POLITECNICO DI TORINO

DEPARTMENT OF CONTROL AND COMPUTER ENGINEERING (DAUIN)

Master degree course in Electronic Engineering

Master Degree Thesis

In-Field Functional Test of CAN Bus Controller

Supervisors:
prof. Matteo Sonza Reorda
prof. Riccardo Cantoro

Candidate
Sandro Sartoni
ID Number: 252308

Academic Year 2018-2019

Summary

The presented thesis work is centered on the topic *functional test of a CAN Bus peripheral*.

The reason behind this work is due to the fact that such peripherals, that are usually employed in the automotive sector, are often part of safety-critical systems, thus making the constant test of their hardware vital, even during the operational phase. Functional test was, hence, adopted since it's the only solution that allows the in-field test during Unit Under Test's idle time intervals without any additional hardware that would otherwise be necessary if a Design for Testability approach was adopted.
The test is based on a test program that resides in the UUT's memory and is launched whenever necessary. The CAN Controller's output values, when running the test, should be compacted and stored in order to compare them against the golden ones, i.e., values from the functioning circuit, in order to check the correct functioning at any time.

In order to develop the test a CAN Peripheral module was needed and, since all of the simulations and tests were conducted via software, no physical hardware was employed; on the contrary, an open RTL implementation of the SJA1000 CAN Bus peripheral, developed in the early 2000s by Philips, now NXP, was adopted.

Controlling the peripheral behavior by means of simulations was a necessary introductory step in order to understand whether there were problems or expedients to adopt when working with such controller.
The main problem in this phase consisted in an abnormal propagation of undefined signals throughout the whole peripheral in the post-synthesis version. This was due to the absence of a reset value - as required by the specifications - for some registers that were involved in the internal logic, leading to the propagation of such undesired values.

Once this problem was fixed, the next step consisted in the integration of the

CAN Controller in a SoC, in particular an OpenRisc one, as most modern systems are based on SoCs that already implement a CAN peripheral. This was achieved by connecting the two interfaces and configuring the CAN Controller's starting address on the CPU's memory map.

Concurrently, the necessary drivers were developed. The drivers have been created by following the description found in the datasheet of the SJA1000 peripheral and checking the open source drivers provided by NXP for their SJA1000 Controller. At the same time, it was necessary to consider the actual functionalities offered by the adopted controller, as they do not exactly overlap with the datasheet's ones.

The drivers' test has been conducted both with the pre-synthesis version of the CAN Controller - necessary to test their correctness from a logical point of view - and with the post-synthesis version, in order to do some fine-tuning.

Once the drivers were working, it was time to develop the actual test program, from here on SelfTest Library, or *STL*.
The test program is based on a testbench configuration that consists of two nodes connected through a CAN bus. The difference between the real case and the proposed testbench is, from an electrical point of view, the absence of a CAN transceiver that converts the `TX` and `RX` lines into the `CANH` and `CANL` the standard requires; however, this was solved by appropriately connecting the two modules.
The program is comprised of many submodules that can be arranged into as many test programs as necessary that can fit in multiple idle state time intervals. Eventually, if the the idle state interval is long enough, the test can be launched as a whole.
The test requires the presence of at least two nodes, as to test both the receiving and the transmitting logic.

While simulating the program running on the SoC, the internal and interface signal activity of the peripheral was recorded and then used to perform a fault simulation based on the *stuck-at* class of faults. In order to check how well the test program was performing, other programs were developed, especially tests that could emulate more realistic scenarios.

The final result shows a test coverage of 90.24% of all of the stuck-at faults.

Acknowledgements

I would like to thank my two supervisors Matteo Sonza Reorda and Riccardo Cantoro for the constant support and help throughout the whole thesis work.

I would also like to thank my parents, without them nothing of what I've achieved so far could have been possible.

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

1.1 CAN Bus Standard

The *Controller Area Network*, or *CAN*, Bus is a communication bus standard first introduced in 1986 by Bosch that is intended to work even in noisy environments such as automotive applications. It's used for serial communication applications among *Microcontroller Units*.
In order to achieve such features, the peripheral has to be robust; this quality is achieved by means of:

- *Differential Signals*: the CAN Bus consists of a pair of lines, named `CANH` and `CANL`; the transmitted data is handled in a way such that any receiving node can restore the original information by evaluating the subtraction between the signal on the `CANH` line and the one on the `CANL` line.
  From an electrical point of view, at least generally speaking as the actual implementation depends on the particular version of the standard adopted, when both signals are idle, i.e., a logical `1'b0` - also known as dominant bit - is being transmitted, they have a $\simeq$ 2.5V level with a 5V power supply, when transmitting a logical `1'b1` - also known as recessive bit - the `CANH` line is on a $\simeq$ 3.5V level, while the `CANL` one is on a $\simeq$ 1.5V level.

- *Bit Stuffing*: the encoding adopted for this protocol is the *Non Return to Zero*, or *NRZ*, and the clock is not transmitted on a separate line along the data bits - as for the SPI protocol; the receiver simply synchronizes on $1 \rightarrow 0$ transitions. If the message presents sections in which the receiver cannot resynchronize frequently enough with the transmitter, the two will drift apart and the receiver will sample the incoming bits when it's not supposed to. To avoid such problem, bit stuffing is adopted as it prevents loss of synchronization between transmitter and receiver.

In general, it is performed by means of inserting an opposite polarity bit when finding a sequence of 5 consecutive bits of the same polarity, e.g. if the transmitted message requires a sequence of five zeros, the next bit will necessarily be a 1 no matter which was the next "scheduled" bit, and viceversa. All of the stuffing bits will be then removed by the receiver(s) node.

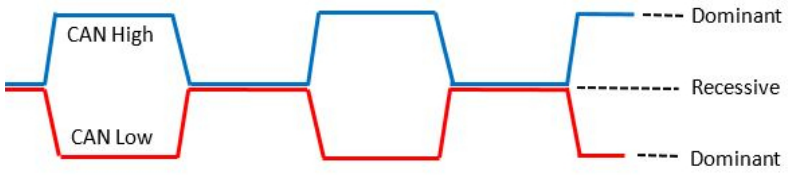

Figure 1.1. CAN typical waveform

The CAN Bus protocol consists of a multi-master bus, each node having the capability of sending messages to any other node as soon as the bus is idle.
There is no need for a Bus Arbiter as there's no possibility of harmful collisions of data, neither from a logical or electric point of view.

From a logical point of view, the standard has been developed in a way such that, in case of collision, the dominant bit always wins over the recessive one. In this way, if two or more nodes are transmitting at the same time and one of them transmits a recessive bit while others are transmitting a dominant one, the bus will always present a dominant bit. The recessive bit's node is then capable of finding the discrepancy and therefore stop its transmission, re-scheduling it once the current one is over. This is also know as *Arbitration Lost State*. A node that has to send a message will send it as soon as the required command is received, if an arbitration loss occurs it will suspend it and retry later.

From an electrical point of view, every node adopts an *open drain* technology which means that the transmitting logic can only drive the `CANL` line to a low voltage state value and the `CANH` line to a high voltage state value (dominant bit), the other state (recessive bit) is obtained by means of a couple of terminating resistors, each one of 120Ω, that ensures a voltage equal to $\frac{V_{DD}}{2}$ on both lines if no dominant bit is sent or no transmission is occurring. In this way, even if there are two nodes transmitting, there can be no short circuit and the whole network is therefore safe.

The typical interconnections configuration can be seen in figure 1.2. For the sake of simplicity only three nodes have been represented even though there could be many more.

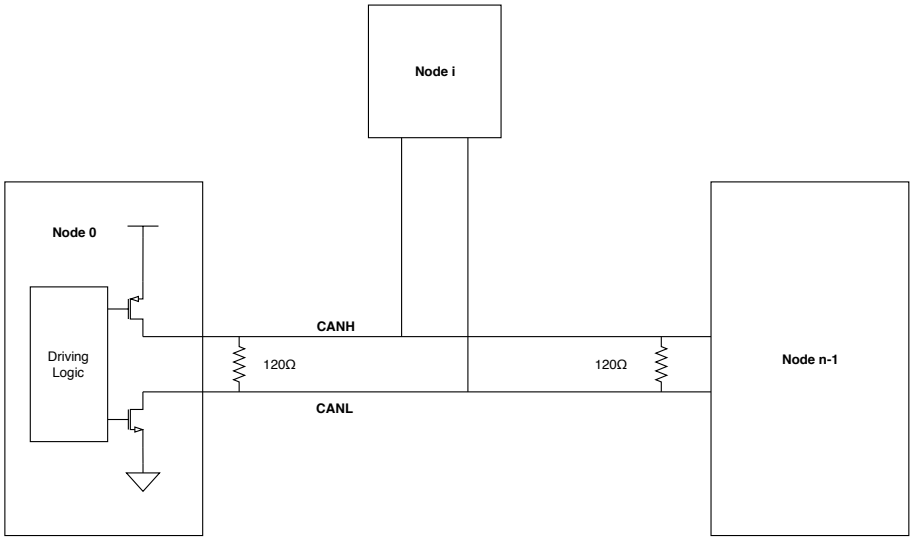

Figure 1.2. Example of CAN Network with terminating resistors

The CAN standard supports four different frames (i.e., types of message), that are:

1. *Data Frame*: a message to transfer data from a sending node to one or more receiving nodes.

2. *Remote Frame*: a node requests data from a source node. A remote frame is followed by a data frame containing the requested data.

3. *Error Frame*: any bus participant may signal an error condition at any time during a transmission.

4. *Overload Frame*: a node can request a delay between two data or remote frames.

As for Data Frame, messages are divided into different fields like the *ID* (the identifier of the recipient of the message), *DLC* or Data Length Code (the number of bytes to be sent), *DB* or Data Bytes (the actual message), and *CRC* for error detection. The recipient notifies the transmitter of the correct reception by means of an Acknowledge bit.
Every node is equipped with an *Acceptance Filter* that is in charge of deciding whether the receive message is intended for that node, based on the ID, RTR and DB sent with the message. Generally speaking, each node has a range of admitted IDs and whatever message which ID does not belong to such range will be correctly received (if the settings are correct) setting, eventually, the Acknowledge bit; the

message, however, will not be stored anywhere so that it cannot actually be retreived by the Controller.

Data Frames come in two formats with respect to the ID size: 11 bits for the *Base* format, and 29 bits for the *Extended* format.

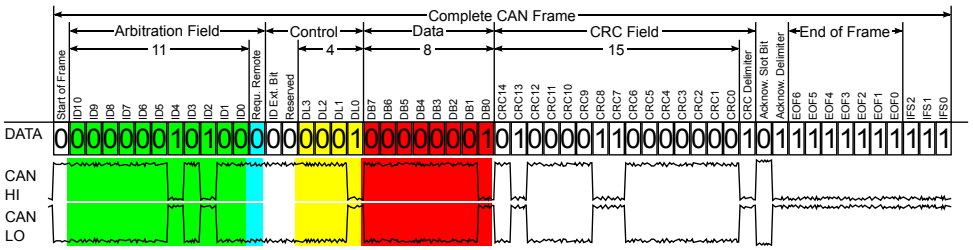

Figure 1.3. Base Format Data Frame

A CAN Bus peripheral always includes a controller which elaborates commands sent from another module (e.g., a microprocessor) and sets everything to correctly send and receive messages. It handles all of the error conditions as well, providing the outer world a register-based interface in which there are command, status and data registers. Details on the registers interface can be found in section 2.2.

Even though every CAN Controller manufacturer ensures the same functionalities, as required by the standard, the actual implementation of such peripheral depends on the producer. A brief list of some of the main implementations is: SJA1000 by NXP, bxCAN by STMicroelectronics, TI TMS230, and Infineon MultiCan.
In chapter 2 the SJA1000 model will be described, since this is the CAN implementation adopted in this work.

1.2 Testing Generalities

The aim of this thesis work consists, as will be thoroughly described later on, in testing the adopted CAN peripheral by functional means.

To test an integrated circuit consists in applying a suitable set of input stimuli to excite and make observable physical defects at - at least - one of the IC's output ports. This is a crucial step for every device in general and, in particular, for all of the applications in which safety and reliability are of paramount importance.
These defects may stem from many different causes and taking each and every one of them into account would be infeasible due to the huge number, this is why the

general approach consists in creating fault classes that model physical defects.

One of the most adopted classes is the *stuck-at fault* one. As the name suggests, this faults' class models physical defects by assuming that a certain line in the netlist is either stuck to a logic `1'b0` or a `1'b1`; the former case is called *stuck-at zero* fault while the latter is the *stuck-at one* fault. This obviously can be applied to any wire in the netlist.

In picture 1.4 there's a rather simplistic example of a stuck-at zero fault on the G wire: whatever value is applied to said wire the only displayed value will be a logic `1'b0`. In order to notice the presence of such fault it's necessary to excite the fault first, that is, to force the opposite value on the line under test: in this case, it means to make sure to have a logic `1'b1` on G.
Next, it's necessary to make the discrepancy observable at the output: in order to do so, F has to be on a `1'b1` level otherwise Z would be equal to `1'b0` no matter what happens in the other circuit section. The input vector that tests `G/0` comes as a consequence of the previous observations.
It's relevant to notice that there could be many different vectors able to test the same fault: in this case, instead of having `CD=2'b10` it could have been the opposite value `CD=2'b01`, or even `CD=2'b11`.

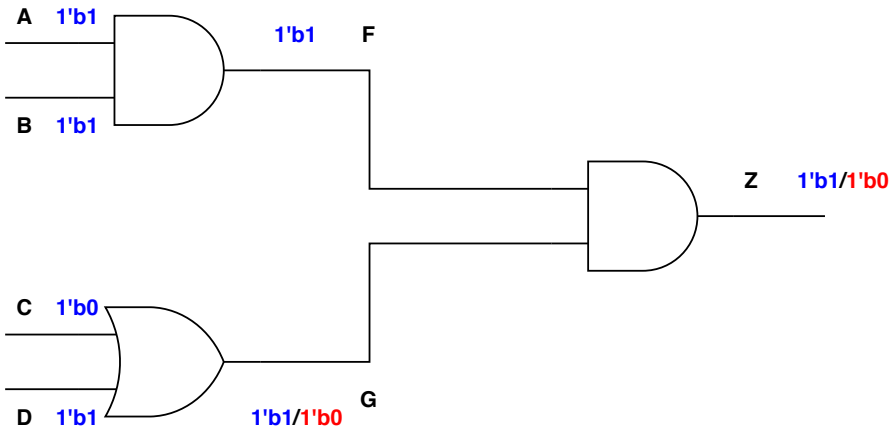

Figure 1.4. Stuck-at fault example

There are several commercial tools capable of performing a fault simulation. Fault simulating consists in, given a list containing all the faults in a DUT and a set of input vectors to apply to the DUT, evaluating, usually in terms of a percentage, how many faults are tested by said stimuli. Eventually, it's even possible to run a software tool named *ATPG*, or *Automatic Test Pattern Generator*, that is capable

of, given the circuit, finding a set of input vectors with an associated fault coverage.

When dealing with purely combinational circuits, ATPG tools work well in most cases and are capable of providing high fault coverages in a relatively small amount of time. The same result is, however, not achieved when dealing with sequential circuits: the presence of flip flops forces the tool to follow the states through which the netlist under test evolves, thus requiring much more time and returning worse results.
Since most integrated circuits contain sequential logic, two main ways, alternative to the usage of the ATPG, can be adopted.

1.2.1 Design for Testability

One way consists in using *Design for Testability* modules, also known as *DfT*. A Design for Testability approach implies a certain number of different solutions that require the introduction of additional hardware and/or the modification of the already present one in order to facilitate the test process.
Even though this section's aim is not to describe in detail how DfT is implemented, in order to demonstrate the additional hardware requirements, two DfT techniques will be shown below.

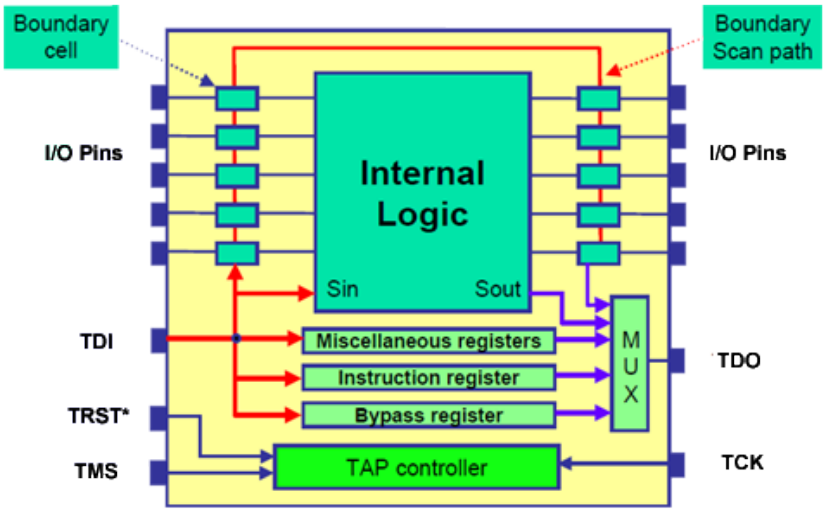

Figure 1.5. An IC including JTAG hardware

In figure 1.5 is reported the internal structure of an integrated circuit that adopts the JTAG standard. The module in green named *Internal Logic* is the original integrated circuit; the JTAG introduces one *boundary scan cell* per input/output port in order to scan in/scan out test vectors that may be used to test the inner

core or the interconnections among chips. Additionally, a *TAP Controller* and few additional registers - needed to drive the whole test process - are introduced.

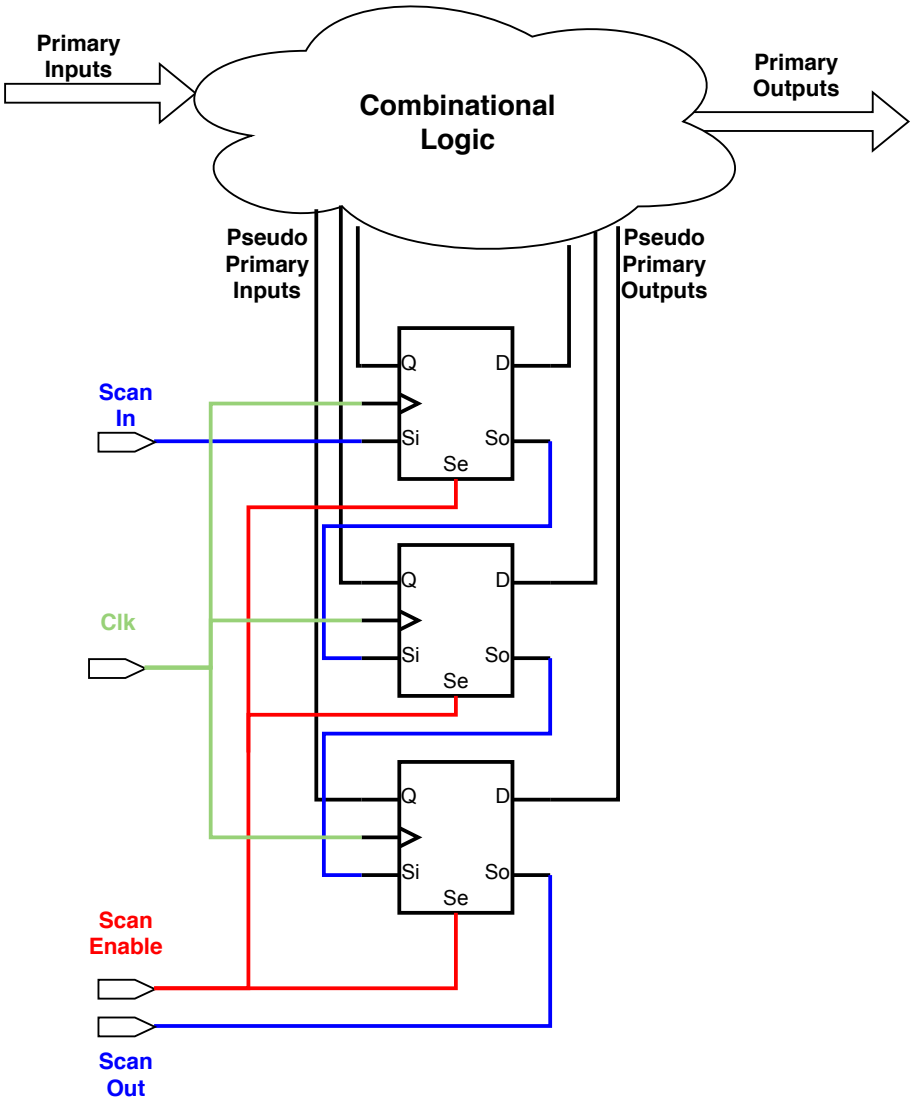

Figure 1.6. Scan Chain implementation

Additionally, in figure 1.6 an IC implementing the scan chain technique is shown. The circuit has been divided into the combinational and sequential sections. In the sequential one it's possible to see how the classical *D-Type flip flops* have been replaced with flip flops suited to perform a scan operation, i.e., to scan in test vectors to apply through the Pseudo Primary Inputs to the Combinational Logic which effects, combined with the Primary Inputs' ones and sampled via the Pseudo

Primary Outputs and Primary Outputs, will eventually be scan out through the daisy chain formed by the connection of all of the `scan in/out` pins. This operation is possible thanks to the presence of a `scan enable` signal.
In this case the hardware overhead is reduced with respect to the JTAG case, still all of the flip flops have to be modified and three additional pins have to be added, at least in the basic implementation.
To conclude, DfT solutions ease the testing process because they allow to increase the controllability and observability of certain faults that would be otherwise difficult to reach by classic means but this comports an area overhead, a possible timing overhead and eventually the cost per chip may increase. Moreover test is possible only at the end of the production phas

1.2.2 Functional Test

The functional approach, on the other hand, does not rely on modified or added hardware as in the previous case. In fact, it can be defined as a test process conducted without resorting to any kind of Design for Testability solution.
The most adopted approach in Functional Test is called *SBST*, which stands for *Software Based Self Test*. This is the employed technique in this work.

SBST can only be performed when working with CPUs or SoCs as it is based on a test program that has to be loaded into memory and then run whenever required. In most applications it's safe to assume that any MicroController Unit has some idle time, i.e., a time interval in which such unit is not performing any task: instead of wasting this time it may be useful to test the unit to check the correct functioning. This is especially true in safety critical applications.
Moreover, other features offered by this metodology are the at-speed test (that could catch much more defects than those found with other approaches that work at a slower speed) and the in-field test as well, that is, test of a circuit in operating conditions during its operative life-cycle.

The general flow is described as follows: the test program, once loaded in memory, is executed and test results are recorded. Results can be taken as is or, more frequently, compacted by means of a *MISR - Multiple Input Signature Register*, a type of linear feedback shift register - in order to process bigger streams of data. These results are then compared to the golden ones, i.e., the ones that should be obtained when the circuit works properly, leading to the knowledge of the working state of the circuit.

This approach shifts the whole workload from hardware design to software design as the effectiveness of the test now relies on the test program.

One of the crucial parts of this work is the development of such test program, in fact, chapter 4 is entirely dedicated on the development of the proposed SelfTest program while in chapter 5 the relative achieved results will be shown and compared to other - still functional - comparison tests.

Chapter 2

Adopted CAN Controller

The presented work is based on the functional test of a CAN Bus peripheral, hence the need to find the hardware description of a CAN Controller as a starting point. Due to this, the choice was based on finding a peripheral that could be open hardware and could be found online.
The result of this search was an implementation of the SJA1000 peripheral.

2.1 SJA1000 CAN Implementation

The SJA1000 is a stand-alone controller for the CAN developed by Philips Semiconductors (now NXP Semiconductors) in the early 2000s.

It is the successor of the PCA82C200 CAN controller (*BasiCAN*) from Philips Semiconductors. Additionally, a new mode of operation is implemented (*PeliCAN*) which supports the CAN 2.0B protocol specification, that is the Extended Format specified in the previous chapter, with several new features.
A block diagram of the peripheral is shown in 2.1.

Here will be provided a brief explanation of the blocks that constitute the peripheral:

1. *Interface Management Logic*: this block contains a set of registers that implements the peripheral interface. The registers interface layout heavily depends on the particular configuration used - whether the peripheral works in *BasiCAN* or *PeliCAN* mode - and it contains the logic necessary to interpret the received commands and drive the whole Controller as well.

2. *Message Buffer*: this block is in charge of providing an interface between the external CPU and other CAN Controller modules that handle messages, both received or to be sent: the Transmit Buffer (TXB) stores the message

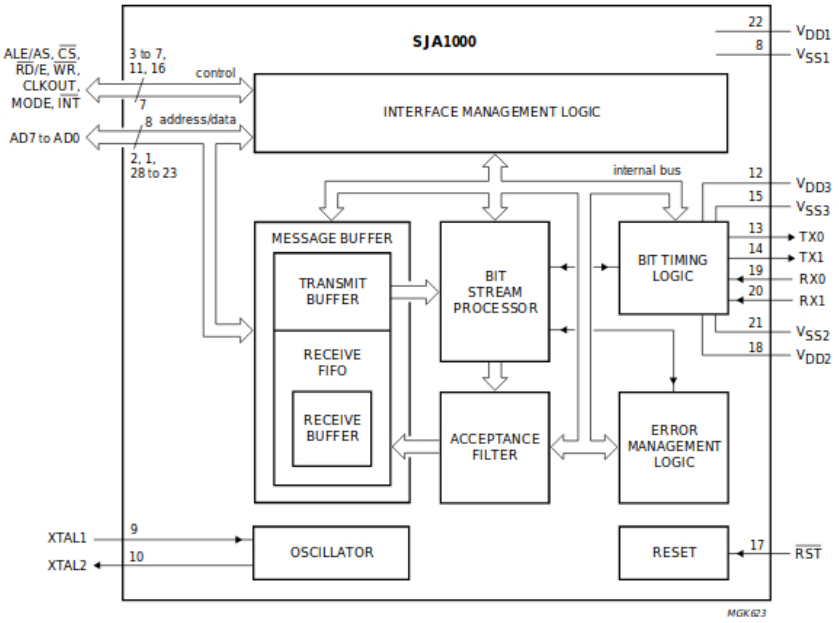

Figure 2.1. Block Diagram of the SJA1000 CAN Controller

ready to be sent that has been assembled by the Bit Stream Processor (BSP), the receive buffer (RXB) stores the received message that comes from the Acceptance Filter. The Receive Buffer belongs to the Receive FIFO, that is a FIFO capable of storing up to 64 bytes of messages: from this point of view, the RXB is a window that shifts through the whole FIFO. It has to be noted that the Message Buffer is not responsible for message manipulation, filtering or assembly, it's just a module in which messages are stored, waiting for further steps.

3. *Bit Stream Processor (BSP)*: this block is the heart of the control and processing unit of the peripheral. It consists of a sequencer which controls the data stream between the transmit buffer and the CAN-bus. It also performs the error detection, arbitration, stuffing and error handling on the CAN bus.

4. *Acceptance Filter (ACF)*: this block is in charge of checking whether the message currently on the bus has to be stored by the peripheral or not.
Everytime a message is being received, at the end of the process - if there was no error - the receiver will set the acknowledge bit; it will then be the ACF's responsibility to decide if the message will be stored in the RXB and, concurrently, set the appropriate bit to notify the acquisition.

This is assessed by means of an acceptance filter that usually works on the ID, Remote Transfer Request (RTR) bits and DB fields of the message via two registers, the Acceptance Mask (AM) and the Acceptance Code (AC).

5. *Bit Timing Logic (BTL)*: the BTL block monitors the serial CAN-bus line and handles the bus line-related bit timing. It is synchronized to the bit stream on the CAN-bus on a recessive-to-dominant (i.e., $1 \rightarrow 0$) transition at the beginning of a message (hard synchronization) and re-synchronized on further transitions during the reception of a message (soft synchronization). The BTL also provides programmable time segments to compensate for the propagation delay times and phase shifts and to define the sample point and the number of samples to be taken within a bit time.

6. *Error Management Logic (EML)*: the EML is responsible for the error confinement of the transfer-layer modules. It receives error announcements from the BSP and then informs the BSP and IML about error statistics.

2.2 Verilog Hardware Description

The hardware implementation of the SJA1000 CAN Controller comes from the OpenCores website, it has been developed using the *Verilog* HDL and the author ensures that the module has been tested on actual hardware - in particular, an FPGA - and verified with the Bosch VHDL Reference System.

This module offers almost every functionality provided by the SJA1000 Controller except for a couple of little discrepancies.
The first one is related to the set of implemented registers: the registers layout can be seen in pictures 2.2, 2.3 and 2.4 for, respectively, the BasiCAN and the PeliCAN modes. The author of the RTL description of the controller decided not to implement two of them, the `output control` and `test` registers.
The `output control` register is in charge of driving the output transistors that will physically drive the signals. It is comprised of 8 bits that are organized as presented in table 2.1.

The two set of P/N transistors that are highlighted by means of the numbers 0 and 1 are due to the fact that the SJA1000 peripheral implements two nodes, as it can be seen in figure 2.1, in fact, on the bus output ports, there are two sets of Tx and Rx signals. The RTL description, on the other hand, only implements one node - thus having only one set of Tx and Rx ports - and their configuration is fixed: whenever the line is in idle state this is kept at `1'b1`, when transmitting the classic Non Return to Zero encoding is adopted.

CAN ADDRESS	SEGMENT	OPERATING MODE		RESET MODE	
		READ	WRITE	READ	WRITE
0	control	control	control	control	control
1		(FFH)	command	(FFH)	command
2		status	–	status	–
3		interrupt	–	interrupt	–
4		(FFH)	–	acceptance code	acceptance code
5		(FFH)	–	acceptance mask	acceptance mask
6		(FFH)	–	bus timing 0	bus timing 0
7		(FFH)	–	bus timing 1	bus timing 1
8		(FFH)	–	output control	output control
9		test	test; note 2	test	test; note 2
10	transmit buffer	identifier (10 to 3)	identifier (10 to 3)	(FFH)	–
11		identifier (2 to 0), RTR and DLC	identifier (2 to 0), RTR and DLC	(FFH)	–
12		data byte 1	data byte 1	(FFH)	–
13		data byte 2	data byte 2	(FFH)	–
14		data byte 3	data byte 3	(FFH)	–
15		data byte 4	data byte 4	(FFH)	–
16		data byte 5	data byte 5	(FFH)	–
17		data byte 6	data byte 6	(FFH)	–
18		data byte 7	data byte 7	(FFH)	–
19		data byte 8	data byte 8	(FFH)	–
20	receive buffer	identifier (10 to 3)	identifier (10 to 3)	identifier (10 to 3)	identifier (10 to 3)
21		identifier (2 to 0), RTR and DLC	identifier (2 to 0), RTR and DLC	identifier (2 to 0), RTR and DLC	identifier (2 to 0), RTR and DLC
22		data byte 1	data byte 1	data byte 1	data byte 1
23		data byte 2	data byte 2	data byte 2	data byte 2
24		data byte 3	data byte 3	data byte 3	data byte 3
25		data byte 4	data byte 4	data byte 4	data byte 4
26		data byte 5	data byte 5	data byte 5	data byte 5
27		data byte 6	data byte 6	data byte 6	data byte 6
28		data byte 7	data byte 7	data byte 7	data byte 7
29		data byte 8	data byte 8	data byte 8	data byte 8
30		(FFH)	–	(FFH)	–
31		clock divider	clock divider; note 3	clock divider	clock divider

Figure 2.2. BasiCAN Registers Address Allocation

OC.7	Output Control Transistor P1
OC.6	Output Control Transistor N1
OC.5	Output Control Polarity 1
OC.4	Output Control Transistor P0
OC.3	Output Control Transistor N0
OC.2	Output Control Polarity 0
OC.1	Output Control Mode 1
OC.0	Output Control Mode 0

Table 2.1. Output Control Register

The `test` register should be used in the production test phase only. The technical datasheet of the SJA1000 IC reports that using this register during normal operation may result in undesired behaviour of the device. Since this phase will not be of interest, this register is not present as well.
There is, though, a set of ports that can be exploited to test the peripheral by means of a BIST approach: if the keyword `CAN_BIST` is defined at compile time, a whole new hardware block that is already present in the hardware description, but not

used, will be added to the peripheral. Even though it may make the whole testing process easier, this goes against the whole idea of not using a Design for Testability approach and will thus be avoided.

CAN ADDRESS	OPERATING MODE				RESET MODE	
	READ		WRITE		READ	WRITE
0	mode		mode		mode	mode
1	(00H)		command		(00H)	command
2	status		–		status	–
3	interrupt		–		interrupt	–
4	interrupt enable		interrupt enable		interrupt enable	interrupt enable
5	reserved (00H)		–		reserved (00H)	–
6	bus timing 0		–		bus timing 0	bus timing 0
7	bus timing 1		–		bus timing 1	bus timing 1
8	output control		–		output control	output control
9	test		test; note 2		test	test; note 2
10	reserved (00H)		–		reserved (00H)	–
11	arbitration lost capture		–		arbitration lost capture	–
12	error code capture		–		error code capture	–
13	error warning limit		–		error warning limit	error warning limit
14	RX error counter		–		RX error counter	RX error counter
15	TX error counter		–		TX error counter	TX error counter
16	RX frame information SFF; note 3	RX frame information EFF; note 4	TX frame information SFF; note 3	TX frame information EFF; note 4	acceptance code 0	acceptance code 0
17	RX identifier 1	RX identifier 1	TX identifier 1	TX identifier 1	acceptance code 1	acceptance code 1
18	RX identifier 2	RX identifier 2	TX identifier 2	TX identifier 2	acceptance code 2	acceptance code 2
19	RX data 1	RX identifier 3	TX data 1	TX identifier 3	acceptance code 3	acceptance code 3
20	RX data 2	RX identifier 4	TX data 2	TX identifier 4	acceptance mask 0	acceptance mask 0
21	RX data 3	RX data 1	TX data 3	TX data 1	acceptance mask 1	acceptance mask 1
22	RX data 4	RX data 2	TX data 4	TX data 2	acceptance mask 2	acceptance mask 2
23	RX data 5	RX data 3	TX data 5	TX data 3	acceptance mask 3	acceptance mask 3
24	RX data 6	RX data 4	TX data 6	TX data 4	reserved (00H)	–
25	RX data 7	RX data 5	TX data 7	TX data 5	reserved (00H)	–
26	RX data 8	RX data 6	TX data 8	TX data 6	reserved (00H)	–

Figure 2.3. PeliCAN Registers Address Allocation [part 1]

The second main difference between the SJA1000 reference module and the Verilog implementation is the latter module's interface. The peripheral is equipped with a standalone-like set of ports, as required by the real module, and it comes with a Wishbone Bus interface as well. The possibility of choosing the interface is related to the declaration of a keyword, `CAN_WISHBONE_IF`: if defined, the interface will be the Wishbone Bus one, otherwise it will be the standalone one. This will be of paramount importance to the integration process in the SoC, as will be explained in the next chapter.

CAN ADDRESS	OPERATING MODE				RESET MODE	
	READ		WRITE		READ	WRITE
27	(FIFO RAM); note 5	RX data 7	–	TX data 7	reserved (00H)	–
28	(FIFO RAM); note 5	RX data 8	–	TX data 8	reserved (00H)	–
29	RX message counter		–		RX message counter	–
30	RX buffer start address		–		RX buffer start address	RX buffer start address
31	clock divider		clock divider; note 6		clock divider	clock divider
32	internal RAM address 0 (FIFO)		–		internal RAM address 0	internal RAM address 0
33	internal RAM address 1 (FIFO)		–		internal RAM address 1	internal RAM address 1
↓	↓		↓		↓	↓
95	internal RAM address 63 (FIFO)		–		internal RAM address 63	internal RAM address 63
96	internal RAM address 64 (TX buffer)		–		internal RAM address 64	internal RAM address 64
↓	↓		↓		↓	↓
108	internal RAM address 76 (TX buffer)		–		internal RAM address 76	internal RAM address 76
109	internal RAM address 77 (free)		–		internal RAM address 77	internal RAM address 77
110	internal RAM address 78 (free)		–		internal RAM address 78	internal RAM address 78
111	internal RAM address 79 (free)		–		internal RAM address 79	internal RAM address 79
112	(00H)		–		(00H)	–
↓	↓		↓		↓	↓
127	(00H)		–		(00H)	–

Figure 2.4. PeliCAN Registers Address Allocation [part 2]

As a last note, the RTL description of the peripheral comes with a Verilog testbench with which the module has been tested. Inside of it are defined few different tasks that are in charge of checking the various logic modules in which the controller is divided.
Thanks to this testbench the first simulations were conducted, in order to get used to the way in which the peripheral works, to check whether everything was going as expected and to understand how the various steps followed in the testbench tasks worked - e.g. which were the necessary steps to send a message or to receive one, etc. - with the aim to emulate them as much as possible with the *C* drivers.

2.3 Standalone Controller Simulation

The standalone controller simulation was conducted in two different scenarios: having the RTL description of the CAN Controller as the DUT and, later, its synthesized version. The first case served as a confirmation of the validity of the logic behind the tasks while the second one is a more realistic simulation. As a last note, the whole process was automated by means of scripts, both in *TCL* and *Bash* languages.

The RTL simulation worked flawlessly, no error was reported and, in particular, the Tx and Rx signals showed the typical waveforms of a CAN message. In figure 2.5, a small snippet of the waveforms of the interface ports when sending and transmitting a message is shown.

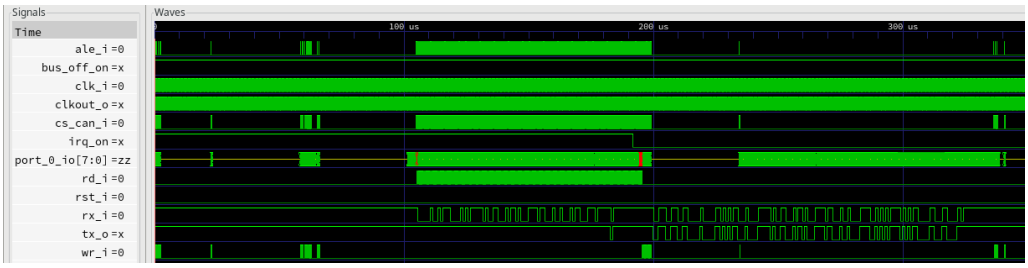

Figure 2.5. Pre-Synthesis Standalone Controller Waveforms

On the other hand, some problems were encountered when simulating the post-synthesis controller. More specifically, there were abnormal *undefined* values that appeared at the output ports, even in ports that were previously on a well defined hardware value.
The same snippet posted before is presented in figure 2.6, now having the post-synthesis peripheral.

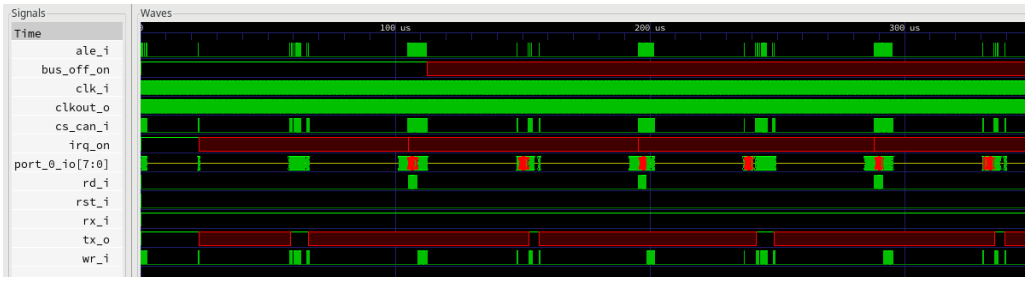

Figure 2.6. Post-Synthesis Standalone Controller Waveforms

In order to understand what was the source of the problem, a thorough investigation was launched.
The adopted approach was the following: starting from an internal signal that showed the transition from a defined value to an undefined one, if such signal kept its original name after the synthesis process - in order to have a better understanding of what drives that signal by looking at the RTL description - then the *TraceX* functionality offered by the simulation software *Questa Sim-64* was used in order to understand which was the cause of such odd behavior. *TraceX* offers the possibility of tracing the source of a $1/0 \rightarrow X$ transition, where X is intended as the undefined value.
Many signals, although not every one of them, were analyzed and the results showed that the origin of such problems was to imply to a subset of registers that did not have a defined value at reset time. The author of this hardware description, in fact, followed the SJA1000 reference datasheet which required these registers not to lose their content at reset, probably not to affect the correct functioning of the peripheral. Some of said registers can be seen in figure 2.7.

REGISTER	BIT	SYMBOL	NAME	VALUE	
				RESET BY HARDWARE	SETTING BIT CR.0 BY SOFTWARE OR DUE TO BUS-OFF
Acceptance code	AC.7 to 0	AC	Acceptance Code	X	X
Acceptance mask	AM.7 to 0	AM	Acceptance Mask	X	X
Bus timing 0	BTR0.7	SJW.1	Synchronization Jump Width 1	X	X
	BTR0.6	SJW.0	Synchronization Jump Width 0	X	X
	BTR0.5	BRP.5	Baud Rate Prescaler 5	X	X
	BTR0.4	BRP.4	Baud Rate Prescaler 4	X	X
	BTR0.3	BRP.3	Baud Rate Prescaler 3	X	X
	BTR0.2	BRP.2	Baud Rate Prescaler 2	X	X
	BTR0.1	BRP.1	Baud Rate Prescaler 1	X	X
	BTR0.0	BRP.0	Baud Rate Prescaler 0	X	X
Bus timing 1	BTR1.7	SAM	Sampling	X	X
	BTR1.6	TSEG2.2	Time Segment 2.2	X	X
	BTR1.5	TSEG2.1	Time Segment 2.1	X	X
	BTR1.4	TSEG2.0	Time Segment 2.0	X	X
	BTR1.3	TSEG1.3	Time Segment 1.3	X	X
	BTR1.2	TSEG1.2	Time Segment 1.2	X	X
	BTR1.1	TSEG1.1	Time Segment 1.1	X	X
	BTR1.0	TSEG1.0	Time Segment 1.0	X	X

Figure 2.7. Some registers which value is not affected at reset time

These registers, though, were used by the author to define other signals that propagate throughout the peripheral and while this undefined value propagation didn't show in RTL simulation, when working with logic ports as in the post-synthesis module these few registers quickly invalidated the remaining part of the circuit.
This problem was not reported on the website, the reason being that when working

on actual hardware the undefined values do not exist, they are a way the simulator warns the user that such values have not been correctly initialized and they could actually be any allowed value. The hypothesis, then, was that any value assigned to these registers could make the circuit work as intended again.
A few tests have been conducted by using the `deposit` command on the script used to simulate the circuit in order to assign, at reset time, a user-defined value that could be possibly overwritten when the relative hardware needed to drive such signals. Every tried value let the circuit work correctly, so in the end the decision was to assign to any of those registers a 0 value at reset, in order to emulate what generally happens in an FPGA.

Following, a portion of the script adopted in order to assign the 0 values to the aforementioned registers. It's important to notice that not every register that didn't have an assigned value at reset was forcibly reset by the script, only those that were crucial in the correct functioning of the peripheral were considered.

```
set PATH /can_testbench/i_can_top/i_can_registers

## Bus Timing 0
force -deposit sim:$PATH/sync_jump_width 2'h0 0
force -deposit sim:$PATH/baud_r_presc 6'h00 0

## Bus Timing 1
force -deposit sim:$PATH/triple_sampling 1'h0 0
force -deposit sim:$PATH/time_segment2 3'h0 0
force -deposit sim:$PATH/time_segment1 4'h0 0

## ACR 0 to 3
force -deposit sim:$PATH/acceptance_code_0 8'h00 0
force -deposit sim:$PATH/acceptance_code_1 8'h00 0
force -deposit sim:$PATH/acceptance_code_2 8'h00 0
force -deposit sim:$PATH/acceptance_code_3 8'h00 0

## AMR 0 to 3
force -deposit sim:$PATH/acceptance_mask_0 8'h00 0
force -deposit sim:$PATH/acceptance_mask_1 8'h00 0
force -deposit sim:$PATH/acceptance_mask_2 8'h00 0
force -deposit sim:$PATH/acceptance_mask_3 8'h00 0
```

Listing 2.1. Portion of the TCL script used in simulation

Once the simulations worked in every possible scenario, it was time to briefly work on the fault simulation analysis.

2.4 Standalone Controller Fault Simulation

2.4.1 Fault Simulation Generalities

The fault simulation is a process launched by means of a specific tool that, given a file containing patterns of DUT signals toggling during simulation time, is capable of evaluating the percentage of covered faults belonging to a certain fault class.
In this work, the adopted tool to perform the whole fault simulation was *Tetramax* and the targeted class of faults was the stuck-at faults, that is, faults that occur whenever a certain line is stuck at a logic 1 or logic 0 value.

The typical flow of a fault simulation process is the following: after reading the technology library file used when synthetizing the IC and the synthetized IC itself, it then, after some analysis, takes as input the dumpports file containing the patterns obtained at simulation time and it then performs a second simulation. This simulation is conducted using the pattern source and it reports any differences between the simulated and expected values.
Finally, it will load all of the faults belonging to a specified class, in this case stuck-at faults, and it will go through the patterns to see how many of the loaded faults are excited and made observable at, at least, one of the output ports of the DUT. In the end, it's possible to export the final reports to `txt` files, eventually even the total list of all the faults that will be divided into few categories like detected, not-detected or undetectable faults.
Generally speaking, patterns are obtained when simulating the DUT by running the command `vcd dumpports`, specifying the name of the file in which said patterns will be saved. An example of script used to automate such process is presented below.

```
## Environment setup
set_environment_viewer -instance_names
set_messages -log $PATH/fault_reports/tmax.log -replace

## Build and DRC
read_netlist $PATH/gate/NangateOpenCellLibrary.v -library
read_netlist $PATH/gate/can_top_ST.v -master_module
run_build_model can_top
add_clocks 0 clk_i
set_drc -initialize_dff_dlat 0
run_drc

## Load and Check Patterns
set_patterns -external $PATH/sim/gate_run/dumpports_gate_0.can_top.
   vcd -sensitive -strobe_rising clk_i -strobe_offset {62500 ps}
run_simulation -sequential

## Fault List
```

```
add_faults -all
#add_faults /i_can_registers
#add_faults /i_can_btl
#add_faults /i_can_bsp/i_can_acf
#read_faults fault_reports/report_faults_list.txt -
   force_retain_code -add

set_simulation -num_processes 7

## Fault Simulation
run_fault_sim -sequential

## Reports
set_faults -fault_coverage
report_faults -level {5 100} > $PATH/fault_reports/
   report_faults_hierarchy.txt
report_faults -level {100 1} -verbose > $PATH/fault_reports/
   report_faults_verbose.txt
write_faults $PATH/fault_reports/report_faults_list.txt -all -
   replace
report_summaries > $PATH/fault_reports/report_summaries.txt

## End of Script
quit
```

Listing 2.2. TCL script used in the fault simulation process

2.4.2 Standalone Fault Simulation Issues

The fault simulation process has been intensively used later on in the work, when the final *C language* test program was completed. In fact, at this point of the work, the covered faults analysis was still not relevant, mainly because it had to be performed by means of software programs based on drivers, both of them still missing in this early stage. Anyway, even though it was not necessary, the decision to launch some fault simulations was still made due to the `force -deposit` command that was used when simulating. The reason of this can be found in the description of the internal simulation *Tetramax* executes before performing the actual fault simulation: if there're mismatches between the presented patterns and the internal simulation the whole process aborts.
Since 0 values were forced on some registers, it was necessary to replicate the same behavior in *Tetramax*, thus motivating the need to launch some simulations just to check whether there were no mismatches, ultimately leading to a successful fault simulation.

In order to avoid such problem, a command - present on the script above - was added, that is `set_drc -initialize_dff_dlat 0`, which assigns a 0 to every flip-flop's output present in the DUT at the very beginning of the simulation, allowing it to change later on. In this way, no mismatch was present and the process could finish without problems. The obtained test coverage related to the Verilog testbench showed that about 51% of the total stuck-at faults was tested. The achieved test coverage was not of interest in this stage, as the tests served only to check the correctness of the automation script, so no further attempt to improve the tasks defined in the testbench were made.

Chapter 3

Peripheral Integration and Drivers Development

The typical application in which CAN Controller peripherals are employed involve the presence of a microprocessor that elaborates the information received and drives the peripheral accordingly, along with other actuators that could be present in the system. In more modern scenarios, instead of having many ICs in a PCB, the microprocessor and its most important peripherals are connected together in a System on Chip.

In this chapter, two main topics will be described: the integration process of the CAN Controller in an already existing SoC and its driver development.

3.1 SoC Integration

Following the approach adopted with the CAN Bus peripheral, the System On Chip that has been used in this thesis work is the *OpenRisc1200* an open hardware device that can be found online at the OpenRisc website[1].
This particular SoC was chosen not only because it's open source but also because it uses a Wishbone Bus to connect the CPU with other peripheral controllers. This allowed the direct connection of the CAN peripheral to the SoC, without the need of bridges between different bus interfaces. Even though the two modules share the same interface, there were still some small problems that had to be fixed.

[1] https://openrisc.io/implementations.html

3.1.1 Interfaces mismatch

The first one is related to the interface's signals: the typical bus interface adopted in the SoC is comprised of 10 signals, along with the `wb_clk_i` and `wb_rst_i` signals that are the wishbone clock and reset signals:

- `wb_cyc_o`: 1-bit bus cycle signal
- `wb_stb_o`: 1-bit strobe (kind of chip select) signal
- `wb_cab_o`: 1-bit constant address burst signal (used for FIFOs)
- `wb_adr_o`: 32-bits address signal
- `wb_sel_o`: 4-bits select input array signal
- `wb_we_o`: 1-bit write enable signal
- `wb_dat_o`: 32-bits data output signal
- `wb_dat_i`: 32-bits data input signal
- `wb_ack_i`: 1-bit acknowledge signal
- `wb_err_i`: 1-bit error signal

where the `i` in the signals' name represents all of those signals that go from the peripheral to the Bus arbiter, while the `o` stands for its viceversa.
The Wishbone interface implemented on the CAN Controller interface, on the other hand, implements only a subset of those signals, in particular: `wb_clk`, `wb_rst`, `wb_dat_i`, `wb_dat_o`, `wb_cyc_i`, `wb_stb_i`, `wb_we_i`, `wb_adr_i`, `wb_ack_o`; moreover, the three 32-bits signals `wb_adr_o`, `wb_dat_o`, `wb_dat_o` are on 8-bits in the CAN Wishbone interface.

This obviously lead to some adjustments while connecting the two modules: while the *data_in* and *data_out* signals were easy to adapt - the 8 least significant bits of the *data_out* signal were provided to the relative input port of the CAN Controller and the *data_out* signal from the CAN Controller was extended with 24 most significant bits - the address was a little less "immediate" than the other two. The reason for this is due to the fact that the SoC bus arbiter implies that every register is 4-bytes wide, thus being able to generate only addresses multiple of 4, i.e. `0x00`, `0x04`, `0x08`, `0x0C` and so on. The CAN Controller instead has 1-byte registers, thus having even in between values as valid addresses, like `0x01`, `0x02` or `0x03`. To solve this problem a 2 position right shift - i.e., a division by four - was performed,

in order to map correctly the addresses.

The final interconnection is shown in figure 3.1.

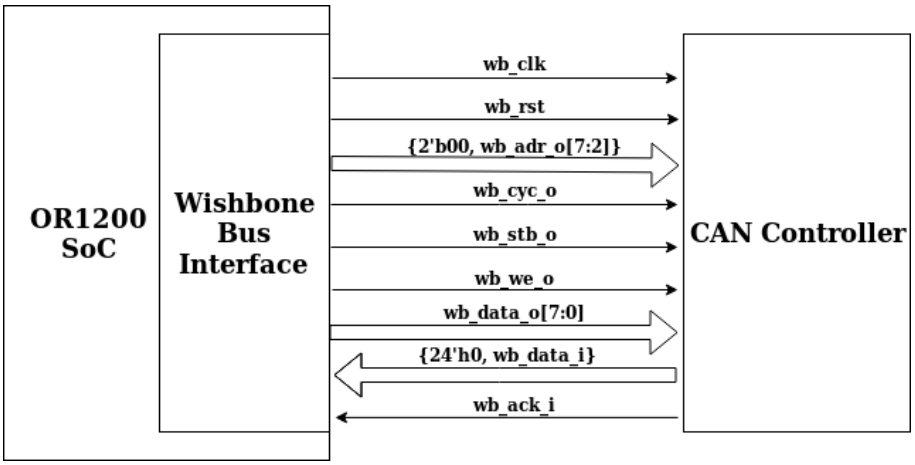

Figure 3.1. CAN Interface Wiring

3.1.2 Address Mapping

Once the wiring between the two modules was completed, it was time to set some parameters in order to correctly address the CAN peripheral. Among the files that compose the SoC, there's a file named `xsv_fpga_defines.v` in which some constant values are defined. In particular, there're two group of parameters that are of interest, one related to interrupts and the other one related to the memory map of the CPU.

In the following list it's possible to see said parameters.

```
//
// Interrupts
//
`define APP_INT_CAN     0
`define APP_INT_RES1    1
//`define APP_INT_UART  2
`define APP_INT_RES2    3
`define APP_INT_ETH     4
//`define APP_INT_PS2   5
//`define APP_INT_HDLC1 6
//`define APP_INT_HDLC2 7
`define APP_INT_RES3    19:8
```

```
//
// Address map
//
`define APP_ADDR_DEC_W  8
`define APP_ADDR_SRAM   `APP_ADDR_DEC_W'h00
`define APP_ADDR_FLASH  `APP_ADDR_DEC_W'h04
`define APP_ADDR_DECP_W  4
`define APP_ADDR_PERIP  `APP_ADDR_DEC_W'h9
`define APP_ADDR_VGA    `APP_ADDR_DEC_W'h97
`define APP_ADDR_ETH    `APP_ADDR_DEC_W'h92
`define APP_ADDR_AUDIO  `APP_ADDR_DEC_W'h9D
//`define APP_ADDR_UART `APP_ADDR_DEC_W'h90
`define APP_ADDR_PS2    `APP_ADDR_DEC_W'h94
//`define APP_ADDR_HDLC `APP_ADDR_DEC_W'h9E
`define APP_ADDR_CAN    `APP_ADDR_DEC_W'h9F
```

Listing 3.1. Snippet from the xsv_fpga_defines.v file

The line `` `define APP_INT_CAN 0 `` is needed when building the vector containing all of the interrupt lines from the peripherals connected to the CPU: in particular, the CPU can support 20 interrupt requests and the CAN Controller's one is the first bit of said vector.
The other added line is `` `define APP_ADDR_CAN `APP_ADDR_DEC_W'h9F `` which can be interpreted as: the starting address of the CAN Peripheral is `0x9F000000`. This parameter will be important when writing the *C* drivers for the peripheral.

3.2 C Drivers

The adopted C drivers have been developed by referring to the open source drivers provided by *NXP* for their SJA1000 Controller[2]. This was necessary in order to understand which are the required functionalities and reproduce them on the adopted CAN Controller, since the online drivers could not be directly applied to the peripheral.

The drivers consist of two files, named *can_addr.h* and *can_addr.c*, the former having only declaration statements while the latter implements functions defined in the *.h* file.

[2]https://www.nxp.com/products/interfaces/can-transceivers/stand-alone-can-controller:SJA1000T?tab=Design_Tools_Tab

3.2.1 Header File: can_addr.h

The header file contains the definitions of the internal addresses for the registers provided by the peripheral, the whole set of `#define` statements can be seen in figures 2.2 and 2.3 plus 2.4; moreover, it contains data structures useful to group together data belonging to the same field and functions' declarations.

To organize internal addresses in a clear way, they have been further divided into three blocks: general registers, which address does not change whether in BasiCAN or PeliCAN mode, showed in listing 3.2, BasiCAN registers, registers that can be addressed only when in BasiCAN Mode, showed in figure 3.3 and PeliCAN registers, addressable only in PeliCAN Mode, showed in figure 3.4.

```
typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned int uint32_t;

#define CAN_OK 0
#define CAN_RX_TIMEOUT 1
#define CAN_TX_TIMEOUT 2
#define CAN_ID_NOT_ACCEPTED 3

#define CAN_BASE_ADDR     ((uint32_t)(0x9F000000))

// CAN Addressing space common to both BasiCAN and PeliCAN modes
// Mode, Command, Status, Interrupt registers
#define CAN_GNRAL_MODE_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x00000000 << 2)))
#define CAN_GNRAL_CMD_REG   ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x00000001 << 2)))
#define CAN_GNRAL_STATUS_REG  ((volatile uint32_t*)(CAN_BASE_ADDR +
     (0x00000002 << 2)))
#define CAN_GNRAL_IRQ_REG   ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x00000003 << 2)))

// Bus Timing0, Bus Timing1 registers
#define CAN_GNRAL_TIM0_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x00000006 << 2)))
#define CAN_GNRAL_TIM1_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x00000007 << 2)))

// Clock Divider register
#define CAN_GNRAL_CLKDIV_REG  ((volatile uint32_t*)(CAN_BASE_ADDR +
     (0x0000001F << 2)))
```

Listing 3.2. General registers mapping in can_addr.h file

```
// CAN Addressing space in BasiCAN Mode
// Acceptance Code, Acceptance Mask, Bus Timing0 , Bus Timing1
    registers
#define CAN_BASIC_ACR0_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x00000004 << 2)))
#define CAN_BASIC_ACM0_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x00000005 << 2)))

// TX registers (ID, RTR, DLC, DATA[0:7])
#define CAN_BASIC_TXB0_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x0000000A << 2)))
#define CAN_BASIC_TXB1_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x0000000B << 2)))
#define CAN_BASIC_TXB2_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x0000000C << 2)))
#define CAN_BASIC_TXB3_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x0000000D << 2)))
#define CAN_BASIC_TXB4_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x0000000E << 2)))
#define CAN_BASIC_TXB5_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x0000000F << 2)))
#define CAN_BASIC_TXB6_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x00000010 << 2)))
#define CAN_BASIC_TXB7_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x00000011 << 2)))
#define CAN_BASIC_TXB8_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x00000012 << 2)))
#define CAN_BASIC_TXB9_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x00000013 << 2)))

// RX registers (ID, RTR, DLC, DATA[0:7])
#define CAN_BASIC_RXB0_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x00000014 << 2)))
#define CAN_BASIC_RXB1_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x00000015 << 2)))
#define CAN_BASIC_RXB2_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x00000016 << 2)))
#define CAN_BASIC_RXB3_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x00000017 << 2)))
#define CAN_BASIC_RXB4_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x00000018 << 2)))
#define CAN_BASIC_RXB5_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x00000019 << 2)))
#define CAN_BASIC_RXB6_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x0000001A << 2)))
#define CAN_BASIC_RXB7_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x0000001B << 2)))
#define CAN_BASIC_RXB8_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x0000001C << 2)))
```

```
#define CAN_BASIC_RXB9_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x0000001D << 2)))
```

Listing 3.3. BasiCAN registers mapping in can_addr.h file

```
// CAN Addessign space in PeliCAN Mode
// Interrupt Enable register
#define CAN_EXTND_IRQEN_REG ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x00000004 << 2)))

// Bus Timing0, Bus Timing1, Arbitration Lost Capture, Error Code
    Capture, RX and TX error counter registers
#define CAN_EXTND_ALC_REG ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x0000000B << 2)))
#define CAN_EXTND_ECC_REG ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x0000000C << 2)))
#define CAN_EXTND_EWLR_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x0000000D << 2)))
#define CAN_EXTND_RXEC_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x0000000E << 2)))
#define CAN_EXTND_TXEC_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
    0x0000000F << 2)))

// Extended Frame TX/RX registers
#define CAN_EXTND_TRX_FRINF_REG ((volatile uint32_t*)(CAN_BASE_ADDR
    + (0x00000010 << 2)))
#define CAN_EXTND_TRXID1_REG  ((volatile uint32_t*)(CAN_BASE_ADDR +
    (0x00000011 << 2)))
#define CAN_EXTND_TRXIDE2_REG ((volatile uint32_t*)(CAN_BASE_ADDR +
    (0x00000012 << 2)))
#define CAN_EXTND_TRXIDE3_REG ((volatile uint32_t*)(CAN_BASE_ADDR +
    (0x00000013 << 2)))
#define CAN_EXTND_TRXIDE4_REG ((volatile uint32_t*)(CAN_BASE_ADDR +
    (0x00000014 << 2)))
#define CAN_EXTND_TRXBE1_REG  ((volatile uint32_t*)(CAN_BASE_ADDR +
    (0x00000015 << 2)))
#define CAN_EXTND_TRXBE2_REG  ((volatile uint32_t*)(CAN_BASE_ADDR +
    (0x00000016 << 2)))
#define CAN_EXTND_TRXBE3_REG  ((volatile uint32_t*)(CAN_BASE_ADDR +
    (0x00000017 << 2)))
#define CAN_EXTND_TRXBE4_REG  ((volatile uint32_t*)(CAN_BASE_ADDR +
    (0x00000018 << 2)))
#define CAN_EXTND_TRXBE5_REG  ((volatile uint32_t*)(CAN_BASE_ADDR +
    (0x00000019 << 2)))
#define CAN_EXTND_TRXBE6_REG  ((volatile uint32_t*)(CAN_BASE_ADDR +
    (0x0000001A << 2)))
#define CAN_EXTND_TRXBE7_REG  ((volatile uint32_t*)(CAN_BASE_ADDR +
    (0x0000001B << 2)))
```

```
#define CAN_EXTND_TRXBE8_REG  ((volatile uint32_t*)(CAN_BASE_ADDR +
    (0x0000001C << 2)))

// Standard Frame TX/RX registers
#define CAN_EXTND_TRXIDS2_REG ((volatile uint32_t*)(CAN_BASE_ADDR +
    (0x00000012 << 2)))
#define CAN_EXTND_TRXBS1_REG  ((volatile uint32_t*)(CAN_BASE_ADDR +
    (0x00000013 << 2)))
#define CAN_EXTND_TRXBS2_REG  ((volatile uint32_t*)(CAN_BASE_ADDR +
    (0x00000014 << 2)))
#define CAN_EXTND_TRXBS3_REG  ((volatile uint32_t*)(CAN_BASE_ADDR +
    (0x00000015 << 2)))
#define CAN_EXTND_TRXBS4_REG  ((volatile uint32_t*)(CAN_BASE_ADDR +
    (0x00000016 << 2)))
#define CAN_EXTND_TRXBS5_REG  ((volatile uint32_t*)(CAN_BASE_ADDR +
    (0x00000017 << 2)))
#define CAN_EXTND_TRXBS6_REG  ((volatile uint32_t*)(CAN_BASE_ADDR +
    (0x00000018 << 2)))
#define CAN_EXTND_TRXBS7_REG  ((volatile uint32_t*)(CAN_BASE_ADDR +
    (0x00000019 << 2)))
#define CAN_EXTND_TRXBS8_REG  ((volatile uint32_t*)(CAN_BASE_ADDR +
    (0x0000001A << 2)))

// Acceptance Code and Mask [0:3] registers
#define CAN_EXTND_ACR0_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
   0x00000010 << 2)))
#define CAN_EXTND_ACR1_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
   0x00000011 << 2)))
#define CAN_EXTND_ACR2_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
   0x00000012 << 2)))
#define CAN_EXTND_ACR3_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
   0x00000013 << 2)))
#define CAN_EXTND_ACM0_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
   0x00000014 << 2)))
#define CAN_EXTND_ACM1_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
   0x00000015 << 2)))
#define CAN_EXTND_ACM2_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
   0x00000016 << 2)))
#define CAN_EXTND_ACM3_REG  ((volatile uint32_t*)(CAN_BASE_ADDR + (
   0x00000017 << 2)))

// Other registers
#define CAN_EXTND_RXMSGCNT_REG  ((volatile uint32_t*)(CAN_BASE_ADDR
    + (0x0000001D << 2)))
#define CAN_EXTND_RXBSA_REG ((volatile uint32_t*)(CAN_BASE_ADDR + (
   0x0000001E << 2)))
#define CAN_EXTND_FIFOADDR_REG  ((volatile uint32_t*)(CAN_BASE_ADDR
    + (0x00000020 << 2)))
```

```
#define CAN_EXTND_TXADDR_REG  ((volatile uint32_t*)(CAN_BASE_ADDR +
    (0x00000060 << 2)))
#define CAN_EXTND_RAMF1_REG ((volatile uint32_t*)(CAN_BASE_ADDR + (
   0x0000006D << 2)))
#define CAN_EXTND_RAMF2_REG ((volatile uint32_t*)(CAN_BASE_ADDR + (
   0x0000006E << 2)))
#define CAN_EXTND_RAMF3_REG ((volatile uint32_t*)(CAN_BASE_ADDR + (
   0x0000006F << 2)))
```

Listing 3.4. PeliCAN registers mapping in can_addr.h file

In listing 3.5 it's possible to see the struct data used to work with the peripheral. There are in total four of them, and they are described as follows:

- `CANPeripHandler`: it contains all of the variables necessary to initialize the peripheral among which there are: `CAN_Mode`, which configures the peripheral either in BasiCAN or PeliCAN Mode, `B_TIM0/1`, to initialize the timing section of the controller, `ACR[4]` and `ACM[4]` which are used to configure the acceptance filter.
- `CANIRQHandler`: struct data used to initialize the irq_enable register, i.e. a register that enables interrupt requests whenever their relative conditions occur.
- `TXMSGHandler`: it is used to configure the message to be transmitted, the fields contained in this struct data reflect the portions of the CAN message adjustable by the user, among which there are the `ID`, `DLC` and `TXBuf[8]`.
- `RXMSGHandler`: this struct data is organized exactly as the previous one, with the only difference that this is used to handle received messages rather than transmitted ones.

```
// Handler used to initialize the peripheral
typedef struct {

  uint8_t CAN_Mode; // Selects between BasiCAN (1'b0) and PeliCAN (
     1'b1) mode
  uint8_t LOM;    // Listen Only Mode => 1'b0: the CAN controller
     would give no acknowledge to the CAN-bus, even if a message is
      received successfully ; 1'b1: normal functioning
  uint8_t STM;    // Self Test Mode => 1'b0: the CAN controller
     will perform a successful transmission, even if there is no
     acknowledge received ; 1'b1: an acknowledge is necessary
  uint8_t AFM;    // Acceptance Filter Mode => 1'b0: two filters
     each with the length of 16 bit are active ; 1'b1: one filter
     with the length of 32 bit is active
```

```
  uint8_t CLKOff;   // If this bit is set, Clkout is not enabled
  uint8_t CD;   // Divider for the clokout signal (111 -> clkout =
     clkin, from 000 to 101 divide clkin by multiples of 2 (ie
     2,4,6,8,10,12,14))
  uint8_t ACR[4];   // Acceptance Code
  uint8_t ACM[4];   // Acceptance Mask
  uint8_t B_TIM0;   // Bus Timing0 reg: |SJW1 |SJW0   |BRP5   |BRP4
       |BRP3   |BRP2   |BRP1   |BRP0   |
  uint8_t B_TIM1;   // Bus Timing1 reg: |SAM  |TSEG2_2|TSEG2_1|
     TSEG2_0|TSEG1_3|TSEG1_2|TSEG1_1|TSEG1_0|
  uint8_t EWL;    // Error Warning Limit, value @ reset 0x60 (
     beyond this value an error is raised)
  uint8_t RBSA;   // Rx Buffer Start Address

} CANPeripHandler;

// IRQ Enable handler struct data
typedef struct {

  uint8_t RIE;    // Receive Interrupt Enable
  uint8_t TIE;    // Transmit Interrupt Enable
  uint8_t EIE;    // Error Warning Interrupt Enable
  uint8_t OIE;    // Data Overrun Interrupt Enable
  uint8_t WUIE;   // Wake-Up Interrupt Enable
  uint8_t EPIE;   // Error Passive Interrupt Enable
  uint8_t ALIE;   // Arbitration Lost Interrupt Enable
  uint8_t BEIE;   // Bus Error Interrupt Enable

} CANIRQHandler;

// TX Message Handler struct data
typedef struct {

  uint32_t ID;    // Identifier of the message
  uint8_t DLC;    // Data Length Code: no of bytes to be sent (1-8)
  uint8_t RTR;    // Remote Transfer Request
  uint8_t FF;   // Frame Format (valid only in Extended CAN Mode):
     0 -> STD Frame, 1 -> EXT Frame
  uint8_t TXBuf[8]; // 8 bytes TX buffer

} TXMsgHandler;

// RX Message Handler struct data
typedef struct {

  uint32_t ID;    // Identifier of the message
  uint8_t DLC;    // Data Length Code: no of bytes to be received (
     1-8)
  uint8_t RTR;    // Remote Transfer Request
```

```
  uint8_t FF;    // Frame Format (valid only in Extended CAN Mode):
     0 -> STD Frame, 1 -> EXT Frame
  uint8_t RXBuf[8]; // 8 bytes RX buffer

} RXMsgHandler;
```

Listing 3.5. Data structures in can_addr.h file

In listing 3.6 it's possible to see function prototypes. Their implementation and functionality will be covered in the following subsection.
All of the data structures and functions have been developed in a way such that they provide an *Hardware Abstraction Layer* so that the user doesn't have to concern with the internal registers settings.

```
/*****************************
 **** Function Prototypes ****
 *****************************/

int canPeriphInit(CANPeripHandler* can_handl);
int irqEnable(CANIRQHandler* irq_handl);
int transmitMsg(TXMsgHandler* tx_handl, uint32_t timeout);
int receiveMsg(RXMsgHandler* rx_handl, uint32_t timeout);
int selfTxRx(TXMsgHandler* tx_handl, RXMsgHandler* rx_handl);
```

Listing 3.6. Function prototypes in can_addr.h file

3.2.2 Source File: can_addr.c

This source file contains the definition of the aforementioned functions. As showed in listing 3.6, there are five functions in total that are described as follows.

canPeriphInit

This function is used to initialize the CAN peripheral. The very first step consists in entering the reset mode by writing a `1'b1` in bit 0 of the `Mode Register`: this is necessary because some registers can only be accessed when in Reset Mode. Once entered in such mode, few registers are initialized.
The first one is the `clock_divider` register, in order to set the CAN Mode, BasiCAN or PeliCAN, and the `clockout` outer port configuration, that is whether or not the clockout signal is active, controlled by the `CLKOff` field, and what's the frequency of said signal in terms of working frequency of the peripheral divided by powers of 2, controlled by the `ClockDivider` or `CD` field.
Then, depending on the working mode, the Acceptance Code and Filter registers are initialized; if in PeliCAN Mode The `SelfTest Mode`, `Listen Only Mode` and

`Acceptance Filter Mode` bits are initialized as well.
Lastly, the Bus Timing registers are written and the controller can exit the Reset Mode.

```
// Initialize CAN Controller according to user specifications
int canPeriphInit(CANPeripHandler* can_handl) {

  // Enter Reset Mode
  *(CAN_GNRAL_MODE_REG) = 0x01;

  // Set CLKDIV register
  uint8_t tmp_clkdiv = ((*can_handl).CD & 0x07);
  tmp_clkdiv += (((*can_handl).CLKOff & 0x01) << 3);
  tmp_clkdiv += (((*can_handl).CAN_Mode & 0x01) << 7);

  *(CAN_GNRAL_CLKDIV_REG) = tmp_clkdiv;


  // Set Acceptance Code and Mask registers
  if(!((*can_handl).CAN_Mode & 0x01)) {
    *(CAN_BASIC_ACM0_REG) = (*can_handl).ACM[0];
    *(CAN_BASIC_ACR0_REG) = (*can_handl).ACR[0];
  }
  else {
    *(CAN_GNRAL_MODE_REG) &= ~(7 << 1);   // Reset Self Test Mode,
       Listen Only Mode and Acceptance Filter Mode bits
    *(CAN_GNRAL_MODE_REG) |= (((*can_handl).AFM & 0x01) << 3) |
       (((*can_handl).STM & 0x01) << 2) | (((*can_handl).LOM & 0x01
       ) << 1);
    *(CAN_EXTND_ACM0_REG) = (*can_handl).ACM[0];
    *(CAN_EXTND_ACM1_REG) = (*can_handl).ACM[1];
    *(CAN_EXTND_ACM2_REG) = (*can_handl).ACM[2];
    *(CAN_EXTND_ACM3_REG) = (*can_handl).ACM[3];
    *(CAN_EXTND_ACR0_REG) = (*can_handl).ACR[0];
    *(CAN_EXTND_ACR1_REG) = (*can_handl).ACR[1];
    *(CAN_EXTND_ACR2_REG) = (*can_handl).ACR[2];
    *(CAN_EXTND_ACR3_REG) = (*can_handl).ACR[3];
    *(CAN_EXTND_EWLR_REG) = (*can_handl).EWL;
    *(CAN_EXTND_RXBSA_REG) = (*can_handl).RBSA;
  }

  // Set Bus Timing 0/1 registers
  *(CAN_GNRAL_TIM0_REG) = (*can_handl).B_TIM0;
  *(CAN_GNRAL_TIM1_REG) = (*can_handl).B_TIM1;

  // Exit Reset Mode
  *(CAN_GNRAL_MODE_REG) &= ~(0x01);

  return CAN_OK;
```

```
}
```

Listing 3.7. canPeriphInit function

irqEnable

This function is used to initialize the register that enables interrupt requests. Depending on the mode there are a certain number of interrupts allowed and for every interrupt there's its relative enabler, as it can be seen in the `CANIRQHandler` struct data from listing 3.5.
When in BasiCAN Mode, interrupt enable bits reside in the `Mode Register`: since this has already been initialized, it's copied in a temporary variable, the relative interrupt enables are added and then the register is re-written.
In PeliCAN Mode on the other hand the `IRQ Enable` register is on its own so the writing process is quite straightforward.

```
// Enable all of the interrupts requested by the user
int irqEnable(CANIRQHandler* irq_handl) {

  // Check CAN Mode
  volatile uint8_t can_mode = (*(CAN_GNRAL_CLKDIV_REG) & (1 << 7))
     >> 7;

  if(!can_mode) { // If we're in BasiCAN mode
    uint8_t tmp_irqen = (((*irq_handl).RIE & 0x01) << 1); // Write
       in a temporary variable all interrupt enables
    tmp_irqen += (((*irq_handl).TIE & 0x01) << 2);
    tmp_irqen += (((*irq_handl).EIE & 0x01) << 3);
    tmp_irqen += (((*irq_handl).OIE & 0x01) << 4);

    volatile uint8_t mode_reg = *(CAN_GNRAL_MODE_REG); // Copy mode
        register (we will overwrite interrupts and keep everything
       else)
    mode_reg &= ~(0x0F << 1);
    mode_reg += tmp_irqen;

    *(CAN_GNRAL_MODE_REG) = mode_reg;
  }
  else { // If we're in PeliCAN Mode, enable
    uint8_t tmp_irqen = (((*irq_handl).RIE & 0x01) << 0);
    tmp_irqen += (((*irq_handl).TIE & 0x01) << 1);
    tmp_irqen += (((*irq_handl).EIE & 0x01) << 2);
    tmp_irqen += (((*irq_handl).OIE & 0x01) << 3);
    tmp_irqen += (((*irq_handl).WUIE & 0x01) << 4);
    tmp_irqen += (((*irq_handl).EPIE & 0x01) << 5);
    tmp_irqen += (((*irq_handl).ALIE & 0x01) << 6);
    tmp_irqen += (((*irq_handl).BEIE & 0x01) << 7);
```

```
    *(CAN_EXTND_IRQEN_REG) = tmp_irqen;
  }


  return CAN_OK;
}
```

Listing 3.8. irqEnable function

transmitMsg

This function is used to send a message which fields have been written by using a `TXMsgHandler` struct data. Along with the `tx_handler` input variable, a `timeout` is required as this function works in polling mode. The transmitting structure initialization highly depends on the mode.
If working in BasiCAN Mode, the ID's 8 MSBs are written in `CAN_BASIC_TXB0_REG`, the lower 3 LSBs of the ID, the DLC and RTR are written in `CAN_BASIC_TXB1_REG` and, lastly, the Data Bytes are written in remaining registers `CAN_BASIC_TXBx_REG`, `x` $\in$ [2,9].
If working in PeliCAN Mode the process is a little bit more complex as the Frame Format has to be taken into account when handling registers, even though the final goal is the same.
Lastly, the actual transmission can begin: once the Start Of Transmission has occurred, the CPU checks periodically the `Transmission Complete Status` bit: if this does not occur before the timeout then the peripheral aborts the transmission and exits with an error condition, otherwise it can exit with a successful return condition.

```
// Function to send a message in polling mode
int transmitMsg(TXMsgHandler* tx_handl, uint32_t timeout) {

  volatile uint8_t can_mode = (*(CAN_GNRAL_CLKDIV_REG) & (1 << 7))
     >> 7;

  if(!can_mode) {
    // Decouple the ID into two registers: TXB0 holds ID[10:3]
    *(CAN_BASIC_TXB0_REG) = ((*tx_handl).ID & 0x000007F8) >> 3;

    // TXB1 holds ID[2:0], RTR, DLC[3:0]
    uint32_t tmp_txb1 = (*tx_handl).DLC;
    tmp_txb1 += ((*tx_handl).RTR & 0x01) << 4;
    tmp_txb1 += ((*tx_handl).ID & 0x00000007) << 5;
    *(CAN_BASIC_TXB1_REG) = tmp_txb1;

    // Now write the data buffer
    *(CAN_BASIC_TXB2_REG) = (*tx_handl).TXBuf[0];
```

```
    *(CAN_BASIC_TXB3_REG) = (*tx_handl).TXBuf[1];
    *(CAN_BASIC_TXB4_REG) = (*tx_handl).TXBuf[2];
    *(CAN_BASIC_TXB5_REG) = (*tx_handl).TXBuf[3];
    *(CAN_BASIC_TXB6_REG) = (*tx_handl).TXBuf[4];
    *(CAN_BASIC_TXB7_REG) = (*tx_handl).TXBuf[5];
    *(CAN_BASIC_TXB8_REG) = (*tx_handl).TXBuf[6];
    *(CAN_BASIC_TXB9_REG) = (*tx_handl).TXBuf[7];
  }
  else {
    // TX Frame Information Regsiter holds the DLC in its 4 LSBs
       and the RTR and FF in the 6th/7th bits
    uint8_t tx_frame_info = (*tx_handl).DLC;
    tx_frame_info += ((*tx_handl).RTR & 0x01) << 6;
    tx_frame_info += ((*tx_handl).FF & 0x01) << 7;

    *(CAN_EXTND_TRX_FRINF_REG) = tx_frame_info;

    // Now write Identifier registers depending whether we've a
       Standard Frame Format (SFF) or Extended Frame Format (EFF)
    *(CAN_EXTND_TRXID1_REG) = ((*tx_handl).ID & 0x1FE00000) >> 21;

    if((*tx_handl).FF & 0x01) { // If we've EFF
      *(CAN_EXTND_TRXIDE2_REG) = ((*tx_handl).ID & 0x001FE000) >>
         13;
      *(CAN_EXTND_TRXIDE3_REG) = ((*tx_handl).ID & 0x00001FE0) >> 5
         ;

      uint8_t trxide4 = 0x00;
      trxide4 |= ((*tx_handl).RTR & 0x01) << 2;
      trxide4 |= ((*tx_handl).ID & 0x1F) << 3;
      *(CAN_EXTND_TRXIDE4_REG) = trxide4 ;

      *(CAN_EXTND_TRXBE1_REG) = (*tx_handl).TXBuf[0];
      *(CAN_EXTND_TRXBE2_REG) = (*tx_handl).TXBuf[1];
      *(CAN_EXTND_TRXBE3_REG) = (*tx_handl).TXBuf[2];
      *(CAN_EXTND_TRXBE4_REG) = (*tx_handl).TXBuf[3];
      *(CAN_EXTND_TRXBE5_REG) = (*tx_handl).TXBuf[4];
      *(CAN_EXTND_TRXBE6_REG) = (*tx_handl).TXBuf[5];
      *(CAN_EXTND_TRXBE7_REG) = (*tx_handl).TXBuf[6];
      *(CAN_EXTND_TRXBE8_REG) = (*tx_handl).TXBuf[7];
    }
    else { // If we've SFF
      uint8_t trxids2 = 0x00;
      trxids2 |= ((*tx_handl).RTR & 0x01) << 4;
      trxids2 |= ((*tx_handl).ID & 0x001C0000) >> 13;
      *(CAN_EXTND_TRXIDS2_REG) = trxids2 ;

      *(CAN_EXTND_TRXBS1_REG) = (*tx_handl).TXBuf[0];
      *(CAN_EXTND_TRXBS2_REG) = (*tx_handl).TXBuf[1];
```

```
      *(CAN_EXTND_TRXBS3_REG) = (*tx_handl).TXBuf[2];
      *(CAN_EXTND_TRXBS4_REG) = (*tx_handl).TXBuf[3];
      *(CAN_EXTND_TRXBS5_REG) = (*tx_handl).TXBuf[4];
      *(CAN_EXTND_TRXBS6_REG) = (*tx_handl).TXBuf[5];
      *(CAN_EXTND_TRXBS7_REG) = (*tx_handl).TXBuf[6];
      *(CAN_EXTND_TRXBS8_REG) = (*tx_handl).TXBuf[7];
    }
  }

  uint32_t cnt = 0;
  volatile uint32_t* status_ptr = CAN_GNRAL_STATUS_REG;

  // Send msg
  *(CAN_GNRAL_CMD_REG) = 0x01;

  // Wait for SOT (Start of Transmission)
  while(!((*status_ptr & 0x20) >> 5));

  // Once the peripheral has started the transmission, wait for it
     to finish/the timeout
  //while((((*status_ptr) & 0x20) >> 5 || ((*status_ptr) & 0x40) >>
      6) && (cnt < timeout)) ++cnt;
  while(!(((*status_ptr) & 0x08) >> 3) && (cnt < timeout)) ++cnt;


  // If there's a timeout, abort transmission and exit
  if(cnt >= timeout) {
    *(CAN_GNRAL_CMD_REG) = 0x02;
    return CAN_TX_TIMEOUT;
  }

  // Otherwise, everything's ok
  return CAN_OK;

}
```

Listing 3.9. transmitMsg function

receiveMsg

This function is used to receive a message. The adopted steps are the same that can be found in the `transmitMsg` function, but reversed.
At first, the receiving is performed by looking at a specific bit that signals the End of Reception: if this occurs before the timeout the received message is decomposed into its fields depending on the working mode, otherwise the program exits with an error condition.

```
// Function to receive a message in polling mode
```

```
int receiveMsg(RXMsgHandler* rx_handl, uint32_t timeout) {

  /* Check the status register and the counter, exit the loop if
     there's currently a message
   * being received or if the counter has received its max value (
     timeout reached)
   */
  uint32_t cnt = 0;
  volatile uint32_t* status_ptr = CAN_GNRAL_STATUS_REG;
  while(!((*status_ptr) & 0x01) && (cnt < timeout)) ++cnt;    //
     Check whether we've received a message or not

  // If we have a timeout, exit with error...
  if (cnt == timeout)
    return CAN_ID_NOT_ACCEPTED;
  // ... otherwise, save into the struct data the ID and the buffer
     and release it
  else {

    volatile uint8_t can_mode = (*(CAN_GNRAL_CLKDIV_REG) & (1 << 7)
       ) >> 7;

    if(!can_mode) {                  // If we're in BasiCAN Mode
      (*rx_handl).DLC = *(CAN_BASIC_RXB1_REG) & 0x0F;       //
         Extract the DLC field
      (*rx_handl).RTR = (*(CAN_BASIC_RXB1_REG) & 0x10) >> 4;
         // Extract the RTR field

      uint32_t tmp_id = (*(CAN_BASIC_RXB1_REG) & 0xE0) >> 5;
         // Extract the ID field
      tmp_id += *(CAN_BASIC_RXB0_REG) << 3;
      (*rx_handl).ID = tmp_id;

      (*rx_handl).RXBuf[0] = *(CAN_BASIC_RXB2_REG);       // RX
         Buffer 1
      (*rx_handl).RXBuf[1] = *(CAN_BASIC_RXB3_REG);       // RX
         Buffer 2
      (*rx_handl).RXBuf[2] = *(CAN_BASIC_RXB4_REG);       // RX
         Buffer 3
      (*rx_handl).RXBuf[3] = *(CAN_BASIC_RXB5_REG);       // RX
         Buffer 4
      (*rx_handl).RXBuf[4] = *(CAN_BASIC_RXB6_REG);       // RX
         Buffer 5
      (*rx_handl).RXBuf[5] = *(CAN_BASIC_RXB7_REG);       // RX
         Buffer 6
      (*rx_handl).RXBuf[6] = *(CAN_BASIC_RXB8_REG);       // RX
         Buffer 7
      (*rx_handl).RXBuf[7] = *(CAN_BASIC_RXB9_REG);       // RX
         Buffer 8
```

```
}
else {                      // Else, if in PeliCAN Mode
  (*rx_handl).FF = (*(CAN_EXTND_TRX_FRINF_REG) & (1 << 7)) >> 7
     ;    // Extract FF, RTR and DLC
  (*rx_handl).RTR = (*(CAN_EXTND_TRX_FRINF_REG) & (1 << 6)) >>
     6;
  (*rx_handl).DLC = *(CAN_EXTND_TRX_FRINF_REG) & 0x0F;
  uint32_t tmp_id = (*(CAN_EXTND_TRXID1_REG)) << 21;       //
     Begin to extract ID

  if((*rx_handl).FF) {              // If EFF
    tmp_id |= (*(CAN_EXTND_TRXIDE2_REG)) << 13;     // Complete
       ID extraction
    tmp_id |= (*(CAN_EXTND_TRXIDE3_REG)) << 5;
    tmp_id |= (*(CAN_EXTND_TRXIDE4_REG) & 0xF8) >> 3;
    (*rx_handl).ID = tmp_id;

    (*rx_handl).RXBuf[0] = *(CAN_EXTND_TRXBE1_REG);     //
       Write received data in rx_handl
    (*rx_handl).RXBuf[1] = *(CAN_EXTND_TRXBE2_REG);
    (*rx_handl).RXBuf[2] = *(CAN_EXTND_TRXBE3_REG);
    (*rx_handl).RXBuf[3] = *(CAN_EXTND_TRXBE4_REG);
    (*rx_handl).RXBuf[4] = *(CAN_EXTND_TRXBE5_REG);
    (*rx_handl).RXBuf[5] = *(CAN_EXTND_TRXBE6_REG);
    (*rx_handl).RXBuf[6] = *(CAN_EXTND_TRXBE7_REG);
    (*rx_handl).RXBuf[7] = *(CAN_EXTND_TRXBE8_REG);
  }
  else {                    // If SFF
    tmp_id |= (*(CAN_EXTND_TRXIDS2_REG) & 0xE0) << 13;    //
       Complete ID extraction
    (*rx_handl).ID = tmp_id;

    (*rx_handl).RXBuf[0] = *(CAN_EXTND_TRXBS1_REG);     //
       Write received data in rx_handl
    (*rx_handl).RXBuf[1] = *(CAN_EXTND_TRXBS2_REG);
    (*rx_handl).RXBuf[2] = *(CAN_EXTND_TRXBS3_REG);
    (*rx_handl).RXBuf[3] = *(CAN_EXTND_TRXBS4_REG);
    (*rx_handl).RXBuf[4] = *(CAN_EXTND_TRXBS5_REG);
    (*rx_handl).RXBuf[5] = *(CAN_EXTND_TRXBS6_REG);
    (*rx_handl).RXBuf[6] = *(CAN_EXTND_TRXBS7_REG);
    (*rx_handl).RXBuf[7] = *(CAN_EXTND_TRXBS8_REG);
  }

}

*(CAN_GNRAL_CMD_REG) = 0x04;         // Tell the CAN controller
   that the msg has been read
return CAN_OK;
```

```
  }

}
```

Listing 3.10. receiveMsg function

selfTxRx

This function can be used only when the peripheral is in SelfTest mode, that is a mode in which the peripheral can communicate with itself without the need of a second node to give the latter the acknowledge.
It basically consists of a mix of the transmit and receive functions: at first the transmitted message is initialized, then sent and received, lastly the received message is decomposed into its fields.

```
// Transmit and receive a message concurrently in Self Test Mode
int selfTxRx(TXMsgHandler* tx_handl, RXMsgHandler* rx_handl) {

  // TX Frame Information Regsiter holds the DLC in its 4 LSBs and
     the RTR and FF in the 6th/7th bits
        uint8_t tx_frame_info = (*tx_handl).DLC;
        tx_frame_info += ((*tx_handl).RTR & 0x01) << 6;
        tx_frame_info += ((*tx_handl).FF & 0x01) << 7;

        *(CAN_EXTND_TRX_FRINF_REG) = tx_frame_info;

        // Now write Identifier registers depending whether we've a
            Standard Frame Format (SFF) or Extended Frame Format (
           EFF)
        *(CAN_EXTND_TRXID1_REG) = ((*tx_handl).ID & 0x1FE00000) >>
           21;

        if((*tx_handl).FF & 0x01) { // If we've EFF
          *(CAN_EXTND_TRXIDE2_REG) = ((*tx_handl).ID & 0x001FE000)
             >> 13;
                *(CAN_EXTND_TRXIDE3_REG) = ((*tx_handl).ID &
                   0x00001FE0) >> 5;

                uint8_t trxide4 = 0x00;
                trxide4 |= ((*tx_handl).RTR & 0x01) << 2;
                trxide4 |= ((*tx_handl).ID & 0x1F) << 3;
                *(CAN_EXTND_TRXIDE4_REG) = trxide4 ;

                *(CAN_EXTND_TRXBE1_REG) = (*tx_handl).TXBuf[0];
                *(CAN_EXTND_TRXBE2_REG) = (*tx_handl).TXBuf[1];
                *(CAN_EXTND_TRXBE3_REG) = (*tx_handl).TXBuf[2];
                *(CAN_EXTND_TRXBE4_REG) = (*tx_handl).TXBuf[3];
                *(CAN_EXTND_TRXBE5_REG) = (*tx_handl).TXBuf[4];
```

```
                *(CAN_EXTND_TRXBE6_REG) = (*tx_handl).TXBuf[5];
                *(CAN_EXTND_TRXBE7_REG) = (*tx_handl).TXBuf[6];
                *(CAN_EXTND_TRXBE8_REG) = (*tx_handl).TXBuf[7];
        }
        else { // If we've SFF
          uint8_t trxids2 = 0x00;
                trxids2 |= ((*tx_handl).RTR & 0x01) << 4;
                trxids2 |= ((*tx_handl).ID & 0x001C0000) >> 13;
                *(CAN_EXTND_TRXIDS2_REG) = trxids2 ;

                *(CAN_EXTND_TRXBS1_REG) = (*tx_handl).TXBuf[0];
                *(CAN_EXTND_TRXBS2_REG) = (*tx_handl).TXBuf[1];
                *(CAN_EXTND_TRXBS3_REG) = (*tx_handl).TXBuf[2];
                *(CAN_EXTND_TRXBS4_REG) = (*tx_handl).TXBuf[3];
                *(CAN_EXTND_TRXBS5_REG) = (*tx_handl).TXBuf[4];
                *(CAN_EXTND_TRXBS6_REG) = (*tx_handl).TXBuf[5];
                *(CAN_EXTND_TRXBS7_REG) = (*tx_handl).TXBuf[6];
                *(CAN_EXTND_TRXBS8_REG) = (*tx_handl).TXBuf[7];
        }


        // Self TxRX request
        *(CAN_GNRAL_CMD_REG) = 0x10;


        volatile uint32_t* status_ptr = CAN_GNRAL_STATUS_REG;

        // Wait for SOT (Start of Transmission)
        while(!((*status_ptr & 0x20) >> 5));

        // Once the peripheral has started the transmission, wait
           for it to finish
        while((!(((*status_ptr) & 0x08) >> 3) || ((*status_ptr) &
           0x40) >> 6));

// The transmission has now ended, chech if the peripheral has
   received its own message
if((*status_ptr) & 0x01) {

  // Now, save everything in rx_handl
  (*rx_handl).FF = (*(CAN_EXTND_TRX_FRINF_REG) & (1 << 7)) >> 7;
                // Extract FF, RTR and DLC
  (*rx_handl).RTR = (*(CAN_EXTND_TRX_FRINF_REG) & (1 << 6)) >> 6;
  (*rx_handl).DLC = *(CAN_EXTND_TRX_FRINF_REG) & 0x0F;
  uint32_t tmp_id = (*(CAN_EXTND_TRXID1_REG)) << 21;
                           // Begin to extract ID

  if((*rx_handl).FF) {
                                                    // If EFF
```

```
      tmp_id |= (*(CAN_EXTND_TRXIDE2_REG)) << 13;
                                // Complete ID extraction
      tmp_id |= (*(CAN_EXTND_TRXIDE3_REG)) << 5;
      tmp_id |= (*(CAN_EXTND_TRXIDE4_REG) & 0xF8) >> 3;
      (*rx_handl).ID = tmp_id;

      (*rx_handl).RXBuf[0] = *(CAN_EXTND_TRXBE1_REG);
                           // Write received data in rx_handl
      (*rx_handl).RXBuf[1] = *(CAN_EXTND_TRXBE2_REG);
      (*rx_handl).RXBuf[2] = *(CAN_EXTND_TRXBE3_REG);
      (*rx_handl).RXBuf[3] = *(CAN_EXTND_TRXBE4_REG);
      (*rx_handl).RXBuf[4] = *(CAN_EXTND_TRXBE5_REG);
      (*rx_handl).RXBuf[5] = *(CAN_EXTND_TRXBE6_REG);
      (*rx_handl).RXBuf[6] = *(CAN_EXTND_TRXBE7_REG);
      (*rx_handl).RXBuf[7] = *(CAN_EXTND_TRXBE8_REG);
    }
    else {

       // If SFF
      tmp_id |= (*(CAN_EXTND_TRXIDS2_REG) & 0xE0) << 13;
                         // Complete ID extraction
      (*rx_handl).ID = tmp_id;

      (*rx_handl).RXBuf[0] = *(CAN_EXTND_TRXBS1_REG);
                           // Write received data in rx_handl
      (*rx_handl).RXBuf[1] = *(CAN_EXTND_TRXBS2_REG);
      (*rx_handl).RXBuf[2] = *(CAN_EXTND_TRXBS3_REG);
      (*rx_handl).RXBuf[3] = *(CAN_EXTND_TRXBS4_REG);
      (*rx_handl).RXBuf[4] = *(CAN_EXTND_TRXBS5_REG);
      (*rx_handl).RXBuf[5] = *(CAN_EXTND_TRXBS6_REG);
      (*rx_handl).RXBuf[6] = *(CAN_EXTND_TRXBS7_REG);
      (*rx_handl).RXBuf[7] = *(CAN_EXTND_TRXBS8_REG);
    }

  }


  return CAN_OK;

}
```

Listing 3.11. selfTxRx function

Chapter 4

SelfTest Library

This chapter is centered on the development of the SelfTest Library, that is the test program that will be used to functionally test the CAN Controller peripheral. In particular, all the subprograms in which such program is divided will be explained both in terms of their implementation and the targeted submodules of the CAN peripheral.

4.1 Case Study

The SelfTest Library highly depends on the adopted testbench configuration. In fact, if every functionality described in the aforementioned drivers was to be used, functions like `receiveMsg` or `transmitMsg` imply the presence of more than one node on the bus.
The CAN Controller offers the possibility of sending and receiving messages without any other node, this operating mode is the `Selftest Mode`, but this wouldn't let the test of the controller logic related to normal transmission and reception of messages. Since a typical application in which CAN communication is required involves many nodes, the same assumption was made for this work.
In the final test-bench used in the experiments, two instances of the OR1200 based SoC are connected through a CAN bus. The reason of having only two nodes is due to the fact that this is the minimum number that allows to test every operating mode without introducing any unnecessary complexity.
The whole system was simulated in RT level, with the exception of the CAN controller of the target SoC (one of the two SoCs is assumed to be the active node, while the other SoC is the passive node), which is simulated at the gate level. Since the test-bench implements an end-to-end communication between two devices, the two SoCs own distinct IDs.
A simple scheduling approach is implemented, which iterates on all the available

test programs. A graphical view of the test environment is depicted in 4.1.

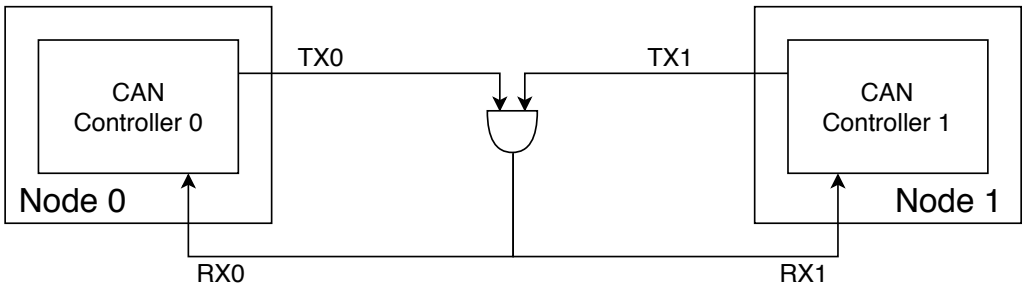

Figure 4.1. Block diagram of the test-bench.

It's important to notice that this schematic is just a logic representation of how the two nodes have been connected in the testbench from a CAN bus point of view: in reality each node's `Tx` and `Rx` signals should be connected to a CAN transceiver that converts the two lines into `CANH` and `CANL` as explained in 1; figure 1.2 shows the concept in a simplified way.

4.2 Test Program

The following test program is installed both in the active and passive nodes. Each program thus comes in two fashions, even though the executed steps are - almost always - symmetric: the two nodes alternatively receive and send messages in the same exact configuration, depending on whether a node is active or passive it will transmit first or receive first.

The main test program is divided into subprograms that are specialized in a specific sub-module or functional feature of the CAN controller. They can be executed together as a whole, grouped in submodules or by themselves one after the other: the idea is that, when the CAN communication is idle, the test can occur and depending on the duration of the idle time one could decide on the test length and hence decide which modules to test together.
Below a description of the subprograms that compose the STL.

4.2.1 Bit Rate test

In this test program, the active node sends a message using multiple bit rates without varying the other configuration parameters. In this experiment, the CAN controllers were configured to work in BasiCAN mode; other parameters like the ID and the data length (DLC) fields were fixed. The Data Bytes, on the other hand, changed

at every transmitted message. For each iteration cycle, the active node transmits a message and expects an acknowledgment from the passive node which will then transmit a message to the active one. At the end of such cycle, the transmission bit rate is modified coherently.

Once tested the ability of the node to work at various bit rates, in order to speed-up the test, the nodes are configured to the fastest possible bit rate for all the remaining phases of the test.

```
// Loop over different bitrates
int bitrate_it;
for(bitrate_it = 0; bitrate_it < 8; bitrate_it ++) {

  // Initialize the peripheral
  can_handl.CAN_Mode = 0x00;
  can_handl.CLKOff = 0x01;
  can_handl.CD = 0x00;

  int count;
  for(count = 0; count < 4; count++) {
    can_handl.ACR[count] = 0x00;
    can_handl.ACM[count] = 0xFF;
  }
  can_handl.B_TIM0 = bustim0[bitrate_it];
  can_handl.B_TIM1 = bustim1[bitrate_it];
  canPeriphInit(&can_handl);

  // Enable all interrupts
  irq_handl.RIE  = 0x00;
  irq_handl.TIE  = 0x00;
  irq_handl.EIE  = 0x00;
  irq_handl.OIE  = 0x00;
  irq_handl.WUIE = 0x00;
  irq_handl.EPIE = 0x00;
  irq_handl.ALIE = 0x00;
  irq_handl.BEIE = 0x00;
  irqEnable(&irq_handl);

        tx_handl.ID = ID_bitrate[bitrate_it];
        tx_handl.DLC = 0x08;
        tx_handl.RTR = 0x00;
        tx_handl.FF = 0x00;

  for(i = 0; i < 8; i++) {
    tx_handl.TXBuf[i] = lfsr[i];
    lfsr[i] = (lfsr[i] << 1) | (((lfsr[i] & (1 << 6)) >> 6) ^ ((
       lfsr[i] & (1 << 4)) >> 4) ^ ((lfsr[i] & (1 << 2)) >> 2) ^
       ((lfsr[i] & (1 << 1)) >> 1));
  }
```

```
    // Check if there's a message to receive
    BRP = bustim0[bitrate_it] & 0x3F;
    TSEG1 = bustim1[bitrate_it] & 0x0F;
    TSEG2 = (bustim1[bitrate_it] & 0x70) >> 4;
    timeout = 2*(BRP+1)*(TSEG1+TSEG2+3)*130;

    if(receiveMsg(&rx_handl,timeout) != CAN_OK) __asm__("l.addi
       r20,r0,0x20");
    if(transmitMsg(&tx_handl,timeout) != CAN_OK) __asm__("l.addi
       r21,r0,0x21");

    BusTiming0Read = *(CAN_GNRAL_TIM0_REG);
    BusTiming1Read = *(CAN_GNRAL_TIM1_REG);

          // Wait some time to let the two peripherals to be ready
             for the next loop
    for(wait = 0; wait<800; wait++);

}
```

Listing 4.1. Bit Rate Test subprogram

4.2.2 Normal Mode test

This test program is intended to test the basic transmission/reception of messages while changing the configuration parameters. Clearly, active and passive nodes change parameters coherently during the test.

The other parameters are changed, such as the CAN operating mode (i.e., BasiCAN and PeliCAN), the Frame Format when in PeliCAN mode, the DLC and ID and the enabled interrupts. Every node sends and receives one message for every available configuration, that is one message sent and received per cycle. In our experiments, data payload for messages have been filled with pseudo-random values. Additionally, we included some deterministic patterns (e.g., 0101..., 1010..., 0011..., 1100...).

After each transmission, the results are retrieved by reading the appropriate registers (i.e., data and status registers), eventually these can be compared with the expected ones (or compacted in a test signature).
Finally, configuration registers are read back after each change in the configuration to detect faults in the configuration flip-flops.

```
uint8_t can_mode = 0x00;
uint8_t ff = 0x00;
```

```
        // Wait some time to let the two peripherals to be ready
           for the next loop
        for(wait = 0; wait<100; wait++);


// Loop over CAN msgs
int tx_it;
for(tx_it = 0; tx_it < 8; tx_it++) {

  // Initialize the peripheral
  can_handl.CAN_Mode = can_mode;
  can_handl.CLKOff = 0x00;
  can_handl.CD = CD[tx_it];
  can_handl.ACR[0] = ACR0[tx_it];
  can_handl.ACR[1] = ACR1[tx_it];
  can_handl.ACR[2] = ACR2[tx_it];
  can_handl.ACR[3] = ACR3[tx_it];
  can_handl.ACM[0] = ACM0[tx_it];
  can_handl.ACM[1] = ACM1[tx_it];
  can_handl.ACM[2] = ACM2[tx_it];
  can_handl.ACM[3] = ACM3[tx_it];
  can_handl.B_TIM0 = bustim0[0];
  can_handl.B_TIM1 = bustim1[0];
  canPeriphInit(&can_handl);

  // Enable interrupts
  irq_handl.RIE  = IRQ_EN[tx_it] & (1 << 0);
  irq_handl.TIE  = (IRQ_EN[tx_it] & (1 << 1)) >> 1;
  irq_handl.EIE  = (IRQ_EN[tx_it] & (1 << 2)) >> 2;
  irq_handl.OIE  = (IRQ_EN[tx_it] & (1 << 3)) >> 3;
  irq_handl.WUIE = (IRQ_EN[tx_it] & (1 << 4)) >> 4;
  irq_handl.EPIE = (IRQ_EN[tx_it] & (1 << 5)) >> 5;
  irq_handl.ALIE = (IRQ_EN[tx_it] & (1 << 6)) >> 6;
  irq_handl.BEIE = (IRQ_EN[tx_it] & (1 << 7)) >> 7;
  irqEnable(&irq_handl);

        //TXMsgHandler tx_handl;
        tx_handl.ID = ID[tx_it];
        tx_handl.DLC = DLC[tx_it];
        tx_handl.RTR = 0x00;
        tx_handl.FF = ff;

  // Generate pseudo-randomic messages to be sent
  for(i = 0; i < 8; i++) {
    tx_handl.TXBuf[i] = lfsr[i];
    lfsr[i] = (lfsr[i] << 1) | (((lfsr[i] & (1 << 6)) >> 6) ^ ((
       lfsr[i] & (1 << 4)) >> 4) ^ ((lfsr[i] & (1 << 2)) >> 2) ^
       ((lfsr[i] & (1 << 1)) >> 1));
  }
```

```
  // Check if there's a message to receive
              BRP = bustim0[0] & 0x3F;
              TSEG1 = bustim1[0] & 0x0F;
              TSEG2 = (bustim1[0] & 0x70) >> 4;
  t_packet = can_mode ? (60 + DLC[tx_it]*8) : (40 + DLC[tx_it]*8)
     ;
              timeout = (2*(BRP+1)*(TSEG1+TSEG2+3)*t_packet) >> 4
                 ;

  // Receive a message
  if(receiveMsg(&rx_handl, timeout) != CAN_OK) __asm__("l.addi
     r20,r20,0x01");

  // Transmit a message
  if(transmitMsg(&tx_handl, timeout) != CAN_OK) __asm__("l.addi
     r21,r21,0x01");

  // Read out some registers to enhance fault coverage on them
  if(can_mode) {
         IRQ_EN_read[tx_it] = *(CAN_EXTND_IRQEN_REG);
    TxData10 = *(CAN_EXTND_TRXBE6_REG);
    TxData11 = *(CAN_EXTND_TRXBE7_REG);
    TxData12 = *(CAN_EXTND_TRXBE8_REG);

    *(CAN_GNRAL_MODE_REG) |= 0x01;
    ACR_read[0][tx_it] = *(CAN_EXTND_ACR0_REG);
    ACR_read[1][tx_it] = *(CAN_EXTND_ACR1_REG);
    ACR_read[2][tx_it] = *(CAN_EXTND_ACR2_REG);
    ACR_read[3][tx_it] = *(CAN_EXTND_ACR3_REG);
         ACM_read[0][tx_it] = *(CAN_EXTND_ACM0_REG);
         ACM_read[1][tx_it] = *(CAN_EXTND_ACM1_REG);
         ACM_read[2][tx_it] = *(CAN_EXTND_ACM2_REG);
         ACM_read[3][tx_it] = *(CAN_EXTND_ACM3_REG);
    *(CAN_GNRAL_MODE_REG) &= ~(0x01);

    ff ^= 0x01;
  }
  can_mode ^= 0x01;

  ModeReg = *(CAN_GNRAL_MODE_REG);
  BusTiming0Read = *(CAN_GNRAL_TIM0_REG);
  BusTiming1Read = *(CAN_GNRAL_TIM1_REG);
  ClockDivRead = *(CAN_GNRAL_CLKDIV_REG);
}
```

Listing 4.2. Normal Mode Test subprogram

4.2.3 Self-Test Mode test

In the Self-Test Mode configuration, the CAN controller sends messages without the need of an acknowledge by other nodes and uses a loop-back to check their correctness autonomously.
The test consists of a transmission and concurrent reception of a single message, all of this in PeliCAN Mode.

```
/************************************************
 ********** Second Step: Self Test Mode **********
 ************************************************/

// Self Test Mode
      can_handl.CAN_Mode = 0x01;
      can_handl.CLKOff = 0x00;
can_handl.STM = 0x01;
can_handl.AFM = 0x01;
can_handl.LOM = 0x00;
      can_handl.CD = 0x00;
      can_handl.ACR[0] = 0x00;
      can_handl.ACR[1] = 0x00;
      can_handl.ACR[2] = 0x00;
      can_handl.ACR[3] = 0x00;
      can_handl.ACM[0] = 0xFF;
      can_handl.ACM[1] = 0xFF;
      can_handl.ACM[2] = 0xFF;
      can_handl.ACM[3] = 0xFF;
      can_handl.B_TIM0 = bustim0[0];
can_handl.B_TIM1 = bustim1[0];
can_handl.EWL = 0x55;
      canPeriphInit(&can_handl);


tx_handl.ID = (uint32_t)0x1582FBACD;
tx_handl.DLC = 0x08;
tx_handl.RTR = 0x00;
tx_handl.FF = 0x01;

      for(i = 0; i < 8; i++) {
        tx_handl.TXBuf[i] = lfsr[i];
              lfsr[i] = (lfsr[i] << 1) | (((lfsr[i] & (1 << 6))
                 >> 6) ^ ((lfsr[i] & (1 << 4)) >> 4) ^ ((lfsr[i]
                 & (1 << 2)) >> 2) ^ ((lfsr[i] & (1 << 1)) >> 1))
                 ;
      }

      // Self transmission and reception
selfTxRx(&tx_handl,&rx_handl);
```

Listing 4.3. Self-Test subprogram

4.2.4 Listen Only Mode test

In the Listen Only Mode configuration, the CAN controller is only capable of receiving and, more specifically, it does not generate the acknowledge bit even if the message has been correctly received.
The test aims at checking the ability of the node to receive and process a message and consists in the reception and check of a single message sent by the passive node.

```
        /*************************************************
         ********* Third Step: Listen Only Mode **********
         *************************************************/


// Listen Only Mode
        can_handl.CAN_Mode = 0x01;
        can_handl.CLKOff = 0x00;
        can_handl.LOM = 0x01;
can_handl.STM = 0x00;
        can_handl.AFM = 0x01;
        can_handl.CD = 0x00;
        can_handl.ACR[0] = 0x00;
        can_handl.ACR[1] = 0x00;
        can_handl.ACR[2] = 0x00;
        can_handl.ACR[3] = 0x00;
        can_handl.ACM[0] = 0xFF;
        can_handl.ACM[1] = 0xFF;
        can_handl.ACM[2] = 0xFF;
        can_handl.ACM[3] = 0xFF;
        can_handl.B_TIM0 = bustim0[0];
        can_handl.B_TIM1 = bustim1[0];
        can_handl.EWL = 0x3F;
canPeriphInit(&can_handl);


        tx_handl.ID = (uint32_t)0x1582FBACD;
        tx_handl.DLC = 0x08;
        tx_handl.RTR = 0x00;
        tx_handl.FF = 0x01;

        for(i = 0; i < 8; i++) {
                tx_handl.TXBuf[i] = lfsr[i];
                lfsr[i] = (lfsr[i] << 1) | (((lfsr[i] & (1 << 6))
                   >> 6) ^ ((lfsr[i] & (1 << 4)) >> 4) ^ ((lfsr[i]
                   & (1 << 2)) >> 2) ^ ((lfsr[i] & (1 << 1)) >> 1))
                   ;
        }

        // Check if there's a message to receive
        BRP = bustim0[0] & 0x3F;
```

```
        TSEG1 = bustim1[0] & 0x0F;
        TSEG2 = (bustim1[0] & 0x70) >> 4;
        t_packet = 104;
        timeout = (2*(BRP+1)*(TSEG1+TSEG2+3)*t_packet) >> 4;


        if(receiveMsg(&rx_handl,timeout) != CAN_OK) __asm__("l.addi
            r20,r0,0x20");
```

Listing 4.4. Listen Only Mode Test subprogram

4.2.5 FIFO test

The basic principle of the test program for the FIFO consists of filling and then emptying it and working on the overrun generation bit. The overrun occurs when the FIFO is already full and the CAN controller tries to write another message, without success.
In order to implement such principle, the passive node has to send enough messages to fill the FIFO (64 messages in our case study). A further message sent by the passive node should produce an overrun, which can be detected by the active node by reading a proper status register (or by means of an interrupt).

In a second phase, while the second node is sending messages, the first one keeps reading them, in order to test the remaining logic of the FIFO. In our case study, the FIFO is implemented as a circular buffer, thus we sent a first packet of 64 messages and then we repeated this with 200 messages.

```
        /************************************************
         ********** Fourth Step: Overrun Error ***********
         ************************************************/


// Extended Mode RX
        can_handl.CAN_Mode = 0x01;
        can_handl.CLKOff = 0x00;
        can_handl.LOM = 0x00;
        can_handl.STM = 0x00;
        can_handl.AFM = 0x01;
        can_handl.CD = 0x00;
        can_handl.ACR[0] = 0x00;
        can_handl.ACR[1] = 0x00;
        can_handl.ACR[2] = 0x00;
        can_handl.ACR[3] = 0x00;
        can_handl.ACM[0] = 0xFF;
        can_handl.ACM[1] = 0xFF;naruto vs neji
        can_handl.ACM[2] = 0xFF;
        can_handl.ACM[3] = 0xFF;
```

```
        can_handl.B_TIM0 = bustim0[0];
        can_handl.B_TIM1 = bustim1[0];
        can_handl.EWL = 0x3F;
can_handl.RBSA = 0x00;
        canPeriphInit(&can_handl);


// Wait for the FIFO to be completely filled
for(wait=0; wait<28500; wait++);

// Free the FIFO
for(i=0; i<64; i++) receiveMsg(&rx_handl,(uint32_t)1000);


// Standard Mode RX
        can_handl.CAN_Mode = 0x00;
        can_handl.CLKOff = 0x00;
        can_handl.LOM = 0x00;
        can_handl.STM = 0x00;
        can_handl.AFM = 0x00;
        can_handl.CD = 0x00;
        can_handl.ACR[0] = 0x00;
        can_handl.ACR[1] = 0x00;
        can_handl.ACR[2] = 0x00;
        can_handl.ACR[3] = 0x00;
        can_handl.ACM[0] = 0xFF;
        can_handl.ACM[1] = 0xFF;
        can_handl.ACM[2] = 0xFF;
        can_handl.ACM[3] = 0xFF;
        can_handl.B_TIM0 = bustim0[0];
        can_handl.B_TIM1 = bustim1[0];
        can_handl.EWL = 0x3F;
        can_handl.RBSA = 0x00;
        canPeriphInit(&can_handl);

        // Wait for the FIFO to be completely filled
        for(wait=0; wait<31000; wait++);

        // Free the FIFO
        for(i=0; i<64; i++) receiveMsg(&rx_handl,(uint32_t)1000);

// Play with DLC values
for(i=0; i<200; i++) receiveMsg(&rx_handl,(uint32_t)1000);

for(wait=0; wait<500; wait++);
```

Listing 4.5. FIFO Test subprogram

4.2.6 Errors test

This test program aims at testing the logic devoted to detect some error conditions. Since in a functional test environment not assisted by ad-hoc hardware it is not possible to precisely work on external (i.e., coming from the physical BUS) errors, the program mainly focuses on testing the situation in which the two nodes do not have the same bit rate.
The proposed test program configures the active and the passive nodes to different bit rates. Then, the passive node sends a message to the active node, which is incapable of receiving it due to the different bit rates. In our case study, after a certain amount of trials, the active node disables itself and goes into *BUS Off Mode*.

Finally, the active node becomes the transmitter and the passive node the one who is subject to the error. In this case, the acknowledgment is not sent by the passive node.

```
/***********************************************
       ******** Fifth Step: Different Bitrates *********
       ***********************************************/


// We'll simulate here what happens when a message is sent
// and the one who transmits and the one who receives have
// different bitrates

// First sub_step: send the message
can_handl.CAN_Mode = 0x01;
can_handl.CLKOff = 0x01;
can_handl.LOM = 0x00;
can_handl.STM = 0x00;
can_handl.AFM = 0x00;
can_handl.CD = 0x00;
can_handl.ACR[0] = ACR0[1];
can_handl.ACR[1] = ACR1[1];
can_handl.ACR[2] = ACR2[1];
can_handl.ACR[3] = ACR2[1];
can_handl.ACM[0] = ACM0[1];
can_handl.ACM[1] = ACM1[1];
can_handl.ACM[2] = ACM2[1];
can_handl.ACM[3] = ACM2[1];
can_handl.B_TIM0 = bustim0[0];
can_handl.B_TIM1 = bustim1[0];
can_handl.EWL = 0x55;
canPeriphInit(&can_handl);

tx_handl.ID = 0x15555555;
tx_handl.DLC = 0x01;
```

```
tx_handl.FF = 0x01;
tx_handl.RTR = 0x01;

EWL_read = *(CAN_EXTND_EWLR_REG);

for(wait=0; wait<500; wait++);

if(transmitMsg(&tx_handl,(uint32_t)3000) == CAN_TX_TIMEOUT)
    __asm__("l.addi r20,r0,0x20");

// We have to re-enable the peripheral, in order to do so it has
   to down count for 127 cycles
// To save time, we set the timing of the peripheral at the
   maximum speed
*(CAN_GNRAL_TIM0_REG) = 0x00;
*(CAN_GNRAL_TIM1_REG) = 0x00;
*(CAN_GNRAL_MODE_REG) &= ~(0x01);

// Wait for the down count to reach 0
      for(wait=0; wait<1200; wait++);

// Once the peripheral has been re-enabled, it's time to test
   rx_errors
      can_handl.CAN_Mode = 0x01;
      can_handl.CLKOff = 0x01;
      can_handl.LOM = 0x00;
      can_handl.STM = 0x00;
      can_handl.AFM = 0x00;
      can_handl.CD = 0x00;
      can_handl.ACR[0] = ACR0[3];
      can_handl.ACR[1] = ACR1[3];
      can_handl.ACR[2] = ACR2[3];
      can_handl.ACR[3] = ACR2[3];
      can_handl.ACM[0] = ACM0[3];
      can_handl.ACM[1] = ACM1[3];
      can_handl.ACM[2] = ACM2[3];
      can_handl.ACM[3] = ACM2[3];
      can_handl.B_TIM0 = bustim0[4];
      can_handl.B_TIM1 = bustim1[4];
      can_handl.EWL = 0xAA;
canPeriphInit(&can_handl);

      tx_handl.ID = 0x15555555;
      tx_handl.DLC = 0x01;
      tx_handl.FF = 0x01;
      tx_handl.RTR = 0x01;

EWL_read = *(CAN_EXTND_EWLR_REG);
```

```
receiveMsg(&rx_handl,(uint32_t)3500);

      for(wait=0; wait<1300; wait++);
```

Listing 4.6. Error Test subprogram

4.2.7 Arbitration test

This program tests the arbitration between nodes. The proposed approach consists of a series of messages sent by the two peripherals concurrently: this is because the arbitration occurs whenever two or more nodes are transmitting at the same time, loss of arbitration is verified if one node is transmitting a recessive bit, `1'b1`, while the other node is transmitting a dominant bit `1'b0`.
The internal CAN Controller logic regarding the arbitration is centered on a register, `Arbitration Lost Capture` or `ALC`, which bits reflect the position in the CAN message in which arbitration has been lost: if `ALC[4:0]==5'h00` then arbitration has been lost in bit 1 of the identifier, if `ALC[4:0]==5'h01` then arbitration has been lost in bit 2 of the identifier, etc.
The `SJA1000` peripheral contemplates loss of arbitration only in the `ID` and `RTR` fields, using 5 bits only, `ALC[4:0]`, the remaining 3 bits are reserved.
In order to perform such test, the idea consisted in keeping the message sent by the passive node as fixed while varying the active node's one to test all of the conditions explained above.

Reproducing such deterministic behavior in a functional environment, however, is not straightforward: the main problem consisted in synchronizing the two modules in order to have them configured such that the messages they send start at the same time. An attempt has been made to achieve this condition, future developments could make use of a slower bit-rate or synchronization by means of internal timers.

```
      /************************************************
       ********* Sixth Step: Arbitration Lost **********
       ************************************************/


      can_handl.CAN_Mode = 0x01;
      can_handl.CLKOff = 0x01;
      can_handl.LOM = 0x00;
      can_handl.STM = 0x00;
      can_handl.AFM = 0x00;
      can_handl.CD = 0x00;
      can_handl.ACR[0] = 0x00;
      can_handl.ACM[0] = 0xFF;
      can_handl.B_TIM0 = bustim0[0];
      can_handl.B_TIM1 = bustim1[0];
```

```
        canPeriphInit(&can_handl);

        tx_handl.DLC = 0x08;
        tx_handl.FF = 0x01;
        tx_handl.RTR = 0x00;

uint32_t tx_ID = 0x3FFFFFFF;
volatile uint32_t alc_reg;
int arb_lost;

tx_handl.ID = tx_ID;
// Ideally, we'd like to send a message concurrently with the
   other node in order to have a loss of arbitration
// By looping and changing the point in which there's an
   arbitration loss we should be able to go through every
   possible state
for(arb_lost = 0; arb_lost < 30; arb_lost ++) {

  tx_ID = tx_ID >> 1;

        for(i = 0; i < 8; i++) {
                tx_handl.TXBuf[i] = lfsr[i];
                lfsr[i] = (lfsr[i] << 1) | (((lfsr[i] & (1 << 6))
                    >> 6) ^ ((lfsr[i] & (1 << 4)) >> 4) ^ ((lfsr[
                   i] & (1 << 2)) >> 2) ^ ((lfsr[i] & (1 << 1))
                   >> 1));
        }

  // Transmit a message concurrently with the other
  if(transmitMsg(&tx_handl,timeout) != CAN_OK) __asm__("l.addi
     r21,r0,0x21");

  // Read the arbitration lost capture register
              alc_reg = *(CAN_EXTND_ALC_REG);

}

// Abort a transmission
      // Send msg
      *(CAN_GNRAL_CMD_REG) = 0x01;

      // Wait for SOT (Start of Transmission)
      while(!((*(CAN_GNRAL_STATUS_REG) & 0x20) >> 5));

// Now abort it
*(CAN_GNRAL_CMD_REG) = 0x02;
```

Listing 4.7. Arbitration Test subprogram

4.2.8 Acceptance Filter test

Every message received by the CAN controller is filtered by comparing its ID against some programmable bit-masks. In order to test the comparators in the acceptance filters, deterministic patterns can be used [1]. Alternatively, patterns can be easily derived by launching an Automatic Test Patterns Generation tool on the combinational logic. Each pattern is then transformed into a message sent by the passive node and filtered by the active node accordingly.

```
        /************************************************
         ****** Seventh Step: Acceptance Filter Test *****
         ************************************************/


// All of these patterns have been generated by means of a ATPG
   script
// Pattern 1
can_handl.CAN_Mode = 0x1;
can_handl.AFM = 0x1;
can_handl.ACR[0] = 0xd;
can_handl.ACR[1] = 0x9b;
can_handl.ACR[2] = 0xbc;
can_handl.ACR[3] = 0x1;
can_handl.ACM[0] = 0xff;
can_handl.ACM[1] = 0xff;
can_handl.ACM[2] = 0xff;
can_handl.ACM[3] = 0xf4;
can_handl.B_TIM0 = bustim0[0];
can_handl.B_TIM1 = bustim1[0];
canPeriphInit(&can_handl);
BRP = bustim0[0] & 0x3F;
TSEG1 = bustim1[0] & 0x0F;
TSEG2 = (bustim1[0] & 0x70) >> 4;
timeout = 2*(BRP+1)*(TSEG1+TSEG2+3)*4;
tx_handl.ID = (uint32_t)(0x948a23e);
tx_handl.DLC = 0x02;
tx_handl.TXBuf[0] = 0x7;
tx_handl.TXBuf[1] = 0xb2;
tx_handl.RTR = 0x00;
tx_handl.FF = 0x1;
#ifdef NODE1
transmitMsg(&tx_handl,timeout);
receiveMsg(&rx_handl,timeout);
#else
receiveMsg(&rx_handl,timeout);
transmitMsg(&tx_handl,timeout);
#endif

can_handl.CAN_Mode = 0x1;
```

```
can_handl.AFM = 0x0;
can_handl.ACR[0] = 0x98;
can_handl.ACR[1] = 0x6a;
can_handl.ACR[2] = 0xae;
can_handl.ACR[3] = 0xa6;
can_handl.ACM[0] = 0xc0;
can_handl.ACM[1] = 0xfb;
can_handl.ACM[2] = 0xc4;
can_handl.ACM[3] = 0xa2;
can_handl.B_TIM0 = bustim0[0];
can_handl.B_TIM1 = bustim1[0];
canPeriphInit(&can_handl);
BRP = bustim0[0] & 0x3F;
TSEG1 = bustim1[0] & 0x0F;
TSEG2 = (bustim1[0] & 0x70) >> 4;
timeout = 2*(BRP+1)*(TSEG1+TSEG2+3)*4;
tx_handl.ID = (uint32_t)(0xaa26f89);
tx_handl.DLC = 0x02;
tx_handl.TXBuf[0] = 0x5d;
tx_handl.TXBuf[1] = 0x1b;
tx_handl.RTR = 0x00;
tx_handl.FF = 0x0;
#ifdef NODE1
transmitMsg(&tx_handl,timeout);
receiveMsg(&rx_handl,timeout);
#else
receiveMsg(&rx_handl,timeout);
transmitMsg(&tx_handl,timeout);
#endif
```

Listing 4.8. Acceptance Filter Test subprogram

Chapter 5

Experimental Results

Details about the test program described in 4.2 are reported in 5.1. The table reports its test application time in clock cycles (column 2), the number of messages exchanged by the nodes (column 3), and the amount of bytes sent by the active node (column 4) and passive node (column 5). The clock signal period adopted in the testbench is 10ns, as a consequence the total amount of time required by the whole test program to run is about 331ms.

Test program	*Duration [clk cycles]*	*#Messages*	*Bits sent*	*Bits received*
Bitrate	1,577,790	8	864	864
Normal Mode	248,652	8	1,144	1,144
Self-Test Mode	16,964	1	108	108
Listen Only Mode	17,564	1	108	108
FIFO	3,157,031	328	0	28,000
Errors	410,452	1	72	72
Arbitration	815,000	30	2,160	2,160
Acceptance Filter	28,856,547	592	44,176	44,176

Table 5.1. Test programs details

Fault simulation experiments were run using a commercial fault simulator as explained in subsection 2.4.1. Moreover, an approach similar to the one described in [2] was applied in order to identify faults in the combinational logic that are untestable in functional mode, due to fixed values on specific flip-flops or primary inputs.

Details about the fault simulation experiments are reported in 5.2. The table reports, for the main sub-modules in the CAN controller, the number of stuck-at faults (column 2), the fault coverage (column 3), which is the ratio between the tested faults above all faults in the circuit, and the test coverage (column 4), where

untestable faults are not taken into account. Since many smaller modules and glue logic are not reported in the table, the number of faults of a given module is higher than the sum of the faults in the reported sub-modules.

On the top-level module of the CAN controller, the final result showed how the 90.24% of the testable faults was covered. The testability analysis allowed to classify as functionally untestable about 1% of faults, although this percentage significantly varies among the modules.

Instance Name	#Stuck-at Faults	Fault Coverage %	Test Coverage%
i_can_registers	5,352	84.91	84.91
acceptance_code_regs	256	100.00	100.00
acceptance_mask_regs	256	100.00	100.00
bus_timing_regs	128	85.94	85.94
clock_divider_regs	118	90.68	90.68
command_reg	134	72.97	79.10
error_warning_reg	152	92.76	92.76
irq_en_reg	64	96.88	96.88
mode_regs	174	91.96	91.96
tx_data_regs	832	100.00	100.00
i_can_btl	1,472	83.24	83.31
i_can_bsp	31,236	91.43	91.44
i_can_crc_rx	380	99.47	99.47
i_can_acf	1,418	85.40	85.40
i_can_fifo	17,382	96.26	96.26
TOTAL	38,490	90.22	90.24

Table 5.2. Faults Report of the CAN Controller

One of the advantages of the developed program is that it can be decomposed in subprograms that can be eventually re-grouped together when scheduling the test phases. As a consequence, even all of the various submodules were fault simulated one by one, the results can be seen in table 5.3.

To gain some insight on the validity of the proposed method and in order to see how it compared against more realistic scenarios, other test programs were developed and tested as well. Each developed program assumes the same testbench employed in the test program.
The additional programs are described as follows:

Subprogram Name	ACF Test	Arbitration Test	Errors Test	FIFO Test
Registers Coverage%	48.18	40.00	30.93	25.37
BTL Coverage%	51.71	52.12	69.45	51.64
BSP Coverage%	69.45	17.44	14.93	60.46
ACF Coverage%	81.95	3.95	0.42	6.77
FIFO Coverage%	80.26	0.77	0.35	81.47
Test Coverage%	66.05	22.56	19.96	55.58
Subprogram Name	LOM Test	Bit-rate Test	Normal Mode Test	ST Mode Test
Registers Coverage%	26.47	39.16	70.98	46.29
BTL Coverage%	67.60	68.84	51.03	67.95
BSP Coverage%	60.84	42.59	38.40	28.93
ACF Coverage%	13.82	3.74	17.49	10.16
FIFO Coverage%	77.30	39.60	24.09	10.35
Test Coverage%	56.63	43.66	44.03	33.48

Table 5.3. Faults Report of the STL subprograms

- `Ten Messages Program`: the active node and passive node transmit, alternatively, 10 messages keeping all of the parameters fixed, except for the data bytes (the actual message).
- `Ten Varying Messages Program`: same as the `Ten Messages Program` but after every message all of the main parameters, like bitrate, ID, DLC and acceptance filter, are changed.
- `One Hundred Messages Program`: the active node and passive node transmit, alternatively, 100 messages keeping all of the parameters fixed, except for the data bytes (the actual message).
- `One Hundred Varying Messages Program`: same as the `One Hundred Messages Program` but after every message all of the main parameters are changed.
- `SelfTest Program`: the active node transmits 100 messages in Selftest mode, changing the main parameters after every message.
- `Listen Only Mode Program`: the active node receives 100 messages in Listen Only Mode, messages are sent by the passive node. At every iteration the bitrate changes accordingly to the passive node's one, other parameters are kept constant.

Program Name	Ten Messages	Ten Varying Messages	Hundred Messages
Registers Coverage%	37.96	44.75	37.32
BTL Coverage%	51.85	70.96	51.58
BSP Coverage%	47.15	42.96	61.96
ACF Coverage%	3.10	4.58	3.10
FIFO Coverage%	48.56	37.90	75.11
Test Coverage%	46.52	44.82	58.45
Program Name	Hundred Varying Messages	SelfTest Program	Listen Only Program
Registers Coverage%	45.65	41.89	26.01
BTL Coverage%	74.38	67.88	67.26
BSP Coverage%	65.27	16.72	60.84
ACF Coverage%	4.58	3.31	13.82
FIFO Coverage%	77.53	0.85	77.30
Test Coverage%	63.18	22.76	56.54

Table 5.4. Faults Report of the comparison test programs

The results achieved by every single program are shown in table 5.4.

While testing and refining the test programs, after a few trials, it was clear that, with the exception of some faults (e.g., inside the ACF or the FIFO) that can be covered by ad-hoc messages, a significant amount of faults were hard to test due to the hardware configuration. Examples of such faults are those related to error and arbitration lost conditions, which impact on the coverage of the glue logic in the BSP.

Errors such as those due to electromagnetic interference or physical problems on the bus cannot be emulated by means of two nodes connected with an ideal wire and thus the test of the related logic would require an external module capable of understanding that a message is being sent and pulling down the line when it should be on a logic `1'b1`. If this approach was adopted in the system (with the related complexity and additional hardware and design cost), the error conditions could be tested more thoroughly.

Concerning arbitration lost, since the CAN Protocol does not provide an arbiter and the CAN Bus is 0 dominant, arbitration is achieved by means of checking the Rx wire while sending the message: for each bit, if the bit sent is different from the one on the bus it means that a higher priority message is being sent at the same time and so the node with the lowest priority aborts the transmission. The arbitration lost could occur in the ID or RTR part of the message. The main problem

here is trying to synchronize the transmission of the two peripherals and dedicated hardware should be implemented to precisely break messages sent by the active node.

Finally, there are limitations due to the sampling precision, which impact on the coverage of the BTL. Even though it is possible to act on the programmable time segments in the BTL, some values cannot be actually used in functional mode.

Due to these reasons faults that could not be tested have been removed by the fault list. This has been done by means of taking the complete fault list and checking all of the not tested faults that reported the keywords `arbitration` or `err`. The assumption though is that this approach, although effective since it allowed the removal of most faults, did not remove the total amount of non-testable faults since some lines lost the original name during the synthesis process, losing the ability to recognize whether that was a testable fault not reached or an untestable one.

Chapter 6

Conclusions

This work specifically targets the structural test of CAN controllers by means of test programs installed on two devices attached to the same CAN bus. The proposed systematic approach gives guidelines about the messages to transmit on the CAN bus to cover the functional blocks in the controller. Experiments on a system composed by two open-source SoCs connected to the same CAN bus shown that 90.24% of the testable stuck-at faults in the CAN controller can be covered by the proposed approach. Among the remain untested faults it's possible that a fraction of them is still functionally untestable, further work can be done in order to identify them in a provable manner.

Given its flexibility and high reusability, the proposed software-based approach is very well suited to be used for in-field test of CAN controller modules embedded in safety-critical systems. The work also aimed at identifying hard to test faults and future activities are planned to cover them, by exploiting more complex configurations (e.g., using more than two nodes). Moreover, timing faults will be targeted by future works, as well as faults affecting the CAN bus due to electrical problems. Finally, the handling and the scheduling of the end-to-end test programs in production will be studied. One further improvement could be done in terms of application time, since this was not the main concern of the work.

Bibliography

[1] H. Grigoryan, G. Harutyunyan, S. Shoukourian, V. Vardanian, and Y. Zorian, "Generic bist architecture for testing of content addressable memories," in *2011 IEEE 17th International On-Line Testing Symposium*, pp. 86–91, July 2011.

[2] R. Cantoro, S. Carbonara, A. Floridia, E. Sanchez, M. Sonza Reorda, and J. Mess, "An analysis of test solutions for COTS-based systems in space applications," in *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pp. 59–64, Oct 2018.