POLITECNICO DI TORINO

Master of Science in Computer Engineering

Master Degree Thesis
in
Embedded Systems

Design of a test platform for CAN BUS in automotive field

Supervisor
prof. Massimo Violante

Candidate
Stefano Cirici

Company supervisor
Intecs Solutions
dott. ing. Filippo Giuliani

July 2019

I hereby declare that the contents and organisation of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

. . . . . .

Stefano Cirici

Turin, 26 July 2019

† To my loving grandpa

You left your poems
on our hearts and minds

You left your prints
on our lives

Abstract

Communications in automotive field became critical since electronic computers have been embedded in cars to implement safety features. Being a safety component, each of them have to stand some standard of security in order to avoid jeopardizing human lives. For this matter, communications between components in automotive field need to pass a process of validation. The majority of these communications happens on the Controller Area Network. Since its introduction, it has become the most common network protocol used in automotive field. This serial bus system is the main actor of this thesis: here is reported how a test platform has been designed in order to specifically test and stress this protocol. The developed module, called "DIANA Disturbance Tool", have been designed starting from a digital design, continuing with the firmware and ending with the analog design, all in such a way as to comply with the existing test bench, the Digital Instrument for Automatic Network Analysis (DIANA). The digital design is the core of the project and provides the possibility to introduce both logic and analog errors and to generate a trigger signal when a specific event occurs. The firmware interacts

between the hardware and the external test bench, granting the entire control of the digital design. The hardware is controlled in order to act on the bus, eventually applying errors. Furthermore, an application able to control the system and to send and receive CAN messages has been developed for testing purposes. The design has been tested with behavioral testbenches and with the help of an Integrated Logic Analyzer. The next step is to integrate the project with the DIANA platform and to improve it to make it compatible with the CAN FD protocol.

Contents

Abstract v

Contents vii

Introduction 1

Scope . . . 2

1 Controller Area Network 3

1.1 History . . . 3

1.2 Description . . . 4

1.3 CAN Physical Layer . . . 5

1.3.1 Signal Level and Bit Representation . . . 5

1.3.2 Transmission Medium . . . 6

1.4 CAN Protocol . . . 7

1.4.1 Bit Timing and Synchronization . . . 7

1.4.2 Object Layer . . . 8

1.4.3 Transfer Layer . . . 8

1.5 CAN Flexible Data Rate . . . 14

2 DIANA Disturbance Tool 16

2.1 DIANA testbench . . . 16

2.2 Disturbance Tool Specifications . . . 17

2.3 Project Partitioning . . . 19

3 Digital Design **20**
3.1 Disturbance Tool . . . 22
3.1.1 CAN Sniffer . . . 22
3.1.2 Trigger Logic . . . 26
3.1.3 Logic Error Generator . . . 27
3.1.4 Analog Error Generator . . . 28
3.1.5 Configuration Registers . . . 29
3.2 ZYNQ Processing System . . . 32
3.3 AXI Interconnect . . . 33

4 Firmware **34**
4.1 Zedboard . . . 34
4.2 Serial Interface . . . 35
4.2.1 Read Line . . . 35
4.2.2 Parse Line . . . 36
4.2.3 Execute . . . 36
4.3 Disturbance Tool Configuration . . . 38
4.4 CAN Peripherals Usage . . . 39
4.5 Registers Usage . . . 41

5 Analog design **43**
5.1 Schematics . . . 43
5.2 Printed circuit board . . . 45

6 Graphical User Interface **47**
6.1 Serial Interface . . . 47
6.2 Graphical Interface . . . 49
6.2.1 Connection Window . . . 49
6.2.2 Main Window . . . 50
6.2.3 Send Window . . . 54
6.2.4 About Window . . . 54

7 Verification **55**
7.1 Behavioral Verification 55
7.1.1 Sniffer Testbench 55
7.1.2 Trigger Testbench 56
7.1.3 Logic Error Testbench 58
7.1.4 Register Test 60
7.2 Hardware Verification 61
7.2.1 Integrated Logic Analyzer 62
7.2.2 Hardware Tests 64

Conclusion **71**

References **73**

Acronyms **74**

Introduction

Since the production of first self-propelled vehicles, humans wanted to implement comfort and safety features to make transportation pleasant and riskless. This means the introduction of more and more facilities and equipment in order to achieve such goals. Furthermore, in the last years, mechanical parts are being replaced by electronic components to satisfy customers' desire of innovative vehicles and to comply with stricter regulations about exhaust emissions. In addition, the incorporation of automation techniques and luxury features triggered a rapid increase in the use of onboard electronics. This is the beginning of the electrification era [1].

Over the years it has been noticed that computers controlling a feature, called Electronic Control Unit (ECU), could drastically enhance vehicle functionality if connected together in order to interact and exchange information. These interconnections were initially realized between each ECU with a physical channel allocated for each signal (point-to-point wiring). This resulted in a massive effort to wire an entire vehicle and excessive complexity in diagnose faults and making modifications. A bus architecture is the only solution for this problem.

Depending on the criticality, the bandwidth and the purpose of the transferred messages, different networks exist. Among them, the Controller Area Network (CAN) is the most widely used communication protocol for in-vehicle networks. Nowadays every new car has at least one CAN system

onboard [2].

Scope

The CAN protocol can be defined as a robust high-speed[1] signal information platform, characterized by reliable data transmissions that satisfy real-time requirements. Be reliable it does not means that it is fault free. Vehicles with CAN are subject to electronic faults as well as older vehicles. Communication problems can occurs if a physical error appears on the bus (wires become grounded, shorted or break), if a problem arise in one ECU (dead battery can cause settings loss) or if an ECU is not behaving properly (e.g. writing on the bus messages not compliant with the protocol).

Creating a platform able to arbitrarily introduce such errors, in order to test the correctness of transactions on the bus, is the goal of this thesis. The main design will be written in VHDL and will be able to inject both logic and hardware perturbations. It will be ready to be used with the DIANA testbench, in place of the existing Vector CANstress, in order to carry out all the disturbances generated in the current state for verification of the testing standards imposed by the FCA carmaker.

[1]The CAN protocol speed is highly dependent on the wire length: on short connections (<40m) it can reach 1 Mbit/s. See chapter 1 – Controller Area Network for more info.

Chapter 1

Controller Area Network

1.1 History

In 1986 Robert Bosch GmbH introduced the Controller Area Network (CAN) protocol at the Society of Automotive Engineers. It was a revolutionary solution to growing material costs, production time and communication reliability. The protocol was an innovation for its non destructive bus arbitration, no central bus master, error detection and handling capabilities. Furthermore, adding CAN hardware to each ECU allowed the creation of a single serial bus network, superseding point-to-point wiring connections. Just one year after, the first CAN controller chips were available [3].
The Bosch CAN specification 2.0 [4] were released in the early 1990s and the ISO 11898 [5] standard was published shortly after. In 2012 Bosh released the CAN Flexible Data rate (CAN FD) (see section 1.5 for more).

Today more than 70 ECUs communicating via the CAN network can be found in vehicles and the protocol has spread in many other industries with different technical applications[1].

[1] CAN is used in elevator systems, ships, trains, aircraft, x-ray machines and other medical equipment, logging equipment, tractors and combines, coffee makers and other major appliances [2].

1.2 Description

The Controller Area Network is a very reliable multi-master serial bus system whose description covers both the Physical Layer (PL) and the Data Link Layer (DLL) of the ISO/OSI seven layer model (see Figure 1.1). The CAN protocol specifies the data communication model. It covers the Data Link Layer (Medium Access Control (MAC), Logical Link Control (LLC)) and the Physical Layer (Physical Layer Signalling (PLS)), treated in section 1.4. CAN specifications also cover the reference model for data communication in the Physical Layer (Physical Medium Attachment (PMA) and Physical Medium Specification (PMS)), treated in section 1.3. No CAN standard exist for the Medium Dependent Interface (MDI)[2].

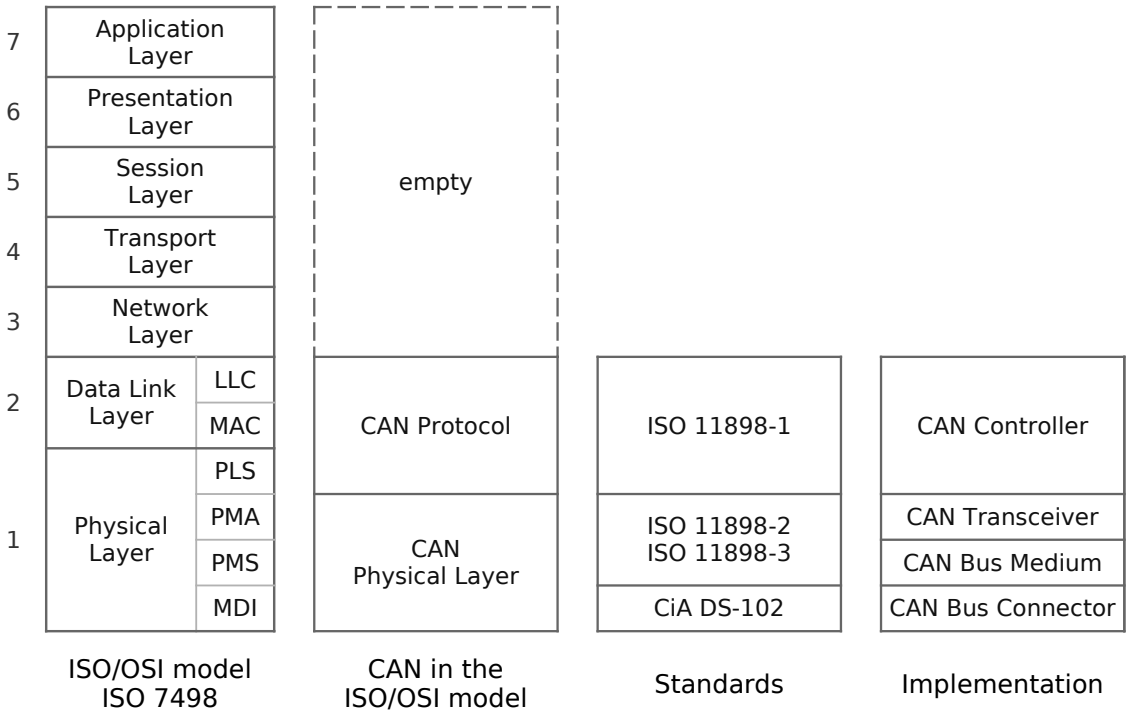

Figure 1.1: CAN Standard and Implementation referred to ISO/OSI levels

[2]The CiA DS-102 recommends the use of a SUB-D9 connector and specify pin assignments. This is a de facto standard.

The network consists of a number of ECUs, called CAN nodes, connected to a physical bus, usually in a line topology (some alternatives contemplate the use of a passive star topology). A node should implement a CAN driver (as an interface for the application, usually a micro controller), a CAN controller and a CAN transceiver. The maximum number of nodes is defined as 32 but more nodes can be used depending on the quality of the network[3] and transceivers.
Each node can transmit on the bus only when is free and the message is broadcasted in the network, readable from every node. There are no addresses, in fact the transmission is message-based (event-driven): each node can select relevant messages (by message identifier and node filtering) and ignore the others.

1.3 CAN Physical Layer

The Physical Layer defines how the actual transfer of bits between different nodes happen with respect to electrical properties. The physical transmission media is a two-wire differential (relative to a common ground) electric cable, often an Unshielded Twisted Pair (UTP). The two CAN differential signals are called `CAN_H` and `CAN_L`. Using a differential line allows to effectively reduce interference from other components in the vehicle. The transmitter drives differential voltages to signal a logic 0 (dominant). Logic 1 (recessive) is assumed when no node is driving the bus.

1.3.1 Signal Level and Bit Representation

The Physical Layer Signalling (PLS) is in charge of managing timing and synchronization of signals on the bus. Both high-speed and low-speed versions of the protocol defines voltage levels for dominant and recessive logic

[3]Important factors are capacitive load, overall line length, network termination concept and connecting line type.

values on the bus (see Table 1.1). Furthermore specifications on timing requirements for transitions (D→R, R→D) are given for transceivers.

CAN types	Logic Value	CAN_H V	CAN_L V	Differential V
High speed	0 D	3,5	1,5	2,0
	1 R	2,5	2,5	0
Low speed	0 D	3,6	1,4	2,2
	1 R	0	5,0	-5,0

Table 1.1: CAN bus levels for high-speed and low-speed standards. Typical (nominal) values for transmission are reported (see ISO [6], [7] for more details)

The bit coding used in CAN is the Non Return to Zero (NRZ) which means that each bit is coded with a single value. This brings to synchronization problems that will be treated in subsection 1.4.1 – Bit Timing and Synchronization, since CAN does not expect the use of an external clock signal. A "wired-and" cabling (open collector) is performed: when at least one node is transmitting a dominant value, a logic 0 (D) will be forced on the bus; only if every node is transmitting a recessive value[4], a logic 1 (R) will be detected on the bus.

1.3.2 Transmission Medium

The typical transmission medium is a twisted pair conductor of an unshielded wire. Twisting makes differential signal communications more effective towards electromagnetic disturbs.

[4]When a node is in sleep state or simply does not want to transmit anything, its CAN controller will broadcast a recessive value on the bus.

1.4 CAN Protocol

The definition of the CAN protocol includes specifications for PLS that describes bit timing and synchronization (see subsection 1.4.1) and for the Data Link Layer. In turn the DLL is divided in the Object Layer, that mainly manages message filtering (see subsection 1.4.2) and the Transfer Layer, that is the kernel of the CAN (see subsection 1.4.3).

1.4.1 Bit Timing and Synchronization

Synchronization is performed at the start of a message (R→D edge, hard-sync) and is repeated at each bit change. Thanks to the use of the "bit stuffing" technique a minimum number of transition is assured: controllers transmitting on the bus insert a complementary bit (stuff bit) after five homogeneous bits. Stuffing is performed starting from the Start Of Frame (SOF) bit until the end of the CRC field. A receiver uses stuff bits for resynchronization (soft-sync) and ignore them for data computing.

The duration of a bit, called Nominal Bit Time, can be divided into separate non-overlapping time segments (as can be seen in Figure 1.2) of the duration of a multiple of a time quantum (tq)[5]. These segments are:

- Synchronization segment (`SYNC_SEG`): is the initial part of the bit time, where there is an edge of the transition, used to synchronize the nodes. It is 1 time quantum long.
- Propagation time segment (`PROP_SEG`): portion of bit time dedicated for delay times compensation[6]. Can be from 1 to 8 tq.

[5] A time quantum is a fixed unit of time, multiple of a local oscillator period (usually programmable to be from 1 to 32×).

[6] It is twice the sum of the signals propagation time on the bus line, the input comparator delay, and the output driver delay [4].

- Phase Buffer Segment 1 and 2 (`PHASE_SEG1/2`): time used to compensate phase errors. These segments are resized at each resynchronization of a maximum number of tq equal to a Synch Jump Width (SJW) value. The first can be from 1 to 8 tq, the second is the maximum between `PHASE_SEG1` and the information processing time (that is at least of 2 tq).

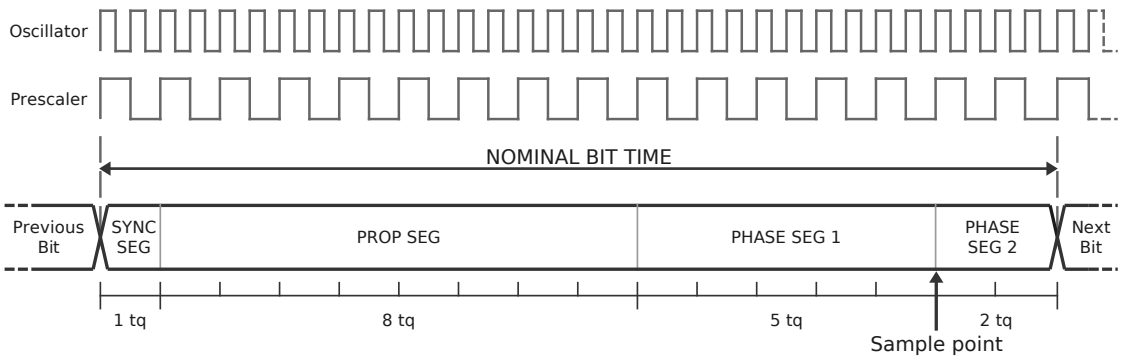

Figure 1.2: Partition of Bit Time with an oscillator clock of 64 MHz (not visible in the chart), prescaler of 2, 16 tq and sample point at 87.5 % for a speed of 1 Mbit/s. Changing the prescaler, maintaining the same values of other parameters, allows to achieve other CAN speed

1.4.2 Object Layer

Characteristics of the Object Layer depends on the particular hardware (CAN controller) in use. Here is implemented message filtering for each node based on the message identifier. Management of messages to be transmitted and interface with Application Layer (Level 7 of ISO/OSI) are also defined.

1.4.3 Transfer Layer

This layer is the kernel of the CAN protocol: most of the standards applies here. It is a bridge between the Physical Layer and the Object Layer, and describes message types (frames), node arbitration, error detection and fault confinement.

1.4.3.1 Data Frame

The Data Frame is the most used message type and serve to transmit data. It can transport a maximum payload of eight bytes. Its format can vary between the Standard Controller Area Network (CAN) and the Extended format as can be seen in Figure 1.3.

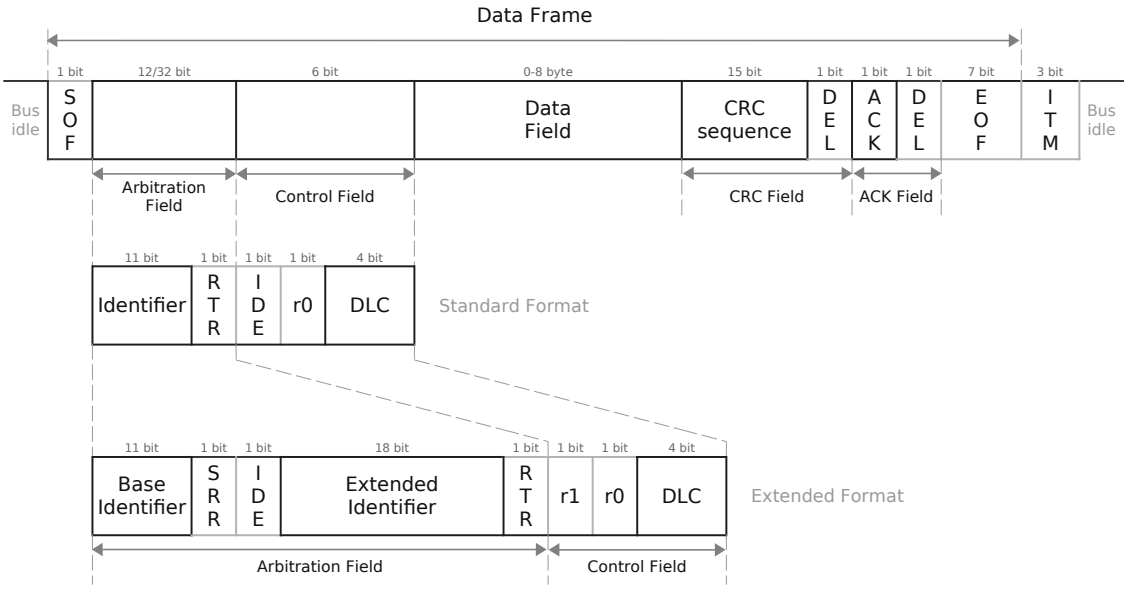

Figure 1.3: Comparison of CAN Data Frame in Standard and Extended Format

SOF The Start Of Frame (SOF) is a dominant bit that assure hard synchronization between all nodes. It can only be transmitted if the bus is in idle state.

Arbitration Field The 11 bit Identifier (ID) sets the priority of the frame and is used by other nodes to identify the content of the message (lower identifiers have higher priority). The extended format expects an additional identifier field of 18 bits. The Remote Transmission Request (RTR) bit is used to distinguish between data frame (D) and remote frame (R). In the extended format this bit is moved after the second identifier and in its place is introduced the Substitute Remote Request (SRR) bit, which is always

recessive.

Control Field The Identifier (IDE) distinguish between standard format (D) and extended format (R)[7]. The Data Length Code (DLC) is a 4 bits field that codify the number of payload bytes that will be transmitted in the data field. Reserved bits r0 and r1 are irrelevant and transmitted as dominant.

Data Field Is the payload of the frame. It can be from 0 to 8 bytes, as dictated by the DLC.

CRC Field Includes a 15 bits checksum sequence of Cycle Redundancy Check type[8] , computed from SOF until the Data Field and a recessive delimiter.

ACK Field Acknowledge field. The ACK slot bit is transmitted recessive and overwritten dominant from receivers if they acknowledge the CRC sequence. It follows a 1 bit recessive delimiter.

EOF The End Of Frame (EOF) signals the end of the frame with 7 recessive bits.

1.4.3.2 Remote Frame

This type of frame has the same structure of a Data Frame except for the data field, which is empty. It is used by nodes to request some specific (identified by the ID) data. Here the RTR bit is recessive[9].

[7]Both Data and Remote standard format frames have the priority. For the extended format, the IDE bit is considered to be in the Arbitration field.

[8]CRC polynomial = `0x4599`, initialization at zero.

[9]If another node is transmitting a data frame with the same ID, it will have the priority and supersede the remote request, since it is already the answer to the requested data.

1.4.3.3 Error Frame

Is a special message that explicitly violates the CAN stuffing rule. It is transmitted by a node that detects an error in a message and can be active (transmit dominant bits) or passive (transmit recessive bits, possibly overwritten by dominant bits by other nodes), depending on the node status. After the error flag the controller transmit recessive bits until it detects a recessive bit on the bus (other nodes have finished sending their dominant flags) and send seven more recessive bits (see Figure 1.4).

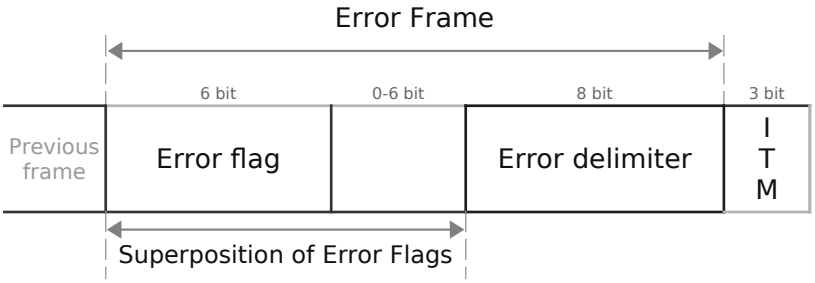

Figure 1.4: CAN active Error Frame

1.4.3.4 Overload Frame

Similarly to the active Error Frame, is composed by an Overload flag of 6 dominant bits (and eventual overload flag echo from other nodes) and a 8 recessive bits Overload delimiter. It can only be transmitted during an Intermission sequence, signalling the overload condition of the node (the node is too busy and requires a delay before the next data reception) and imposing a delay for future data transmission.

1.4.3.5 Interframe Space

The Interframe Space is a sequence of recessive bits that separate any kind of frame from Data and Remote Frames. It consist of 3 recessive bits, Intermission (ITM), after which the bus is considered to be in idle state (remains

recessive) until the next frame is transmitted. Intermission can only be interrupted by an Overload Frame.

1.4.3.6 Bus Access and Arbitration

Unlike other networks protocols (such as Ethernet), CAN uses a nondestructive bus arbitration approach. If two or more nodes start transmitting at the same time, the bus access conflict is resolved by a bit-wise arbitration during the Identifier transmission (MSB first). If a controller detect a dominant value on the bus when transmitting a recessive value, the node acknowledge that has lost arbitration and must withdraw without sending one more bit [4]. This method is called Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) and ensure a prioritization of CAN messages among the network.

1.4.3.7 Error Detection and Signalling

There are five error types:

- Bit Error: generated by a node transmitting on the bus that detects a bit value different than the one sent. Exceptions are the Arbitration Field, the ACK slot and a passive Error Flag.
- Stuff Error: detected by a node if the bit stuffing method is not respected (the sixth bit of a sequence to be stuffed does not generate a transition).
- CRC Error: generated by a node if the internally computed CRC is different from the one broadcasted on the bus[10].
- Form Error: detected if a field with fixed bit contains an illegal value (e.g. reserved bit r0 or r1 transmitted recessive).

[10]Thanks to CRC, up to 5 randomly distributed errors and any odd number of errors in a message are detected.

- ACK Error: detected by the transmitter if the ACK slot is not overwritten dominant.

A node can signal a detected error sending an Error Frame. Transmission of CRC errors starts after the ACK delimiter, for other errors it starts immediately.

1.4.3.8 Message Validation

A transmitter can consider a message as valid when no error occurs from the SOF until the EOF. When errors are detected, re-transmission starts automatically as soon as the bus is idle.

Receivers consider valid a message when the are no errors until the last but one bit of the EOF. Each receiver that correctly detects a data or remote frame, sends a dominant bit on the ACK slot.

1.4.3.9 Fault Confinement

Each CAN node has a Receive Error Counter (REC) and a Transmit Error Counter (TEC). Starting from an 'error active' state when REC & TEC $<$ 128, a node can normally take part in bus communications and send active Error Flag. When REC | TEC $\geq$ 128 (but $<$256) the node is in 'error passive' state: it can only send passive Error Flag and has to wait a Suspend Transmission Time (8 bits) before sending multiple frames. Finally, when REC | TEC $\geq$ 256 the node is in 'bus off' state and can not interfere with bus transactions. A node in 'bus off' state can become active after an hardware reset or 128 occurrence of 11 consecutive recessive bits are detected on the bus. The REC and TEC are increased when errors occurs (errors have different weights) and decreased on successful operations accordingly to CAN specifications (see [4, pp. 24-26]).

1.5 CAN Flexible Data Rate

With a raising number of ECUs in vehicles and increasing complexity of functions, CAN buses are becoming more and more crowded and bus load is breaking its bandwidth limits (1 Mbit/s). Other protocols, such as Media Oriented Systems Transport (MOST) that reaches up to 150 Mbit/s, are being used for infotainment purposes while a more expensive protocol, the FlexRay (up to 10 Mbit/s), is used for deterministic scopes. Local Interconnect Network (LIN) is used as a cheap alternative for low-speed interconnections. Nevertheless, CAN remains the predominant bus system in vehicles for its implementation flexibility and cost effectiveness.

The limiting factor of the Controller Area Network is due to its multiple access capabilities. Since several nodes can transmit on the bus at the same time, the Nominal Bit Time must not be shorter than twice the propagation time of the signal between the two most distant nodes. This happens during the arbitration phase and in the ACK slot. In between these two fields, increasing the bit rate is safe (only one node is transmitting): this is the fundamental idea of the CAN Flexible Data rate (CAN FD).

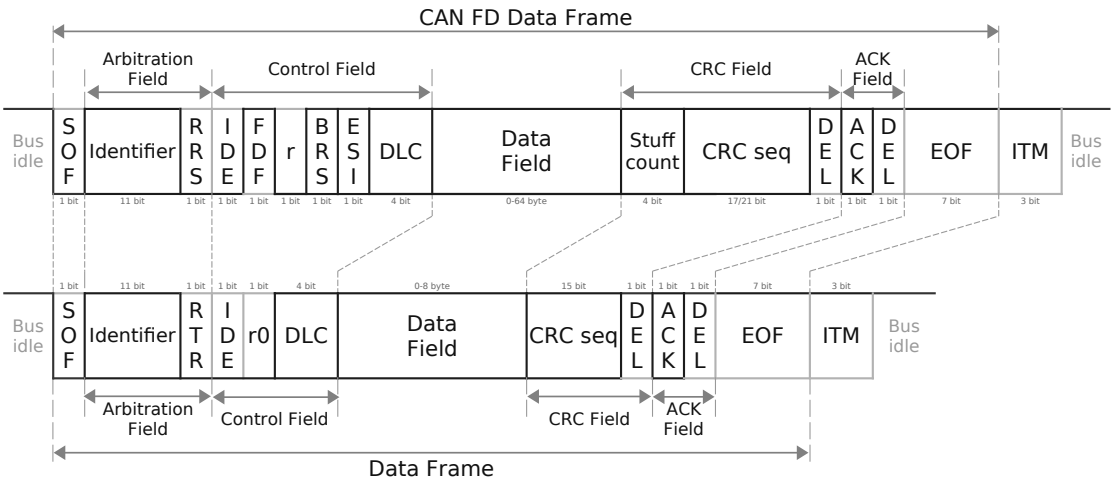

Figure 1.5: CAN FD Data Frame (above) compared to Classic CAN (below)

CAN FD, standardized in ISO 11898-1 [5], is backward compatible[11] with classic CAN and offers speeds up to 8 Mbit/s with a 0-64 bytes payload while reducing data overhead, bus load problems and data segmentation.
The data frame transmission (Figure 1.5) differs from Classical CAN for the use of the 'r0' reserved bit as a recessive Flexible Data rate Format (FDF) bit. There are no remote frames, the RTR bit is replaced by the dominant Remote Request Substitution (RRS) bit. The Bit Rate Switch (BRS) bit dictate the possibility to have an higher transmission speed (between the BRS and the CRC delimiter) when dominant. Stuffing rules and CRC calculation differs to guarantee data reliability.

[11] CAN FD ECUs can manage both classical CAN and FD messages. Classical CAN controllers that receives FD messages will raise a form error on bit r0.

Chapter 2

DIANA Disturbance Tool

Introduction

The scope of the project is to design an electronic embedded system able to inject perturbations on the CAN bus. This tool should be able to implement main functions of the "Vector Informatic CANStress" commercial product, that is currently used in the Intecs DIANA test bench, in order to supersede it. The Disturbance Tool should be able to generate all perturbations currently used in the Automatic Test Equipment for the verification of testing standards imposed by the FCA carmaker.

2.1 DIANA testbench

The Digital Instrument for Automatic Network Analysis (DIANA) is a test bench able to automate the validation process of network layer of the control units [8]. In particular it enables the validation of all layers of the ECU CAN stack. Is composed by different sub-systems:

- A National Instrument PXI system and its connectivity panel
- An application able to control the PXI board and to execute functions

and generate reports

- An ECU that is the Device Under Test
- A physical CAN bus.

The type of disturb to be generated can be set from the DIANA software (and CANstress gui), located in the PXI controller. An automatic process can be performed by loading in the tool a script file. This file allows to automatically launch test procedures standardized by the carmaker.

2.2 Disturbance Tool Specifications

The DIANA, in order to perform some Network Management and CAN BUS physical layer tests, needs an external system capable of generating Logic and Analog disturbs. Furthermore a trigger functionality is needed in order to recognize specific patterns inside a message.

Logic perturbations Some bits of the message are modified from recessive to dominant[1]. In particular, after any field in the frame, a programmable number of stuffing bits could be inserted for a programmable number of repetitions.

Analog perturbations Physical fault are generated between the CAN BUS lines. In particular:

- CANH - CANL short circuit
- CANL - VBAT short circuit
- CANH - GND short circuit

[1]A logic bit can not be changed from dominant to recessive (using a CAN compliant controller) because of the CAN protocol electrical properties (see subsection 1.3.1).

- CANH open circuit
- CANL open circuit

Trigger The tool should generate a trigger signal in any point of the CAN frame, even without a disturb. The tool should generate a trigger upon the recognition of a single bit or a field in the CAN frame.

HW and SW Integration The tool should be integrated into the DIANA without the need to modify the current test scripts already present. It should communicate via USB by emulating a serial port and generate disturbs directly on the CAN BUS. In Figure 2.1 can be seen the whole system interconnection.

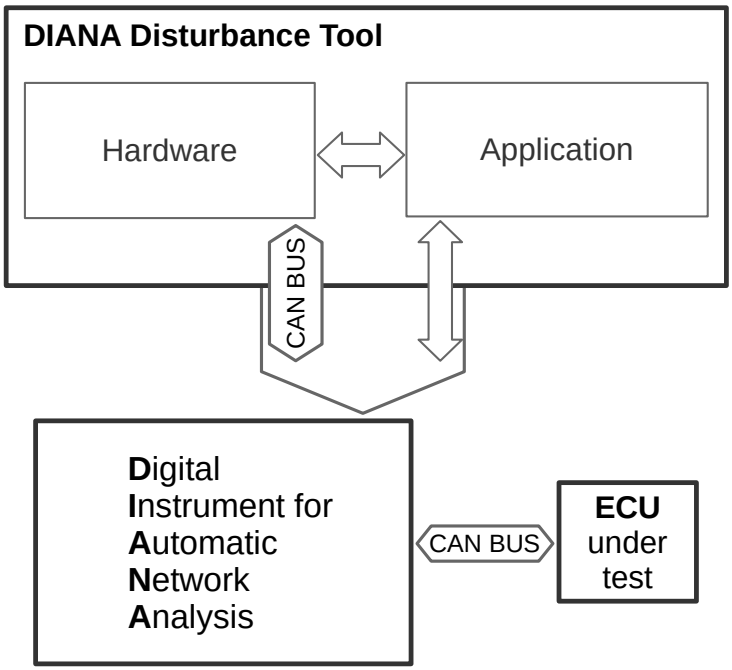

Figure 2.1: DIANA - Disturbance tool connection

All these functionalities have to be performed in real time. For this reason the choice of an FPGA based system is optimal for its performance and to reduce design and production costs.

2.3 Project Partitioning

The design have been partitioned in four sub-systems (see Figure 2.2) that will be treated separately in this thesis :

- Digital Design: VHDL code to be deployed on a FPGA. It includes all the components needed to generate logic errors and triggers. It also includes the analog error activation logic (see chapter 3)
- Firmware: C code running on the FPGA core in order to receive serial commands and set registers value of the digital design (see chapter 4)
- Analog design: hardware circuit used to input error and read messages on the CAN BUS (see chapter 5)
- Graphical User Interface: Application that simplify sending serial commands to the FPGA using a graphical environment (see chapter 6)

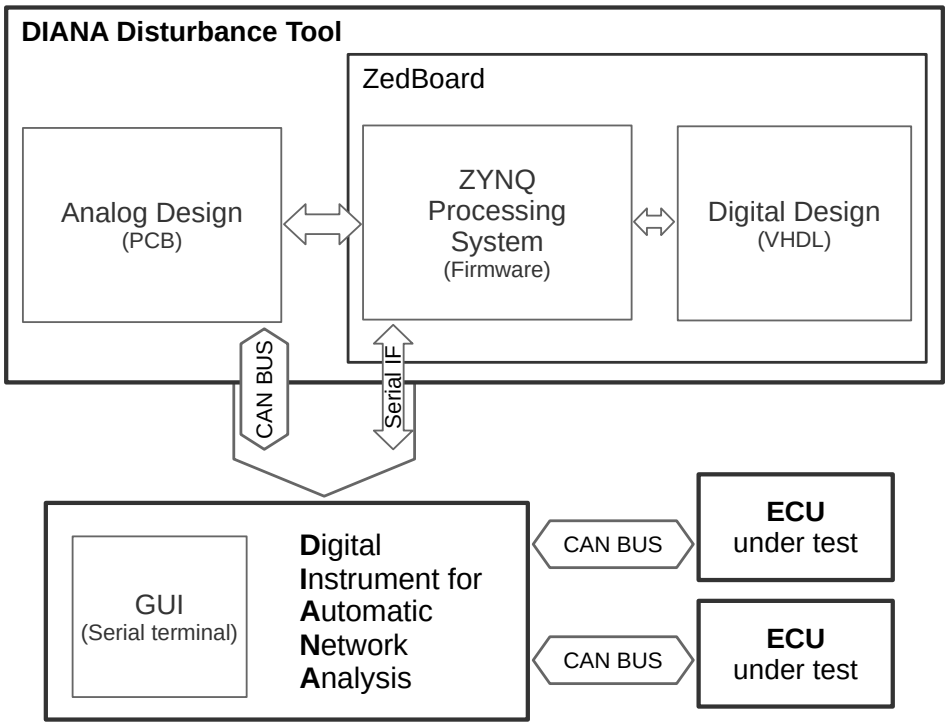

Figure 2.2: Disturbance tool project sub-parts

Chapter 3

Digital Design

Introduction

The digital design includes all the VHDL code, to be deployed on the FPGA, that describes the behavior of various sub-systems capable of generating disturbance and trigger as the project specifications (see section 2.2 – Disturbance Tool Specifications).

A Block Design is used to integrate the developed VHDL entity in the form of an IP core with the Processing System (section 3.2) with the use of an AXI peripheral interconnection (section 3.3). A reset controller module called `rst_ps7` is used to manage system reset and a clocking wizard called `clk_wiz` is used to scale the system clock to the actual value of 160 MHz. The representation of the Block Design is in Figure 3.1 – "View of Block Design from Vivado".

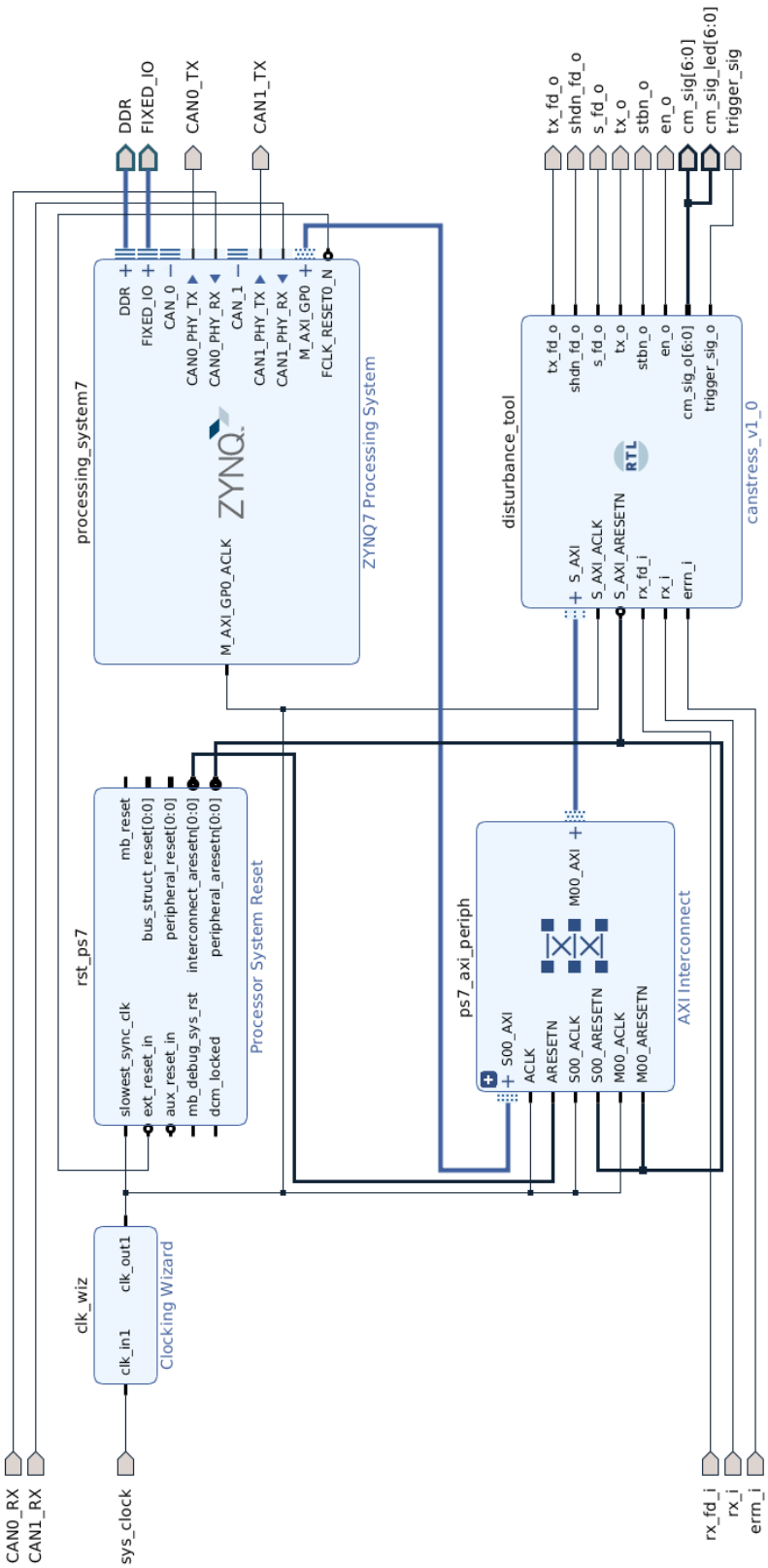

Figure 3.1: View of Block Design from Vivado

3.1 Disturbance Tool

Five components belong to the Disturbance Tool design (`canstress_v1_0` entity) as can bee seen in Figure 3.2 and they will be further detailed below.

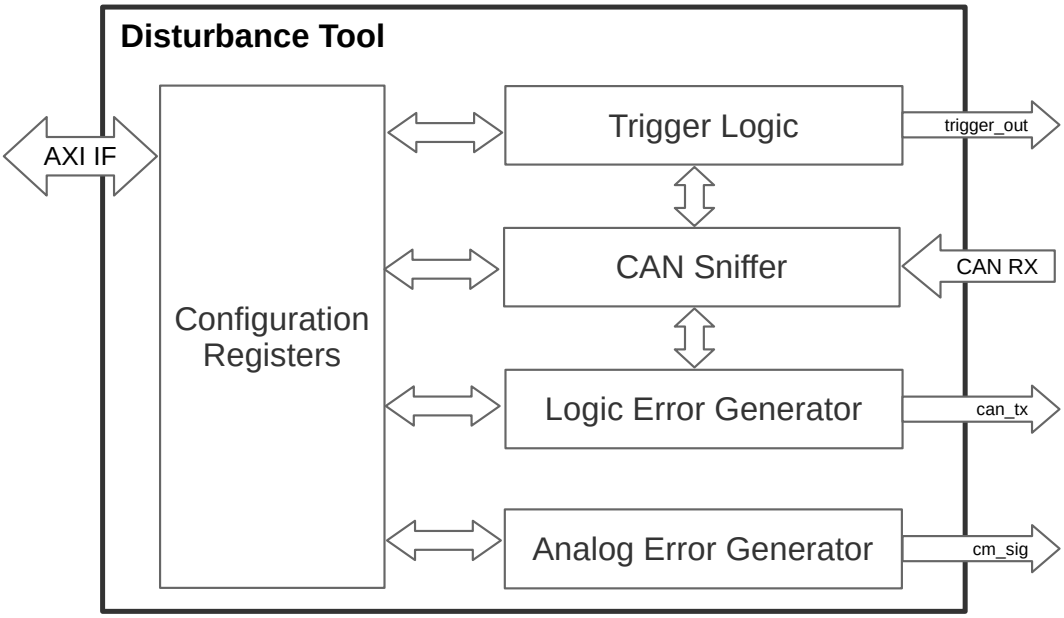

Figure 3.2: Disturbance Tool components

3.1.1 CAN Sniffer

The `can_sniffer` is a real-time CAN protocol sniffer able to recognize the current field type of the transmitted message and extract its data, making all the information available to other components. It is a controller from opencores [9] based on the Philips SJA1000 stand-alone CAN controller that has been heavily modified in order to be silent on the bus, disabling transmission and Wishbone features[1]. A diagram of its component can be seen in Figure 3.3.

[1]The opencores controller uses the Wishbone bus for its configuration. The CAN sniffer configuration takes places thanks to the CAN Registers (subsubsection 3.1.1.1).

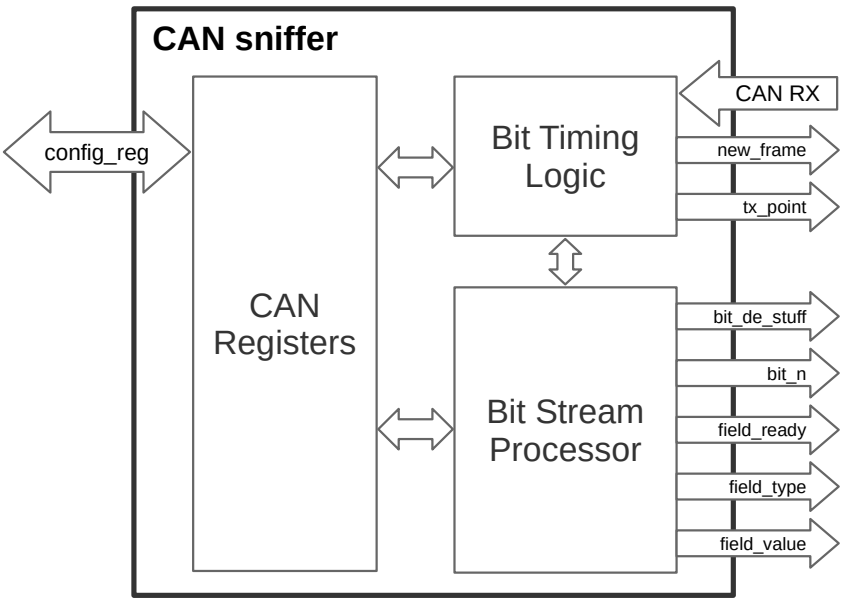

Figure 3.3: CAN sniffer components

3.1.1.1 CAN Registers

These are local registers that holds information of the controller (sniffer) such as configuration parameters, errors counters, acceptance masks, data to be transmitted and some others protocol info.

The acquisition of configuration parameters have changed by removing the wishbone functionalities and implementing a system for reading the general configuration register (described in subsection 3.1.5). When in configuration mode (`config_mode=0x01`) the Bit Timing Register (BTR) 0 and 1 are read and stored as new parameters for the controller (see Figure 3.4) and made available to others components.

The acceptance mask registers are forced to the value `0xFF` while acceptance code registers to `0x00`. In this way the controller will not perform acceptance

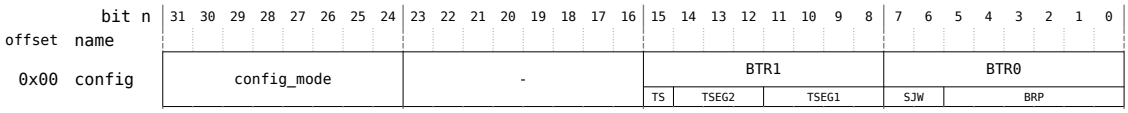

Figure 3.4: Configuration register, line 0: CAN sniffer configuration

filtering, reading all input messages without discarding anyone.

The `read` and `listen_only` signals are forced active, virtually disabling the possibility of transmitting messages.

3.1.1.2 Bit Timing Logic

The Bit Timing Logic (BTL) is in charge of sampling input bits at the right time, based on the controller configuration. The prescaler value is used to divide the system clock into a clock suitable for reading messages on the bus. This value have to be set at the system startup and all controllers on the CAN BUS have to agree to the transmission speed.

Several VHDL process implements the division of the nominal bit time into quantum and segments, following the rules discussed in subsection 1.4.1 – Bit Timing and Synchronization.

The sync, hard-sync and late edge detection logic is in charge of moving the sample point at the correct time to perform sampling and triple sampling, if enabled.

As output from the sniffer it provides the `new_frame` signal that coincides with the `hard_sync`: it rises for one clock cycle when the first dominant bit, Start Of Frame (SOF) bit, is detected on the bus. It also generates the `tx_point` that signals the exact moment when a controller should start transmitting on the bus the current bit value. Is used when transmitting dominant bits in the Logic Error Generator (see subsection 3.1.3) and to

synchronize the trigger signal with the bit transmission in the Trigger Logic (see subsection 3.1.2).

3.1.1.3 Bit Stream Processor

The Bit Stream Processor (BSP) receives the sampled bit value and reconstruct the CAN message, generating information about the message field and its value.

The input bit stream pass through a counter that count consecutive repetition of the same value. If five dominant or recessive bits in a row are detected, the `bit_de_stuff` signal rises indicating that the successive bit should be discarded.

Another counter is used to count received bits from the start of the message. The `bit_n` signal is reset at the start of the frame and increased if the new read bit is not a stuffing bit.

To understand which field is currently being transmitted on the bus, several VHDL processes generates signals indicating which field (or special bit) the bit just read is part of. In this way the `field_type` value is deduced based on these signals.

Signals in the form of `go_rx_<field_name>` are generated in a combinational way. These are really useful to discern when a field should start on the bus. Similar signals are produced for the error and overload frames and in general for error detected on the bus. The `field_ready` signal is a combination of the previous start signals.

The `field_value` is the value of the field indicated by the `field_type` signal. It can be considered correct and stable only from when the `field_ready` is high and is valid until its next rise.

The Cycle Redundancy Check (CRC) and Acceptance Code Filter (ACF) are disabled as well as other logic that implements error recognition and transmission management. Their code is still present (even if is not being used) because it might be useful for adding additional functionalities. It does not impact on design size or performance, since unused modules are not synthesized nor implemented during the building phase.

3.1.2 Trigger Logic

This module is in charge of generating a trigger when a series of conditions on the frame are met. It get to read configuration data from general registers that contains, for each CAN field, information about if the match have to be performed and, possibly, with what value.

A VHDL process generate a `frame` signal that is high when a frame is being transmitted on the bus, that is from the `new_frame` signal until the Intermission field begins. This is like an enable signal since operations can only occur when a frame is on the bus.

There are two trigger modes: bit trigger and mask trigger. When the `mode` bit is at 1 in the trigger line of general registers (see Figure 3.5) the bit trigger mode is active.

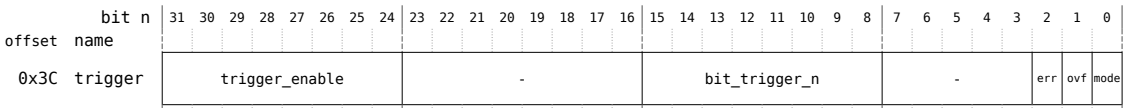

Figure 3.5: Configuration register, line 15: Trigger Logic configuration

In bit trigger mode a trigger is risen (in sync with the `tx_point`) when the configured number `bit_trigger_n` coincides with the current `bit_n` received from the CAN Sniffer. Bit trigger is not performed nor calculated when in

idle, error or overload field or when the current bit is a stuff bit.

When in mask trigger mode (`mode` bit at 0) a mask comparison between the read value (`field_value`) of the current field (`field_type`) and the match value stored in the general registers is performed. The `compare_mask` function perform a bit by bit comparison only if the corresponding bit in the mask register is at 1. If at least one masked bit does not match, the function returns 0. If all masked bit match, it return 1. If no mask is applied (mask is all zeros), the function returns 0. If the function has returned 1, the match is confirmed and the triggered is risen (in sync with the `tx_point`).

To manage the transmission of different consecutive data sets, a counter called `data_counter` is increased upon the reception of a new data byte and used to address the `data_array`, needed for the mask comparison. The `data_array` is an array of general configuration register lines (line 5 and 7) splitted bytewise: each line of the array holds a single byte as it would be transmitted on the bus. Another array, the `data_array_en`, is generated in the same way (from line 6 and 8) and contains the masks of each single byte.

The `trigger_out` signal is risen only for one clock cycle (the general clock cycle used is scaled from the system one to 160 MHz). When the trigger has risen at least once during a frame transmission, a `triggered` signal is set to high until a new frame begins.

3.1.3 Logic Error Generator

The Logic Error Generator is the module in charge of generating dominants bit on the CAN BUS. It can produce a maximum of 127 stuffing bits for a maximum of 127 repetitions after any of the field in a message. The logic line in the configuration register (as shown in Figure 3.6) allows to set these values.

bit n	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
offset name				
0x40 logic	logic_error_enable	-	stuff_field	stuff_count	stuff_num

Figure 3.6: Configuration register, line 16: Logic Error Generator configuration

Stuffing, and therefore the transmission of a dominant bit, can occur only between the `start_stuff` and `end_stuff` impulses. When the current field (`field_type`) is the same field set in the configuration registers, the next field is the one to be stuffed and, as soon as it begins, the `start_stuff` pulse is generated and the stuffing process begins.

At the beginning of the stuffing process an internal `bit_cnt` is set to the number of consecutive stuffing bits (`stuff_num`). At every `tx_point` the value to be transmitted on the bus (`can_tx`) is updated and the `bit_cnt` decreased. When the counter is at zero the `end_stuff` impulse is generated and the stuffing stops, transmitting a recessive value on `can_tx`.

This process is repeated for a `rep_cnt` times (initialized with a `stuff_count` value), allowing the generation of multiple consecutive errors on the CAN BUS.

The `stuff_complete` signal is risen when all repetition of stuffing occurrences are over (`rep_cnt=0`) and remains high until a new stuff configuration is set.

3.1.4 Analog Error Generator

This is a simple logic that set the relays activation signals (`cm_sig`) based on the values of the configuration register (see Figure 3.7). When the `analog_error_enable=0x01` the signals are updated and the `error_active` signal rises if at least one relay is active. When errors are disabled by the register, all values for relays activation are set to zero.

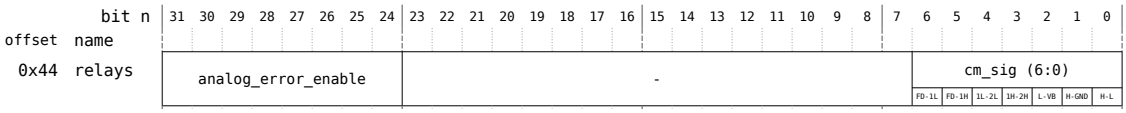

Figure 3.7: Configuration register, line 17: Analog Error Generator configuration

3.1.5 Configuration Registers

The register file is a set of registers that interfaces with the processor through an AXI4 Lite bus[2]. The AXI clock is the same used for the entire system, as said before, set to 160 MHz. In the "`can_type.vhd`" some constants are defined for register size and addresses values (see Listing 1). The AXI data and address width depends on the register size.

There are 17 registers 32 bit wide therefore 5 bits for the addressing are enough. More registers (up to a total of $2^5 = 32$) can always be added later without major modifications, for example to save data regarding CAN FD fields . The constant `REG_WRITE_END` signals the last writable register. Registers after the 17th (in this case the 18 only) are not writable. This mechanism could be improved in the future adding a bit for each register signalling its read/write access permission.

Constants for register addresses are used to easily recognize them in the code. The whole register file can be seen in Figure 3.8.

Some VHDL processes manage the AXI4 Lite standard signals[3]:

- `AWREADY` – Write address ready: slave is ready to accept a write address and control signals. Since the slave (register file) only perform read and writes from the master, it will always be ready for any operation.

[2]The AXI4 Lite is part of the ARM AMBA AXI control interface.

[3]The master in the communication is the SoC processor, the slave is the register file.

```
26      ...
27      constant REG_DATA_WIDTH : integer := 32;
28      constant REG_ADDR_WIDTH : integer := 5;
29
30      constant AXI_DATA_WIDTH : integer := REG_DATA_WIDTH;
31      constant AXI_ADDR_WIDTH : integer := REG_ADDR_WIDTH+2;
32
33      constant REG_START      : integer := 0;
34      constant REG_WRITE_END  : integer := 17;
35      constant REG_LENGTH     : integer := 19;
36
37      subtype register_line is std_logic_vector(REG_DATA_WIDTH-1 downto
        ↪  0);
38      type register_type is array(0 to REG_LENGTH-1) of register_line;
39
40      constant config_addr    : integer := 0;
41      constant arb_addr       : integer := 1;
42      constant ctrl_addr      : integer := 3;
43      constant data_addr1     : integer := 5;
44      constant data_addr2     : integer := 7;
45      constant crc_addr       : integer := 9;
46      constant ack_addr       : integer := 11;
47      constant end_addr       : integer := 13;
48      constant trigger_addr   : integer := 15;
49      constant logic_addr     : integer := 16;
50      constant relays_addr    : integer := 17;
51      constant out_addr       : integer := 18;
52      ...
```

Listing 1: Some constant definitions from `can_type.vhd`

- `AWADDR` – Write address: 32 bit wide, only a latch is performed.
- `WREADY` – Write (data) ready: slave is ready to receive a data write, no outstanding transactions are expected.
- `BVALID` – Write response valid: slave acknowledge the write response.
- `BRESP` – Write response: slave return the status of the write transaction, always `"00"`.
- `ARADDR` – Read address: equals to `REG_ADDR_WIDTH+2=7`, latched.

- `ARREADY` – Read address ready: slave is ready to receive read address and control signals. As the `AWREADY`, slave is always ready to perform a read operation.
- `RVALID` – Read valid: slave acknowledge the read data process
- `RRESP` – Read response: slave return the status of the data transfer, `"00"` is an OK response.

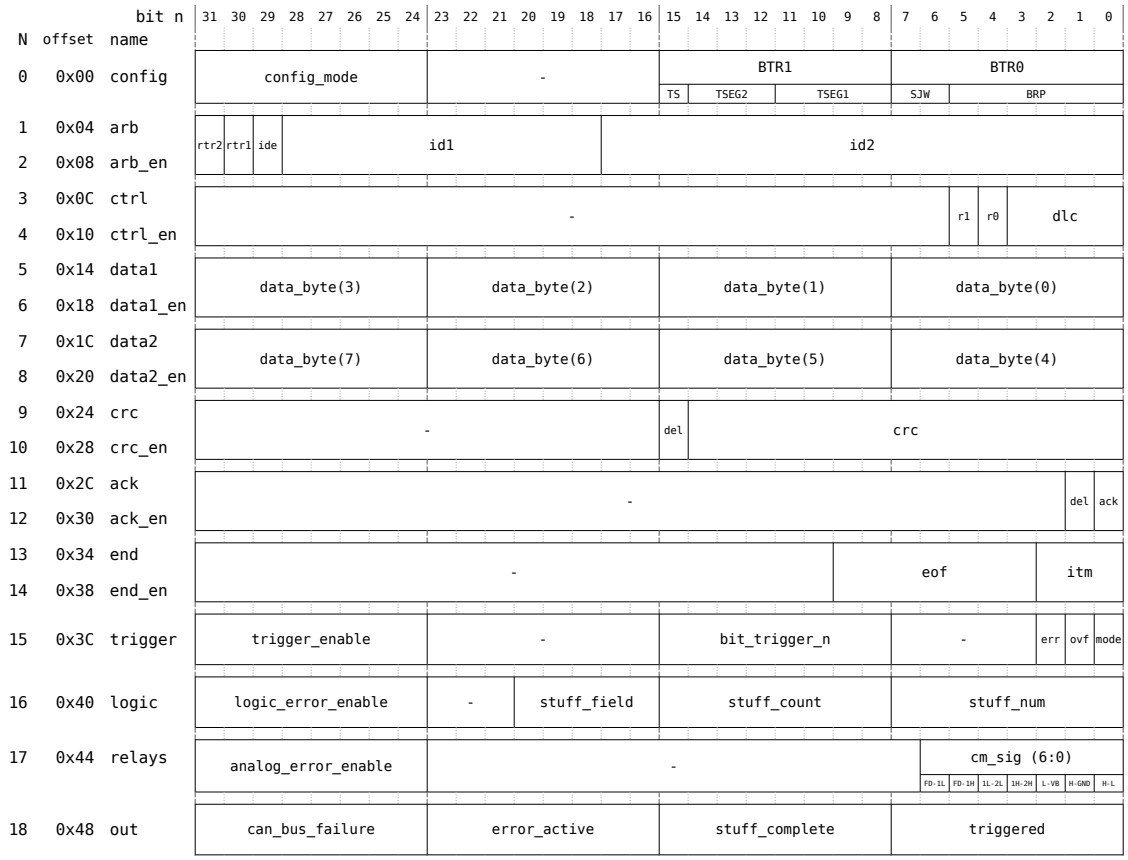

Figure 3.8: Configuration registers

Data writes are performed when `WREADY`, `WVALID`, `AWREADY` and `AWVALID` are high and only if the write address (`AWADDR`) is in the allowed range, between the `REG_START` and the `REG_WRITE_END`, thus from `0` to `17` inclusive. Each

of the 4 byte of a register can be written individually using the AXI `WSTRB` signal. Writes are directly performed to the `registers` signal, of the type `register_type` (see lines 37-38 of Listing 1), which actually implements the register file. During this write process, hardware components can directly write in the `out` register through to the `register_in` port. This avoid the usage of the AXI protocol for writes of a single register from components internal to the design, simplifying the digital design structure and preventing useless signals crowding.

Reads are performed in two VHDL processes. The `reg_data_out` signal holds the register file value at the address `ARADDR` until it is output on the `RDATA` signal (when `ARREADY` and `ARVALID` are both high).

Reads and writes to registers with AXI are performed only by the SoC processor (see section 3.2). Internal VHDL components have direct access to the lines of the register file they need for read operations and can directly write in the `out` register (line `out_addr=18`). This simplifies and speeds up the numerous I/O transactions.

3.2 ZYNQ Processing System

The ZYNQ Processing System is an Intellectual Property (IP) from Xilinx that provides a software interface around the Zynq-7000 System on Chip (SoC) integrated in the zedboard (see section 4.1 – Zedboard for more info about the hardware platform). Using the Vivado IP Integrator is possible to integrate the processing system with customized VHDL designs and embedded IP cores. It is generally used to manage (enable and disable) I/O peripherals, AXI ports, MIO, EMIO, clocking and interrupts (see Figure 3.9 for a system internal view).

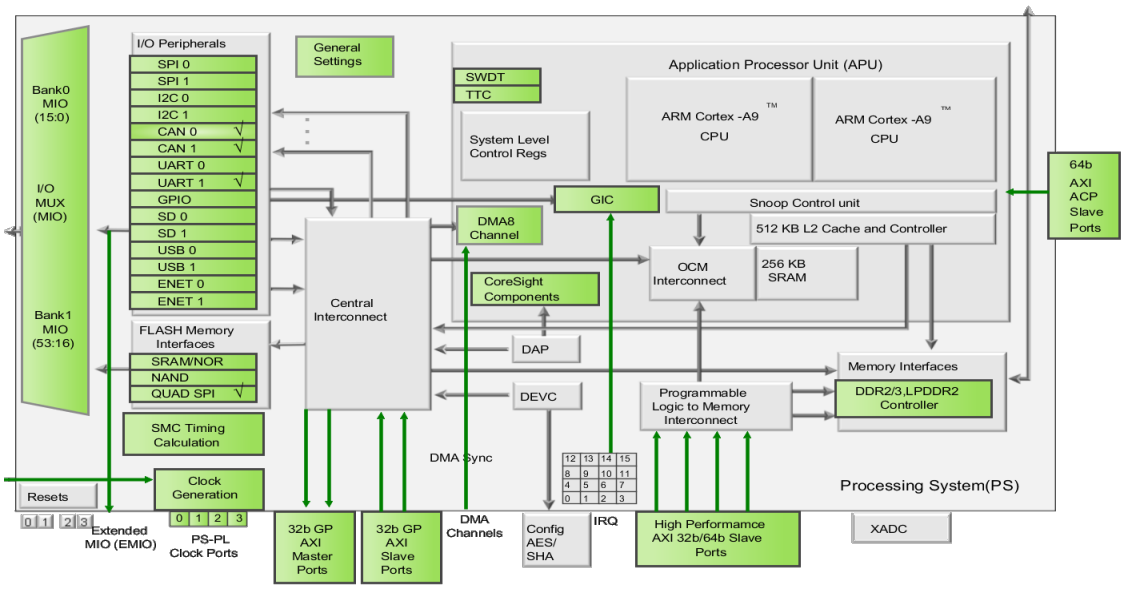

Figure 3.9: Internal view of the Block Design of the Zynq Processing System: modules highlighted are active and ready to use

In the system a quad SPI Flash memory is used to store the configuration of the Programmable Logic and the system program. Two CAN peripherals are used for testing purposes (more on that in section 4.1 – Zedboard) and a UART port to manage the serial interface with a baud rate of 115200 bit/s (see section 4.2 – Serial Interface). Clock is let to be managed by the Clocking Wizard.

3.3 AXI Interconnect

The Advanced eXtensible Interface (AXI) Interconnect is a Xilinx IP used to connect the Processing System, that acts as a master, to the Disturbance Tool, the register file in it that is the slave, with an AXI4 LITE bus in memory mapped way. Its configuration is simple and is let to be managed by the Vivado tool.

Chapter 4

Firmware

Introduction

This section will deal with the firmware deployed in the Zedboard development kit and how it interacts with the Digital Design and with the DIANA testbench with its serial interface.

4.1 Zedboard

The Zedboard is a Xilinx development kit built around a ZYNQ 7000 All Programmable System on Chip (SoC) for high-end embedded-system applications. In particular it features a ZYNQ XC7Z020-CLG484-1 SoC based on a dual ARM Cortex A9 MPCore[1] with 512 MB of RAM, 256 Mb of Quad-SPI Flash, onboard USB JTAG interface for programming and an Artix-7 Field Programmable Gate Array with 85K programmable Logic Cells.

The Quad-SPI will accommodate the Digital Design (saved as a bitstream image generated from Vivado), that will be deployed at each reset to the FPGA

[1] 32-bit processor by ARM implementing the ARMv7-A architecture.

fabric, and the firmware code that will be running on the ARM MPCore. The firmware creation, build and deployment (program of the flash memory) is performed in C using the Xilinx Software Development Kit (SDK).

Two CAN controllers are configured and used in this project mainly for testing purposes. Thanks to them is possible to simply send and receive CAN messages to test the behavior of the design. This two modules can be useful in the future.

4.2 Serial Interface

Communications between the DIANA and the Disturbance Tool take place via a serial interface on a USB cable. The only function provided in the Xilinx libraries for reading data from the UART is the `XUartPs_RecvByte` that receives a byte from the serial device in a blocking way (polling occurs on the peripheral blocking the system). This is not a problem since the system functions only upon the reception of a command.

The `main` function perform an initial setup of the peripherals and the system, and then enters an infinite (until a reset) while loop. In the loop the functions to read from serial, parse the result and execute it are executed.

4.2.1 Read Line

Since read from serial occurs one character (1 byte) at a time, the `read_line` function is in charge of reading characters from the input buffers until an enter char (ASCII code = 13) or a `LINE_MAX_LENGTH` number of chars are read. A global variable `cmd_line` is used to store the result. Only alphanumeric chars and backspace are elaborated. No special nor escape characters are managed thus the serial terminal does not allow to move the cursor or perform

advanced task. This function is therefore implementing a serial terminal with basic functionalities.

4.2.2 Parse Line

In the `parse_line` function the line read from the terminal is parsed. The `cmd_line` is split into substring separated by a space using the `strtok` function. This create the vector `args` which holds the command in the position `0` and its arguments in the following array addresses.

4.2.3 Execute

Here the command (argument zero of the vector) is compared to a constant list of possible commands called `commands_str` (defined as in Listing 2). On match the corresponding function (with the same index) in the vector `commands_func` is directly called and its result will be returned as the result of the `execute` function. If the `args` vector is empty (the user pressed enter without inserting any command) the function will return `NOERR`. If the command is not present in the list (i.e. is an invalid command) the function will return `FAILURE`.

```
175 ...
176 //List of functions pointers corresponding to each command
177 const int (*commands_func[])() = { cmd_help, cmd_configure,
    ↪  cmd_register, cmd_send };
178
179 //List of command names
180 const char *commands_str[] = {
181     "help",
182     "configure",
183     "register",
184     "send"
185 };
186 ...
```

Listing 2: Commands constant definitions from `can_test.c`

If the execution returned `SUCCESS` (the only way is that the function called by `execute` i.e. `args[0]` returns `SUCCESS`) an `OK` return value will be printed on the serial terminal. If it returned a `FAILURE`, `KO` will be printed instead. `KO` is returned for parsing errors too.

```
> help

Intecs CANstress 0.5

usage: [help] [configure]* [register] [send]** <command>

        help  <value> : shows this help.
                        use any command argument for more info
                        ↪  (e.g. > help send)

   configure  <value> : configure internal and PL CAN controllers.
                speed  : configure speed
                  sp   : configure sample point

    register  <value> : configure internal and PL CAN sample point.
                 get  : get register value
                 set  : set register value
                enable : enable register
               disable : disable register
                toggle : toggle register value

        send  <value> : send a CAN frame.
                CAN<n> : CAN device that will transmit (CAN0 or CAN1)
                         type "help send" for info about the frame setup

  * configuration of PL CAN controller can also be directly performed
   ↪  setting appropriate registers
 ** send option available only for testing purpose while internal
  ↪  controllers are implemented
```

Listing 3: Terminal output of the `help` command

A `help` command is available to display some information about the functioning of all others commands. If executed alone it returns a general text (see Listing 3), if followed by any of the other commands ('configure', 'register', 'send'), it will return information about it.

4.3 Disturbance Tool Configuration

Initial configurations of the Disturbance Tool consists on the first setup of the two CAN controllers (see section 4.4). The `CANConfig` function configures the Programmable Logic CAN (the VHDL sniffer) as well as the two external ones calculating and setting the Baud Rate Prescaler Register (BRPR) and the Bit Timing Register (BTR) based on the sample point `sp` and `speed` global configuration variables (see Table 4.1 for configuration values). The SJW value is always set to 1. The final `CONFIG_REG` value is the bitwise OR between itself (the value in the table) and the `prescaler` value and it is sent to be written in the register file at the configuration offset (line 0).

Sample point			0 (87.5%)	1 (90%)
`SEG1`			12	7
`SEG2`			1	0
`CONFIG_REG`			0x1C40	0x0740
Prescaler when speed is	0	(1000 Kbps)	4	7
	1	(500 Kbps)	9	15
	2	(250 Kbps)	19	31
	3	(125 Kbps)	39	63

Table 4.1: CAN controllers configuration parameters

The same configuration could be performed using the `configure` command, as can be seen in Listing 4. As said in the `help` command, configuration of the Programmable Logic CAN controller (the sniffer) can also be directly performed setting appropriate registers (see section 4.5) but this method is faster and prevents the entry of unsuitable values in the registers. Using the `reset` option will reset the two internal controller as well as the VHDL sniffer, imposing `sp=0` (87.5%) and `speed=0` (1000 Kbps).

```
> help configure

Intecs CANstress 0.5

usage: configure [options]

Available options:

       speed  <value> : configure internal and PL CAN controllers speed.
                 0    : 1000 Kbps (default)
                 1    : 500 Kbps
                 2    : 250 Kbps
                 3    : 125 Kbps

          sp  <value> : configure internal and PL CAN sample point.
                 0    : 87.5 % (default)
                 1    : 90 %

       reset          : reset internal CAN controllers.
```

Listing 4: Terminal output of the `help configure` command

4.4 CAN Peripherals Usage

The two CAN controllers are managed by the `XCanPs` libraries. At startup a self-test is run to verify basic sanity of the device and the driver.
In the initial configuration, the system interrupt controller is set to respond calling the interrupt handler, that determines its source and calls the appropriate callback function. It can handle events interrupts such as RX FIFO Overflow/Underflow, Bus Off status, TX FIFO full, lost arbitration and error interrupts such as ACK, bit, stuff, form and CRC errors. Only the Bus Off status will perform a reset of the controllers (otherwise they would not be usable, see subsubsection 1.4.3.9 – Fault Confinement) while other interrupts will only print to terminal a status message. Messages ant their error codes can bee seen in Listing 8 and will be treated in chapter 6 – Graphical User Interface.

```
> help send

Intecs CANstress 0.5

usage: send <can> [sorted_options]

Available options: (must be in this order)

       <can> : Select CAN controller that will send the message.
             : 0 for CAN0, 1 for CAN1

       <rtr> : Remote Transmission Request bit.
             : 0 for Data Frame, 1 for Remote Frame

       <ide> : Identifier Extension bit.
             : 0 for standard CAN, 1 for Extended CAN

        <id> : Identifier.
             : 11 bit hex value, MSB first

      <id_e> : Extended Identifier, only if ide = 1.
             : 18 bit hex value, MSB first

       <dlc> : Data Length Code.
             : int number of data bytes in payload, from 0 to 8

     <data1> : First 4 bytes of data (if data frame).
             : 32 bit hex value, MSB first

     <data2> : Second 4 bytes of data (if data frame). Used if dlc > 4.
             : 32 bit hex value, MSB first
```

Listing 5: Terminal output of the `help send` command

Messages can be sent from these peripherals with the `send` command (see Listing 5). Order of arguments matters and each CAN field can be customized. The `cmd_send` function called in the execution phase will call the `frame_set` function that is in charge of reading the arguments and composing the CAN message to be sent (`TxFrame`). Here each argument is controlled to be compliant with the CAN protocol and an error will be returned if it is not. In the end the `SendFrame` function will send the frame (a status message will be printed on serial terminal) calling the `XCanPs_Send` and catching any

errors.

A receive handler (`RecvHandler`) will be called if a controller recognize a new CAN message on the bus, if is not the controller that is transmitting the message. No filtering is active so any new message will be detected. A status message will be printed on the serial terminal stating the info about the incoming frame. Errors on read are reported too.

4.5 Registers Usage

The last serial command, called `register`, allows to read and write the registers activating the Disturbance Tool functions (see Listing 7 for all command options).

```
59 ...
60 /* registers offset address */
61 #define CONFIG_OFFSET       0x00
62 #define ARB_OFFSET          0x04
63 #define ARB_EN_OFFSET       0x08
64 #define CTRL_OFFSET         0x0C
65 #define CTRL_EN_OFFSET      0x10
66 #define DATA1_OFFSET        0x14
67 #define DATA1_EN_OFFSET     0x18
68 #define DATA2_OFFSET        0x1C
69 #define DATA2_EN_OFFSET     0x20
70 #define CRC_OFFSET          0x24
71 #define CRC_EN_OFFSET       0x28
72 #define ACK_OFFSET          0x2C
73 #define ACK_EN_OFFSET       0x30
74 #define END_OFFSET          0x34
75 #define END_EN_OFFSET       0x38
76 #define TRIGGER_OFFSET      0x3C
77 #define LOGIC_OFFSET        0x40
78 #define RELAYS_OFFSET       0x44
79 #define OUT_OFFSET          0x48
80 ...
```

Listing 6: Offset constant definitions from `can_test.c`

```
> help register

Intecs CANstress 0.5

usage: register [options]

Available options:

       get  <addr>  : get value of register at specified address offset.
           reg_addr : register offset, from 0x00 to 0x48, 32bit aligned
            ↪  (multiple of 4)

       set  <a> <v> : set value of register at specified address offset.
         reg_addr   : register offset, from 0x00 to 0x44, 32bit aligned
          ↪  (multiple of 4)
          reg_value : 32 bit hexadecimal value to be put into register

    enable  <addr>  : enable configuration register.
        reg_addr_en : register offset enable, from 0x3F to 0x47, 32bit
        ↪  aligned (multiple of 4)

   disable  <addr>  : disable configuration register.
        reg_addr_en : register offset enable, from 0x3F to 0x47, 32bit
        ↪  aligned (multiple of 4)

    toggle  <addr>  : toggle configuration register.
        reg_addr_en : register offset enable, from 0x3F to 0x47, 32bit
        ↪  aligned (multiple of 4)
```

Listing 7: Terminal output of the `help register` command

With the `get` and `set` options is possible to respectively read and write the registers using as address the offsets like reported in Listing 6. Validity checks on the address and value are performed. Register line at `OUT_OFFSET` can only be read as designed. Operation on registers are performed through the AXI bus.

The `enable`, `disable` and `toggle` options are only available as shortcut for setting or toggling the higher part of the register as it is in most case the activation byte for a function (see subsection 3.1.5 – Configuration Registers for register partitioning).

Chapter 5

Analog design

Introduction

This chapter will cover the design of the hardware that is in charge of managing the Zedboard I/Os and connecting them to the ECU and to the DIANA testbench. While the schematics were designed to be used in a real word scenario, the PCB was designed only to perform tests in the initial phase of this project (see chapter 7 – Verification).

The schematics and Printed Circuit Board (PCB) have been designed with the KiCad Electronics Design Automation Suite in a Linux environment.

5.1 Schematics

Schematics design was performed by Intecs Solutions and revised for the scope of this thesis. Have been redrawn on the KiCad schematic editor (Eeschema). Will not be available for consultation on this document.

Two CAN transceivers, one for the standard CAN and one for the CAN FD, are in charge of reading messages for the sniffer functionalities and writing

dominant bits (stuffing bits) for the generation of logic errors. They connect to the bus in a mutually exclusive way thanks to the use of two Single Pole Double Throw (SPDT) relays (one for the CANH line and one for the CANL line), activated by the `cm_sig(5)` and `cm_sig(6)` signals.

A relay network receive all `cm_sig(0:4)` to perform analog errors. Five Single Pole Single Throw (SPST) relays accomplish the connection of a CAN BUS line to a logic level (or another CAN line) and the other two relays are the ones mentioned before. Each relay is driven by an NPN transistor that converts the 3.3V levels of the Zedboard into a 5V level and provides the load (the relay coil) enough current. A flyback diode is used to eliminate any voltage spike on the coil (inductive load).

The trigger generation is managed by another NPN transistor on a 5V supply and a low-pass filter to reduce noise disturbance.

CANstress external board **Rev: 1.0**

Intecs Solutions S.p.A 2019-06-05

	References	Value	Footprint	Quantity
1	C3, C4, C5, C6, C7, C8, C9, C10, C11, C12, C13, C14, C15, C16, C17, C18, C19, C20, C21, C22, C24, C25, C26, C28, C29, C30, C31, C32	10nF	C_Disc_D5.0mm_W2.5mm_P2.50mm	28
2	C1, C2, C23	100nF	C_Disc_D5.0mm_W2.5mm_P2.50mm	3
3	C27	1nF	C_Disc_D5.0mm_W2.5mm_P2.50mm	1
4	R4, R5, R6, R8, R17, R18, R19, R22	10K	R_Axial_DIN0411_L9.9mm_D3.6mm_P12.70mm_Horizontal	8
5	R2, R3, R7, R15, R16	866	R_Axial_DIN0411_L9.9mm_D3.6mm_P12.70mm_Horizontal	5
6	R1, R14	511	R_Axial_DIN0411_L9.9mm_D3.6mm_P12.70mm_Horizontal	2
7	R10, R11	3.3k	R_Axial_DIN0411_L9.9mm_D3.6mm_P12.70mm_Horizontal	2
8	R20, R21	1K	R_Axial_DIN0411_L9.9mm_D3.6mm_P12.70mm_Horizontal	2
9	R12, R13	500	R_Axial_DIN0414_L11.9mm_D4.5mm_P20.32mm_Horizontal	2
10	R9	1K	R_Axial_DIN0414_L11.9mm_D4.5mm_P20.32mm_Horizontal	1
11	D1, D2, D3, D4, D5, D6, D7	1N4001	D_DO-41_SOD81_P7.62mm_Horizontal	7
12	U1	TCAN330GDR	TCAN330GDR_on_SOIC-8-DIP	1
13	U2	TJA1055T/3/C,518	TJA1055T_on_SOIC-14-DIP	1
14	K1, K5	TSC-105D3H,000	Relay_SPDT_HsinDa_Y14	2
15	K2, K3, K4, K6, K7	G5NB-1A4_DC5	Relay_SPST_Omron-G5NB-1A4-DC5	5
16	Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8	PN2222A	TO-92_Inline	8
17	J1, J2, J3, J4, J5, J6	ST_01x02	Altech_AK300_1x02_P5.00mm_45-Degree	6
18	J7, J8, J9	Conn_01x08	PinHeader_1x08_P2.54mm_Vertical	3

Figure 5.1: Bill of material for the purchase request

5.2 Printed circuit board

The Printed Circuit Board (PCB) have been designed from scratch starting from the schematics and using the KiCad PCB layout editor (PcbNew).

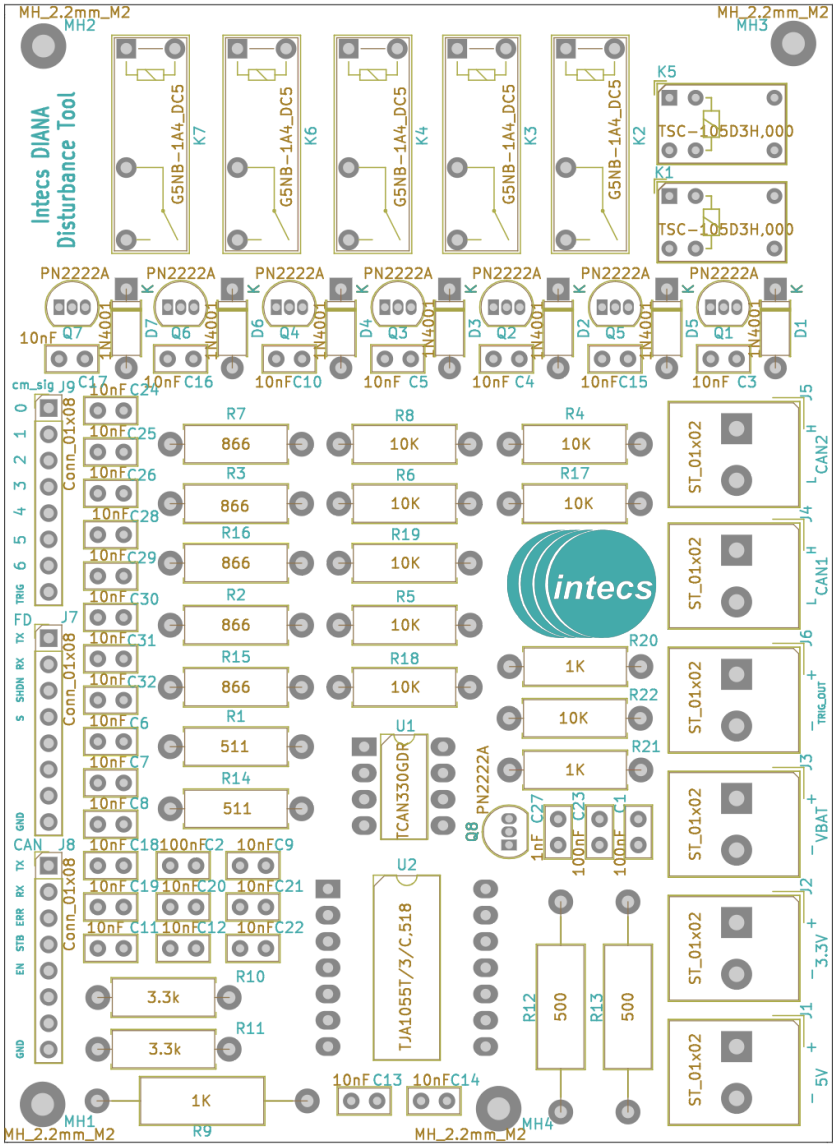

Figure 5.2: PCB front view with components footprint and silk layer

For the ease of component mounting and soldering, the vast majority of footprints have been chosen of the Trough Hole Technology (THT) type. The two

transceivers (U1 and U2 in Figure 5.2) are the only components available in a small Surface Mount Device (SMD) footprint and will be placed with the help of a SOIC to DIP adapter.

PCB tracks were optimized (during the place and routing phase) for the CAN differential lines, trying to keep them as close as possible and of the same global length in order to reduce disturbance.

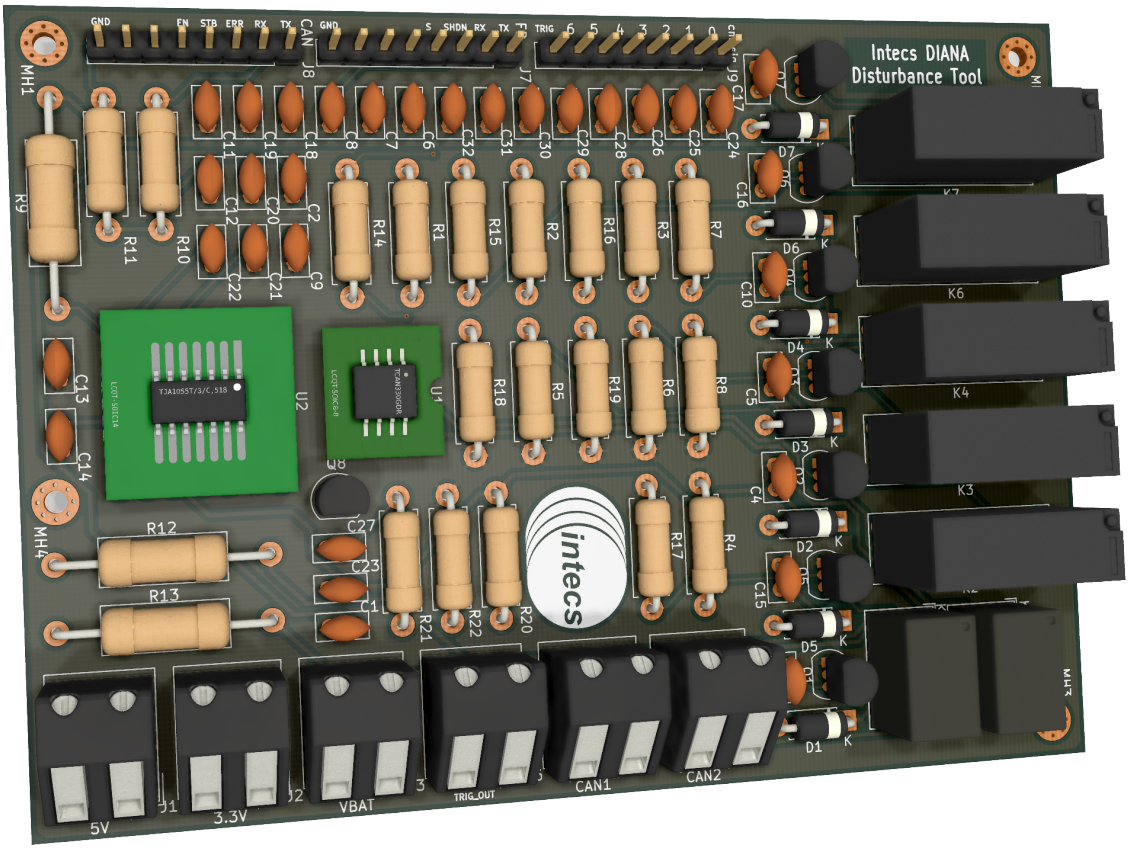

Figure 5.3: 3D render of the Disturbance Tool board

The PCB is ready to be manufactured for testing purposes but the arrival of the components is not foreseen before the end of this thesis, which is why it will not be possible to carry out tests on this design. A realistic 3D image have been rendered to give the idea of the finished product (see Figure 5.3).

Chapter 6

Graphical User Interface

Introduction

The Graphical User Interface (GUI) have been designed in order to simplify testing procedures in absence of the DIANA testbench. Instead of sending commands through the serial terminal, a graphical interface permits to modify the Disturbance Tools settings easily and quickly.

The design took place in a Linux environment with the GTK toolkit but it is compatible with both Linux and Windows Operating Systems. Compiled on Linux with GCC, on Windows with MSYS2. The use of the Glade Rapid Application Development (RAD) tool simplifies the graphic interface design process.

6.1 Serial Interface

The `libserialport` is the library used for managing serial ports in the application. It is written in C, is cross-platform and it was chosen because it is compatible with both Linux and Windows. Is distributed under a GNU LGPL license.

A set of return values have been defined (in the firmware) to agree on the transmission status of a message between the application and the firmware on the Zedboard. The `"OK"` status is returned when the command sent on serial has been successfully executed. The `"KO"` message is a general message signalling a failure on the execution of the received command. All the other messages in a `'K<val>'` format are error messages and in an `'O<val>'` format are success messages. They are codified as can be seen in Listing 8, with `res_val` as a list of possible response values and `res_val_ext` as a list of descriptions with corresponding indexes.

```
80 ...
81 const char * res_val_ext[] = {"OK", "ERR",
82     "Error while receiving a frame",
83     "Frame received correctly",
84     "Frame sent correctly",
85     "ACK error",
86     "BIT error",
87     "STUFF error",
88     "FORM error",
89     "CRC error",
90     "Error: entering bus off state",
91     "Lost ARBitration error"};
92 const char * res_val[] = {"OK", "KO", "KREC", "OREC", "OSEN", "KACK",
93     "KBIT", "KSTU", "KFOR", "KCRC", "KBOF", "KLAR"};
94 ...
```

Listing 8: Serial return values from `CANstress_gui.h`

The `serial_write` function is in charge of sending commands to the device and reading the response. If errors occurs on writing, the function will be re-called for a `WR_ERR_MAX` number of times after which, if still not writing, the application will disconnect the device returning a write error. On successful writes, the response is read and parsed, searching for error messages as defined in Listing 8. If reads fails for a `TO_ERR_MAX` number of times, the application will disconnect the device returning a timeout error.

The `serial_check` function will be called before each write operation on the

serial device to check if the device is still connected to the pc and responding to commands.

6.2 Graphical Interface

The interface is divided in several windows that will be treated down below. If the application is launched from a terminal, debug messages (such as extended messages on error, full serial communication messages etc.) will be available for debugging purposes.

6.2.1 Connection Window

At startup the application loads the connection window (see Figure 6.1), scanning for serial devices connected to the pc and displaying a list of them calling the `serial_update` function.

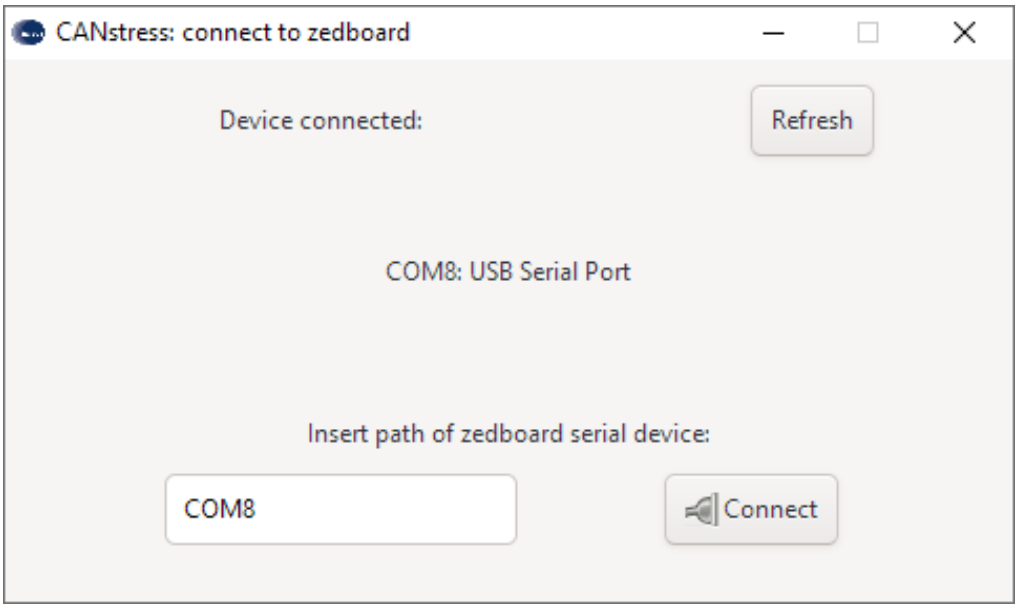

Figure 6.1: Screen of the application connection window

To get the list of serial devices connected to the pc, the `serial_update` function uses the `libserialport` library if on a Linux client or a custom made `win_get_devices` function that enumerates COM ports using SetupAPI on Windows.

The first device of the list is proposed as device to be connected to (can be manually changed). When clicking the connect button, the `serial_connect` function will be called checking if the name of the device entered is valid (the device exists) and trying to connect to if. On success the connection window will be hidden and the main window will show up.

6.2.2 Main Window

The main window have a menu bar with four options with a drop down menu:

- **File** – allow to reconnect to the serial device (loading the connection window) and to exit from the application. In future version will permit to load configuration files.
- **Configure** – give the option to configure the CAN speed, sample point and to reset them, as discussed in section 4.3 – Disturbance Tool Configuration, using the commands shown in Listing 4.
- **View** – has two check boxes for enabling and disabling the "Debug" perspective and tho show and hide the "Send window" (see subsection 6.2.3).
- **Help** – show the possibility to display the "About" window (see subsection 6.2.4).

A stack with three options permits to switch between the "trigger", "stuff" and "analog" application tabs. To the right three icons will show the status of the three aforementioned functions.

6.2.2.1 Trigger

The trigger tab allows to easily setup the Disturbance Tool to generate a trigger. Each section of each field could be set with a boolean value `"0"` or `"1"` or a "don't care" value `"X"`. This last one will set the mask registers to 0.

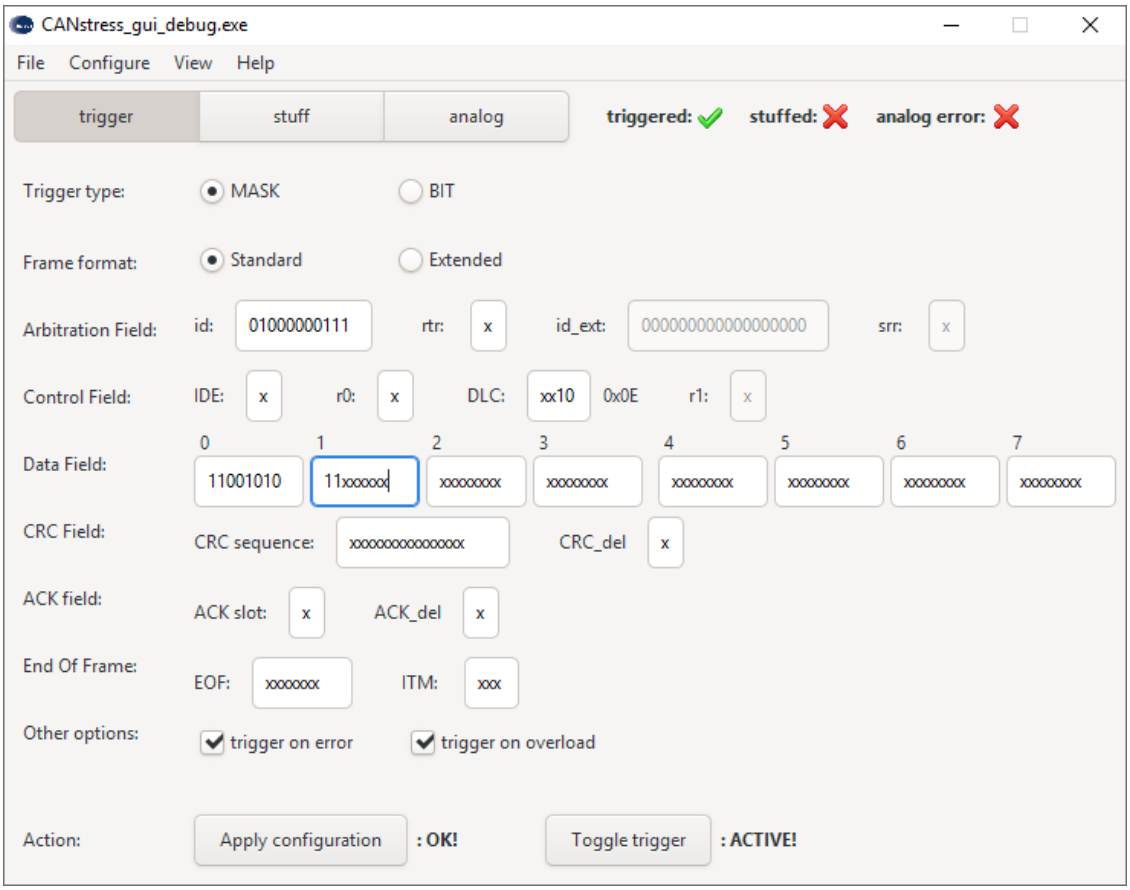

Figure 6.2: Screen of the application main window, trigger tab with configuration for test as in subsubsection 7.2.2.2 – Trigger Tests

Applying the configuration will only set all the registers values into the design. The trigger function can be activated clicking on the "Toggle trigger" button. These two buttons will display a status message on their right side. If the debug perspective is active, values of all registers will be shown on the right side of the application (in Figure 6.2 the dubug perspective is disabled for spacing issue). Enabling the BIT triggering will hide all CAN fields and

will show the “bit n” field.

6.2.2.2 Stuff

The stuff tab allows to set the stuff bit number and repetitions and the field after which stuffing should take place. Apply and Toggle buttons functions as said above. In Figure 6.3 the debug perspective is active and shows the value of the `logic` and the `out` registers lines.

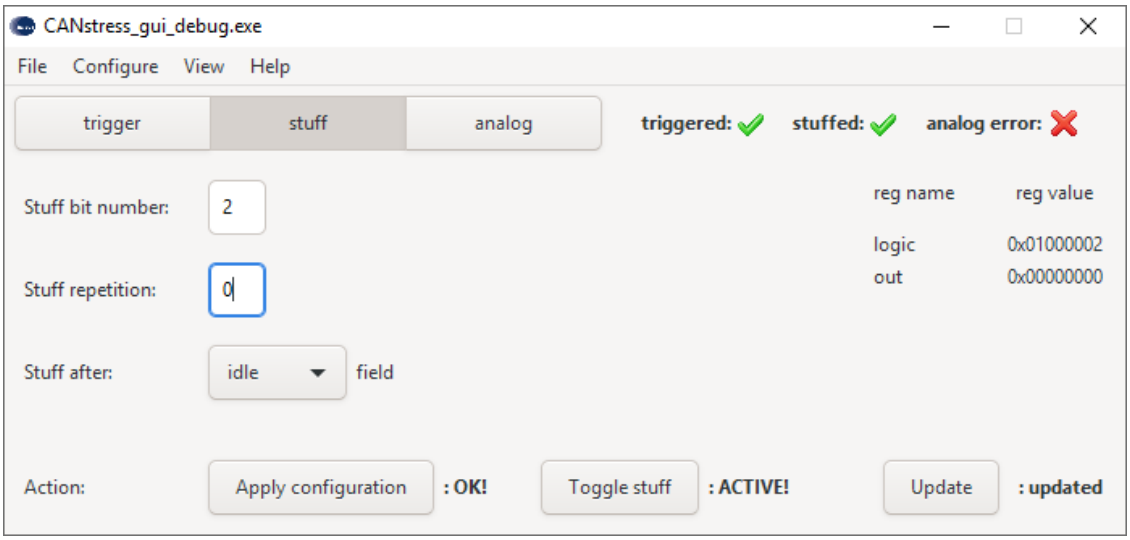

Figure 6.3: Screen of the application main window, stuff tab (debug view active) with configuration for test as in subsubsection 7.2.2.3 – Logic Error Tests

6.2.2.3 Analog

The analog tab provides three check boxes for the shorting and two for the open circuit errors (see Figure 6.4). Apply and Toggle buttons functions as said above. The debug perspective will show the `relays` and the `out` registers lines.

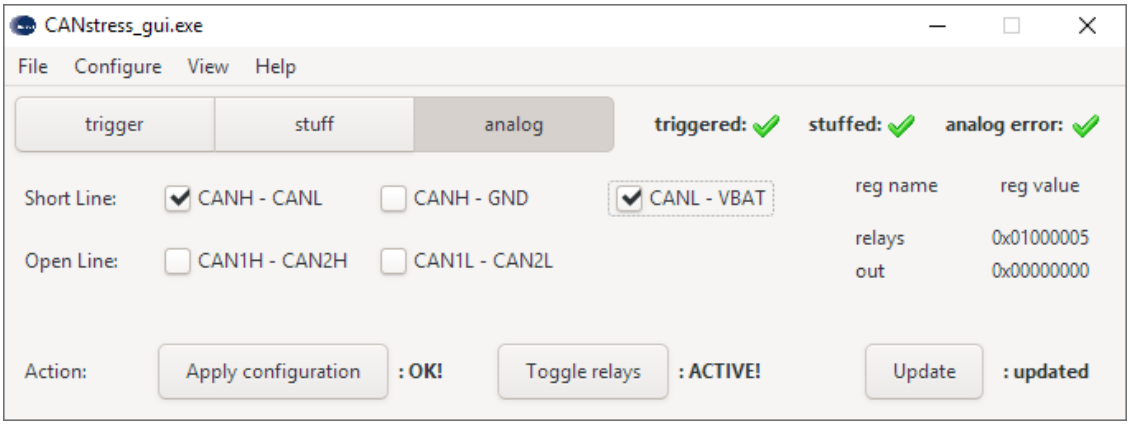

Figure 6.4: Screen of the application main window, analog tab (debug view active) with analog error activated (trigger and stuff are active too).

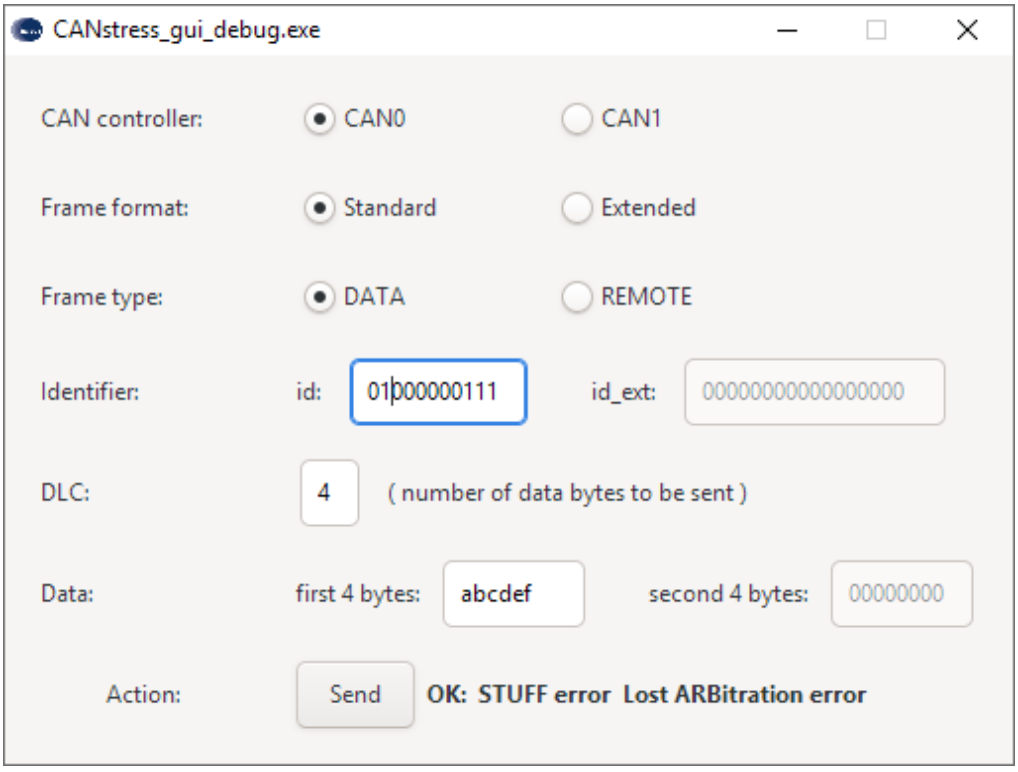

Figure 6.5: Screen of the application send window, with configuration for test as in subsubsection 7.2.2.3 – Logic Error Tests. Message has been sent correctly (OK), Stuffing error and Lost Arbitration error are reported

6.2.3 Send Window

The send window allows to send CAN messages choosing between the "CAN0" and "CAN1" as transmitting controllers. Both data and remote frames can be sent and standard and extended CAN formats can be chosen. Send status is reported rigth to the Send button. Eventual errors are reported too. See Figure 6.5 for more details.

6.2.4 About Window

The about window (Figure 6.6) shows information about the GUI application version, copyrights and credits.

Figure 6.6: Screen of the application about window

Chapter 7

Verification

Introduction

Verification have been performed in two phases: on the digital design at an early stage of the project and on the whole design, including hardware (not the PCB), when the project was complete.

7.1 Behavioral Verification

Behavioral verification have been performed on Vivado as a simulation of components behavior. Each of the five components of the digital design have been tested separately and will be treated below.

7.1.1 Sniffer Testbench

The sniffer was the first module to be developed and the most complex to be tested separately. A procedure called `send_frame` is used to receive a stream of bits (the frame to be sent) and convert it into a CAN messages, taking into accounts stuffing bits and interframe space.

In Figure 7.1 an example waveform of a CAN message is shown at a transmission speed of 1000 Kbit/s. The `new_frame` signal rises at the detection of the SOF bit and activates the bit counter (`bit_n` signal) and the other functions. Even if the bit counter value (`bit_n`) is not always readable in the figure, we can see that it does not update when a stuffing bit is detected on the bus. When the `field_ready` signal rises, the previous field on the bus is finished and a new field begins. The value of the previous field is available in the `field_value` signal until the end of the current (next) field. For example when the field is 9 (data field) we can see that the field value is 1 because it refers to the previous field, the DLC, and its value is correct, since only one byte of data is being transmitted.

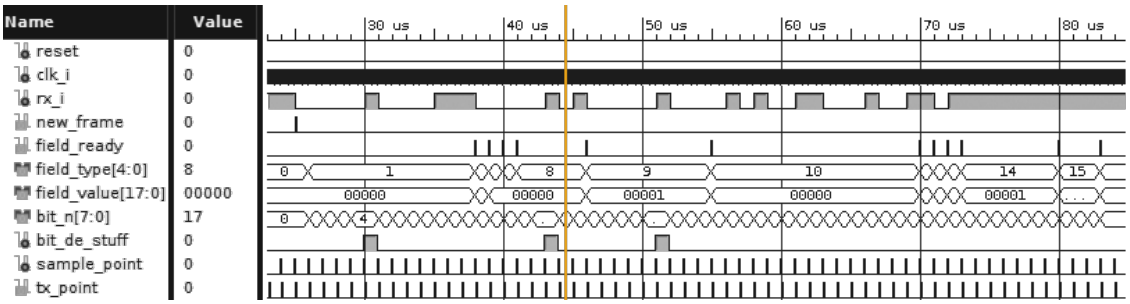

Figure 7.1: Waveform from sniffer testbench: ID=0x07, data=0x00, crc=0x2989

Some test cases have been chosen trying to send different frames (standard and extended, data and remote) with different payloads and data length at different CAN speeds. Some tests were performed causing CRC errors, ACK errors and generating overload and error frames. Messages were sent out of sync (waiting a time different than a period) to check the hard sync and soft sync capabilities. All tests passed successfully.

7.1.2 Trigger Testbench

The testbench of the trigger logic includes the sniffer too. It is necessary to generate all the signals needed for the correct functioning of the trigger logic. A `registers_forced` signal of type `register_type` is used to emulate the

register file, not used in this testbench.

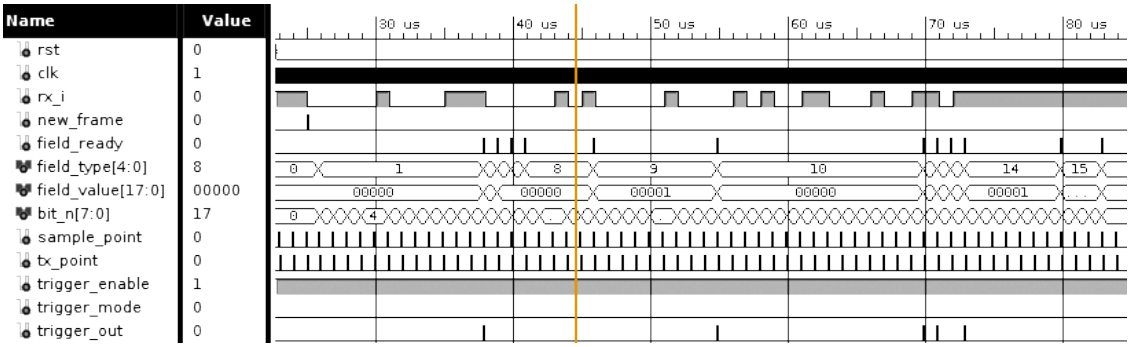

Figure 7.2: Waveform from trigger testbench: ID=0x07, data=0x00, crc=0x2989

```
209  ...
210  -- mask_trigger, forcing register value
211  registers_forced(arb_addr)      <= x"001c0000";  -- id1 match
212  registers_forced(arb_addr+1)    <= x"1ffc0000";  -- id1 masked
213  registers_forced(ctrl_addr)     <= x"00000000";
214  registers_forced(ctrl_addr+1)   <= x"00000000";
215  registers_forced(data_addr1)    <= x"00000000";  -- data match
216  registers_forced(data_addr1+1)  <= x"000000ff";  -- data mask enable
217  registers_forced(data_addr2)    <= x"00000000";
218  registers_forced(data_addr2+1)  <= x"00000000";
219  registers_forced(crc_addr)      <= x"0000a989";  -- crc and del match
220  registers_forced(crc_addr+1)    <= x"0000ffff";  -- crc and del enable
221  registers_forced(ack_addr)      <= x"00000003";  -- ack and del match
222  registers_forced(ack_addr+1)    <= x"00000003";  -- ack and del enable
223  registers_forced(end_addr)      <= x"00000000";
224  registers_forced(end_addr+1)    <= x"00000000";
225  ...
```

Listing 9: Line of code in `trigger_logic_tb.vhd` for register configuration for the output in Figure 7.2

In Figure 7.2 the same frame of the figure in the sniffer testbench is sent. Before that, registers have been configured as in Listing 9 to trigger on id1 match at value 0x07, data match with value 0x00, CRC match with value 0x2989 and CRC delimiter match (the value in register is different because it includes the 1 of the del value), trigger on ACK bit and ACK delimiter bit. From the figure we can see that the `trigger_out` is actually generated

at the end of the field of type 1 (id1), 9 (data), 10 (CRC), 11 (CRC del) and 13 (ACK del). The trigger is not generated on the ACK bit (12) because the trigger value was set to 1 but the actual value was 0.

Other test cases have been chosen trying a wide variety of different values for the various field of different CAN frames (data, remote, extended etc). Trigger have been tested also on erroneous CAN messages that generates error frames or overload frames. Trigger at a bit number have been tested too. All behavioral test were performed successfully.

7.1.3 Logic Error Testbench

The testbench for the logic error generator module includes the sniffer component for the generation of some useful signals. The `send_frame` procedure (used in the Sniffer Testbench) is used to send CAN messages on the bus (signal `tx`) emulating a transmitting controller. The logic error module transmits on the signal `tx_stuff`. Another procedure called `error_check` is in charge of reading the bus and checking the CAN stuffing rule (see subsubsection 1.4.3.7 – Error Detection and Signalling for errors info). If stuffing rule is violated, it generates an error frame on the `tx_err` signal, acting like a controller. The `error` signal is high for the whole duration of the error frame. The final transmission on the bus will be the combination (logic or, due to CAN specifications) of three values: the signal of the transmitting controller (`tx`), the dominant bits eventually injected by the logic error module and a third controller generating the error frame. This resulting signal is called `can_bus`.

In Figure 7.3 a behavioral simulation of a test case is shown. The module has been configured to generate 8 stuffing bits (no repetitions) after the field 1 (id1). The configuration register is therefore written at its `logic` address with the value `0x00010008` (see subsection 3.1.5 – Configuration Registers for register info). After the module activation a frame is sent on the CAN

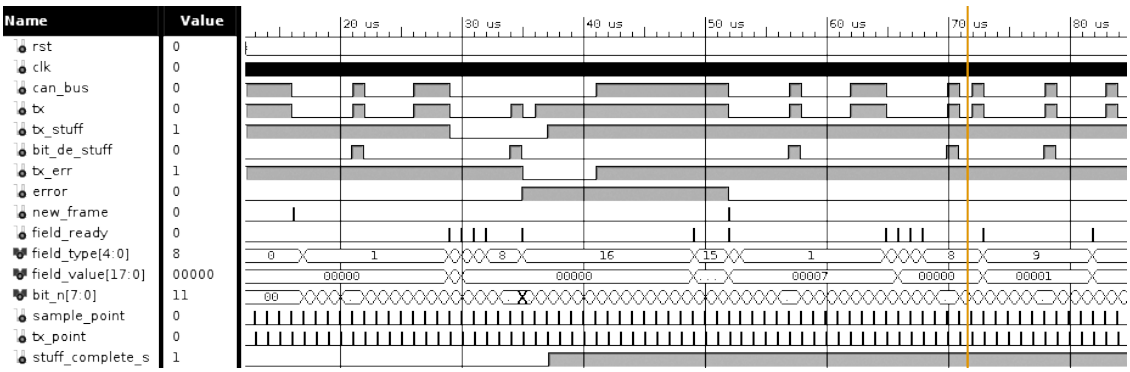

Figure 7.3: Waveform from logic error testbench with error generated after id1

BUS. Stuffing begins after the first field (stuffing from the rtr1) and continues until the first stuffing bit is not generated as recessive. Here the controller reading the bus (`error_check` procedure) acknowledge the stuff error and starts transmitting the error frame. The sniffer notices the error too (`field_type=16`, error frame). As soon as the stuff finishes, even if the stuffing bits have become bits of the error frame, the logic error module signals the end of the stuffing procedure raising the `stuff_complete` signal which will remain high until a new configuration (de-activation and activation) of the module. When the error message finishes, the transmitting controller retries the transmission of the previous message on the bus, as it would do a real controller.

This testing procedure have been used for testing the error generation (and the behavior of the sniffer at error reception). Tests on stuffing different CAN messages at different positions (field) with different stuff bit length and repetitions have been performed to ensure the correct behavior of the digital design.

7.1.4 Register Test

Register verification have been performed on a separated Vivado Block Design that can be seen in Figure 7.4. The design includes a Zynq processing system with a similar configuration to the one in the digital design (see section 3.2 – ZYNQ Processing System). The `config_registers` is the register file component used in the digital design (see subsection 3.1.5 – Configuration Registers) and is the device under test.

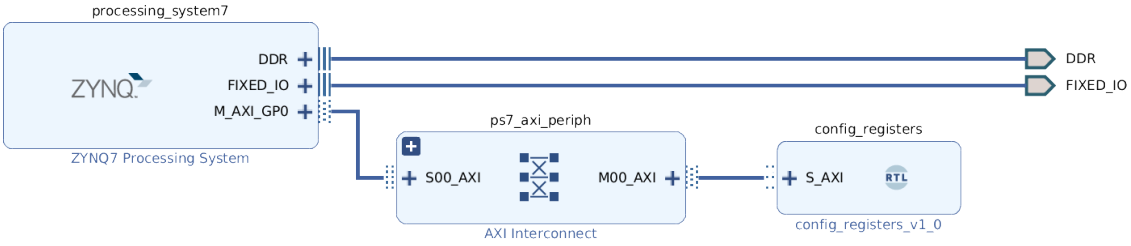

Figure 7.4: Block Design of the configuration register digital design

To test the correctness of the AXI4 Lite protocol implementation, the design have been synthesized and uploaded on the Zedboard with a test firmware. The firmware is in charge of writing different values in the register at different address offset and then reading them back to check the consistency. Tests have been performed with different system clock speeds and writing to the read-only register have been tested too.

For the purpose of this test the functions used to read and write a register (`get_register` and `set_register`) and to enable and disable it (write 1 or 0 in the first byte) have been developed and are used in the final version of the firmware.

7.2 Hardware Verification

Hardware verification have been performed using the two CAN controllers integrated in the firmware design (included specifically for this purpose, see Figure 7.5), since it was not possible to prepare a complete test platform with the DIANA and an ECU (device under test).

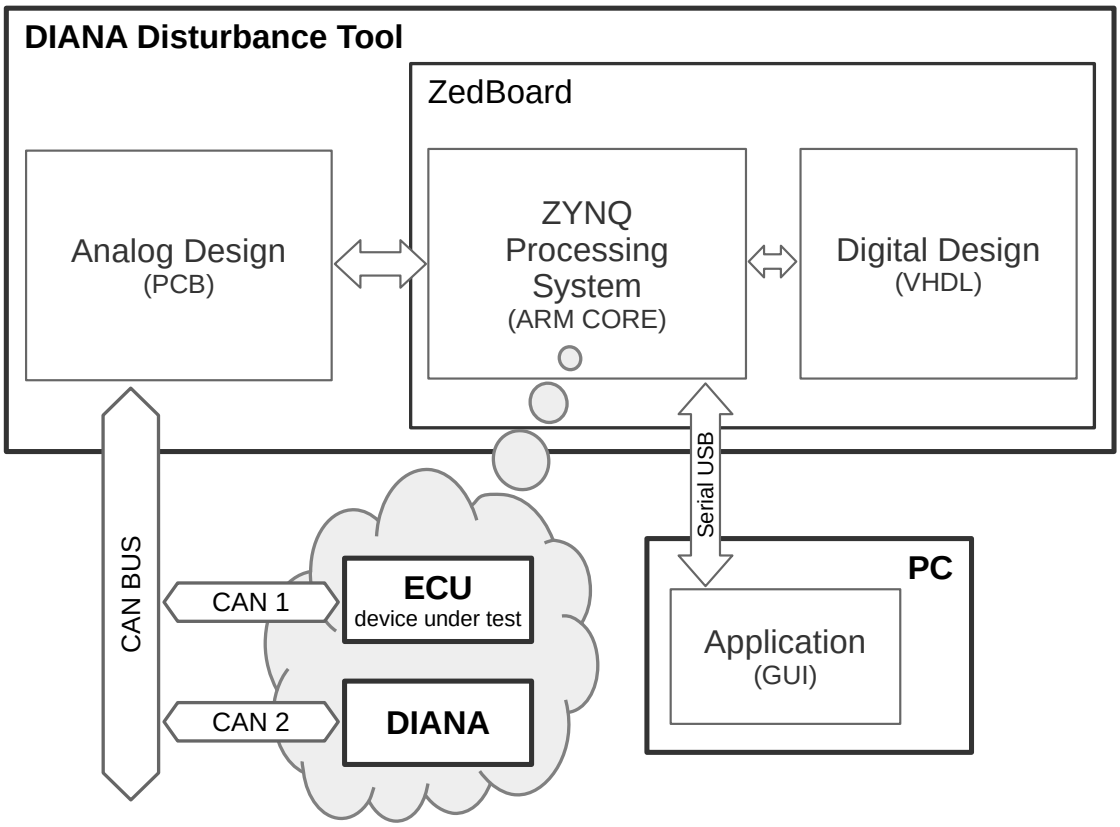

Figure 7.5: CAN controllers inside the Zedboard

Messages are sent from the application (serial commands could be used too) and will tell the integrated controller to generate the appropriate CAN frame on the Zedboard dedicated output pins. Three CAN transceiver boards (based on a TJA1051 chip) are used to interconnect the two controllers and the sniffer to the CAN BUS as can bee seen in Figure 7.6.

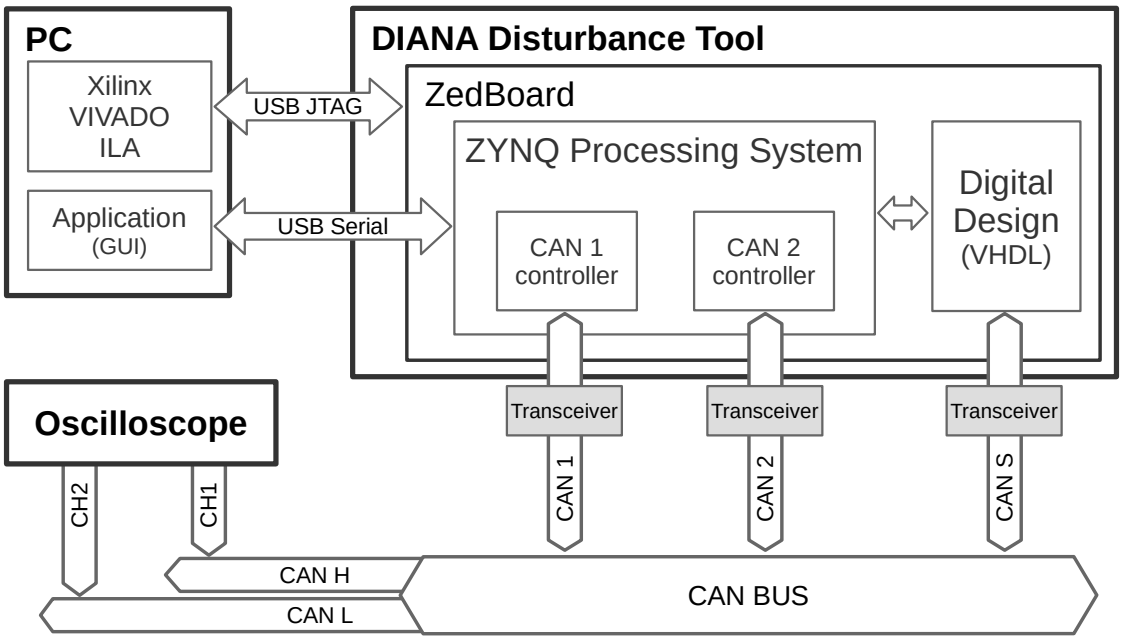

Figure 7.6: System configuration for Hardware testing with external transceiver boards

7.2.1 Integrated Logic Analyzer

Messages and some other digital design signals can be tracked in the Vivado application by means of the Xilinx Integrated Logic Analyzer (ILA). It is an IP, inserted in the design at the synthesis stage, that acts like a logic analyzer. It allows to monitor internal signals otherwise not visible and its activation can be triggered by a signal equation or edge transition.

Introducing the logic analyzer in the design, the number of FPGA LUTs increased from 1 713 to 8 235, not a problem for the Zynq 7020 that have 53 200 LUTs, with an utilization factor of 15%. The current settings allow to start the monitoring when a new frame is detected (rising edge of the `new_frame` signal) and to access some signals (such as the sample and tx points, field ready, type and values, etc) for debugging purposes. Choosing a sample data depth of $2^{16} = 65\,536$ for the ILA, the utilization of the Programmable Logic

block ram (BRAM) reaches the 71%. This means that increasing the data depth to the next step ($2^{17} = 131\,072$) is not possible since there are not enough BRAM (reducing the number of monitored signals could be a solution but it would reduce the visibility within the design).

Because the ILA is synchronous to the design and the system clock is chosen at 160 MHz, the monitoring bus time[1] t_{ILA} is of:

$$t_{ILA} = data\_depth \times \frac{1}{f_{clk}} = \frac{2^{16}}{160\ MHz} \approx 409{,}6\ \mu s \tag{7.1}$$

With a maximum CAN frame length of 128 bits for an extended message[2] and taking into account (in the worst case) a maximum number of 26 stuffing bits and 3 interframe spacing bits, the maximum Data Rate (DR) for different speeds would be:

$$DR_{speed} \geq \frac{speed\ (Kbit/s)}{max\_num\_of\_bits\_in\_frame} = \frac{speed\ (Kbit/s)}{128 + 26 + 3}$$

$$\begin{aligned}
DR_{125} &\geq \frac{125\ Kbit/s}{157} \approx 796\ frames/s & t_{fr_{125}} &\approx 1256\ \mu s \\
DR_{250} &\geq \frac{250\ Kbit/s}{157} \approx 1592\ frames/s & t_{fr_{250}} &\approx 628\ \mu s \\
DR_{500} &\geq \frac{500\ Kbit/s}{157} \approx 3184\ frames/s & t_{fr_{500}} &\approx 314\ \mu s \\
DR_{1000} &\geq \frac{1000\ Kbit/s}{157} \approx 6369\ frames/s & t_{fr_{1000}} &\approx 157\ \mu s
\end{aligned} \tag{7.2}$$

[1] The time that elapses between the trigger (if chosen at the beginning of the window) until the memory is completely full.

[2] The maximum length is calculated, for an extended message with 8 bytes of data, summing the number of bits field by field as 1+11+1+1+18+1+2+4+64+15+1+1+1+7 = 128 bits.

With their inverse being the maximum time that a frame takes to be transferred (t_{fr}) for different speeds. Being the ILA monitoring time of 409,6μs, using speeds of 125 $Kbit/s$ and 250 $Kbit/s$ does not allow to see the entire CAN frame if it is too long while using 1000 $Kbit/s$ permits to see on average three long frames (2,6 in the worst case). That is why the maximum speed is used for the majority of the tests performed.

7.2.2 Hardware Tests

Tests with hardware have been performed with test cases similar to the behavioral ones. The use of an oscilloscope connected to both CANH and CANL signals confirmed the correct functioning of the ILA and allows to see the analog CAN BUS waveforms (connection diagram in Figure 7.6).

Down below are reported some test cases for the different digital design modules. Other tests have been performed with extended messages, multiple data values and length, remote message and various other configurations, trying different modules at the same time.

7.2.2.1 Sniffer Tests

Sniffer have been tested sending various CAN messages and checking their correct reception and analysis (retrieving of the field values) with the help of the ILA and the oscilloscope. The integrated CAN controllers have been tested with this method at the same time.

As an example, a simple CAN message can be seen in Figure 7.7 (is the same example of the sniffer and trigger behavioral testbench). Here is shown how the two integrated CAN controllers work. The CAN0 device is the one transmitting on the BUS and both the CAN1 and the sniffer (signal `rx_i`) are receiving the information. In the ACK slot CAN0 is transmitting a recessive

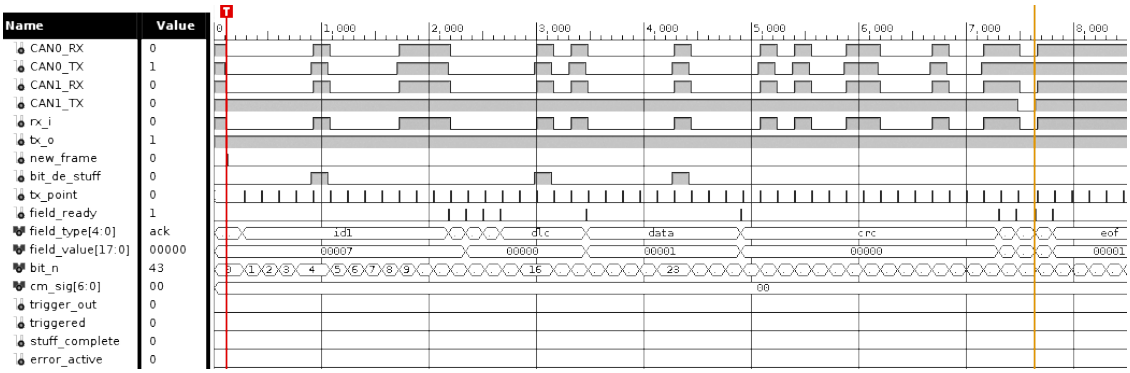

Figure 7.7: Waveform from ILA, transmission of a CAN message.

bit while CAN1 is overwriting with a dominant value, resulting a 0 in all RX signals, communicating to the transmitting controller the correct reception of the message.

7.2.2.2 Trigger Tests

The trigger module have been tested trying the generation of triggers with different masks in every CAN field. Bit triggering have also been tested at different positions.

As a test case for testing trigger functionalities, the application have been setup with the following settings (see Figure 6.2 for a view of the application window):

- `id1`: `01000000111`
- `DLC`: `xx10`
- `data0(0)`: `11001010` (hex `0xCA`)
- `data0(1)`: `11xxxxxx`

The application debug terminal returned the registers configuration values that can be seen in Listing 10, confirming the correct application of values and masks.

```
Writing configuration:
arb:    0x081C0000, arb_m:      0x1FFC0000.
ctrl:   0x00000002, ctrl_m:     0xFFFFFFC3.
data0:  0x0000C0CA, data0_m:    0x0000C0FF.
data1:  0x00000000, data1_m:    0x00000000.
crc:    0x00000000, crc_m:      0xFFFF0000.
ack:    0x00000000, ack_m:      0xFFFFFFFC.
end:    0x00000000, end_m:      0xFFFFFC00.
```

Listing 10: Terminal message from GUI on trigger application button press

In Figure 7.8 is shown the ILA waveforms when sending a message with `id1=0x207`, `DLC=0x02` and `data=0x0000CAFE`. A correct `trigger_out` signal is risen after the id1 (match of the exact value), after the DLC (match of value with mask), after the first data byte (complete match of value 0xCA) and after the second data byte (mask match with value `0xFE`). The triggered signal is risen after the first trigger and will remain high until a new frame is received.

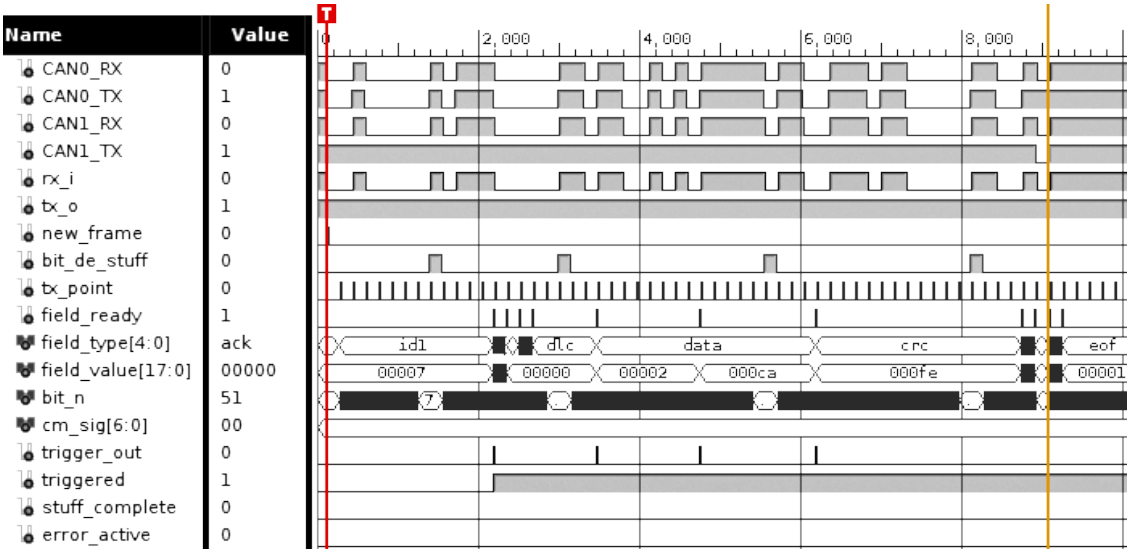

Figure 7.8: Waveform from ILA, triggering on a CAN message

7.2.2.3 Logic Error Tests

Tests for the Logic Error generation module have been performed trying different configurations, stuffing every CAN field (one at a time) with different number of stuff bits for different repetition times. In this way several errors can be generated: stuffing a fixed bit (such as reserved or delimiters) with an illegal value will generate form errors, stuffing on CRC field could[3] generate CRC errors, stuffing a stuff bit will generate a stuff error (see subsubsection 1.4.3.7 – Error Detection and Signalling for errors description).

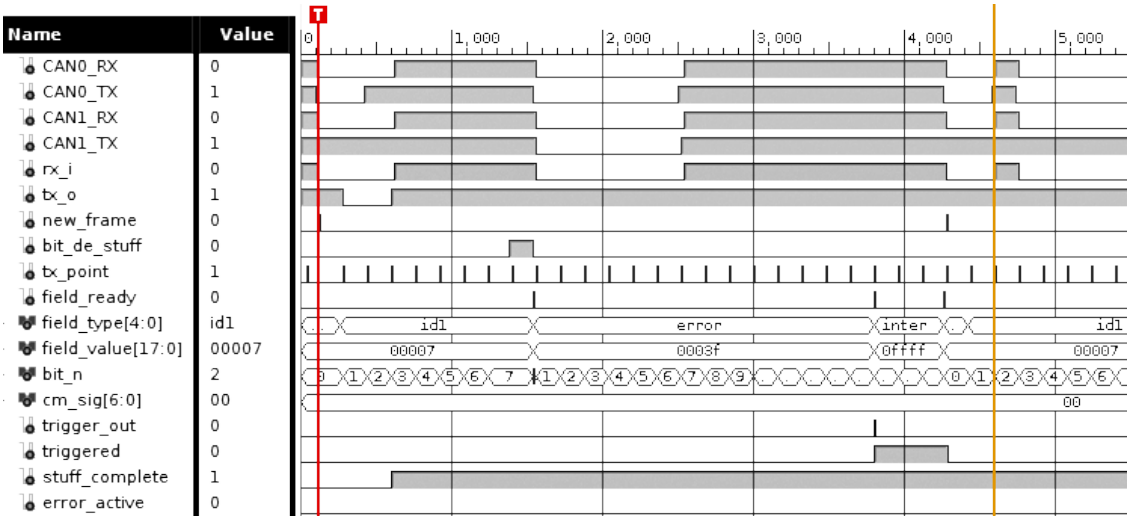

Figure 7.9: Waveform from ILA, logic error on a CAN message

The test case shown in Figure 7.9 uses the trigger function to generate a trigger signal on error frames and the logic error module to inject dominant bits on the message (see Figure 6.3 for a view of the application window). The stuffing is set to start after the idle field (so the id1 field will be stuffed) and two stuffing bits will be sent only one time (no repetitions). The message sent have an id1=`0x0207` so that the second bit is 1 and will be stuffed (see

[3]If the transmitting controller detects that the bit on the bus is different from the one it want to transmit, it will generate a bit error before the completion of the CRC field (neglecting an eventual CRC error).

Figure 6.5 for a view of the send window). The `tx_o` signal (transmission signal from the logic error component) starts at the beginning of the id1 field (in sync with the `tx_point`) and lasts 2 CAN bit time. When stuffing ends (no repetitions have been set) the `stuff_complete` signal rises and will be high until the next configuration of the logic error module.

```
> send 0 0 0 207 8 CAFECAFE 01234567

INT: CAN0-- Stuff ERROR detected -n1---
KSTU01
INT: CAN0--  Lost bus arbitration  ----
KLAR0
INT: CAN1-- Stuff ERROR detected -n2---
KSTU12

STU error from CAN0
LAR error from CAN0

CAN0 sent a frame!

CAN0 sending frame...
ID: 0x40E00000
DLC: 0x80000000
DATA1: 0xCAFECAFE
DATA2: 0x01234567

OK

CAN1 received a frame:
ID: 0x40E00000
DLC: 0x800061E8
DATA1: 0xCAFECAFE
DATA2: 0x01234567
```

Listing 11: Terminal message from GUI on logic error test.

When the transmitting controller on the BUS, the CAN0, detects that the second bit of the id1 is dominant (instead of its transmitted recessive value), a Lost Arbitration error (`KLAR`, see section 6.1 – Serial Interface for error info) is generated. The controller understands that another device is transmitting

on the line with an higher message priority (id) and stops its transmission. The bus remains recessive for 6 CAN bit time (no other controller is really transmitting after CAN0 lost bus access) and a stuff error (`KSTU`) is detected and signaled from both CAN0 and CAN1. The two controllers start sending an error frame. When the bus returns idle, CAN0 tries to send again the message it was trying to send before it lost the arbitration.

In the application debug terminal (Listing 11) can be seen the CAN0 device sending the frame and detecting the Lost Arbitration. Both controllers detect stuff errors and, at the end, the message is sent and received correctly. Line order is not (always) correct because of the different interrupt priority of the routines for sending/receiving messages, detecting errors and printing on the UART.

7.2.2.4 Analog Error Tests

Tests for the analog error generation module was performed in a different way since the relay network and the PCB board were not available at the time of testing. All the possible configuration from the application have been tested, checking the output of the design with the help of the leds integrated on the Zedboard (leds from LD0 to LD6 were mapped on the `cm_sig`). Consistency of output from the board pins for the different test cases were checked with a digital voltmeter.

In Figure 7.10 it can be seen an oscilloscope view of the two CAN line with CANH on CH1 (upper half of the signal) and CANL on CH2 (lower half) when a message with id1=`0x207`, DLC=`1` and data=`0xAB` is sent. An offset of 2,5V have been set so that the two differential signals are aligned with origin (abscissa).

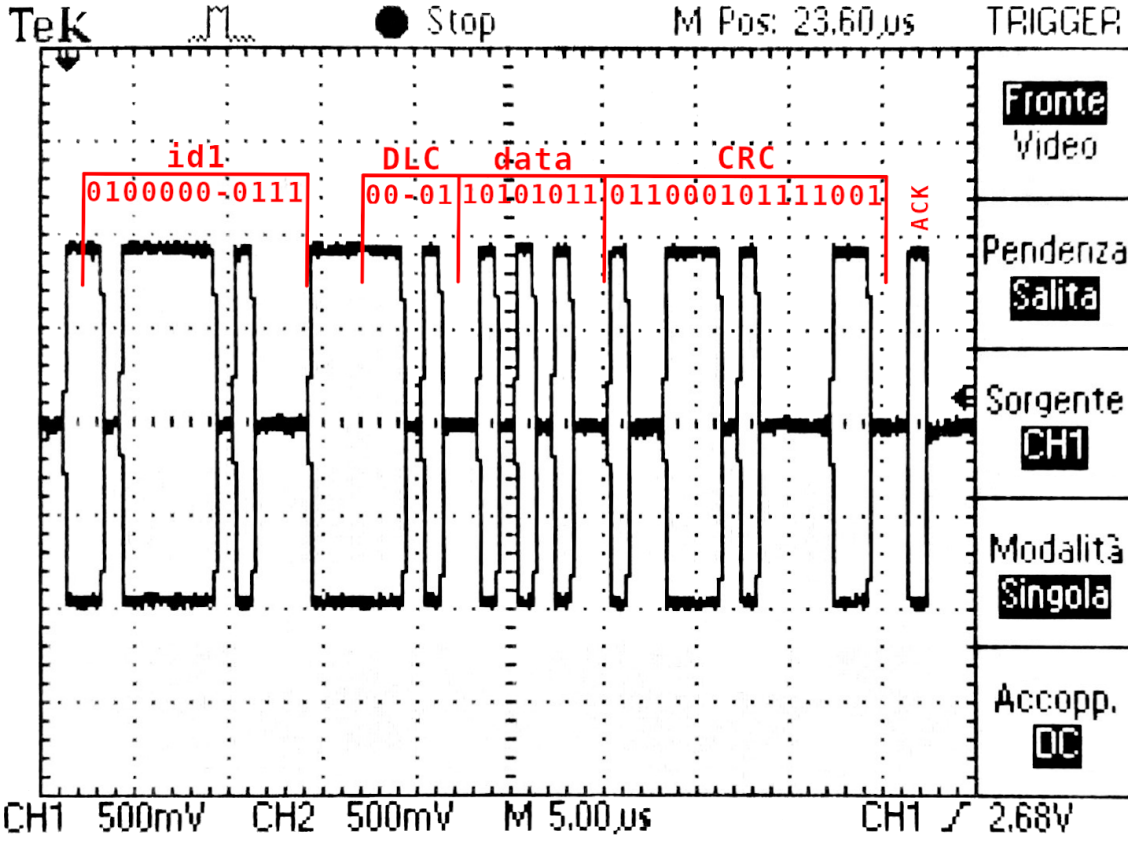

Figure 7.10: Oscilloscope view of a CAN message

To perform various tests, a single short at a time has been manually applied to the system (physically connecting the wires) and the output was controlled on the oscilloscope (to see CANH and CANL signals) and on the ILA (to see the effective message). All tests confirmed the correct functioning of the analog error design.

Conclusion

The scope of this thesis was to create a system able to inject disturbances on a CAN BUS in order to supersede the Vector CANstress tool used in the DIANA testbench.

The designed system proved to be very effective in reading CAN messages and analyze them in order to extract information and proceed with their use in different ways. Its strength is in being able to predict which one should (could) be the next field, based on the protocol, and associate the data read on the bus to the corresponding field. In this way is possible to analyze the data passing on the bus and then perform certain actions almost immediately. In particular, actions are undertaken by different modules, based on their configuration and activation, eventually introducing disturbances on the bus or generating a trigger signal, if certain conditions are met.

These peculiarities allow the Disturbance Tool to be able to carry out the expected functionalities, satisfying all the imposed specifications and requirements.

The planned work will be to test it at its best (introducing code coverage or formal verification tests) and with the help of the PCB in a real system with the DIANA and an ECU. More automation can be introduced by adding the capability of reading a script file with configuration parameters and with the possibility of performing tests in batches, saving the results in a log file.

Furthermore the design versatility will allow it to be easily improved in the future to introduce new features, such as CAN FD support or advanced field conditional checking, and even to be used as a CAN message analyzer, thanks to the two embedded controllers.

References

[1] Vector. (2018). CAN_E – Motivation for CAN, [Online]. Available: http://elearning.vector.com/mod/page/view.php?id=334.

[2] W. Dubitzky and T. Karacay, "CAN – From its early days to CAN FD," *CAN Newsletter*, pp. 8–11, 1/2013, *Improved CAN*.

[3] CAN in Automation. (2019). History of CAN technology, [Online]. Available: https://www.can-cia.org/can-knowledge/can/can-history/.

[4] Robert Bosch GmbH, *CAN Specification*, version 2.0, 1991.

[5] BS ISO 11898-1:2015, "Road vehicles – Controller area network (CAN), Part 1: Data link layer and physical signalling," The British Standards Institution, Standard, 2016.

[6] BS ISO 11898-2:2016, "Road vehicles – Controller area network (CAN), Part 2: High-speed medium access unit," The British Standards Institution, Standard, 2017.

[7] BS ISO 11898-3:2006, "Road vehicles – Controller area network (CAN), Part 3: Low-speed, fault-tolerant, medium-dependent interface," The British Standards Institution, Standard, 2007.

[8] Intecs Solutions. (2019). D.I.A.N.A. Test Bench, Digital Instruments for Automatic Network Analysis, [Online]. Available: http://www.en.intecs.it/page/diana.

[9] I. Mohor. (2009). A VHDL CAN Protocol Controller, [Online]. Available: https://opencores.org/projects/a_vhdl_can_controller.

Acronyms

ACF Acceptance Code Filter

ACK Acknowledge

ASCII American Standard Code for Information Interchange

ATE Automatic Test Equipment

AXI Advanced eXtensible Interface

BRPR Baud Rate Prescaler Register

BRS Bit Rate Switch

BSP Bit Stream Processor

BTL Bit Timing Logic

BTR Bit Timing Register

CAN Controller Area Network

CAN FD CAN Flexible Data rate

CiA CAN in Automation

CRC Cycle Redundancy Check

CSMA/CA Carrier Sense Multiple Access with Collision Avoidance

DIANA Digital Instrument for Automatic Network Analysis

DIP Dual In-line Package

DLC Data Length Code

DLL Data Link Layer

ECU Electronic Control Unit

ECUs Electronic Control Units

EOF End Of Frame

FDF Flexible Data rate Format

FIFO First In First Out

FPGA Field Programmable Gate Array

GCC GNU Compiler Collection

GIMP GNU Image Manipulation Program

GNU GNU's Not Unix

GTK GIMP ToolKit

GUI Graphical User Interface

IDE Identifier

ILA Integrated Logic Analyzer

IP Intellectual Property

ISO International Organization for Standardization

ITM Intermission

LGPL Lesser General Public License

LIN Local Interconnect Network

LLC Logical Link Control

MAC Medium Access Control

MDI Medium Dependent Interface

MOST Media Oriented Systems Transport

MSB Most Significant Bit

MSYS2 Minimal SYStem 2

NRZ Non Return to Zero

OSI Open System Interconnection

PCB Printed Circuit Board

PL Physical Layer

PLS Physical Layer Signalling

PMA Physical Medium Attachment

PMS Physical Medium Specification

RAD Rapid Application Development

REC Receive Error Counter

RRS Remote Request Substitution

RTR Remote Transmission Request

SDK Software Development Kit

SJW Synch Jump Width

SMD Surface Mount Device

SoC System on Chip

SOF Start Of Frame

SOIC Small Outline Integrated Circuit

SPDT Single Pole Double Throw

SPST Single Pole Single Throw

SRR Substitute Remote Request

TEC Transmit Error Counter

THT Trough Hole Technology

tq time quantum

UART Universal Asynchronous Receiver-Transmitter

USB Universal Serial Bus

UTP Unshielded Twisted Pair

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuits