HELLENIC MEDITERRANEAN UNIVERSITY

DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

Intelligent Systems & Computer Architecture Lab

(I.S.C.A. Lab)

Thesis Report

Control of CAN-BUS TIME-TRIGGER Messages for Adaptive Networking

Author

Vyron Varsamis

Supervisor

Dr. Georgios Kornaros

Document version

v0.2

July 5, 2021

Abstract

Controller Area Network(CAN-bus) created and mostly is used in automobile system such as cars, however his use expand in trams,light railways,operating room at hospitals many others including outer space applications. The reason of this is the low cost, the credibility of not losing messages and flexibility from several different physical layers. Every one of them has their complexity and different usage. The diversity varies from application to application , however the time integrity is always important. Taking advantage of this application, will adapt the frequency to the maximum beneficial output for time trigger message systems, instead of using a fixed data rate that will waste the optimal results. Controller Area Network(CAN-bus) is a system that include nodes who communicate with each other, whenever the bus is free the compete with each other to send their messages to accomplice their tasks. In some applications with specific requirements the bus can be idle with nodes not sending messages wasting time. We produce a application that adapt measuring the optimal frequency of sending messages while maintains the message integrity by calculating with worst case scenario of our system. The research was based in trial and error, repeatedly testing same frequencies to observe their differences. Would the results of the same frequency will change in a practical test or would be the exact same. Our observation conclude with some interesting results. The variety of the results could differ in different systems that could be applied, but adjusting our application in wider systems maximizing their potential by minimizing their reaction time and preserve the message integrity could upgrade existing or future systems.

Abstract

Controller Area Network(CAN-bus) δημιουργήθηκε και κυρίως χρησιμοποιήθηκε στα αυτόκίνητα συστήματα όπως των αμαξιών, παρόλα αυτά η χρήση του επεκτάθηκε στα τραμς, τραίνα ελαφρού τύπου, νοσοκομειακά χειρουργία και σε πολλά άλλα καθώς και σε εφαργμογές που αφορούν το διάστημα. Ο λόγος ο οποίο συμβαίνει αυτό είναι το χαμηλό κόστος του, η αξιοπιστία του και η ευελιξία του σε πολλά διαφορετικά επίπεδα. Κάθε επίπεδο έχει την δικιό του επίπεδο δυσκολίας και την δική του χρήση. Η ποικιλία αυτή μεταβάλλεται από εφαρμογή σε εφαρμογή, παρ΄όλα αυτά η ακεραιότητα τοου χρόνου διατηρείται σημαντική. Εκμεταλλευόμενοι την εφαρμογή, η συχότητα θα προσαρμόζεται στην μεγαλύτερη ωφέλιμη έξοδο για συστήματα που εξιδικεύονται σε σεγκεκριμένη χρονική αποστολή μηνύμάτων, αντί να χρησιμοποιηθεί συσκεκριμένη ροή δεδομένων η οποία θα σπαταλήσει θα βέλτιστα αποτελέσματα. Controller Area Network(CAN-bus) είναι σύστημα επικοινωνίας μεταξύ κόμβων, οποιαδήποτε στιγμή ο δίαυλος είναι διαθέσιμος οι κόμβοι ανταγωνίζονται μεταξύ τους έτσι ώστα να στείλει ο καθένας το μήνυμα του έτσι ώστε να ολοκληρώσουν τον σκοπό τους. Κάποιες εφαρμογές με συγκεκριμένες προϋποθέσεις ο δίαυλος μπορεί να μένει αδρανής, χωρίς τους κόβους να στέλνουν μηνύματα ο χρόνος σπαταλάτε. Εμείς κατασκευάσαμε μία εφαρμογή η οποία προσαρμόζεται στην βέλτιστη συχνότητα αποστολής μηνυμάτων διατηρώντας την ακαιραιώτητα τους, μέσω της χείριστης υπάρχουσας περίπτωσης που μπορεί να υπάρξει στο σύστημά μας. Η έρευνα αυτή βασίστηκε σε δοκιμές και λάθη, δοκιμάζοντας επαναλαμβανόμενα τις ίδιες συχνότητες για να παρακολουθήσουμε τις διαφορές μεταξύ τους. Ήταν τα αποτελέσματα της ίδιας συχνότητας τηα αλλάξουν σε μία πρακτική δοκιμή ή θα είναι ακριβώς τα ίδια. Οι παρατήρηση μας αυτή κατέληξε σε ενδιαφέρον αποτελέσματα. Η ποικιλία των αποτελεσμάτων μπορεί να διαφοροποιείται σε διαφορετικά συστήματα στα οποία μπορεί να εφαρμοστεί, η προσαρμοστηκότητα της εφαρμογής σε ευρήτερα συστήματα μπορεί να μεγιστοποιήσει τις δυνατότητες τους, ελαχιστοποιώντας τον χρόνο αντίδρασης καθώς διατηρώντας την ακεραιότητα των μηνυμάτων θα μπορούσε να αναβαθμίσει μελοντικά συστήματα.

Contents

List of Abbreviations

ACK	**Ack**nowledgement
bps	**B**its **P**er **S**econd
CAN	**C**ontroller **A**rea **N**etwork
CRC	**C**yclic **R**edundancy **C**heck
DLC	**D**ata **L**enght **C**ode
ECU	**E**lectronic **C**ontrol **U**nits
EOF	**E**nd **O**f **F**rame
FPGA	**F**ield-**P**rogrammable **G**ate **A**rray
Id	**I**dentifier
IDE	**ID**entifier **E**xtension bit
IOF	**I**ntermission **O**f **F**rame
IRQ	**I**nterrupt **R**erquest **L**ine
MO	**M**idpoint **O**perate case
Rx	**R**eceiver Node
RTOS	**R**eal **T**time **O**peration **S**ystem
RTR	**R**emote **T**ransmission **R**equest
SRR	**S**ubstitute **R**emote **R**equest
Tx	**T**ransmitter Node

Chapter 1

1.1 Introduction

Controller Area Network(CAN bus) protocol was created by Robert Bosch GmbH and released in 1986 at the society of automotive engineering. One of the results of this protocol was the reduction of the wiring harnesses but this wasn't the intention it was a by-product. The mainly purpose of the protocol presented the day of the release was a non-destructive arbitration mechanism. Without a central bus master, grants access to the frame with the highest priority. Furthermore, they implemented many error detection mechanisms for the error handling.

It's a vehicle standard design that allows micro-controllers and devices to communicate with each other without a host computer. It's a message broadcast system with maximum signal of 1 Megabit per second (bps), that can broadcast many short length messages to every node. It has the privilege to not be a point to point connection under a bus master supervision, but it cant send large length of data. A typical vehicle is equipment with at least 70 Electronic Control Units(ECUs) divided in groups with different role per group. The ECUs and sensors communicate via CAN bus with each other to fulfil the group's role. This importance of the roles can be from air-bags, power steering, battery recharging systems to mirror adjustment, audio system etc.

Every message is programmed to has unique Id before transmission. Their innovation include that sending priority is based on message's content and not the transmitter's or receiver's node Identifier. Messages can sporadic or periodic depending on the system's requirements. Using specific requirements the frequency of transmitting message could be longer, without knowing the true potential of the application.As a result, bus staying idle wasting time. Although,when CAN bus reliability is one of the strong points, using a wrong hypothetical minimal frequency will have unwanted results as losing frames.

So the adaptation of the application/firmware that communicates over CAN-bus[7][8][9][10] to address the challenges:

(i) Variances of bus traffic and thus better - automatic - fault tolerance and more robust, reliable communication,

(ii) Malicious traffic which intends to affect proper pre-designed traffic patterns,

(iii) Exploration, bus/networking investigation,

(iv) Variance of application in terms of real-time requirements.

Achieving such challenges we can make an adaptive frequency application that will explore the system by running it and adapt to the closest optimal frequency. Depending of the system that is given will run with the worst case scenario and change suitably to match users needs without sacrificing the reasons why we choose to use CAN-bus. This application will provide robust communication between nodes and etc. without any need of adjustment of the user. It can choose the path and the correction that the sees more appropriate by the setting that is installed. The adaptability of firmware[9]10] can be used as a breakthrough and foundation of many projects that can follow in the future of CAN bus taking advantage every source of a system. With automotive progressing and securing vehicular communication[12] plus utilizing one time programmable ECUs[13] give prominence to designate the importance of an adaptive system. Importance that a system that can auto-correct it self will be useful even if the needs change.

It became very popular to many industries as robust protocol, applied to a variety applications. The easy and simply implementation, the reliability of message integrity help held the popularity at top.

This paper is focused in auto-improving application. A system that the message-integrity is a top priority, minimizing the possibility of corruption,a continues message flow while focusing on the highest frequency data rate. This is organized as following Chapter 2 will introduce CAN-Bus 2.0B basic knowledge about Data transmission . Chapter 3 will represent the software and hardware implementation of the application. Chapter 4 include the theoretical and practical approach of our application.Chapter 5 has the results of testing the application and the theory behind them.

Chapter 2

2.1 Data transmission

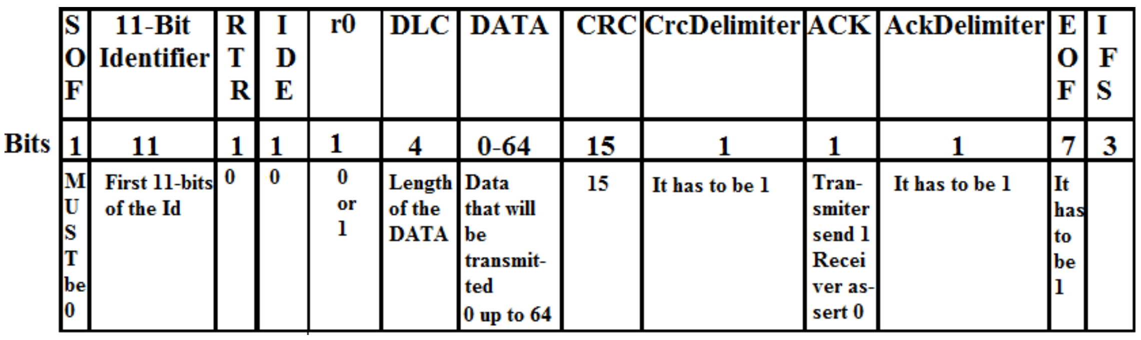
CAN data transmission specifications is designed to whenever the bus is free every node can compete to take the priority to transmit his data. They use the tern of dominant and recessive bit, where dominant is logical 0 and recessive logical 1. When they start transmitting, the priority is taken by the node with lowest message Identifier ,prioritizing the important messages. If a node lose the arbitration (priority) it enters the receive state. This happens to prevent any data to be destroyed or lost. Message identifier is unique for every node so there won't be a case with two nodes take priority at by winning the arbitration at same time.

2.1.1 Frames

Frames in CAN bus is the messages that the node broadcast in the bus. There are different types of frames and each one of them has his role. Every message can have different length however there is a maximum size.An application that support extended CAN frame(2.0B), include standard CAN frames(2.0A) too. There are two types of frames (messages) formats. The difference between them is that standard frame CAN 2.0A with 11 bit identifier and the extended frame CAN 2.0B with 29 bit identifier, however applications that support CAN 2.0B can also send standard frame CAN 2.0A.

Basic frame Format

The extended frame CAN 2.0B format has 29 bits identifier plus a Substitute Remote Request(SRR) field instead of 11 bits identifier of standard CAN frame. As we already said an application that supports extended CAN frame can send standard CAN frame too, but their frames fields differs. In the next figures 2.1 and 2.2 we define the differences and explain the role of Substitute Remote Request(SRR) and it position. In Figure 2.1 we can see the standard Data frame format and in table 2.1 will show us the size, roles, and possible values.



	SOF	11-Bit Identifier	RTR	IDE	r0	DLC	DATA	CRC	CrcDelimiter	ACK	AckDelimiter	EOF	IFS
Bits	1	11	1	1	1	4	0-64	15	1	1	1	7	3
	MUST be 0	First 11-bits of the Id	0	0	0 or 1	Length of the DATA	Data that will be transmit-ted 0 up to 64	15	It has to be 1	Tran-smiter send 1 Recei ver as-sert 0	It has to be 1	It has to be 1	

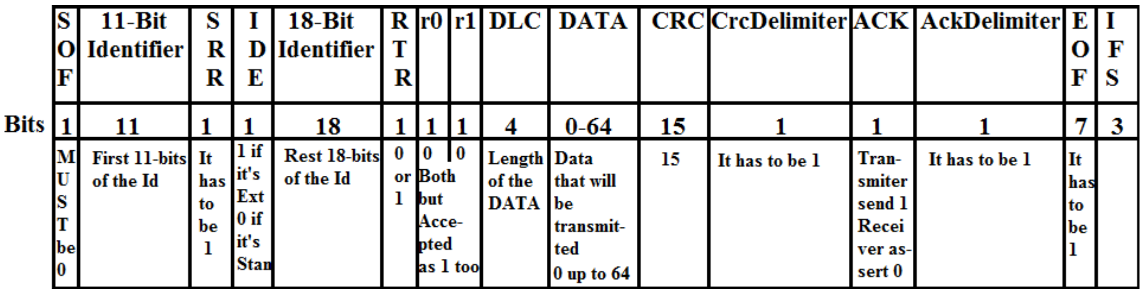
Figure 2.1: Basic frame format

As we see there is only a 11 bit Identifier and no Substitute Remote Request(SRR).Furthermore Table 2.1 will show us info about a Standard CAN data frame.

Table 2.1: Standard CAN data frame properties and info

Acronyms	Size	Role	Value
Start **O**f **F**rame	1	Commence the start of frames	0
11-Bit Identifier	11	The first 11 bits of The message ID	1rst bit 1,rest 1 or 0
Remote **T**transmission **R**equest	1	Data frame or Remote Frame	0 or 1
Identifier **E**xtension Bit	1	Standard CAN data frame	0
R0	1	Reserved Bit	0 or 1
Data **L**ength **C**ode	4	Data's Size	0000-1000 bytes
DATA	0-64	Data that will be transmitted	0x00-0xFF per Data
Cyclic **R**edundancy **C**heck	15	Receivers check for error in messages	Inserted by CAN bus
CRC Delimiter	1	Must be 1	0 or 0 or 1
Acknowledgment	1	Transceiver Receiver asserts	1 0
ACK Delimiter	1	Must be 1	0 or 1
End **O**f **F**rame	7	Commence the end of frame	1111111
Intermission **O**f **F**rame	3	Help with Synchronization	111

We have to note that Remote Transmission Request(RTR) has value 0 to be a data frame, either it's a Standard CAN data frame or Extended CAN data frame. Notice too in Figure 2.2 that Remote Transmission Request(RTR) field moved further in to the CAN frame and his place was taken by Substitute Remote Request(SRR) field.



	SOF	11-Bit Identifier	SRR	IDE	18-Bit Identifier	RTR	r0	r1	DLC	DATA	CRC	CrcDelimiter	ACK	AckDelimiter	EOF	IFS
Bits	1	11	1	1	18	1	1	1	4	0-64	15	1	1	1	7	3
	MUST be 0	First 11-bits of the Id	It has to be 1	1 if it's Ext 0 if it's Stan	Rest 18-bits of the Id	0 or 1	0 Both but Acce- pted as 1 too	0	Length of the DATA	Data that will be transmit- ted 0 up to 64	15	It has to be 1	Tran- smiter send 1 Recei ver as- sert 0	It has to be 1	It has to be 1	

Figure 2.2: Extended frame format

This happens because Standard CAN data frames have bigger priority in the bus from the Extended CAN data frames. As we can see in Table 2.2 Substitute Remote Request(SRR) of Extended CAN data frame has value 1 and Remote Transmission Request (RTR) in standard has 0. Table 2.2 shows the size,roles, and possible values of the an Extended CAN data frame.

Table 2.2: Extended CAN data frame properties and info

Acronyms	Size	Role	Value
Start **O**f **F**rame	1	Commence the start of frames	0
11-Bit Identifier	11	The first 11 bits of The message ID	1rst bit 1,rest 1 or 0
Substitute **R**emote **R**equest	1	Replace RTR bit	1
Identifier **E**xtension Bit	1	Standard CAN frame or Extended Can frame	0 1
18-bit Identifier	18	The rest 18 bits of the message ID	0 or 1
Remote **T**transmission **R**equest	1	Data frame or Remote Frame	0 or 1
R0	1	Reserved Bit	0 or 1
R1	1	Reserved Bit	0 or 1
Data **L**ength **C**ode	4	Data's size	0000-1000
DATA	0-64	Data that will be transmitted	0x00-0xFF per Data
Cyclic **R**edundancy **C**heck	15	Receivers check for error in messages	Inserted by CAN bus
CRC Delimiter	1	Must be 1	0 or 1
Acknowledgment	1	Transceiver Receiver asserts	1 0
ACK Delimiter	1	Must be 1	0 or 1
End **O**f **F**rame	7	Commence the end of frame	1111111
Intermission **O**f **F**rame	3	Help with Synchronization	111

There are 4 different types of frames.
- Data frame
- Remote frame
- Error frame
- Overload frame

Data Frame

Data packets are the most commonly used. Their purpose is to contain the Data that will be transmitted in the payload.

Remote Frame

Request data from a node. Remote frame doesn't contains data , but DLC has the value of data that are requested. To submit an Remote frame RTR has to be recessive.

Error Frame

Two fields in Error Frame:

- Error Flags contributed from different stations (6–12 dominant/recessive bits)

  ○ Active Error flag has 6 dominant bits and is transmitted by a node when there is an error on the network and is in "error active" state

  ○ Passive Error Flag has 6 recessive bits and is transmitted when there is an error active frame with the state "error passive".

- Error Delimiter (8 recessive bits).

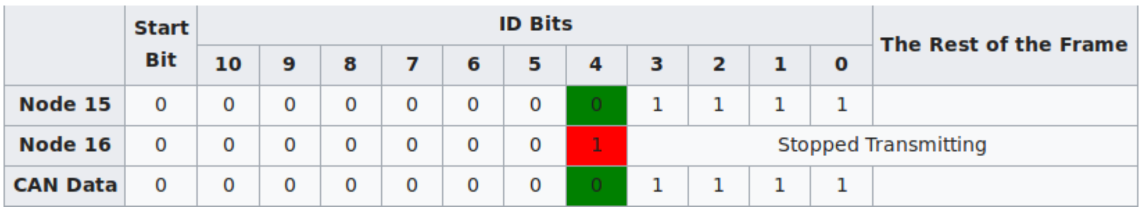
Overload Frame

Overload frame is transmitted from a node when it becomes to busy. Its similar to Error frame regarding the format but isn't increase the error counter or does not cause retransmission.

2.1.2 Arbitration

When two node start transmitting simultaneously there will be an arbitration that decides who will take the priority to broadcast the message. Priority will be taken by the node with lowest message Identifier. To check the lowest identifier there will be a bit to bit compare. Nodes will start transmitting their frame, when a node transmit a dominant bit and other node transmit recessive bit, instantly the node with the recessive has lost the arbitration and goes into recessive state. This happens until only one node has won all the comparisons. It's important that every message id will be unique so the "winner" of the arbitration will be only one and the rest will receive the broadcast message.

For example we have two node that transmit simultaneously. Both of them will start transmitting their frame. As we see in figure 2.3 the data at first is identical so both will keep transmit the bits.



	Start Bit	ID Bits											The Rest of the Frame
		10	9	8	7	6	5	4	3	2	1	0	
Node 15	0	0	0	0	0	0	0	0	1	1	1	1	
Node 16	0	0	0	0	0	0	0	1	Stopped Transmitting				
CAN Data	0	0	0	0	0	0	0	0	1	1	1	1	

Figure 2.3: Arbitration

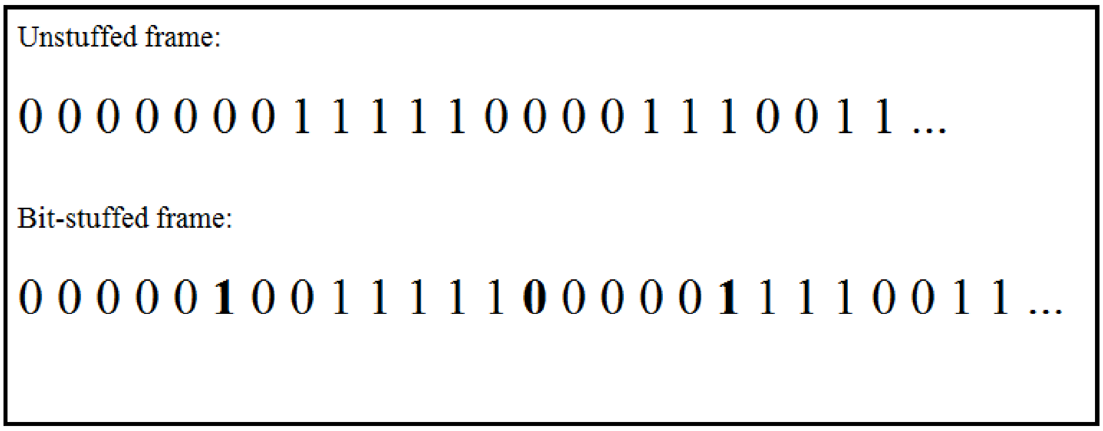
At ID 4, node 16 has transmitted a recessive bit and node 15 a dominant. At that time node 16 will stop transmit,go in receive state, and the data in the bus will be same as node 15.

In a case that several nodes start transmit frames with the same identifier, bus if following some ground rules. Either if its Standard CAN frame or Extended CAN frame, Data frames have higher priority than remote frame and as we already mention Standard over Extended. In a case that a remote frame of a Standard frame is transmitting and an Extended data frame as well, that happens to have the first 11 bit equal, the Standard remote frame will win the arbitration.

2.1.3 Bit-stuffing

Bit-stuffing is used to maintain the synchronization. Every 5 consecutive bits of the same polarity a bit of the opposite polarity will be placed .The stuffed bit after the placement will be participate to the count for next 5 consecutive bits.Six consecutive bits of the same polarity are considered error. All fields in CAN frame can be bit-stuffed except CRC delimiter , ACK field , ACK delimiter and the end of frame. After transmission the frame will be de-stuffed by the receiver. The only way we can interfere with bit stuffing is by placing specific values to the frame,however the bit stuffing is handled by the CAN bus it self and it's not visible by the user.

Example :



Unstuffed frame:

0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 1 1 1 0 0 1 1 ...

Bit-stuffed frame:

0 0 0 0 0 **1** 0 0 1 1 1 1 1 **0** 0 0 0 0 **1** 1 1 1 0 0 1 1 ...

Figure 2.4: Bit Stuffing

Chapter 3

3.1 Adaptive application Software implementation

The application is focused on software development. Processing the existed software of CAN bus to adjust it for our goals. Some of the changes was to enable interrupts, include and implement timers and other. More details are explicate bellow.

3.1.1 Frame of the application

Hans-Christian Reuss in his research mention the minimum and maximum sizes of a frame:

Table 3.1: Frame size

Conditions	Size of Standard frame	Size of Extended frame
Regular bits	47+8·DLC	67+8·DLC
With Bit-stuffing	55+10·DLC	80+10·DLC

Standard frame format

- Without bit-stuffing: 111 us.
- With bit-stuffing :135 us.

Extended frame format

- Without bit-stuffing: 131 us.
- With bit-stuffing :160 us.

Based on Hans-Christian Reuss research the largest size of a frame has 160 bits size per message after being bit-stuffed.The closer case we achieved was a frame with size of 155 bits of the 160. The reason we artificial made this frame was to test our application to the limits by overloading our system with the worst case scenario. Unfortunately, as user we can't put specific values to fields such as CRC, CRC delimiter, ACK, ACK delimiter and EOF and the result of this was not to reach Hans-Christian Reuss mark of 160 bits.

Frame has values of:

- Frame ID:0x1F0C3C3C
- IDE:1
- DLC:8
- DATA[0]:0x3C
- DATA[1]:0x3C
- DATA[2]:0x3C
- DATA[3]:0x3C
- DATA[4]:0x3C
- DATA[5]:0x3C
- DATA[6]:0x3C
- DATA[7]:0x3C

Table 3.2 will display the detailed hexadecimal and binary values from the fields of the extended CAN frame and the stuffed bits that occurs based on the bit-stuffing theory.

Table 3.2: Extended CAN frame binary

Name	**Value(hex)**	**Binary**
Start of Frame	0	0
11-bit Identifier	0x1F0 and 2 bits of C	11111 **0**0000 **1**11
SRR	1	1
IDE	1	1
18-bit Identifier	2 bits of C and 0x3C3C	**0**0000 **1**1111 **0**0000 **1**1111 **0**00
Remote Transmission Request	1	1
R0	1	1
R1	1	1
Data Length Code	0x8	1000
DATA[0]	0x3C	00**1**1111**0**00
DATA[1]	0x3C	00**1**1111**0**00
DATA[2]	0x3C	00**1**1111**0**00
DATA[3]	0x3C	00**1**1111**0**00
DATA[4]	0x3C	00**1**1111**0**00
DATA[5]	0x3C	00**1**1111**0**00
DATA[6]	0x3C	00**1**1111**0**00
DATA[7]	0x3C	00**1**1111**0**00
CRC	Putted by CAN Controller	
CRC Delimiter	Putted by CAN Controller	
ACK	Putted by CAN Controller	

Table continued on next page

Name	Value(hex)	Binary
ACK Demimiter	Putted by CAN Controller	
End Of Frame	Putted by CAN Controller	
Intermission Of Frame	Putted by CAN Controller	111

3.1.2 Interrupts

Interrupt is a signal created by hardware or software when an event or a process needs to be executed. It alerts the processor with a high priority process that requires execution. There are two kinds of interrupts software and hardware interrupt.

Software Interrupts

Software Interrupts are interrupts which is requested by the processor to execute when certain conditions are met. This interrupts can be intentionally produced by executing a special instruction. An software interrupt can be :

- Periodic Interrupt: Occurs at fixed interval in timeline .
- Aperidic Interrupt: Can not be predicted.
- Synchronous Interrupt: Are dependent to the system clock and in phase to the system clock.
- Asynchronous Interrupt: Are independent to the system clock and are not phase to the system clock.

Hardware Interrupts

Hardware interrupt is the state of the hardware currently happens to be. All the devices are connected to Interrupt Request Line(IRQ) or detected by the embedded operating system, if there is no an operating system then the bare-metal program that is running. Hardware interrupts can arrive asynchronously however they have respect to the processor clock. Hardware interrupts arise in low level protocols or electrical conditions, the already running block code will stay in stand by and the interrupt handler manage the occurred event.

Interrupt Handler

Interrupt Handler is a block of code with the purpose to handle and execute specific interrupt conditions. In systems that many devices exist they have a particular interrupt request line (IRQ). This happens to determine the needed service of a device, or the event that occurred. Hardware interrupts is used to have high priority therefore it stops the running code to handle the interrupt. Later it was found convenient to be able to trigger this mechanism by software too, instead of hard-coded blocks.

Interrupts Checking methods

With multiple devices in a system there can be multiple IRQ signals simultaneously, then additional information is needed to decide which one will be managed first. Therefore the are some methods to decide the priority:

- Polling: The first device set the IRQ bit set is the device that will be handled first. Although it's simple to implement and is commonly used, it waste time checking IRQs of all devices.
- Vector Interrupts: They are identified by special code with the privilege to identify even the device that generate the interrupt.
- Interrupt Nesting: Priority is organised by the high-priority of the devices. This means that high priority device is recognised and low priority device is not.

3.2 Adaptive application Hardware implementation

This paper was focused on software development, however the hardware implementation is important to be addressed.

3.2.1 ZedBoard Zynq-7000 ARM/FPGA SoC Development Board

The communication via CAN bus was between two ZedBoard Zynq-7000. This board contains everything necessary to build a Windows, Linux,Android, Real Time Operating system(RTOS). Additionally, it provides all the necessary interfaces and supporting functions to enable a wide range of applications. This application was running on bare machine(bare metal). Having this opportunity one of the boards was used as Transmitter(Tx) and the other as Receiver(Rx) .

3.2.2 Timers

In this application two types of timers were enabled from the ZedBoard Zynq-7000:
•SCU timer.
•AXI timer.

SCU Timer

SCU timer is implemented to the board design and is running with frequency of 666.666.687 Hz. When is loaded with a value and start, will decrease the value until it will become 0.

The purpose of SCU timer was to represent the period of transmitting messages. Every time the timer will expire an interrupt will occur that enables the transmit of a message.This value is the a number cycles. However making the SCU timer have a specific period we must loaded with the correct cycles that correspond to the period.

To load the timer with cycles that correspond to the frequency we made a formula:

$$SCU_{Cycles} = (\frac{SCU_{Hz}}{2}) \cdot period$$

Where $\frac{SCU_{Hz}}{2}$ is frequency of the SCU timer divided by 2 can be counted per second, and $period$ is putted by user.

AXI timer

The AXI timer that was implemented in the design has frequency of 100.000.000 Hz. When the AXI start running from default will take a value and increase it until become equal to a value that is loaded. AXI has two timers that were enabled and used.

There was cases when the user could put a $period$ that interrupts occurred before the CAN bus send the frame, as result frame will be lost or didn't send with the desirable order.Therefore are two cases that AXI timer was used.The first AXI timer used to major updates to SCU timer and the second used for minor updates.To resolve cases like this if an interrupt occurred before another was cleared the first AXI timer counted the how much time was needed and added to the SCU timer.

But, as we can saw the AXI timer is has almost $\frac{1}{3}$ frequency of a SCU. So after AXI calculate we need to made sure that:

$$SCU_{Cycles} = AXI_{Cycles} \cdot 4$$

We multiplexed with 4 to be sure that the "new" SCU$_{Cycles}$ will be enough.
The second AXI timer was used to update if the delay inside the transmit started to getting longer.That was implemented to make the transmission more stable.

Chapter 4

4.1 SW Adaptive frequency

Time and message integrity is very important for CAN bus, we can have fixated time messages but in some applications we can decrease it to improve it based on our demands. It's an application that is able to find fixated minimum frequency of a system, maintaining the periodic consistency of the messages. Lowering the correspondence time without sacrificing the integrity, can be innovative for researches. Testing the potential of their system and real-time upgrading their application without additional workload to their part.

The application is supporting the extended CAN frame(CAN 2.0B).User can define different $periods$ due to his preferences. If $period$ is long enough to send the frame, transmission will complete without any changes. However there are cases where the adaptation is needed. The length of $period$ also depends from the baud rate of the system, and that is because baud rate is a measure of the number of bits that a system can transmit.In this application $Baud\ rate$ of 1,000 bit/s .

4.1.1 Theoretical Adaptive frequency application

This application is programmed to send frames between two zedboard. One will represent the transmitter(Tx) and the other the receiver(Rx). After the $period$ is defined, the SCU timer will be loaded with the cycles that represent it. $Period$ will be set at the Transmitter(Tx), and every time the SCU timer expire an interrupt will occur from the timer, enabling the block code to start the transmission. Depending the $period$, the program will not alter the frequency .If the frequency of the messages is low, the system wouldn't find delay between them it will not modify the frequency. As we can see at Figure 4.1 there is a stable $period$ between SCU timer's interrupts happen.

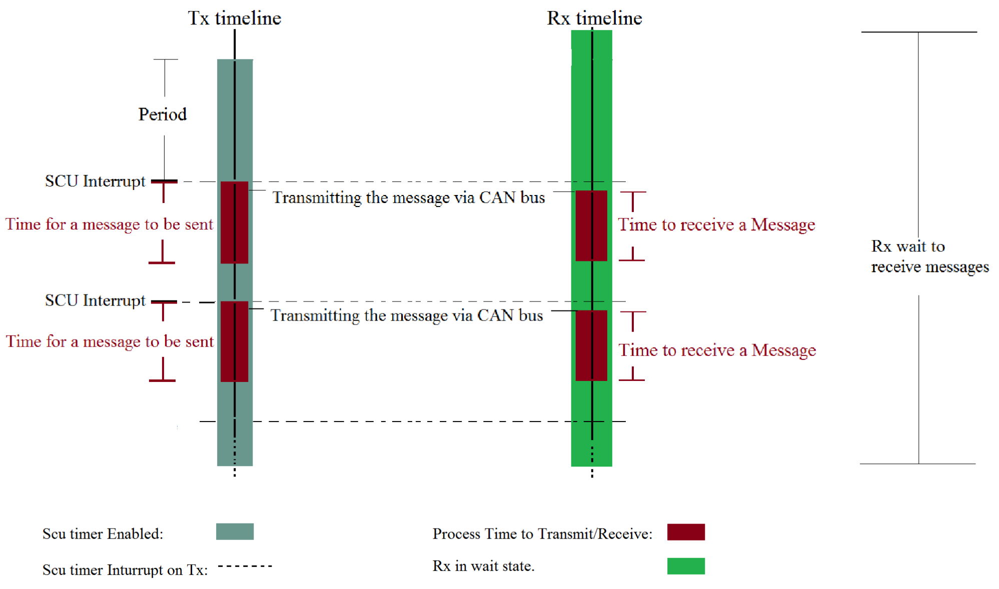

Tx timeline
Rx timeline
Period
SCU Interrupt
Time for a message to be sent
Transmitting the message via CAN bus
Time to receive a Message
SCU Interrupt
Time for a message to be sent
Transmitting the message via CAN bus
Time to receive a Message
Rx wait to receive messages
Scu timer Enabled:
Scu timer Inturrupt on Tx:
Process Time to Transmit/Receive:
Rx in wait state.

Figure 4.1: Transmission message process

Because of the long $period$ for the frame to be transmitted, received and wait until the next interrupt to occur there was no need for adjusting. However, if the $period$ was not enough or the delay inside of the transmitting process was accumulated the frequency would be changed.

In a case that the frequency of the messages needs to be changed we will have the Transmitter(Tx) to send frames to Receiver(Rx) without losing any frame and in the proper order. The fixated $period$ set by the user it may be less than the system can handle.Problems arise in this case. It was either "missing messages" or the disarranged messages from the order that was supposed to be sent. Adjusting the frequency of sending frames can resolve both of the problems. In case that the $period$ is low the transmit message process will be changed to an optimal frequency. At the follow figure 4.2 we are going to see the theoretical approach behind the process of the adapted $period$. The SCU timer will start count down and create an interrupt. When the interrupt occur the interrupt handler will be enabled and the transmission will begin. Before the process of the transmission ends another interrupt occurred defining that the frequency of transmitting messages is not enough. This event enabled the AXI timer start counting.

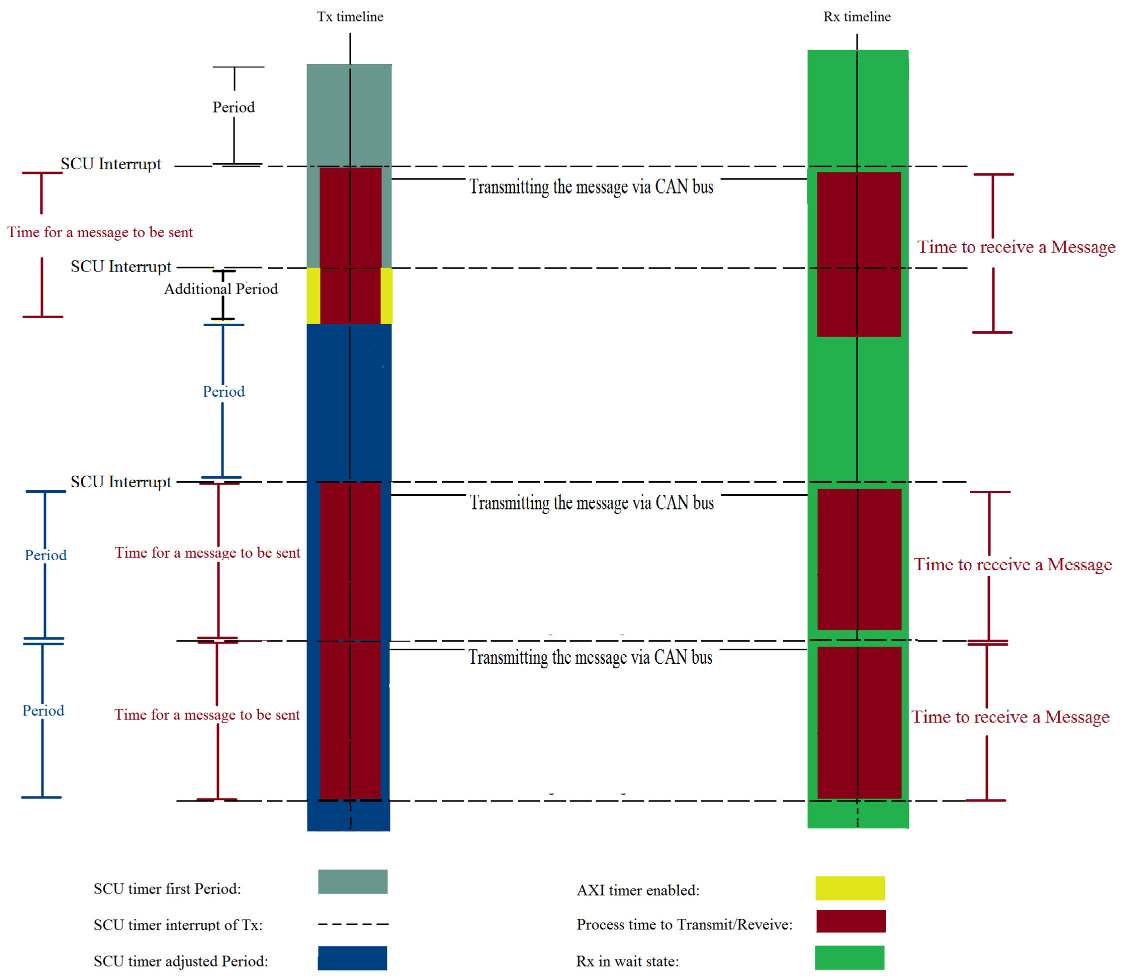

Tx timeline
Rx timeline
Period
SCU Interrupt
Transmitting the message via CAN bus
Time for a message to be sent
SCU Interrupt
Additional Period
Time to receive a Message
Period
SCU Interrupt
Transmitting the message via CAN bus
Period
Time for a message to be sent
Time to receive a Message
Transmitting the message via CAN bus
Period
Time for a message to be sent
Time to receive a Message
SCU timer first Period:
SCU timer interrupt of Tx:
SCU timer adjusted Period:
AXI timer enabled:
Process time to Transmit/Reveive:
Rx in wait state:

Figure 4.2: Adapt Transmission message process

After transmitting the frame, inside the handler a block of code is enabled for calculating the additional $period$ needed. The additional $period$ will be added to the existed or replace it and the SCU timer will be loaded with the new one; start the count down for the next frame. Adjusting the frequency of sending frames resolved out problems of "missing messages" and the received order of the frames by Rx. After testing it with the largest possible frame made by the user and succeed , we changed our approach to investigate if the frames wasn't received by Rx with the order that were made; to secure the message sending integrity.

4.1.2 Practical Adaptive frequency application

The purpose of this was because inside the CAN bus there were already a default delay if two messages with about to be send from a node. Some times because of the high frequency a third message occurred and happen to be send before the second had a chance. Even if the message was sent and received later, we aimed for the accurate order and uppermost integrity. Another way to secure the uppermost integrity was to interfere with the default delay. By interfere, we count the time that the packages were inside of the wait loop. To avoid the possibility frames to be collapse in the wait or lose their priority; we count the time with the for a frame in wait loop to be executed and set some parameters. If the parameters were fulfilled, minors adjustments ensue to the period.

To ensure the integrity of the messages and order we gave DATA[0] value equal to 0 and increase it every time we send a package, Rx will receive the frame and compare if the value of DATA[0] is the one that would be expected. If the the value of DATA[0] didn't match the Rx will stop receiving. The following Table 4.1 will show us the variables that is important and some changed unlike tests we did with the largest frame that we create and run before. Every block that will be shown is an important highlight of the code that can change the message transmit flow.

Table 4.1: Settings

Name	**Variable**	**Value**
period	TPERIOD	Set by the user(us)
SCU_{Hz}	FFACTOR	(CPU CORTEXA9 0 CPU CLK FREQ HZ)/ 2
SCU_{cycles}	TIMER LOAD VALUE	((u32)FFACTOR*TPERIOD)
Frames	Messages	Set by the user
DATA[0]	CanFrm->DATA[0]	0x00

A frame is send with a $period$ set by the user, calculate the cycles that correspond the the $period$ and load the value to the SCU timer; Start the count down and wait for an interrupt to occur.

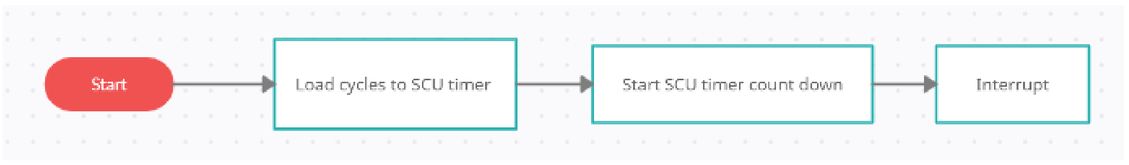


Figure 4.3: Phase 1

After the interrupt occurred the transmission of the frame will start and check if a frame is already in progress of transmitting. If it is not then the SCU timer will start the count down again,

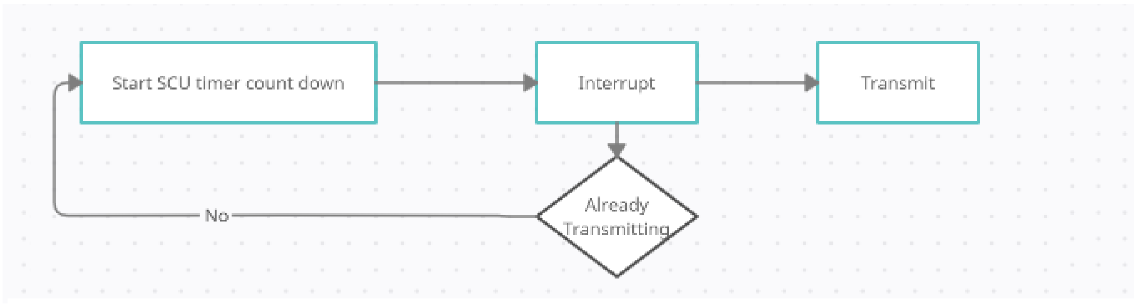


Figure 4.4: Phase 2 case 1

However if a transmission is already in progress then one of AXI's timers will be enabled to count the extra period needed,wait until the send ends. Next it will convert the additional period to cycles and reload The SCU timer.

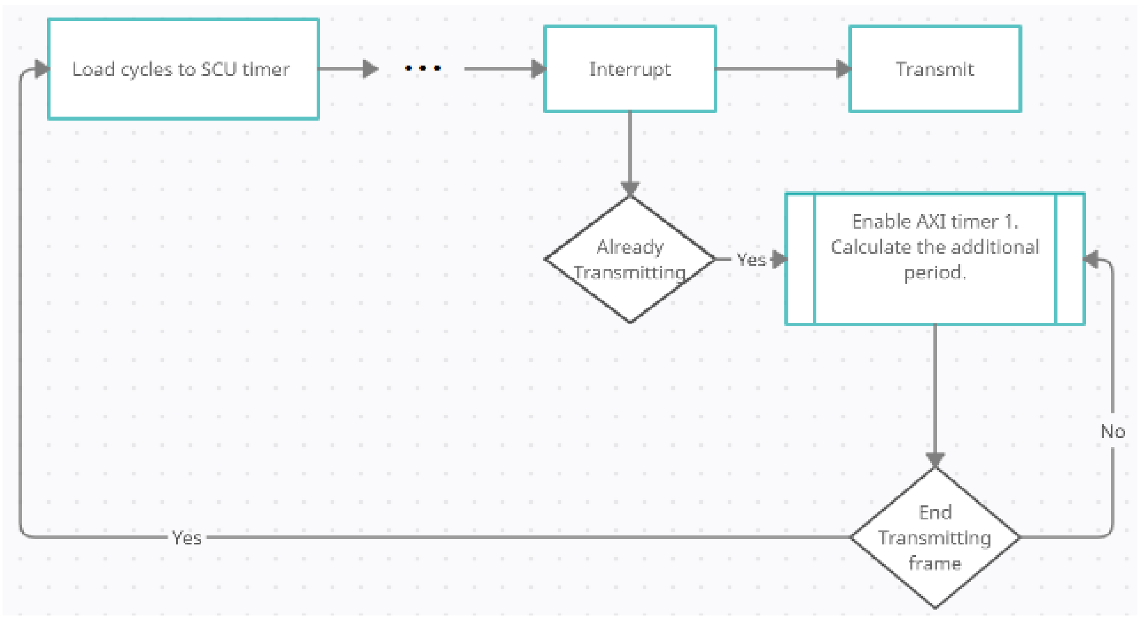


Figure 4.5: Phase 2 case 2

This will be repeating until the $period$ becomes the optimal for the system. Furthermore, there is the case with the minor adjustments that AXI timer$_1$ manage. Minor changes can happen either after the period changed or before depends on the $period$ set by the user.

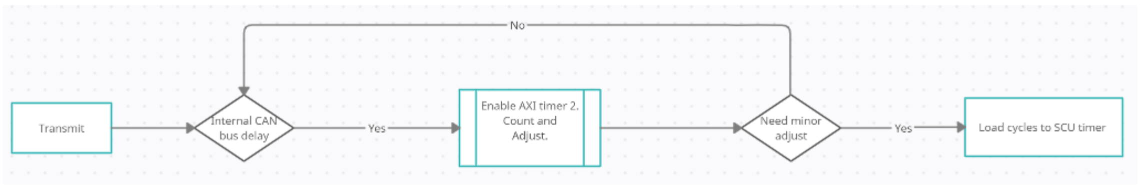


Figure 4.6: Phase 2 case 3

The was stable message frequency and better synchronized nodes.So changing the frequency to an optimal for sending messages was what we aimed.

After seeing all the stages and cases that a transmission of a frame can take enabling different parts of the code, at the follow figure we will see the complete flowchart it looks like.

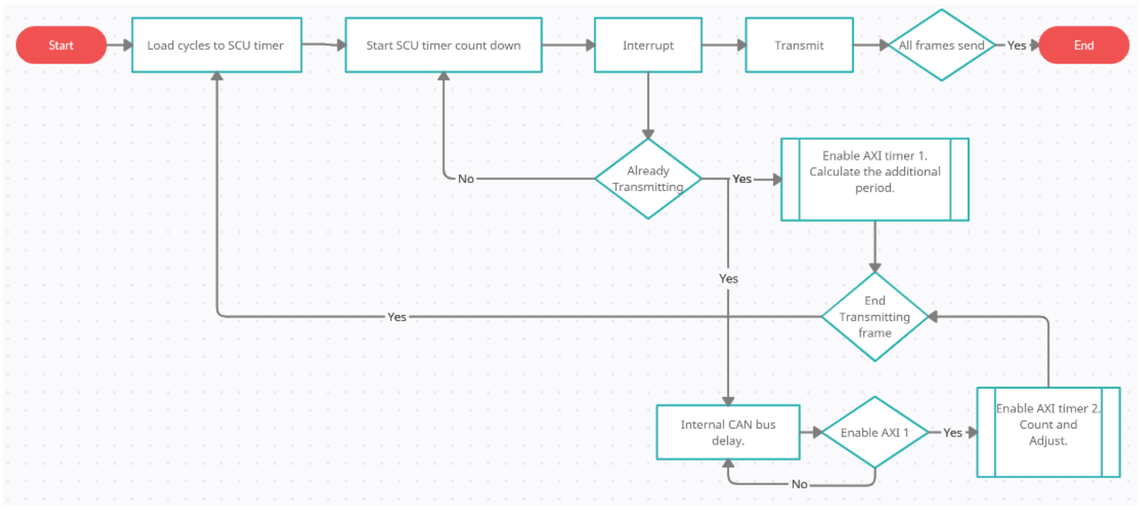


Figure 4.7: Flowchart of the transmitting process

Chapter 5

5.1 Test and results

In this chapter will be shown the samples, the results and we will provide information about them. What happen during the transmissions, what changed and what blocks of code were enabled. Bellow we will see the variables that we already mention at Chapter 4 section 4.1.2. The follow variables of the Tx will be seen and discussed later on.

Transmitter(Tx):

- SCU$_{cycles}$: Is the Load Value representing Cycles Loaded to SCU timer
- Interrupt$_{SCU}$: is The timer Expired representing how many times SCU timer expired
- AXIcounter$_0$:If the AXI timer$_0$ was used for major changes in $period$
- AXIcounter$_1$:If the AXI timer$_1$ was used for minor changes in $period$
- Delay in side the Bus : txBusyPkt the delay inside of the bus during the transmission.
- Adaption Times :How many times the $period$ change.
- Last SCU$_{cycles}$: The adjusted value of the SCU$_{cycles}$.
- Time(seconds): Time that Tx needed to send all the frames.

And Receiver(Rx) as well will show us the number of packages that receive, the value of the last package and the time was needed for the whole process.

Receiver(Rx):

- Packets Receive: Frame that was received by Tx.
- Last Package Value: The value of DATA[0] of the last frame that was re-

ceived.

- Time(seconds) : The time needed for the transmission of all frames.
- Time(seconds): Time that Rx needed to receive all the frames.

The test and results that follow was selected based on the robust outcome. Between 500ms and 150 us there was no major changes but later the adjustments needed we took more samples.

SCU period: 500ms

Setting $period$ equal to $\frac{1}{2}$ we accomplish 500ms and running the application multiple times the results was.

Table 5.1: Tx results case 500ms

Run / Result	1	2	3	4	M.O.
SCU$_{cycles}$	166,666,671	166,666,671	166,666,671	166,666,671	166,666,671
Interrupt$_{SCU}$	100	100	100	100	100
Packets Send	100	100	100	100	100
Delay Inside Bus	0	0	0	0	0
AxiTimer$_0$	0	0	0	0	0
AxiTimer$_1$	0	0	0	0	0
Adaption Times	0	0	0	0	0
Last SCU$_{cycles}$	166,666,671	166,666,671	166,666,671	166,666,671	166,666,671
Time(seconds)	50.000004	50.000004	50.000004	50.000004	50.000004

As we saw at Table 5.1 the $period$ between frames was long enough so no changes happen. Therefore at Figure 5.1 we can see a graph behaviour of the SCU$_{cycles}$ in every transmission of the frame and Figure 5.2 the analytic values of cycles that is loaded in SCU timer from frame to frame send per send.

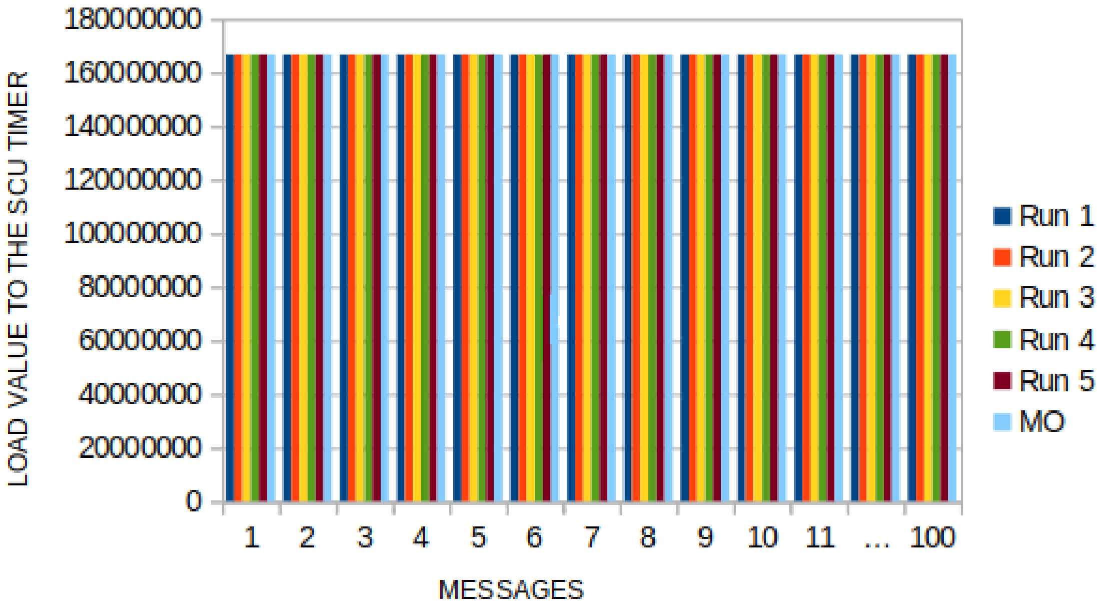


Figure 5.1: Graph SCU$_{cycles}$ per Packet

Here we see the value of SCU$_{cycles}$ in every transmission .

MESSAGES	Run 1	Run 2	Run 3	Run 4	Run 5	MO
1	166666671	166666671	166666671	166666671	166666671	166666671
2	166666671	166666671	166666671	166666671	166666671	166666671
3	166666671	166666671	166666671	166666671	166666671	166666671
4	166666671	166666671	166666671	166666671	166666671	166666671
5	166666671	166666671	166666671	166666671	166666671	166666671
6	166666671	166666671	166666671	166666671	166666671	166666671
7	166666671	166666671	166666671	166666671	166666671	166666671
8	166666671	166666671	166666671	166666671	166666671	166666671
9	166666671	166666671	166666671	166666671	166666671	166666671
10	166666671	166666671	166666671	166666671	166666671	166666671
11	166666671	166666671	166666671	166666671	166666671	166666671
...	166666671	166666671	166666671	166666671	166666671	166666671
100	166666671	166666671	166666671	166666671	166666671	166666671

Figure 5.2: Analytic SCU$_{cycles}$ per frame

Rx results identical except time as we can see at Table 5.2 . The times receiving cant me identical cause we in real time system many factor can change the results.

Table 5.2: Rx results

Result \ Run	1	2	3	4	MO
Packets Received	100	100	100	100	100
Last Packet Value	99	99	99	99	99
Time(seconds)	50.476523	50.372872	50.416506	50.556708	50.455652

Without the need to change the SCU$_{cycles}$ in our program we observe that the $period$ was long enough to need adjustments. Without adjustments the only blocks or lines of code were enabled was:

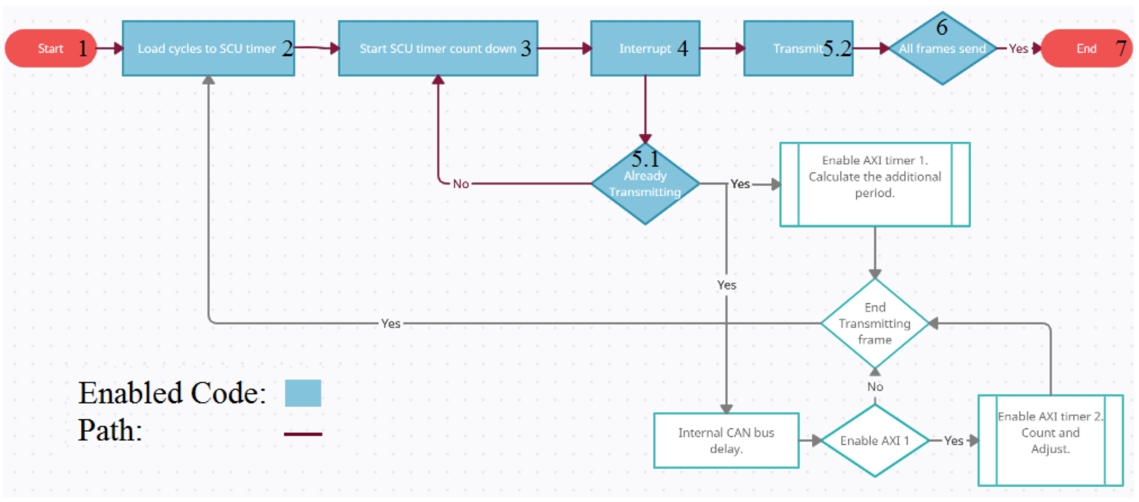


Figure 5.3: Blocks/lines of code enabled in 500ms

With SCU$_{cycles}$ staying the same from the start to the end and no AXI timer enabled. As we see in Figure 5.3 the program run with the follow steps:

1. Start the program
2. Load the calculated value as cycles to SCU timer: 166,666,671.
3. The SCU timer start the count down: Until SCU timer equal to 0.
4. When it end, an interrupt occurs:
5. Simultaneously:
   - 5.1. Check if there is already a frame transmitting.
   - 5.2. Transmit the frame.
6. Check if the all the frames has been send:100/100.
7. End:Show the results and Terminate the program .

Step 4 has two paths that happens at the same time and both of them has different goals. Path 5.1 is to check if there is another frame in the transmit process and path 5.2 the transmit process. However even if step 5.1 depends on step 5.2 will happen parallelly so time is not wasted and when the step 6 is completed everything unnecessary for the rest of the program will stop.

SCU period: 1ms

Setting $period$ equal to $\frac{1.}{1000}$ we accomplish 1 ms.

Table 5.3: Tx results case 1ms

Run / Result	1	2	3	4	M.O.
SCU$_{cycles}$	333,333	333,333	333,333	333,333	333,333
Interrupt$_{SCU}$	100	100	100	100	100
Packets Send	100	100	100	100	100
Delay Inside Bus	0	0	0	0	0
AxiTimer$_0$	0	0	0	0	0
AxiTimer$_1$	0	0	0	0	0
Adaption Times	0	0	0	0	0
Last SCU$_{cycles}$	333,333	333,333	333,333	333,333	333,333
Time(seconds)	0.100005	0.100005	0.100005	0.100005	0.100005

As we see in Table 5.3 frames was send without a delay, no changes in frequency was needed. Figure 5.4 show as the behaver oh SCU$_{cycles}$ per run. In every transmit the SCU$_{cycles}$ didn't change in each run.

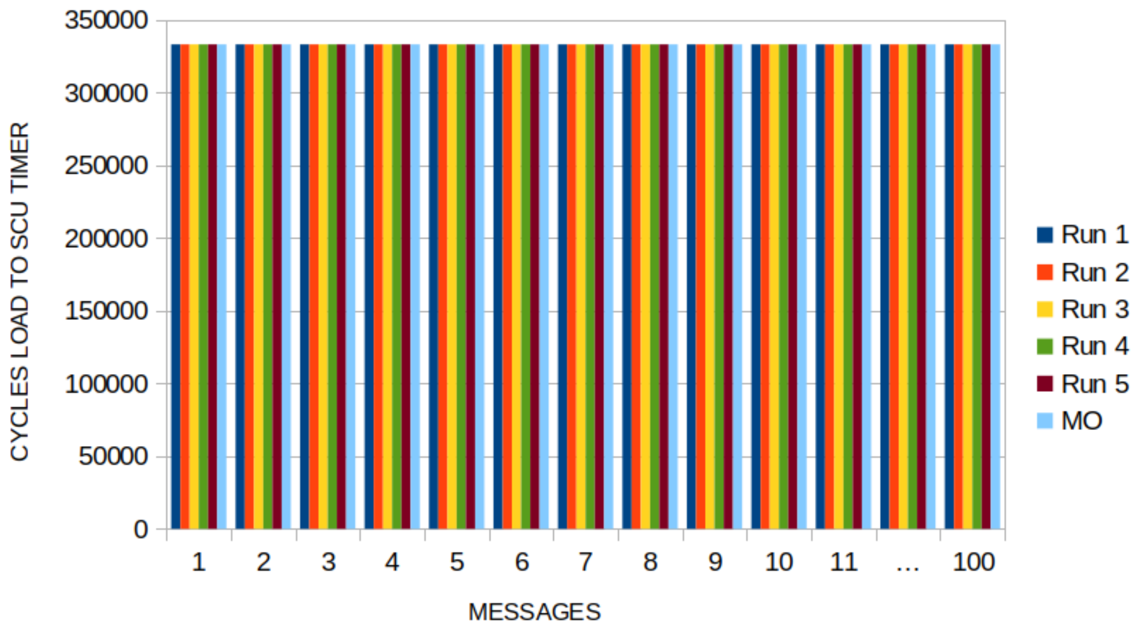
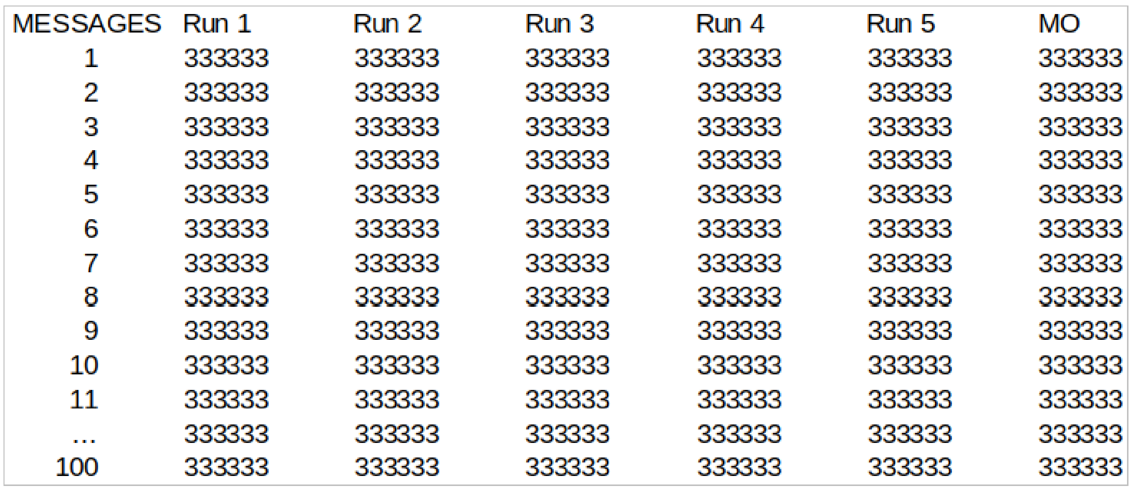


Figure 5.4: Graph SCU$_{cycles}$ per Packet

Figure 5.5 show as analytic values of SCU$_{cycles}$ for every test of the sequence. After testing it multiple times non of the wanted results didn't differ from each other. Pointing out that the period overall was long enough not even need the default delay of the bus as we see in table 5.3 variable Inside the bus was equal to 0.



MESSAGES	Run 1	Run 2	Run 3	Run 4	Run 5	MO
1	333333	333333	333333	333333	333333	333333
2	333333	333333	333333	333333	333333	333333
3	333333	333333	333333	333333	333333	333333
4	333333	333333	333333	333333	333333	333333
5	333333	333333	333333	333333	333333	333333
6	333333	333333	333333	333333	333333	333333
7	333333	333333	333333	333333	333333	333333
8	333333	333333	333333	333333	333333	333333
9	333333	333333	333333	333333	333333	333333
10	333333	333333	333333	333333	333333	333333
11	333333	333333	333333	333333	333333	333333
...	333333	333333	333333	333333	333333	333333
100	333333	333333	333333	333333	333333	333333

Figure 5.5: Analytic SCU$_{cycles}$ per Packet

As we see in the graph and analytic as the values stayed same until the end of our testing. And Rx results was:

Table 5.4: Rx results

Run / Result	1	2	3	4	MO
Packets Received	100	100	100	100	100
Last Packet Value	99	99	99	99	99
Time(seconds)	0.487357	0.311256	0.511107	0.317147	0,40671675

This case has the same results with the period of 500 ms. Only the values of SCU$_{cycles}$ and the time needed to execute the whole transmission were different. Without major or minor changes in the frequency the line or blocks of code were enabled was:

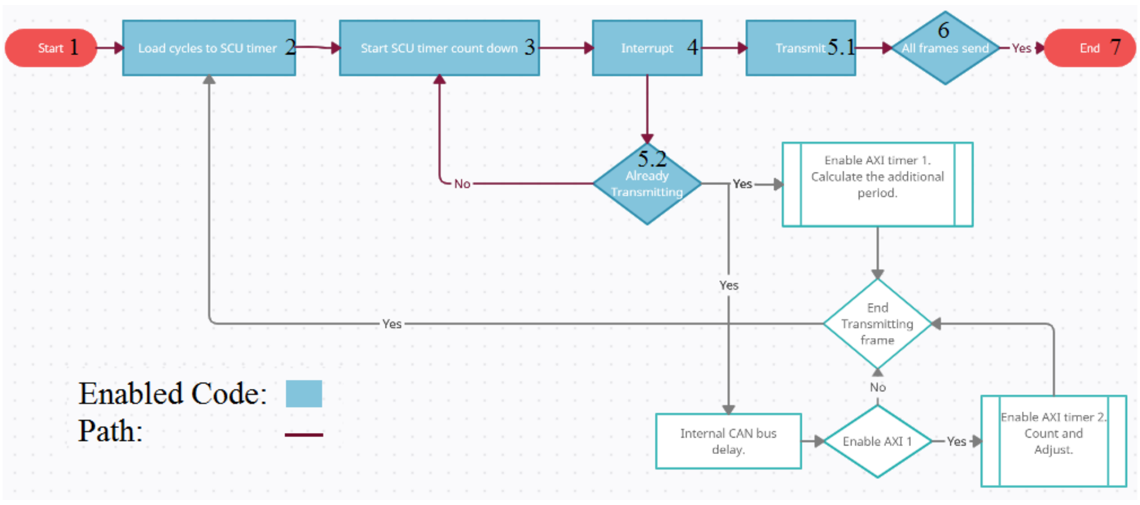


Figure 5.6: Analytic SCU$_{cycles}$ per Packet

1. Start the program
2. Load the calculated value as cycles to SCU timer: 333,333.
3. The SCU timer start the count down: SCU timer until is equal to 0.
4. When it end, an interrupt occurs:
5. Simultaneously:
   - 5.1. Check if there is already a frame transmitting.
   - 5.2. Transmit the frame.
6. Check if the all the frames has been send:100/100.
7. End:Show the results and Terminate.

As we can see after testing 500 ms and then 1 ms we skipped many periods that we could show, but the results would be the same with the only value that differs would be the execute time. Notice that even after this major skip of frequency the results didn't change and not even a delay inside the bus occurred. This was bound to happen as we read at at Chapter 3 section 3.1.1 the research of Hans-Christian Reuss.

SCU period: 150us

With many trails the first minor differences appeared in 150 us. Setting the $period$ to $\frac{150.}{100,000}$(150us) we can see the results at Table 5.5. The CAN bus could send the frames properly without any error because the default code had an mechanism to delay a frame if the bus was not busy. So this mechanism handle some of the frames.

Table 5.5: Tx results case 150us

Run / Result	1	2	3	4	M.O.
SCU$_{cycles}$	50,0000	50,0000	50,0000	50,0000	50,0000
Interrupt$_{SCU}$	100	100	100	100	100
Packets Send	100	100	100	100	100
Delay Inside Bus	161	8	157	42	131
AxiTimer$_0$	0	0	0	0	0
AxiTimer$_1$	0	0	0	0	0
Adaption Times	0	0	0	0	0
Last SCU$_{cycles}$	50,0000	50,0000	50,0000	50,0000	50,0000
Time(seconds)	0.100005	0.100005	0.100005	0.100005	0.100005

As we already discuss based on Hans-Christian Reuss research and our maximum bit frame; theoretically in period equal to 150us it should have be changed. This should happen because our frame has the length of 155 bits and it would need at least 155us to be transmitted. However, changing the value of DATA[0] we change the bit-stuffing as well, as a result we have a frame with the length of 139 bits. This is where we can see that the theoretical approach differs from practical application. Even with the frame of 139 bits the frequency of 150 us enabled only the default delay of the bus. The default delay was enough to handle the message transmit frequency, as result no changes needed as we saw at table 5.5 and we will see at the follow graph Figure 5.9.

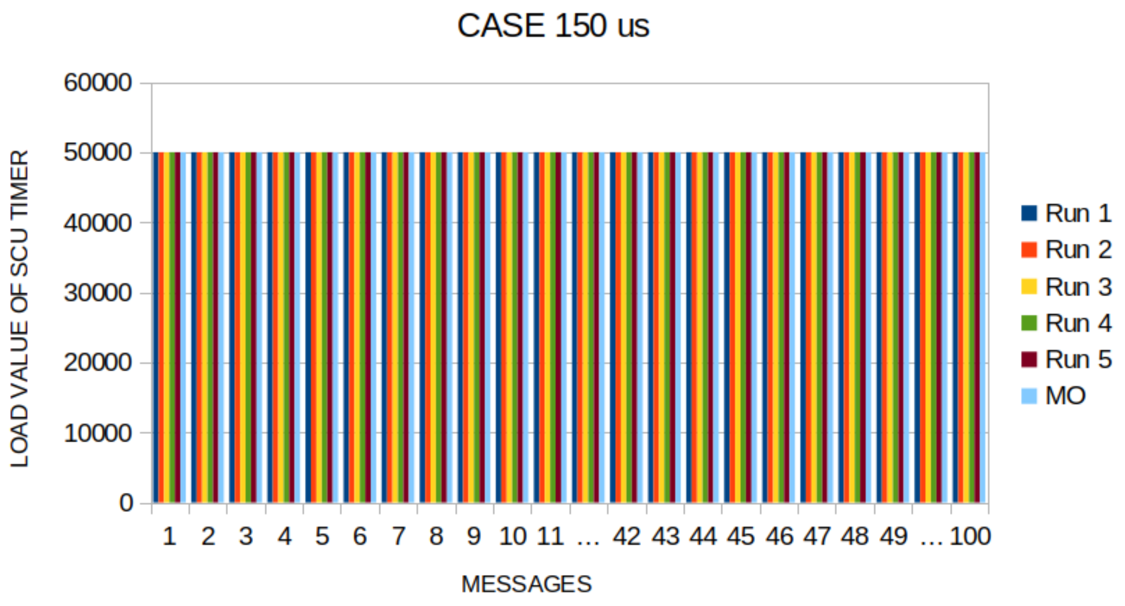
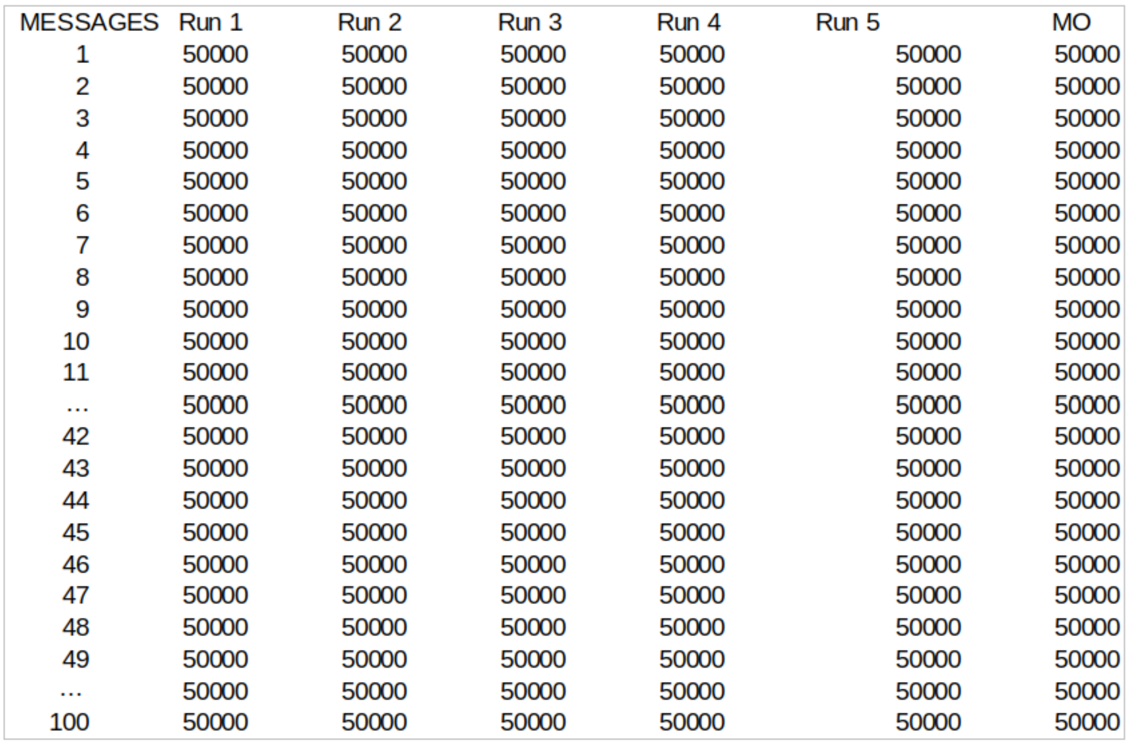


Figure 5.7: Graph SCU$_{cycles}$ per Packet

As we can see at analytic values at Figure 5.8 the follow values in every transmission stayed the same because there was no need either for major or minor adjust of the period.



MESSAGES	Run 1	Run 2	Run 3	Run 4	Run 5	MO
1	50000	50000	50000	50000	50000	50000
2	50000	50000	50000	50000	50000	50000
3	50000	50000	50000	50000	50000	50000
4	50000	50000	50000	50000	50000	50000
5	50000	50000	50000	50000	50000	50000
6	50000	50000	50000	50000	50000	50000
7	50000	50000	50000	50000	50000	50000
8	50000	50000	50000	50000	50000	50000
9	50000	50000	50000	50000	50000	50000
10	50000	50000	50000	50000	50000	50000
11	50000	50000	50000	50000	50000	50000
…	50000	50000	50000	50000	50000	50000
42	50000	50000	50000	50000	50000	50000
43	50000	50000	50000	50000	50000	50000
44	50000	50000	50000	50000	50000	50000
45	50000	50000	50000	50000	50000	50000
46	50000	50000	50000	50000	50000	50000
47	50000	50000	50000	50000	50000	50000
48	50000	50000	50000	50000	50000	50000
49	50000	50000	50000	50000	50000	50000
…	50000	50000	50000	50000	50000	50000
100	50000	50000	50000	50000	50000	50000

Figure 5.8: Graph SCU$_{cycles}$ per Packet

The results of Rx will show how much time was needed for the receive of all packages in every case. This case was the first one that a delay occurred, but seeing that Rx show us the results means that the frames was receive by the order that were made in Tx. Otherwise, as we already mention in chapter 4 section 4.1.1 Rx would deny the messages. The results of Rx was:

Table 5.6: Rx results

Run / Result	1	2	3	4	MO
Packets Received	100	100	100	100	100
Last Packet Value	99	99	99	99	99
Time(seconds)	0.219557	0.219558	0.212057	0.220015	0.21779675

The code that was enabled at 150 us and the code lines/blocks was :

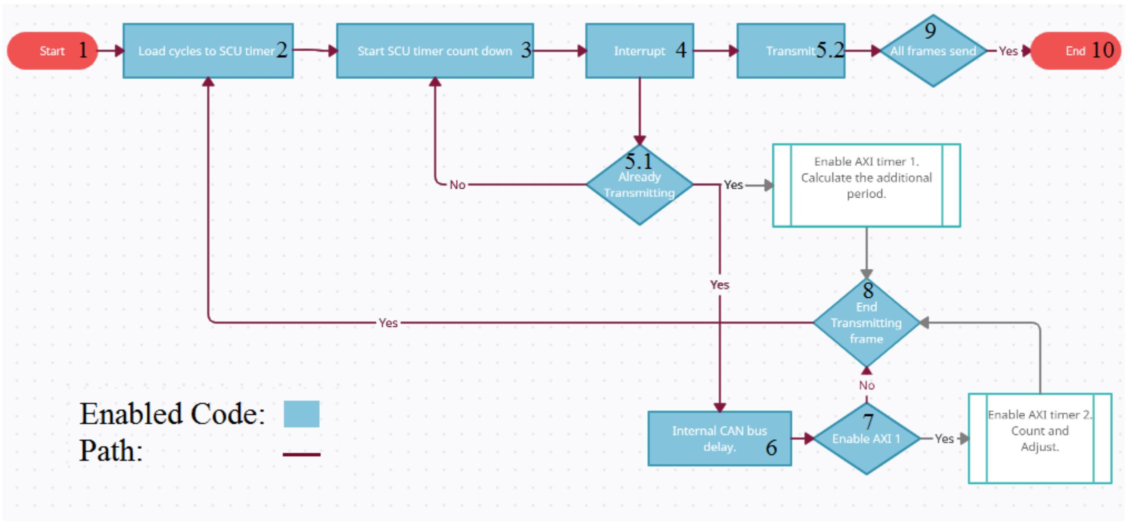

Start 1
Load cycles to SCU timer 2
Start SCU timer count down 3
Interrupt 4
Transmit 5.2
9 All frames send
Yes
End 10
5.1 Already Transmitting
No
Yes
Enable AXI timer 1. Calculate the additional period.
Yes
8 End Transmitting frame
Yes
No
Enabled Code:
Path:
Internal CAN bus delay. 6
7 Enable AXI 1
Yes
Enable AXI timer 2. Count and Adjust.

Figure 5.9: Graph SCU$_{cycles}$ per Packet

1. Start the program
2. Load the calculated value as cycles to SCU timer: 50,000.
3. The SCU timer start the count down: SCU timer until is equal to 0.
4. When it end, an interrupt occurs (at the same time):
5. Simultaneously:

    5.1. Check if there is already a frame transmitting.

    5.2. Transmit the frame.

6. Frame entering the CAN bus default delay.
7. Check if the Delay is long to enable AXI timer$_1$.
8. Adjust or not the Period of the system.
9. Check if the all the frames has been send:100/100.
10. End:Show the results and end the program.

As we saw after step 4 there are two paths of our program that happens at the same time. Step 5.1 can lead to a major or minor adjustment and 5.2 is the transmit. The process of transmit can be delayed if there is already a frame in this procedure, entering in the default wait of the bus. If the requirements are met then a minor or a major adjustment can happen. In case that the frame stays inside the delay for long enough then AXI timer$_1$ will be enabled to adjust the frequency, if another interrupt occur then AXI timer$_0$ will be enabled for a major adjustment. With $period$ equal to 150 us, the frame stay only inside the default delay and no interrupt occurred. However, the delay wasn't enough to enable AXI timer$_1$, so no adjustments happen.

SCU period: 140ms

Setting $period$ at $\frac{140.}{100{,}000}$ the first major change happen. However because of the adaptiveness, our application didn't lost any packages. After $period$ changed even the delay inside bus was 0 cause the frequency was optimal to transmit packets in this system.

Table 5.7: Tx results case 140us

Run / Result	1	2	3	4	M.O.
SCU$_{cycles}$	46,666	46,666	46,666	46,666	46,666
Interrupt$_{SCU}$	100	100	100	100	100
Packets Send	100	100	100	100	100
Delay Inside Bus	0	0	0	0	0
AxiTimer$_0$	14,112	14,107	14,116	14,106	14,110
AxiTimer$_1$	0	0	0	0	0
Adaption Times	1	1	1	1	1
Last SCU$_{cycles}$	56,448	56,428	56,464	56,444	56,441
Time(seconds)	0.015677	0.015673	0.015680	0.015671	0.015674

As we already mention and we can see at table 5.7 an interrupt occurred before the previous frame was send enabling AXI timer$_0$. Until the previous frame end the transmission AXI timer$_0$ count the extra period that we need add in SCU timer. The average value of additional cycles we needed was 14.110 cycles. Figure 5.10 that the SCU$_{cycles}$ until the delay started to pounding. At that point an Interrupt$_{SCU}$ has occurred before the last one transmitted.So the AXI Timer$_0$ enabled and starting to count for optimal frequency. After the message transmission served the next value SCU$_{cycles}$ increased for no furthermore delay. We can see as result at table 5.7 there is no delay inside the bus too.

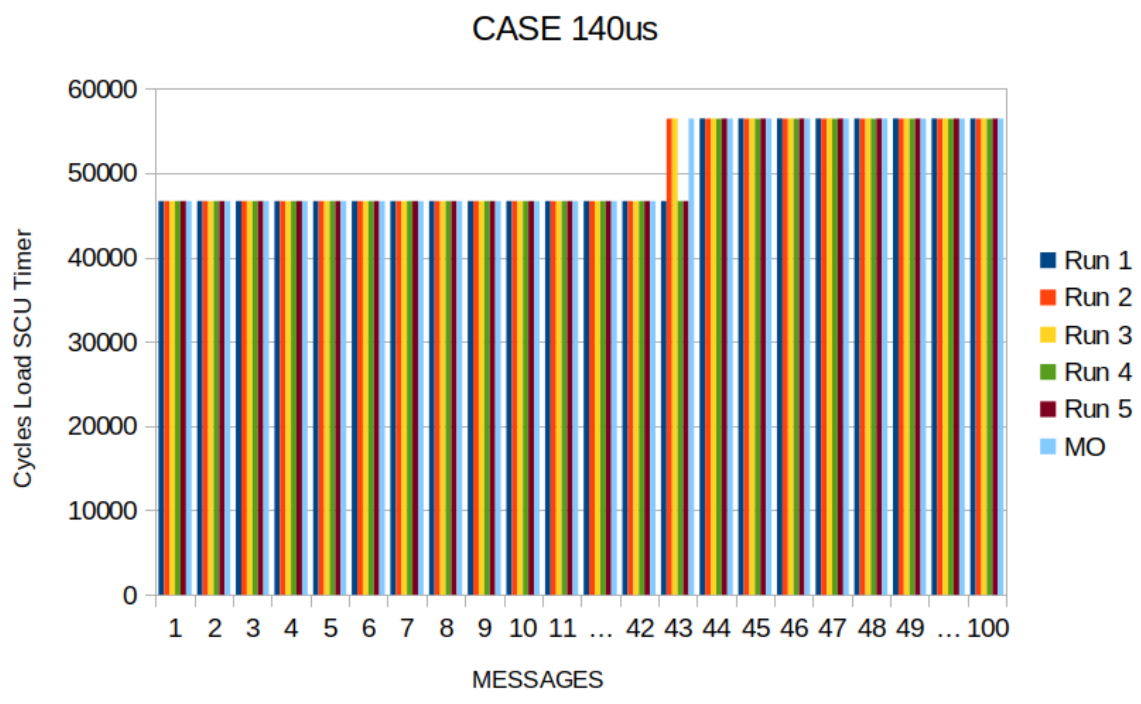
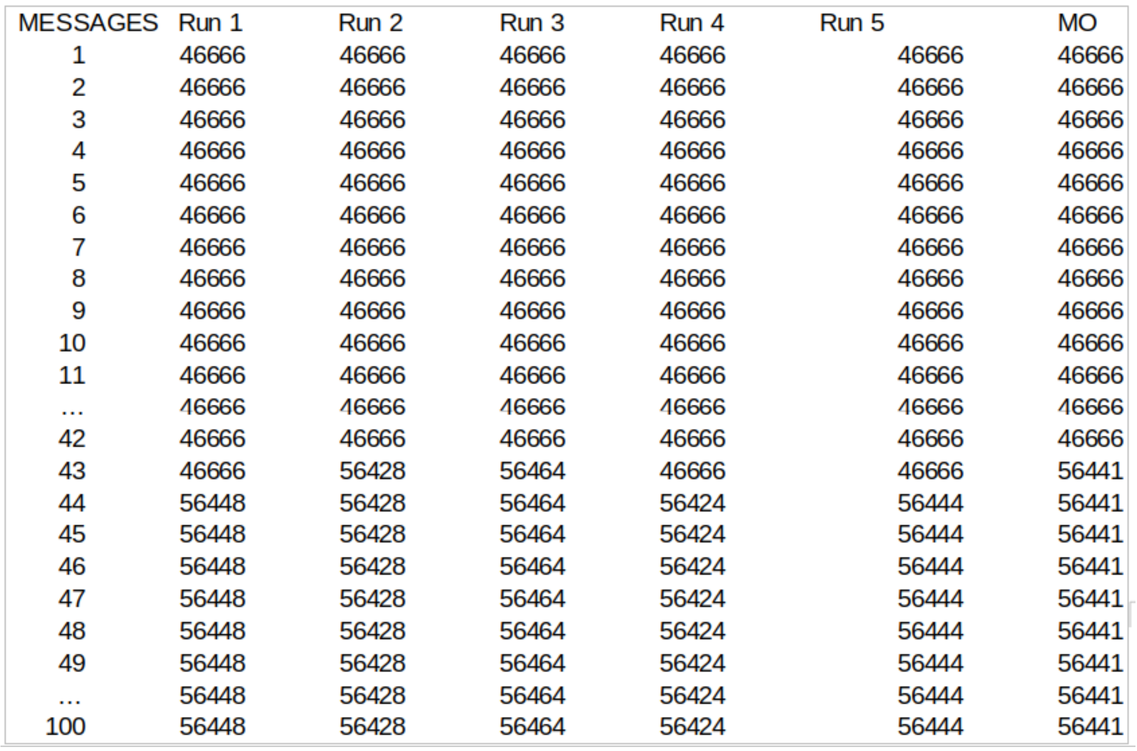


Figure 5.10: Graph SCU$_{cycles}$ per Packet

Figure 5.11 will show the exact values and on what messages the SCU$_{cycles}$ changed. We can see that even if the hardware or software was the same, results wasn't identical however they were close enough.



MESSAGES	Run 1	Run 2	Run 3	Run 4	Run 5	MO
1	46666	46666	46666	46666	46666	46666
2	46666	46666	46666	46666	46666	46666
3	46666	46666	46666	46666	46666	46666
4	46666	46666	46666	46666	46666	46666
5	46666	46666	46666	46666	46666	46666
6	46666	46666	46666	46666	46666	46666
7	46666	46666	46666	46666	46666	46666
8	46666	46666	46666	46666	46666	46666
9	46666	46666	46666	46666	46666	46666
10	46666	46666	46666	46666	46666	46666
11	46666	46666	46666	46666	46666	46666
…	46666	46666	46666	46666	46666	46666
42	46666	46666	46666	46666	46666	46666
43	46666	56428	56464	46666	46666	56441
44	56448	56428	56464	56424	56444	56441
45	56448	56428	56464	56424	56444	56441
46	56448	56428	56464	56424	56444	56441
47	56448	56428	56464	56424	56444	56441
48	56448	56428	56464	56424	56444	56441
49	56448	56428	56464	56424	56444	56441
…	56448	56428	56464	56424	56444	56441
100	56448	56428	56464	56424	56444	56441

Figure 5.11: Analytic SCU$_{cycles}$ per Packet

The results of Rx will show that in every case all packets arrived in "safe" and the times that needed

Table 5.8: Rx results

Run / Result	1	2	3	4	MO
Packets Received	100	100	100	100	100
Last Packet Value	99	99	99	99	99
Time(seconds)	0.415609	0.415591	0.415621	0.415623	0.415611

The first major adjustment happen with our $period$ equal to 140 us and our frames having the size 139 bits. For this to happen the blocks and lines of code that were enabled was :

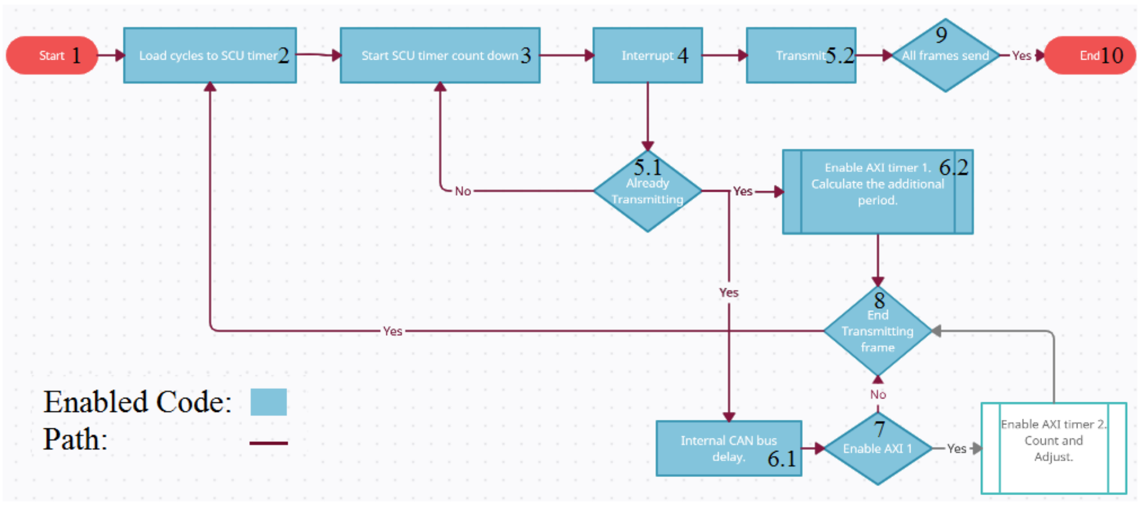

Start 1
Load cycles to SCU timer 2
Start SCU timer count down 3
Interrupt 4
Transmit 5.2
9 All frames send
Yes
End 10
5.1 Already Transmitting
No
Yes
Enable AXI timer 1. Calculate the additional period. 6.2
Yes
8 End Transmitting frame
Yes
No
Internal CAN bus delay. 6.1
7 Enable AXI 1
Yes
Enable AXI timer 2. Count and Adjust.
Enabled Code:
Path:

Figure 5.12: Analytic SCU$_{cycles}$ per Packet

And the follow steps that explain the figure 5.12 :

1. Start the program
2. Load the calculated value as cycles to SCU timer: 46,666.
3. The SCU timer start the count down: SCU timer until is equal to 0.
4. When it end, an interrupt occurs.
5. Simultaneously:

   5.1. Check if there is already a frame transmitting.

   5.2. Transmit the frame.

6. Simultaneously:

   6.1. Frame entering the CAN bus default delay.

   6.2. Enable AXI timer$_0$ and adjust the AXI$_{cycles}$ to SCU$_{cycles}$.

7. Check if the Delay is long to enable AXI timer$_1$.

8. The end of the transmission of the last frame and Load the new value of the cycles to SCU timer.

9. Check if the all the frames has been send:100/100.

10. End:Show the results and end the program.

Step 4 has two paths that will be enabled simultaneously with the interrupt, path 5.1 was enabled properly until frame 43. At frame 43 the block of 5.2 was enabled and led to blocks of code 6.1 and 6.2. The frame insert in the internal delay and the AXI timer$_0$ start to count. Until the AXI timer$_0$ ends, path 7 was checked to see if there is need to enable AXI timer$_1$. When our frame start his transmission, AXI timer$_0$ stop the count and a specific block of code made the adjustment and load the new value as SCU$_{cycles}$ in to SCU timer. With the SCU timer having a new value, the period changed and continue until all the frames was sent without further changes. That conclude the program and show us the results of the new adjusted period.

SCU period: 120 us

The $period$ was set at $\frac{120.}{100{,}000}$. cause of the adaptiveness, our application didn't lost any packages. Even after the $period$ changed the delay inside bus was accumulated so minor adjustment occurred by AXI timer$_1$.

Table 5.9: Tx results case 120us

Run / Result	1	2	3	4	M.O.
SCU$_{cycles}$	40,000	40,000	40,000	40,000	40,000
Interrupt$_{SCU}$	100	100	100	100	100
Packets Send	100	100	100	100	100
Delay Inside Bus	42	98	60	13	53
AxiTimer$_0$	12,111	12,113	11,700	11,199	11,780
AxiTimer$_1$	3,156	3,140	1,692	3,328	2,579
Adaption Times	1	1	1	1	1
Last SCU$_{cycles}$	51,600	51,592	50,492	51,324	51,252
Time(seconds)	0.015419	0.015416	0.015303	0.015498	0.015489

As we saw at table 5.11, AXI timer$_1$ was enabled to do the final adjustment for our system. The result of this was for an utmost synchronisation of the messages maintaining the delay inside the bus at low values. We can see that the follow figure 5.13 and figure 5.14 the bus kept a steady transmit sequence until frame 93 that needed the major adjustment. However the internal delay accumulated and a second adjust happen to the frequency.

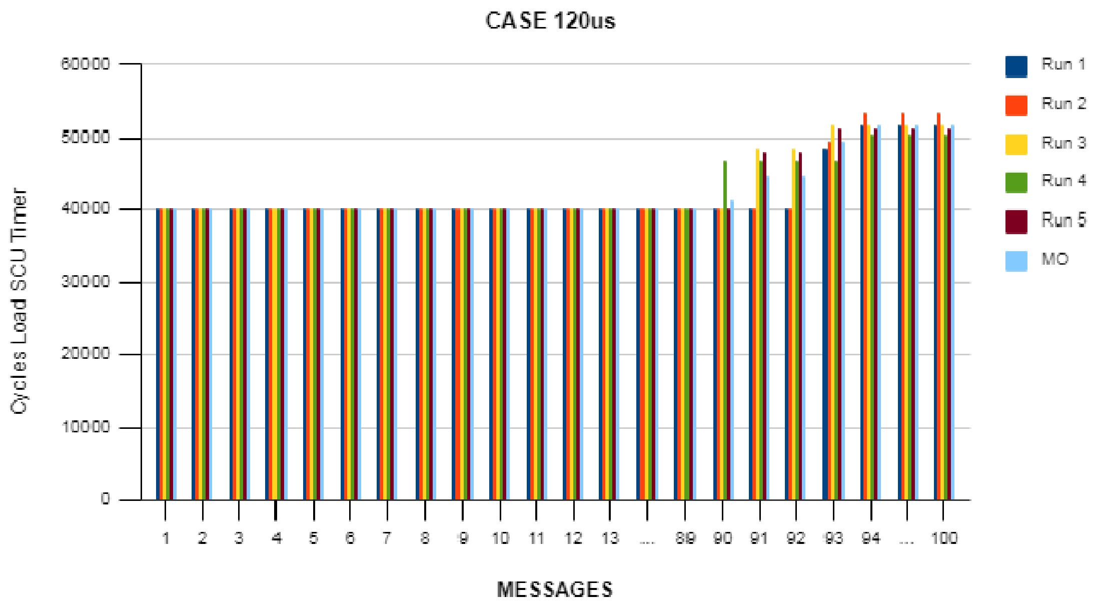
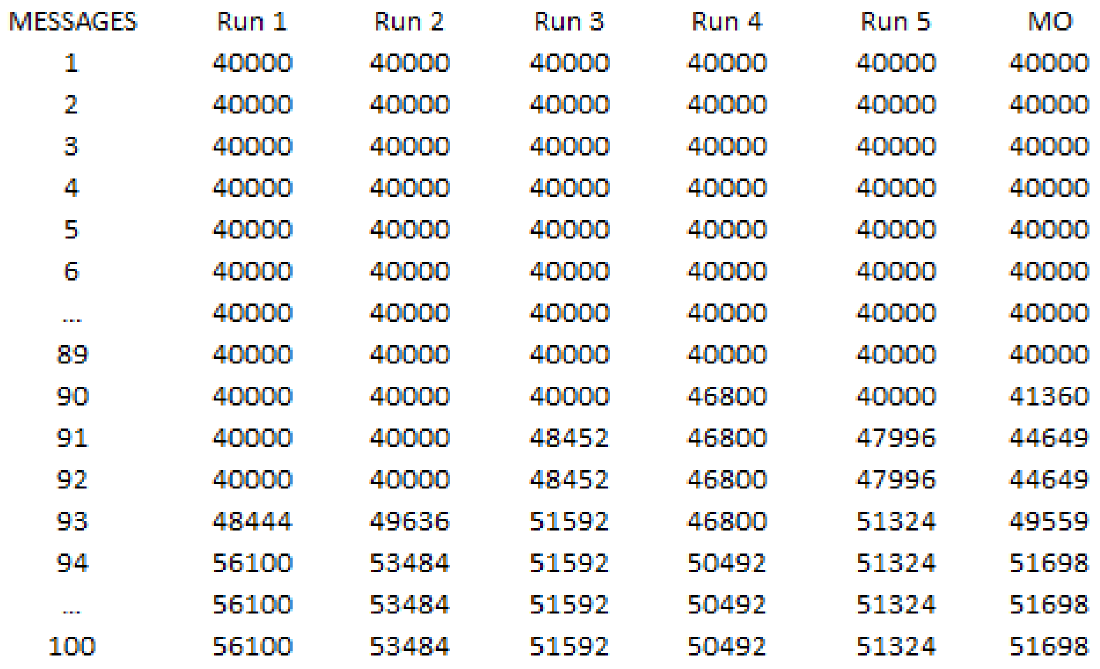


Figure 5.13: Graph SCU$_{cycles}$ per Packet

And here we see the specific changes and values of the SCU$_{cycles}$ per transmission of the frames.



MESSAGES	Run 1	Run 2	Run 3	Run 4	Run 5	MO
1	40000	40000	40000	40000	40000	40000
2	40000	40000	40000	40000	40000	40000
3	40000	40000	40000	40000	40000	40000
4	40000	40000	40000	40000	40000	40000
5	40000	40000	40000	40000	40000	40000
6	40000	40000	40000	40000	40000	40000
...	40000	40000	40000	40000	40000	40000
89	40000	40000	40000	40000	40000	40000
90	40000	40000	40000	46800	40000	41360
91	40000	40000	48452	46800	47996	44649
92	40000	40000	48452	46800	47996	44649
93	48444	49636	51592	46800	51324	49559
94	56100	53484	51592	50492	51324	51698
...	56100	53484	51592	50492	51324	51698
100	56100	53484	51592	50492	51324	51698

Figure 5.14: Graph SCU$_{cycles}$ per Packet

The results of Rx in period equal to 120 us was:

Table 5.10: Rx results

Run / Result	1	2	3	4	MO
Packets Received	100	100	100	100	100
Last Packet Value	99	99	99	99	99
Time(seconds)	0,450890	0,450894	0,450846	0,450899	0,450882

This was the first case that the two adjustments happen enabling the follow block code we will see at the follow figure.

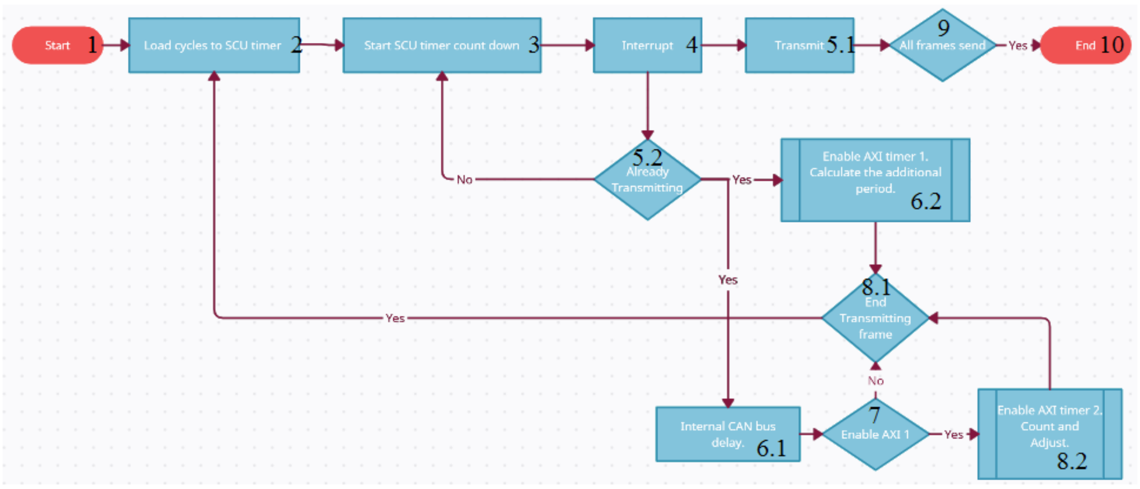

Start 1
Load cycles to SCU timer 2
Start SCU timer count down 3
Interrupt 4
Transmit 5.1
9 All frames send
Yes
End 10
5.2 Already Transmitting
No
Yes
Enable AXI timer 1. Calculate the additional period. 6.2
Yes
Yes
8.1 End Transmitting frame
No
Internal CAN bus delay. 6.1
7 Enable AXI 1
Yes
Enable AXI timer 2. Count and Adjust. 8.2

Figure 5.15: Graph SCU$_{cycles}$ per Packet

The flow of our system executing the block codes is explained in the follow steps :

1. Start the program
2. Load the calculated value as cycles to SCU timer: 40.000
3. The SCU timer start the count down: SCU timer until is equal to 0.
4. When it end, an interrupt occurs.
5. Simultaneously:

    5.1. Check if there is already a frame transmitting.

5.2. Transmit the frame.

6. Simultaneously:

   6.1. Frame entering the CAN bus default delay.

   6.2. Enable AXI timer$_0$ and adjust the AXI$_{cycles}$ to SCU$_{cycles}$.

7. Check if the Delay is long to enable AXI timer$_1$.

8. Either:

   8.1. The end of the transmission of the last frame and Load the new value of the cycles to SCU timer..

   8.2. Enable AXI timer$_1$ count and adjust the AXI$_{cycles}$ to SCU$_{cycles}$.

9. Check if the all the frames has been send:100/100.

10. End:Show the results and end the program.

Until now the as we already discuss many of the steps is the same, the difference is that in Step 8 there is two separate path that the system will follow. However, step 8.2 will lead to step 8.1 when the transmit of the last frame end his transmit. In this case the after the major adjustment happen another was was needed and the block of code 8.2 was enabled. The MO was 2.579 cycles of the AXI timer$_1$ to achieve the optimal synchronisation sequence of the messages and the minimum delay in the internal delay of the bus. Lastly, the rest of the messages was send without any furthermore adjustments.

SCU period: 90 us

The $period$ was set at $\frac{90.}{100,000}$. Because of the adaptiveness, our application didn't lost any packages. However, because some of specification didn't met and the AXI timer$_0$ cycles wasn't enough there were added to the first value of the SCU$_{cycles}$.

Table 5.11: Tx results case 90us

Run / Result	1	2	3	4	M.O.
SCU$_{cycles}$	30,000	30,000	30,000	30,000	30,000
Interrupt$_{SCU}$	100	100	100	100	100
Packets Send	100	100	100	100	100
Delay Inside Bus	269	112	23	87	49
AxiTimer$_0$	9.106	8.936	9.096	8.873	9.009
AxiTimer$_1$	0	0	0	0	0
Adaption Times	1	1	1	1	1
Last SCU$_{cycles}$	66.424	65.744	66.384	65.492	66.036
Time(seconds)	0.019932	0.018156	0.019184	0.016941	0.018778

This was the first case that instead of replacing the already SCU$_{cycles}$ with the new value the program chose to add it to the existed. This happen because to replace the value or to add it some statements of code needed to be fulfilled. In next figure 5.16 .

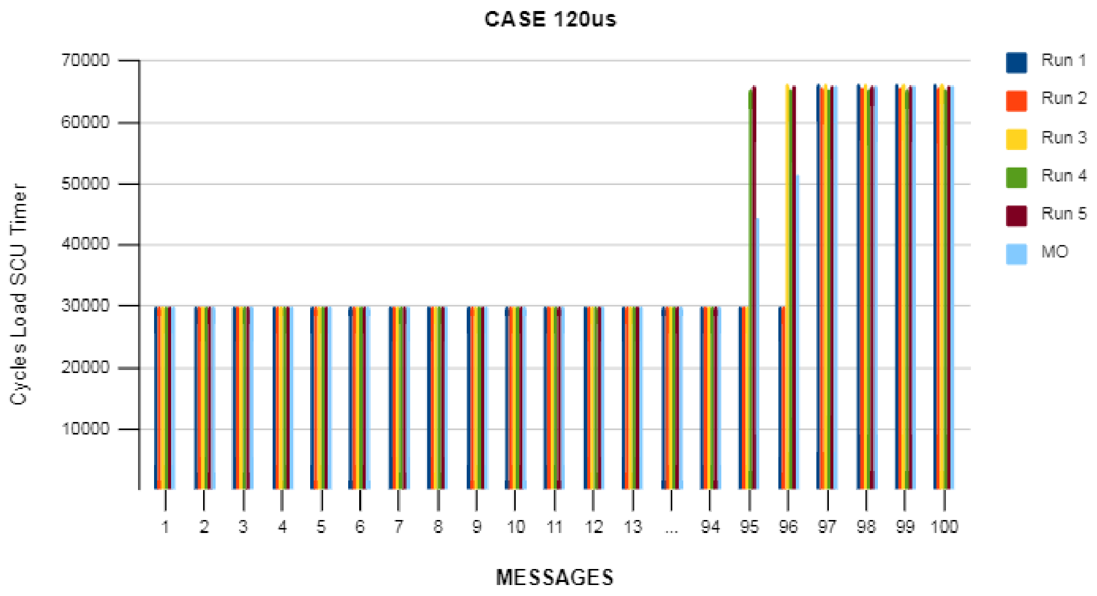
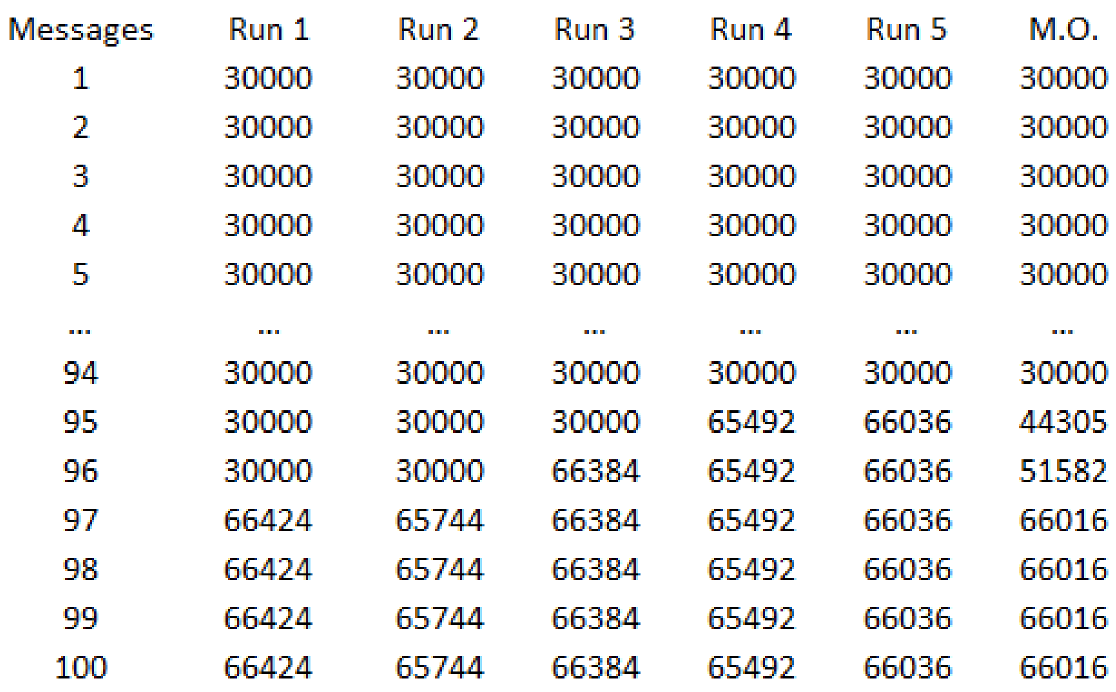


Figure 5.16: Graph SCU$_{cycles}$ per Packet

In the next figure 5.17 we are going to see the analytic changes of the values of the the cycles that was load to SCU timer.



Messages	Run 1	Run 2	Run 3	Run 4	Run 5	M.O.
1	30000	30000	30000	30000	30000	30000
2	30000	30000	30000	30000	30000	30000
3	30000	30000	30000	30000	30000	30000
4	30000	30000	30000	30000	30000	30000
5	30000	30000	30000	30000	30000	30000
...	...	...	...	...	...	...
94	30000	30000	30000	30000	30000	30000
95	30000	30000	30000	65492	66036	44305
96	30000	30000	66384	65492	66036	51582
97	66424	65744	66384	65492	66036	66016
98	66424	65744	66384	65492	66036	66016
99	66424	65744	66384	65492	66036	66016
100	66424	65744	66384	65492	66036	66016

Figure 5.17: Graph SCU$_{cycles}$ per Packet

After the analytic we can see in Rx too that the frame were receive properly:

Table 5.12: Rx results

Run / Result	1	2	3	4	MO
Packets Received	100	100	100	100	100
Last Packet Value	99	99	99	99	99
Time(seconds)	0.9812482	0.9812500	0.98124504	0.9812490	0,981248

Adding the cycles of AXI timer$_0$ to the already existed SCU$_{cycles}$ create an period long enough without the need for furthermore adjustments. So the step to enable the code of block of AXI timer$_1$ was skipped.

The enabled blocks was:

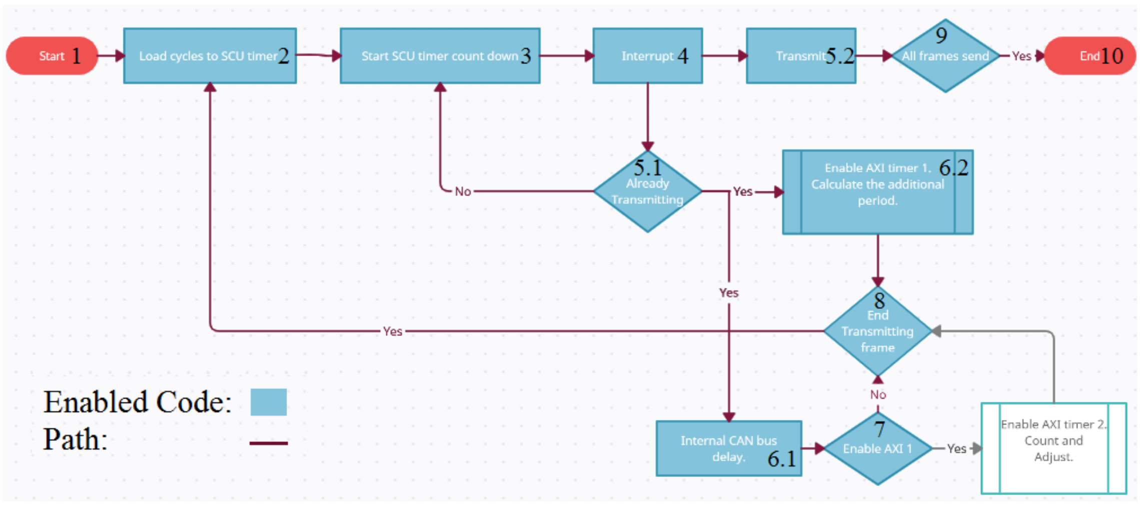

Start 1
Load cycles to SCU timer 2
Start SCU timer count down 3
Interrupt 4
Transmit 5.2
9 All frames send
Yes
End 10
5.1 Already Transmitting
No
Yes
Enable AXI timer 1. Calculate the additional period. 6.2
Yes
8 End Transmitting frame
Yes
No
Internal CAN bus delay. 6.1
7 Enable AXI 1
Yes
Enable AXI timer 2. Count and Adjust.
Enabled Code:
Path:

Figure 5.18: Analytic SCU$_{cycles}$ per Packet

And the next steps explain the flow 5.18.

1. Start the program
2. Load the calculated value as cycles to SCU timer: 30,000.
3. The SCU timer start the count down: SCU timer until is equal to 0.
4. When it end, an interrupt occurs.
5. Simultaneously:
   5.1. Check if there is already a frame transmitting.
   5.2. Transmit the frame.
6. Simultaneously:
   6.1. Frame entering the CAN bus default delay.
   6.2. Enable AXI timer$_0$ and adjust the AXI$_{cycles}$ to SCU$_{cycles}$.
7. Check if the Delay is long to enable AXI timer$_1$.
8. The end of the transmission of the last frame and load the new value of the cycles to SCU timer.
9. Check if the all the frames has been send:100/100.

10. End:Show the results and end the program.

The same steps we already saw them before, however even if in the flow 5.18 is the same the the process inside the block of code in step 6.2 was different. In this case the requirements of changing the whole SCU$_{cycles}$ didn't met, but the requirements to add the cycles to SCU timer met; ignoring the chance to enable AXI timer$_1$ because the adapted frequency fit for the role. In case that the period still had delay in the default mechanism of the bus, it could do a minor adjustment or even a major. Depends on what the program choose as more suitable.

SCU period: Frequency's equalization overflowed

A case of wrong $period$ might occur if the user don't insert a proper value. Setting the $period$ as fraction ($\frac{Numerator}{Denominator}$), user has to put a dot(.) to numerator or the compiler will resulting to an overflow. The mathematical operation is

$$SCU_{Cycles} = (\frac{SCU_{Hz}}{2}) \cdot period \Rightarrow$$

$$SCU_{Cycles} = (\frac{SCU_{Hz}}{2}) \cdot \frac{Numerator}{Denominator}$$

But the Compiler will read it as:

$$SCU_{Cycles} = [\frac{(Numerator \times SCU_{Hz})}{2}] \div Denominator$$

Because of this there is a chance that the result of multiplication will exceed the maximum value of an integer and the compiler will take re remain value of the subtraction: $Overflow_{int}$

$$SCU_{Cycles} = [\frac{Overflow_{int}}{2}] \div Denominator$$

At the end SCU$_{cycles}$ will be inserted with wrong value. Placing the dot(.) to numerator the compiler will operate it as float.

Overflow example:

Table 5.13: Application Overflow Settings

Name	**Value**
period	13/75000
SCU$_{Hz}$	(CPU CORTEXA9 0 CPU CLK FREQ HZ)/ 2
SCU$_{cycles}$	period * SCU$_{Hz}$

The value of zedboard's CPU CORTEXA9 0 CPU CLK FREQ HZ is equal to 666,666,687 and mathematical sequence that will be executed:

$$SCU_{Cycles} = [\frac{(Numerator \times SCU_{Hz})}{2}] \div Denominator \Rightarrow$$

$$SCU_{Cycles} = [\frac{(13 \times 666,666,687)}{2}] \div 75,000 \Rightarrow$$

The result of $\frac{(Numerator \times SCU_{Hz})}{2}$ is equal to 4,333,333,465 when the maximum value of an integer is 2^{32} (4,294,967,296). There is our $Overflow_{int}$ that will have the value of:

$$Overflow_{int} = 4,333,333,465 - 4,294,967,296 \Rightarrow Overflow_{int} = 38366169$$

Concluding the mathematical sequence with

$$SCU_{Cycles} = 38366169 \div 75,000 \Rightarrow$$

$$SCU_{Cycles} = 511$$

Overflow adaptiveness test case

As the user insert $period$ equal to $\frac{13}{75{,}000}$ that cause the overflow case,application handled it. Table 3.12 show a the values of every run. Even if the SCU$_{Cycles}$ compared to other tests had tremendous difference the results was satisfying with no losses of packets and send at the right order. After the adaption $period$ was set at 168us.

Table 5.14: Tx results case Overflow

Run / Result	1	2	3	4	M.O.
SCU$_{cycles}$	511	511	511	511	511
Interrupt$_{SCU}$	100	100	100	100	100
Packets Send	100	100	100	100	100
Delay Inside Bus	1,373	1,376	1,278	1,301	1,332
AxiTimer$_0$	13,994	13,991	13,986	13,987	13,989
AxiTimer$_1$	0	0	0	0	0
Adaption Times	2	2	2	2	2
Time(seconds)	0.016312	0.016292	0.016289	0.016291	0.016295

The follow Figures 5.19 and 5.20 show behavioural of our system at the graph for SCU$_{cycles}$ per message and the precise values of the runs.Differences between ever run was minimal that can't be easily distinguish in graph but the exact value can be shown in Figure 5.20 . In this case the adaption of SCU$_{cycles}$ happen twice,both of them from AXI timer$_0$.Because the first time wasn't enough it had to calculate the next optimal value.After inserting the optimal value in SCU timer there at the transmission of the second frame there was no need of any other changes. Even with a delay of calculating and reprogramming the SCU timer frames send with no losses or miss-order.

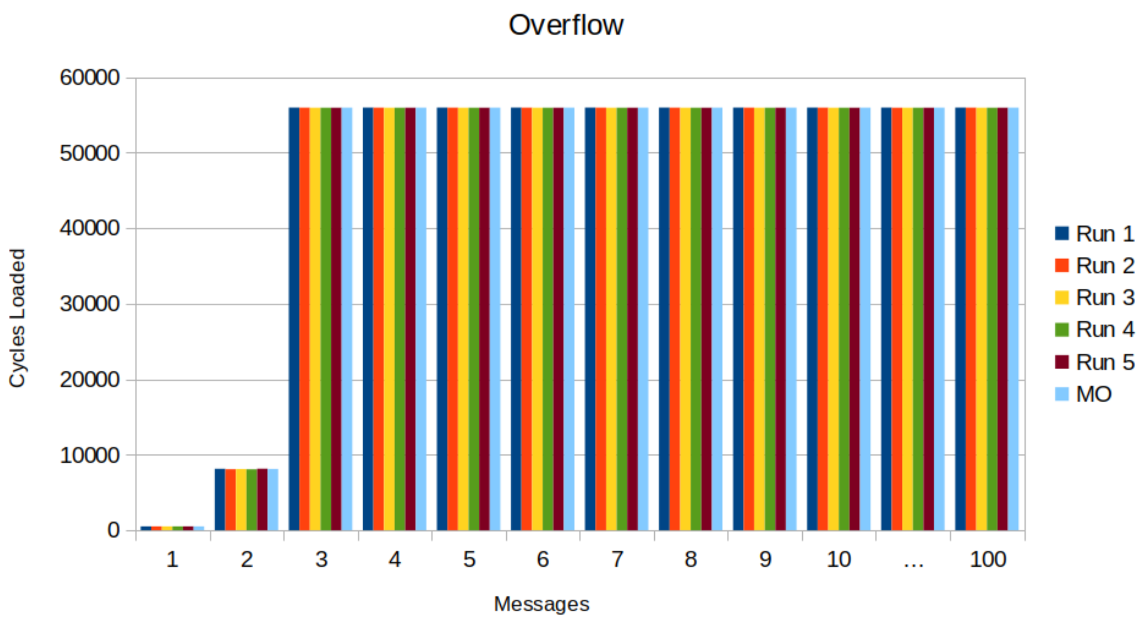
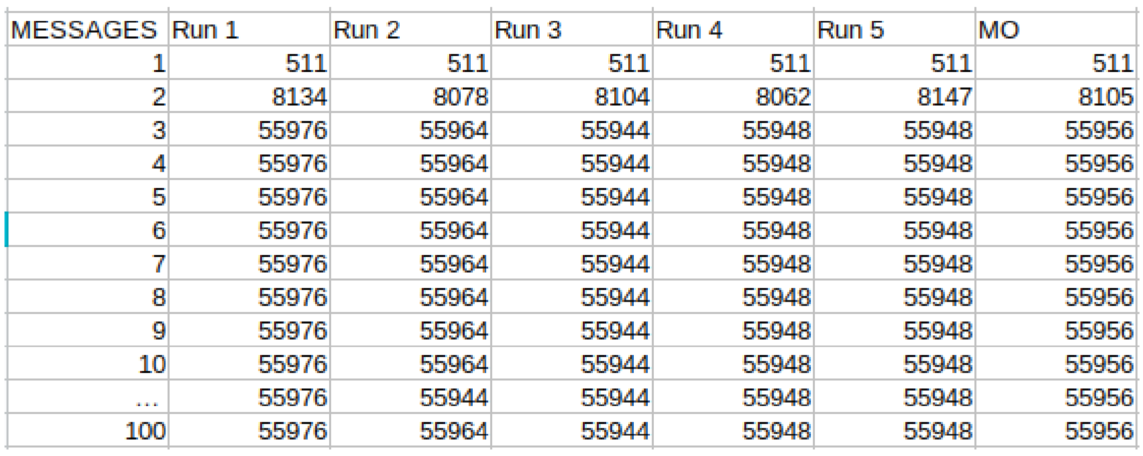


Figure 5.19: Graph SCU$_{cycles}$ per Packet

The specific values in every change of the period was:



MESSAGES	Run 1	Run 2	Run 3	Run 4	Run 5	MO
1	511	511	511	511	511	511
2	8134	8078	8104	8062	8147	8105
3	55976	55964	55944	55948	55948	55956
4	55976	55964	55944	55948	55948	55956
5	55976	55964	55944	55948	55948	55956
6	55976	55964	55944	55948	55948	55956
7	55976	55964	55944	55948	55948	55956
8	55976	55964	55944	55948	55948	55956
9	55976	55964	55944	55948	55948	55956
10	55976	55964	55944	55948	55948	55956
…	55976	55944	55944	55948	55948	55956
100	55976	55964	55944	55948	55948	55956

Figure 5.20: Analytic SCU$_{cycles}$ per Packet

As we already mention AXI timer$_0$ did both changes in the period. The user cant be involved to choose the proper adjustment, however the system by it self will choose the optimal.

As for the Rx results Table certificate that all frames received and in order.

Table 5.15: Rx results

Run / Result	1	2	3	4	MO
Packets Received	100	100	100	100	100
Last Packet Value	99	99	99	99	99
Time(seconds)	0,856550	0,856565	0,856548	0,856577	0,856560

Even if a miscalculation happen by the user, our system was prepared and correspond perfectly to the needs that is programmed without any problems. All the frames arrived at Rx without sacrificing the message integrity of CAN-bus or even a misplaced frame in the order that we wanted. The following flow will show as the blocks of code were enabled.

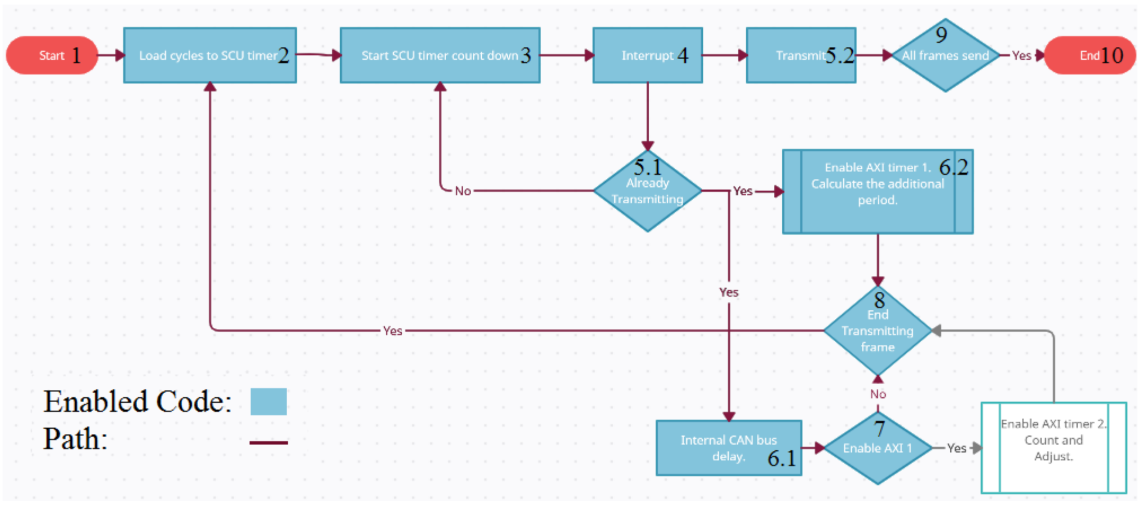

Start 1
Load cycles to SCU timer 2
Start SCU timer count down 3
Interrupt 4
Transmit 5.2
9 All frames send
Yes
End 10
5.1 Already Transmitting
No
Yes
Enable AXI timer 1. Calculate the additional period. 6.2
Yes
Yes
8 End Transmitting frame
No
Internal CAN bus delay. 6.1
7 Enable AXI 1
Yes
Enable AXI timer 2. Count and Adjust.
Enabled Code: 🟦
Path: —

Figure 5.21: Analytic SCU$_{cycles}$ per Packet

And next we will discuss the steps of the process:

1. Start the program
2. Load the calculated value as cycles to SCU timer: 511.
3. The SCU timer start the count down: SCU timer until is equal to 0.
4. When it end, an interrupt occurs.
5. Simultaneously:

    5.1. Check if there is already a frame transmitting.

    5.2. Transmit the frame.

6. Simultaneously:

    6.1. Frame entering the CAN bus default delay.

    6.2. Enable AXI timer$_0$ and adjust the AXI$_{cycles}$ to SCU$_{cycles}$.

7. Check if the Delay is long to enable AXI timer$_1$.

8. The end of the transmission of the last frame and Load the new value of the cycles to SCU timer.

9. Check if the all the frames has been send:100/100.

10. End:Show the results and end the program.

The program start with 511 calculating the period that the user set. And load the overflowed value to SCU timer. The SCU timer expired generating a Interrupt. Next, Step 5.2 the first frame start transmitting and step 5.1 was enabled simultaneously. With no other frame transmitting at the time SCU timer start again the count down and an generate another interrupt. Having this happen in so low period step 6.1 and 6.2 enabled. However, our first frame was send and AXI timer$_0$ did the first adjustment giving the SCU timer a new value and repeat step 3 and step 4. Without time to even end the second transmission an interrupt occurred again and AXI timer$_0$ was enabled to calculate the second adjustment that the period needed. After the second change of the period every message was send to Rx safely and received.

Chapter 6

6.1 Conclusions and Future Extensions

Having thoughtfully evaluate this application working, we resorted that an application that can investigate, adapt, correct itself so can't be easily replaced. This happens because it's an innovative application that can be used in many different systems and will choose the optimal solution based on the system and not on the user. The importance of this, is because a user can insert "wrong" values, however the application can change something "wrong" to the appropriate to work with, with reasonable results that represent the existed system and not a hypothetical scenario inserted by mistake.

Even if its an application that cant be easily replaced, this doesn't mean that can't be upgraded. In future it can take any frequency low and high and find the optimal for the system, without wasting any time making it more robust that already is. We could adjust it in system that support only Standard CAN frame or make it to have the application it self easily swap the option based on our desires.

Bibliography

[1] Steve Corrigan. *Introduction to the Controller Area Network (CAN)*.
`http://www.ti.com/lit/an/sloa101b/sloa101b.pdf`

[2] Hans-Christian Reuss. *Extended Frame Format A New Option of the CAN Protocol*.
`https://home.isr.uc.pt/ rui/str/EXTENDED.pdf.`

[3] Wikipedia. *Controller Area Network (CAN bus)*.
`https://en.wikipedia.org/wiki/CAN`$_{b}us$

[4] Wilfried Voss. *Controller Area Network (CAN Bus) - Message Frame Architecture*.
`https://copperhilltech.com/blog/controller-area-network-can-bus-messa`

[5] Kvaser. *CAN bus messages*.
`https://www.kvaser.com/about-can/the-can-protocol/can-messages-13/`

[6] J. A. Cook. J. S. Freudenberg. *Controller Area Network (CAN bus)*.
`https://www.eecs.umich.edu/courses/eecs461/doc/CAN`$_{n}otes.pdf$

[7] G. Kornaros, O. Tomoutzoglou, D. Mbakoyiannis, N. Karadimitriou, M. Coppola,E. Montanari, I. Deligiannis, G. Gherardi. *Towards Holistic Secure Networking in Connected Vehicles through Securing CAN-bus Communication and Firmware-over-the-Air Updating", Journal of Systems Architecture (2020), vol. 109, pp. 101761*.
`https://doi.org/10.1016/j.sysarc.2020.101761`

[8] G. Kornaros, E. Wozniak, O. Horst, N. Koch, C. Prehofer, A. Rigo, M. Coppola. *Solutions for Cyber-Physical Systems Ubiquity, Editors: N. Druml, A. Genser, A. Krieg, M. Menghin and A. Hoeller, IGI Global book series Advances in Systems Analysis, Software Engineering, and High Performance Computing (ASASEHPC) (ISSN: 2327-3453; eISSN: 2327-3461), IGI Global, Hershey, PA United States, 2018, pp. 301-324*
`https://10.4018/978-1-5225-2845-6.ch012`

[9] G. Kornaros and S. Leivadaros *Securing Dynamic Firmware Updates of Mixed-Critical Applications, 3rd IEEE International Conference on Cybernetics (CYBCONF)*.
`https://10.1109/CYBConf.2017.7985807`

[10] I. Deligiannis, G. Kornaros *Adaptive memory management scheme for MMU-less embedded systems, 11th IEEE Symposium on Industrial Embedded Systems, SIES 2016, Krakow, Poland, May 23-25, 2016, pp. 254-261*
`https://10.1109/SIES.2016.7509439`

[11] C. Prehofer, G. Kornaros, M. Paolino. *TAPPS - Trusted Apps for Open Cyber-Physical Systems., E-Democracy - Citizen Rights in the World of the New Computing Paradigms - 6th International Conference, E-Democracy 2015, Athens, Greece, December 10-11, 2015, Proceedings, pp. 213-216*
`https://10.1007/978-3-319-27164-4`$_1$8

[12] G. Trouli and G. Kornaros. *Automotive Virtual In-sensor Analytics for Securing Vehicular Communication, in IEEE Design and Test, vol.37, issue 3, pp. 91-98, June 2020*
`10.1109/MDAT.2020.2974914`

[13] G. Kornaros, O. Tomoutzoglou and M. Coppola. *Hardware-assisted Security in Electronic Control Units Utilizing One-Time-Programmable Network-on-Chip and Firewalls, IEEE Micro, Volume: 38, Issue: 5, Sep./Oct. 2018, pp. 63-74, 2018*

[14] Davis, Robert I. and Burns, Alan and Bril, Reinder J. and Lukkien, Johan J. *Controller Area Network (CAN) Schedulability Analysis: Refuted, Revisited and Revised, Volume: 35, Issue: 3, April 2007, pp. 239–272, 2007*

[15] Tindell and Hansson and Wellings. *Analysing real-time communications: controller area network (CAN), pp. 259-263*.
`https://doi.org/10.1109/REAL.1994.342710`

[16] Christoforakis, Ioannis and Tomoutzoglou, Othon and Bakoyiannis, Dimitrios and Kornaros, George. *Runtime Adaptation of Embedded Tasks with A-Priori Known Timing Behavior Utilizing On-Line Partner-Core Monitoring and Recovery, pp. 1-8*.
`https://doi.10.1109/EUC.2014.10`

[17] Kornaros, George and Christoforakis, Ioannis and Astrinaki, Maria. *An auto-*

mated infrastructure for real-time monitoring of multi-core Systems-on-Chip, pp. 56-61.
`https://doi.10.1109/DDECS.2012.6219025`