MODEL-BASED DESIGN AND ANALYSIS OF AUTOMOTIVE SYSTEMS USING
TIME-TRIGGERED CONTROLLER AREA NETWORKS

by

RUOSHI ZHANG

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2015
Acknowledgements

I would like to express my sincere gratitude to Professor Taylor Johnson, who very patiently and nicely helped me with my study and research during my two year’s study in University of Texas at Arlington.

For this project I also would to highlight Randy Long, a very diligent and talented student who created the whole control system of the E15 car, I have learnt a lot of things from him and his program.

The last but not the least, I am very appreciate the help from my family, because of your support that I could have this honored degree today.

April 9, 2015
Abstract

MODEL-BASED DESIGN AND ANALYSIS OF AUTOMOTIVE SYSTEMS USING
TIME-TRIGGERED CONTROLLER AREA NETWORKS

Ruoshi Zhang, M.S

The University of Texas at Arlington, 2015

Supervising Professor: Taylor Johnson

As the numbers of electrical and electronics (E/E) components vastly increases in modern motor vehicles, vehicle safety risks and recalls due to E/E problems have become issues for both auto manufactures and customers. This thesis describes methods to assist designers to analyze legacy designs incorporating the Time-Triggered Controller Area Network (TTCAN) bus in a post-design phase. We present a case study for the UT-Arlington Formula SAE all-electric race car, which has 18 controller area network (CAN) PIC microcontroller nodes using the TTCAN protocol. The contributions include the general approach to re-using legacy software in MBD processes, the automotive case study and analysis thereof with SLSF simulation, and resulted in identifying and correcting two bugs in the legacy software for the car.
Table of Contents

Acknowledgements ........................................................................................................ iii

Abstract ......................................................................................................................... iv

Table of Contents .......................................................................................................... v

List of Illustrations ........................................................................................................ vii

Chapter 1 Introduction ................................................................................................... 1

Chapter 2 CAN Bus Basics and Time-Triggered CAN Bus ............................................ 5
  2.1 CAN Basics ........................................................................................................... 5
  2.2 Time-Triggered CAN .......................................................................................... 9

Chapter 3 Modeling and Simulation ............................................................................. 13
  3.1 Controller Implementation .................................................................................. 13
  3.2 Schedule Analysis .............................................................................................. 18
  3.3 Code Analysis ..................................................................................................... 19
    3.3.1 ACM master controller side ......................................................................... 19
    3.3.2 ACW wing controller side ........................................................................... 22
  3.4 Simulation Environment Setup .......................................................................... 26
    3.4.1 Mex Function Implementation .................................................................... 26
    3.4.2 Source Code implementation in Stateflow Charts .................................... 28

Chapter 4 Simulation and Experimental Results ....................................................... 35
  4.1 Simulation Results .............................................................................................. 35
  4.2 Bug Detection ...................................................................................................... 39
  4.3 Experimental Results ......................................................................................... 41

Chapter 5 Conclusion .................................................................................................... 44
  5.1 Conclusions ......................................................................................................... 44
  5.2 Future Work ........................................................................................................ 44
List of Illustrations

Figure 1-1 E/E Notification Trending [2]................................................................. 1
Figure 2-1 Recessive and Dominant Signal [9] ......................................................... 5
Figure 2-2 Typical CAN Frame [10] ...................................................................... 6
Figure 2-3 Standard CAN Frame [11] .................................................................... 7
Figure 2-4 Extended CAN Frame [11] .................................................................... 7
Figure 2-5 Actual CAN Bus Traffic [11] ................................................................. 8
Figure 2-6 an Example of TTCAN Schedule/Matrix [12] .................................... 10
Figure 3-1 Communication System Topology in the UTA FSAE E15 Car ............ 14
Figure 3-2 Wing Layout ...................................................................................... 15
Figure 3-3 ACM Routines and Program Flow Diagram ...................................... 16
Figure 3-4 ACM Routines and Program Flow Diagram ...................................... 17
Figure 3-5 Time Matrix Used in E15 .................................................................. 18
Figure 3-6 Data Flow Illustration Receive Process ............................................. 20
Figure 3-7 Data Flow Illustration of Transmission Process .............................. 21
Figure 3-8 Bit Calculation for A Message ............................................................ 25
Figure 3-9 Including Source and Header Files .................................................... 31
Figure 3-10 ACM CAN 1 Block ......................................................................... 32
Figure 3-11 Delay Block Inserted ....................................................................... 33
Figure 3-12 Hierarchal Structure of the Model ..................................................... 34
Figure 4-1 ACM (0x460) Timer 1 Block Output ................................................... 36
Figure 4-2 ACW (0x462) Timer 1 Block Output .................................................. 37
Figure 4-3 ACM and ACW Outputs Together .................................................... 38
Figure 4-4 ACM Timer 3 Routine Heartbeat Counter Output ............................ 39
Figure 4-5 Failure of Communication Due to the Flag Not Been Set ................. 41
Figure 4-6 Synchronization Issue ................................................................. 42
Chapter 1

Introduction

Compared with their earlier predecessors, modern motor vehicles have greatly improved, because they are now equipped with more sophisticated sensors, microcontrollers and actuators, resulting in better fuel economy, improved safety, among other improvements. A statistical analysis of recent research suggests that, in the United States and Canada from 1970s to 2010s, the number of electrical and electronics (E/E) component usage is firmly increasing, and today there are on average 50 to 75 ECUs distributed in one motor vehicle. Back in 1970s average amount dollar spent on electrical components was $110, in 2001 it increase to $1,800 [1]. Due to electrical problems the vehicle notification, also followed this trend as shown in Figure 1-1 [2]. Only in 2014, there were 52 million cars and trucks been recalled [3] in United States. In order to assist manufacturers decrease electrical components failure, this research uses a FSAE racing car’s control program as an example, focus on the analysis of its TTCAN communication system by Simulink/Stateflow (SLSF).

As its name implies, controller area network bus, known as the CAN bus, is originally developed by the German engineering and electronics company Bosch to dealt with the design issue caused by increasing number of electrical sensor/actuators integrated
into modern cars, which is the result of vehicle designers would like to bring a more reliable and precise driving experience to their customers [4]. While at that time current inter-chip communication schemes like IIC or even parallel are less electronically stable and are not very extendable on larger scale, or they may cause huge number of wires existing in the vehicle. Since been introduced into motor vehicle manufacture in 1991, CAN bus has gained its dominance not only in car designs but also been implemented onto many different engineering and scientific aspects. It supports/ connects every components in a vehicle from anti-lock braking system (ABS), engine throttle control and cruise control, etc. [5].

Early CAN scheme is event triggered. Which means every node on the bus could transmit a message at any time when they have anything to transmit, for instance the driver turns on the window brush or a passenger lower the temperature of the AC. The worst case scenario would be all nodes transmitting message simultaneously and cause very severe bus latency. Under certain circumstance the latency could lead to dangerous situation. On the other hand in some special applications when the updating rate of every component is crucial, designers would like to see more guarantee on real time performance: such as electrical device heavily depended formula racing cars is a good example.

As a solution, time-triggered CAN scheme come onto the stage. It based on standardized CAN bus physical layer and data link layer while implement a higher level protocol on these two layers. It reorganizes random transmission into prearranged as well as predefined communication – messages been allocated into windows and cycles, each cycle recalibrate timing and every node has its own chance to speak on the bus. This paper discovers a method to analysis the robustness of this scheme by using a FSAE racing car as an example, while the performance of TTCAN scheme have proved and not discussing in this paper [6].
Simulink/Stateflow (SLSF) provides a powerful simulation environment including numerical simulation, C and Fortran code compilation, Hardware-in-the-Loop Simulation [7] and even embedded system code generation. In our application we focus onto simulate the behavior of C code. Although lower level implementations that done on micro-controller cannot or even difficult to be introduced into this environment, but since we focus on the higher level implementation in application layer, the advantage of this powerful tool can be fully utilized. Since this communication scheme is based on time triggering, all core functioning codes hook up with timers or interrupts. This structure therefore decide the structure of simulation model: every single timer/interrupt subroutine comprise a solid Stateflow chart; an unified clock source provides universal timing information to each chart; they share information by global variables within C code or designated signal bus. One consideration here needed to be noticed is that under real scenario all timers/interrupts are entered sequentially rather than parallelly, but if we do not take care of this point all Stateflow charts could executing at the same time, which could be an ill analogy to the real setup and leading to false result.

From a higher level of view, in this thesis, we present a case study of a design method to integrate legacy embedded software into modern model-based design (MBD) processes. Systems designed today typically use tools like Simulink/Stateflow (SLSF) for design, testing, and even implementation when using code generation tools for target embedded hardware platforms. Legacy systems were not frequently designed with an MBD process and relied on embedded software engineers writing most of the embedded software manually. Our case study uses the car designed by the UT-Arlington Formula Society of Automotive Engineering (FSAE) team for the 2015 SAE collegiate electric car competition. The car is named E15 as it is an electric car for the 2015 competition. The designers of this car have developing this full electrical car for years and recently the most
parts and electrical control system have its shape. By this chance we gained real field experience with real motor vehicle and contribute their design by located two bugs.

Generally speaking, the analysis method can be implemented onto any high level analysis/testing of embedded system programming; single thread programs could use a single Stateflow chart structure or a series of Stateflow chart in a series manner; timer-triggered or operating system controlled multi-thread programs could also use the solution discussed in this paper.
Chapter 2

CAN Bus Basics and Time-Triggered CAN Bus

2.1 CAN Basics

Controller Area Network (CAN) bus now is the most popular communication protocol applied in multiple technical areas, which includes motor vehicle, aerospace and even robotics. As the standard introduced by its creator BOSCH Company in 1991, version 2.0 specifies the two fundamental layers of CAN bus [8]: The data link layer and the physical layer.

From low to high sequence, the physical layer defines the transmitting media and part of electrical properties. If do not consider power lines, a typical CAN network consist two wires that conveys a pair of differential signal. Under normal understanding of electrical signal concept, a logical one should corresponding to a high voltage and a low voltage would represent logical zero, while CAN has different rule: A logical one is named as “recessive” bit which uses 2.5V (typical value) on both wire and a logical zero is called “dominant” which deploys 1.5V and 3.5V on CANH and CANL wire accordingly. This layer corresponding to the physical layer of ISO/OSI model.

![Recessive and Dominant Signal](image)

Figure 2-1 Recessive and Dominant Signal [9]

The data link layer has two sublayers: MAC – medium access control sublayer and LLC - logical link control sublayer. LLC sublayer mainly handles package filtering, overload
notification and recovery management [8]. According to CAN standard from BOSCH, the MAC sublayer represent the core definition of the protocol, which defines message frame format, frame type, acknowledgement and error detection information.

<table>
<thead>
<tr>
<th>SOF</th>
<th>ARBITRATION</th>
<th>CONTROL</th>
<th>DATA</th>
<th>CRC</th>
<th>ACK</th>
<th>EOF</th>
</tr>
</thead>
</table>

Figure 2-2 Typical CAN Frame [10]

Figure 2-2 Typical CAN Frame is an illustration of a typical CAN frame. Each message starts with a “SOF – start-of-frame” bit. When a node initiates a SOF on the bus, other nodes become quiet and start listen to the message. Depending on whether it is a standard ID frame or extended ID frame, the arbitration field holds 12 or 31 bits correspondingly. For a standard ID frame, this field contains an 11-bit identifier and a 1-bit RTR signal, and for an extended ID frame it holds an 11-bit standard and an 18-bit extended ID field and a SRR and an IDE field in between. Since CAN bus is a broadcasting system [11], once a node puts a message onto the bus, it is accessible to all other nodes. Naturally, this character requires each node has a filter in front of their receiving buffer so that they could only receive what they interested. In additional, combined with bus electrical states (recessive and dominate), the bus itself could serve as a judge – when two nodes wish to send message simultaneously, the node with small ID number will win the arbitration. Consider node 1 sending a message with ID = 1 and node 2 sending a message with ID = 2. The 4 digits binary presentation of 1 is 0001 and for 2 is 0011, count from right, the second digit for 1 is 0 while for 2 is 1. This will lead to a dominate state (0) versus recessive state (1), as a result dominate will win this arbitration and eventually the message with ID = 1 will win the bus and the lose side will keep quiet and listen until it wins arbitration.
or the bus is free. Figure 2-3 shows a standard CAN frame and Figure 2-4 shows an extended CAN frame.

<table>
<thead>
<tr>
<th>SOF</th>
<th>11-bit Identifier</th>
<th>RTR</th>
<th>IDE</th>
<th>r0</th>
<th>DLC</th>
<th>0...8 Bytes Data</th>
<th>CRC</th>
<th>ACK</th>
<th>EOF</th>
<th>IFS</th>
</tr>
</thead>
</table>

Figure 2-3 Standard CAN Frame [11]

<table>
<thead>
<tr>
<th>SOF</th>
<th>11-bit Identifier</th>
<th>SR R</th>
<th>IDE</th>
<th>18-bit Identifier</th>
<th>RTR</th>
<th>r1</th>
<th>r0</th>
<th>DLC</th>
<th>0...8 Bytes Data</th>
<th>CRC</th>
<th>ACK</th>
<th>EOF</th>
<th>IFS</th>
</tr>
</thead>
</table>

Figure 2-4 Extended CAN Frame [11]

Right after 12th bit, there is a RTR or SRR bit in either frame. This bit is related to remote frame function. The 14th IDE bit for both frame is identical, each node uses this bit to identify the frame type: 1 means extended frame and 0 means standard frame. IDE, R0 and DLC together makes up a control field for standard frame and R1, R0 and DLC makes up extended frame's control field. As their name suggested, R0 and R1 are reserved bits and they both keep dominant state. Data length code (DLC) field takes 4 bits to record how many bytes will this message contain right after this field, maxim value is 8. A 16-bit CRC field followed Data field to provide checksum information.

A complete CAN communication requires at least two parties join a conversation, or one sender with multiple listeners. When the sender transmitting everything just before the ACK bit, other interested listeners are keep listening; right after last bit of CRC, the sender sends the ACK bit, a recessive bit, on the bus and it expecting someone happy with what it just said and reply an ACK on the bus, which is a logic 0 (dominate). One of the listener will reply correspondingly while the sender is still sending this bit, so on the bus this bit is been overwitten to dominate. Sender accept it and finally give a 7-bit long EOF (end of frame) to indicating finish of communication.
Figure 2-5 shows the data stream observed from an oscilloscope of actual CAN communication. In this diagram node A sends data first, when it is done node B and C give first pair of ACK bit; after a period of silence (EOF + inter-frame space, the IFS) node B and node C initiate transmission and node C wins arbitration; When node C done transmitting node A and B acknowledge node C they have successfully received data by an ACK bit; eventually node B got its chance to send and its listeners acknowledge receive.

Above is a rough discussion of how CAN standard specifies its physical layer, on top of it comes data link layer. Data link layer including functions such as message filtering, overload notification, data encapsulation/encapsulation and so on [9]. These processes including physical layer signal generation are all covered in the CAN module on MCUs, and in this analysis we will cover them in customized code.

From above discussion we notice that messages are prioritized by their ID number, hence this arrangement gives no guarantee on message response time to messages with low priority. When a situation requires that each message should update within a fixed period so that the ECU could plan its next control decision accordingly, event-triggered
scheme would considered as not very plausible. With this design requirement presented, people developed a new session layer above previous standard, this is the time-triggered CAN. In this new layer, instead of using predefined ID for priority and dynamic arbitration process, the new layer defines a static schedule and assign a position to each node on the bus. As a result the communication is unified onto this schedule and response time could be bounded.

2.2 Time-Triggered CAN

Time-Triggered Controller Area Network (TTCAN) is a revised version of normal CAN protocol. TDMA (time division multiple access) scheme is widely used in telecommunication systems: on a specific frequency it divides time into message slots and assign them to different parties. Similarly, a universal time schedule is introduced into TTCAN system above CAN specification. It defines a relative time among all nodes that have to follow and definite action executes accordingly; Instead of each node’s priority been decided by ID field, all nodes are separated into two kinds: the master who initiates each cycle by transmitting reference message and the slaves who synchronize with the master by receiving the reference message. So the new schedule looks like this: the designer creates a universal matrix with several lines and rows; each line is a basic cycle and it has a fixed number of slots, these slots are called windows. Every window is assigned with a specific message from a specific node with a specific meaning. When one basic cycle could not include all messages the designer wishes to send, a new basic cycle has to be created and repeat the process until every node has its place. The complete schedule is called a Matrix Cycle. So the new idea creates another question: how does each node synchronize? The answer is the reference message in the beginning of each cycle. In our project, the reference message carries cycle information as its data, every
listening node start their local timer and set received cycle information right after receives this message. Figure 2-6 shows an example schedule.

Figure 2-6 an Example of TTCAN Schedule/Matrix [12]

Every time the monitor or master ECU initiates a new cycle, a reference message is received by all other nodes like one of a wing controller in our project or a sensor node; they will synchronize their local time with the master. For example in our project it is done by start a local timer and this timer will transmitting messages by the same period with the schedule, after the last window is transmitted it will turn itself off.

Compared with old event-triggered standard, the time-triggered CAN scheme has its own advantages and drawbacks. From system integrity view, since the windows and messages are predefined, the action hooked upon each message is decoupled from bus load [13]. Hence a communication scheme simulation before implementation is possible and each components on the bus could be developed individually. On the other hand this feature will cause the system lack of flexibility; before deeper design into each component
the designer has to come up with the schedule, under simple situation this drawback could be relatively easy to handle with while as the complexity goes higher such as a full functioning motor vehicle this drawback no wonder will be difficult to deal with. Another drawback will be the synchronization issue; a bad schedule planning may assign too much or too less windows onto one basic cycle, if too much the system may lose synchronization eventually due to clock shift on different micro-controller; if too less the percentage of synchronization messages would be too many and cause low system throughput. A lot of researches have been done to tackle this issue, researchers proposed several methods to assist/optimize window distribution within a cycle, such as AL (Average-Loading) algorithm. Based on this algorithm researchers evaluated the actual performance of TTCAN and draws conclusion that the probability of certain message exceeding their deadline is extremely low [6]. Also in research carried out by Leen and Heffernan, the deterministic behavior and safety property of time triggered CAN protocol have been formally verified by UPPAAL model checker based tool set [14] [15].

Compared with event-triggered structure, TTCAN gains an edge that the master/monitor MCU on the bus could easily detect if one node is fail or malfunctioning. When TTCAN implemented for instance, the supervisor node (ECU) just finish transmitting then turns to listening mode and expecting one of its sensor controller feedback a temperature reading of the engine, but this did not happen within a safe period of time. Now that the ECU should fire a temperature sensor fail alarm to notify the driver there could be an engine overheat, further more slowly down the vehicle or respond accordingly. While in an event-triggered structure, when this sensor for some reason malfunctioning and lost response, the ECU may never know it's gone eventually lead the engine blows. The event-triggered CAN communication scheme is suitable for relatively low bus load situations, even 50% utilization factor is rarely exceeded between non-crucial parts
and even lower for critical parts such as engine management, which rarely exceed 20~30% [12]. This relative low utilization rate is necessary since enough space have to made in case of transmission fail and avoid delay caused by message collision. On the contrary TTCAN propose a promising solution for this problem [12]. A final feature of TTCAN is its security. This predefined matrix is a close loop structure, one component changes every other has to change. Under certain circumstance this behavior may protect unwanted bus access.
Chapter 3
Modeling and Simulation

3.1 Controller Implementation

This E15 car is built in 2014 and the designers are still working on its design, analysis, fabrication and testing. E15 equipped with 6 AC battery packs, each wheel is driven by a dedicated motor, along with the ECUs these comprise the main power chain of the car. Compared with traditional car power chain with single engine and transmission mechanism this arrangement will give designers both a control system design challenge and a much more flex room to improve performance by implementing precise control of each wheel. The most parts of the race car have already realized electrical control: for example the peddle box is measured by two limit switches instead of mechanical structure, this arrangement could help lighten the car body weight. Every micro-controller equipped on the car are from Microchip dsPIC33EP series, these MCUs have 16-bits Modified Harvard Architecture CPU core with 70 MIPS speed, compatible with multiple motor control applications, while provides a high energy efficient and extended life expectancy. All these advantages made it ideal choice for the design.

Figure 3-1 demonstrates a layout of the electrical and control structure of the car. For safety consideration, every component did not connect to a same bus (which will lead to a complicated TTCAN schedule), instead they use a hierarchy structure: battery packs respond to their own bus which is referred to Battery on the graph; motors and pedal box talk on a same bus, which is referred to as Tractive on the graph. In the middle of the graph “D” representing the dash board controller and “S” representing the supervisor controller, together with batteries and motors they comprise the backbone of the control system; wings and their controller comprise a single aerodynamic system, functionally they have
nothing to do with the power chain so they are separated from this critical zone and hooked upon to the whole system through the Tractive bus.

![Communication System Topology in the UTA FSAE E15 Car](image)

The main subsystem describe in this paper is an active aerodynamic control system used to produce downforce with spoilers (wings) when turning at high-speeds (these are race cars). The active aerodynamic control system is fully measured and controlled by its own ECUs and accelerometer sensors. There are totally 8 wing blades been installed onto the vehicle, according to their positions can be divided into four
quadrants: left front (LF), right front (RF), left rear (LR) and right rear (RR). Each quadrant again owns a upper wing blade and a lower wing blade as shown in Figure 3-2: arrow 1 points to right rear upper and arrow 4 points to left rear lower and so on so forth. Under each wing blade, two parallel connected limit switches are installed and report wing blade position to corresponding ACW controller: when the wing blade is closed it will rest on the switch and when they are open the switches will be released. This feeds back the actual position of the wing blades to the ACM controller for the motor position command, which is essentially a PWM signal that is transmitted back to the ACW controllers over the CAN bus. If the command does not match with feedback, an alert should fire to the driver.

![Figure 3-2 Wing Layout](image)

Aero Control Master (ACM) is the MCU takes responsibility of: (CAN 1 routine) receive all intended messages, (AD1 routine) read the G-Force from the accelerometer in three-axis, (Timer 4 routine) convert the acceleration reading into corresponding PWM setting for servos, (Timer 3 routine) process received messages, prepare messages to transmit, and track ACW controllers’ states with heartbeats, and (Timer 1 routine) transmit
converted data to corresponding ACW controllers and other information to supervisor node. Figure 3-3 demonstrates the program state flow of ACM side. In this figure, tasks are separate by interrupt subroutines, each one have a single or a series of states. For example in Timer3 routine it has multiple states while in CAN1 subroutine the program only responsible for receiving messages.

Figure 3-3 ACM Routines and Program Flow Diagram

Aero Control Wing (ACW) controllers are the four “slaves” of the ACM controller, they only receive messages from ACM and respond to ACM. Their tasks include: (CAN 1 routine) receiving heartbeats and PWM register values from ACM, (Timer 3 routine) update the wing blade switch information and drive the servo motors, and (Timer 1 routine) transmit wing blade limit switch information back to ACM. Figure 3-4 demonstrates the
program state flow of ACW side. ACW side program owns very similar structure to that ACM side, all data process are handled in the timer 3 routine, and other interrupt subroutine only responsible for a single task, while there is no ADC routine and related timer 4 routine since there is no ADC attached onto it.

Figure 3-4 ACM Routines and Program Flow Diagram
3.2 Schedule Analysis

Before dig into the program, we should examine the communication matrix first.

<table>
<thead>
<tr>
<th>c/w</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x460: aAcmSycAer 0x3200</td>
<td>0x460: aAcmDatAer 0x1A00</td>
<td>0x460: aAcmSnvAer 0x1C08</td>
<td>0x460: aAcmSnvAer 0x1C0A</td>
<td>0x462: aAw0HbtAer 0x464</td>
<td>0x464: aAw1HbtAer 0x464</td>
<td>0x462: aAw0LswAer 0x1E00</td>
<td>0x462: aAw1LswAer 0x1E00</td>
<td>0x466: aAw0LswAer 0x1E04</td>
<td>0x464: aAw3LswAer 0x1E06</td>
</tr>
<tr>
<td>1</td>
<td>0x460: aAcmSycAer 0x3200</td>
<td>0x460: aAcmDatAer 0x1C00</td>
<td>0x460: aAcmSnvAer 0x1C08</td>
<td>0x460: aAcmSnvAer 0x1C0A</td>
<td>0x466: aAw2HbtAer 0x466</td>
<td>0x468: aAw3HbtAer 0x468</td>
<td>0x462: aAw0LswAer 0x1E00</td>
<td>0x462: aAw1LswAer 0x1E00</td>
<td>0x464: aAw2LswAer 0x1E04</td>
<td>0x468: aAw3LswAer 0x1E06</td>
</tr>
<tr>
<td>2</td>
<td>0x460: aAcmSycAer 0x3200</td>
<td>0x460: aAcmDatAer 0x1C02</td>
<td>0x460: aAcmSnvAer 0x1C08</td>
<td>0x460: aAcmSnvAer 0x1C0A</td>
<td>0x460: aAcmHbtAer 0x460</td>
<td>Reserved</td>
<td>0x462: aAw0LswAer 0x1E00</td>
<td>0x464: aAw1LswAer 0x1E02</td>
<td>0x466: aAw0LswAer 0x1E04</td>
<td>0x468: aAw3LswAer 0x1E06</td>
</tr>
</tbody>
</table>

Figure 3-5 Time Matrix Used in E15

Figure 3-5 is a demonstration of the schedule: the hexadecimal number atop within each window indicates the address of each device/node on the bus. For instance, 0x460 represents ACM controller; 0x462 and 0x464 represent left and right front wing accordingly; 0x466 and 0x468 together represent the left rear wing and right rear wing, since they move together their content should keep same. In this study we focus on the behavior of three nodes: the ACM (blue parts), the left front wing (orange part) and the right rear wing (green part).

The first window of each cycle is the reference message with ID equals to 0x3200 and it carries current cycle number. The second window carries data needed to transfer to the supervisor. The third and fourth windows with ID equals to 0x1c08 and 0x1c0a will transmit servo position information to corresponding ACW controllers. From the fifth and sixth window, every node will broadcast a heartbeat message with ID equals to their node number. From seventh to tenth window, each ACW controller will report its own limit switch information.

In this TTCAN schedule there are three types of messages: 1. Reference (synchronization) message from master controller, as long as all wing controllers receives this message they will start their local timer and use the current cycle number encapsulated
in data field of this message to keep track of current cycle; 2. In order to make sure every wing controller on the bus is properly functioning, we introduce a heartbeat message in the schedule. On ACM controller there are 4 counters corresponding to each wing controller counting down at each cycle from initial value of 50. Within each cycle ACM count down by 1 from each of the four counters no matter receive heartbeat or not; if received, initial value 50 will be put back to this counter; 3. Data messages, they carry the servo position and every other information.

3.3 Code Analysis

For maintainability purpose, all codes for all nodes on the bus have included into the same project, all the programs for different nodes are logically sorted in the same project although all source files are gathered in a same directory. If one particular software need to be compiled, the corresponding compile option have to be selected from “Configuration” menu from Toolbars, by doing this will activate background scripts to define preprocessor to match the selected software.

Basic software structure within a particular program for one node in a manner that all tasks hook up with interrupts. This structure is a bit like an operating system but much more simpler implemented on functionality-wise and structure-wise. Hardware and interrupt subroutine initialization come first and then start each timer to start each task. Since in this study is focus on the analysis of communication scheme and due to the limitation of simulation software platform, hardware initialization part of the code is not cover in the research.

3.3.1 ACM Master Controller Side

ADC interrupt routine: the ADC module accepts input from a 3-axis digital accelerometer ADXL345 and battery voltage. One input reads battery voltage and three other inputs read acceleration value from x-axis, y-axis and z-axis. Every iteration ADC
program stores its raw reading to structure stAcmSensor, then the acceleration data are converted and stored into corresponding structure member through Timer 4 and Timer 3 routine. For simplicity consideration the ADC data processing task is assigned to ACM side rather than transmit raw ADC data to each ACW controller then decide wing angle locally, this arrangement reduces redundant processing work and gain maintainability.

CAN module interrupt subroutine: in our project, only CAN 1 module is activated. Before iteration starts the program will initial a receiving table called aCanTableRx, and it only receive specific messages. In each iteration of this interrupt, the program check the main receiving buffer interrupt flag bit then down to each RXFUL (receive buffer full) bits to check which receiving buffer is full. The content of this buffer will be translated and stored into an intermediate CAN data structure stMessage, which contains every detail of a CAN message: message ID, ID type, data, data length, filter information and message processing status. According to the ID information, this message will be allocated into the final receiving table: aCanTableRx. Data flow of CAN receiving is shown in the following Figure 3-6:

![Data Flow Illustration Receive Process](image)

Figure 3-6 Data Flow Illustration Receive Process

For this interrupt, we store heartbeat messages with ID 0x460, 0x462, 0x464, 0x468; wing limit switches with ID 0x1e00, 0x1e02, 0x1e04 and 0x1e06; reference messages with ID 0x3200; shutdown broadcast from supervisor with ID 0x200 and other messages come from Tractive bus (will not discuss in this paper).
Timer 1 interrupt routine: timer 1 routine takes responsibility for CAN message transmission with period of 1ms. On both ACM side and ACW side, timer 1 routine must keep synchronized with the schedule. During each iteration, according to current cycle and window information the timer 1 routine fetch a new message from transmission table aCanTableTx, which contains following messages: heartbeat message of ACM itself; active aero control sensor reading of x,y,z axis acceleration data; servo range; PWM settings for servo position; active aero system state information and a shutdown message. Especially if this message is the reference frame, current cycle number will be put in its data field, after that it will be packed and sent. This process is just the reverse of receiving.

![Data Flow Diagram](image)

**Figure 3-7 Data Flow Illustration of Transmission Process**

Timer 3 routine is responsible for CAN messages processing with a 10ms time interval, as Figure 3-3 shows it has the following states: 1. clears sample enable bit of ADC, which enables ADC hardware to initiate a data conversion; 2. then the program updates the dash board switch reading; 3. CAN messages processing begins from here, it fetches every CAN messages from the receiving table (aCanTableRx) by message ID which contains raw data received from other nodes, then process and eventually store them into corresponding data structure or update heartbeat counter if heartbeat is received; 4. Pack transmitting table (aCanTableTx) with messages needed by other nodes from local data structures, then Timer 1 routine could transmit them later; 5. Decrement heartbeat counter.

On ACM side there are 4 heartbeat counters for each of the ACW controllers, every ACW
controller have one cell in the time matrix for their heartbeat message. Once a heartbeat message is received by ACM, it will reset the corresponding counter to initial value (50 for this case) in state 3. Besides, all counters have to decrement 1 in Timer 3 routine at the very end of it. If one counter reaches zero the program would assume that node is malfunction and the driver should be noticed.

Timer 4 routine has only one purpose, to calculate duty cycle of PWM for servos that controls the wing angle. As we know when the vehicle is driving in high speed due to aerodynamic principle the body may have the tendency to “floating” on the track, leading to less grip between tire and ground. While on the other hand designers wish the vehicle would tightly grip the ground – more than on the straight – when the car enters a corner in order to avoid drift and even out of control. This requires the wings could change their angle to provide different downforce accordingly. For straight track, we would like to have relatively small down force so that the car could speed up in short time. As a result the wings have set to almost parallel to the body of the car. While in corner we would like to see higher down force so that the car could be tightly pressed onto the ground so the wings are almost closed (straight to the ground) to provide down force. At this moment only these two angle settings are implemented.

Next we give a description of all these interrupt routines. The ADC routine detects acceleration changes during running and processes this data into G force. The Timer 4 routine calculates duty cycle of PWM signal for the servos according to the G force. The Timer 3 routine updates in-coming messages and packs generated duty cycle information and prepare for transmission. The Timer 1 routine eventually puts all messages on the bus.

3.3.2 ACW Wing Controller Side

Basic idea of wing controller is similar to master controller side. In CAN 1 subroutine, only interested messages are received instead of all messages sent by master
controller. For instance, in right rear wing controller, we only receive these messages: its own heartbeat (0x468), rear wing servo position PWM signal (0x1c0a), synchronization (0x3200), ACM heartbeat (0x460) and shutdown (0x200). Every time CAN 1 module receives a synchronization message it will pass the current cycle number from data field to the local variable which keeps track of cycle and then start timer 1.

Same with ACM side, all data process such as unpack servo position signal and update heartbeat counter happens in Timer 3 routine, as shown in Figure 3-4, one different is that when ACM side packing the PWM cycle duty signal into message 0x1c08 and 0x1c0a, program pass target position into “iPosCurrent” member of servo structure then feed into data field of the message; when ACW side unpacking the program did the reverse process that fetch the data from message then store them into position member of Timer 3’s sensor structure and eventually move the servo. Following code snippet demonstrate this process: From the Timer 4 routine in ACM:

```c
void servo_mwSetPosition(ServoData * stServoData, uint16_t iPosition)
{
    ...
    stServoData->iPosTarget = iPosition;
    ...
}
```

Then in the Timer 3 routine in ACM:

```c
void acmCanOutputServoLimits(void)
{
    ...
    stMessage = ctGetMessageById(CanTable_Transmit, 0x1c08);
    stMessage->bWriting = 1;
    stMessage->aData[0] = (uint8_t)(aAcmServo[ACM_SERVO_LFL].iPosCurrent >> 8);
    stMessage->aData[1] = (uint8_t)aAcmServo[ACM_SERVO_LFL].iPosCurrent;
```
On ACW side, to unpack from the Timer 3 routine:

```c
void acwCanInputServoLF_1(void)
{
    uint16_t iPosLower = 0;
    stMessage = ctGetMessageById_acw_1(CanTable_Receive, 0x1c08);
    if(stMessage->bPending == 1 && stMessage->bWriting == 0)
    {
        stMessage->bReading = 1;
        // Unpack data for the lower servo.
        iPosLower = stMessage->aData[0];
        iPosLower <<= 8;
        iPosLower |= stMessage->aData[1];
        iPosLower &= 0xfff;
    }
}
```

Another task of Timer 3 routine is to update switch position information to ACM controller, this task is the feedback of receiving servo position command: under each wing there are two independent switches parallelly connected to ACW controller indicating the condition of the wings, when the wing is at fully closed position it will rest on the switch so the switch will set high voltage signal to the corresponding bit to member `iSwLimit` of `stAcwSensor` structure. This information will be transmitted back to ACM controller each transmission time and inform the driver each wing condition.

Timer 1 routine on ACW controllers responsible for transmitting wing switch information and heartbeat to ACM controller, the program only initiate message 0x462, 0x0x464, 0x466,0x468 as heartbeat messages according to different wing controller, and 0x1e00, 0x1e02, 0x1e04 and 0x1e06 as switch limit information according to their position.
One thing need to mention here is that the consideration of synchronization issue between ACM side and ACW side, because CAN bus is a serial bus and serial transmission takes a period of time to transmit on the bus, this would lead to a time mismatch between ACM and ACW: ACM sends reference message on the bus and after a period of time ACW receives it then start Timer 1 on ACW controller. This transmission time will cause a time delay between transmitter and receiver. To prevent this problem from happening, compensation have been made. The time needed for reference message is given by following calculation:

From physical layer specification that CAN message is NRZ (non-return zero) code, so the baud rate is simply equals to bit rate. A reference message has extended ID and 1 byte in data field so the total length of synchronization message can be calculated according to Figure 3-8.

![Figure 3-8 Bit Calculation for A Message](image)

Total number of bits in a message: 1 (SOF) + 29 (ID) + 1 (SRR) + 1 (IDE) + 1 (RTR) + 1 (RB1) + 1 (RB0) + 4 (DLC) + 8 (DATA) + 16 (CRC) + 2 (ACK) + 7 (EOF) + 3 (IFS) = 75bits.

The baud rate is 250kbps which is equals to bit rate, so the reference message takes: 75 bits / 250,000 bit/second = 0.3 ms to transmit. As a result whenever the
acwTimer1Start() function been called TMR1 register always been loaded with 1200 to compensate this time difference.

3.4 Simulation Environment Setup

3.4.1 Mex Function Implementation

Executing C program in Matlab environment have multiple ways, nevertheless all of them involving the "mex" command. Every external programs have to go through this Matlab Executable External interface function to execute in Matlab environment.

In order to execute C code within Matlab environment a C compiler is necessary. Under Windows environment we have many choices such as Microsoft Windows SDK 7 package, Visual C++ series 2008 version above and Intel C++ composer. For convenience and compatible consideration we select the most common combination: Windows 7 64-bits version and Visual C++ 2013 version in this research. Type "mex -setup" in Command Window of Matlab interface is the first thing to do. The purpose of this command is to make sure this Windows PC have a proper compiler been installed and ready to compile any C code.

Reconstruct a mex-function style source code is the most intuitive way of executing C code in Matlab environment. The following code snippet shows mex style programming.

```c
#include "mex.h" // Always include this

/* user defined functions */

void mexFunction(int nlhs, mxArray *plhs[], // Output variables

                int nrhs, const mxArray *prhs[]) // Input variables
{
    /*user code here*/

    return;

    }
```
The `mexFunction()` is the entrance of whole program, the first two parameters are related to output and the last two describe input to the function.

Before adding any line of code we need to locate the header file "mex.h", which located under Matlab installation directory, then "extern" and "include".

- `nlhs` short for "number of left hand side", which indicates the number of output arguments, or the size of output array.
- `plhs` short for "pointer to left hand side", this pointer points to the output array.
- `nrhs` short for "number of right hand side", this is the number of input argument or size of input array;
- `prhs` short for "pointer to right hand side". We pass the input argument through this pointer into `mexFunction()`.

Basically, the concept here is we just need to add the header file and "mexFunction()" function body into the C source code, then assign input and output argument and their quantity to the `mexFunction()`. The real `main()` function of C source code have to be called from `mexFunction()`. When all set save the “mex” source code into Matlab workspace folder and type “mex” followed by all source files, for instance type:

```
mex source_1.c source_2.c
```

In command window to compile the source code. Or we can specify the outputted object name in following way:

```
mex source_1.c source_2.c -output Object_1
```

After compilation “Object_1” will be treated as an independent function and can be called normally in both Matlab environment or through “Matlab function” block in Simulink/Stateflow (SLSF) model [16][17]. However this method must compile each source file separately, and could not visually represent each state of the program execution. As a result we switch to focus on Stateflow chart implementation.
3.4.2 Source Code Implementation in Stateflow Charts

Due to the advantages of Stateflow chart owns we choose it instead of Matlab function blocks, code reorganization such as adding `mexFunction()` will become unnecessary. Although source code still could not be introduced from the original project directly, we only have to identify the parts that we interested from source project then copy them into a separate source file and declare them in a header file [18]. I will explain this method in detail.

From previous discussion the whole program can be divided as low level hardware initialization, mid-ware translation and high level software implementation. Due to the limitation of simulation platform we can only focus on high-level software. The real communication such as transmitting and receiving messages are simplified as direct connection from source to destination block with calculated transmission delay; some mid-ware implementations been rewrite to help easy observation such as the transmission function in Timer 1 routine “can1TxMessage()”, instead of using data shifting and bit flipping to match real register organize we created a substitute function “directTxMessage()” so that the communication process is simplified only presenting ID and data field on the bus. Also when receiving messages we insert a small piece of customized code to introduce received ID and data from SLSF model into the C code. On the other hand for easy result observation, after every interrupt subroutine iteration we output interested data through customized code to Simulink Environment.

For all high-level software which hook up with interrupts, we create four separate source files for four different interrupt subroutines on ACM side and named them just by their function names, for the CAN 1 module: acmlsrC1.c and acmlsrC1.h contain almost all related functions. In real scenario the code may contain some special registers such as `C1RXFUL1bits.RXFUL3`, we simply rewrite them as `C1RXFUL1bits_RXFUL3` and define
them as global variables. Follow the same pattern we have reconstructed `acmIsrT1` routine, `acmIsrT3` routine and `acmIsrT4` routine. An extra step is that all functions the source file contains must be declared in a header file with same name. For some functions shared by several interrupt subroutines we created a new source file named as `acm.c` to store them; for all number defines, structures defines, global variable defines and all shared functions declares we put them into `acm.h` file.

Reconstruction of ACW wing controller side of CAN 1, Timer 1 and Timer 3 routines can be implemented following the same rule. However we have encountered a design problem: in reality the example program is organized in a same project which contains code for all nodes in the car, by specify the compilation profile at compile time, corresponding program can be deployed onto different MCUs. While in simulation environment both side program have to executing in the same platform, some global variables and even functions that shared same name among ACM and ACW will definitely cause problems and conflicts. For example both ACM and ACW side have CAN message store function “`ctSetMessage()`” in their CAN 1 subroutine, it stores incoming messages into a receiving table according to their ID, if both side call this function it means they will share a same receiving table, which is not an expected behavior.

One possible way to work that around is to extract normal Stateflow blocks into “Atomic Subchart” then create Simulink Library to hold them and call these charts from there. However this way requires directly define variables within the Stateflow block rather than within source code and call the source file from Stateflow block. Naturally this way does not solve the problem. So the solution is to rename all shared functions and global variables. For example the following function is used through all nodes:

```c
void heartbeat_mwDecrement(uint8_t iTimer)
{
```
if(aHeartbeatTimer[iTimer] != 0)
    aHeartbeatTimer[iTimer]--;
}

So it has to be renamed to:

void heartbeat_mwDecrement_acw_1(uint8_t iTimer)
{
    if(aHeartbeatTimer_acw_1[iTimer] != 0)
        aHeartbeatTimer_acw_1[iTimer]--;
}

for the first ACW controller.

It follows the rule that add "_acw" to indicate this function or variable belongs to ACW side; add "_number" after "acw" to indicate occurrence of this entity. In this study one ACM controller and two of the ACW controllers with node address of 0x462 and 0x468 have been implemented.

Before create a new model the most fundamental thing to do is to include all source code and header files into SLSF model: open Model Configuration Parameters by click the gear icon on the Toolbars; select Custom Code under Simulation Target from left menu; uncheck "Parse custom code symbols", in "Include custom C code in generated" panel, select “Header file” and add all header files with their relative path into the blank on the right; then add all source code relative path and file name into “Source file” field under “Include list of additional” panel. As shown in Figure 3-9:
A clock source input to the Stateflow chart is necessary. According to original program a “Pulse Generator” block with frequency of 1000Hz has been selected as clock source for Timer 1 and CAN 1 block; another one provides 100Hz clock to Timer 3 and 4 block. CAN 1 block is not based on timer but on incoming event so we feed it with 1000Hz clock signal while use a “switch” block to actually control whether it is on or off.

Next step is to create charts for each interrupt subroutine and hook them up with the C source code from inside. For example within ACM CAN 1 module we have two states: “init” (short for initialization) and “isr” (short for interrupt subroutine), all array and receiving table initialization happens in “isr” state then after one clock cycle it enters “isr” state. In “isr” state every clock cycle the block will iterate custom receiving code, the CAN 1 routine and the custom output code. Figure 3-10 demonstrate CAN 1 block:
Output of CAN 1 hooked up with a scope to observe what information have been received by CAN 1, but it is only for debugging purpose since all data flow are archived by C code through global variables. Follow the same rule all other Stateflow charts can be constructed. In reality these interrupt subroutines cannot executing at the same time however if no special care taken in SLSF all Stateflow chart with same clock source could execute at the same time. So the treatment could be disable any potentially concurrent executing chart or just plug in a “delay” block on some charts clock signal to delay their execution. After a research we found that the later method would be an easy implementation and proper analogy. Figure 3-11 shows how the “delay” block is used:
Same rule applies to other part of the model. An illustration of high-level hierarchy structure of the model is given in Figure 3-12 for details please refer to the whole model.
Figure 3-12 Hierarchal Structure of the Model
Chapter 4
Simulation and Experimental Results

In order to assist car designers to analyze their design we would like to conduct a series of testing. In phase 1 we observe simulation output, check if it matches with the TTCAN schedule; in phase 2, the model on PC serves as ACM side and use a CAN interface NI-8473s from National Instrument between PC and real car to test the robustness of the bus and observe the movement of the wings.

4.1 Simulation Results

First of all the output from ACM timer 1 block have to be checked. In order to give a straightforward result observation the data field is not included. Messages represented by their IDs are adequate in this case since the schedule is more important and real data are difficult to distinguish from scope. We should be able to observe messages represented by ID: 0x3200, 0x1a00, 0x1c08 and 0x1c0a when cycle number is 0 and window from 0 to 3, represented by the first wave of upper graph in Figure 4-1; ID with 0x3200, 0x1c00, 0x1c08 and 0x1c0a when cycle number is 1 and window from 0 to 3, the second wave of upper graph in Figure 4-1; 0x3200, 0x1c02, 0x1c08, 0x1c0a and 0x460 when cycle number is 2 and window number through 0 to 4, which matches with the third wave of upper graph in Figure 4-1, and all of their duration is 1ms. Notice here is that this is only data represented by their ID instead of actual electrical signal carried on the bus. For a reference message with 75 bits in a frame (calculation is given before) it needs 300 micro seconds to fully transmit; for a normal data message with 131 bits \((75 + (7 \times 8 \text{ bits}) = 131)\) which means it needs 524 micro second to finish transmission. So the 1ms time interval of transmission period is fulfilled for all messages.
Now let us check the output from ACW side: from top to bottom in Figure 4-2 are transmitted message ID from ACW side, current cycle, and window reading of ACW side and received ID from ACM. From observation, during cycle 0 ACW sends messages with ID 0x462 and 0x1e00, and 0x1e00 only when cycle is 1 and 2, this result matches with the time schedule. One thing need to be noticed here is that when ACW received reference message (the tallest one in lowest graph, ID equals to 0x3200) from ACM, its local cycle and window information did not update. This is due to the delay of transmission time: when CAN 1 receives a reference message it starts Timer 1 with a preload TMR (corresponding to 300ms, equals to reference message transmission time) value to compensate this transmission delay, so in this model we give same clock to Timer 1 block on both side while CAN 1 block is given a 500 micro second (500us is for normal message, calculation given before) delay to mimic this transmission delay. Before this reference message coming, the last window of previous cycle update window number to 9 and keep same cycle number in Timer 1 block, then Timer 1 block turn off itself. In the next 1ms the reference message is sent and due to the transmission delay, 500us later Timer 1 block is activated by CAN 1
module, as a result in this neighboring 2ms window and cycle information did not update. It is not a programming bug.

Next we check both ACW controller output on the bus. In the model both ID signal are added together through an adder block and feed into a new scope along with received ID from ACM. This can help compare simulation with the schedule. In Figure 4-3 the upper graph is the transmitted ID from both ACW side (message with ID equals to 0x462 and 0x468), the lower one is received message ID from ACM side. After messages from ACM we observe when cycle number is 0 there are 0x462, 0x1e00 then 0x1e06; in next cycle there are 0x468, 0x1e00 and 0x1e06; in the last cycle there are only 0x1e00 and 0x1e06. Each one of them takes up to 1ms length and the empty time interval between them are either 1ms or integer times of 1ms. This pattern match with the TTCAN schedule. Again we could observe ID 0x1e06 overlap with next transmission cycle, this is because of the same reason stated before.

Figure 4-2 ACW (0x462) Timer 1 Block Output
Another aspect to check is the heartbeat function. Every node on the bus have their own heartbeat message, the purpose of this message is to let other node on the bus keep track of their “physical” status. We observe heartbeat of ACW controllers through the heartbeat output of Timer 3 block of ACM side. From Figure 4-4, the upper curve is the heartbeat counter output of Left Front wing controller and the lower curve is from Rear Right wing controller. We can see that at the beginning of time all of them set with an initial value of 50, in each iteration of timer 3 routine this counter will decrement by one no matter it receives heartbeat or not. Every time Timer 3 block receives a heartbeat message from one of the wing controller it will reload the initial value of 50 into the counter. If it does not see the heartbeat for a half second: 500ms = 50 * 10ms (timer 3 period), the counter will decrement to zero, then the driver should be alarmed that one of the node might be malfunctioning. Each node have one chance to send its heartbeat message within the 3-cycle time table, which will left two cycles without its heartbeat, as we can observe from
Figure 4-4: after the counter initialization, timer 3 routine decrement by one so the highest peak shows 49; then decrement twice until it receives the heartbeat message again and this process repeats.

![ACM Timer 3 Routine Heartbeat Counter Output](image)

4.2 Bug Detection

During this modeling and analysis process two bugs have been successfully located in the original program. The first one was from code analysis stage, in function `ctInit()`, array `aCanTableTx` been initialed twice, while `aCanTableRx` array been missed out, as following program snippet shows:

```c
void ctInit(void)
{
    ...
    for(iIndex = 0; iIndex < CAN_TABLE_RX_LENGTH; iIndex++)
        canInitMessage(&aCanTableTx[iIndex]);
    // Set all message items within the TX message table to blank.
    for(iIndex = 0; iIndex < CAN_TABLE_TX_LENGTH; iIndex++)
        ...
    ...
}
```
canInitMessage(&aCanTableTx[iIndex]);

...}

This bug may cause unwanted clear operation since CAN_TABLE_RX_LENGTH is larger than CAN_TABLE_TX_LENGTH. When all elements are initialized in aCanTableTx the program continues setting zeros to memory locations out of that array.

The other bug is located during simulation, in ACW side CAN 1 routine the “pending” flag did not set along with storing message into the receiving table, in the last line of the following code snippet, stMessage.bPending = 1; does not have any effect, because ctSetMessage() function requires all flags set before it is called.

if(C1RXFULL1bits.RXFUL1)
{
  canInitMessage(&stMessage);
  can1RxMessage(1, &stMessage);
  C1RXFULL1bits.RXFUL1 = 0;
  // Store this message in the CAN message table
  iCanTableIndex = ctGetMessageIndex_acw(CanTable_Receive, stMessage.iId);
  ctSetMessage(&stMessage, CanTable_Receive, iCanTableIndex);
  stMessage.bPending = 1;
}

In Timer 3 routine when inputting all received data from receiving table (aCanTableRx) into corresponding data structure such as stAcwSensor and aAcwServo, the program checks the “pending” flag comes with the carrier (stMessage) data structure at the first place, if the carrier data structure did not present “pending” flag it
will eventually lead Timer 3 routine fail to access the content in which the carrier carries, the communication fail. Figure 4-5 shows the scope with wrong output.

![Figure 4-5 Failure of Communication Due to the Flag Not Been Set](image)

This bug lead to a series of program rearrangement and brought up new agenda to replant all tasks onto a real-time operating system to make sure similar problem should not happen.

### 4.3 Experimental Results

In order to assist design in a more interactive way a field test is designed. We select NI USB-8473s CAN bus interface to connect ACM side of the model (SLSF model on PC) with the real car. The input of ACM side of the model could have multiple testing scenarios for instance to test the servo motor response time. This device has a USB connection with PC, on the other side it provides a DB-9 port to hook up with any CAN bus. During real field testing we found current software setting could not support this test, the problems are: 1. Message out of synchronization. TTCAN schedule requires that every message transmitted at a specific time, or the time period between two messages are deterministic. While under Simulink environment, CAN transmission block can only assign a specific transmission period. This causes the problem of when the CAN block transmitting message on the bus, the TTCAN schedule may not providing data, instead the data present
to the block could be all zeros. A possible way to work around is that when unnecessary turn off the transmission block but Simulink does not support run-time block disable. Another important limit factor is also time, the simulation have to be real time to interact with real world activities, but under our current software and hardware setting it is very hard to archive. Figure 4-6 demonstrate this issue, at each red arrow, one transmission operation should happen while we cannot precisely control it.

![Figure 4-6 Synchronization Issue](image.png)

Nevertheless we still conducted a field test with the real car, through oscilloscope the messages on the bus did not follow the way we wished. The first problem is that the time interval between each messages is very small no matter what transmission period we set in the software, we have tried SLSF model and Matlab script. The other problem is that the ID field observed from oscilloscope is incorrect: if we send 0x1c08, 0x1ffc1c08 is observed.

For the first problem we could use code generation function of Simulink to download testing model to a CAN bus supported hardware platform such as Beaglebone then use Beaglebone as the interface. The second problem could be solved by deploying a Simulink Real-Time software onto a dedicated computer equipped with real-time kernel.
Due to the limitation of time and resource, these two problem could be solved in further research.
Chapter 5

Conclusion

5.1 Conclusions

In this thesis, we successfully transplant a multi-MCU embedded platform application into the Simulink/Stateflow (SLSF) environment. We also perform analysis and test the feasibility of the TTCAN protocol in SLSF. During code analysis and simulation, we also helped designers locate two valuable bugs that led to a series of updates to their original embedded program. Overall, the methodology is to take legacy applications from embedded platforms and integrate them into model-based design (MBD) and analysis tools like SLSF. Many legacy systems were not designed using an MBD process, while current systems are frequently designed with MBD processes. This thesis describes a case study of using this methodology to integrate legacy-embedded code into modern MBD processes. In industrial applications, this method could be used to assist post-design phase performance analysis and functionality testing, and additionally help designers test systems that use legacy code and subsystems in new systems designed following an MBD process.

5.2 Future Work

MathWorks’ SLSF provides a real-time solution for some applications that requires real-time interaction between SLSF models and actual hardware, such as hardware in loop simulation. This system consists of a development computer, Speedgoat target [19] [7] computer hardware and the physical system needed to test. The SLSF model exists on the development computer, which can be a normal Windows PC with a C compiler, Matlab Simulink, Simulink Coder and Matlab Coder; the Speedgoat target computer acting as a bridge between the model and real physical system, it equipped with a Simulink real-time kernel with concurrent execution support. These two computers could directly connect
through a private point-to-point network or indirectly through LAN or Ethernet. The physical system directly hooks up with the real-time computer [7]. In addition, this real-time software and hardware platform supports any block from the Simulink library. We consider this as one promising method that would help to solve the time mismatching issues we have encountered as described in Chapter 4.

In order to solve the synchronization issues between hardware and the SLSF model, a customized testing platform can be created with Simulink Coder [20]. The process is to create a model for the interface and deploy it on CAN compatible 3rd party hardware such as Beaglebone or Raspberry Pi to serve as the bridge between SLSF and the real hardware instead of the NI CAN device we attempted to use, as described in Chapter 4. The customized platform improves configuration capability by deploying our own code.

For improvement of the SLSF model itself, we could involve all components of the actual racing car into the model: including battery packs and its supporting circuits, motors, wheels, servo motors that control the wing blades, wing blades, aerodynamics and physical dynamics. For a closed-loop control model, consistent of a physical plant and the software controller, we could then use a translation tool such as HyLink [21] to go from the SLSF model to a model with formal semantics such as a hybrid automaton. Then, formal verification of properties of the closed-loop control system could be analyzed using techniques like reachability analysis. This has advantages compared to traditional simulation while also allowing us to use common Simulink blocks, etc. Other approaches to analyze the SLSF models would include using static analysis methods such as Abstract Simulation, which focuses on correctness criteria of the Simulink model [22], or deploying dynamic analysis tool such as Hynger [23] to assist identify any assumption mismatches. Other directions would be to perform symbolic analysis that aims to improve simulation coverage for a hybrid system like the whole aero control system [24].
References


Biographical Information

Mr. Zhang finished his Bachelor’s degree of Telecommunication Engineering in Tianjin University in 2012, after one year of working in Beijing as hardware engineer, he came to University of Texas at Arlington to continue his graduate education, under Dr. Johnson’s supervisor and successfully earned Master of Science degree in Electrical Engineering department.