SCPU > <55395000 ps> ------------------------------------------------------------
# <56025000 ps> <SCPU > <56025000 ps> write D[0x00000010]; A[0xB000020C] (0)[455]
# <57555000 ps> <SCPU > <57555000 ps> read D[0x00000010]; A[0xB000020C] (0)[456]
# <57555000 ps> <SCPU > <57555000 ps> ------------------------------------------------------------
# <57555000 ps> <SCPU > <57555000 ps> -- Start transmission
# <57555000 ps> <SCPU > <57555000 ps> ------------------------------------------------------------
# <57615000 ps> <SCPU > <57615000 ps> read D[0xUUUUUUUU]; A[0x10000000] (0)[464]
# <57705000 ps> <SCPU > <57705000 ps> read D[0xUUUUUUUU]; A[0x10010000] (0)[465]
# <57795000 ps> <SCPU > <57795000 ps> read D[0xUUUUUUUU]; A[0x10020000] (0)[466]
# <57885000 ps> <SCPU > <57885000 ps> read D[0xUUUUUUUU]; A[0x10030000] (0)[467]
# <57975000 ps> <SCPU > <57975000 ps> read D[0xUUUUUUUU]; A[0x10040000] (0)[468]
# ...
# <112245000 ps> <SCPU > <112245000 ps> read D[0xUUUUUUUU]; A[0x10020000] (0)[466]
# <112335000 ps> <SCPU > <112335000 ps> read D[0xUUUUUUUU]; A[0x10030000] (0)[467]
# <112425000 ps> <SCPU > <112425000 ps> read D[0xUUUUUUUU]; A[0x10040000] (0)[468]
# <112515000 ps> <SCPU > <112515000 ps> read D[0xUUUUUUUU]; A[0x10000000] (0)[464]
# <112605000 ps> <SCPU > <112605000 ps> read D[0xUUUUUUUU]; A[0x10010000] (0)[465]
# <112695000 ps> <SCPU > <112695000 ps> read D[0xUUUUUUUU]; A[0x10020000] (0)[466]
# <112695000 ps> <SCPU > <112695000 ps> ----------------------------------------
# <112695000 ps> <SCPU > <112695000 ps> -- Interrupt service routine called.
# <112695000 ps> <SCPU > <112695000 ps> ----------------------------------------
# <112695000 ps> <SCPU > <112695000 ps>    -- Read the status register of the IRQC.
# <113265000 ps> <SCPU > <113265000 ps> read D[0x00000080]; A[0xB3000084] (0)[574]
# <113265000 ps> <SCPU > <113265000 ps>    -- Read + clear the status of the CAN IRQ.
# <113925000 ps> <SCPU > <113925000 ps> read D[0x00001500]; A[0xB000010C] (0)[579]
# <113925000 ps> <SCPU > <113925000 ps>    -- Check interrupt register (PIR) value
# <113925000 ps> <SCPU > <113925000 ps>    -- (0x00001500 = Tx (successful Tx of message), TxEmpty (all messages sent)
# <113925000 ps> <SCPU > <113925000 ps>    -- Read + clear the status of the SCPU IRQ.
# <113925000 ps> <SCPU > <113925000 ps> read D[0x00000001] from IRQ register.[585]
# <113955000 ps> <SCPU > <113955000 ps>    -- Clear the IRQC interrupt.
# <114495000 ps> <SCPU > <114495000 ps> write D[0x00000001]; A[0xB3000010] (0)[590]
# <114765000 ps> <SCPU > <114765000 ps>    -- Check if the status of the IRQC is 0x00000000.
# <115335000 ps> <SCPU > <115335000 ps> read D[0x00000000]; A[0xB3000084] (0)[597]
# <115335000 ps> <SCPU > <115335000 ps> ----------------------------------------
# <115395000 ps> <SCPU > <115395000 ps> read D[0xUUUUUUUU]; A[0x10030000] (0)[467]
# <115485000 ps> <SCPU > <115485000 ps> read D[0xUUUUUUUU]; A[0x10040000] (0)[468]
# <115575000 ps> <SCPU > <115575000 ps> read D[0xUUUUUUUU]; A[0x10000000] (0)[464]
# <115665000 ps> <SCPU > <115665000 ps> read D[0xUUUUUUUU]; A[0x10010000] (0)[465]
# <115755000 ps> <SCPU > <115755000 ps> read D[0xUUUUUUUU]; A[0x10020000] (0)[466]
# <115845000 ps> <SCPU > <115845000 ps> read D[0xUUUUUUUU]; A[0x10030000] (0)[467]
# <115935000 ps> <SCPU > <115935000 ps> read D[0xUUUUUUUU]; A[0x10040000] (0)[468]
# <116025000 ps> <SCPU > <116025000 ps> read D[0xUUUUUUUU]; A[0x10000000] (0)[464]
# <116115000 ps> <SCPU > <116115000 ps> read D[0x153UUUUU]; A[0x10010000] (0)[465]
# <116205000 ps> <SCPU > <116205000 ps> read D[0x153UUUUU]; A[0x10020000] (0)[466]
# <116295000 ps> <SCPU > <116295000 ps> read D[0x153UUUUU]; A[0x10030000] (0)[467]
# <116385000 ps> <SCPU > <116385000 ps> read D[0x153UUUUU]; A[0x10040000] (0)[468]
# <116385000 ps> <SCPU > <116385000 ps> -- check CAN node 0
# <116475000 ps> <SCPU > <116475000 ps> read D[0x153UUUUU]; A[0x10000000] (0)[473]
# <116475000 ps> <SCPU > <116475000 ps> -- check CAN node 1
# <116565000 ps> <SCPU > <116565000 ps> read D[0x153UUUUU]; A[0x10010000] (0)[478]
# <116565000 ps> <SCPU > <116565000 ps> -- check CAN node 2
# <116655000 ps> <SCPU > <116655000 ps> read D[0x153UUUUU]; A[0x10020000] (0)[483]
# <116655000 ps> <SCPU > <116655000 ps> -- check CAN node 3
# <116745000 ps> <SCPU > <116745000 ps> read D[0x153UUUUU]; A[0x10030000] (0)[488]
# <116745000 ps> <SCPU > <116745000 ps> -- check CAN node 4
# <116835000 ps> <SCPU > <116835000 ps> read D[0x153UUUUU]; A[0x10040000] (0)[493]
# <116835000 ps> <SCPU > <116835000 ps> --------------------------------------------------
# <116835000 ps> <SCPU > <116835000 ps> -- check CAN registers
# <116835000 ps> <SCPU > <116835000 ps> --------------------------------------------------
# <116835000 ps> <SCPU > <116835000 ps> -- check Status register
# <117495000 ps> <SCPU > <117495000 ps> read D[0x00000000]; A[0xB0000004] (0)[504]
# <117495000 ps> <SCPU > <117495000 ps> -- check Tx Channel interrupt register
# <118155000 ps> <SCPU > <118155000 ps> read D[0x00000000]; A[0xB0000214] (0)[510]
# <118155000 ps> <SCPU > <118155000 ps> -- check Tx Write register (should be 0x00000010)
# <118815000 ps> <SCPU > <118815000 ps> read D[0x00000010]; A[0xB000020C] (0)[516]
# <118815000 ps> <SCPU > <118815000 ps> -- check Tx Read register  (should be 0x00000010)
# <119475000 ps> <SCPU > <119475000 ps> read D[0x00000010]; A[0xB0000210] (0)[522]
# <119475000 ps> <SCPU > <119475000 ps> -- check Tx channel CTRL register (should be 0x00000001: enabled, not ongoing)
```

```
# <120135000 ps> <SCPU ><120135000 ps> read D[0x00000001]; A[0xB0000200] (0)[528]
# <120135000 ps> <SCPU ><120135000 ps>
# <120135000 ps> <SCPU ><120135000 ps> --- PASS: All tests successful. --
# <120135000 ps> <SCPU ><120135000 ps>
# <120135000 ps> <SCPU ><120135000 ps>
# <120135000 ps> <SCPU ><120135000 ps> --- TEST COMPLETED ---
# <120135000 ps> <SCPU ><120135000 ps>
# <120135000 ps> <SCPU ><120135000 ps> write D[0x00003F04] to IO value register[0].[562]
# <120135000 ps> <SCPU ><120135000 ps> end[563]
# <120145000 ps> ------------------------------------------------------------------
# <120145000 ps>                    -- Testbench ended --
# <120145000 ps> ------------------------------------------------------------------
# ** Failure: Assertion violation.
#    Time: 120145 ns  Iteration: 0  Process: /tb_cim_board/p_simulation File: ../tbench/tb_cim_board.vhd
# Break at ../tbench/tb_cim_board.vhd line 920
```

C
Poster

CAN BUS IN ADPMS

ADVANCED DATA & POWER MANAGEMENT SYSTEM



Opdracht

Implementatie van een CAN bus in een ADPMS (Advanced Data & Power Management System), een boordcomputer voor kleine satellieten. Toepassing: on-board communicatie met vb. propulsiesystemen of sensors.

CAN transceiver

CAN controller

Simulatie

Prototype

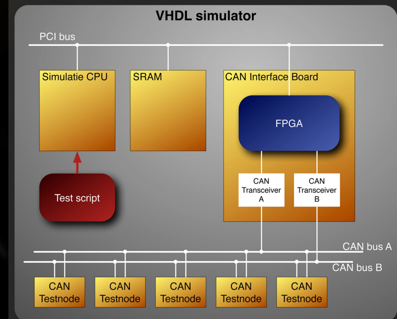
Stralingsbestendig

Hardware tests

VHDL IP cores

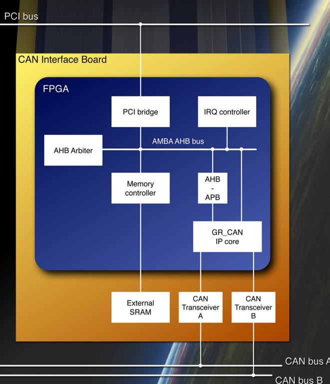
Simulaties

- VHDL testbench opgebouwd uit VHDL modellen van een simulatie CPU, SRAM module, CAN interface board en CAN testnodes.
- CAN interface board en CAN testnodes werden zelf gemodelleerd, andere modellen waren reeds beschikbaar.
- Verificatie CAN testnodes met afzonderlijke testbench.
- Simulaties met ModelSim: test scripts in assembly worden uitgevoerd door de simulatie CPU.
- Resultaat: succesvolle register access, data transmission en data reception tests.



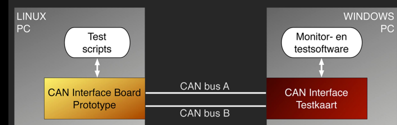
Ontwerp

- CompactPCI module met een dubbele CAN interface met behulp van 2 CAN transceivers en een FPGA.
- Stralingsbestendige CAN transceivers worden gebouwd met RS-485 transceivers.
- Stralingsbestendige FPGA: on-chip AMBA AHB bus met AHB arbiter, PCI bridge, interrupt controller, memory controller en CAN core.
- VHDL cores afkomstig van QinetiQ Space, CAN core van Gaisler Research.



Hardware tests

- Prototype CAN interface board in testopstelling met twee computers en commerciële testkaart.
- Resultaat: succesvolle register access, data transfer, error injection en performance tests.



Stijn Wielandt
2010-2011

Promotors:
KAHO: Jean-Pierre Goemaere, Geoffrey Ottoy
QinetiQ Space: Peter Matthijs, Koen Puimège

QinetiQ Space nv

05/04/2011

D

Structuur van de bijgeleverde DVD

De bijgeleverde DVD is uitsluitend bestemd voor de promotoren van deze masterproef. Deze DVD bevat zelf geschreven code en resultaten van de tests. De structuur wordt weergegeven in figuur D.2 op de pagina hierna.

Omwille van intellectual property bevat deze DVD geen code die gedeeltelijk of volledig geschreven werd door QinetiQ Space NV of Gaisler Research. Het project kan dus niet gesimuleerd worden met de informatie die beschikbaar is op de DVD. Daarom werden in afzonderlijke mappen de gedetailleerde resultaten van simulaties en hardware tests toegevoegd.

Verder werd ook de directory structuur van het volledige project toegevoegd om het verrichte werk te visualiseren. Dit wordt voorgesteld in figuur D.2 op de volgende pagina.

Naam	Commentaar
▼ CAN in ADPMS	
▼ can_node	*CAN testnode model*
▶ build	*.tcl script voor compilatie*
▶ sim	*Bestanden en .tcl scripts voor simulaties*
▶ tbench	*VHDL ontwerp test bench*
▶ vhdl	*VHDL simulatiemodel*
▶ Datasheets	*Datasheets van o.a. de GRCAN IP core*
▶ Directory structuur project	*Directory structuur die gebruikt werd voor het project*
▼ Hardware test results	*Resultaten van de hardware tests*
▶ BUS A	*Alle hardware tests op CAN bus A*
▶ BUS B	*Alle hardware tests op CAN bus B*
▶ Register Access	*Registertoegang test*
▶ Wrong Bus	*Wrong bus test*
▼ Resultaten	*Resultaten van de masterproef*
artikel.pdf	*Wetenschappelijk artikel*
scriptie.pdf	*Scriptie*
▶ Poster	*Poster in verschillende formaten*
▶ Simulation results	*Transcripts van de simulaties*

Figuur D.1: De directory structuur van de bijgeleverde DVD

Naam	Commentaar
CIM	*CAN Interface Module VHDL ontwerp en simulaties*
board	*Simulatiemodel van het CAN interface board*
cim_board	
build	*.tcl script voor compilatie*
sim	*Bestanden en .tcl scripts voor simulaties*
tbench	*VHDL ontwerp algemene test bench*
tsource	*Test scripts voor SCPU*
vhdl	*VHDL simulatiemodel van het CAN interface board*
cores	*QinetiQ Space IP cores*
libraries	*VHDL bibliotheken*
models	*Simulatiemodellen*
can_node	*CAN testnode model*
build	*.tcl script voor compilatie*
sim	*Bestanden en .tcl scripts voor simulaties*
tbench	*VHDL ontwerp test bench*
vhdl	*VHDL simulatiemodel*
k6r4016v1c	*Geheugenmodel*
s_processor	*Simulatie CPU model*
sram	*Geheugenmodel*
module	*CAN Interface FPGA ontwerp*
cim_module	
build	*.tcl script voor compilatie*
pr_proasic	*Scripts en resultaten van place & route*
sim	*Bestanden en .tcl scripts voor simulaties*
syn_proasic	*Scripts en resultaten van synthese*
vhdl	*VHDL FPGA ontwerp*
third_party_cores	*Externe IP cores*
grcan_10	*GRCAN core versie 10*
grcan_13	*GRCAN core versie 13*

Figuur D.2: De gebruikte directory structuur voor het ontwerp