Escola Politècnica Superior
d'Enginyeria de Vilanova i la Geltrú

UNIVERSITAT POLITÈCNICA DE CATALUNYA

TREBALL FINAL DE MÀSTER

TÍTOL: DISSENY D'UNA EINA DE DIAGNÒSTIC CAN BASADA EN FPGA

AUTORS: GALINDO HURTADO, CARLOS

DATA DE PRESENTACIÓ: 10 de Juliol, 2017

COGNOMS: Galindo Hurtado **NOM: Carlos**

TITULACIÓ: Màster Universitari en Enginyeria de Sistemes Automàtics i Electrònica Industrial

PLA: Bolonia 2010

DIRECTOR: Mariano López Garcia

DEPARTAMENT: EEL – Departament d'Enginyeria Electrònica

QUALIFICACIÓ DEL TFM

TRIBUNAL

PRESIDENT **SECRETARI** **VOCAL**

DATA DE LECTURA:

AQUEST PROJECTE TÉ EN COMPTE ASPECTES MEDIAMBIENTALS: □ SÍ □ NO

RESUM

L'objectiu d'aquest projecte és dissenyar i implementar un sistema de comunicació CAN (*Controller Area Network*) de baix cost sobre una plataforma FPGA complint al màxim amb la normativa ISO 11898.

La metodologia empleada en el disseny del sistema és el mètode en V, molt popular en sectors com el de l'automoció, que busca garantir la qualitat del disseny relacionant les diferents fases del desenvolupament amb les especificacions de tests de cada fase. Això s'ha traduït en una millor definició dels requisits del sistema, ja que permet de forma iterativa anar redefinint-los fins arribar a un requisit simple, el qual es pugui associar un test concret, o al menys, una justificació de que es compleix el requisit sense necessitat de testejar-lo.

El sistema s'ha separat en dos mòduls hardware independents, la placa de desenvolupament Avnet LX9 Microboard y un disseny hardware propi en forma de PCB que permet adaptar les senyals d'entrada i sortida del mòdul d'expansió de la placa de desenvolupament als requisits de la capa física del protocol de comunicació CAN.

La Avnet LX9 Microboard integra una Spartan-6, FPGA bastant limitada en recursos, però ideal per garantir un ús eficient dels recursos disponibles. Aquesta FPGA de Xilinx s'ha equipat amb un sistema Microblaze que permet una programació híbrida al emular un microprocessador software sobre la FPGA. D'aquesta manera, el codi que gestiona el protocol de comunicació a baix nivell, bit a bit, s'ha desenvolupat en llenguatge VHDL i integrat dins d'un mòdul IP, i el codi dels *driver*s d'alt nivell i l'aplicació s'han, dissenyat en llenguatge C.

En el disseny també s'ha tingut en compte afegir mòduls, com per exemple, un ADC i un display, que permetin desenvolupar aplicacions on es demostri el correcte funcionament de l'aplicació.

Paraules clau (màxim 10):

CAN	FPGA	Microblaze	VHDL
C	IP	Avnet	LX9
Xilinx	Spartan		

ABSTRACT

The aim of this project is to design and implement a low-cot CAN (Controller Area Network) communication system inside an FPGA platform that fulfils, as much as possible, with the ISO 11898 standard.

The methodology applied to this design is the V-model, state-of-the-art methodology in industries such as the automotive, which aims to ensure the quality of the design by establishing links between the different development phases and the test specifications of each phase. This leads into an improvement of the system requirements definition due to the fact, that this method allows you to iteratively refine the requirements until they become simple and testable. If finally a non-testable requirement is identified, it is required to justify why this requirement can be fulfilled without directly testing it.

The system is divided in two independent hardware modules, an evaluation board Avnet LX9 Microboard and self-developed hardware consisting on a PCB that can adapt input-output signals of the expansion modules of the Avnet LX9 Microboard, to the requirements of the CAN communication physical layer.

Avnet LX9 Microboard allocates a Spartan-6, very resource limited FPGA which is ideal to guarantee an efficient use of the available resources. This Xilinx FPGA has been equipped with a Microblaze system that enables a hybrid programming by emulating a software microprocessor inside de FPGA. In this way, the low-level communication protocol code, which handles bit by bit the communication, is implemented in VHDL programming language, and the *driver*s and the application can be coded in C programming language.

In addition, this design integrates more modules, such as an ADC and a display, to allow the user to develop application that demonstrates the correct performance of the system.

Keywords (10 maximum):

CAN	FPGA	Microblaze	VHDL
C	IP	Avnet	LX9
Xilinx	Spartan		

SUMARI

SUMARI DE FIGURES

1. INTRODUCCIÓ

L'objectiu d'aquest projecte és dissenyar i implementar un sistema de comunicació CAN (*Controller Area Netowork*) de baix cost sobre una plataforma FPGA i que complexi el màxim possible amb la norma ISO 11898. L'objectiu és integrar dins d'aquesta FPGA el microcontrolador i el controlador CAN, aconseguint així:

- Eliminar l'integrat que realitza les funcions de controlador CAN en aquells sistemes on el microcontrolador no l'integra. D'aquesta manera no caldria un sistema de comunicació addicional entre el microcontrolador i el circuit integrat, com per exemple SPI o I2C i, a més, es podria reduir la complexitat del sistema i millorar-ne la fiabilitat.
- En el cas dels microcontroladors amb el controlador CAN integrat, s'obtindria més flexibilitat ja que es podria treballar a nivell de bit i, si fos necessari, dissenyar nodes amb una implementació simplificada del bus CAN.

Un punt important en la reducció de costos és que no cal utilitzar el mòdul IP que proporciona Xilinx per implementar la comunicació CAN. Aquest va subjecte a una llicència que depèn del número d'unitats dissenyades i obliga a pagar un cànon per cada dispositiu en què s'utilitzi, forçant a incrementar el preu de venta del producte final.

La implementació es realitzarà sobre una Avnet LX9 *Microboard* que integra una FPGA Spartan-6. Aquest model de FPGA, encara que sigui antic i limitat en recursos, és ideal per garantir que els recursos disponibles s'utilitzen de la forma més eficient possible.

El disseny serà híbrid, gràcies a la integració d'un sistema Microblaze amb el mòdul IP de baix nivell del bus de comunicació CAN en codi VHDL i les capes més altes (*driver*s i aplicació) en llenguatge C. D'aquesta manera es podrà, d'una banda, gestionar la comunicació a nivell de bit a la freqüència corresponent programant a baix nivell i, de l'altra, desenvolupar l'aplicació de forma més intuïtiva i senzilla, agrupant els beneficis del dos tipus de programació: hardware (VHDL) i software (C).

Per poder realitzar la validació de la implementació, es dissenyarà un hardware en forma de PCB (*Printed Circuit Board*) que permetrà adaptar les senyals digitals d'entrada i sortida de la placa de probes LX9 Microboard a la capa física del bus de comunicació CAN.

La planificació i realització del treball es farà utilitzant un mètode molt estès en el sector de l'automoció que s'anomena Model en V. Aquest mètode ajuda a definir un procés de disseny que minimitza els riscos i millora la qualitat del projecte.

2. DESCRIPCIÓ TEORICA

2.1 Model en V

El Model en V és un model de cicle de vida molt útil pel desenvolupament de productes ja que relaciona les diferents fases del desenvolupament amb les especificacions dels tests de cada fase.

Com es pot veure en la Figura 1, el nom del model ve donat per la forma del diagrama de blocs que el representa. Aquesta figura mostra el diagrama del Model V simplificat. En aquest diagrama, el desenvolupament comença a dalt a l'esquerra i va descendint, per trobar l'anàlisi de requisits, el disseny i la implementació; després torna a pujar i comença pel test a nivell d'unitat i acabant pels test de validació.

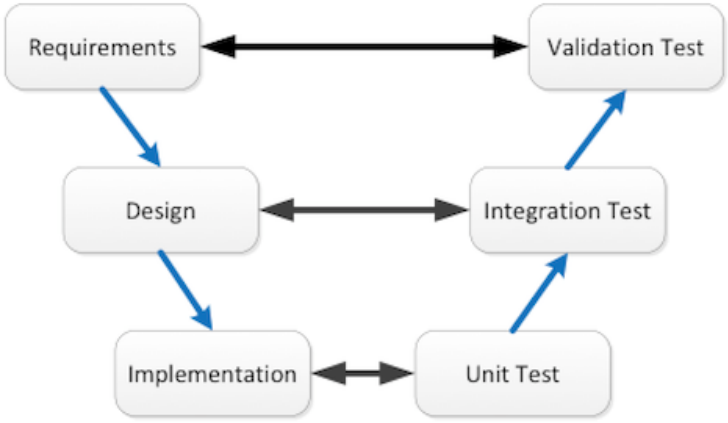


Figura 1. Model en V simplificat

Es tracta d'un model que trenca amb el punt de vista tradicional que oferia el model en forma de cascada, relaciona les fases del disseny amb les fases necessàries per la validació de cadascuna d'elles. La idea és que un cop definits, revisats i validats els requisits del sistema ja es pugui definir el pla de validació. A més, també ajuda a que els requisits definits siguin més entenedors i fàcils de validar.

Al llarg d'aquest projecte s'anirà seguint aquest model, tal i com s'explica a l'apartat 3.1.

2.2 Protocol de comunicació CAN

El CAN és un protocol de comunicació sèrie que permet el control en temps real d'un sistema distribuït, de forma eficient i amb un nivell seguretat elevat.

Es va crear amb el propòsit de suportar l'increment de dispositius electrònics dins dels automòbils. La idea era poder connectar tots els dispositius a un bus de comunicació que fos fiable, robust i que permetés altes velocitats de transmissió en entorns exigents. El resultat va ser un èxit ja que, a dia d'avui, és un dels sistemes de comunicació més populars, especialment en l'automoció, on està present en tots els automòbils que es venen.

A grans trets, el protocol de comunicació CAN ofereix:

- Una alta immunitat a les interferències
- Un protocol estandarditzat que permet simplificar i fer més econòmic el disseny
- Possibilitat de reduir la càrrega del processador
- Transmissió multi-màster i recepció multicast

El protocol de comunicació CAN està definit a la normativa ISO 11898 i queda emmarcat dins el model OSI amb l'estructura que s'observa a la Taula 1.

Taula 1. Model OSI del protocol de comunicació CAN

Nº	Capa	Protocol
7	Capa d'aplicació	Usuari
6	Capa de presentació	Només s'implementen per protocols CAN d'alt nivell, com per exemple: CANopen, MilCAN, SAE J1939 o ISO 1132
5	Capa de sessió	
4	Capa de transport	
3	Capa de xarxa	
2	Capa d'enllaç	- Detecció d'errors i senyalització - Classificació d'errors - Retransmissió automàtica de missatges deteriorats - Validació de missatges - Reconeixement de recepció - Priorització de missatges - Encapsulament de missatge - Velocitat de transmissió i temps
1	Capa física	- Nivell de senyals i representació de bit - Medi de transmissió

En els següents subapartats es descriuran amb més detall les diferents capes i mecanismes que incorpora el protocol de comunicació CAN.

2.2.1 Capa física

El medi de transmissió del bus CAN és en forma diferencial mitjançant un parell trenat, CAN_H (CAN high) i CAN_L (CAN low).

Al bus CAN es defineixen dos nivells lògics, el dominant i el recessiu. En l'estat de repòs, quan cap dels nodes està transmetent, el nivell és recessiu i pot ser sobreescrit per qualsevol node connectat al bus amb un nivell dominant. Com s'explicarà al capítol 2.2.13, això permet realitzar l'arbitratge de missatges i el reconeixement de lectura dels missatges.

La Figura 2 mostra els nivells de tensió de cadascun dels terminals respecte la referència i el resultat de restar el voltatge de CAN_H a CAN_L. També es mostren els rangs de voltatge pel qual es pot considerar el nivell del bus, dominant o recessiu. Com es pot veure en aquesta figura, la tolerància a l'entrada és bastant més gran que a la sortida. L'explicació és que la tolerància de sortida fa referència a les especificacions que ha de tenir el dispositiu que generi la senyal i la tolerància d'entrada als valors acceptables per determinar el nivell lògic. D'aquesta manera i gràcies a la transmissió diferencial, s'aconsegueix un nivell d'immunitat electromagnètica molt elevat.

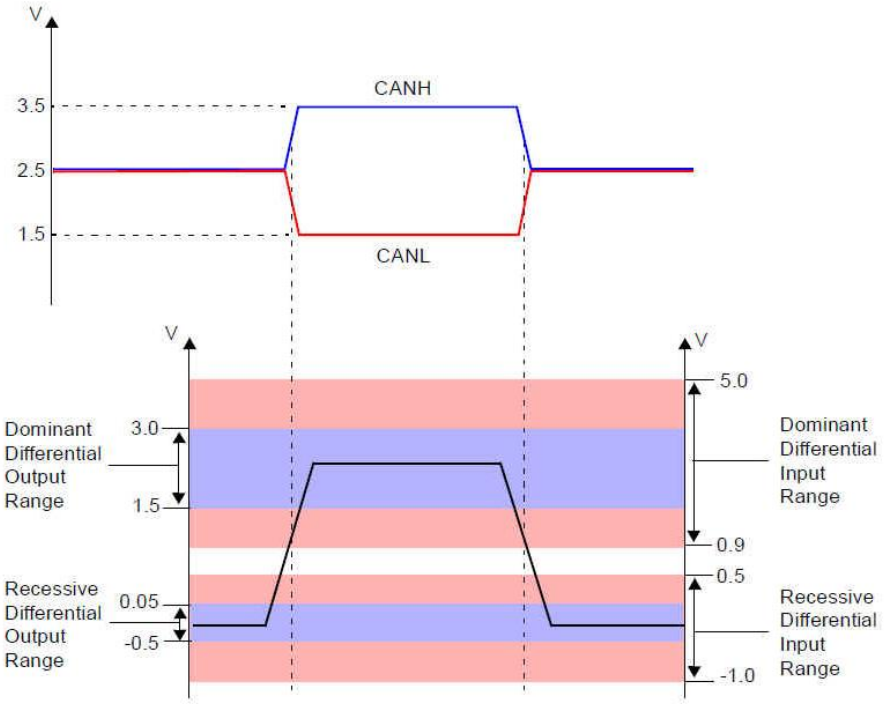


Figura 2. Relació entre els nivells de tensió i els nivells lògics del bus CAN

Al permetre velocitats de transmissió elevades, la reflexió de les senyals transmeses pot provocar que els nodes receptors no llegeixin correctament els missatges. Per aquest motiu, als dos extrems més allunyats del bus de comunicació s'inclou una resistència d'aproximadament 120 Ω. Aquesta resistència s'anomena resistència de terminació CAN. La correcta selecció dels nodes que inclouran la resistència de terminació és molt important, ja que si no estan situades en extrems oposats o s'afegeixen més de dues, els missatges poden ser impossibles de llegir.

En el cas de busos amb dos nodes i una longitud de cable molt curta, amb una resistència de terminació seria suficient.

2.2.2 Temps de bit

Es considera velocitat nominal de bit, el nombre de bits per segon que es poden transmetre en el cas que no hi hagi re-sincronització (cap bit de *stuff*, veure 2.2.10) i per un transmissor ideal. La velocitat de transmissió ha de ser uniforme i fixa en tot el sistema. Les unitats en que s'expressa és bps (bits per segon) i la velocitat màxima que a la que es pot arribar és de 1 Mbps. En el cas de l'automoció, el valor més habitual és de 500 kbps.

El temps de bit nominal és la inversa de la velocitat nominal de bit i es pot dividir en diferents segments de temps (veure Figura 3):

- **Segment de sincronització:** és el temps necessari perquè tots els nodes del bus es sincronitzin; per tant, és necessari que hi hagi un flanc de pujada o baixada en aquest temps.
- **Segment de temps de propagació:** és el temps de marge que es deixa per compensar el temps de propagació del bit pel bus. Aquest temps és calcula com el doble de la suma del temps de propagació del bit pel bus, el retràs del comparador d'entrada i el retràs del *driver* de sortida.

- **Segment 1 i 2 de fase de buffer:** aquests dos segments s'utilitzen per compensar possibles errors en els flancs de sincronització. La suma total de temps d'aquests dos segments sempre és igual però es poden reajustar entre ells per re-sincronitzar ja que el mostreig del bit es fa precisament entre aquests dos segments.

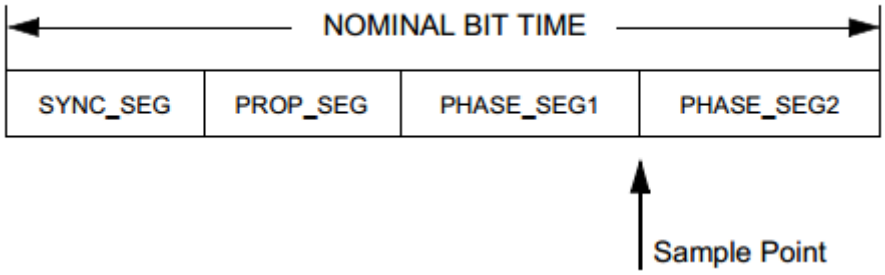


Figura 3. Segments que formen el temps nominal de bit

2.2.3 Sincronització

Els nodes receptors han de sincronitzar-se amb el node transmissor per poder llegir correctament els bits transmesos. Els nodes receptors han d'incorporar dos sistemes de sincronització:

- **Sincronització *hard*:** és la sincronització inicial amb el bit d'inici de trama (bit de SoF) que es produeix amb un canvi de recessiu a dominant en el bus quan aquest està en repòs.
- **Re-sincronització:** és quan la sincronització es du a terme amb els flancs de pujada o baixada durant la transmissió de les trames pel bus.

Les regles per la sincronització són les següents:

- Només es pot sincronitzar un cop durant el temps de bit
- Només es poden utilitzar flancs, ascendents o descendents, per sincronitzar quan el valor actual difereix de l'últim valor mostrejat
- La re-sincronització amb flancs dominant-recessiu és obligatòria i amb flancs recessiu-dominat opcional

2.2.4 Encapsulament de missatges

Hi ha tres tipus de trames CAN:

- Trama de dades: inclou dades del transmissor cap al receptor
- Trama remota: l'envia el transmissor per demanar dades a un altre node
- Trama d'error: es transmet pel bus al detectar un error al bus
- Trama de sobrecàrrega: s'utilitza per afegir un retràs extra entre successives trames de dades o remotes

2.2.5 Trama de dades

Una trama de dades està formada per (veure Figura 4):

- **SoF (*Start of Frame*):** indica l'inici d'una trama de dades o remota i consisteix

en un sol bit dominant.

- **ID (*IDentifier*):** té una longitud de 29 o 11 bits depenent si s'utilitza un identificador ampliat o no. Els bits es transmeten de més significatiu a menys i s'utilitzen per identificar el contingut de la trama.
- **RTR (*Remote Transmission Request*):** és un sol bit que indica si la trama és de dades o remota. En el cas d'una trama de dades, el RTR serà dominant i, pel contrari, en una remota serà recessiu.
- **IDE (*IDentifier Extended*):** és un sol bit que indica si s'utilitza un identificador ampliat (identificador de 29 bits) o no (identificador de 11 bits), depenent de si el valor és recessiu o dominant, respectivament.
- **Bit reservat:** l'encapsulament de les trames de dades i remotes compta amb un bit reservat per un futur i que s'ha de transmetre com dominant encara que el node o nodes receptors han d'acceptar qualsevol combinació.
- **DLC (*Data Length Code*):** té un longitud de 4 bits i indica el nombre de bytes que conté la trama de dades. El valor pot anar des de 0 fins a 8.
- **Camp de dades:** és on s'inclouran les dades de la trama i pot contenir entre 0 i 8 bytes depenent del DLC indicat. Cada byte consistirà en 8 bits que es transmetran de més a menys significatiu.
- **CRC (*Cyclic Redundant Check*):** té una longitud de 15 bits i el transmissor afegeix el resultat d'aplicar un codi BCH als bits enviats per tal que els diferents nodes del sistema puguin executar el mateix càlcul i comprovar la integritat de les dades (veure 2.2.14).
- **Delimitador CRC:** és un bit recessiu que s'envia just després de l'enviament del CRC serveix per marcar el final del CRC.
- **Bit de ACK (*ACKonowledge*):** té una longitud d'un bit i serveix al node transmissor per determinar si algun node ha llegit el missatge. La idea és que el node transmissor posa el nivell de sortida en recessiu, mentre que qualsevol altre node que hagi rebut el missatge el posa en dominant.
- **Delimitador de ACK:** és un bit recessiu extra al bit de ACK, de manera que aquest està envoltat de dos bits recessius.
- **EoF (*End of Frame*):** té una longitud de 7 bits recessius i serveix per indicar el final de la transmissió.

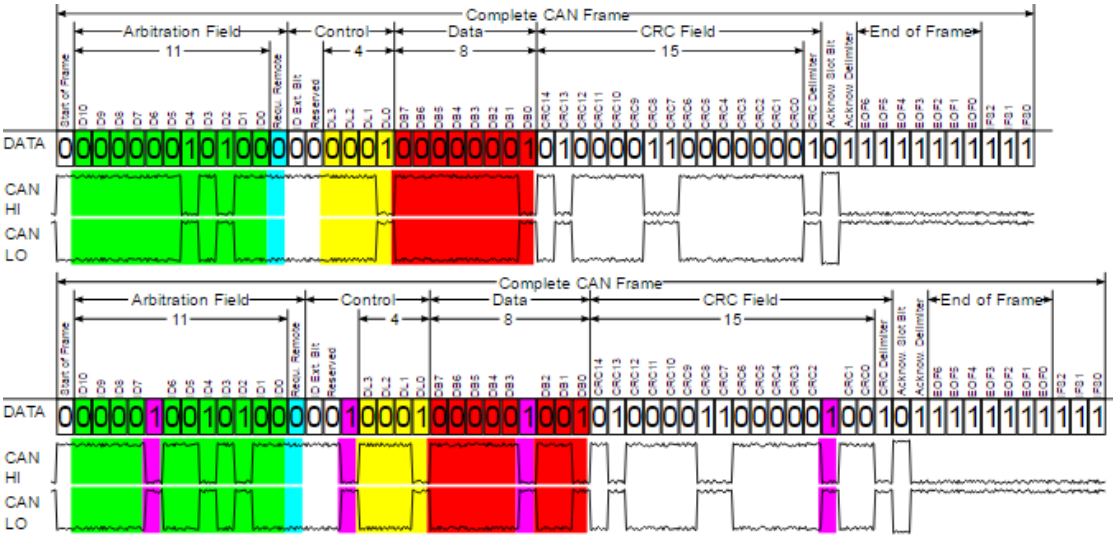


Figura 4. Encapsulament del missatge amb bit *stuffing* (a baix) i sense (a dalt)

2.2.6 Trama remota

Un node actuant com a receptor per unes dades concretes pot iniciar la transmissió d'aquestes dades per ell mateix enviant una trama remota.

Les diferències respecte a la trama de dades són:

- El bit de RTR serà recessiu, indicant que s'està transmetent una trama remota.
- No hi camp de dades, ja que aquesta trama només serveix per demanar aquestes dades.
- El DLC inclourà la longitud de les dades que es rebran en la trama de dades que enviarà com a resposta el transmissor.

2.2.7 Trama d'error

La trama d'error consisteix en dos camps diferents:

- ***Flags* d'error:** hi ha dos tipus de *flags* d'error, els actius i els passius. Els actius consisteixen en 6 bits dominants consecutius, mentre que els passius, en 6 bits recessius consecutius que poden ser sobreescrits pels bits dominants d'altres nodes.
- **Delimitador d'error:** té una longitud de 8 bits recessius. Cada node posa un bit recessiu i, després de ser detectat pels nodes, es transmeten els 7 restants.

Depenent de l'estat del node (estat d'error actiu o passiu, segons l'apartat 2.2.12) s'enviaran errors actius o passius respectivament.

Els *flags* d'error incompleixen la regla de bit *stuffing* (capítol 2.2.9) i destrueixen el format fixe del ACK i EoF. Com a conseqüència, tots els nodes detecten la condició d'error i prenen part en la transmissió de *flags* d'error.

2.2.8 Trama de sobrecàrrega

La trama de sobrecàrrega pot ser enviada per qualsevol node del bus quan:

- El node receptor necessita més temps abans de rebre una nova trama de dades o remota.
- Es detecta un bit dominant durant la fase d'*intermission* (veure apartat 2.2.9).

La trama de sobrecàrrega està formada per dos camps diferents:

- ***Flag* de sobrecàrrega:** té una longitud de 6 bits dominants. D'aquesta manera es destrueix la forma fixa d'*intermission* i tots els altres nodes detecten una condició de sobrecàrrega. En el cas dels nodes que no interpretin correctament aquest *flag*, s'acabaran adonant de la situació d'error al veure que s'incompleix la norma del bit de *stuff* (veure apartat 2.2.10).
- **Delimitador de trama de sobrecàrrega:** té una longitud de 8 bits recessius.

2.2.9 Separació entre trames

Les trames de dades i remotes estan separades de les trames anteriors (independentment del tipus) per un espai anomenat espai *interframe*. Per contra, les trames de sobrecàrrega i d'error no estan separades per cap espai.

L'espai de *interframe* està format pels camps de:

- ***Intermission*:** té una longitud de 3 bits recessius. Durant aquesta fase cap node té permès iniciar una trama de dades o remota. L'única acció que es pot portar a terme és la senyalització d'una sobrecàrrega.
- ***Suspend transmisión*:** aquest camp només s'afegeix en el cas que el node que transmet el missatge estigui en estat d'error passiu (veure apartat 2.2.12). Aquest camp té una longitud de 8 bits recessius consecutius després del camp d'*intermission,* per afegir més temps abans de considerar que el bus està en estat de repòs.
- ***Bus idle*** o estat de repòs**:** té una longitud arbitrària i és l'espai de temps on qualsevol node pot accedir al bus per transmetre.

2.2.10 Bit *stuffing*

La codificació dels bits aplica el mètode de *bit stuffing*. Aquest mètode obliga al transmissor, després d'enviar 5 bits de idèntic valor, a afegir-ne un complementari de valor contrari abans de continuar amb l'enviament. D'aquesta manera, es força una re-sincronització almenys cada 6 bits.

Aquest mètode s'aplica a tots els camps de les trames de dades i remotes a excepció del delimitador de CRC, el bit de ACK, el delimitador de ACK i el camp de EoF.

2.2.11 Detecció i senyalització d'errors

Un node que detecta una condició d'error ha de senyalar-la mitjançant la transmissió d'un *flag* d'error. Els diferents tipus d'error que es poden donar al bus de comunicació CAN són:

- **Error de bit:** durant la transmissió, el sistema llegeix el que s'està enviant. En el cas de que es detecti una diferència entre el bit que s'envia i el que es rep, sempre que no sigui en el camp d'arbitratge o de *acknowledge*, es considerarà que hi ha un error de bit.
- **Error de *stuff*:** es llegeixen al bus sis bits consecutius o més del mateix valor i per tant, s'incompleix amb la norma de bit *stuffing*.
- **CRC error:** el camp de CRC enviat pel node transmissor i el calculat pel node receptor no coincideixen.
- **Error de forma:** un o més bits d'algun camp dins les trames que hauria de contenir un valor fixe definit, en conté un altre que no era l'esperat pel receptor.
- **Error de *acknowledgement*:** durant la transmissió d'una trama CAN de dades o remota, cap node força un nivell de bit dominant. Per tant, s'interpreta que no hi ha cap node llegint el missatge.

2.2.12 Estat dels nodes del bus

Depenent de la quantitat d'errors detectats al bus, els nodes poden estar en tres estats diferents:

- **Error actiu:** el node pot comunicar-se pel bus i enviar *flags* d'error actiu quan es detecta un error.
- **Error passiu:** el node pot seguir comunicant-se pel bus, però tal com s'explica al capítol 2.2.9, després de cada transmissió s'espera una mica més de temps per arribar a l'estat de repòs. A més a més, els *flags* d'error s'envien passius en comptes d'actius.
- **Bus off:** el node no pot seguir comunicant-se pel bus ni influir de cap manera en els seu estats.

L'estat del node dependrà de dos comptadors, el d'errors de transmissió i el d'errors de recepció. Les regles per incrementar o disminuir el comptador d'errors de transmissió són:

- Si el transmissor envia un *flag* d'error, el comptador d'errors de transmissió s'incrementa en 8. Hi ha dues excepcions a on el comptador es quedaria immòbil:
  - Si el transmissor es troba en estat d'error passiu i detecta un error d'ACK, perquè no s'ha rebut un bit dominant en el camp d'ACK i no detecta un bit dominant mentre s'envia el *flag* d'error passiu.
  - Si el transmissor envia un *flag* d'error perquè un error de *stuff* ha succeït durant la fase d'arbitratge quan el bit de *stuff* hauria d'estar just davant del bit de RTR quan hauria de ser recessiu i es enviat com recessiu però es llegeix dominant.
- Si un transmissor detecta un bit d'error mentre envia un *flag* d'error actiu o de sobrecàrrega, el comptador d'errors s'incrementa en 8.
- Després d'enviar correctament un missatge (recepció de ACK correcte i cap error abans del camp de EoF), el comptador d'errors de transmissió es disminueix en 1 sempre que no sigui 0.
- Si després d'un *flag* d'error passiu es detecten 8 bits consecutius dominants, s'incrementa en 8 el comptador d'errors de transmissió. Per cada 8 bits consecutius addicionals, s'incrementa en 8 més el comptador.

Les regles per incrementar o disminuir el comptador d'errors de recepció són les següents:

- Si el receptor detecta un error, el comptador d'errors de recepció s'incrementa en 1, excepte si l'error detectat és un bit d'error durant la transmissió d'un *flag* d'error actiu o de sobrecàrrega.
- Si el receptor detecta un bit dominant com a primer bit després d'enviar un *flag* d'error, el comptador d'errors de recepció s'incrementa en 8.
- Si un receptor detecta un bit d'error mentre envia un *flag* d'error actiu o un *flag* de sobrecàrrega, el comptador d'errors de recepció s'incrementa en 8.
- Si després d'un *flag* d'error passiu es detecten 8 bits consecutius dominants, s'incrementa en 8 el comptador d'errors de recepció. Per cada 8 bits consecutius addicionals, s'incrementa en 8 més el comptador.
- Després de rebre correctament un missatge (cap error fins el bit de ACK i la transmissió correcta d'aquest), el comptador d'errors de recepció es disminueix en 1 sempre que no sigui 0.

Un node passarà a estat d'error passiu si el comptador d'errors de transmissió o de recepció és igual o més gran que 128. Quan s'arriba a una condició que provoca que el node passi a estat d'error passiu, el node ha d'enviar un *flag* d'error actiu.

Un node en estat error passiu passarà a error actiu de nou quan el comptador d'errors de transmissió i recepció sigui igual o menor a 127.

Un node passarà bus off quan el comptador d'errors de transmissió és més gran o igual a 256. Un cop el node està en off, només es permetrà torna a error actiu quan els comptador d'errors de transmissió i recepció és tornen 0 després de que hi hagi 128 successos amb 11 bits recessius consecutius al bus.

2.2.13 Prioritització de missatges

Quan el bus està en repòs qualsevol node pot començar a transmetre. Si dos unitats comencen a transmetre al mateix temps, el conflicte d'accés al bus es resol amb un procés d'arbitratge basat en la utilització de l'identificador del missatge CAN.

En aquest procés d'arbitratge, el node o nodes transmissors comproven el nivell del bit transmès. Si aquest valor és igual, el node pot seguir transmetent i, si no ho és, el node que detecta la diferència ha de deixar de transmetre. Això passa quan el nivell transmès és recessiu i un altre node transmet un nivell dominant. Per tant, quan més baix és el número identificador del missatge (identificador amb un bit dominant en un bit més significatiu) més probabilitats té de guanyar l'arbitratge. Així,es pot establir un ordre de prioritats a través de l'assignació d'identificadors als diferents missatges que es transmetran pel bus.

En el cas de que dos nodes transmetin el mateix identificador, guanyarà l'arbitratge la trama de dades ja que el nivell del bus serà igual fins arribar al camp de RTR, on la trama de dades posarà un nivell de bit dominant.

Un node que ha perdut l'arbitratge podrà intentar enviar el missatge després que s'acabi la fase d'*intermission* (capítol 2.2.9).

2.2.14 Validació de missatges

Les trames de dades i remotes estan protegides mitjançant un polinomi de CRC. El transmissor calcula un *checksum* (sumatori de comprovació) dels bits transmesos i transmet el resultat dins del camp de CRC. El receptor utilitza aquest mateix polinomi per calcular el *checksum* dels bits llegits pel bus. Aquest resultat es compara amb el resultat enviat pel transmissor pel bus.

Si ambdós resultats són iguals, el node receptor transmetrà un bit dominant dins el bit de ACK, sobreescrivint el bit recessiu enviat pel transmissor, tot indicant la correcta recepció del missatge. En cas contrari, els resultats difereixen i el node receptor enviarà una trama d'error després de llegir el bit delimitador de ACK.

El polinomi que s'utilitza pel càlcul del CRC és un polinomi de 15 bits ($x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + x^0$) amb el qual s'aconsegueix, idealment, detectar fins a 5 errors de bit distribuïts de forma aleatòria a la trama. Com això només és cert quan les dimensions de les trames són constants, l'efectivitat només es pot garantir per la detecció d'un únic bit d'error. Malgrat això, la probabilitat que no s'arribi a detectar un

error és molt baixa:

$$\text{Ratio de missatges erronis} \times 4.7 \times 10^{-11}$$

La forma senzilla que es proposa per integrar el càlcul del CRC es representa a la Figura 5.

```
REPEAT
        CRCNXT = NXTBIT EXOR CRC_RG(14);
        CRC_RG(14:1) = CRC_RG(13:0);          // shift left by
        CRC_RG(0) = 0;                        // 1 position
        IF CRCNXT THEN
                CRC_RG(14:0) = CRC_RG(14:0) EXOR (4599hex);
        ENDIF
UNTIL (CRC SEQUENCE starts or there is an ERROR condition)
```

Figura 5. Proposta de codi pel càlcul del CRC

3. EL CONCEPTE D'OPERACIÓ

La Figura 6 mostra a nivell conceptual l'operació del sistema amb les seves entrades i sortides.

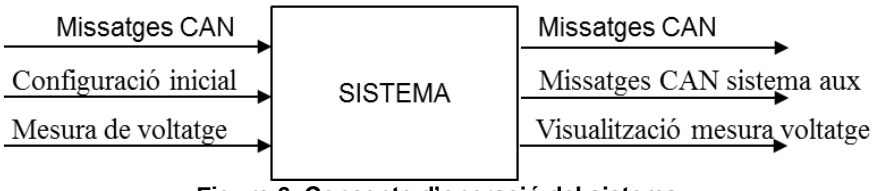


Figura 6. Concepte d'operació del sistema

Les entrades del sistema són:

- Missatges CAN a través del bus de comunicació CAN (recepció)
- Una configuració inicial que permetrà ajustar el sistema per la seva correcta integració al bus de comunicació CAN (velocitat de transmissió, freqüència de transmissió, etc.)
- La mesura de voltatge d'un sensor que tingui sortida de voltatge (exemple: sensor de temperatura TMP35)

Les sortides del sistema són:

- Missatges CAN a través del bus de comunicació CAN (transmissió)
- Missatges CAN convertits a un sistema de comunicació sèrie auxiliar (RS-232)
- Display numèric per la visualització de la mesura del sensor (voltatge, corrent, temperatura, etc.)

3.1 Anàlisi de requisits de l'aplicació

Abans de començar el disseny i seguint les pautes del model en V, primer s'han analitzat els requisits de l'aplicació. Aquests requisits s'han extret de diverses fonts com l'anàlisi teòric, el model conceptual del sistema i bones pràctiques de disseny i programació.

Un dels propòsits del model en V es poder associar els requisits del sistema amb els test de validació d'aquest. Per aquest motiu, en aquest projecte es proposa un sistema d'identificació de requisits que garanteixi la seva traçabilitat entre requisit, disseny i test.

Aquest sistema d'identificació classifica els requisits identificats segons el seu origen, és a dir, segons si són requisits hardware o requisits software. Cadascun d'aquest requisits tindrà un identificador únic amb el següent format:

- **Referencia requisits hardware:** [REQ_HW_XX]
- **Referencia requisits software:** [REQ_SW_XX]

On les XX seran substituïdes per un número.

3.1.1 Hardware

Tenint en compte els objectius del projecte i la definició del concepte d'operació s'ha determinat que el sistema estarà dividit en dos subsistemes. El primer, una plataforma hardware que serà la base del sistema i és on es podrà carregar el software i, el segon, un sistema dissenyat per implementar la resta de funcions hardware que no es poden suplir amb la plataforma base.

Els requisits definits per la plataforma hardware base del sistema són:

[REQ_HW_1] La plataforma base del sistema ha de ser un sistema que estigui disponible al mercat i que integri una FPGA de Xilinx.

[REQ_HW_2] La FPGA de Xilinx integrada a la plataforma base ha de disposar dels recursos suficients per permetre implementar el software, però no excessius per poder assegurar que es fa un ús eficient d'aquests recursos.

[REQ_HW_3] La FPGA de Xilinx integrada a la plataforma base ha de permetre llegir i escriure bit a bit en paral·lel a la velocitat establerta del bus CAN.

[REQ_HW_4] La FPGA de Xilinx integrada a la plataforma base ha de permetre executar processos a velocitats més altes que la velocitat establerta pel bus CAN per poder implementar les diferents fases del temps de bit nominal CAN.

[REQ_HW_5] La FPGA de Xilinx integrada a la plataforma base ha de permetre la transmissió i recepció de missatges a velocitats de fins a 1 Mbps amb una periodicitat d'almenys 1ms entre missatges.

[REQ_HW_6] La plataforma base ha de disposar d'un programador integrat que sigui fàcil d'utilitzar des d'un PC amb sistema operatiu Windows i compatible amb l'entorn de programació de Xilinx.

[REQ_HW_7] La plataforma base ha de disposar d'un mòdul d'expansió d'entrades i sortides que permeti, almenys, implementar les funcionalitats definides al capítol XX. Tenir un mòdul d'expansió d'entrades i sortides digitals amb pins d'alimentació per poder alimentar els mòduls addicionals.

[REQ_HW_8] La plataforma base ha de disposar d'un sistema de comunicació RS-232 integrat per comunicar amb el PC.

Tenint en compte que es compleixen els requisits anteriors, es defineixen els requisits per la resta del sistema hardware:

[REQ_HW_9] Els nivells de tensió de les entrades i sortides de la FPGA s'han d'adaptar a la tensió diferencial del bus CAN, CAN_H i CAN_L.

[REQ_HW_10] S'ha de poder incloure fàcilment una resistència de terminació CAN, ja que el sistema ha de poder-se connectar a diferents punts del bus CAN, siguin extrems del bus o no.

[REQ_HW_11] Ha de permetre mesurar fàcilment la tensió en els punts necessaris per validar el correcte funcionament del hardware i el software, mitjançant punts de proba aptes per sondes d'oscil·loscopi.

[REQ_HW_12] El hardware a dissenyar s'ha de poder connectar i desconnectar de

forma senzilla a la plataforma hardware base a través del mòdul d'expansió d'entrades i sortides.

[REQ_HW_13] La connexió al bus CAN s'ha de poder fer mitjançant un connector DSUB-9 amb el pin-out estàndard pel protocol CAN.

[REQ_HW_14] Qualsevol sistema hardware addicional a la plataforma hardware base s'ha d'alimentar a través d'aquesta. Per tant, s'ha de tenir en compte valors de voltatge i el consum del sistema hardware addicionals.

[REQ_HW_15] La connexió del bus CAN ha d'estar protegida contra descàrregues electromagnètiques per evitar un trencament prematur dels components.

[REQ_HW_16] La connexió del bus CAN condueix senyals d'una freqüència elevada;per tant, el sistema a dissenyar ha d'evitar radiar emissions electromagnètiques i evitar que les d'altres sistemes l'afectin.

[REQ_HW_17] Disposar d'un sistema per mesurar voltatges. El sistema ha de permetre la flexibilitat de poder connectar diferents sensor que tinguin sortida de voltatge no-diferencial.

[REQ_HW_18] Visualització de la mesura de tensió (o valor convertit si fos el cas) a través d'un display de 7 segments i dos dígits.

3.1.2 Software

Els requisits software generals del sistema són:

[REQ_SW_1] El *driver* CAN ha de ser modular per poder escollir entre optimitzar rendiment o recursos utilitzats. Aquesta configuració ha de ser el més senzilla possible perquè no calgui un coneixement detallat de com funciona el *driver*.

[REQ_SW_2] El codi ha de ser net i contenir comentaris entenedors per la seva reutilització i ampliació, manteniment.

[REQ_SW_3] La creació d'aplicacions ha de ser molt senzilla per facilitar-ne el seu ús.

[REQ_SW_4] El sistema ha de ser robust i permetre fer un diagnòstic del seu funcionament.

[REQ_SW_5] Permetre transmissió i recepció de missatges a velocitats de fins a 1 Mbps a una periodicitat de menys d' 1ms entre missatges.

De la part teòrica del bus CAN (veure apartat 2.2), es poden extreure els següents requisits:

[REQ_SW_6] Sincronització inicial de la lectura CAN amb un canvi recessiu-dominant al bus quan aquest està en repòs.

[REQ_SW_7] Re-sincronització de la lectura CAN amb els flancs de pujada i baixada al bus mentre s'està llegint un missatge.

[REQ_SW_8] La re-sincronització només es realitzarà un cop durant el temps de bit.

[REQ_SW_9] La re-sincronització es realitzarà tant amb flancs ascendents com descendents.

[REQ_SW_10] Es podran enviar trames de dades d'acord amb l'encapsulament de dades exposat en l'apartat 2.2.4.

[REQ_SW_11] Es podran llegir trames de dades d'acord amb l'encapsulament de dades exposat en l'apartat 2.2.4.

[REQ_SW_12] Només s'escriuran trames de dades amb identificador no ampliat, és a dir, el camp de les trames ID tindrà unes dimensions de 11 bits i al camp de IDE sempre tindrà nivell dominant.

[REQ_SW_13] Durant la transmissió de missatges CAN s'afegiran els bits de *stuff* necessaris.

[REQ_SW_14] Durant la lectura de missatges CAN es comprovarà que el transmissor del missatge afegeix els bit de *stuff* correctament.

[REQ_SW_15] Durant la lectura de missatges CAN s'eliminaran els bit de *stuff* llegits per quedar-se només amb les dades reals.

[REQ_SW_16] Durant la transmissió de missatges CAN es deixarà de transmetre si algun altre node està transmetent un missatge CAN amb una prioritat més alta.

[REQ_SW_17] Si durant la transmissió de missatges CAN, s'ha aturat la transmissió degut a que un altre node està transmetent un missatge CAN amb una prioritat més alta, es començarà a transmetre el missatge CAN des del principi quan el missatge CAN més prioritari s'hagi acabat d'enviar.

[REQ_SW_18] Durant la transmissió de missatges CAN es calcularà el CRC.

[REQ_SW_19] Durant la transmissió de missatges CAN s'afegirà el valor de CRC calculat al camp de CRC de les trames de dades.

[REQ_SW_20] Durant la transmissió de missatges CAN i en la fase de ACK es comprovarà si algun node ha posat un nivell dominant al bus. Només s'indicarà a l'aplicació que el missatge s'ha enviat correctament, si hi ha almenys un node que reconeix el missatge com a vàlid.

[REQ_SW_21] Després d'escriure un missatge CAN, l'aplicació haurà de fer un *reset* del mòdul encarregat de la transmissió per poder enviar un altre missatge.

Pel control del mòdul des de l'aplicació, s'identifiquen els següents requisits:

[REQ_SW_22] Funció per configurar i inicialitzar la interfície que gestiona el bus de comunicació CAN.

[REQ_SW_23] En el cas que s'intenti configurar i inicialitzar la interfície CAN i no sigui possible, indicar-ne el motiu.

[REQ_SW_24] Indicar a l'aplicació quan un missatge s'ha escrit correctament.

[REQ_SW_25] Indicar a l'aplicació si la interfície CAN s'està utilitzant i, per tant, no està disponible per una nova transmissió.

[REQ_SW_26] Indicar a l'aplicació que s'ha rebut una trama de dades CAN.

[REQ_SW_27] Indicar a l'aplicació si s'ha rebut una trama de dades CAN correcte o perquè no es considera correcte.

[REQ_SW_28] Si la trama de dades llegida és correcta, s'ha de posar el ACK en el camp de ACK per reconèixer la lectura de la trama.

[REQ_SW_29] L'aplicació ha de disposar d'una funció senzilla per enviar trames CAN sense preocupar-se de què hi passa dins.

[REQ_SW_30] L'aplicació ha de disposar d'una funció senzilla per saber si hi ha trames CAN disponibles i si hi ha, llegir-les.

Hi ha una sèrie de funcionalitats que són necessàries pel total compliment amb la ISO 11898 però que, en l'àmbit d'aquest projecte, s'han deixat com a treball futur. Aquestes funcionalitats són:

- Gestió de trames remotes, error i sobrecàrrega. Les trames es llegiran però no es processaran.
- Sistema de sincronització més sofisticat, tenint en compte les diferents fases de temps de bit.

4. DISSENY DEL SISTEMA

4.1 Arquitectura

Tenint en compte els requisits del sistema, aquest s'ha separat en dos mòduls hardware independents:

- PCB de probes Avnet LX9 Microboard
- PCB d'adaptació CAN

La PCB de proves Avnet LX9 Microboard integra una FPGA Spartan-6 que compleix amb els requisits de temps i d'espai que ocuparà l'aplicació. A més a més, al disposar de recursos bastant limitats en comparació amb els models més actuals, ajudarà a fer un ús de recursos més racional i eficient. Aquesta PCB de proves es pot adquirir online i tota la documentació, incloent esquemàtics, és lliure i només cal registrar-se a la pàgina web de Avnet per obtenir-la.

La PCB d'adaptació CAN s'encarrega de:

- Adaptar les senyals digitals d'entrada i sortida de la FPGA Spartan-6, als nivells de tensió diferencials requerits pel protocol de comunicació CAN.
- Conversió d'analògic a digital (ADC) de dos entrades analògiques a la que es poden connectar sensors amb sortida de voltatge.
- Visualització de les mesures de voltatge a través d'un display de 7-segments i dos dígits.

La Figura 7 mostra l'arquitectura del sistema amb els diferents blocs que la formen i les connexions entre ells.

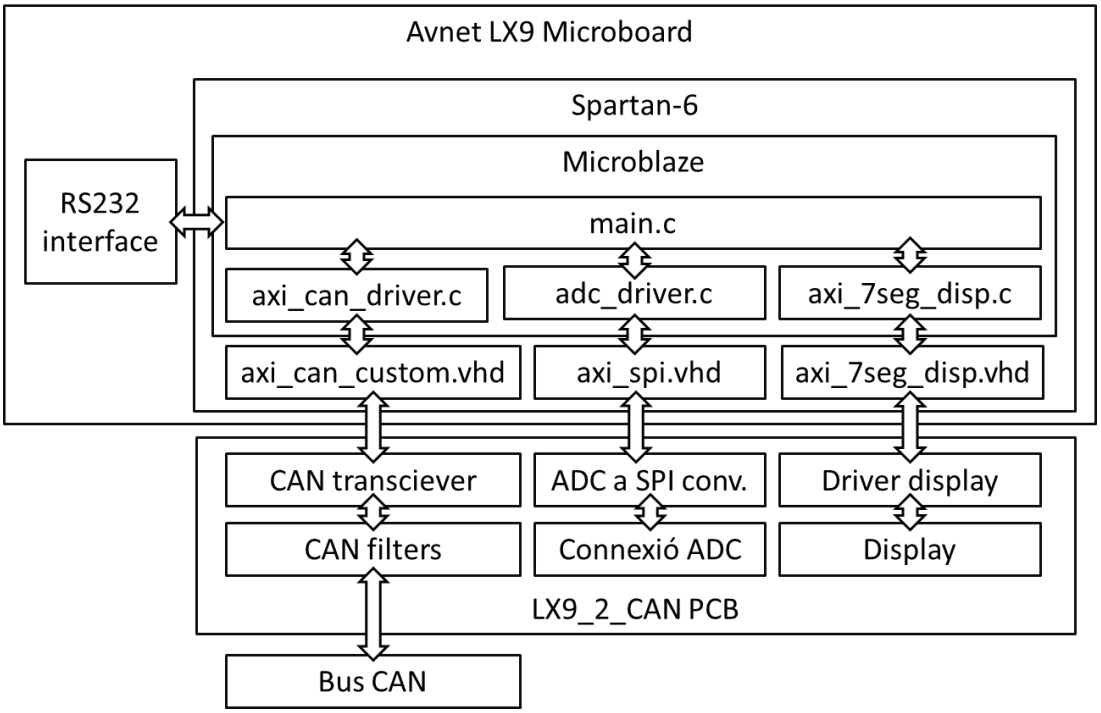


Figura 7. Arquitectura de l'aplicació

En els següents apartats es tractaran els diferents mòduls que formen l'arquitectura del sistema amb més detall, tot distingint entre les parts del hardware i del software.

4.2 Eines de disseny utilitzades

4.2.1 Cadsoft Eagle

Pel disseny de la PCB s'ha fet servir el software de disseny de PCBs EAGLE (*Easily Applicable Graphical Layout Editor*) de l'empresa CADSOFT.

Aquesta eina conté un editor d'esquemàtics pel disseny dels circuits i un editor pel layout de la PCB. Tots dos editors treballen simultàniament de tal forma que qualsevol canvi efectuat a l'esquemàtic es pot observar al layout.

La creació de nous components es fa mitjançant un tercera eina que permet dibuixar el símbol del component, el layout i unir-los en un dispositiu dins d'una llibreria de components.

El software de disseny Cadsoft Eagle és molt popular al món de l'electrònica *DIY* (Do It Yourself) amb empreses com Sparkfun, Arduino o Adafruit proporcionant els fitxers dels seus dissenys o components en format Eagle.

4.2.2 Xilinx Embedded Devolpment Kit

El Xilinx Embedded Development Kit és un entorn de desenvolupament de sistemes encastats de processament que inclou dos entorns de programació diferents, Xilinx Platform Studio (XPS) i Xilinx Software Development Kit (SDK).

L'entorn de programació Xilinix Platform Studio permet construir ràpidament plataformes hardware personalitzades basades en processadors software Microblaze. Disposa tant d'eines gràfiques com de línia de comandes per realitzar-ne la configuració.

També integra diferents assistents de configuració per generar nous sistemes encastats amb molt poc temps. Un dels més interessants i que s'ha usat en aquest projecte és el de *Create or Import Peripheral,* que permet importar o generar les plantilles per desenvolupar mòduls IP de forma fàcil i ràpida.

El Xilinx Software Development Kit és un entorn gràfic des d'on es pot escriure, compilar, carregar i validar aplicacions en GNU C i C++ que corren sobre sistemes hardware desenvolupats amb el Xilinx Platform Studio.

4.2.3 Xilinx Project Navigator

L'entorn de programació Xilinx Project Navigator permet organitzar els arxius de disseny, crear o afegir nous arxius, implementar el disseny i programar targetes que continguin FPGAs de Xilinx.

També és capaç de generar diversos informes amb resultats de les diferents etapes prèvies a la implementació.

4.2.4 Eines de validació

Per tal de poder realitzar probes del sistema de comunicació CAN implementat, és necessari disposar d'un sistema fiable capaç de llegir i escriure missatges CAN.

La solució per la que s'ha optat és la de dissenyar aquesta eina amb un Arduino Uno i un CAN-Shield per Arduino de Sparkfun. El CAN-Shield de Sparkfun permet a l'Arduino poder comunicar-se per CAN, llegir i escriure, a través d'una comunicació SPI gràcies a l'integrat de Microchip, MCP2515.

Modificant els exemples disponibles dins de la llibreria Arduino pel CAN-Shield, s'ha dissenyat un codi que llegeix missatges del bus CAN i els retransmet pel bus sèrie USB. D'aquesta manera els missatges que es llegeixen correctament es poden visualitzar pel monitor sèrie en qualsevol PC.

Aquest programa també està dissenyat perquè pugui enviar periòdicament el missatge CAN desitjat.

La Figura 8 mostra la placa Arduino Uno amb el CAN-Shield de Sparkfun muntat sobre els headers de l'Arduino Uno.

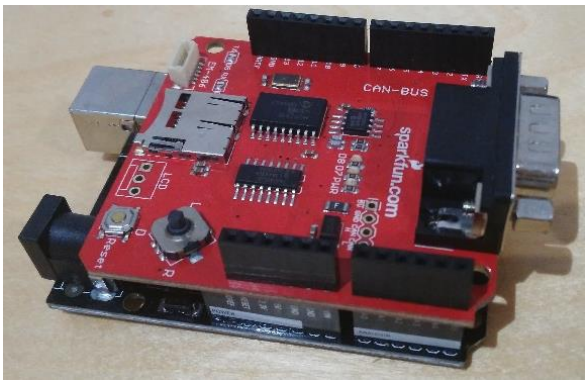


Figura 8. Arduino Uno + CAN-Shield

Per precaució, s'ha de tenir en compte que el pin-out del connector DSUB-9 del CAN-Shield no és estàndard i que, per tant, només es podran utilitzar cables CAN adaptats.

4.3 Hardware

4.3.1 Avnet LX9 Microboard

Pel desenvolupament del projecte, s'ha utilitzat una placa de baix cost del fabricant Avnet anomenada LX9 Microboard (requisit [REQ_HW_1]). Aquesta placa forma part d'un kit de desenvolupament d'aplicacions per la FPGA Spartan-6 i és ideal per prototipar i testejar noves aplicacions de propòsit general, com la que es proposa en aquest projecte. La Figura 9 mostra la vista superior d'aquesta placa i l'emplaçament dels diferents mòduls dels que està formada i la Figura 10 mostra el diagrama de blocs d'aquests mòduls.

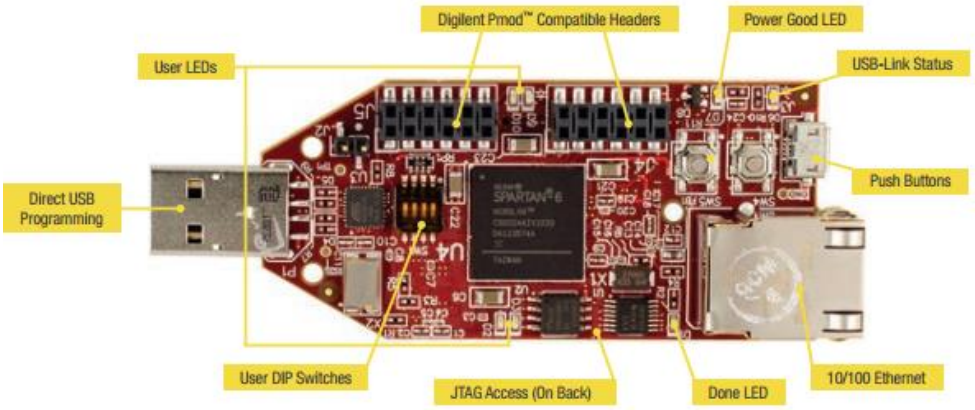


Figura 9. Avnet LX9 Microboard vista superior

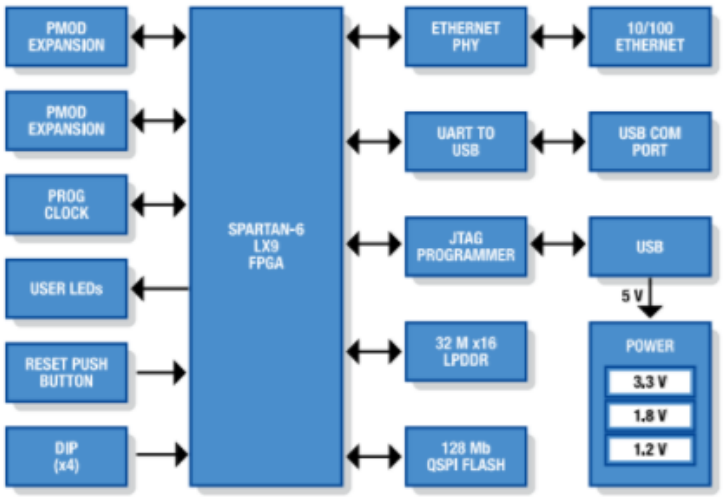


Figura 10. Avnet LX9 Microboard diagrama de blocs

Com es mostra a la Figura 10, entre d'altres mòduls, l' Avnet LX9 Miccoboard integra un programador JTAG accessible a través d'un port USB (requisit [REQ_HW_6]), un bus de comunicació sèrie també accessible a través d'un altre port USB (requisit [REQ_HW_8]) i diversos ports d'expansió d'entrades i sortides (requisit [REQ_HW_7]).

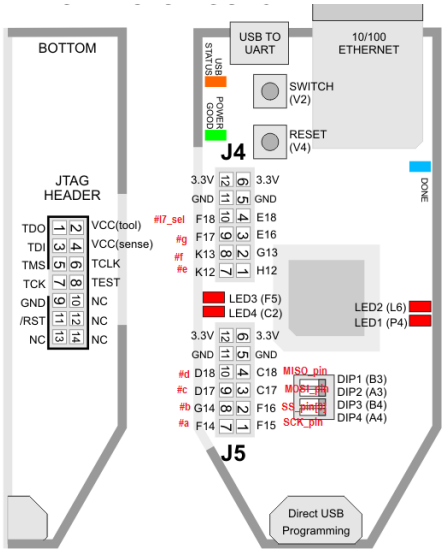


Figura 11. Pin-out Avnet LX9 Microboard

Encara que no s'ha pogut trobat la informació sobre la corrent màxima que es pot absorbir a través dels pins de 3.3V dels mòduls d'expansió de la Avnet LX9 Microboard, mirant els esquemàtics d'aquesta PCB es pot trobar que la font d'alimentació que genera aquesta alimentació és un integrat per la gestió de la potència de sistemes encastats TPS65708 del fabricant Texas Instruments. La sortida en concret que connecta amb el mòdul d'expansió pot donar fins a 400 mA.

Encara que d'aquesta sortida s'alimenten diversos mòduls interns de la Avnet LX9 Microboard, s'estimarà que almenys un 25% de la capacitat està disponible per un mòdul extern, és a dir, 100 mA. Aquest serà el valor de referència per determinar si el sistema compleix o no amb el requisit [REQ_HW_14].

Creant un sistema Microblaze sobre la Spartan-6, es pot obtenir un rellotge de 66 MHz; per tant, en el pitjor dels casos, on la transmissió es requereixi a 1 Mbps, el rellotge permetria executar 66 processos síncrons en el mòdul IP. D'aquesta manera, es pot determinar que la freqüència de la FPGA Spartan-6 és suficient fins i tot en el cas més crític i que es complirà amb els requisits [REQ_HW_3], [REQ_HW_4] i [REQ_HW_5].

Respecte al requisit [REQ_HW_2], es complicat fer una estimació de si la FPGA Spartan-6 disposa de l'espai i recursos necessaris per cobrir totes les necessitats del software però es suposarà que sí.

4.3.2 Microblaze

Microblaze és un microprocessador software de propòsit general dissenyat per Xilinx per a les seves FPGAs. Aquest sistema ofereix moltes possibilitats de configuració com, per exemple, les dimensions de *cache*, els perifèrics i la profunditat del *pipeline*. També permet triar entre optimització per àrea o rendiment, adaptant-se així a les necessitats de cada usuari.

Una de les eines més interessants del sistema Microblaze és que disposa de sistemes d'interconnexió molt diversos que permeten el desenvolupament de gran varietat d'aplicacions encastades, ja que a través d'aquests sistemes, es poden connectar diferents mòduls IP disponibles a les llibreries de Xilinx o crear-ne de personalitzats.

Alguns d'aquests exemples són sistemes de comunicació com SPI, Ethernet, UART, etc. Els sistemes de connexió més populars disponibles en la Spartan-6 són el CoreConnect PLB, el AXI i el AXI-Lite.

A més a més, el sistema Microblaze també és capaç de allotjar sistemes operatius com Linux o FreeRTOS augmentant encara més la versatilitat del sistema.

La configuració del sistema Microblaze per aquest projecte està descrita a l'annex 4.

4.3.3 LX9_2_CAN PCB

Per tal de complir amb els requisits hardware 9 a 17 (veure apartat 3.1.1), s'ha dissenyat una PCB anomenada LX9_2_CAN. La Figura 12 mostra l'arquitectura d'aquesta PCB que conté els diferents mòduls que s'han dissenyat per complir amb els requisits del sistema i que es detallaran en els següents apartats.

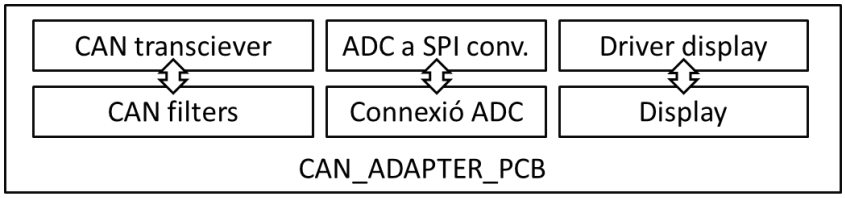


Figura 12. Arquitectura de la LX9_2_CAN PCB

4.3.3.1 Headers Avnet LX9 Microboard

Atenent al requisit [REQ_HW_12], s'ha dissenyat la PCB amb una connexió compatible amb la del placa Avnet LX9 Microboard. D'aquesta manera, la connexió és directa i no calen ni cables addicionals, ni plaques de prototipatge..

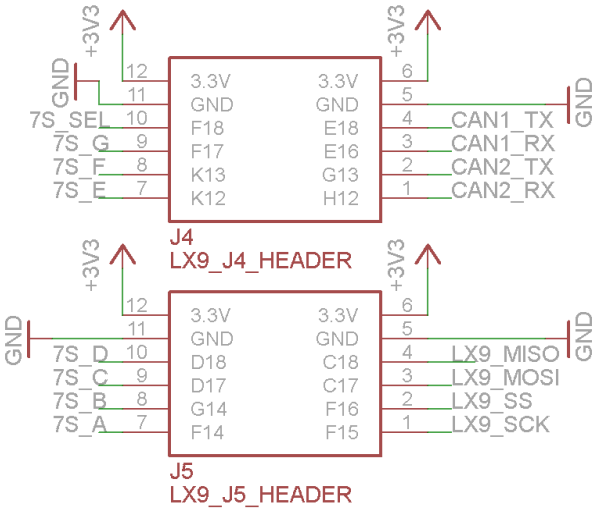


Figura 13. Esquemàtic de la connexió amb els header J4 i J5 de la Avnet LX9 Microboard

La Figura 13 mostra l'esquema de connexions entre els headers de la placa Avnet LX9 Microboard, J4 i J5, amb la PCB dissenyada.

4.3.3.2 Circuit CAN

El circuit CAN de la Figura 14 consisteix en un CAN transceiver, una sèrie de filtres, la resistència de terminació opcional, la protecció contra descàrregues electrostàtiques i el connector extern. Aquest circuit s'ha reproduït dos cops per permetre integrar 2 busos CAN diferents en un mateix node.

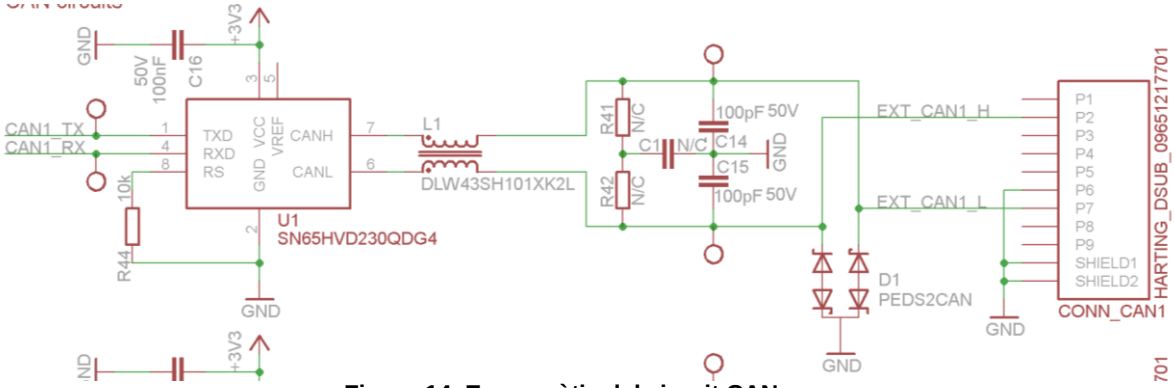


Figura 14. Esquemàtic del circuit CAN

El CAN *transceiver*, component SN65HVD230QDG4, realitza la conversió de les senyals de la FPGA a una senyal diferencial compatible amb el bus CAN:

- Adapta senyals digitals de 0 a 3.3V, aproximadament, a tensions diferencials d'acord a la capa física definida per la ISO 11898, [REQ_HW_9].
- Aquest CAN transceiver està dissenyat per treballar a velocitats de transmissió de fins 1 Mbps, [REQ_HW_5].
- Una alta protecció contra descàrregues electrostàtiques, [REQ_HW_15].
- El voltatge de l'alimentació és 3.3 V amb un consum d'uns 725 mW, per tant, compatible amb la sortida de la Avnet LX9 Microboard, [REQ_HW_14].

El CAN transceiver utilitzat permet habilitar diferents modes de funcionament per optimitzar el consum del sistema. En aquest cas, s'ha afegit una resistència de *pull-down* al pin RS perquè el CAN transceiver estigui permanentment actiu.

El circuit preveu espai a la PCB per dues resistències que, depenent de si es solden o no al circuit, permeten realitzar la funció de resistència de terminació CAN, [REQ_HW_10]. En els esquemàtic, aquestes dues resistències s'han definit com N/C, no connectades, ja que només es soldaran si són necessàries.

Per tal de complir amb [REQ_HW_16], el circuit inclou dos filtres. El primer és una bobina *choke*, L1, que permet suprimir el soroll de mode comú sense empitjorar el comportament de les senyals d'alta freqüència. El segon està format per dos condensadors, C14 i C15, en cas que no s'inclogui la resistència de terminació, o tres, C14, C15 i C1, en el cas que s'inclogui la resistència de terminació. Aquest segon filtre també suprimeix el soroll de mode comú però d'alta freqüència.

L'integrat PEDS2CAN permet augmentar la protecció contra descàrregues electrostàtiques al incloure dos supressors de voltatge transitori específics per bus CAN, [REQ_HW_15].

Per últim, s'inclou el connector DSUB-9 amb el pin-out estàndard del bus CAN per connectors DSUB-9. Amb aquest connector es compleix amb el requisit

[REQ_HW_13].

4.3.3.3 Display

Per la visualització de la mesura de voltatge, [REQ_HW_17], s'ha seleccionat el display de 7 segments i dos dígits DA03-11SURKWA. Com es mostra a la Figura 15, cada segment que forma cadascun del dos dígits d'aquest display és un *led* i tots els ànodes de cadascun dels segments que formen el dígit estan connectats entre ells. En canvi, el càtode d'un segment només està unit al càtode corresponent al mateix segment però de l'altre dígit. Amb aquesta configuració és possible mostrar dos dígits simultàniament però no de diferent valor. Per fer-ho, cal anar intercanviant el dígit actiu i la selecció de segments a una velocitat suficientment alta perquè el ull humà no pugui distingir que fan pampallugues.

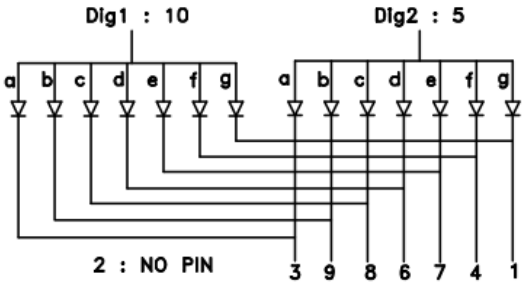


Figura 15. Esquemàtic intern del display de 7-segments i dos dígits DA03-11SURKWA

A la Figura 16 es mostra el circuit implementat per controlar aquest display des de la FPGA. La idea és que els càtodes, amb la resistència de polarització adequada, poden anar connectats directament a les sortides de la FPGA, ja que cada sortida només haurà de drenar com a molt la corrent de polarització de dos *leds*. Per contra, la sortida cap als ànodes comuns, selecció de dígit, ha de ser capaç de proporcionar la corrent necessària per alimentar fins a 7 *leds*. Per tant, s'ha afegit un integrat capaç de proporcionar aquesta corrent, porta inversora de 6 canals que, a més a més, permet que amb una sola sortida de la FPGA es pugui seleccionar el dígit desitjat.

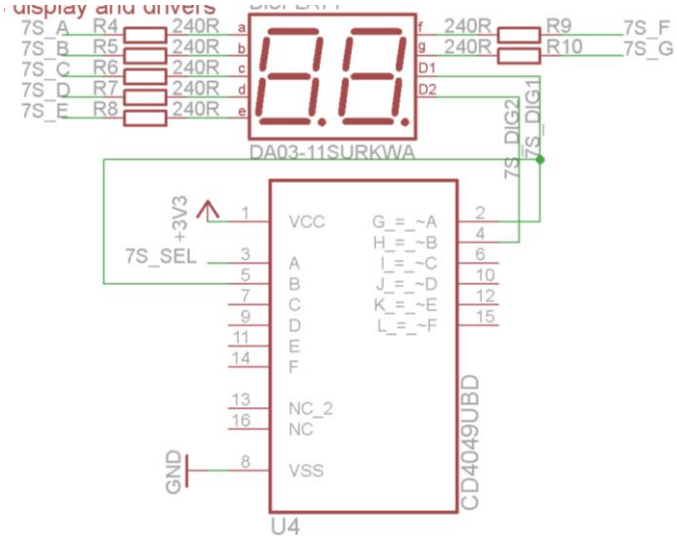


Figura 16. Esquemàtic del display DA03-11SURKWA i el seu *driver*

4.3.3.4 Mesura de voltatge

Al contrari que la majoria de microcontroladors, una FPGA no integra un convertidor analògic-digital. Per tant, la conversió s'ha de realitzar externament. En aquest cas, la solució triada per mesurar el voltatge, [REQ_HW_17], és la utilització d'un integrat que realitza la conversió analògica a SPI. El integrat utilitzat, MCP3002, permet mesurar dos canals amb una resolució de 10 bits.

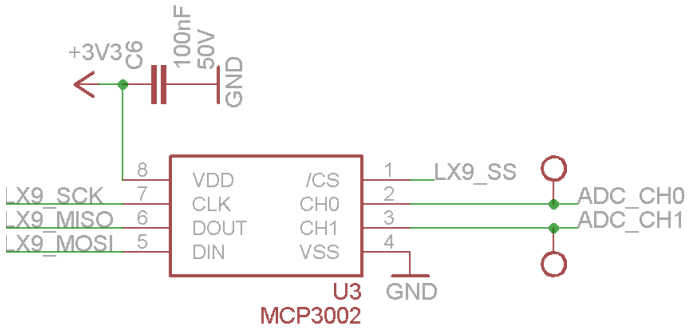


Figura 17. Esquemàtic del convertidor analògic a SPI

Com es mostra a la Figura 17, la integració de l'integrat que realitza la conversió és molt senzill. L'alimentació del dispositiu pot anar entre 2.7 i 5V, així que és compatible amb la Avnet LX9 Microboard i no incompleix el requisit [REQ_HW_14]. L'únic desavantatge que es troba és que al reduir la tensió de alimentació de l'integrat es perd velocitat de mostreig i disminueix el rang d'entrada.

4.3.3.5 Layout

El disseny de la PCB s'ha fet en dues capes, s'ha situat un pla de GND a la capa inferior i de 3.3 V a la capa superior. Això ajuda a la connexió de les senyals i redueix la radiació d'emissions electromagnètiques.

A la imatge de la Figura 18 s'inclou el layout del disseny final de la PCB, així com una llegenda amb les capes més importants de la PCB.

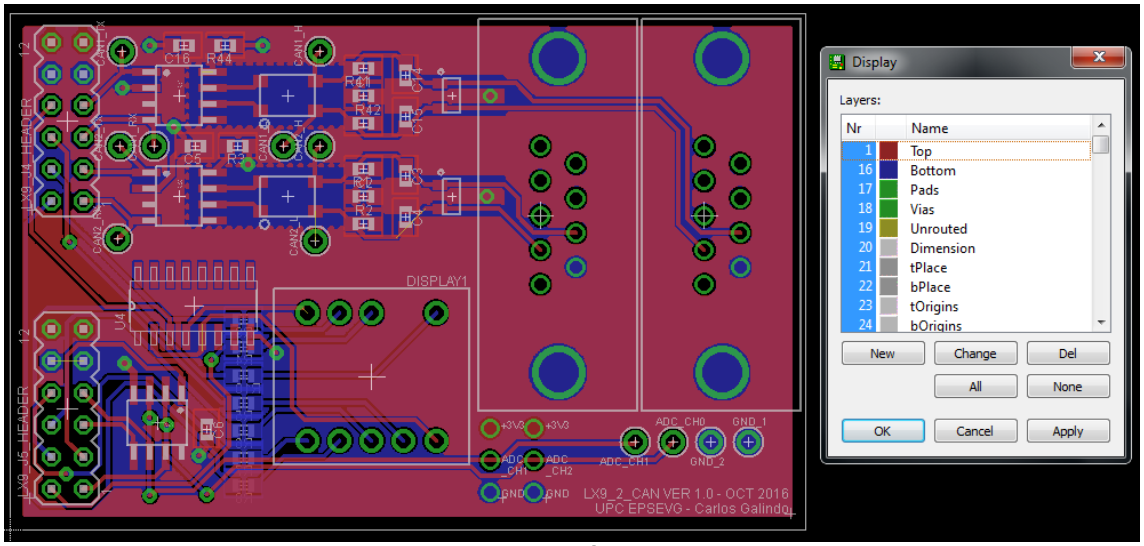


Figura 18. Layout de la PCB LX9_2_CAN versió 1.0 amb la llegenda de les diferents capes

En el disseny de la PCB s'ha donat especial importància a les senyals d'alta freqüència com les del CAN i SPI (convertidor analògic-SPI), ja que són les que poden generar més radiacions electromagnètiques. S'ha procurat que en la mesura del possible aquestes senyals siguin el més curtes possibles i amb poques curvatures.

En el cas de la senyal CAN després del CAN transceiver, és a dir, quan ja està en mode diferencial, s'ha tingut en compte que totes dues senyals, CAN_H i CAN_L, viatgin el més properes possible i que les longituds de via siguin molt semblants. També s'ha retret el pla de 3.3 V d'algunes zones on quedava atrapat entre totes dues línies per no induir-hi soroll. El punt més crític ha sigut col·locar els punts de mesura necessaris per validar la comunicació CAN de manera que tinguin els mínims efectes perjudicials sobre la radiació electromagnètica.

S'ha intentat mantenir les dimensions de la PCB el més reduïdes possible mantenint una forma rectangular perquè fabricació de la PCB és més assequible, però finalment s'ha hagut d'adaptar i condicionar a les mides dels connectors i el display.

A l'hora de fer el *layout*, s'ha afegit vies a GND i 3.3V per alimentar o referenciar, si fos necessari, el sensor que s'utilitzi. També s'ha afegit punts de prova connectats a GND per connectar la massa de les sondes d'oscil·loscopi.

Cal destacar que la PCB conté el nom, la versió i la data de generació dels fitxers de fabricació per poder fer un seguiment en el cas que es realitzés una nova versió en el futur.

4.3.3.6 Fabricació i muntatge

Un cop acabat el disseny, s'ha verificat que aquest fos correcte i complís amb els requisits i capacitats del fabricant de PCBs seleccionat, en aquest cas PCB-Way, amb un sistema de verificació automàtica inclòs en el software de disseny de PCBs.

Aquest sistema de verificació permet detectar errors (per exemple: una via superposada amb una altra diferent) o incompatibilitats amb les capacitats de producció del fabricant (per exemple: vies molt juntes unes de les altres o forats de mides de broca no disponibles).

Un cop passada la inspecció automàtica del disseny, es van generar els arxius de fabricació i enviar al fabricant per la seva producció. El resultat de la fabricació de la PCB es mostra a la Figura 19.

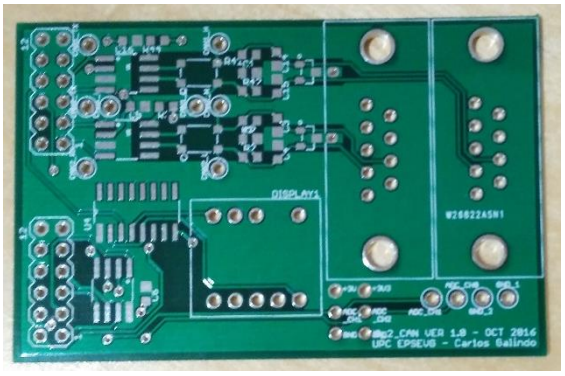


Figura 19. PCB LX9_2_CAN

A la Figura 20 es mostra el resultat de la PCB soldada.

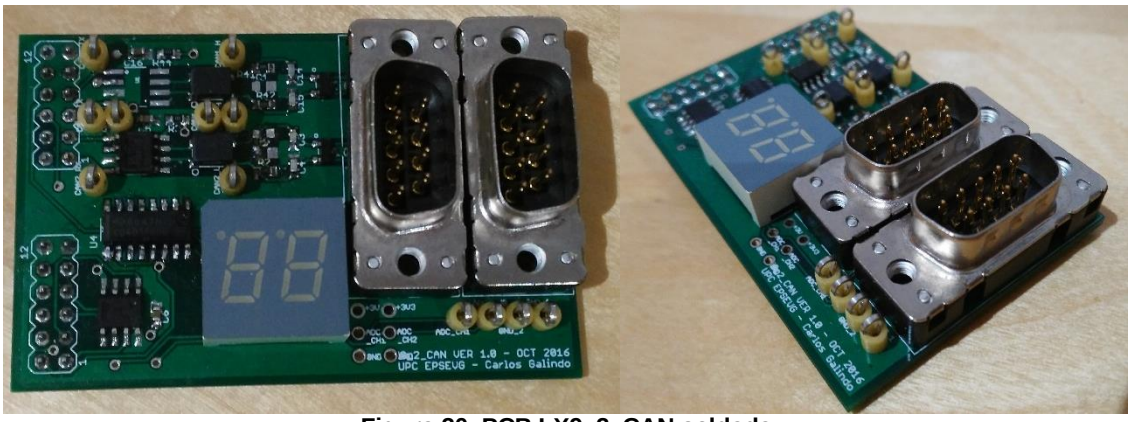


Figura 20. PCB LX9_2_CAN soldada

La PCB s'ha soldat manualment però amb molta cura de que les soldadures estiguessin ben fetes, és a dir, de manera que no creïn capacitats paràsites, ni es puguin trencar amb facilitat degut a vibracions o altres esforços mecànics sobre la placa.

4.4 Software

El desenvolupament software s'ha realitzat en diferents nivells i plataformes. La Figura 21 mosta l'arquitectura del software que es carrega sobre la FPGA Spartan-6.

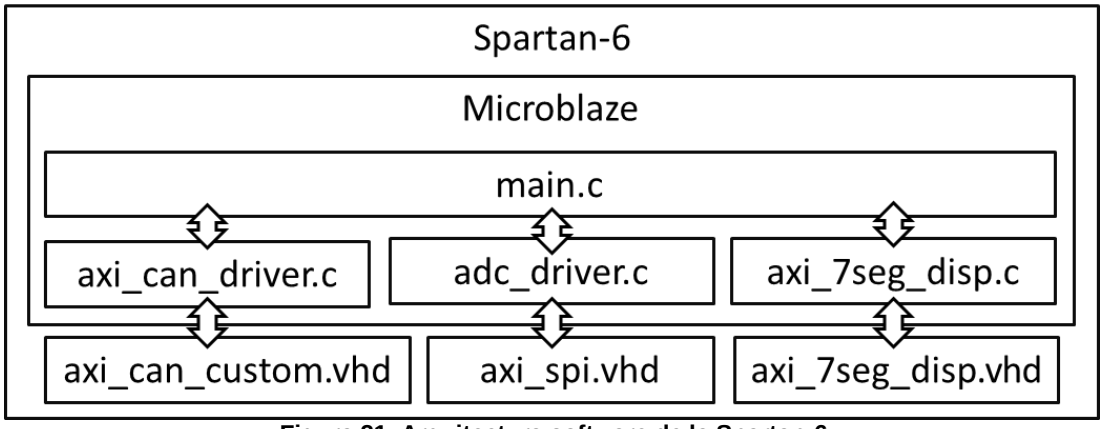


Figura 21. Arquitectura software de la Spartan-6

El sistema Microblaze permet executar la aplicació escrita en codi C i connectar els *driver*s dels perifèrics d'aquesta aplicació amb la seva implementació en baix nivell escrita en VHDL.

El mòdul axi_spi.vhd està disponible a les llibreries de Xilinx amb llicencia lliure, així que no cal preocupar-se per la comunicació SPI. En canvi, els mòduls IP axi_can_custom.vhd i axi_7seg_disp.vhd s'han dissenyat expressament.

4.4.1 Mòdul IP AXI_CAN_CUSTOM

El mòdul IP AXI_CAN_CUSTOM implementa la lectura i escriptura de missatges CAN, d'acord amb els requisits software definits anteriorment. Les dades que s'escriuen al bus es controlen a través d'una interfície entre el mòdul IP i el axi_can_*driver*.c que consisteix en una sèrie de registres accessibles des dels dos mòduls que permet compartir dades.

Per crear el mòdul IP, s'ha fet servir l'assistent de configuració disponible dins el XPS. Els passos seguits són:

1. Hardware → "*Create or Import Peripheral...*"
2. Seleccionar la opció "*Create templates for a new peripheral*" i clicar a "*Next*"
3. Seleccionar l'opció "*To an EDK user repository*" i definir la ruta del repositori on es vol guardar i clicar a "*Next*"
4. Definir un nom i versió pel mòdul IP, en aquest cas, axi_can_custom i versió 1.11a, i clicar a "*Next*"
5. Selecciona el sistema d'interconnexió "*AXI4-Lite: Simple, non-burst control registre style interface*" i clicar a "*Next*"
6. Només deixar seleccionada l'opció "*User Logic Software Registers*" i clicar a "*Next*"
7. Seleccionar el nombre de registres necessaris. En una versió anterior, s'havia utilitzat el màxim nombre de registres disponibles, però després s'ha pogut optimitzar l'ús d'aquests registres i fer servir només la meitat, 16. D'aquesta manera no només s'estalvia espai al reduir el nombre de registres sinó que la gestió del bus també es simplifica perquè les adreces són de 16 bits en comptes de 32. Després, clicar a "*Next*"
8. Deixar seleccionades les senyals per defecte i clicar a "*Next*"
9. No seleccionar l'opció de generació de Bus Functional Models i clicar "*Next*"
10. Seleccionar l'opció de "*Generate ISE and XST project files to help you implement the peripheral using XST flow*" i clicar a "*Next*"
11. Clicar a "*Finish*"

Seguint aquest procés es creen diferents arxius, entre ells, els fitxers VHDL que serveixen com a plantilla per escriure el mòdul IP.

Abans d'incloure el codi del mòdul IP, s'han configurat les plantilles axi_can_custom.vhd i USER_LOGIC_I perquè totes dues entitats disposin dels ports d'entrada i sortida cap a l'exterior i interconnectar-los.

Amb els fitxers generats automàticament i els codis dissenyats, l'estructura del mòdul IP dissenyada té la següent estructura:

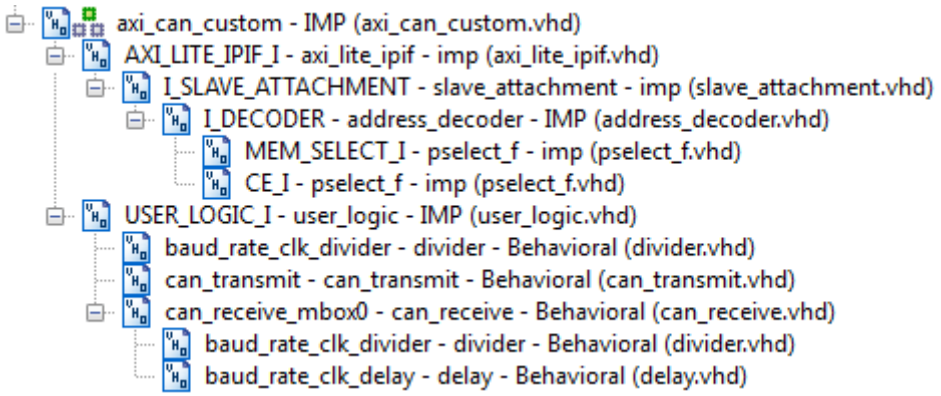


Figura 22. Estructura del mòdul IP AXI_CAN_CUSTOM

4.4.1.1 Entitat DIVIDER

L'entitat DIVIDER està definida al fitxer DIVIDER.vhd. La Figura 23 mostra la representació gràfica de les entrades i sortides del sistema.

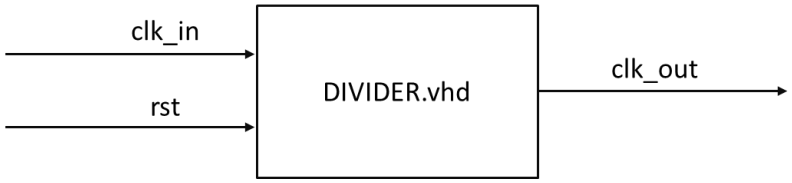


Figura 23. Entitat divider.vhd

La funció de l'entitat DIVIDER és dividir la senyal de rellotge del sistema per obtenir una senyal de rellotge a una freqüència més baixa. La entrada *rst* permet inicialitzar la senyal de rellotge de sortida.

Aquesta entitat s'utilitza per generar el rellotge per la màquina d'estats que gestiona el mòdul IP i per la sincronització de la lectura i escriptura de missatges CAN. L'entrada *rst* és d'especial importància per permetre la re-sincronització del rellotge de lectura. A més a més, el *reset* ha d'assegurar que hi hagi un flanc de pujada quan s'activa per disparar la sincronització de la màquina d'estats de la lectura CAN amb el canvi de nivell del pin d'entrada CAN.

4.4.1.2 Entitat DELAY

L'entitat DELAY està definida al fitxer DELAY.vhd. La Figura 24 mostra la representació gràfica de les entrades i sortides del sistema.

Figura 24. Entitat delay.vhd

La funció de l'entitat DELAY és endarrerir el flanc de pujada generat per l'entitat DIVIDER un número fixe de senyals del rellotge del sistema. D'aquesta manera es possible endarrerir el mostreig del nivell del bus fins a que s'hagi estabilitzat.

4.4.1.3 Entitat CAN_TRANSMIT

L'entitat CAN_TRANSMIT està definida al fitxer CAN_TRANSMIT.vhd. La Figura 25 mostra la representació gràfica de les entrades i sortides del sistema.

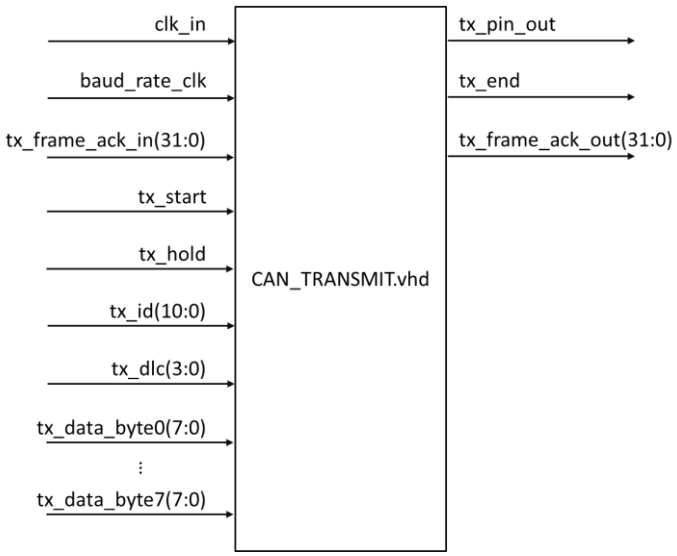


Figura 25. Entitat can_transmit.vhd

L'entitat CAN_TRANSMIT conté una màquina d'estats que permet implementar la transmissió de missatges CAN. Les senyals d'entrada s'utilitzen dins la màquina d'estats per:

- **clk_in:** senyal de rellotge que dispara l'execució del procés que conté la màquina d'estats.
- **baud_rate_clk:** és un rellotge més lent que el del sistema i que marca la velocitat de transmissió dels bits.
- **tx_frm_ack_in(31:0):** és una senyal que només es carrega a la sortida tx_frm_ack_out quan el missatge s'ha enviat correctament.
- **tx_start:** dispararà des de l'exterior l'enviament del missatge CAN.
- **tx_hold:** és una senyal d'entrada que permet aturar la transmissió del missatge CAN i transmetre'l de nou quan la senyal es desactiva.
- **tx_id(10:0):** conté l'identificador del missatge CAN que es vol enviar. Només es carregarà un cop per transmissió, ja que només s'actualitza quan arriba el *flag* tx_start.
- **tx_dlc(3:0):** conté el nombre de bytes a transmetre dins del missatge CAN que es vol enviar. Només es carregarà un cop per transmissió, ja que només s'actualitza quan arriba el *flag* tx_start.
- **tx_data_byte0(7:0) a tx_data_byte7(7:0):** conté el valor de cadascun dels bytes que es vol transmetre dins del missatge CAN. Només es carregarà un cop per transmissió, ja que només s'actualitza quan arriba el *flag* tx_start.

La màquina d'estats genera les següents sortides:

- **tx_pin_out:** senyal a través de la qual es transmeten els missatges CAN
- **tx_end:** un *flag* que indica que l'entitat ha acabat de transmetre el missatge CAN
- **tx_frm_ack_out(31:0):** serveix per assegurar que el missatge enviat és realment el que es volia enviar.

La Figura 26 i Figura 27 mostren els diagrames implementats per l'entitat can_transmit. El diagrama principal, Figura 26, controla el mecanisme de bit *stuffing*, permet aturar la transmissió quan ho indiqui l'entitat can_receive (en principi quan el node perd l'arbitratge) i la retransmissió dels missatges aturats. La idea és poder

gestionar quan s'executa la màquina d'estats que s'encarrega de transmetre correctament els diferents camps que formen les trames de dades, Figura 27.

Com s'observa a la Figura 26, la implementació del mecanisme de bit *stuffing* es basa en comptar bits d'igual valor consecutius. Si el compte arriba a 5, s'afegeix un bit de *stuff*, és a dir, un bit de valor contrari al actual. El següent bit que s'enviarà després d'enviar el bit de *stuff* serà el bit que tocava enviar abans i que havia estat guardat a la variable next_bit_to_send.

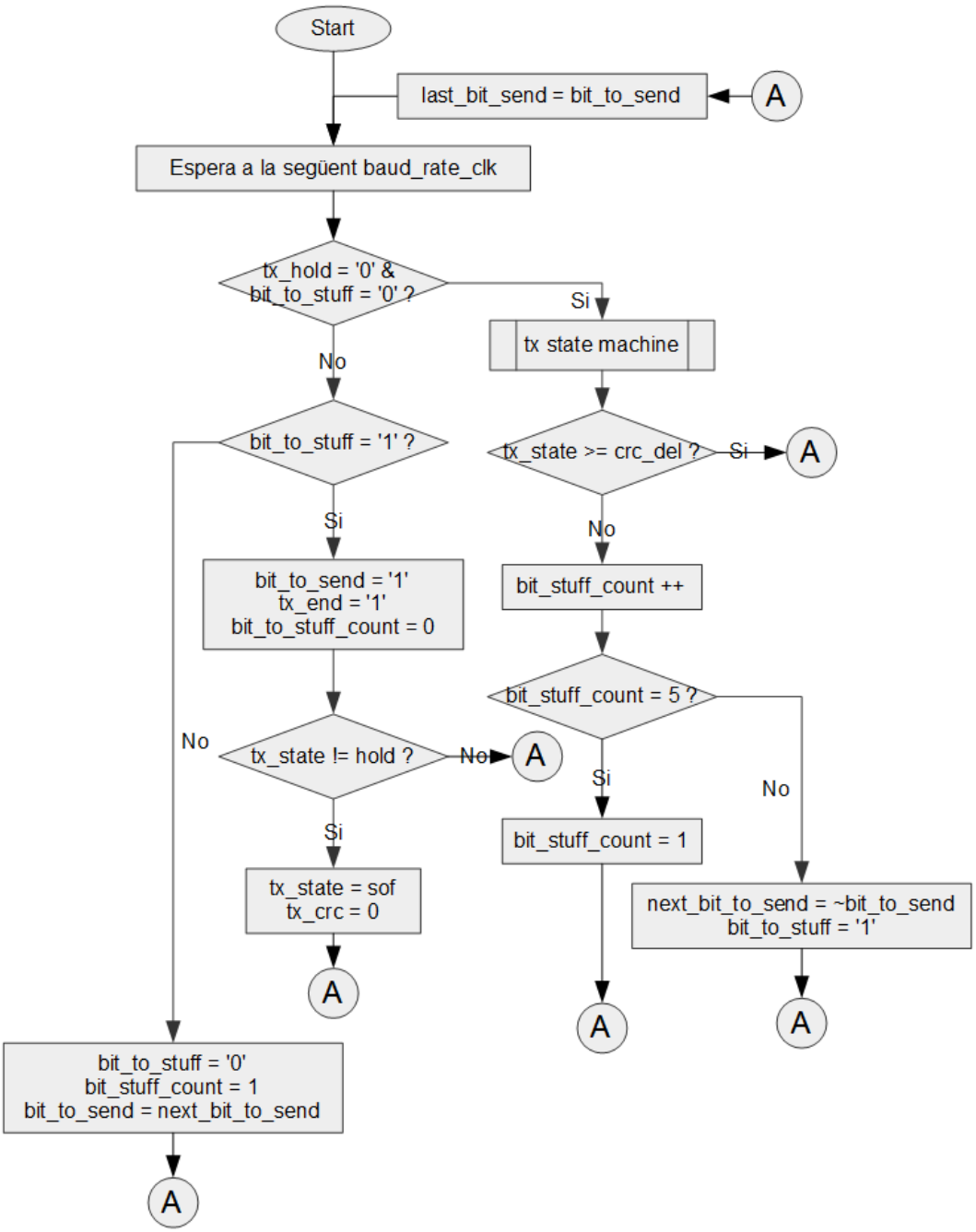


Figura 26. Diagrama de flux de l'entitat can_transmit

La interpretació del diagrama de la màquina d'estats tx_state_machine és el següent: cada el·lipse representa un estat i, a cada crida d'aquesta màquina d'estats des de la

general, permetrà saltar a un altre estat o retornar al mateix passant per una sèrie d'accions que dependran de diversos paràmetres. És important destacar que com a màxim es podrà canviar un cop d'estat per crida i que la següent crida començarà des de l'estat en què s'ha acabat la execució anterior.

Per més informació sobre l'entitat can_transmit, veure el codi amb els comentaris explicatius a l'annex 3.

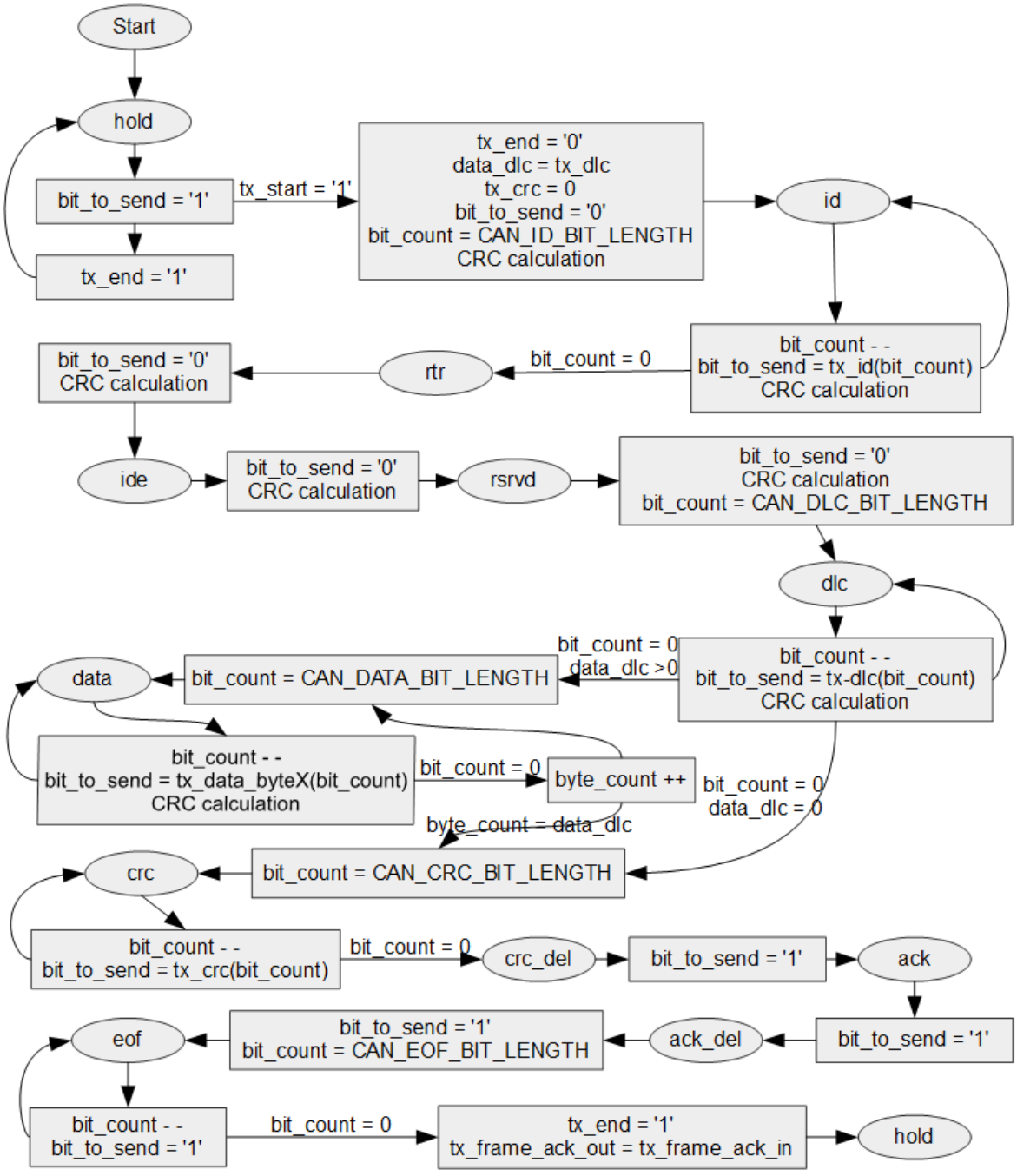


Figura 27. Màquina d'estats tx_state_machine

4.4.1.4 Entitat CAN_RECEIVE

L'entitat CAN_RECEIVE està definida al fitxer CAN_RECEIVE.vhd. La Figura 28 mostra la representació gràfica de les entrades i sortides del sistema.

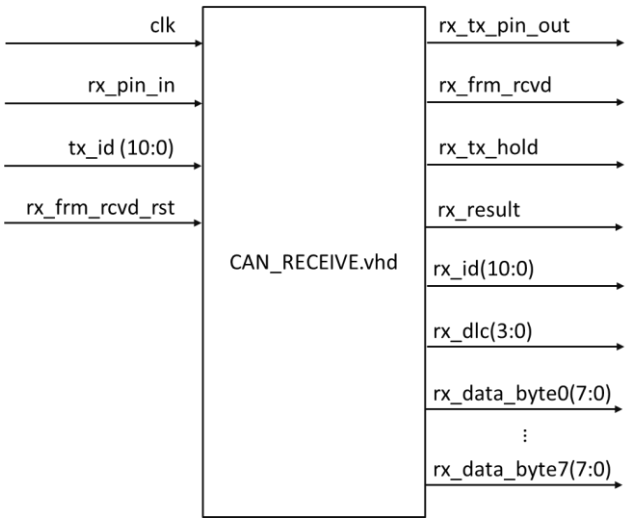


Figura 28. Entitat can_receive.vhd

L'entitat CAN_TRANSMIT conté una màquina d'estats que permet implementar la transmissió de missatges CAN. Les senyals d'entrada s'utilitzen dins la màquina d'estats per:

- **clk:** senyal de rellotge que dispara l'execució del procés que conté la màquina d'estats.
- **rx_pin_in:** senyal a través de la qual es llegeixen el missatges CAN.
- **tx_id(10:0):** conté l'identificador del missatge CAN que es vol enviar. Permet implementar l'arbitratge de missatges segons l'identificador més prioritari per poder bloquejar l'enviament.
- **rx_frm_rcvd_rst:** senyal per realitzar un *reset* del *flag* de missatge rebut i de les dades llegides anteriorment.

La màquina d'estats genera les següents sortides:

- **rx_tx_pin_out:** senyal que permet enviar el bit d'*acknowledge* en el cas que es llegeixi correctament el missatge CAN.
- **rx_frm_rcvd:** és un *flag* que indica que l'entitat ha rebut un missatge CAN independentment de si és correcte o no.
- **rx_hold:** és una senyal de sortida que indica la transmissió d'aturar o no, i si s'ha retingut, quan pot ser reactivada.
- **rx_result(3:0):** indica si el missatge rebut és correcte o quin és el motiu pel qual no es considera correcte.
- **tx_id(10:0):** conté l'identificador del missatge CAN rebut. Només es carregarà si el missatge rebut és correcte.
- **tx_dlc(3:0):** conté el nombre de bytes del missatge CAN rebut. Només es carregarà si el missatge rebut és correcte.
- **tx_data_byte0(7:0) a tx_data_byte7(7:0):** conté el valor de cadascun dels bytes del missatge CAN rebut. Només es carregarà si el missatge rebut és correcte.

La Figura 28 i Figura 29 mostren els diagrames implementats per l'entitat can_receive. El diagrama principal, Figura 28, controla el mecanisme de bit *stuffing*, elimina els bits de *stuff*, comprova si tenen el valor correcte i sincronitzar i re-sincronitzar la lectura

quan hi ha un canvi de nivell al bus de comunicació CAN.

Com es pot observar a la Figura 28, la implementació del mecanisme de bit *stuffing* es basa en comptar bits d'igual valor consecutius. Si aquesta compta arriba a 5, s'esperarà llegir un bit de valor contrari al actual. Si es així, aquest bit es descartarà i sinó, es detectarà un error i es pararà de llegir.

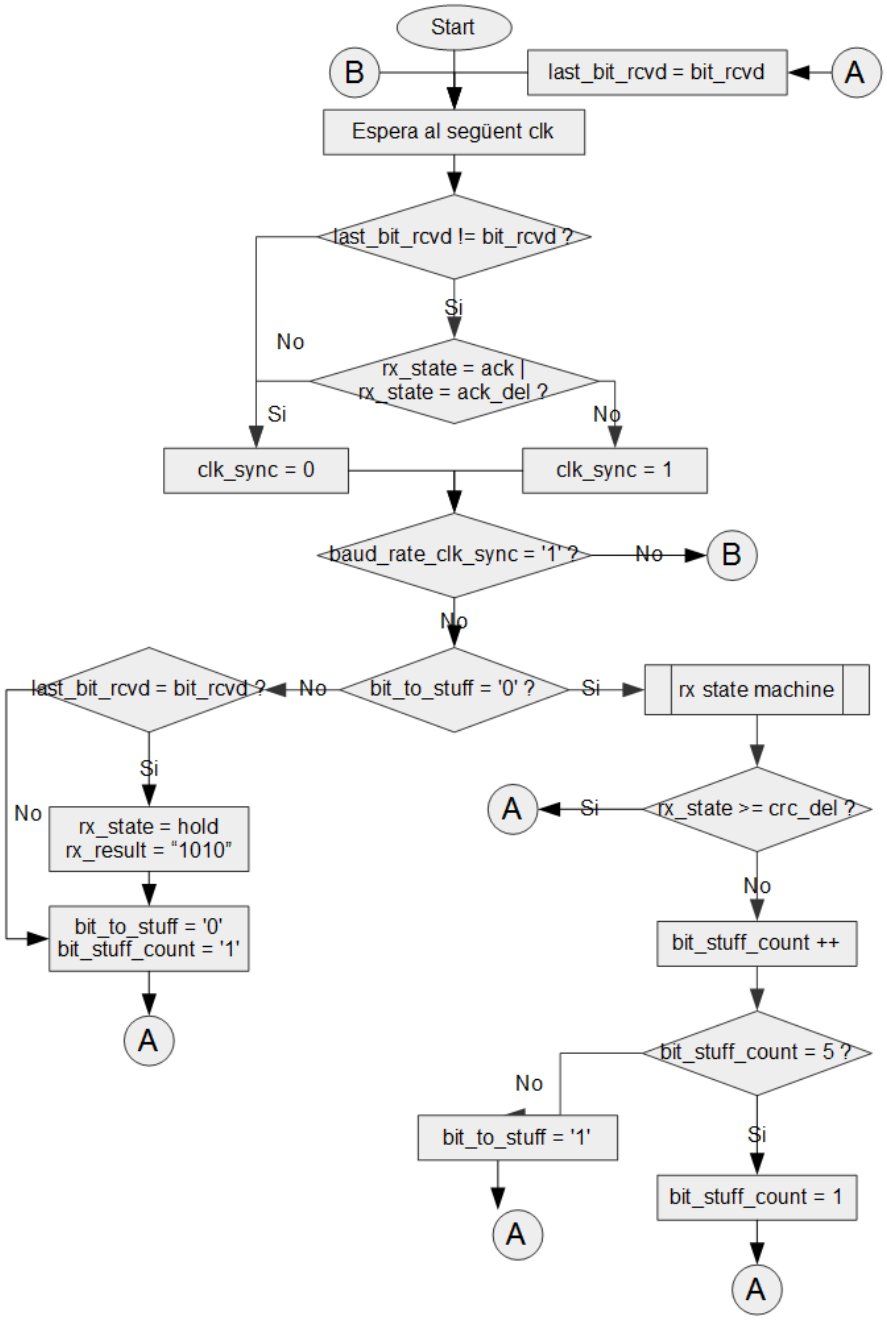


Figura 29. Diagrama de flux de l'entitat can_receive

La interpretació del diagrama de la màquina d'estats rx_state_machine és la mateixa que en l'apartat 4.4.1.3: cada el·lipse representa un estat i a cada crida d'aquesta màquina d'estats des de la general, permetrà saltar a un altre estat o retornar al mateix

passant per una sèrie d'accions que dependran de diversos paràmetres. És important destacar que com a màxim es podrà canviar un cop d'estat per crida i que la següent crida començarà des de l'estat en què s'ha acabat l'execució anterior.

Per més informació sobre l'entitat can_receive, veure el codi amb els comentaris explicatius a l'annex 3.

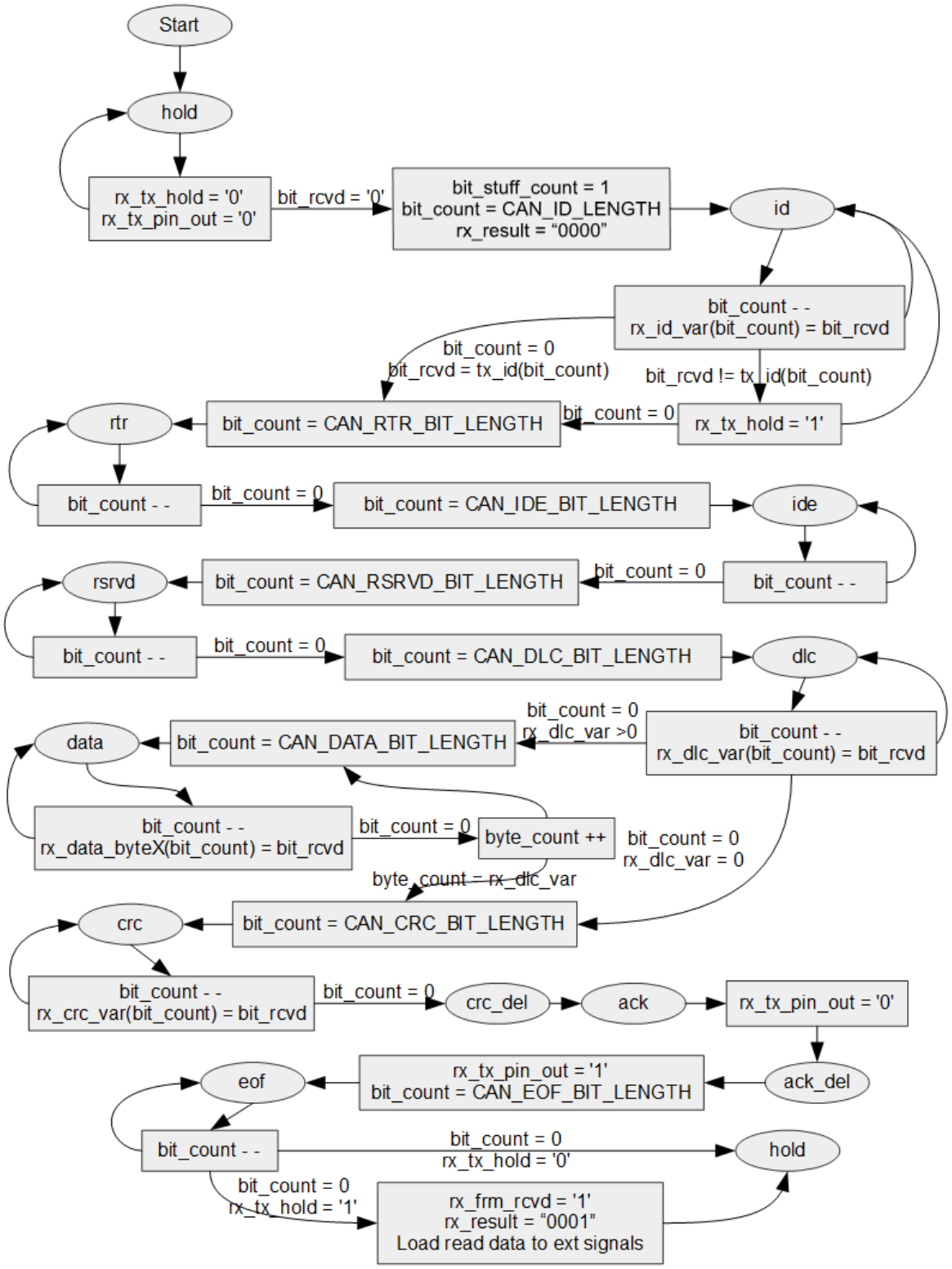


Figura 30. Màquina d'estats rx_state_machine

4.4.1.5 Paquet CAN_VHDL_PACKAGE

Pel càlcul del CRC és necessari implementar un codi com el que es descriu a l'apartat 2.2.14. Afegir aquest codi dins la màquina d'estats de la transmissió o recepció de missatges CAN embrutaria molt codi, així que s'ha implementat el càlcul del CRC dins una funció que podrà ser utilitzada des de qualsevol entitat.

Per dissenyar una funció és necessari emplaçar-la dins d'un paquet que podria englobar-ne varies més, encara que en aquest cas només hi haurà una anomenada CRC_CALCULATION. El codi d'aquesta funció es pot consultar a l'annex 3.

4.4.1.6 Entitat USER_LOGIC

L'entitat CAN_RECEIVE està definida al fitxer USER_LOGIC.vhd. La Figura 31 mostra la representació gràfica de les entrades i sortides del sistema.

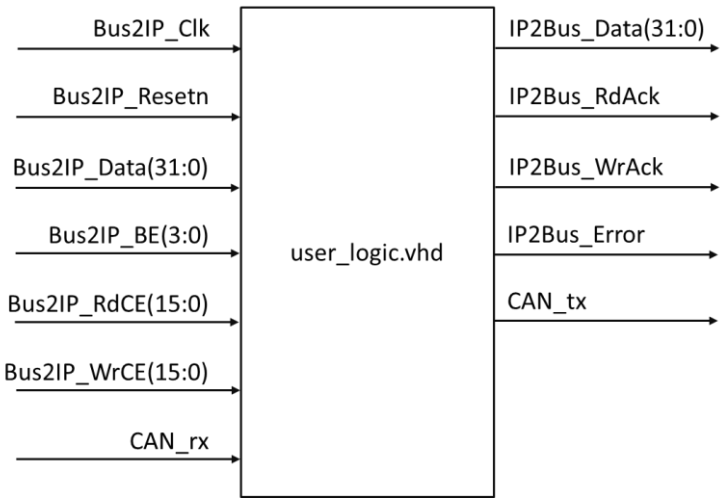


Figura 31. Entitat user_logic.vhd

L'entitat user_logic fa d'unió entre el codi en VHDL, mòdul IP, i el codi en C, *driver*s. Aquesta unió es realitza mitjançant 16 registres de 32 bits cadascun. A través d'aquests registres, l'entitat user_logic implementa les següents funcionalitats:

- Redirecció dels *flags* cap a les instàncies can_transmit i can_receive o el *driver* CAN. Depèn de cada cas.
- Enviament del contingut de les trames llegides pel bus CAN cap al *driver* CAN
- Recepció del contingut de les trames CAN a enviar des del *driver* CAN
- Màquina d'estats simple (veure Figura 32) per gestionar la instància de l'entitat can_transmit. Control del començament de l'enviament de trames CAN, càrrega del contingut del missatge CAN a enviar, inhibició de l'enviament si la instància està ocupada, *reset* de la instància i sincronització de la senyal de rellotge que marca la freqüència de transmissió de missatges CAN.
- Interconnexió de senyals entre la instància can_transmit i can_receive.

En principi, es podrien crear varies instàncies de l'entitat can_receive per augmentar el nombre de missatges que es poden llegir sense sobreescriure el missatge anterior o perdre els actuals.

La idea del procés implementat dins l'entitat user_logic és carregar les dades per l'enviament de la trama CAN provinents de l'aplicació només amb flancs ascendents de l'entrada tx_start_input. L'entrada tx_start_input s'envia des de l'aplicació al mòdul IP per transmetre una trama CAN, així que amb aquest sistema s'evita enviar la trama múltiples cops si no s'esborra el *flag* des de l'aplicació.

Aquest algoritme s'ha implementat seguint la lògica del diagrama de flux de la Figura 32. El procediment és carregar les dades quan tx_start_input canvia de 0 a 1 i deshabilitar que es pugui tornar a entrar a carregar les dades fins que no s'ha tornat a habilitar, tx_start_input torna a ser 0. Aquest diagrama de flux també inclou un sistema per restejar?? el sistema durant la configuració de la interfície CAN.

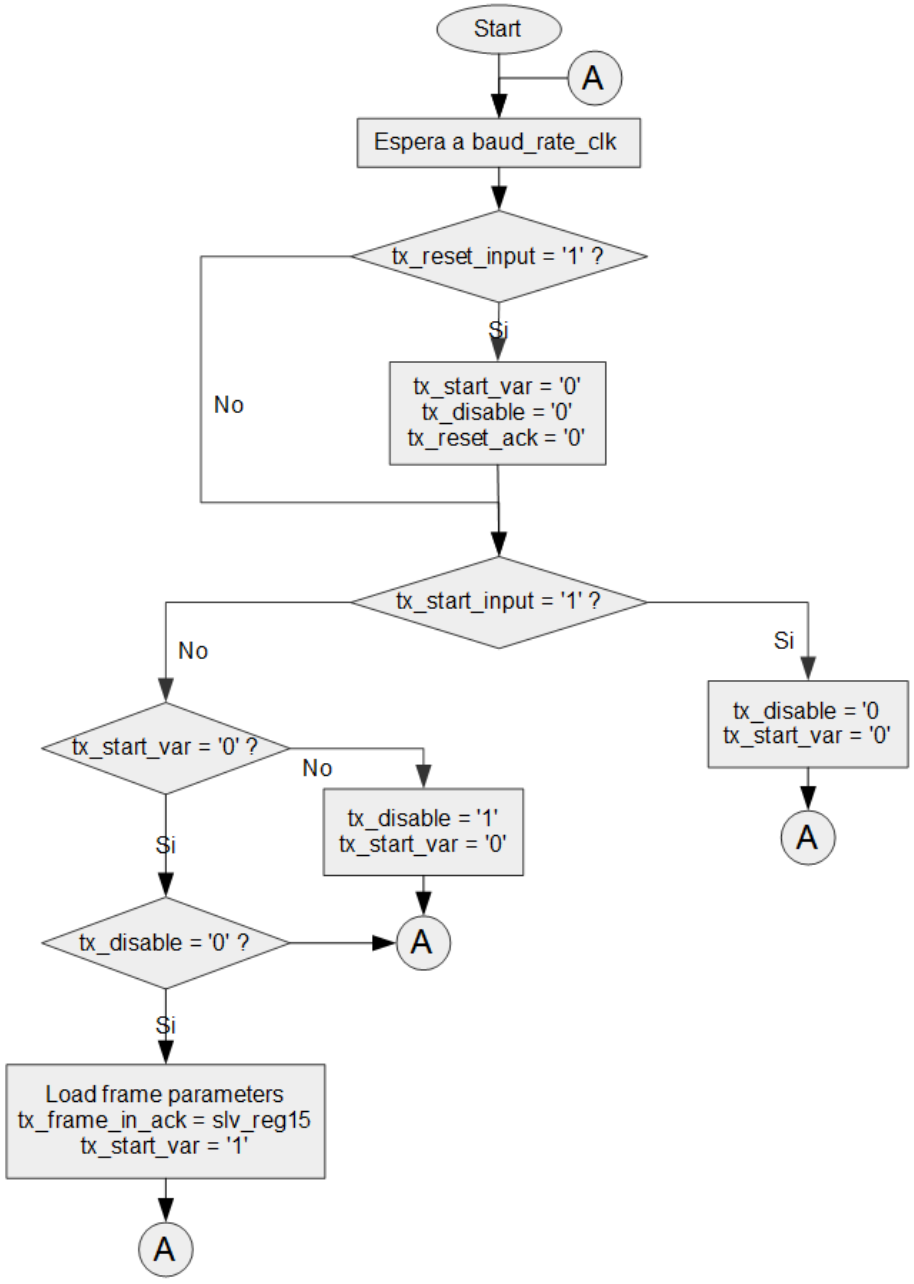


Figura 32. Diagrama de flux del procés de l'entitat user_logic

Per més informació sobre l'entitat user_logic, veure el codi amb els comentaris explicatius a l'annex 3.

4.4.2 Mòdul IP AXI_7SEG_DISPLAY

El mòdul IP AXI_7SEG_DISPLAY s'encarrega de controlar el display de 7 segments i dos dígits, DA03-11SURKWA. L'aplicació envia a través d'un registre al mòdul IP un valor del 0 al 99 i el mòdul genera les senyals necessàries per mostrar el número correctament al display. En el cas que el número passat sigui més gran que 99, és a dir, que necessiti més de dos dígits per ser representat, es mostrarà el valor EE.

D'igual manera que amb el mòdul IP AXI_CAN_CUSTOM (veure 4.4.1), s'ha creat el mòdul IP AXI_7SEG_DISPLAY. La diferència principal és que aquest mòdul IP és molt més simple i només cal 1 registre per intercanviar dades entre l'aplicació i el mòdul IP.

La Figura 33 mostra l'estructura d'aquest mòdul. Es pot observar que s'ha reaprofitat l'entitat DIVIDER.vhd per generar una senyal de rellotge més lenta que permeti mostra els dos dígits del display de 7 segments.

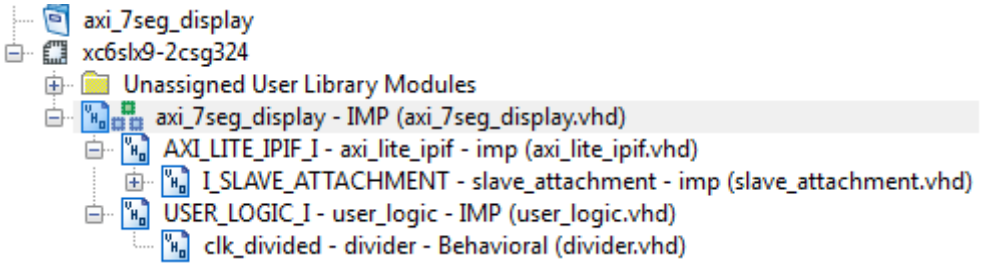


Figura 33. Estructura del mòdul IP AXI_7SEG_DISPLAY

A l'entitat USER_LOGIC_I s'ha afegit un codi en llenguatge VHDL que cada senyal d'un rellotge de 100 Hz, el dígit mostrat s'intercanvia i es converteix el valor d'aquest dígit al codi per representar el número en 7 segements.

4.4.2.1 Configuració addicional

Amb el codi desenvolupat i encara que sigui sintetitzable amb el Project Navigator, no és suficient perquè es pugui integrar dins del Microblaze.

Abans és necessari modificar els fitxers axi_can_custom_v2_1_0.mpd i axi_can_custom_v2_1_0.pao generats automàticament amb l'assistent de configuració.

El fitxer .mpd conté la descripció del perifèric i és on s'ha de definir els ports d'entrada i sortida del mòdul IP.

```
25  OPTION ARCH_SUPPORT_MAP = (others=DEVELOPMENT)

45  ## Ports
46  PORT CAN_rx = "", DIR = I
47  PORT CAN_tx = "", DIR = O
48  PORT baud_rate_rx = "", DIR = O
49  PORT baud_rate_tx = "", DIR = O
```

Figura 34. Extracte del fitxer axi_can_custom_v2_1_0.mpd

A la Figura 34 es mostren la línies afegides al fitxer axi_can_custom_v2_1_0.mpd. Apart de la connexió del mòdul amb els ports externs, es configura l'opció ARCH_SUPPORT_MAP perquè cada cop que es generi el *bitstream* del sistema hardware, es sintetitzi el mòdul axi_can_custom de nou. Així els canvis que es

realitzen durant el desenvolupament es reflecteixen en el nou *bitstream*.

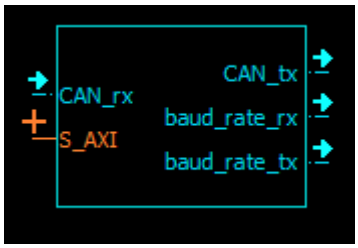


Figura 35. Diagrama del mòdul IP AXI_CAN_CUSTOM

A la Figura 35 es mostra el resultat de la configuració aplicada sobre el fitxer .mpd. És important perquè el canvi tingui efecte re-escanejar la llibreria de mòduls IP i en algun cas reiniciar el programa.

L'altre fitxer a modificar, axi_can_custom_v2_1_0.pao, defineix l'ordre d'anàlisis dels submòduls que formen el mòdul IP. En un mòdul IP simple no caldria modificar-lo però en aquest cas, és necessari afegir la referencia al mòdul VHDL afegits.

```
1  ##############################################################################
2  ## Filename:          C:/Xilinx/14.7/ISE_DS/edk_user_repository/MyProcessorIPLib/pcores/
3  ## Description:       Peripheral Analysis Order
4  ## Date:              Sun Jun 25 11:34:17 2017 (by Create and Import Peripheral Wizard)
5  ##############################################################################
6
7  lib proc_common_v3_00_a proc_common_pkg vhdl
8  lib proc_common_v3_00_a family_support vhdl
9  lib proc_common_v3_00_a pselect_f vhdl
10 lib proc_common_v3_00_a ipif_pkg vhdl
11 lib axi_can_custom_v1_11_a divider vhdl
12 lib axi_can_custom_v1_11_a delay vhdl
13 lib axi_can_custom_v1_11_a CAN_VHDL_PACKAGE vhdl
14 lib proc_common_v3_00_a counter_f vhdl
15 lib axi_lite_ipif_v1_01_a address_decoder vhdl
16 lib axi_can_custom_v1_11_a can_transmit vhdl
17 lib axi_can_custom_v1_11_a can_receive vhdl
18 lib axi_lite_ipif_v1_01_a slave_attachment vhdl
19 lib axi_can_custom_v1_11_a user_logic vhdl
20 lib axi_lite_ipif_v1_01_a axi_lite_ipif vhdl
21 lib axi_can_custom_v1_11_a axi_can_custom vhdl
22
```

Figura 36. Fitxer axi_can_custom_v2_1_0.pao

4.4.3 Mòdul IP AXI_7SEG_DISPLAY

El mòdul IP AXI_7SEG_DISPLAY s'encarrega de controlar el display de 7 segments i dos dígits, DA03-11SURKWA. L'aplicació envia a través d'un registre al mòdul IP un valor del 0 al 99 i el mòdul genera els senyals necessaris per mostrar el número correctament al display. En el cas que el número passat sigui més gran que 99, és a dir, que necessiti més de dos dígits per ser representat, es mostrarà el valor EE.

De igual manera que amb el mòdul IP AXI_CAN_CUSTOM (veure 4.4.1), s'ha creat el mòdul IP AXI_7SEG_DISPLAY. La diferència principal és que aquest mòdul IP és molt més simple i només cal 1 registre per intercanviar dades entre l'aplicació i el mòdul IP.

La Figura 33 mostra l'estructura d'aquest mòdul. Es pot observar que s'ha reaprofitat l'entitat DIVIDER.vhd per generar una senyal de rellotge més lenta que permeti mostrar els dos dígits del display de 7 segments.

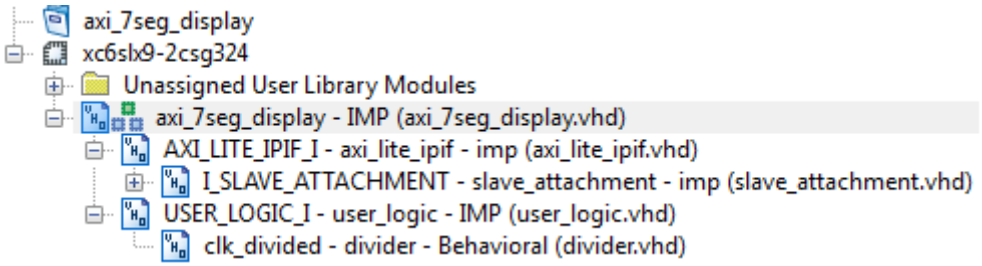


Figura 37. Estructura del mòdul IP AXI_7SEG_DISPLAY

A l'entitat USER_LOGIC_I s'ha afegit un codi en llenguatge VHDL que cada senyal d'un rellotge de 100 Hz, el dígit mostrat s'intercanvia i es converteix el valor d'aquest dígit al codi per representar el número en 7 segements. Aquest codi està disponible a la Figura 38.

```
159   digit: process (Bus2IP_Clk) is
160      variable dig_sel_var: std_logic := '0';
161      variable dig_val: std_logic_vector(3 downto 0);
162   begin
163      if(Bus2IP_Clk'event and Bus2IP_Clk='1')then
164         if(clk_slow = '1')then
165            if(dig_sel_var = '0')then
166               dig_val := slv_reg0(3 downto 0);
167               dig_sel <= '0';
168            else
169               dig_val := slv_reg0(7 downto 4);
170               dig_sel <= '1';
171            end if; -- digit_selected = '0'
172            dig_sel_var := not dig_sel_var;
173            case dig_val is
174               when "0000" => segments <= "1000000"; -- 0
175               when "0001" => segments <= "1111001"; -- 1
176               when "0010" => segments <= "0100100"; -- 2
177               when "0011" => segments <= "0110000"; -- 3
178               when "0100" => segments <= "0011001"; -- 4
179               when "0101" => segments <= "0010010"; -- 5
180               when "0110" => segments <= "0000010"; -- 6
181               when "0111" => segments <= "1111000"; -- 7
182               when "1000" => segments <= "0000000"; -- 8
183               when "1001" => segments <= "0010000"; -- 9
184               when "1010" => segments <= "0001000"; -- A
185               when "1011" => segments <= "0000011"; -- B
186               when "1100" => segments <= "1000110"; -- C
187               when "1101" => segments <= "0100001"; -- D
188               when "1110" => segments <= "0000110"; -- E
189               when others => segments <= "0001110"; -- F
190            end case;
191         end if; -- clk_slow = '1'
192      end if; -- Bus2IP_Clk'event and Bus2IP_Clk='1'
193   end process digit;
```

Figura 38. Procés dígit de l'entitat USER_LOGIC del mòdul IP AXI_7SEG_DISPLAY

Al realitzar la configuració dels fitxes .pao i .mpd s'obté el bloc lògic del mòdul IP AXI_7SEG_DISPLAY (veure Figura 39).

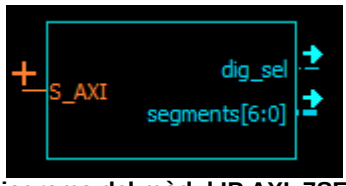


Figura 39. Diagrama del mòdul IP AXI_7SEG_DISPLAY

4.4.4 *Driver* axi_can_*driver*

La llibreria en codi C dissenyada per interferir de forma simple i fàcil des de l'aplicació s'anomena aix_can_*driver* i implementa les següents funcions:

Taula 2. Descripció de la funció can_configuration

Definició	int can_configuration(int can_interface)
Funció	S'encarrega de configurar i inicialitzar el mòdul CAN. També verifica que el mòdul s'hagi iniciat correctament
Paràmetre – can_interface	Identificador de la interfície CAN
Retorn	XST_SUCCESFUL si el resultat del test és correcte, sinó, identificador de l'error

Taula 3. Descripció de la funció can_send

Definició	int can_send(int can_interface, u16 id, u8 dlc, u32 *data, u32 ack)
Funció	Envia un missatge CAN amb l'identificador i les dades proporcionades
Paràmetre – can_interface	Identificador de la interfície CAN
Paràmetre - id	Identificador del missatge CAN
Paràmetre – dlc	Número de bytes de les dades proporcionades
Paràmetre – data	Dades per enviar dins el missatge CAN
Paràmetre – ack	Permet comprovar que el missatge enviat és correcte, comparació entre el ack passat i el generat pel mòdul IP
Retorn	XST_SUCCESFUL si el resultat del test és correcte, sinó, identificador de l'error

Taula 4. Descripció de la funció can_receive

Definició	int can_receive(int can_interface, u16 mbox, u16 *id, u8 *dlc, u8 *data)
Funció	Comprovar si hi ha dades a llegir i si n'hi ha, llegir-les.
Paràmetre – can_interface	Identificador de la interfície CAN
Paràmetre - id	Identificador del missatge CAN llegit
Paràmetre – dlc	Número de bytes de les dades llegides
Paràmetre – data	Dades del missatge CAN llegides
Retorn	CAN_RX_EMPTY si no hi ha dades a llegir o XST_SUCCESFUL si es llegeixen dades correctament. En cas contrari, identificador de l'error.

4.4.5 *Driver* disp_7seg_display

El *driver* disp_7seg_display s'encarrega de mostrar un valor del 0 al 99 pel display de 7 segments i dos dígits. Aquest *driver* conté una funció descrita, Taula 5.

Taula 5. Descripció de la funció set_display_value

Definició	void set_display_value(u8 value)
Funció	Interferir amb el mòdul IP que controla el display per configurar el valor a mostrar
Paràmetre – value	Valor a mostrar pel display
Retorn	La funció no retorna cap valor

4.4.6 *Driver* adc_*driver*

La lectura del ADC es fa a través de l'integrat MCP3002 que es comunica amb la FPGA per SPI. Per tant, el *driver* dissenyat per llegir els valors del ADC s'encarrega de comunicar-se amb aquest correctament mitjançant les funcions del mòdul IP SPI de Xilinx.

A la Taula 6 i Taula 7 es mostren les dues funcions que conté aquest *driver*.

Taula 6. Descripció de la funció ADC_ini

Definició	int ADC_ini(XSpi* xspi)
Funció	Configurar i inicialitzar la interfície SPI que s'utilitzarà per comunicar-se amb el ADC. L'integrat MCP3002 no requereix una configuració inicial. La configuració es realitza cada cop que es vol llegir un valor
Paràmetre – xspi	Conté la referència i configuració de la interfície SPI
Retorn	XST_SUCCESFUL si la interfície SPI s'ha configurat correctament

Taula 7. Descripció de la funció ADC_read

Definició	int ADC_read(XSpi* xspi, u8 *meas)
Funció	Envia un missatge SPI per obtenir una lectura del ADC
Paràmetre – xspi	Conté la referència i configuració de la interfície SPI
Paràmetre – meas	Contindrà el valor llegit del ADC
Retorn	XST_SUCCESFUL si s'ha pogut llegir el valor del ADC correctament

5. RESULTATS EXPERIMENTALS

Seguint el model en V, cada requisit ha de tenir un test associat que permeti determinar si està implementat correctament. Com es mostrarà a continuació, un mateix test pot servir per comprovar diversos requisits. La Taula 8 mostra aquestes relacions, així com el resultat d'aquests tests:

Taula 8. Resultats de la validació dels requisits hardware del sistema (1 de 3)

Requisit	Test	Comentaris	Resultat
[REQ_HW_1]	No cal cap test	La plataforma Avnet LX9_2_CAN està disponible al mercat i integra una FPGA de Xilinx.	Correcte
[REQ_HW_2]	Revisar els informes del disseny generats pel XPS. En principi si el *bitstream* s'ha generat és que la FPGA pot suportar el sistema	S'ha pogut carregar el disseny sobre la FPGA amb totes les funcionalitats encara que ha sigut necessari millorar i optimitzar molt el codi perquè capigués i complís amb les limitacions de temps. Possiblement amb un sistema més modern i potent hagués funcionat sense més modificacions.	Correcte
[REQ_HW_3]	Configurar l'aplicació perquè transmeti missatges CAN a 1 Mbps cada 1 ms. Comprovar amb l'oscil·loscopi que el missatge s'envia correctament	Detalls disponibles a 5.2	Correcte
[REQ_HW_4]	Configurar l'aplicació perquè transmeti missatges CAN a 1 Mbps cada 1 ms. Comprovar amb l'oscil·loscopi que el missatge s'envia correctament	Detalls disponibles a 5.2	Correcte
[REQ_HW_5]	Configurar l'aplicació perquè transmeti missatges CAN a 1 Mbps cada 1 ms. Comprovar amb l'oscil·loscopi que el missatge s'envia correctament	Detalls disponibles a 5.2	Correcte
[REQ_HW_6]	No cal cap test	La Avnet LX9 Microboard és un sistema comercial i molt venut, així que es pressuposa que el programador integrat a través de l'USB funciona	Correcte

Taula 9. Resultats de la validació dels requisits hardware del sistema (2 de 3)

Requisit	Test	Comentaris	Resultat
[REQ_HW_7]	Revisar que el nombre d'entrades i sortides de la Avnet LX9 Microboard és suficient per implementar totes les funcions	El nombre d'entrades-sortides del sistema és suficient per l'aplicació, ja que quedaven alguns lliures i s'han aprofitat per afegir un bus CAN addicional	Correcte
[REQ_HW_8]	No cal cap test	Sistema comercial i molt venut, es pressuposa que la comunicació RS-232 a través de l'USB funciona.	Correcte
[REQ_HW_9]	Comprovar que el voltatge del bus CAN està dins els límits establers per la ISO 11898.	Detalls disponibles a l'apartat 5.1	Correcte
[REQ_HW_10]	Provar que la resistència de terminació CAN es pot soldar a la LX9_2_CAN PCB fàcilment	És relativament fàcil de soldar. Com a punt de millora es podria separar més l'emplaçament de les resistències dels punts de prova perquè, si aquests estan muntats, és més complicat de soldar les resistències.	Correcte
[REQ_HW_11]	Provar continuïtat entre els punts de prova i la senyal connectada. Avaluar si les sondes de l'oscil·loscopi encaixen correctament.	A l'apartat 5.2 es pot veure com s'han connectat dues sondes d'oscil·loscopi a la LX9_2_CAN PCB sense cap problema.	Correcte
[REQ_HW_12]	Provar que la LX9_2_CAN PCB encaixa correctament amb el header de la Avnet LX9 Microboard i es pot connectar i desconnectar fàcilment.	La placa encaixa sense problemes i es pot muntar i desmuntar ràpidament.	Correcte
[REQ_HW_13]	Provar continuïtat entre els pins del DSUB-9 i la senyal a la que ha de estar connectada	Hi ha continuïtat entre les senyals CAN i el pin corresponent del connector DSUB-9	Correcte
[REQ_HW_14]	Mesurar la corrent consumida màxima per la LX9_2_CAN PCB	Detalls disponibles a l'apartat 5.3	Correcte
[REQ_HW_15]	Comprovar si el hardware segueix funcionant després d'aplicar descàrregues electrostàtiques a les entrades	S'ha aplicat aquest criteri en el disseny hardware, però no es disposa dels equips necessaris per poder verificar la immunitat a descàrregues electrostàtiques. Es considera correcte perquè fins el moment no s'ha donat cap problema que demostri el contrari.	Correcte

Taula 10. Resultats de la validació dels requisits hardware del sistema (3 de 3)

Requisit	Test	Comentaris	Resultat
[REQ_HW_16]	Mesurar les radiacions electromagnètiques emeses. Radiar ones de diferents freqüències sobre el hardware i comprova que aquest segueix funcionant correctament.	S'ha aplicat aquest criteri en el disseny hardware, però no es disposa dels equips necessaris per poder verificar les radiacions electromagnètiques i la immunitat a les emissions radiades. Es considera correcte perquè fins el moment no s'ha donat cap problema que demostri el contrari.	Correcte
[REQ_HW_17]	Col·locar un valor de tensió coneguda a l'entrada i comprovar els resultats mesurats per l'ADC.	El valor s'ha comprovat amb l'aplicació software per poder escriure/llegir per SPI. El valor llegit respecte el simulat és mou en un rang d'acord amb les especificacions de l'integrat MCP3002.	Correcte
[REQ_HW_18]	Provar cadascun dels segments de tots dos dígits manualment	Mateix muntatge que a l'apartat 5.3, però combinant les connexions per provar tots els segments individual i conjuntament. Tots els dígits estan operatius.	Correcte

Taula 11. Resultats de la validació dels requisits software del sistema (1 de 5)

Requisit	Test	Comentaris	Resultat
[REQ_SW_1]	No s'ha dissenyat cap test específic	El mòdul IP s'ha dissenyat de tal manera que es poden afegir i connectar fàcilment més o menys instàncies de l'entitat can_receive. Per millorar el rendiment del sistema, si es disposa de recursos suficients, només s'han d' afegir més i connectar-les correctament.	Correcte
[REQ_SW_2]	No s'ha dissenyat cap test específic	El codi s'ha realitzat de la forma més neta possible i afegint comentaris detallats.	Correcte

Taula 12. Resultats de la validació dels requisits software del sistema (2 de 5)

Requisit	Test	Comentaris	Resultat

[REQ_SW_3]	No s'ha dissenyat cap test específic	Per operar el *driver* CAN dissenyat només cal utlitzar tres funcions molt sencilles. Per tant, es considera que crear noves aplicacions és ràpid i fàcil.	Correcte
[REQ_SW_4]	No s'ha dissenyat cap test específic	El *driver* informa a l'aplicació quan apareix un error i dóna informació addicional de quin és aquest error. Es considera que satisfà les necessitats de diagnòstics del sistema.	Correcte
[REQ_SW_5]	Configurar l'aplicació perquè transmeti missatges CAN a 1 Mbps cada 1 ms.	Detalls disponibles a 5.2	Correcte
[REQ_SW_6]	Externalitzar la senyal de clk_delay que utilitza l'entitat can_receive i observar si es sincronitza quan toca.	Detalls disponibles 5.6	Correcte
[REQ_SW_7]	Externalitzar la senyal de clk_delay que utilitza l'entitat can_receive i observar si es sincronitza quan toca.	Detalls disponibles 5.6	Correcte
[REQ_SW_8]	Externalitzar la senyal de clk_delay que utilitza l'entitat can_receive i observar si es sincronitza quan toca.	Detalls disponibles 5.6	Correcte
[REQ_SW_9]	Externalitzar la senyal de clk_delay que utilitza l'entitat can_receive i observar si es sincronitza quan toca.	Detalls disponibles 5.6	Correcte
[REQ_SW_10]	Enviar trames de dades CAN al CAN-shield d'Arduino, comprovar que l'Arduino rep les dades com s'espera.	Detalls disponibles a l'apartat 5.4	Correcte
[REQ_SW_11]	Enviar trames de dades des del CAN-Shield d'Arduino al sistema. Comprovar que es rep el que s'espera i no hi ha cap trama d'error al bus.	Detalls disponibles a l'apartat 5.5	Correcte

Taula 13. Resultats de la validació dels requisits software del sistema (3 de 5)

Requisit	Test	Comentaris	Resultat

[REQ_SW_12]	Enviar trames de dades CAN al CAN-shield d'Arduino, comprovar que l'Arduino rep el mateix identificador que s'envia.	Detalls disponibles a l'apartat 5.4	Correcte
[REQ_SW_13]	Enviar trames de dades CAN al CAN-shield d'Arduino. Si l'Arduino interpreta les dades correctament es que s'afegeixen els bit de *stuff* correctament.	Detalls disponibles a l'apartat 5.4	Correcte
[REQ_SW_14]	Enviar trames de dades des del CAN-Shield d'Arduino al sistema. Si les dades llegides coincideixen amb les enviades per l'Arduino s'està gestionant els bit de *stuff* correctament.	Detalls disponibles a l'apartat 5.5	Correcte
[REQ_SW_15]	Enviar trames de dades des del CAN-Shield d'Arduino al sistema. Si les dades llegides coincideixen amb les enviades per l'Arduino s'està gestionant els bit de *stuff* correctament	Detalls disponibles a l'apartat 5.5	Correcte
[REQ_SW_16]	Modificar el mòdul IP per simular una situació on l'entitat can_transmit perd l'arbitratge. S'espera que es deixi d'enviar la trama de dades.	Detalls disponibles a l'apartat 5.10	Correcte
[REQ_SW_17]	Modificar el mòdul IP per simular una situació on l'entitat can_transmit perd l'arbitratge. S'espera que un cop s'ha acabat d'enviar la trama que ha guanyat l'arbitratge, es retransmeti la trama de dades aturada.	Detalls disponibles a l'apartat 5.10	Correcte
[REQ_SW_18]	Enviar trames de dades CAN al CAN-shield d'Arduino. Si el Arduino interpreta les dades correctament es que el CRC s'ha calculat i afegit correctament a la trama.	Detalls disponibles a l'apartat 5.4	Correcte

Taula 14. Resultats de la validació dels requisits software del sistema (4 de 5)

Requisit	Test	Comentaris	Resultat

[REQ_SW_19]	Enviar trames de dades CAN al CAN-shield d'Arduino. Si l'Arduino interpreta les dades correctament és que el CRC s'ha calculat i afegit correctament a la trama.	Detalls disponibles a l'apartat 5.4	Correcte
[REQ_SW_20]	Enviar trames cíclicament cap al CAN-Shield i desconnectar l'alimentació de l'Arduino. Al ser l'únic node del bus, la Spartan-6 no rebrà el ACK i es podrà observar com reacciona.	Detalls disponibles a l'apartat 5.7	Correcte
[REQ_SW_21]	Modificar el *driver* en C perquè no faci un *reset* del *flag* de transmissió i comprovar que només és possible enviar un missatge. Comprovar la indicació d'errors.	Detalls disponibles a l'apartat 5.8	Correcte
[REQ_SW_22]	No s'ha definit cap test	L'aplicació disposa d'una funció que, amb pocs paràmetres i sense un coneixement del funcionament intern, permet configurar i iniciar la interfície CAN.	Correcte
[REQ_SW_23]	No s'ha definit cap test	L'aplicació disposa d'una funció per configurar i iniciar la interfície CAN que, en cas de no obtenir resposta del mòdul a alguna configuració, indicaria quin ha estat el problema.	Correcte
[REQ_SW_24]	Modificar el *driver* en C perquè no faci un *reset* del *flag* de transmissió i comprovar la diferència entre quan s'ha pogut escriure un missatge i quan no.	Detalls disponibles a l'apartat 5.8	Correcte
[REQ_SW_25]	Modificar el mòdul IP CAN perquè no enviï el *flag* de finalització de l'enviament. Comprovar que l'aplicació reconeix l'error.	Detalls disponibles a l'apartat 5.9	Correcte

Taula 15. Resultats de la validació dels requisits software del sistema (5 de 5)

Requisit	**Test**	**Comentaris**	**Resultat**

[REQ_SW_26]	Enviar trames de dades des del CAN-Shield d'Arduino al sistema. Si l'aplicació s'adona de que el mòdul IP ha rebut la trama, la indicació de trama CAN rebuda funciona.	Detalls disponibles a l'apartat 5.7	Correcte
[REQ_SW_27]	Enviar trames de dades des del CAN-Shield d'Arduino al sistema. Si l'aplicació s'adona de que el mòdul IP ha rebut la trama i el resultat és ok, la indicació de trama CAN rebuda funciona.	Detalls disponibles a l'apartat 5.7	Correcte
[REQ_SW_28]	Enviar trames de dades des del CAN-Shield d'Arduino al sistema. Comprovar que s'envia el ACK i no es reben trames d'error.	Detalls disponibles a l'apartat 5.5	Correcte
[REQ_SW_29]	No s'ha definit cap test	L'aplicació disposa d'una funció que, amb pocs paràmetres i sense un coneixement del funcionament intern, permet enviar trames de dades CAN fàcilment.	Correcte
[REQ_SW_30]	No s'ha definit cap test	L'aplicació disposa d'una funció que, amb pocs paràmetres i sense un coneixement del funcionament intern, permet rebre trames de dades CAN fàcilment.	Correcte

5.1 Voltatge de les senyals de transmissió del bus CAN

L'objectiu d'aquest test és comprovar que els nivells de tensió que posa el CAN *transceiver* al bus CAN estan dins dels límits marcats per la normativa ISO 11898. Amb aquest test es pretén validar el requisit [REQ_HW_9].

Per fer-ho, s'ha capturat amb l'oscil·loscopi la transmissió d'un missatge pel bus CAN i mesurat els voltatges màxim i mínim de cada senyal. La Figura 40 mostra el resultat obtingut i la Taula 16 recull les diferents mesures de voltatges extretes de la captura de l'oscil·loscopi.

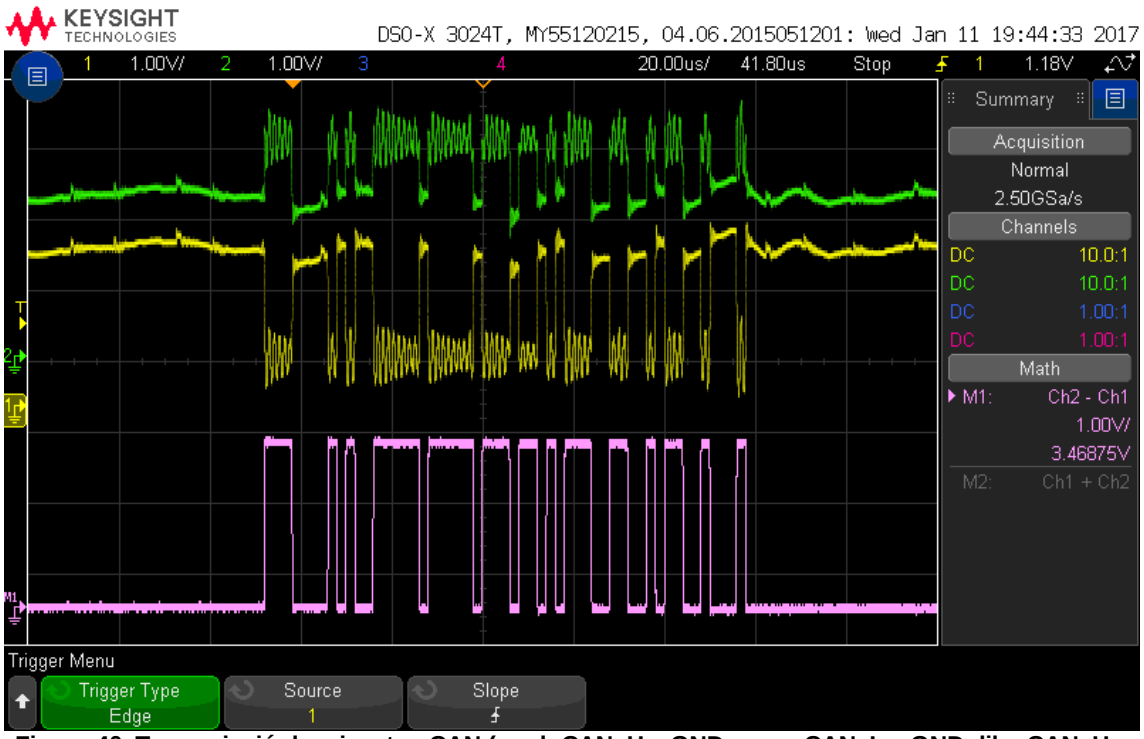


Figura 40. Transmissió de missatge CAN (verd: CAN_H – GND, groc: CAN_L – GND; lila: CAN_H – CAN_L, calculat amb la funció matemàtica)

Taula 16. Comparació entre el voltatge de les senyals CAN, el valor nominal i el límit

Senyal	Mesura	Valor nominal	Rang vàlid*
CAN_H – GND (màx)	3.4 V	3.5 V	-
CAN_H – GND (mín)	2.2 V	2.5 V	-
CAN_L – GND (màx)	2.3 V	2.5 V	-
CAN_L – GND (mín)	1.1 V	1.5 V	-
CAN_H – CAN_L (màx)	2.5 V	2.25V	1.5 – 3.0 V
CAN_H – CAN_L (mín)	0 V	0 V	-0.5 – 0.05 V

* Valor extret de la ISO 11898.

Amb les dades de la Taula 16, es pot demostrar que la capa física dissenyada per la comunicació CAN es comporta dins els límits de la normativa i que, per tant, compleix amb el requisit [REQ_HW_9].

5.2 Test de velocitat de la comunicació CAN

L'objectiu d'aquest test és comprovar que es pot transmetre a velocitats de fins 1 Mbps cada 1 ms. Amb aquest test es pretén comprovar els requisits: [REQ_HW_3], [REQ_HW_4], [REQ_HW_5] i [REQ_SW_5].

Per realitzar el test s'ha configurat l'aplicació perquè transmeti trames de dades CAN, a una velocitat de 1 Mbps cada 1 ms. La trama de dades CAN programada conté el màxim de bytes de dades que es poden enviar i valors de bit que obliguen a afegir més bits de *stuff*. Així es simularà la situació més desfavorable per complir els requisits.

A la Figura 41 es mostra el resultat obtingut amb l'oscil·loscopi a l'observar la trama amb detall i, a la Figura 42, les trames enviant-se cada 1 ms. Com es pot observar en aquestes captures, la transmissió de dades funciona correctament. Cal destacar que el

nivell de soroll de les línies de transmissió ha augmentat considerablement al pujar la velocitat de transmissió. Tot i amb això, el bus diferencial és capaç de cancel·lar aquest soroll i mantenir els valors de voltatge dins els límits establerts per la normativa ISO 11898.

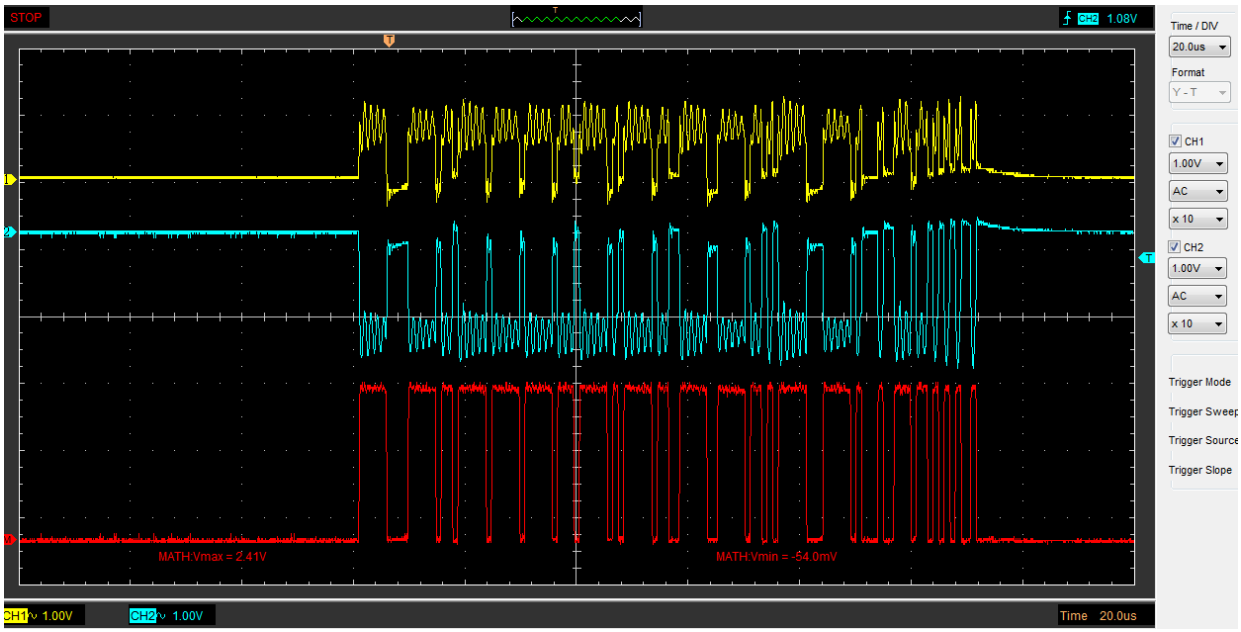


Figura 41. Trama de dades a 1Mbps enviada per la LX9_2_CAN PCB (groc: CAN_H – GND; blau: CAN_L – GND; vermell: CAN_H – CAN_L)

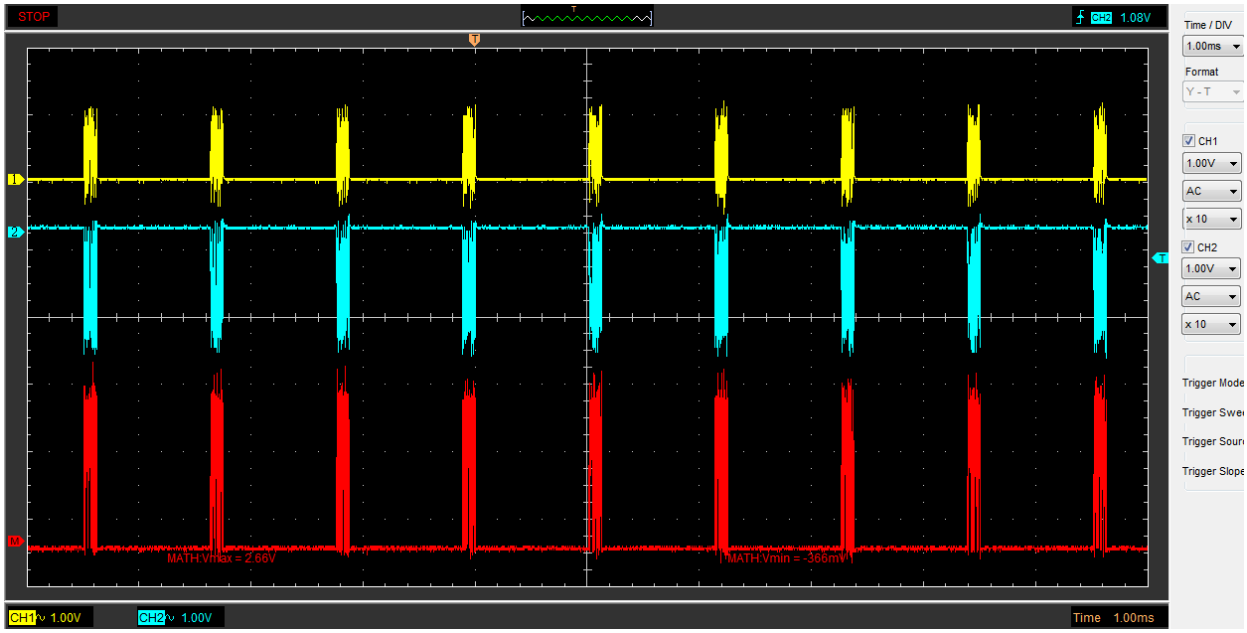


Figura 42. Trames dades enviades cada 1 ms a 1 Mbps per la LX9_2_CAN PCB (groc: CAN_H – GND; blau: CAN_L – GND; vermell: CAN_H – CAN_L)

La Figura 43 mostra el muntatge realitzat per fer els test de velocitat i obtenir els resultats anteriors.

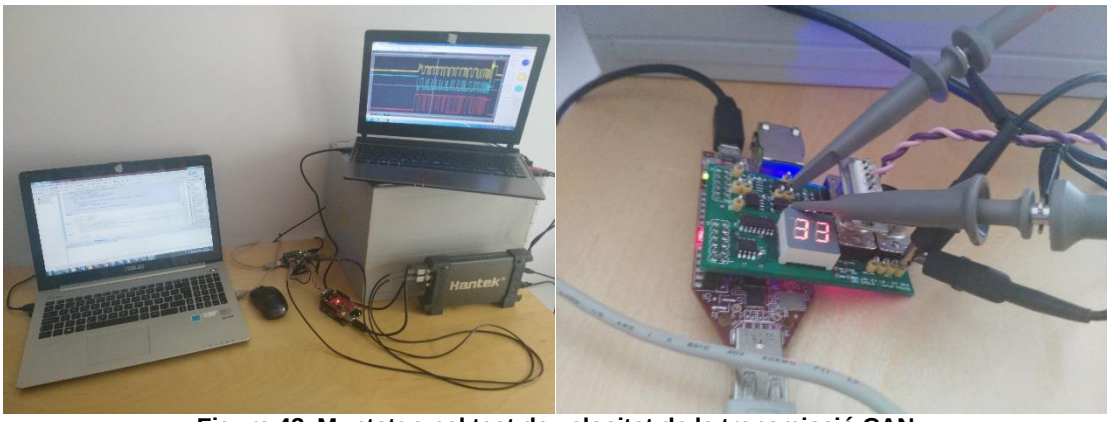


Figura 43. Muntatge pel test de velocitat de la transmissió CAN

5.3 Consum LX9_2_CAN PCB

Per assegurar que es compleix el requisit [REQ_HW_14], s'ha alimentat la LX9_2_CAN PCB des d'una font d'alimentació regulable i mesurat la corrent consumida per la PCB. Per assegurar que es compleix el requisit de potència consumida inferior a 100 mA (veure apartat 4.3.1) és necessari que la PCB estigui consumint el màxim possible i mesurar aquest consum.

La forma d'aconseguir el consum més elevat possible és tenint tots els integrats alimentats i un dígit del display de 7 segments amb tots els seus segments activats (dígit número 8). Per la manera en què ha estat implementat, només és possible tenir un dels dos dígits activat i s'aconsegueix curtcircuitant tots els càtodes i el pin de selecció de dígit a GND.

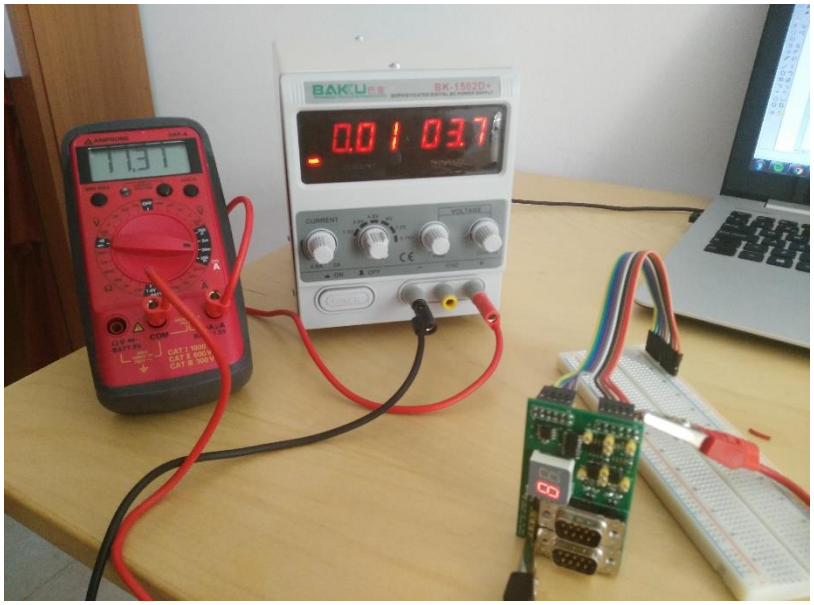


Figura 44. Muntatge per la verificació del consum màxim de la LX9_2_CAN PCB

La Figura 44 mostra el muntatge realitzat per verificar el consum màxim de la LX9_2_CAN PCB. La corrent consumida es mesura amb un multímetre per obtenir més precisió que amb la font d'alimentació. Es pot comprovar com el consum és de 11,3 mA, molt per sota del límit de 100 mA. Per tant, el sistema no tindrà problemes d'alimentació deguts a un consum excessiu.

5.4 Test de transmissió de trames de dades CAN

L'objectiu d'aquest test és comprovar que es transmeten trames de dades des de l'aplicació correctament. Amb aquest test es pretén validar els requisits: [REQ_SW_10], [REQ_SW_12], [REQ_SW_13], [REQ_SW_18] i [REQ_SW_19] .

El procediment d'aquest test és configurar l'aplicació perquè enviï tres trames de dades diferents seguides cada segon. El contingut d'aquestes trames de dades és:

- **Identificador** = 0x700 = 1792; **DLC** = 8; **Data** = [0,1,2,3,4,5,6,7]
- **Identificador** = 0x701 = 1793; **DLC** = 5; **Data** = [0,1,2,3,4]
- **Identificador** = 0x702 = 1794; **DLC** = 3; **Data** = [0,1,2]

La Figura 45 mostra la configuració de l'aplicació i trames CAN, les dades de *debug* que envia la Spartan-6 pel port sèrie i el resultat de llegir les trames de dades des de l'Arduino amb el CAN-Shield.

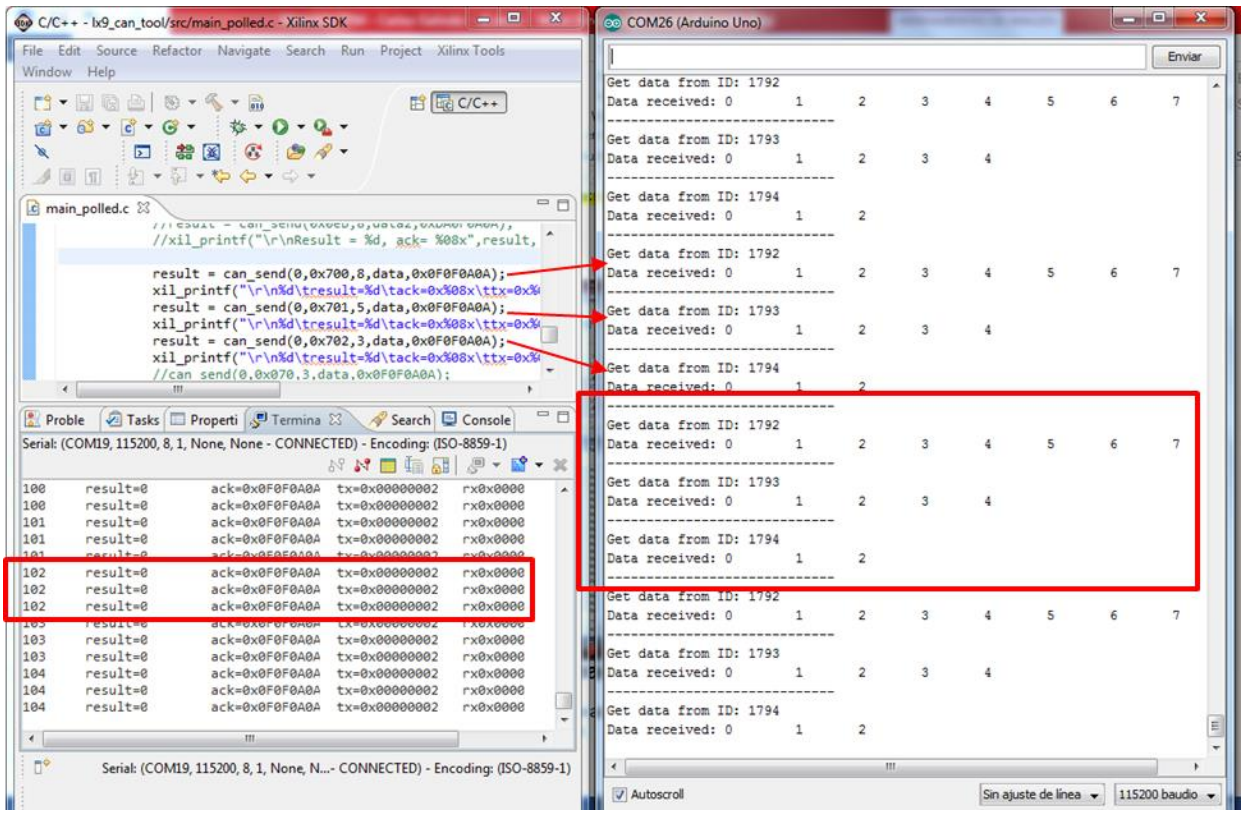


Figura 45. Resultats del test de transmissió de dades CAN

Els resultats demostren que el sistema dissenyat és capaç d'enviar les dades sol·licitades dins de trames de dades CAN i que estan correctament codificades. Cal destacar que sense complir amb els requisits esmentats al principi de l'apartat, un altre node CAN no seria capaç de descodificar correctament les dades.

A més a més, a la Figura 46 es pot comprovar que l'Arduino també dóna el reconeixement de lectura per cadascuna de les trames de dades. Com el DLC dels missatges és diferent, es pot veure com les trames de dades tenen una duració, més o menys llarga, en funció del nombre de bytes enviats per trama.

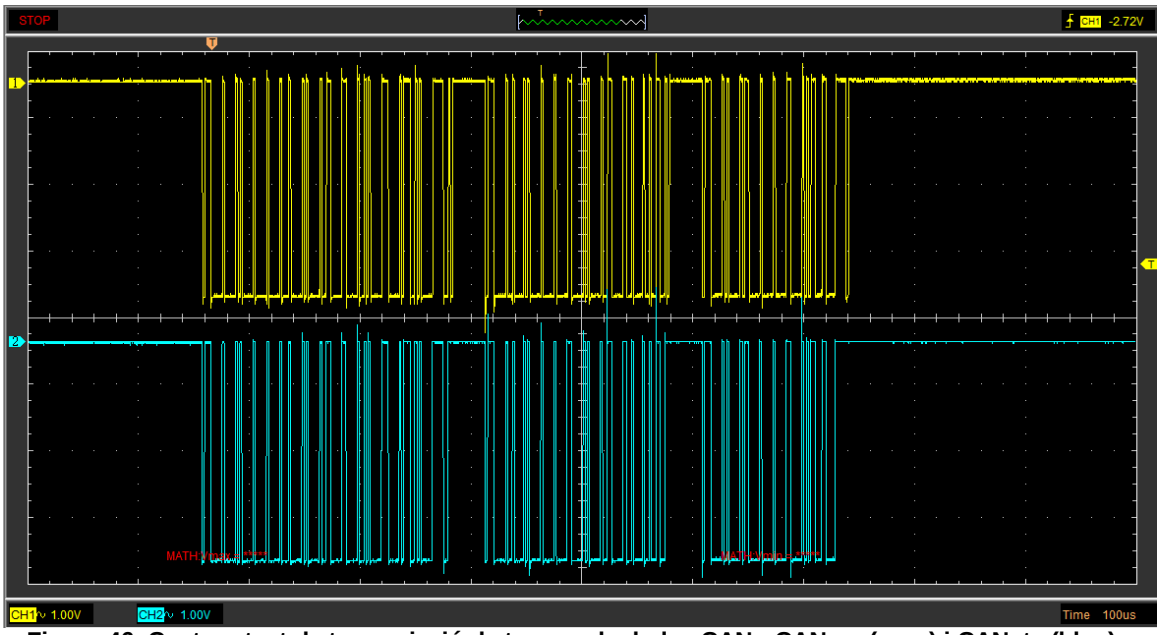


Figura 46. Captura test de transmissió de trames de dades CAN - CAN_rx (groc) i CAN_tx (blau)

5.5 Test de recepció de trames dades CAN

L'objectiu d'aquest test és comprovar que es llegeixen trames de dades des de l'aplicació correctament. Amb aquest test es pretén validar els requisits: [REQ_SW_11], [REQ_SW_14], [REQ_SW_15] i [REQ_SW_28].

El procediment d'aquest test és configurar l'aplicació perquè enviï tres trames de dades diferents seguides cada segon. El contingut d'aquestes trames de dades és:

- **Identificador** = 0x1FF = 511; **DLC** = 1; **Data** = [0]
- **Identificador** = 0x2FF = 767; **DLC** = 6; **Data** = [0,1,2,3,4,5]
- **Identificador** = 0x3FF = 1023; **DLC** = 3; **Data** = [0,1,2,3,4,5,6,7]

La Figura 47 mostra la configuració de l'Arduino per enviar les trames de dades anteriors, el codi de l'aplicació per llegir les dades CAN i enviar-les pel port sèrie, i la sortida del port sèrie.

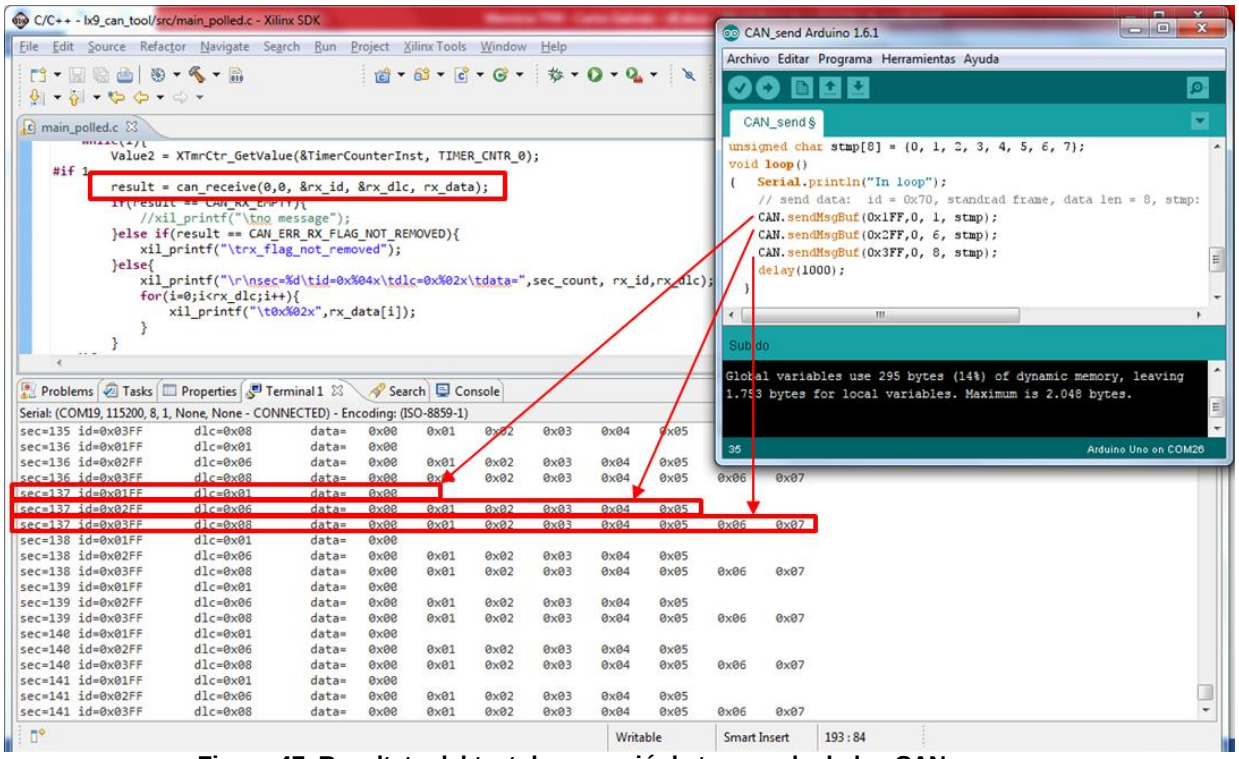


Figura 47. Resultats del test de recepció de trames de dades CAN

Els resultats demostren que el sistema dissenyat és capaç de llegir les trames de dades enviades per l'Arduino i que es descodifiquen correctament. Cal destacar que sense complir amb els requisits esmentats al principi de l'apartat, la Spartan-6 no seria capaç de descodificar les dades correctament.

A més a més, a la Figura 46, es pot comprovar com la Spartan-6 envia el reconeixement, ACK, a les tres trames. Al no haver-hi trames d'error ni cap altre node al bus, es comprova que el ACK s'envia correctament. Com el DLC dels missatges és diferent, es pot veure com les trames de dades tenen una duració més o menys llarga en funció del nombre de bytes enviats per trama.

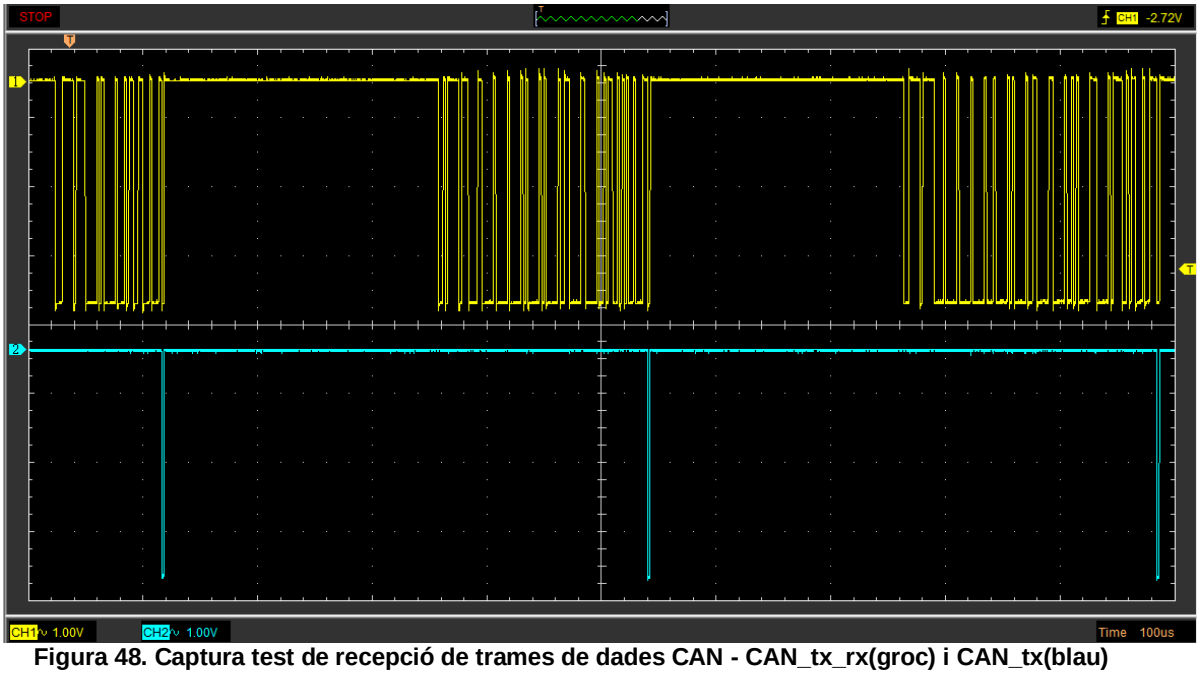


Figura 48. Captura test de recepció de trames de dades CAN - CAN_tx_rx(groc) i CAN_tx(blau)

5.6 Test de sincronització de la recepció de trames CAN

L'objectiu d'aquest test és assegurar que la sincronització i re-sincronització de la senyal de rellotge que gestiona la recepció de trames CAN es realitza correctament. Amb aquest test es pretén validar els requisits: [REQ_SW_6], [REQ_SW_7], [REQ_SW_8] i [REQ_SW_9].

La recepció de bits CAN es gestiona mitjançant un rellotge que es sincronitza quan hi ha un canvi de nivell al bus. El mostreig del valor del bit està desfasat respecte al rellotge per tal d'assegurar que el nivell del bit és estable quan es mostreja. Per comprovar que aquest mecanisme funciona correctament, s'ha utilitzat una sortida de la FPGA que no s'estava utilitzant per portar cap al exterior la senyal de rellotge desfasada. Monitoritzant aquesta senyal i la de recepció, es podrà verificar la sincronització del rellotge desfasat.

La Figura 49 mostra els resultats d'aquests test. La senyal groga representa la recepció CAN que genera el *transceiver* CAN al rebre els missatges del CAN-Shield controlat per l'Arduino. La senyal blava representa la senyal del rellotge desfasat. Per ajudar a comprovar el funcionament, s'han sobreposat les dues senyals. Observant el comportament d'aquestes senyals es pot veure com la senyal del rellotge sempre arriba una mica més tard que el canvi de nivell. A més a més, en aquesta captura es mostra com el bus passa de repòs (nivell alt) a rebre trames CAN. Si s'observa durant una estona es pot apreciar com, encara que les trames de dades CAN s'envien de forma aleatòria, sempre hi ha la mateixa distància entre el canvi i la senyal del rellotge desfasat.

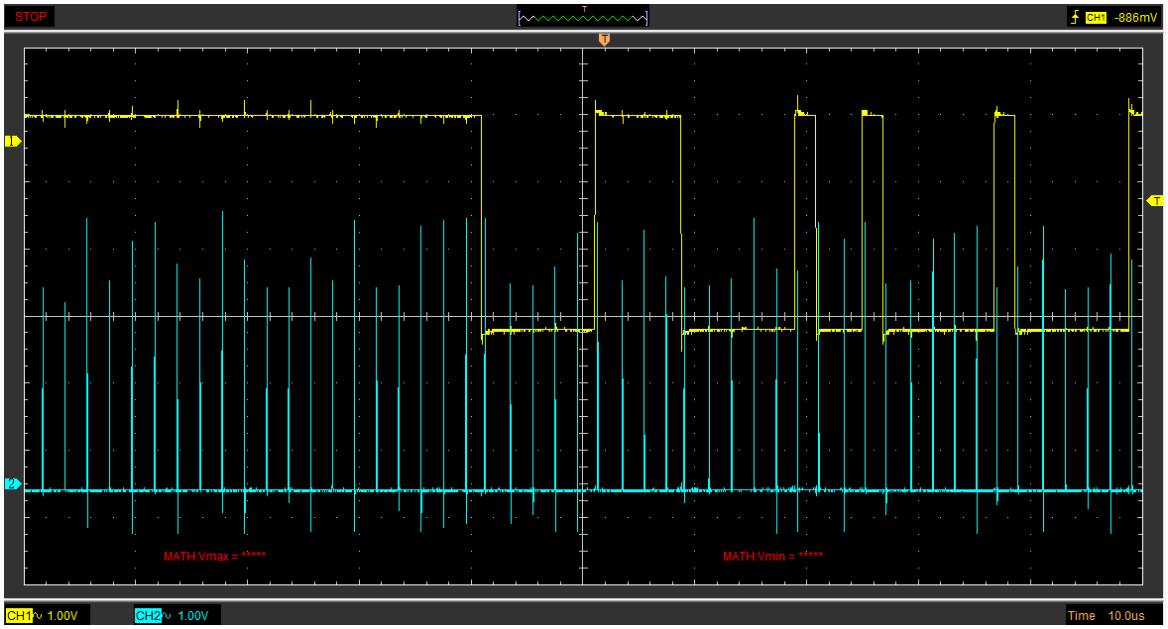


Figura 49. Captura dels senyals CAN_rx (groc) i baud_rate_clk_sync(blau) a l'inici de l'enviament

La Figura 50 mostra que passa al finalitzar la transmissió. És interessant observar aquest cas perquè amb l'últim bit de ACK no es re-sincronitza el rellotge. El motiu és evitar que es re-sincronitzi més d'un cop en el mateix temps de bit.

El problema és que el nivell dominant del ACK pot ser generat per diferents nodes simultàniament però probablement amb un desfasament suficient perquè la senyal es torni inestable i una re-sincronització pugui provocar una lectura incorrecta.

Observant la captura, s'evidencia que al no sincronitzar-se amb aquest últim bit dominant, el temps que hi ha entre el canvi de nivell i el mostreig del bus és una mica més gran que amb l'anterior bit. Per tant, es pot deduir que sense aquesta re-sincronització, en algun punt de la seqüència es podria anar dessincronitzant completament amb el transmissor. D'aquesta manera, es pot verificar que la re-sincronització fa la seva feina correctament.

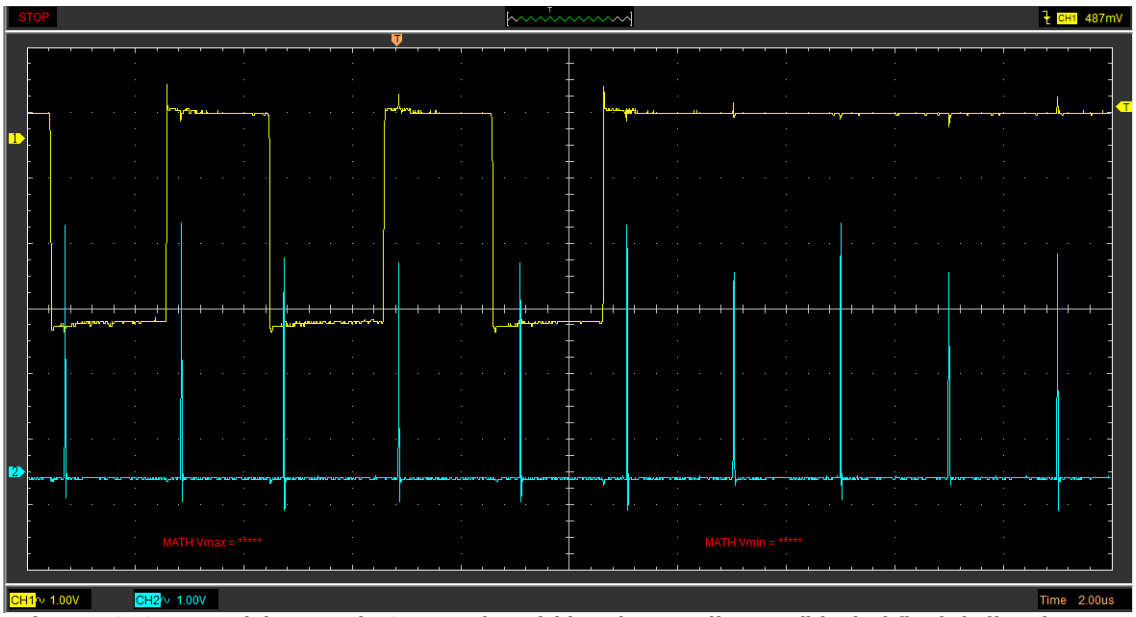


Figura 50. Captura dels senyals CAN_rx (groc) i baud_rate_clk_sync(blau) al final de l'enviament

5.7 Test d'enviament del reconeixement de lectura

L'objectiu d'aquest test és comprovar que es verifica el reconeixement de recepció de la trama CAN. Amb aquest test es pretén validar els requisits [REQ_SW_16], [REQ_SW_17] i [REQ_SW_20].

Per realitzar aquesta prova s'ha utilitzat una sortida de la FPGA que no s'estava utilitzant per portar cap al exterior la senyal de tx_hold. Aquesta senyal permet a l'entitat can_recevie aturar la transmissió si es detecta algun problema o es perd l'arbitratge. En qualsevol d'aquestes dues situacions, quan l'entitat can_receive esborri la senyal tx_hold i el bus estigui en repòs, l'entitat can_transmit començarà a enviar la trama de dades aturada des del començament.

El procediment seguit per aquest test ha sigut desconnectar l'alimentació de l'Arduino mentre s'estan enviant trames de dades CAN des del sistema cap a l'Arduino. Abans de desconnectar l'alimentació, l'Arduino posarà el nivell dominat al camp de ACK i no hi haurà retransmissió per part del sistema dissenyat. Un cop s'hagi desconnectat l'alimentació, ja no hi haurà cap node llegint el missatge i el sistema haurà de detectar-ho i reenviar les trames de dades.

La Figura 51 mostra el resultat d'aquest test. Aquesta captura il·lustra el moment on es desconnecta l'alimentació de l'Arduino. La senyal CAN_rx mostra els missatges que apareixen al bus i la senyal tx_hold quan desapareix el ACK. Després de desaparèixer el ACK, s'observa com es retransmet la trama de dades CAN.

A la Figura 52 s'observa amb més detall com la senyal tx_hold apareix després de no rebre el ACK i desapareix quan el bus torna a l'estat de repòs. Immediatament després

de desaparèixer, la transmissió de la trama de dades no llegida es torna a transmetre. Al ser la mateixa reacció que quan es perd l'arbitratge, es pot concloure que la retransmissió funcionarà. Queda pendent de verificar si es detecta correctament que s'ha perdut l'arbitratge.

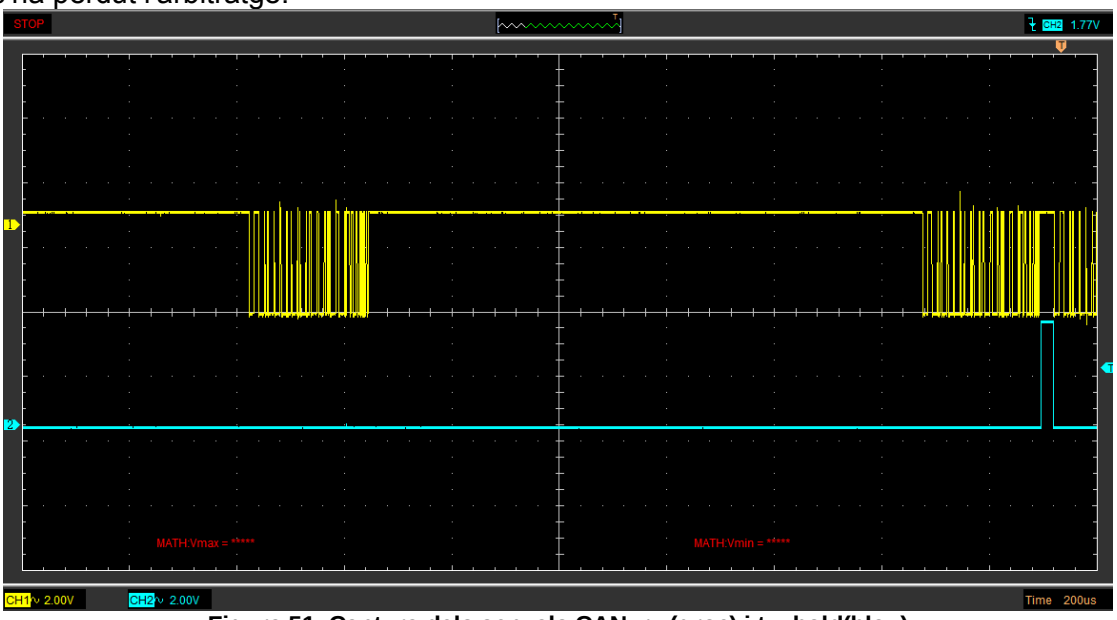


Figura 51. Captura dels senyals CAN_rx (groc) i tx_hold(blau)

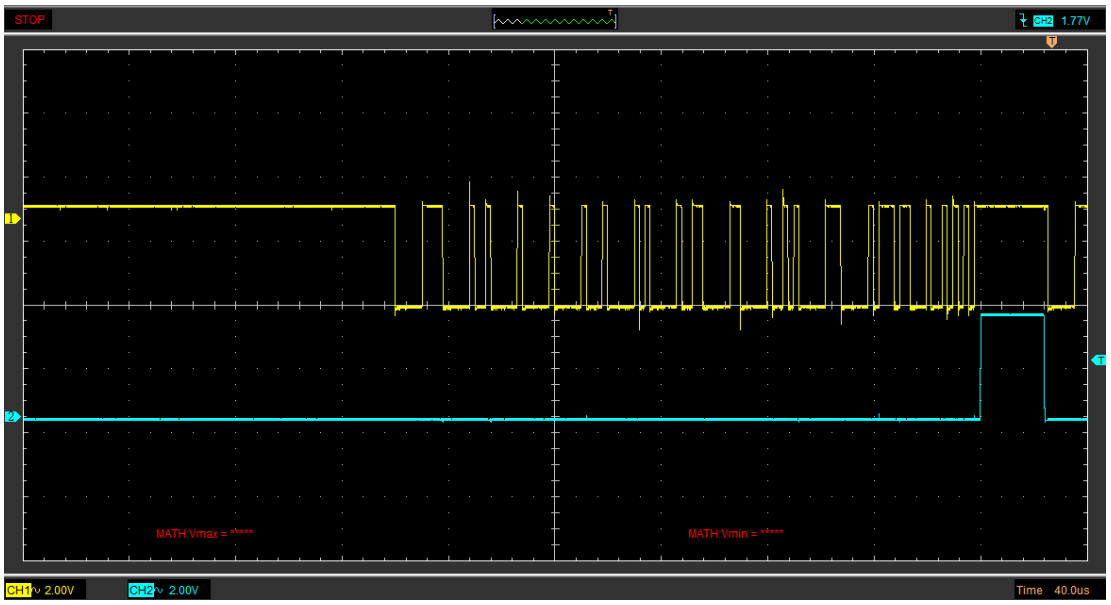


Figura 52. Captura dels senyals CAN_rx (groc) i tx_hold(blau) amb zoom

5.8 Test *flag* d'inici de transmissió

L'objectiu d'aquest test és comprovar que si el *flag* de start no s'esborra, no serà possible iniciar una nova transmissió de trames de dades. Amb aquest test es pretén validar els requisits: [REQ_SW_21], [REQ_SW_24] i [REQ_SW_25].

El *driver* CAN dissenyat automàticament esborra el *flag* de start per poder començar una nova transmissió. Com es mostra a la Figura 53, s'ha comentat la línia de codi on s'executa aquesta instrucció per comprovar que passa si no s'esborra aquest *flag*.

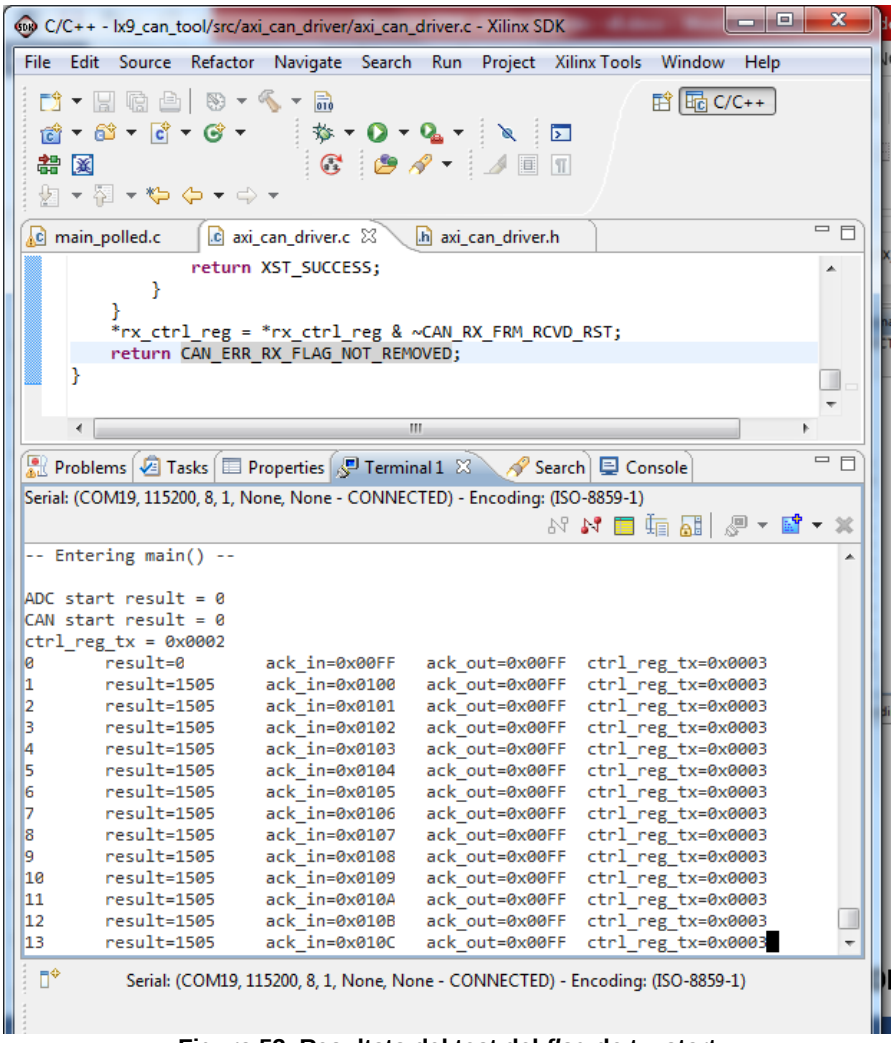


Figura 53. Resultats del test del *flag* de tx_start

Observant els resultats de la Figura 53, s'observa que el primer missatge CAN es pot enviar correctament, resultat igual a 0 i el ACK d'entrada és igual al ACK de sortida. En canvi, els següents missatges es rep com a resultat l'error 1505, que correspon a l'error que es produeix quan s'intenta enviar amb el *flag* d'start encara actiu. També es pot comprovar com el ACK de sortida no s'actualitza amb el d'entrada.

5.9 Test *flag* de trama enviada

L'objectiu d'aquest test és comprovar que passa quan no es rep el *flag* de trama enviada correctament. Amb aquest test es pretén validar el requisit [REQ_SW_25].

Per realitzar aquest test, s'ha modificat el codi del mòdul IP AXI_CAN_CUSTOM perquè quan s'acabi de transmetre, no s'enviï el *flag* de trama enviat correctament.

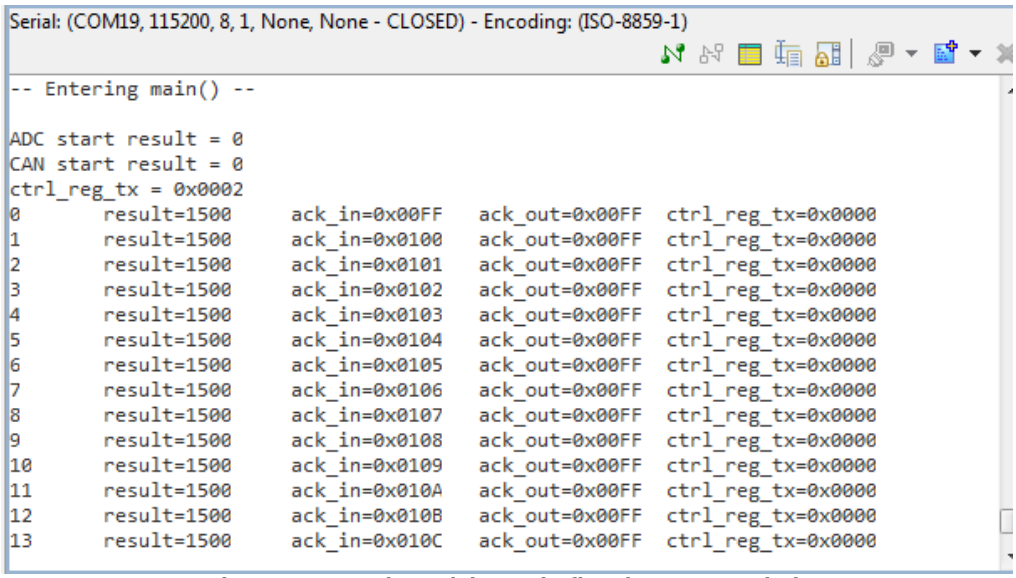


Figura 54. Resultats del test de *flag* de trama enviada

Observant els resultats de la Figura 54, la configuració de la interfície CAN és correcte ja que el resultat és 0 i el resultat d'executar la funció d'enviar trames de dades CAN és 1500. Aquest número d'error indica que no s'ha pogut llegir el *flag* d'enviament de trama correcta o que el *flag* no estava present abans de començar la transmissió. En la primera trama de dades, el *flag* és correcte abans de l'enviament i, per tant, s'envia la trama. La demostració és que el ACK d'entrada i sortida són iguals. En les següents trames de dades, el *flag* d'enviament correcte no està present i, per tant, es detecta l'error abans d'enviar. La demostració és que el ACK de sortida no s'actualitza.

5.10 Test de l'arbitratge de trames de dades

L'objectiu d'aquest test és comprovar que passa quan es comença a transmetre una trama de dades pel bus CAN i hi ha un altre node al bus que comença a transmetre simultàniament. Amb aquest test es pretén validar els requisits: [REQ_SW_16] i [REQ_SW_17].

Per realitzar aquest test, s'ha modificat el mòdul IP perquè l'entitat can_receive cregui que llegeix un missatge amb l'identificador 0x7FE. Llavors, quan s'intenti enviar una trama de dades amb un identificador menys prioritari es perdrà l'arbitratge. Per exemple, la trama amb l'identificador 0x7FF és menys prioritària que la trama amb l'identificador 0x7FE; per tant, 0x7FE guanyarà l'arbitratge a 0x7FF. En aquest cas l'entitat can_transmit s'haurà d'esperar a que s'acabi la transmissió per retransmetre les dades.

La Figura 55 mostra el resultat d'enviar un missatge amb l'identificador 0x7FF. Com es pot observar, en l'últim bit d'identificador enviat és perd l'arbitratge. En aquest instant la senyal tx_hold s'activa i evita que l'entitat can_transmit pugui seguir enviant. Al no haver-hi cap altre node transmetent de veritat, el bus queda en repòs i l'entitat can_receive detecta un error de *stuff* i s'atura. També es pot observar que després de posar a 0 el tx_hold, l'entitat can_transmit intenta retransmetre la trama de dades. En principi, al estar configurada per perdre l'arbitratge, hauria de tornar a repetir-se la situació anterior, però no és el cas. El tema és que el CAN-Shield de l'Arduino també detecta aquest error de *stuff* i inhibeix la comunicació amb una trama d'error (bit dominants).

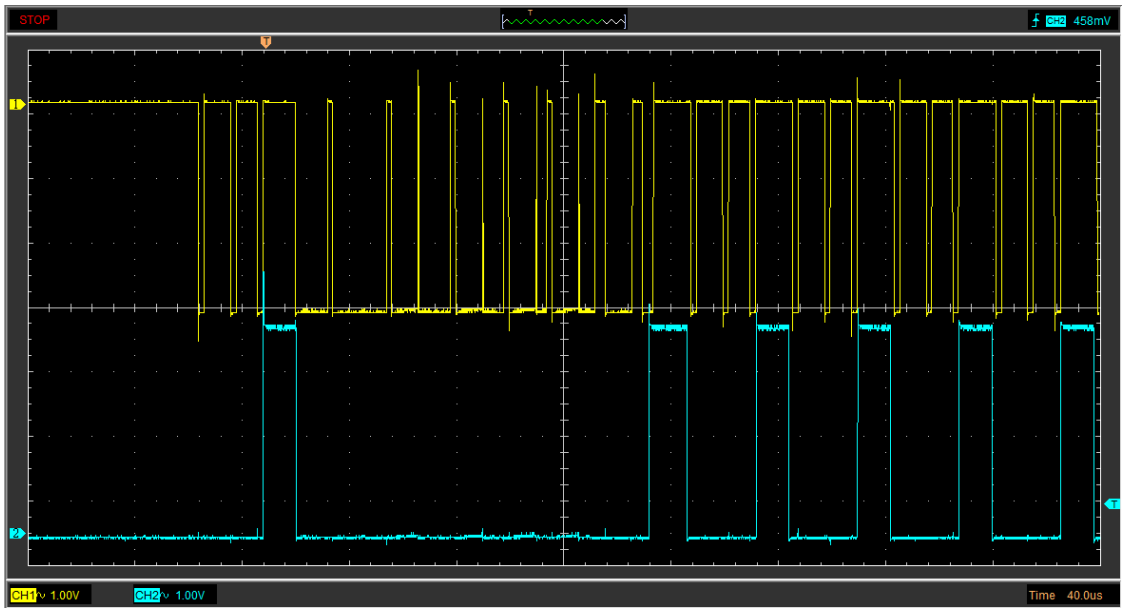


Figura 55. Resultats del test d’arbitratge de trames de dades, CAN_rx (groc) i tx_hold (blau)

6. CONCLUSIONS

La principal conclusió del projecte és que es pot dissenyar i implementar un sistema de comunicació CAN de baix cost sobre una plataforma FPGA, tot complint amb la normativa ISO 11898. Malgrat això, hi ha hagut algunes funcionalitats que no s'han pogut dur a terme i que seria interessant aprofundir en un futur:

- La gestió de trames remotes, error i sobrecàrrega, ja que actualment es llegeixen aquestes trames però, no es processen
- Un sistema de sincronització més sofisticat tenint en compte les diferents fases de temps de bit per fer el mostreig dels bits rebuts.

Tanmateix, aquest sistema és modular i es pot integrar en sistemes amb recursos molt limitats, com la FPGA Spartan-6, o millorar el rendiment afegint més mòduls de forma ràpida i senzilla. La creació de noves aplicacions no suposa cap problema, ja que el *driver* dissenyat en llenguatge de programació C requereix de només tres funcions, configurar, enviar i rebre, per equipar el sistema amb comunicació CAN.

Gràcies al mètode de treball utilitzat, mètode en V, s'ha assegurat l'èxit, la qualitat i la rigorositat del projecte. Aquest mètode garanteix que els requisits del disseny tinguin associats un test que verifiqui que s'han implementat correctament i exigeix justificar els casos en què els requisits no s'hagin pogut definir.

Els reptes més importants del projecte han estat els següents:

- La capacitat limitada de la Spartan-6 ha obligat a replantejar la implementació del *driver* diverses vegades des de 0.
- Els diferents entorns de programació utilitzats són molt potents però requereixen d'un gran esforç de configuració quan es vol implementar un canvi important dins del codi.
- Algunes de les funcions més avançades necessàries per la creació d'un mòdul IP complex, com el que s'ha plantejat en aquest projecte, no estan gaire ben documentades.
- Les eines de simulació disponibles de Xilinx van molt bé per entitats VHDL simples però, per les més sofisticades, cal invertir una quantitat de temps molt gran. En la majoria dels casos, només ha sigut possible trobar els errors amb l'oscil·loscopi o amb les dades enviades pel port sèrie.
- La sintetització del mòdul IP i el sistema Microblaze requereix molt de temps i fer proves s'allarga molt.

Així doncs, malgrat ha suposat un gran esforç, ha estat un projecte molt interessant en el que de segur es continuarà aprofundint més endavant, ja sigui en relació a les funcionalitats esmentades com en d'altres que puguin aparèixer.

7. BIBLIOGRAFIA

- Documentació referent a la descripció teòrica:

Robert Bosch GmbH. *CAN Specification* [en línia]. Stuttgart: Robert Bosch, 1991. Revisió 2.0 [Consulta: 10 maig 2017]. Disponible a: <http://esd.cs.ucr.edu/webres/can20.pdf>

Blackman, Jason; Monroe, Scott. *SLLA337: Overview of 3.3V CAN (Controller Area Network) Transceivers* [en línia]. Dallas: Texas Instruments, 2013 [Consulta: 22 novembre 2016]. Disponible a: <http://www.ti.com/lit/an/slla337/slla337.pdf>

CAN in Automotion group. *Cyclic redundancy Check (CRC) in CAN frames* [en línia]. Nuremberg [Consulta: 2 desembre 2016]. Disponible a: <https://www.can-cia.org/can-knowledge/can/crc/>

ISO. *ISO 11898-1:2015, Road vehicles – Controller Area Network (CAN) – Part 1: Data link layer and physical signalling.* ISO, 2015.

ISO. *ISO 11898-2:2016, Road vehicles – Controller Area Network (CAN) – Part 2: High-speed mèdium access unit.* ISO, 2016.

ISO. *ISO 11898-3:2006, Road vehicles – Controller Area Network (CAN) – Part 3: Low-speed, fault-tolerant, mèdium dependent interface.* ISO, 2006.

- Fulls de característiques de components electrònics:

Microchip. *MCP2515: Stand-Alone CAN Controller with SPI Interface* [en línia]. Chandler: Microchip. Actualització 11 juliol 2016. [Consulta: 9 novembre 2016]. Disponible a: <http://ww1.microchip.com/downloads/en/DeviceDoc/20001801H.pdf>

Microchip. *MCP3002: 2.7V Dual Channel 10-Bit A/D Converter with SPI™ Serial Interface* [en línia]. Chandler: Microchip. Actualització 8 febrer 2011. [Consulta: 9 novembre 2016]. Disponible a: <http://ww1.microchip.com/downloads/en/DeviceDoc/21294E.pdf>

Pang, Sukkin; Owens, Ryan. *Canbus-shield-v12* [en línia]. Colorado: SparkFun Electronics, 2010 [Consulta: 8 novembre 2016]. Disponible a: <https://www.sparkfun.com/datasheets/DevTools/Arduino/canbus_shield-v12.pdf>

Texas Instruments. *CD4049UB and CD4050B CMOS Hex Inverting Buffer and Converter* [en línia]. Dallas: Texas Instruments, 1998. Actualitzat setembre 2016 [Consulta: 9 novembre 2016]. Disponible a: <http://www.ti.com/lit/ds/symlink/cd4050b.pdf>

Texas Instruments. *SN65HVD230Q-Q1, SN65HVD231Q-Q1, SN65HVD232Q-Q1: 3.3-V CAN TRANSCEIVERS* [en línia]. Dallas: Texas Instruments, 2002. Actualitzat abril 2008 [Consulta: 23 novembre 2016]. Disponible a: <http://www.ti.com/lit/ds/sgls398a/sgls398a.pdf>

Kingbright. *DA03-11SURKWA: 7.62mm (0.3INCH) DUAL DIGIT NUMBER DISPLAY* [en línia]. Taiwan: Kingbright, 2015. Revisió V.2A [Consulta: 9 novembre 2016]. Disponible a: <http://www.kingbrightusa.com/images/catalog/SPEC/DA03-11SURKWA.pdf>

Analog Devices. *TMP35/TMP36/TMP37: Low Voltage Temperature Sensors* [en línia] Norwood: Analog Devices. Revisió H [Consulta: 8 novembre 2016]. Disponible a: <http://www.analog.com/media/en/technical-documentation/data-sheets/TMP35_36_37.pdf>

- Manuals de programació:

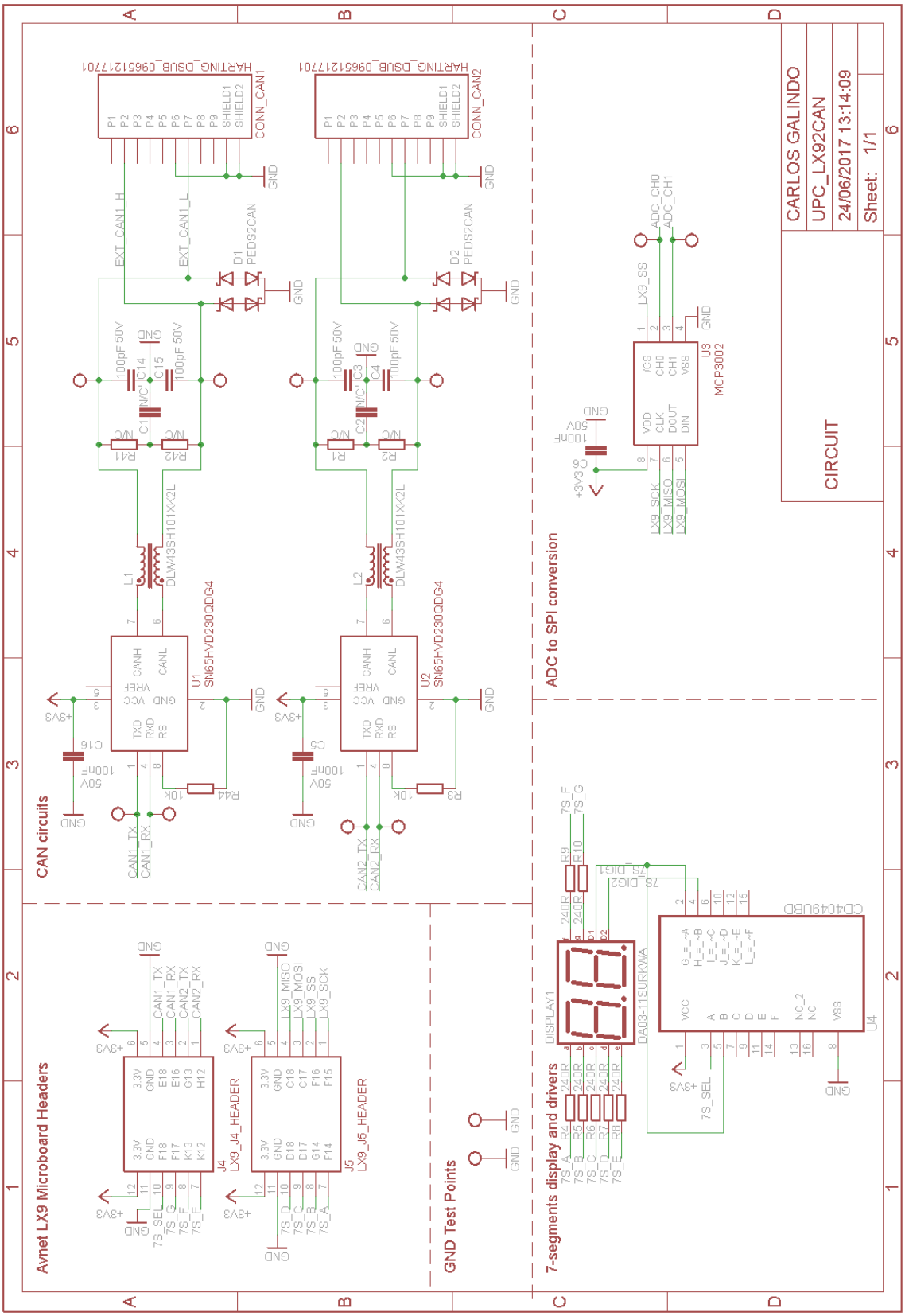
AVNET Engineering Services. *Spartan-6 LX9 MicroBoard Configuration Guide: Configuring the Xilinx Spartan-6 LX9 MicroBoard* [en línia]. Phoenix: AVNET Engineering Services, 2011. Actualització 1 setembre 2011. Revisió 13.2.1 [Consulta: 18 maig 2017]. Dispnible a: <https://www.avnet.com/opasdata/d120001/medias/docus/7/AES-S6MB-LX9-G-User-Guide-Configuration-Guide-IDS-13_2.pdf>

AVNET Engineering Services. *Spartan-6 LX9 MicroBoard Embedded Tutorial – Tutorial 3: Adding Custom IP to an Embedded System* [en línia]. Phoenix: AVNET Engineering Services, 2011. Actualització 16 maig 2011. Revisió 13.1.01 [Consulta: 2 febrer 2017]. Disponible a: <https://www.avnet.com/opasdata/d120001/medias/docus/7/AES-S6MB-LX9-G-EDK-13_1-Adding-Custom-IP.pdf>

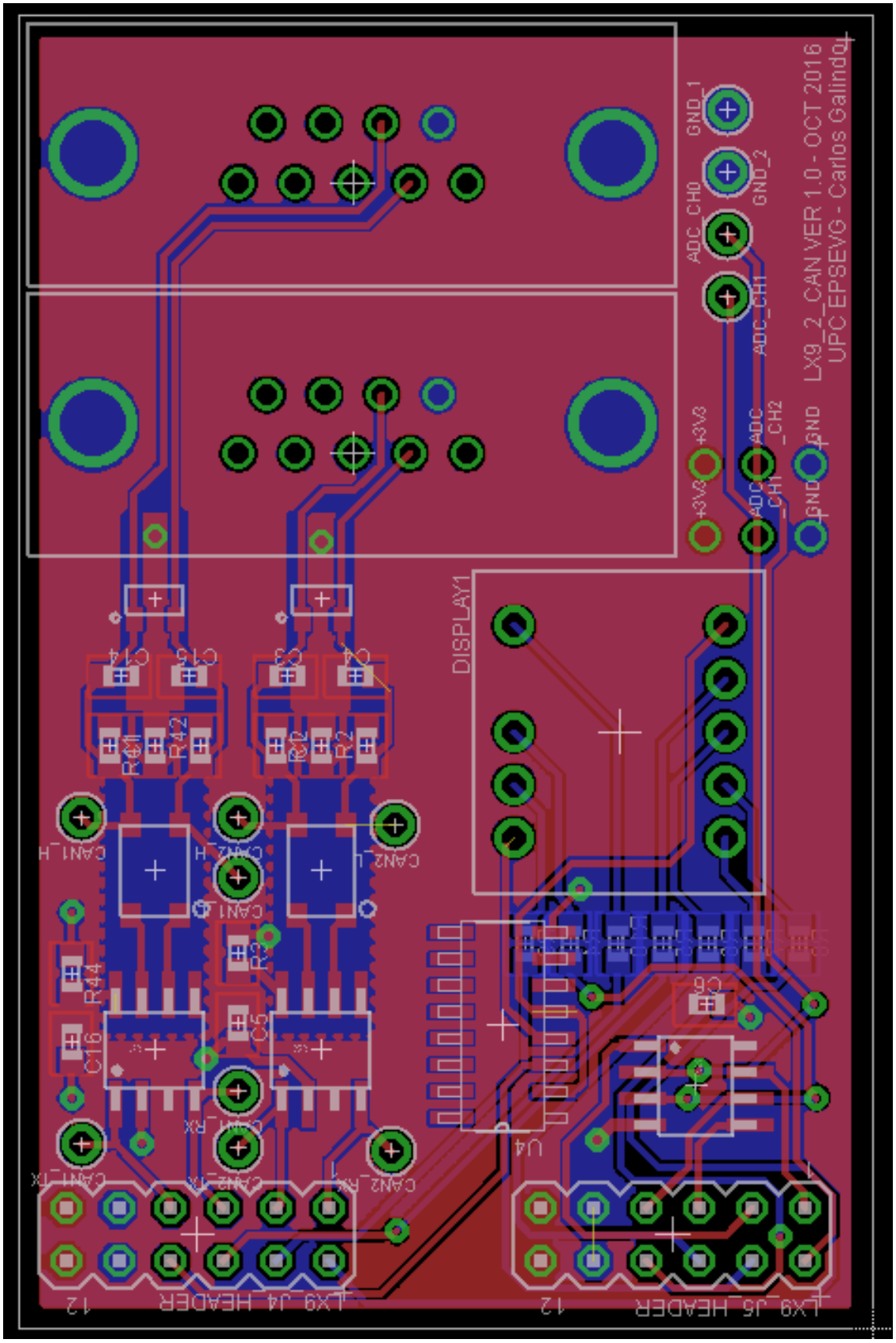
AVNET Engineering Services. *Spartan-6 LX9 Microboard Rev C* [en línia]. Phoenix: AVNET Engineering Services, 2012. Actualització 2 agost 2012. Revisió C [Consulta: 2 febrer 2017]. Disponible a: <https://www.avnet.com/opasdata/d120001/medias/docus/7/AES-S6MB-LX9-G-Schematics-RevC-Schematics.PDF>

ANNEX 1 – Fitxers de disseny LX9_2_CAN

1. Esquemàtics



2. Layout


LX9_2_CAN VER 1.0 - OCT 2016
UPC EPSEVG - Carlos Galindo
GND_1
GND_2
ADC_CH0
ADC_CH1
ADC_CH2
+3V3
GND
DISPLAY1
CAN1_H
CAN1_L
CAN2_H
CAN2_L
CAN1_RX
CAN1_TX
CAN2_RX
CAN2_TX
LX9_J4_HEADER
LX9_J5_HEADER
U4

ANNEX 3 – Codi mòdul IP: AXI_CAN_CUSTOM

1. Entitat DIVIDER.VHD

```
----------------------------------------------------------------------------------
-- Company: UPC
-- Engineer: Carlos Galindo
--
-- Create Date:    19:55:16 05/02/2016
-- Module Name:    divider - Behavioral
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity divider is
   generic(fact: natural:=4); -- Indicates the number of divisions
   port(clk_in: in std_logic; -- System clock
      rst: in std_logic; -- External reset signal
      clk_out: out std_logic); -- Divided clock signal (only on for one sys clk period
end divider;

architecture Behavioral of divider is
begin
   p:process(clk_in)is
      variable count: natural:=0;
      variable rst_disable: std_logic;
   begin
      if(clk_in'event and clk_in='1')then
         if (rst = '1')then -- Synchronous reset
            count := 0; -- Reset the counter
            clk_out <= '1'; -- Start with on state
         elsif (count<fact-1) then -- Check if it is below limit
            count:=count+1; -- Increase counter
            clk_out<='0'; -- Off state
         else
            count:=0; -- Reset counter
            clk_out <= '1'; -- On state
         end if;
      end if;
   end process p;


end Behavioral;


```

2. Entitat DELAY.VHD

```
----------------------------------------------------------------------------------
-- Company: UPC
-- Engineer: Carlos Galindo
--
-- Create Date:    19:55:16 05/02/2016
-- Module Name:    delay - Behavioral
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity delay is
   generic(fact: natural:=4); -- Indicates the number of clk periods to delay out clk signal
   port(clk_in: in std_logic; -- System clock
      clk_divided: in std_logic; -- Divided clock signal to be delayed
      clk_out: out std_logic); -- Divided clck signal delayed
end delay;
```

```
21  begin
22     p:process(clk_in)is
23        variable count: natural:=0;
24        variable enable: std_logic:='0';
25        variable clk_divided_last: std_logic := '0';
26     begin
27        if(clk_in'event and clk_in='1')then
28           -- Start counting when there is a rising edge
29           if (clk_divided = '1' and clk_divided /= clk_divided_last)then
30              count := 0; -- Reset counter
31              enable := '1'; -- Trigger delay counter start
32           end if;
33           -- Check if delay counter is enabled
34           if(enable = '1') then
35              -- Check if counter is below the limit
36              if (count<fact-1) then
37                 count:=count+1; -- Increase counter
38                 clk_out<='0'; -- Off state
39              else
40                 count:=0; -- Reset counter
41                 clk_out <= '1'; -- On state
42                 enable := '0'; -- Disable the counter
43              end if; -- count<fact-1
44           else
45              clk_out <= '0'; -- If disabled, output signal always off
46           end if; --enable = '1'
47           clk_divided_last := clk_divided;  -- Update last status with the current one
48        end if; --clk_in'event and clk_in='1'
49     end process p;
50  end Behavioral;
```

3. Paquet CAN_VHDL_PACKAGE.VHD

```
5   library IEEE;
6   use IEEE.STD_LOGIC_1164.all;
7   use ieee.numeric_std.all;
8   use ieee.std_logic_unsigned.all;
9
10  package CAN_VHDL_PACKAGE is
11     function CRC_CALCULATION (crc_next_bit: std_logic;
12                               signal crc_reg_in: std_logic_vector(14 downto 0))
13     return std_logic_vector;
14  end CAN_VHDL_PACKAGE;
15
16  package body CAN_VHDL_PACKAGE is
17     function CRC_CALCULATION (crc_next_bit: std_logic;
18                               signal crc_reg_in: std_logic_vector(14 downto 0))
19     return std_logic_vector is
20        variable crc_next: std_logic;
21        variable crc_reg_out: std_logic_vector(14 downto 0);
22     begin
23        crc_next := crc_next_bit xor crc_reg_in(14);
24        crc_reg_out(14 downto 1) := crc_reg_in(13 downto 0);
25        crc_reg_out(0) := '0';
26        if (crc_next = '1') then
27           crc_reg_out(14 downto 0) := crc_reg_out(14 downto 0) xor "100010110011001"; -- 100010110011001 -> 0x4599
28        end if;
29        return crc_reg_out;
30     end CRC_CALCULATION;
31  end CAN_VHDL_PACKAGE;
```

4. Entitat CAN_RECEIVE.VHD

```
1   ----------------------------------------------------------------------------------
2   -- Company: UPC
3   -- Engineer: Carlos Galindo
4   --
5   -- Create Date:    21:57:05 05/31/2017
6   -- Module Name:    can_receive - Behavioral
7   ----------------------------------------------------------------------------------
8   library ieee;
9   use ieee.std_logic_1164.ALL;
10  use ieee.numeric_std.all;
11  use ieee.std_logic_unsigned.all;
12
13  library axi_can_custom_v1_11_a;
14  use axi_can_custom_v1_11_a.divider;
15  use axi_can_custom_v1_11_a.delay;
16  use axi_can_custom_v1_11_a.can_vhdl_package.all;
```

```
entity can_receive is
   generic(
      CAN_ID_BIT_LENGTH            : integer              := 11; -- Non-extended ID
      CAN_RTR_BIT_LENGTH           : integer              := 1;
      CAN_IDE_BIT_LENGTH           : integer              := 1;
      CAN_RSRVD_BIT_LENGTH         : integer              := 1;
      CAN_DLC_BIT_LENGTH           : integer              := 4;
      CAN_DATA_BIT_LENGTH          : integer              := 8;
      CAN_CRC_BIT_LENGTH           : integer              := 15;
      CAN_EOF_BIT_LENGTH           : integer              := 7;
      CAN_MAX_DATA_FRAMES          : integer              := 8;
      BAUD_RATE_PRESCALER_VAL      : integer              := 133 -- 500 kBauds
   );
   port(
      clk                       : in  std_logic; -- System clock
      baud_rate_clk_out         : out  std_logic; -- Clock at CAN bus baud rate for sync
      rx_frm_rcvd_rst           : in  std_logic; -- Flag to reset the frame received flag
      rx_frm_rcvd               : out std_logic; -- Flag that indicates frame received
      rx_tx_hold                : out std_logic; -- Flag to indicate can_transmission to wait
      rx_result                 : out std_logic_vector(3 downto 0); -- Provides feedback about frame integrity
      tx_id                     : in std_logic_vector(CAN_ID_BIT_LENGTH-1 downto 0); -- Transmit frame ID
      rx_id                     : out std_logic_vector(CAN_ID_BIT_LENGTH-1 downto 0); -- Frame ID
      rx_dlc                    : out std_logic_vector(CAN_DLC_BIT_LENGTH-1 downto 0); -- Frame DLC
      rx_data_byte0             : out std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0); -- Frame byte 0
      rx_data_byte1             : out std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0); -- Frame byte 1
      rx_data_byte2             : out std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0); -- Frame byte 2
      rx_data_byte3             : out std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0); -- Frame byte 3
      rx_data_byte4             : out std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0); -- Frame byte 4
      rx_data_byte5             : out std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0); -- Frame byte 5
      rx_data_byte6             : out std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0); -- Frame byte 6
      rx_data_byte7             : out std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0); -- Frame byte 7
      rx_crc                    : out std_logic_vector(CAN_CRC_BIT_LENGTH-1 downto 0); -- Frame CRC
      rx_pin_in                 : in std_logic; -- CAN rx pin
      rx_tx_pin_out             : out std_logic -- CAN rx tx pin, used to acknowdlege in messages
   );
end can_receive;

architecture Behavioral of can_receive is
   type t_state is (hold,sof, id,rtr,ide,rsrvd,dlc,data,crc,crc_del,
                    ack,ack_del,eof);
   signal rx_state : t_state;
   signal clk_sync : std_logic;
   signal baud_rate_clk : std_logic;
   signal baud_rate_clk_sync : std_logic;
   --signal bit_to_stuff : std_logic;
begin

--Instance of the divider to generate clk signal at CAN baud rate frequency
baud_rate_clk_divider: entity axi_can_custom_v1_11_a.divider
   generic map(fact => BAUD_RATE_PRESCALER_VAL)
   port map (clk_in => clk,
             rst => clk_sync,
             clk_out => baud_rate_clk);

-- Instance of the delay to delay baud_rate_clk 15 system clock periods
baud_rate_clk_delay: entity axi_can_custom_v1_11_a.delay
   generic map(fact => 15)
   port map(clk_in => clk,
            clk_divided => baud_rate_clk,
            clk_out => baud_rate_clk_sync);

-- Drive delayed signal to a digital output for debugging
baud_rate_clk_out <= baud_rate_clk_sync;
can_receive: process (clk)
      variable bit_stuff_count: integer range 0 to 6; -- Counter to check bit stuffing
      variable bit_to_stuff: std_logic; -- Flag to check if stuff bit is in place
      variable bit_count: integer range 0 to 15; -- Counter to read specific bits from CAN fields
      variable data_byte_count: integer range 0 to 8; -- Counter to read specific CAN data byte
      variable data_dlc: integer range 0 to 8; -- Number of bytes to read

      variable bit_rcvd: std_logic; -- Variable to store current CAN rx pin value
      variable last_bit_rcvd: std_logic; -- Variable to store last bit received processed

      variable rx_frm_rcvd_var: std_logic; -- Flag to indicate CAN frame received
      variable rx_tx_hold_var : std_logic; -- Flag to indicate if CAN frame read is being send by the controller
      variable rx_id_var           : std_logic_vector(CAN_ID_BIT_LENGTH-1 downto 0); -- Frame ID
      variable rx_dlc_var          : std_logic_vector(CAN_DLC_BIT_LENGTH-1 downto 0); -- Frame DLC
      variable rx_data_byte0_var   : std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0); -- Frame byte 0
      variable rx_data_byte1_var   : std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0); -- Frame byte 1
      variable rx_data_byte2_var   : std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0); -- Frame byte 2
      variable rx_data_byte3_var   : std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0); -- Frame byte 3
      variable rx_data_byte4_var   : std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0); -- Frame byte 4
      variable rx_data_byte5_var   : std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0); -- Frame byte 5
      variable rx_data_byte6_var   : std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0); -- Frame byte 6
      variable rx_data_byte7_var   : std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0); -- Frame byte 7
      variable rx_crc_var          : std_logic_vector(CAN_CRC_BIT_LENGTH-1 downto 0); -- Frame CRC

      variable rx_tx_pin_out_var   : std_logic; -- Flag to override transmission CAN tx value to send ACK
```

```
108   begin
109      if(clk'event and clk = '1')then
110         bit_rcvd := rx_pin_in; -- Load with CAN rx pin level
111
112         --- Below code implement: [REQ_SW_6], [REQ_SW_7], [REQ_SW_8], [REQ_SW_9] ---
113         if(last_bit_rcvd /= bit_rcvd)then -- Check if bit read is different from previous one
114
115            --- Below code implement: [REQ_SW_28] ---
116            -- Do not re-synchronise in ack_del or ack state
117            if(rx_state = ack_del or rx_state = ack)then
118               clk_sync <= '0';
119            else -- It is allowed to synchronise
120               clk_sync <= '1';
121            end if;
122            --- Upper code implement: [REQ_SW_28] ---
123
124         else
125            clk_sync <= '0';
126         end if;
127         --- Upper code implement: [REQ_SW_6], [REQ_SW_7], [REQ_SW_8], [REQ_SW_9] ---
129         if(baud_rate_clk_sync = '1')then -- State machine is executed with delayed baud rate signal
130            if(bit_to_stuff = '0')then -- Only execute state machine if there is no sutff bit
131
132               --- Below code implement: [REQ_SW_11] ---
133               case rx_state is
134                  when id =>
135                     bit_count := bit_count - 1; -- Decrease bit counter
136                     rx_id_var(bit_count) := bit_rcvd;
137                     -- Check if id from CAN tx is equal to the one being read, once a difference
138                     -- is detected, it is blocked to avoid modifications
139                     if(bit_rcvd /= tx_id(bit_count) and rx_tx_hold_var = '0') then
140                        rx_tx_hold_var := '1'; -- Set flag to indicate message from a different node
141                     end if;
142                     if(bit_count = 0)then -- If bit counter is 0, move to RTR state
143                        bit_count := CAN_RTR_BIT_LENGTH;
144                        rx_state <= rtr;
145                     end if;
146                  when rtr =>
147                     bit_count := bit_count - 1; -- Decrease bit counter
148                     if(bit_count = 0)then -- If bit counter is 0, move to IDE state
149                        bit_count := CAN_IDE_BIT_LENGTH;
150                        rx_state <= ide;
151                     end if;
152                  when ide =>
153                     -- In a conventional CAN controller this value will be check to know if a
154                     -- retransmission is required but so far it is not implemented
155                     bit_count := bit_count - 1; -- Decrease bit counter
156                     if(bit_count = 0)then -- If bit counter is 0, move to RSRVD state
157                        bit_count := CAN_RSRVD_BIT_LENGTH;
158                        rx_state <= rsrvd;
159                     end if;
160                  when rsrvd =>
161                     -- CAN standard request not to check this value since it can be dominant
162                     -- in the future in order to add new functionality
163                     bit_count := bit_count - 1; -- Decrease bit counter
164                     if(bit_count = 0)then -- If bit counter is 0, move to DLC state
165                        bit_count := CAN_DLC_BIT_LENGTH;
166                        rx_state <= dlc;
167                     end if;
168                  when dlc =>
169                     bit_count := bit_count - 1; -- Decrease bit counter
170                     rx_dlc_var(bit_count) := bit_rcvd; -- Store dlc value
171                     if(bit_count = 0) then -- Move to next state if bit counter is 0
172                        data_dlc := to_integer(unsigned(rx_dlc_var));
173                        if (data_dlc = 0) then -- If there is no data to read, move to CRC state
174                           bit_count := CAN_CRC_BIT_LENGTH;
175                           rx_state <= crc;
176                        else -- If there is data to read, move to DATA state
177                           bit_count := CAN_DATA_BIT_LENGTH;
178                           data_byte_count := 0;
179                           rx_state <= data;
180                        end if;
181                     end if;
```

```
182                 when data =>
183                    bit_count := bit_count - 1; -- Decrease bit counter
184                    case data_byte_count is -- Bit to read depends on bit and byte counters
185                       when 0 => rx_data_byte0_var(bit_count) := bit_rcvd;
186                       when 1 => rx_data_byte1_var(bit_count) := bit_rcvd;
187                       when 2 => rx_data_byte2_var(bit_count) := bit_rcvd;
188                       when 3 => rx_data_byte3_var(bit_count) := bit_rcvd;
189                       when 4 => rx_data_byte4_var(bit_count) := bit_rcvd;
190                       when 5 => rx_data_byte5_var(bit_count) := bit_rcvd;
191                       when 6 => rx_data_byte6_var(bit_count) := bit_rcvd;
192                       when others => rx_data_byte7_var(bit_count) := bit_rcvd;
193                    end case;
194                    if (bit_count = 0) then -- If bit counter is equal to 0, move to next byte or state
195                       data_byte_count := data_byte_count + 1; -- Increase byte to read
196                       bit_count := CAN_DATA_BIT_LENGTH;
197                       if (data_byte_count = data_dlc) then -- If byte is equal to dlc, move to CRC state
198                          bit_count := CAN_CRC_BIT_LENGTH;
199                          rx_state <= crc;
200                       else -- If byte to read is less than dlc, read another byte
201                          bit_count := CAN_DATA_BIT_LENGTH;
202                       end if;
203                    end if;
204                 when crc =>
205                    bit_count := bit_count - 1; -- Decrease bit counter
206                    rx_crc_var(bit_count) := bit_rcvd; -- Read CRC field
207                    if(bit_count = 0)then -- If bit count is equal to 0, move to CRC_DEL state
208                       rx_state <= crc_del;
209                    end if;
210                 when crc_del =>
211                    rx_state <= ack; -- Directly move to ACK state, no check
212                 when ack =>
213
214                    --- Below code implement: [REQ_SW_28] ---
215                    rx_tx_pin_out_var := '0'; -- Set dominant level bit in the CAN tx pin as ACK
216                    --- Upper code implement: [REQ_SW_28] ---
217
218                    rx_state <= ack_del;  -- Directly move to ACK_DEL state
219                 when ack_del =>
220
221                    --- Below code implement: [REQ_SW_28] ---
222                    rx_tx_pin_out_var := '1'; -- Set recessive level bit in the CAN tx pin
223                    --- Upper code implement: [REQ_SW_28] ---
224
225                    bit_count := CAN_EOF_BIT_LENGTH;
226                    rx_state <= eof; -- Directly move to EOF state
227                 when eof =>
228                    bit_count := bit_count - 1; -- Decrease counter
229                    if(bit_count = 0)then -- If bit counter is equal to 0, message is read
230                       rx_state <= hold; -- Move to bus idle
231                       rx_tx_pin_out_var := '1'; -- Set to recessive just in case
232                       -- If message read is transmited by other node than this one,
233                       -- move load the read data to output data
234                       if(rx_tx_hold_var = '1')then
235
236                          --- Below code implement: [REQ_SW_26] ---
237                          rx_frm_rcvd_var := '1'; -- Indicate CAN message received
238                          --- Below code implement: [REQ_SW_26] ---
239
240                          --- Below code implement: [REQ_SW_27] ---
241                          rx_result <= "0001"; -- Indicate the result of reading the message is ok
242                          --- Below code implement: [REQ_SW_27] ---
243
244                          rx_id <= std_logic_vector(rx_id_var);
245                          rx_dlc <= std_logic_vector(rx_dlc_var);
246                          rx_data_byte0 <= std_logic_vector(rx_data_byte0_var);
247                          rx_data_byte1 <= std_logic_vector(rx_data_byte1_var);
248                          rx_data_byte2 <= std_logic_vector(rx_data_byte2_var);
249                          rx_data_byte3 <= std_logic_vector(rx_data_byte3_var);
250                          rx_data_byte4 <= std_logic_vector(rx_data_byte4_var);
251                          rx_data_byte5 <= std_logic_vector(rx_data_byte5_var);
252                          rx_data_byte6 <= std_logic_vector(rx_data_byte6_var);
253                          rx_data_byte7 <= std_logic_vector(rx_data_byte7_var);
254                          rx_crc <= std_logic_vector(rx_crc_var);
255                       end if;
256                    end if;
257                 when others => -- CAN bus is idle
258                    rx_tx_hold_var := '0'; -- Reset this flag
259                    rx_tx_pin_out_var := '1'; -- Ensure transmission is not on hold
260                    if(bit_rcvd = '0')then -- Check for message in the CAN bus
261                       -- Uncomment below line to not allow new message to be read if not read by application
262                       --if(rx_frm_rcvd_var = '0' and rx_frm_rcvd_rst = '0')then
263                       -- Uncomment upper line to not allow new message to be read if not read by application
264                          bit_stuff_count := 1;
265                          bit_count := CAN_ID_BIT_LENGTH;
```

```
                        --- Below code implement: [REQ_SW_27] ---
                        rx_result <= "0000"; -- Reset reading results
                        --- Upper code implement: [REQ_SW_27] ---

                        rx_state <= id; -- Start reading the CAN message
                     -- Uncomment below line to not allow new message to be read if not read by application
                     --end if;
                     -- Uncomment uper line to not allow new message to be read if not read by application
                  end if; -- bit-rcvd = '0'
                  -- Check if read message is read by application

                  --- Below code implement: [REQ_SW_26] ---
                  if(rx_frm_rcvd_rst = '1')then
                     rx_frm_rcvd_var := '0'; -- Remove the new message read flag

                      -- Delete stored read values
                      rx_id <= (others => '0');
                      rx_dlc <= (others => '0');
                      rx_data_byte0 <= (others => '0');
                      rx_data_byte1 <= (others => '0');
                      rx_data_byte2 <= (others => '0');
                      rx_data_byte3 <= (others => '0');
                      rx_data_byte4 <= (others => '0');
                      rx_data_byte5 <= (others => '0');
                      rx_data_byte6 <= (others => '0');
                      rx_data_byte7 <= (others => '0');
                   end if; -- rx_frm_rcvd_ack = '1'
                   --- Upper code implement: [REQ_SW_26] ---

               end case;
               --- Upper code implement: [REQ_SW_11] ---
            --- Below code implement: [REQ_SW_14], [REQ_SW_15] ---
            -- Implement bit stuffing for al frame fields except crc delimiter,
            -- ack and eof
            if(rx_state = sof or rx_state = id or rx_state = rtr or rx_state = ide or
               rx_state = rsrvd or rx_state = dlc or rx_state = data or
               rx_state = crc)then

               if (last_bit_rcvd = bit_rcvd) then
                  -- If bit to send is equal to last bit sent, increase the
                  -- repeated bit counter and check counter overflow
                  bit_stuff_count := bit_stuff_count + 1;
                  if(bit_stuff_count = 5)then
                     -- If more than 5 consecutive bits of the same value
                     -- are sent, sent the opposite value and reset the count
                     bit_to_stuff := '1';
                  end if;
               else
                  bit_stuff_count := 1;
               end if;
               --- Upper code implement: [REQ_SW_14], [REQ_SW_15]  ---

            end if;
         else -- bit_to_stuff = '1'
            --- Below code implement: [REQ_SW_14], [REQ_SW_15] ---
            -- Check if bit stuffed has the correct value, if not stop reading and indicate
            -- an error as result
            if(last_bit_rcvd = bit_rcvd)then
               rx_state <= hold; -- Move to CAN bus idle

               --- Below code implement: [REQ_SW_27] ---
               rx_result <= "1010"; -- Indication of error
               --- Below code implement: [REQ_SW_27] ---

               rx_frm_rcvd_var := '1'; -- Indication of message read but with error
            end if;
            bit_to_stuff := '0'; -- Acknowledge stuffed bit
            bit_stuff_count := 1; -- Reset bit stuff counter
            --- Upper code implement: [REQ_SW_14], [REQ_SW_15] ---

         end if; --bit_to_stuff = '0'
         last_bit_rcvd := bit_rcvd; -- Update last bit received
         rx_frm_rcvd <= std_logic(rx_frm_rcvd_var);
         rx_tx_hold <= std_logic(rx_tx_hold_var);
         rx_tx_pin_out <= std_logic(rx_tx_pin_out_var);
      end if; -- baud_rate_sync = '1'
   end if; -- clk'event = '1' and clk = '1'

end process can_receive;
end Behavioral;
```

5. Entitat CAN_TRANSMIT.VHD

```
----------------------------------------------------------------------------------
-- Company: UPC
-- Engineer: Carlos Galindo
--
-- Create Date:    00:38:50 03/15/2017
-- Module Name:    can_transmit - Behavioral
----------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

library axi_can_custom_v1_11_a;
use axi_can_custom_v1_11_a.can_vhdl_package.all;

entity can_transmit is
   generic(
      CAN_ID_BIT_LENGTH            : integer               := 11; -- Non-extended ID
      CAN_RTR_BIT_LENGTH           : integer               := 1;
      CAN_IDE_BIT_LENGTH           : integer               := 1;
      CAN_RSRVD_BIT_LENGTH         : integer               := 1;
      CAN_DLC_BIT_LENGTH           : integer               := 4;
      CAN_DATA_BIT_LENGTH          : integer               := 8;
      CAN_CRC_BIT_LENGTH           : integer               := 15;
      CAN_EOF_BIT_LENGTH           : integer               := 7;
      CAN_MAX_DATA_FRAMES          : integer               := 8);
   port (clk                      : in  std_logic; --System clock
         baud_rate_clk            : in  std_logic; -- Clock signal at CAN baud rate frequency
         tx_start                 : in  std_logic; -- External signal to trigger CAN transmission start
         tx_hold                  : in  std_logic; -- External signal to stop/resend CAN message
         tx_end                   : out std_logic; -- Flag to indicate that message has been correctly sent
         tx_frame_ack_in          : in  std_logic_vector(31 downto 0); -- Reference to identified sent frame
         tx_frame_ack_out         : out std_logic_vector(31 downto 0); -- Reference to check if sent frame is sent
         tx_id                    : in  std_logic_vector(CAN_ID_BIT_LENGTH-1 downto 0); -- ID for transmission
         tx_dlc                   : in  std_logic_vector(CAN_DLC_BIT_LENGTH-1 downto 0); -- DLC for transmission
         tx_data_byte0            : in  std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0); -- Byte0 for transmission
         tx_data_byte1            : in  std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0); -- Byte1 for transmission
         tx_data_byte2            : in  std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0); -- Byte2 for transmission
         tx_data_byte3            : in  std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0); -- Byte3 for transmission
         tx_data_byte4            : in  std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0); -- Byte4 for transmission
         tx_data_byte5            : in  std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0); -- Byte5 for transmission
         tx_data_byte6            : in  std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0); -- Byte6 for transmission
         tx_data_byte7            : in  std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0); -- Byte7 for transmission
         tx_pin_out               : out std_logic -- CAN tx pin
       );
end can_transmit;

architecture Behavioral of can_transmit is
   type t_state is (hold,sof,id,rtr,ide,rsrvd,dlc,data,crc,crc_del,
                    ack,ack_del,eof);
   signal tx_state : t_state;
   signal tx_crc : std_logic_vector(CAN_CRC_BIT_LENGTH-1 downto 0);
begin

-- Transmission state machine implementation
   can_send: process(clk)

      variable bit_stuff_count: integer range 0 to 6; -- Counter to implement bit stuffing
      variable bit_to_stuff: std_logic; -- Flag to enable the addition of a stuff bit
      variable bit_count: integer range 0 to 15; -- Counter to send specific bits from CAN fields
      variable data_byte_to_sent: integer range 0 to 8; -- Counter to send specific CAn data byte
      variable data_dlc: integer range 0 to 8; -- Number of bytes requested to send
      variable bit_to_send: std_logic; -- Value of the bit that will be sent
      variable last_bit_sent: std_logic; -- Value of the bit in the previous iteration
      variable next_bit_to_send: std_logic; -- Bit level to send as stuff bit
      variable tx_end_var : std_logic; -- Flag to indicate correct message transmission

   begin
      if(clk'event and clk = '1') then
         -- Force state machine to work at CAN bus baud rate, other nodes will synchronise to it
         if(baud_rate_clk='1')then
            -- Only execute state machine if there is no bit to stuff and transmission is not in hold

            --- Below code implement: [REQ_SW_13], [REQ_SW_16] ---
            if (bit_to_stuff = '0' and tx_hold = '0') then
```

```
                                                                    
                --- Below code implement: [REQ_SW_10] ---
                case tx_state is
                   when sof =>
                      bit_to_send := '0'; -- Start of Frame has to be always recessive (1)

                      --- Below code implement: [REQ_SW_12] ---
                      bit_count := CAN_ID_BIT_LENGTH; -- Reset the bit count sent to id state
                      --- Upper code implement: [REQ_SW_12] ---

                      tx_state <= id; -- Go directly to next state
                      tx_crc <= CRC_CALCULATION(bit_to_send, tx_crc); --- Implement: [REQ_SW_18]
                   when id =>
                      -- CAN ID sending process
                      bit_count := bit_count - 1; -- Decrease counter
                      bit_to_send := tx_id(bit_count); -- Load bit for transmission
                      if(bit_count = 0) then -- Move to RTR state if counter reaches 0
                         bit_count := CAN_RTR_BIT_LENGTH;
                         tx_state <= rtr;
                      end if;
                      tx_crc <= CRC_CALCULATION(bit_to_send, tx_crc); --- Implement: [REQ_SW_18]
                   when rtr =>
                      bit_to_send := '0'; -- Always dominant, no remote frame request
                      tx_state <= ide; -- Go directly to next state
                      tx_crc <= CRC_CALCULATION(bit_to_send, tx_crc); --- Implement: [REQ_SW_18]
                    when ide =>
                       bit_to_send := '0'; --Always dominant for non-extended CAN frame
                       tx_state <= rsrvd; -- Go directly to next state
                       tx_crc <= CRC_CALCULATION(bit_to_send, tx_crc); --- Implement: [REQ_SW_18]
                    when rsrvd =>
                       bit_to_send := '0';  -- Always send dominant bit
                       bit_count := CAN_DLC_BIT_LENGTH;
                       tx_state <= dlc;
                       tx_crc <= CRC_CALCULATION(bit_to_send, tx_crc); --- Implement: [REQ_SW_18]
                    when dlc =>
                       bit_count := bit_count - 1;  -- Decrease counter
                       bit_to_send := tx_dlc(bit_count);
                       if(bit_count = 0) then -- Move to next state if counter reaches 0
                          if (data_dlc = 0) then -- If dlc is equal to 0, move to CRC state
                             bit_count := CAN_CRC_BIT_LENGTH;
                             tx_state <= crc;
                          else -- If dlc is higher than 0, move to data state
                             bit_count := CAN_DATA_BIT_LENGTH;
                             data_byte_to_sent := 0; -- Start sending byte0
                             tx_state <= data;
                          end if;
                       end if;
                       tx_crc <= CRC_CALCULATION(bit_to_send, tx_crc); --- Implement: [REQ_SW_18]
                 when data =>
                    bit_count := bit_count - 1; -- Decrease counter
                    case data_byte_to_sent is -- Bit for transmission depends on bit and byte counters
                       when 0 => bit_to_send := tx_data_byte0(bit_count);
                       when 1 => bit_to_send := tx_data_byte1(bit_count);
                       when 2 => bit_to_send := tx_data_byte2(bit_count);
                       when 3 => bit_to_send := tx_data_byte3(bit_count);
                       when 4 => bit_to_send := tx_data_byte4(bit_count);
                       when 5 => bit_to_send := tx_data_byte5(bit_count);
                       when 6 => bit_to_send := tx_data_byte6(bit_count);
                       when others => bit_to_send := tx_data_byte7(bit_count);
                    end case;
                    if (bit_count = 0) then -- If bit counter is equal 0, move to next byte or state
                       data_byte_to_sent := data_byte_to_sent + 1; -- Increase byte to send
                       if (data_byte_to_sent = data_dlc) then -- If byte is equal to dlc, move to CRC state
                          bit_count := CAN_CRC_BIT_LENGTH;
                          tx_state <= crc;
                       else -- If byte to send is less than dlc, send another byte
                          bit_count := CAN_DATA_BIT_LENGTH;
                       end if;
                    end if;
                    tx_crc <= CRC_CALCULATION(bit_to_send, tx_crc); --- Implement: [REQ_SW_18]
```

```
                when crc =>
                   bit_count := bit_count - 1; -- Decrease counter
                   bit_to_send := tx_crc(bit_count);--- Implement: [REQ_SW_19]
                   if(bit_count = 0) then -- If bit count is equal to 0, move to CRC_DEL state
                      bit_count := 0;
                      tx_state <= crc_del;
                   end if;
                when crc_del =>
                   bit_to_send := '1'; -- Always recessive
                   tx_state <= ack; -- Directly move to ACK state
                when ack =>
                   bit_to_send := '1'; --Always recessive, other nodes shall set is as dominant
                   tx_state <= ack_del; -- Directly move to ACK_DEL state
                when ack_del =>
                   bit_to_send := '1'; --Always recessive
                   bit_count := CAN_EOF_BIT_LENGTH;
                   tx_state <= eof; -- Directly move to EOF state
                when eof =>
                   bit_count := bit_count - 1; -- Decrease counter
                   bit_to_send := '1'; -- Always recessive
                   if(bit_count = 0) then -- If bit counter is equal to 0, bus idle
                      tx_state <= hold; -- Move to hold to accept new messages to send

                      --- Below code implement: [REQ_SW_24], [REQ_SW_25] ---
                      tx_end_var := '1'; -- Indicate correct message is transmited
                      --- Upper code implement: [REQ_SW_24], [REQ_SW_25] ---

                       tx_frame_ack_out <= tx_frame_ack_in; -- Copy ACK to output to check message tx reference
                   end if;
                when others => -- Bus in idle state
                   bit_stuff_count := 0; -- Reset bit stuff counter
                   last_bit_sent := '1'; -- Update last bit sent
                   next_bit_to_send := '1'; -- Update nex bit to sent
                   bit_to_send := '1'; -- During idle only recessive bits are allowed
                   if(tx_start = '1') then -- Check if there is a request for tx
                      -- Prepare the controller for transmittion
                      data_dlc := to_integer(unsigned(tx_dlc)); -- Save the number of bytes to send
                      tx_crc <= (others => '0');
                      tx_state <= sof; -- Move to start of frame

                      --- Below code implement: [REQ_SW_24], [REQ_SW_25]  ---
                      tx_end_var := '0'; -- Remove flag indicating message transmited
                      --- Upper code implement: [REQ_SW_24], [REQ_SW_25] ---

                   else
                      tx_end_var := '1'; -- Set flag indicating message is transmitted
                   end if;
             end case;
             --- Upper code implement: [REQ_SW_10] ---

             --- Below code implement: [REQ_SW_13] ---
             -- Implement bit stuffing for al frame fields except crc delimiter,
             -- ack and eof
             if(tx_state = sof or tx_state = id or tx_state = rtr or tx_state = ide or
                tx_state = rsrvd or tx_state = dlc or tx_state = data or
                tx_state = crc)then

                if (bit_to_send = last_bit_sent) then
                   -- If bit to send is equal to last bit sent, increase the
                   -- repeated bit counter and check counter overflow
                   bit_stuff_count := bit_stuff_count + 1;
                   if(bit_stuff_count = 5)then
                      -- If more than 5 consecutive bits of the same value
                      -- are sent, sent the opposite value and reset the count
                      next_bit_to_send := not bit_to_send;
                      bit_to_stuff := '1';
                      bit_stuff_count := 1;
                   end if; -- bit_stuff_count = 5
                else
                   bit_stuff_count := 1;
                   last_bit_sent := bit_to_send;
                end if; -- bit_to_send = last_bit_sent
             end if; -- bit_stuffing algorithm
             --- Upper code implement: [REQ_SW_13] ---
```

```
221
222             elsif(bit_to_stuff = '1')then
223                -- In the last iteration the sequence has stuffed a bit, so this iteration
224                -- is to insert the not sent bit
225                bit_stuff_count := 1;
226                bit_to_stuff := '0';
227                bit_to_send := next_bit_to_send; -- Negate bit_to_send to send original bit
228                last_bit_sent := bit_to_send; -- Save bit_to_send value as the last bit sent
229             --- Upper code implement: [REQ_SW_13] ---
230
231             else -- transmission is on hold by higher priority message being transmitted
232                bit_to_send := '1'; -- Always send recessive bits
233                tx_end_var := '1'; -- Indicate transmission ended
234                bit_to_stuff := '0'; -- Reset bit to stuff enable
235                bit_stuff_count := 0; -- Reset bit to stuff counter
236                last_bit_sent := '1'; -- Update last bit sent
237                tx_crc <= (others => '0'); -- reset CRC calculation
238
239                --- Below code implement: [REQ_SW_17] ---
240                if(tx_state /= hold)then -- If current state is different from holde, retransmit not sent message
241                   tx_state <= sof;
242                end if;
243                --- Upper code implement: [REQ_SW_17] ---
244
245             end if;
246             --- Upper code implement: [REQ_SW_13], [REQ_SW_16] ---
247
248          end if; -- baud_rate = '1'
249          tx_pin_out <= bit_to_send; -- Load CAN tx pin with variable value
250          tx_end <= std_logic(tx_end_var); -- Laod transmission end flag with variable value
251       end if; -- clk'event and clk = '1'
252    end process;
253 end Behavioral;
```

6. Entitat USER_LOGIC.VHD

```
51  -- DO NOT EDIT BELOW THIS LINE ---------------------
52  library ieee;
53  use ieee.std_logic_1164.all;
54  use ieee.std_logic_arith.all;
55  use ieee.numeric_std.all;
56  use ieee.std_logic_unsigned.all;
57
58  library proc_common_v3_00_a;
59  use proc_common_v3_00_a.proc_common_pkg.all;
60
61  -- DO NOT EDIT ABOVE THIS LINE ---------------------
62
63  library axi_can_custom_v1_11_a;
64  use axi_can_custom_v1_11_a.all;
65
66  ------------------------------------------------------------------------------
67  -- Entity section
68  ------------------------------------------------------------------------------
69  -- Definition of Generics:
70  --   C_NUM_REG                     -- Number of software accessible registers
71  --   C_SLV_DWIDTH                  -- Slave interface data bus width
72  --
73  -- Definition of Ports:
74  --   Bus2IP_Clk                    -- Bus to IP clock
75  --   Bus2IP_Resetn                 -- Bus to IP reset
76  --   Bus2IP_Data                   -- Bus to IP data bus
77  --   Bus2IP_BE                     -- Bus to IP byte enables
78  --   Bus2IP_RdCE                   -- Bus to IP read chip enable
79  --   Bus2IP_WrCE                   -- Bus to IP write chip enable
80  --   IP2Bus_Data                   -- IP to Bus data bus
81  --   IP2Bus_RdAck                  -- IP to Bus read transfer acknowledgement
82  --   IP2Bus_WrAck                  -- IP to Bus write transfer acknowledgement
83  --   IP2Bus_Error                  -- IP to Bus error response
84  ------------------------------------------------------------------------------
```

```
 85
 86  entity user_logic is
 87    generic
 88    (
 89      -- ADD USER GENERICS BELOW THIS LINE ---------------
 90      CAN_ID_BIT_LENGTH              : integer              := 11;
 91      CAN_RTR_BIT_LENGTH             : integer              := 1;
 92      CAN_IDE_BIT_LENGTH             : integer              := 1;
 93      CAN_RSRVD_BIT_LENGTH           : integer              := 1;
 94      CAN_DLC_BIT_LENGTH             : integer              := 4;
 95      CAN_DATA_BIT_LENGTH            : integer              := 8;
 96      CAN_CRC_BIT_LENGTH             : integer              := 15;
 97      CAN_EOF_BIT_LENGTH             : integer              := 7;
 98      CAN_MAX_DATA_FRAMES            : integer              := 8;
 99      --BAUD_RATE_PRESCALER_VAL      : integer              := 66; -- 1 Mbps
100      BAUD_RATE_PRESCALER_VAL        : integer              := 133; -- 500 kbps
101      --BAUD_RATE_PRESCALER_VAL      : integer              := 533; -- 125 kbps
102
103      -- ADD USER GENERICS ABOVE THIS LINE ---------------
104
105      -- DO NOT EDIT BELOW THIS LINE ----------------------
106      -- Bus protocol parameters, do not add to or delete
107      C_NUM_REG                      : integer              := 16;
108      C_SLV_DWIDTH                   : integer              := 32
109      -- DO NOT EDIT ABOVE THIS LINE ----------------------
110    );
111    port
112    (
113      -- ADD USER PORTS BELOW THIS LINE ------------------
114      CAN_rx                         : in std_logic;
115      CAN_tx                         : out std_logic;
116      baud_rate_rx                   : out std_logic;
117      baud_rate_tx                   : out std_logic;
118      -- ADD USER PORTS ABOVE THIS LINE ------------------
119
120      -- DO NOT EDIT BELOW THIS LINE ----------------------
121      -- Bus protocol ports, do not add to or delete
122      Bus2IP_Clk                     : in  std_logic;
123      Bus2IP_Resetn                  : in  std_logic;
124      Bus2IP_Data                    : in  std_logic_vector(C_SLV_DWIDTH-1 downto 0);
125      Bus2IP_BE                      : in  std_logic_vector(C_SLV_DWIDTH/8-1 downto 0);
126      Bus2IP_RdCE                    : in  std_logic_vector(C_NUM_REG-1 downto 0);
127      Bus2IP_WrCE                    : in  std_logic_vector(C_NUM_REG-1 downto 0);
128      IP2Bus_Data                    : out std_logic_vector(C_SLV_DWIDTH-1 downto 0);
129      IP2Bus_RdAck                   : out std_logic;
130      IP2Bus_WrAck                   : out std_logic;
131      IP2Bus_Error                   : out std_logic
132      -- DO NOT EDIT ABOVE THIS LINE ----------------------
133    );
134
135    attribute MAX_FANOUT : string;
136    attribute SIGIS : string;
137
138    attribute SIGIS of Bus2IP_Clk    : signal is "CLK";
139    attribute SIGIS of Bus2IP_Resetn : signal is "RST";
140
141  end entity user_logic;
142
143  ------------------------------------------------------------------------------
144  -- Architecture section
145  ------------------------------------------------------------------------------
```

```
146
147 architecture IMP of user_logic is
148
149   --USER signal declarations added here, as needed for user logic
150   -- can_transmit signals
151    signal tx_start               : std_logic := '0'; -- slv_reg0(0) - signal from sdk
152    signal tx_end                 : std_logic := '0'; -- slv_reg0(1) - signal to sdk
153    signal tx_reset               : std_logic := '0'; -- slv_reg0(2) - signal from sdk
154    signal tx_reset_ack           : std_logic := '0'; -- slv_reg0(3) - signal to sdk
155    signal tx_hold                : std_logic := '0'; -- slv_reg0(4) - internal signal (from receive to transmit)
156    signal tx_frame_ack_in        : std_logic_vector(C_SLV_DWIDTH-1 downto 0) := (others => '0');
157    signal tx_frame_ack_out       : std_logic_vector(C_SLV_DWIDTH-1 downto 0) := (others => '0');
158    signal tx_frame_ack           : std_logic_vector(C_SLV_DWIDTH-1 downto 0) := (others => '0');
159    signal tx_id                  : std_logic_vector(CAN_ID_BIT_LENGTH-1 downto 0) := (others => '0');
160    signal tx_dlc                 : std_logic_vector(CAN_DLC_BIT_LENGTH-1 downto 0) := (others => '0');
161    signal tx_data_byte0          : std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0) := (others => '0');
162    signal tx_data_byte1          : std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0) := (others => '0');
163    signal tx_data_byte2          : std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0) := (others => '0');
164    signal tx_data_byte3          : std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0) := (others => '0');
165    signal tx_data_byte4          : std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0) := (others => '0');
166    signal tx_data_byte5          : std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0);
167    signal tx_data_byte6          : std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0);
168    signal tx_data_byte7          : std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0);
169
170    -- can_receive_mbox0 signals
171    signal rx_mbox0_frm_rcvd_rst  : std_logic := '0'; -- Flag to reset the frame received flag
172    signal rx_mbox0_frm_rcvd      : std_logic := '0'; -- Flag that indicates frame received
173    signal rx_mbox0_result        : std_logic_vector(3 downto 0) := (others => '0'); -- Provides feedback about frame integrity
174    signal rx_mbox0_id            : std_logic_vector(CAN_ID_BIT_LENGTH-1 downto 0) := (others => '0'); -- Frame ID
175    signal rx_mbox0_dlc           : std_logic_vector(CAN_DLC_BIT_LENGTH-1 downto 0) := (others => '0'); -- Frame DLC
176    signal rx_mbox0_data_byte0    : std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0) := (others => '0');-- Frame byte 0
177    signal rx_mbox0_data_byte1    : std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0) := (others => '0'); -- Frame byte 1
178    signal rx_mbox0_data_byte2    : std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0) := (others => '0'); -- Frame byte 2
179    signal rx_mbox0_data_byte3    : std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0) := (others => '0'); -- Frame byte 3
180    signal rx_mbox0_data_byte4    : std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0) := (others => '0'); -- Frame byte 4
181    signal rx_mbox0_data_byte5    : std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0) := (others => '0'); -- Frame byte 5
182    signal rx_mbox0_data_byte6    : std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0) := (others => '0'); -- Frame byte 6
183    signal rx_mbox0_data_byte7    : std_logic_vector(CAN_DATA_BIT_LENGTH-1 downto 0) := (others => '0'); -- Frame byte 7
184    signal rx_mbox0_crc           : std_logic_vector(CAN_CRC_BIT_LENGTH-1 downto 0) := (others => '0'); -- Frame CRC
185
186    -- axi_can_custom_ul_interface signals
187    signal baud_rate_clk          : std_logic := '0';
188    signal tx_start_input         : std_logic := '0';
189    signal tx_reset_input         : std_logic := '0';
190    signal CAN_tx_rx_mbox0        : std_logic := '0';
191    signal CAN_tx_tx              : std_logic := '0';
192
193   ------------------------------------------
194   -- Signals for user logic slave model s/w accessible register example
195   ------------------------------------------
196   signal slv_reg0                     : std_logic_vector(C_SLV_DWIDTH-1 downto 0);
197   signal slv_reg1                     : std_logic_vector(C_SLV_DWIDTH-1 downto 0);
198   signal slv_reg2                     : std_logic_vector(C_SLV_DWIDTH-1 downto 0);
199   signal slv_reg3                     : std_logic_vector(C_SLV_DWIDTH-1 downto 0);
200   signal slv_reg4                     : std_logic_vector(C_SLV_DWIDTH-1 downto 0);
201   signal slv_reg5                     : std_logic_vector(C_SLV_DWIDTH-1 downto 0);
202   signal slv_reg6                     : std_logic_vector(C_SLV_DWIDTH-1 downto 0);
203   signal slv_reg7                     : std_logic_vector(C_SLV_DWIDTH-1 downto 0);
204   signal slv_reg8                     : std_logic_vector(C_SLV_DWIDTH-1 downto 0);
205   signal slv_reg9                     : std_logic_vector(C_SLV_DWIDTH-1 downto 0);
206   signal slv_reg10                    : std_logic_vector(C_SLV_DWIDTH-1 downto 0);
207   signal slv_reg11                    : std_logic_vector(C_SLV_DWIDTH-1 downto 0);
208   signal slv_reg12                    : std_logic_vector(C_SLV_DWIDTH-1 downto 0);
209   signal slv_reg13                    : std_logic_vector(C_SLV_DWIDTH-1 downto 0);
210   signal slv_reg14                    : std_logic_vector(C_SLV_DWIDTH-1 downto 0);
211   signal slv_reg15                    : std_logic_vector(C_SLV_DWIDTH-1 downto 0);
212   signal slv_reg_write_sel            : std_logic_vector(15 downto 0);
213   signal slv_reg_read_sel             : std_logic_vector(15 downto 0);
214   signal slv_ip2bus_data              : std_logic_vector(C_SLV_DWIDTH-1 downto 0);
215   signal slv_read_ack                 : std_logic;
216   signal slv_write_ack                : std_logic;
217
218 begin
219
220   --USER logic implementation added here
221    -- CAN baud rate clock for communication data transmition and reception
222    baud_rate_clk_divider: entity axi_can_custom_v1_11_a.divider
223       generic map(fact => BAUD_RATE_PRESCALER_VAL)
224       port map (clk_in => Bus2IP_Clk,
225                 rst => '0',
226                 clk_out => baud_rate_clk);
```

```
227
228     baud_rate_tx <= '1';
229
230     can_transmit: entity axi_can_custom_v1_11_a.can_transmit
231     port map (
232        clk => Bus2IP_Clk, -- IN: system clk
233        baud_rate_clk => baud_rate_clk, -- IN: CAN baud rate clock
234
235        tx_start => tx_start,
236        tx_hold => tx_hold,
237        --tx_hold => '0',
238        tx_end => tx_end,
239        tx_frame_ack_in => tx_frame_ack_in,
240        tx_frame_ack_out => tx_frame_ack_out,
241        tx_id => tx_id,
242        tx_dlc => tx_dlc,
243        tx_data_byte0 => tx_data_byte0,
244        tx_data_byte1 => tx_data_byte1,
245        tx_data_byte2 => tx_data_byte2,
246        tx_data_byte3 => tx_data_byte3,
247        tx_data_byte4 => tx_data_byte4,
248        tx_data_byte5 => tx_data_byte5,
249        tx_data_byte6 => tx_data_byte6,
250        tx_data_byte7 => tx_data_byte7,
251
252        tx_pin_out => CAN_tx_tx
253     );
254
255     can_receive_mbox0: entity axi_can_custom_v1_11_a.can_receive
256     generic map(BAUD_RATE_PRESCALER_VAL => BAUD_RATE_PRESCALER_VAL)
257     port map (
258        clk => Bus2IP_Clk, -- IN: system clk
259        baud_rate_clk_out => baud_rate_rx, -- IN: CAN baud rate clock
260
261        rx_frm_rcvd_rst   => rx_mbox0_frm_rcvd_rst, -- Flag to reset the frame received flag
262        rx_frm_rcvd       => rx_mbox0_frm_rcvd, -- Flag that indicates frame received
263        rx_tx_hold        => tx_hold,
264        rx_result         => rx_mbox0_result, -- Provides feedback about frame integrity
265        tx_id             => tx_id,
266        rx_id             => rx_mbox0_id, -- Frame ID
267        rx_dlc            => rx_mbox0_dlc, -- Frame DLC
268        rx_data_byte0     => rx_mbox0_data_byte0, -- Frame byte 0
269        rx_data_byte1     => rx_mbox0_data_byte1, -- Frame byte 1
270        rx_data_byte2     => rx_mbox0_data_byte2, -- Frame byte 2
271        rx_data_byte3     => rx_mbox0_data_byte3, -- Frame byte 3
272        rx_data_byte4     => rx_mbox0_data_byte4, -- Frame byte 4
273        rx_data_byte5     => rx_mbox0_data_byte5, -- Frame byte 5
274        rx_data_byte6     => rx_mbox0_data_byte6, -- Frame byte 6
275        rx_data_byte7     => rx_mbox0_data_byte7, -- Frame byte 7
276        rx_crc            => rx_mbox0_crc, -- Frame CRC
277
278        rx_pin_in         => CAN_rx, -- CAN rx pin
279        rx_tx_pin_out     => CAN_tx_rx_mbox0
280     );
281
282  ------------------------------------------------------------------------------
283  -- Interfaces between Microblaze CAN driver and VHDL CAN driver
284  ------------------------------------------------------------------------------
285
286  CAN_tx <= CAN_tx_tx and CAN_tx_rx_mbox0;
287  tx_start_input <= slv_reg0(0);
288  tx_reset_input <= slv_reg0(2);
289  rx_mbox0_frm_rcvd_rst <= slv_reg4(0); -- rx_frm_rcvd_rst
290
291  axi_can_custom_ul_interface: process(Bus2IP_Clk)
292     variable count : integer range 0 to 2500000;
293     variable tx_reset_ack_var, tx_start_var, tx_disable: std_logic;
294
295  begin
296     --- Below code implement: [REQ_SW_21]  ---
297     if(Bus2IP_Clk'event and Bus2IP_Clk = '1') then
298        -- Sync with CAN baud rate
299        if(baud_rate_clk = '1')then
300
301           if(tx_reset_input = '1')then
302              tx_start_var := '0';
303              tx_disable := '0';
304              tx_reset_ack_var := '1';
305           else
306              tx_reset_ack_var := '0';
307           end if; -- tx_reset_input = '1'
```

```
308
309
310             if(tx_start_input = '1') then
311                -- CAN message transmission started or ongoing and flag reset not
312                -- enabled, increase counter
313
314                if(tx_start_var = '0')then
315                   -- Prevent rentry when tx_start_input is 0, and command to start has been reset
316                   if(tx_disable = '0')then
317                      -- Copy message to send information to can_controller signals (only once)
318                      tx_id <= slv_reg1(CAN_ID_BIT_LENGTH-1 downto 0);
319                      tx_dlc <= slv_reg1(CAN_DLC_BIT_LENGTH+CAN_ID_BIT_LENGTH-1 downto CAN_ID_BIT_LENGTH);
320                      tx_data_byte0 <= slv_reg2(CAN_DATA_BIT_LENGTH-1 downto 0);
321                      tx_data_byte1 <= slv_reg2(CAN_DATA_BIT_LENGTH*2-1 downto CAN_DATA_BIT_LENGTH);
322                      tx_data_byte2 <= slv_reg2(CAN_DATA_BIT_LENGTH*3-1 downto CAN_DATA_BIT_LENGTH*2);
323                      tx_data_byte3 <= slv_reg2(CAN_DATA_BIT_LENGTH*4-1 downto CAN_DATA_BIT_LENGTH*3);
324                      tx_data_byte4 <= slv_reg3(CAN_DATA_BIT_LENGTH-1 downto 0);
325                      tx_data_byte5 <= slv_reg3(CAN_DATA_BIT_LENGTH*2-1 downto CAN_DATA_BIT_LENGTH);
326                      tx_data_byte6 <= slv_reg3(CAN_DATA_BIT_LENGTH*3-1 downto CAN_DATA_BIT_LENGTH*2);
327                      tx_data_byte7 <= slv_reg3(CAN_DATA_BIT_LENGTH*4-1 downto CAN_DATA_BIT_LENGTH*3);
328
329                      tx_frame_ack_in <= slv_reg15;
330                      tx_start_var := '1';
331                   end if; -- tx_disable = '0'
332                else
333                   tx_disable := '1';
334                   tx_start_var := '0';
335                end if; -- tx_start_var = '0'
337             else
338                tx_start_var := '0';
339                tx_disable := '0';
340                count := 0;
341             end if;
342             tx_start <= std_logic(tx_start_var);
343             tx_reset_ack <= std_logic(tx_reset_ack_var);
344          end if; -- baud_rate = '0'
345       end if;
346       --- Upper code implement: [REQ_SW_21]  ---
347    end process axi_can_custom_ul_interface;
348   ------------------------------------------
349   -- Example code to read/write user logic slave model s/w accessible registers
350   --
351   -- Note:
352   -- The example code presented here is to show you one way of reading/writing
353   -- software accessible registers implemented in the user logic slave model.
354   -- Each bit of the Bus2IP_WrCE/Bus2IP_RdCE signals is configured to correspond
355   -- to one software accessible register by the top level template. For example,
356   -- if you have four 32 bit software accessible registers in the user logic,
357   -- you are basically operating on the following memory mapped registers:
358   --
359   --    Bus2IP_WrCE/Bus2IP_RdCE   Memory Mapped Register
360   --                      "1000"   C_BASEADDR + 0x0
361   --                      "0100"   C_BASEADDR + 0x4
362   --                      "0010"   C_BASEADDR + 0x8
363   --                      "0001"   C_BASEADDR + 0xC
364   --
365   ------------------------------------------
366   slv_reg_write_sel <= Bus2IP_WrCE(15 downto 0);
367   slv_reg_read_sel  <= Bus2IP_RdCE(15 downto 0);
368   slv_write_ack     <= Bus2IP_WrCE(0) or Bus2IP_WrCE(1) or Bus2IP_WrCE(2) or Bus2IP_WrCE(3)
369                        or Bus2IP_WrCE(4) or Bus2IP_WrCE(5) or Bus2IP_WrCE(6) or Bus2IP_WrCE(7) or Bus2IP_WrCE(8)
370                        or Bus2IP_WrCE(9) or Bus2IP_WrCE(10) or Bus2IP_WrCE(11) or Bus2IP_WrCE(12) or Bus2IP_WrCE(13)
371                        or Bus2IP_WrCE(14) or Bus2IP_WrCE(15);
372   slv_read_ack      <= Bus2IP_RdCE(0) or Bus2IP_RdCE(1) or Bus2IP_RdCE(2) or Bus2IP_RdCE(3)
373                        or Bus2IP_RdCE(4) or Bus2IP_RdCE(5) or Bus2IP_RdCE(6) or Bus2IP_RdCE(7) or Bus2IP_RdCE(8)
374                        or Bus2IP_RdCE(9) or Bus2IP_RdCE(10) or Bus2IP_RdCE(11) or Bus2IP_RdCE(12) or Bus2IP_RdCE(13)
375                        or Bus2IP_RdCE(14) or Bus2IP_RdCE(15);
376
377   -- implement slave model software accessible register(s)
378   SLAVE_REG_WRITE_PROC : process( Bus2IP_Clk ) is
379   begin
380
381     if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
382       if Bus2IP_Resetn = '0' then
383         slv_reg0 <= (others => '0');
384         slv_reg1 <= (others => '0');
385         slv_reg2 <= (others => '0');
386         slv_reg3 <= (others => '0');
387         slv_reg4 <= (others => '0');
388         slv_reg5 <= (others => '0');
389         slv_reg6 <= (others => '0');
390         slv_reg7 <= (others => '0');
391         slv_reg8 <= (others => '0');
392         slv_reg9 <= (others => '0');
393         slv_reg10 <= (others => '0');
394         slv_reg11 <= (others => '0');
395         slv_reg12 <= (others => '0');
396         slv_reg13 <= (others => '0');
397         slv_reg14 <= (others => '0');
398         slv_reg15 <= (others => '0');
399       else
```

```
400         case slv_reg_write_sel is
401           when "1000000000000000" =>
402             for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
403               if ( Bus2IP_BE(byte_index) = '1' ) then
404                 slv_reg0(byte_index*8+7 downto byte_index*8) <= Bus2IP_Data(byte_index*8+7 downto byte_index*8);
405               end if;
406             end loop;
407           when "0100000000000000" =>
408             for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
409               if ( Bus2IP_BE(byte_index) = '1' ) then
410                 slv_reg1(byte_index*8+7 downto byte_index*8) <= Bus2IP_Data(byte_index*8+7 downto byte_index*8);
411               end if;
412             end loop;
413           when "0010000000000000" =>
414             for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
415               if ( Bus2IP_BE(byte_index) = '1' ) then
416                 slv_reg2(byte_index*8+7 downto byte_index*8) <= Bus2IP_Data(byte_index*8+7 downto byte_index*8);
417               end if;
418             end loop;
419           when "0001000000000000" =>
420             for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
421               if ( Bus2IP_BE(byte_index) = '1' ) then
422                 slv_reg3(byte_index*8+7 downto byte_index*8) <= Bus2IP_Data(byte_index*8+7 downto byte_index*8);
423               end if;
424             end loop;
425           when "0000100000000000" =>
426             for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
427               if ( Bus2IP_BE(byte_index) = '1' ) then
428                 slv_reg4(byte_index*8+7 downto byte_index*8) <= Bus2IP_Data(byte_index*8+7 downto byte_index*8);
429               end if;
430             end loop;
431           when "0000010000000000" =>
432             for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
433               if ( Bus2IP_BE(byte_index) = '1' ) then
434                 slv_reg5(byte_index*8+7 downto byte_index*8) <= Bus2IP_Data(byte_index*8+7 downto byte_index*8);
435               end if;
436             end loop;
437           when "0000001000000000" =>
438             for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
439               if ( Bus2IP_BE(byte_index) = '1' ) then
440                 slv_reg6(byte_index*8+7 downto byte_index*8) <= Bus2IP_Data(byte_index*8+7 downto byte_index*8);
441               end if;
442             end loop;
443           when "0000000100000000" =>
444             for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
445               if ( Bus2IP_BE(byte_index) = '1' ) then
446                 slv_reg7(byte_index*8+7 downto byte_index*8) <= Bus2IP_Data(byte_index*8+7 downto byte_index*8);
447               end if;
448             end loop;
449           when "0000000010000000" =>
450             for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
451               if ( Bus2IP_BE(byte_index) = '1' ) then
452                 slv_reg8(byte_index*8+7 downto byte_index*8) <= Bus2IP_Data(byte_index*8+7 downto byte_index*8);
453               end if;
454             end loop;
455           when "0000000001000000" =>
456             for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
457               if ( Bus2IP_BE(byte_index) = '1' ) then
458                 slv_reg9(byte_index*8+7 downto byte_index*8) <= Bus2IP_Data(byte_index*8+7 downto byte_index*8);
459               end if;
460             end loop;
461           when "0000000000100000" =>
462             for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
463               if ( Bus2IP_BE(byte_index) = '1' ) then
464                 slv_reg10(byte_index*8+7 downto byte_index*8) <= Bus2IP_Data(byte_index*8+7 downto byte_index*8);
465               end if;
466             end loop;
467           when "0000000000010000" =>
468             for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
469               if ( Bus2IP_BE(byte_index) = '1' ) then
470                 slv_reg11(byte_index*8+7 downto byte_index*8) <= Bus2IP_Data(byte_index*8+7 downto byte_index*8);
471               end if;
472             end loop;
472              end loop;
473            when "0000000000001000" =>
474              for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
475                if ( Bus2IP_BE(byte_index) = '1' ) then
476                  slv_reg12(byte_index*8+7 downto byte_index*8) <= Bus2IP_Data(byte_index*8+7 downto byte_index*8);
477                end if;
478              end loop;
479            when "0000000000000100" =>
480              for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
481                if ( Bus2IP_BE(byte_index) = '1' ) then
482                  slv_reg13(byte_index*8+7 downto byte_index*8) <= Bus2IP_Data(byte_index*8+7 downto byte_index*8);
483                end if;
484              end loop;
485            when "0000000000000010" =>
486              for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
487                if ( Bus2IP_BE(byte_index) = '1' ) then
488                  slv_reg14(byte_index*8+7 downto byte_index*8) <= Bus2IP_Data(byte_index*8+7 downto byte_index*8);
489                end if;
490              end loop;
491            when "0000000000000001" =>
492              for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
493                if ( Bus2IP_BE(byte_index) = '1' ) then
494                  slv_reg15(byte_index*8+7 downto byte_index*8) <= Bus2IP_Data(byte_index*8+7 downto byte_index*8);
495                end if;
496              end loop;
497            when others => null;
498         end case;
```

```
499         end if;
500       end if;
501
502     end process SLAVE_REG_WRITE_PROC;
503
504     -- implement slave model software accessible register(s) read mux
505     SLAVE_REG_READ_PROC : process( slv_reg_read_sel, slv_reg0, slv_reg1, slv_reg2,
506                                    slv_reg3, slv_reg4, slv_reg5, slv_reg6, slv_reg7,
507                                    slv_reg8, slv_reg9, slv_reg10, slv_reg11, slv_reg12,
508                                    slv_reg13, slv_reg14, slv_reg15 ) is
509     begin
510
511       case slv_reg_read_sel is
512         when "1000000000000000" => slv_ip2bus_data(0) <= slv_reg0(0); -- tx_start
513                                    slv_ip2bus_data(1) <= tx_end;
514                                    slv_ip2bus_data(2) <= slv_reg0(2); -- tx_reset
515                                    slv_ip2bus_data(3) <= tx_reset_ack;
516                                    slv_ip2bus_data(4) <= tx_hold;
517                                    slv_ip2bus_data(C_SLV_DWIDTH-1 downto 5) <= slv_reg0(C_SLV_DWIDTH-1 downto 5);
518         when "0100000000000000" => slv_ip2bus_data <= slv_reg1; -- tx_id_dlc
519         when "0010000000000000" => slv_ip2bus_data <= slv_reg2; -- tx_data_byte0-3
520         when "0001000000000000" => slv_ip2bus_data <= slv_reg3; -- tx_data_byte4-7
521         when "0000100000000000" => slv_ip2bus_data(0) <= slv_reg4(0);
522                                    slv_ip2bus_data(1) <= rx_mbox0_frm_rcvd;
523                                    slv_ip2bus_data(7 downto 2) <= slv_reg4(7 downto 2);
524                                    slv_ip2bus_data(11 downto 8) <= rx_mbox0_result;
525                                    slv_ip2bus_data(C_SLV_DWIDTH-1 downto 11) <= (others => '0');
526         when "0000010000000000" => slv_ip2bus_data(CAN_ID_BIT_LENGTH-1 downto 0) <= rx_mbox0_id; -- slv_reg5
527                                    slv_ip2bus_data(CAN_DLC_BIT_LENGTH+CAN_ID_BIT_LENGTH-1 downto CAN_ID_BIT_LENGTH) <= rx_mbox0_dlc;
528                                    slv_ip2bus_data(C_SLV_DWIDTH-1 downto CAN_ID_BIT_LENGTH+CAN_ID_BIT_LENGTH) <= (others => '0');
529         when "0000001000000000" => slv_ip2bus_data (C_SLV_DWIDTH-1 downto 0)<= rx_mbox0_data_byte3 &
530                                                                                rx_mbox0_data_byte2 &
531                                                                                rx_mbox0_data_byte1 &
532                                                                                rx_mbox0_data_byte0;
533         when "0000000100000000" => slv_ip2bus_data (C_SLV_DWIDTH-1 downto 0)<= rx_mbox0_data_byte7 &
534                                                                                rx_mbox0_data_byte6 &
535                                                                                rx_mbox0_data_byte5 &
536                                                                                rx_mbox0_data_byte4;
537         when "0000000010000000" => slv_ip2bus_data <= slv_reg8;
538         when "0000000001000000" => slv_ip2bus_data <= slv_reg9;
539         when "0000000000100000" => slv_ip2bus_data <= slv_reg10;
540         when "0000000000010000" => slv_ip2bus_data <= slv_reg11;
541         when "0000000000001000" => slv_ip2bus_data <= slv_reg12;
542         when "0000000000000100" => slv_ip2bus_data <= slv_reg13;
543         when "0000000000000010" => slv_ip2bus_data <= tx_frame_ack_out; --slv_reg14
544         when "0000000000000001" => slv_ip2bus_data <= slv_reg15;
545         when others => slv_ip2bus_data <= (others => '0');
546       end case;
547
548     end process SLAVE_REG_READ_PROC;
549
550     ---------------------------------------
551     -- Example code to drive IP to Bus signals
552     ---------------------------------------
553     IP2Bus_Data  <= slv_ip2bus_data when slv_read_ack = '1' else
554                     (others => '0');
555
556     IP2Bus_WrAck <= slv_write_ack;
557     IP2Bus_RdAck <= slv_read_ack;
558     IP2Bus_Error <= '0';
559
560   end IMP;
```

Annex 4 – Llibreries de *driver*s en llenguatge C

1. Llibreria axi_can_*driver*

```
/*
 * axi_can_driver.h
 *
 *  Created on: 22/01/2017
 *      Author: carlos
 */

#ifndef AXI_CAN_DRIVER_H_
#define AXI_CAN_DRIVER_H_

#include "xparameters.h"
#include "xstatus.h"

#define XPAR_AXI_CAN_CUSTOM_OFFSET      (0x04)

#define CAN_TX_OK                       (0)
#define CAN_ERR_TX_BUSY                 (1500L)
#define CAN_ERR_RST_ACK_FLG_NOT_RMVD    (1501L)
#define CAN_ERR_RST_FLG_NOT_ACK         (1502L)
#define CAN_RX_EMPTY                    (1503L)
#define CAN_ERR_RX_FLAG_NOT_REMOVED     (1504L)
/* CAN RX STATUS FLAGS */
#define CAN_RX_NO_FLAG                  (0)
#define CAN_RX_FRM_RCVD_RST             (1<<0)
#define CAN_RX_FRM_RCVD                 (1<<1)

#define TX_ID_DLC_ADDR                  SLV_REG1_ADDR
#define TX_DATA_BYTE_3_0_ADDR           SLV_REG2_ADDR
#define TX_DATA_BYTE_7_4_ADDR           SLV_REG3_ADDR

#define RX_ID_DLC_ADDR                  SLV_REG5_ADDR
#define RX_DATA_BYTE_3_0_ADDR           SLV_REG6_ADDR
#define RX_DATA_BYTE_7_4_ADDR           SLV_REG7_ADDR
#define TX_CTRL_REG_ADDR                SLV_REG0_ADDR
#define RX_CTRL_REG_ADDR                SLV_REG4_ADDR
#define CAN_FRAME_ACK_GET               SLV_REG14_ADDR
#define CAN_FRAME_ACK_SET               SLV_REG15_ADDR

#define SLV_REG0_ADDR                   (XPAR_AXI_CAN_CUSTOM_0_BASEADDR + 0*XPAR_AXI_CAN_CUSTOM_OFFSET)
#define SLV_REG1_ADDR                   (XPAR_AXI_CAN_CUSTOM_0_BASEADDR + 1*XPAR_AXI_CAN_CUSTOM_OFFSET)
#define SLV_REG2_ADDR                   (XPAR_AXI_CAN_CUSTOM_0_BASEADDR + 2*XPAR_AXI_CAN_CUSTOM_OFFSET)
#define SLV_REG3_ADDR                   (XPAR_AXI_CAN_CUSTOM_0_BASEADDR + 3*XPAR_AXI_CAN_CUSTOM_OFFSET)
#define SLV_REG4_ADDR                   (XPAR_AXI_CAN_CUSTOM_0_BASEADDR + 4*XPAR_AXI_CAN_CUSTOM_OFFSET)
#define SLV_REG5_ADDR                   (XPAR_AXI_CAN_CUSTOM_0_BASEADDR + 5*XPAR_AXI_CAN_CUSTOM_OFFSET)
#define SLV_REG6_ADDR                   (XPAR_AXI_CAN_CUSTOM_0_BASEADDR + 6*XPAR_AXI_CAN_CUSTOM_OFFSET)
#define SLV_REG7_ADDR                   (XPAR_AXI_CAN_CUSTOM_0_BASEADDR + 7*XPAR_AXI_CAN_CUSTOM_OFFSET)
#define SLV_REG8_ADDR                   (XPAR_AXI_CAN_CUSTOM_0_BASEADDR + 8*XPAR_AXI_CAN_CUSTOM_OFFSET)
#define SLV_REG9_ADDR                   (XPAR_AXI_CAN_CUSTOM_0_BASEADDR + 9*XPAR_AXI_CAN_CUSTOM_OFFSET)
#define SLV_REG10_ADDR                  (XPAR_AXI_CAN_CUSTOM_0_BASEADDR + 10*XPAR_AXI_CAN_CUSTOM_OFFSET)
#define SLV_REG11_ADDR                  (XPAR_AXI_CAN_CUSTOM_0_BASEADDR + 11*XPAR_AXI_CAN_CUSTOM_OFFSET)
#define SLV_REG12_ADDR                  (XPAR_AXI_CAN_CUSTOM_0_BASEADDR + 12*XPAR_AXI_CAN_CUSTOM_OFFSET)
#define SLV_REG13_ADDR                  (XPAR_AXI_CAN_CUSTOM_0_BASEADDR + 13*XPAR_AXI_CAN_CUSTOM_OFFSET)
#define SLV_REG14_ADDR                  (XPAR_AXI_CAN_CUSTOM_0_BASEADDR + 14*XPAR_AXI_CAN_CUSTOM_OFFSET)
#define SLV_REG15_ADDR                  (XPAR_AXI_CAN_CUSTOM_0_BASEADDR + 15*XPAR_AXI_CAN_CUSTOM_OFFSET)
u32 can_status();
int can_configuration(int can_interface);
int can_send(int can_interface, u16 id, u8 dlc, u32 *data, u32 ack);
int can_receive(int can_interface, u16 mbox, u16 *id, u8 *dlc, u8 *data);

#endif /* AXI_CAN_DRIVER_H_ */
```

Figura 56. axi_can_*driver*.h

```c
/*
 * axi_can_driver.c
 *
 *  Created on: 22/01/2017
 *      Author: carlos
 */

#include "axi_can_driver.h"
#include "xstatus.h"

#define AXI_CAN_DEBUG 1


#include "xil_exception.h"
#include "xbasic_types.h"
#include <stdio.h>

/** This function is used to configure and start the CAN interface correctly.
 *
 *  @param  can_conf is used to identify the CAN interface
 *
 *  @return XST_SUCCESS if the Test is successful, otherwise specific error.
 ******************************************************************************/
int can_configuration(int can_conf){

    /* Create pointers for control register tx and rx */
    u32 *tx_ctrl_reg = (u32 *)TX_CTRL_REG_ADDR;
    u32 *rx_ctrl_reg = (u32 *)RX_CTRL_REG_ADDR;

    /* Set reset flags on overriding any present flag */
    *rx_ctrl_reg = CAN_RX_FRM_RCVD_RST;
    *tx_ctrl_reg = CAN_RST_FLAG;

    int i;
    for(i=0;i<100;i++){

        /* Check acknowledge reset from CAN TX driver */
        if(*tx_ctrl_reg & CAN_RST_ACK_FLAG){

            /* Remove reset flags */
            *tx_ctrl_reg = *tx_ctrl_reg & ~CAN_RST_FLAG;
            *rx_ctrl_reg = CAN_RX_NO_FLAG;

            int j;
            for(j=0;j<100;j++){
                /* Check if acknowledge ack flag is removed */
                if(!(*tx_ctrl_reg & CAN_RST_ACK_FLAG)){
                    return XST_SUCCESS; //Everything is ok
                }
            }
            return CAN_ERR_RST_ACK_FLG_NOT_RMVD; //Reset ack flag not removed
        }
    }
    return CAN_ERR_RST_FLG_NOT_ACK; //Reset flag has not been acknowledge

}
/** This function is to the status of the CAN transmission.
 *
 *  @param  can_conf is used to identify the CAN interface
 *
 *  @return XST_SUCCESS if the Test is successful, otherwise specific error.
 ******************************************************************************/
u32 can_tx_status(int can_conf){
    u32 *tx_ctrl_reg;
    tx_ctrl_reg = (u32 *)TX_CTRL_REG_ADDR;
    return *tx_ctrl_reg;
}
```

Figura 57. axi_can_*driver*.c (part 1 de 3)

```
int can_send(int can_interface, u16 id, u8 dlc, u32 *data, u32 ack){

    /* Create pointer for control register tx */
    u32 *ctrl_reg = (u32 *)TX_CTRL_REG_ADDR;

    /*Check if CAN driver is busy, if so return failure */
    int i;
    for(i=0;i<10;i++) {
        if(*ctrl_reg & CAN_TX_END_FLAG){
            break;
        }
    }
    if(i >= 10){
        return CAN_ERR_TX_BUSY;
    }

    /* Copy input CAN ID to TX_ID_ADDR register */
    u32 *id_dlc_reg = (u32 *)TX_ID_DLC_ADDR;
    *id_dlc_reg = ((u32)id & 0x000007FF) | (((u32)dlc<<11) & 0x7800);

    /* Copy input data array to TX_DATA_BYTE_X_ADDR registers */
    u32 *data_reg0_3 = (u32 *)TX_DATA_BYTE_3_0_ADDR;
    *(data_reg0_3) = ((u32)data[0] & 0x000000FF) |
                     (((u32)data[1] << 8) & 0x0000FF00) |
                     (((u32)data[2] << 16) & 0x00FF0000) |
                     (((u32)data[3] << 24) & 0xFF000000);

    u32 *data_reg7_4 = (u32 *)TX_DATA_BYTE_7_4_ADDR;
    *(data_reg7_4) = ((u32)data[4] & 0x000000FF) |
                     (((u32)data[5] << 8) & 0x0000FF00) |
                     (((u32)data[6] << 16) & 0x00FF0000) |
                     (((u32)data[7] << 24) & 0xFF000000);

    u32 *ack_set_reg = (u32 *)CAN_FRAME_ACK_SET;
    *ack_set_reg = ack;

    /* Request CAN frame to be sent by the driver */
    *ctrl_reg = *ctrl_reg | CAN_TX_ENA_FLAG;

    /* Request CAN frame to be sent by the driver */
    *ctrl_reg = *ctrl_reg | CAN_TX_ENA_FLAG;

    for(i=0;i<100;i++) {}
    for(i=0;i<1200;i++){
        if(*ctrl_reg & CAN_TX_END_FLAG){
            *ctrl_reg = *ctrl_reg & ~CAN_TX_ENA_FLAG;
            return XST_SUCCESS;
        }
    }
    *ctrl_reg = *ctrl_reg & ~CAN_TX_ENA_FLAG;
    return CAN_ERR_TX_BUSY;
}
int can_receive(int can_interface, u16 mbox, u16 *id, u8 *dlc, u8 *data){

    u32 *rx_ctrl_reg = (u32 *)RX_CTRL_REG_ADDR;
    u32 i;

    /* Check if mailbox contains a message */
    if(!(*rx_ctrl_reg & CAN_RX_FRM_RCVD)){
        return CAN_RX_EMPTY;
    }

    /* Copy input CAN ID to TX_ID_ADDR register */
    u32 *id_dlc_reg = (u32 *)RX_ID_DLC_ADDR;
    *id = (u16)(*(id_dlc_reg) & 0x000007FF);
    *dlc = (u8)((*(id_dlc_reg) & 0x00007800) >> 11);
```

Figura 58. axi_can_*driver*.c (part 2 de 3)

```
    /* Copy input data array to TX_DATA_BYTE_X_ADDR registers */
    u32 *data_reg3_0 = (u32 *)RX_DATA_BYTE_3_0_ADDR;
    data[0] = (u8)(*(data_reg3_0) & 0x000000FF);
    data[1] = (u8)((*(data_reg3_0) & 0x0000FF00) >> 8);
    data[2] = (u8)((*(data_reg3_0) & 0x00FF0000) >> 16);
    data[3] = (u8)((*(data_reg3_0) & 0xFF000000) >> 24);

    /* Copy input data array to TX_DATA_BYTE_X_ADDR registers */
    u32 *data_reg7_4 = (u32 *)RX_DATA_BYTE_7_4_ADDR;
    data[4] = (u8)(*(data_reg7_4) & 0x000000FF);
    data[5] = (u8)((*(data_reg7_4) & 0x0000FF00) >> 8);
    data[6] = (u8)((*(data_reg7_4) & 0x00FF0000) >> 16);
    data[7] = (u8)((*(data_reg7_4) & 0xFF000000) >> 24);

    /*Request reset CAN receive mbox */
    *rx_ctrl_reg = *rx_ctrl_reg | CAN_RX_FRM_RCVD_RST;
    for(i=0;i<1200;i++){
        if(!(*rx_ctrl_reg & CAN_RX_FRM_RCVD)){
            *rx_ctrl_reg = *rx_ctrl_reg & ~CAN_RX_FRM_RCVD_RST;
            return XST_SUCCESS;
        }
    }
    *rx_ctrl_reg = *rx_ctrl_reg & ~CAN_RX_FRM_RCVD_RST;
    return CAN_ERR_RX_FLAG_NOT_REMOVED;
}
```

Figura 59. axi_can_*driver*.c (part 3 de 3)

2. Llibreria adc_*driver*

```
/*
 * adc_driver.h
 *
 *  Created on: 08/06/2017
 *      Author: Carlos Galindo
 */

#ifndef ADC_DRIVER_H_
#define ADC_DRIVER_H_

#include <xparameters.h>
#include <xstatus.h>
#include <xil_exception.h>
#include <xbasic_types.h>
#include "xspi.h"
#include <xspi_l.h>
#include <stdio.h>

#define SGL     1
#define DIFF    0
#define ODD     1
#define SIGN    0

#define SPI_DEVICE_ID   XPAR_SPI_0_DEVICE_ID

int ADC_ini(XSpi* xspi);
int ADC_read(XSpi* xspi, u8 *meas);

#endif /* ADC_DRIVER_H_ */
```

Figura 60. adc_*driver*.h

```c
/*
 * adc_driver.c
 *
 *  Created on: 08/06/2017
 *      Author: Carlos Galindo
 */

#include "adc_driver.h"

int ADC_ini(XSpi* xspi){
    int status;
    XSpi_Config *config_ptr;

    /* Set up the device */

    config_ptr = XSpi_LookupConfig(SPI_DEVICE_ID);
    if (config_ptr == NULL) {
        return XST_DEVICE_NOT_FOUND;
    }

    status = XSpi_CfgInitialize(xspi, config_ptr,
                  config_ptr->BaseAddress);
    if (status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    /*
     * Perform a self-test to ensure that the hardware was built correctly.
     */

    /*
     * Perform a self-test to ensure that the hardware was built correctly.
     */
    status = XSpi_SelfTest(xspi);
    if (status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    status = XSpi_SetOptions(xspi, XSP_MASTER_OPTION |
                             XSP_CR_CLK_PHASE_MASK);
    if(status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    status = XSpi_SetSlaveSelect(xspi, 0x01);
    if(status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    /*
     * Disable Global interrupt to use polled mode operation
     */
    //XSpi_IntrGlobalDisable(&xspi);

    XSpi_Start(xspi);
    XSpi_IntrGlobalDisable(xspi);

    return XST_SUCCESS;
}
```

Figura 61. adc_*driver*.c (part 1 de 2)

```
int ADC_read(XSpi* xspi, u8 *meas){
    int status;
    u8 write_buffer[2];
    write_buffer[0] = 0x68;

    status = XSpi_Transfer(xspi, write_buffer, meas,2);
    if(status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    return XST_SUCCESS;
}
```

Figura 62. adc_*driver*.c (part 2 de 2)

3. Llibreria adc_*driver*

```
/*
 * disp_7seg_driver.h
 *
 *  Created on: 22/05/2017
 *      Author: Carlos Galindo
 */

#ifndef DISP_7SEG_DRIVER_H_
#define DISP_7SEG_DRIVER_H_

#include <xparameters.h>
#include <xstatus.h>
#include <xil_exception.h>
#include <xbasic_types.h>
#include <stdio.h>


#define DISP_7SEG_DRIVER_OFFSET     (0x04)

#define DISP_7SEG_ADDR              (XPAR_AXI_7SEG_DISPLAY_0_BASEADDR + 0*DISP_7SEG_DRIVER_OFFSET)

void set_display_value(u8 value);

#endif /* AXI_CAN_DRIVER_H_ */
```

Figura 63. disp_7seg_*driver*.h

```c
/*
 * disp_7seg_driver.c
 *
 *  Created on: 22/05/2017
 *      Author: Carlos Galindo
 */

#include "disp_7seg_driver.h"

/** This function is used to configure and start the CAN interface correctly.
 *
 *  @param  can_conf is used to identify the CAN interface
 *
 *  @return XST_SUCCESS if the Test is successful, otherwise specific error.
 ******************************************************************************/
void set_display_value(u8 value){
    u32 * ctrl_reg = (u32 *)DISP_7SEG_ADDR;
    u8 decimal_val = 0;
    u8 unit_val = value;
    //*ctrl_reg = 0x00000077;
    if(value > 99){
        *ctrl_reg = 0x000000EE;
    }else{
        while(unit_val > 9){
            decimal_val ++;
            unit_val = unit_val - 10;
        }
        *ctrl_reg = (u32)(decimal_val) | (u32)(unit_val<< 4);
    }

}
```

Figura 64. Dsp_7seg_*driver*.c

Annex 5 – CONFIGURACIÓ MICROBLAZE

La configuració del sistema Microblaze s'ha realitzat amb l'assistent de configuració XPS Core Config. La primera pàgina que s'obre (veure Figura 65) mostra un resum de la configuració i una indicació de les recursos utilitzats, freqüència, àrea i rendiment.

S'ha seleccionat l'opció d'optimització per àrea i deshabilitat l'ús del sistema de *debug* per que sigui possible generar el sistema dins les possibilitats de la Spartan-6.

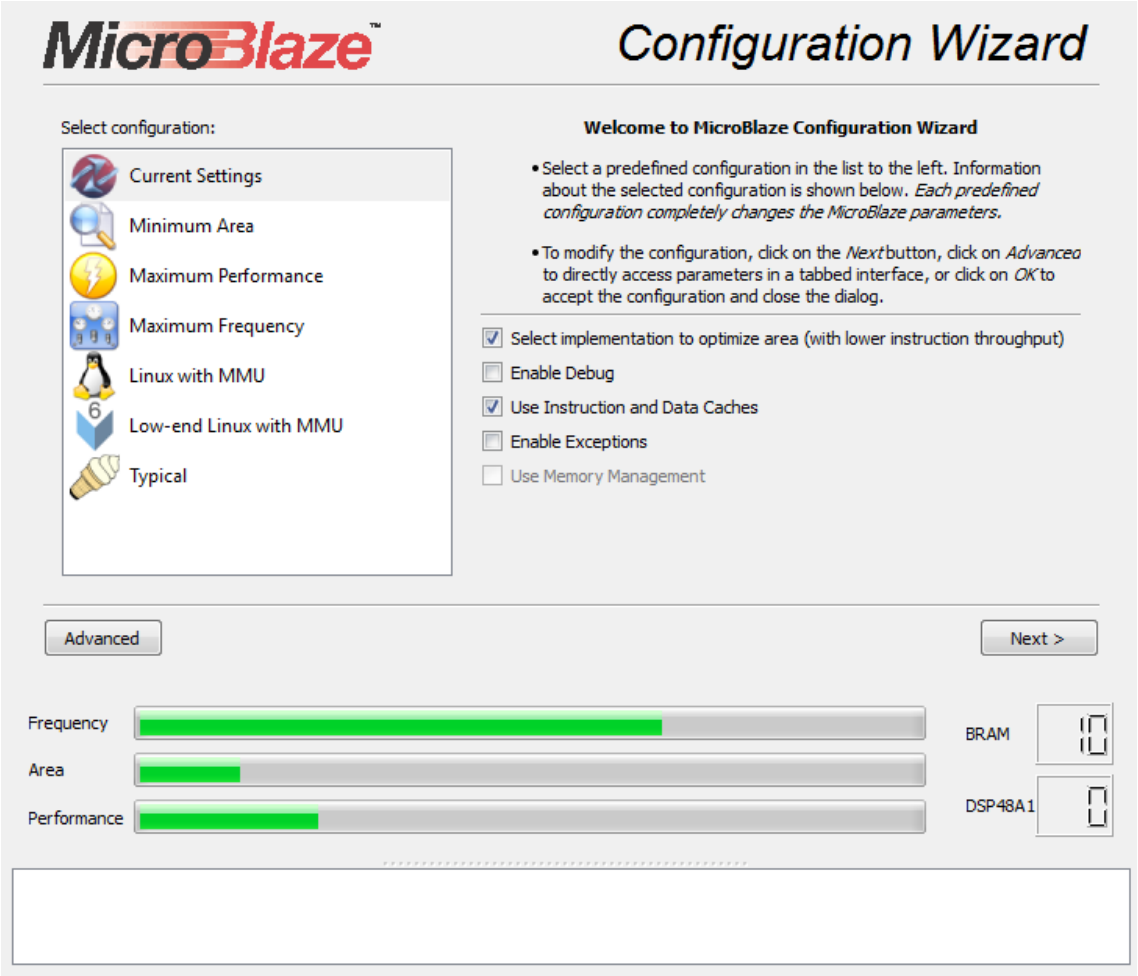


Figura 65. Pàgina 1 de 4 de l'assistent de configuració XPS Core Config

Les opcions més importants seleccionades pel disseny són:

- Unitat de punt flotant i de multiplicació i divisió de nombres enters deshabilitada (veure Figura 66)
- Les dimensions del cache de dades i d'instruccions s'ha limitat a 8 kB cadascun. (veure Figura 67)
- La configuració del PVR i els bussos de comunicació interns no s'ha modificat, es a dir, s'ha deixat la configuració per defecte (veure Figura 68)

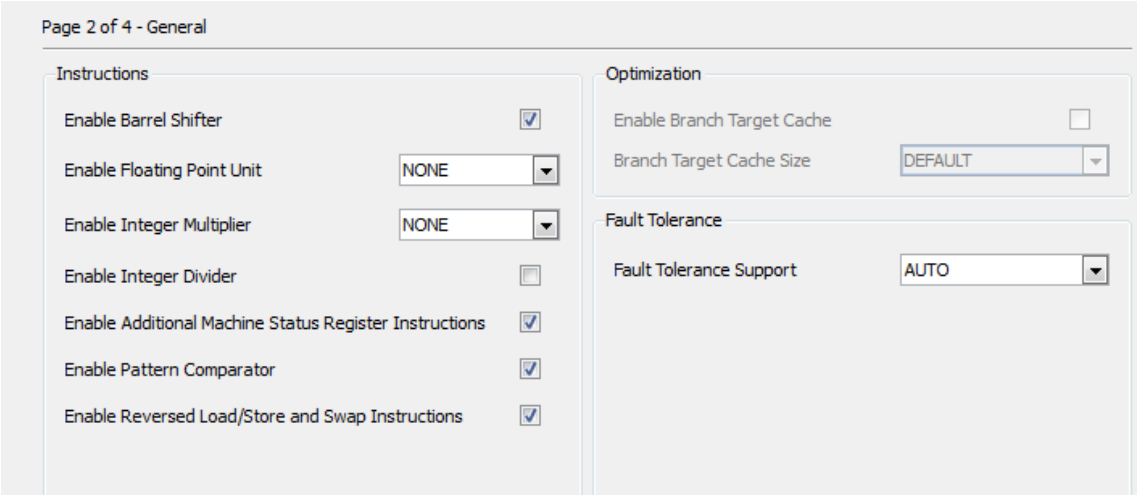


Figura 66. Pàgina 2 de 4 de l'assistent de configuració XPS Core Config

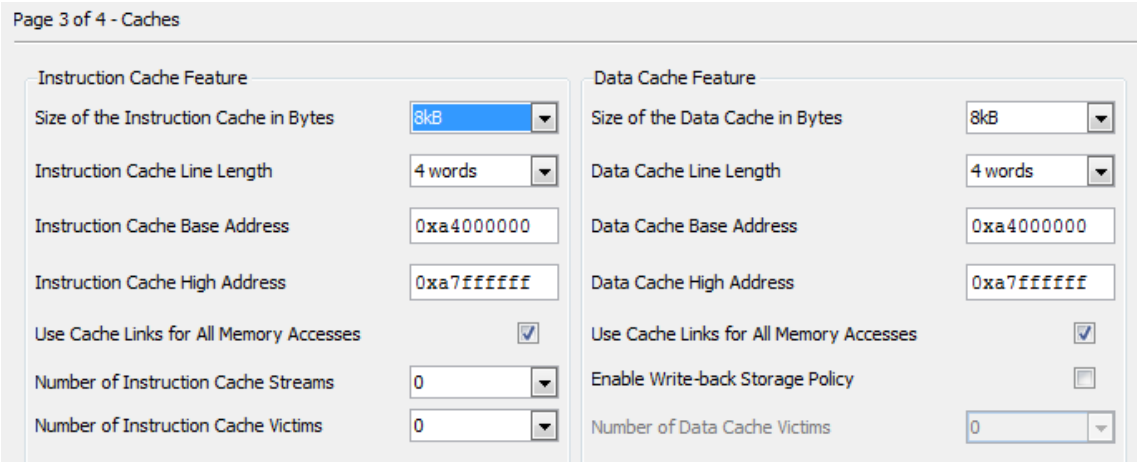


Figura 67. Pàgina 3 de 4 de l'assistent de configuració XPS Core Config

Figura 68. Pàgina 4 de 4 de l'assistent de configuració XPS Core Config