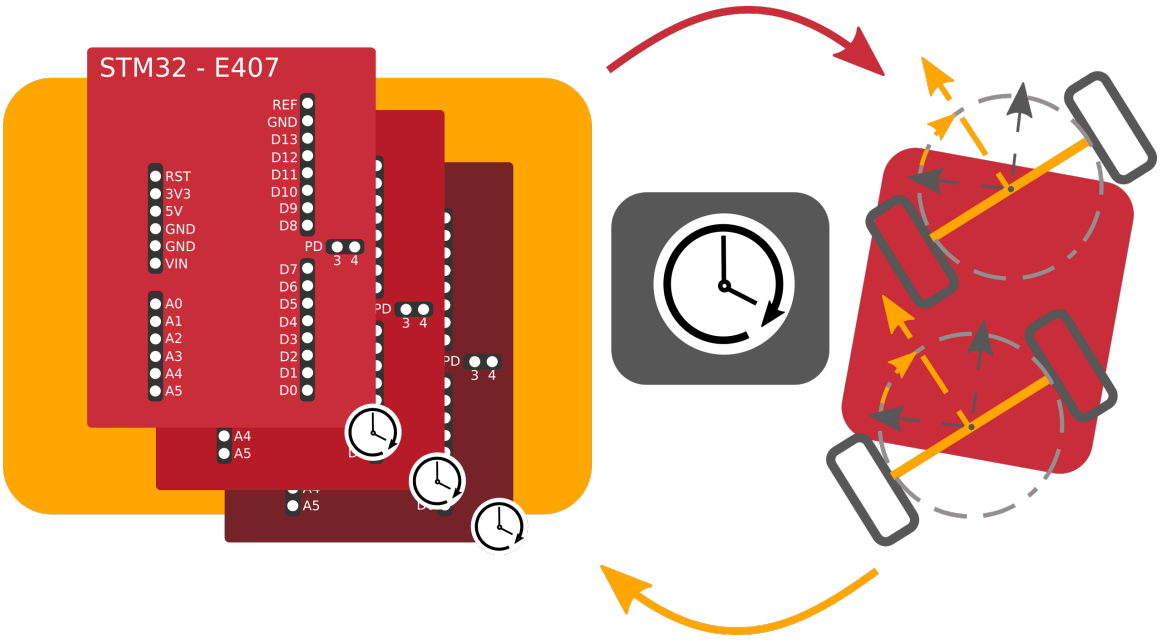
HAN_UNIVERSITY OF APPLIED SCIENCES

HAN MASTER MAJOR PROJECT

Time-Triggered Architecture, fully redundant, Real-Time embedded distributed system with HANcoder

SOFTWARE DOCUMENTATION



Diego Martín López

February 2022

17th February 2022- v.0.0

Contents

1 Introduction

This document serves as a guide for the software developed in the master thesis Time-Triggered Architecture, fully redundant, Real-Time embedded distributed system with HANcoder. The project has two main products: the TTA template and the TTA controller. The first is the template prepared to create distributed embedded applications with a Time-Triggered Architecture (TTA) schedule and the second is an application example with the controller of a two-axle vehicle showcasing how to use the template.

The author of the master thesis and the first version of this document is Diego Martín López. However, the start of the TTA template software involved many other students as it was developed during the minor project of the embedded systems module in the master in engineering systems at the HAN. Mentioning at least some of those who contributed the most towards the completion of the minor project with a working first prototype is the least I can do. The order presented is merely random: Dominic Pendery, Jordy Koole, Jamie Cheng, Praveen Rajendran, Francesco Felloni, Jasper Verhoef, Emiel Gerrits and Adam Sali.

Even though the program is not especially extensive nor complex, in the beginning, it might appear challenging to cope with all its functionalities at once. This document shows every part of the software in a comprehensive way interrelating the most important systems. The main goal of the document is to present enough information for the next generations to improve the current version and allow the template to reach its uttermost potential. There are some problems limiting the functionality of the software discussed in the master project report linked to this documentation. Understanding how the code works and following the advice from the master's report will lead the next programmers towards the software improvement.

The structure is compounded of two main sections, the HANcoder code and the system debug and prototype measurements. The first is the most extensive part and guides the reader towards the whole software understanding, starting with an installing guide and following with a software tour, from the highest levels of hierarchy to the deepest systems in the Simulink model. The second part briefly introduces the measurement and test procedures with smaller test programs, HANTune and a logic analyzer.

The documentation focuses on presenting the whole TTA controller software, as the TTA template is contained within the controller. The only difference between both is that the template only contains the first basic cycle from the controller matrix cycle. Every other basic cycle in the TTA controller is not present in the TTA template program, but everything else remains the same.

2 HANcoder code

The software installation, code overview and systems guide is presented in this section. It starts by introducing all the elements required to make the program work deploying it into the prototype ensemble. It follows with an overview of the MATLAB startup file, making special emphasis on the most important configuration options. Next comes a software overview that gives a first understanding of the code structure. Later, a brief software execution order explanation clarifies how the different Simulink blocks are activated in the program during runtime. From then on, every system in the code is explored with a brief explanation of its main subsystems and functionalities. The last part of this section presents a list of every variable and signal in the system.

2.1 Download, install and setup

The software is written in MATLAB Simulink using the HANcoder extension. The program is deployed in STM32-E407 boards using Microboot. Real time diagnosis is done with HANTune and post diagnosis with a logic analyzer and python scripts. This section briefly shows where to find each program with the versions employed for the project development and the main steps required to start working with the prototype.

The MATLAB version used during the project development has been R2018b. It can be downloaded here. HANcoder 1.0 was used to access the board digital pin control and CAN communication functionalities. HANTune build 68 was chosen to monitor the Simulink signals. This HANTune version allows python scripting to process signals during runtime. When creating a new HANTune file it is important to set the Communication setting to XCP on USB/UART if the connection with the board is going to be done over USB. The panel with the correct option can be seen in figure (1).

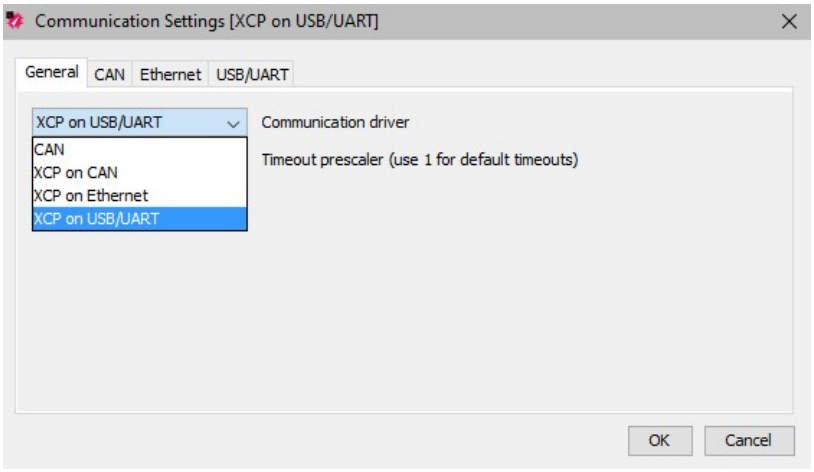


Figure 1: HANtune communication settings.

The logic analyzer employed is from the company Az-Delivery. It can be found here and it counts with 8 channels and 24 MHz resolution. It is possible to connect directly with the device using the program Saleae Logic, with which the digital signals can be easily recorded and exported in .csv files for post-processing with Python.

The whole software project can be found in this repository in GitHub. Downloading the whole code structure allows for running the software, build it and deploy it. In figure (2) it is shown where to press to download the whole project.

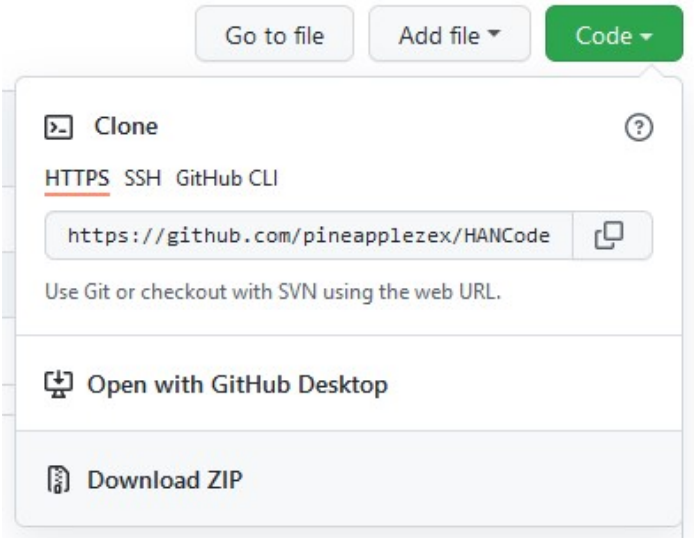


Figure 2: GitHub code pannel with the Download ZIP option at the bottom.

It is also possible to just copy the .slx Simulink files into a HANcoder folder. However, be aware of file shadowing when cloning projects, as a Simulink project could be using a different HANcoder source file to compile than the one expected. To avoid this, change your Simulink preferences in the File tab, and inside Model File tick the option saying: "Do not load models that are shadowed on the MATLAB path". Compiling and deploying is as easy as pressing the build button shown in figure (3). When the compilation finishes the MicroBoot application will automatically pop-up and wait for a board to be reset (or just connected) via USB. However, big files such as the TTA_Controller program may take up to several minutes to compile. To avoid having to wait for a full compilation to deploy the program into the boards, it is possible to use the MicroBoot program manually and select the .srec file generated after compilation. The MicroBoot program can be found in the Host folder of the project.

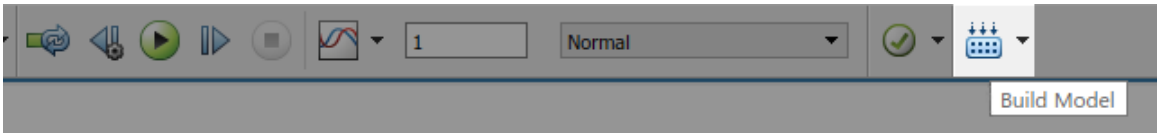


Figure 3: Simulink's build button.

The same program must be deployed in every board. Distinguishing between boards is done by connecting 5 V into digital input pins D2 to D5. A full hardware connection diagram can be found in figure (4).

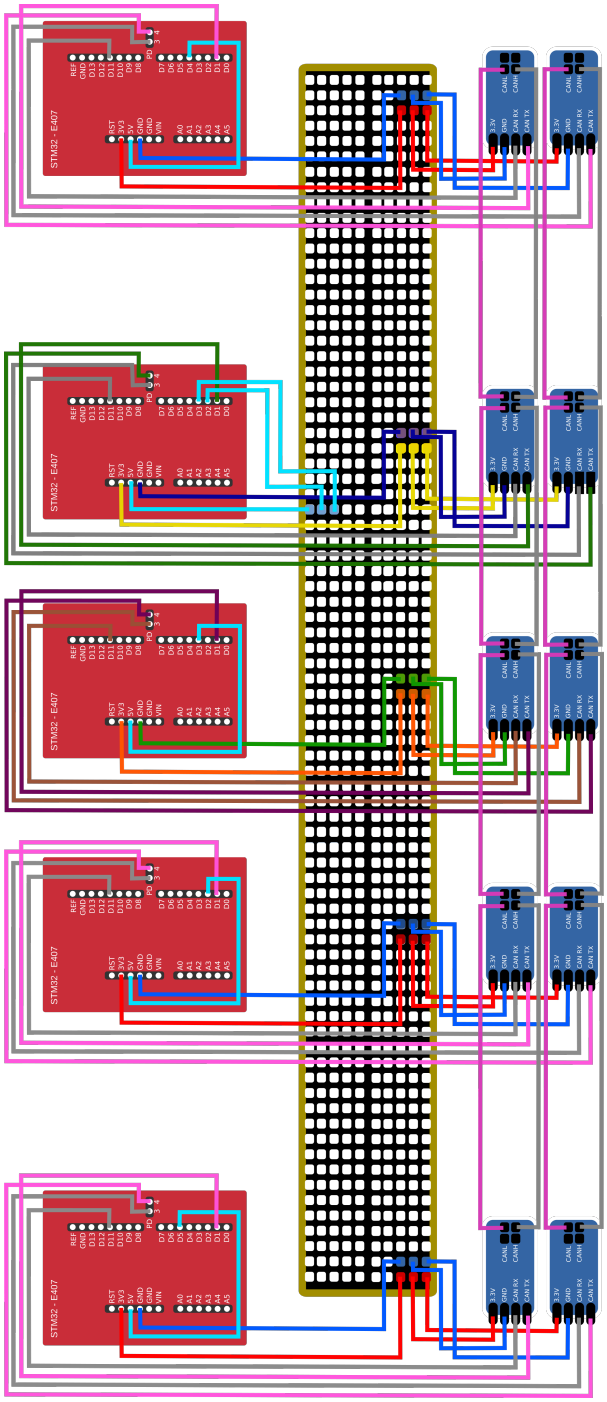


Figure 4: Electric scheme for the TTA_Controller prototype. The red squares represent the STM32-E407 boards and the blue rectangles the CAN transcievers.

2.2 Matlab startup file

The MATLAB startup file is executed when the Simulink file associated to it is executed. It contains information about constants, parameters and type definitions. The startup file for the TTA_Controller is divided in different sections, starting with "Measurements". Here the different flags governing the logic analyzer measurements can be activated. When set, each flag allows for the activation of the appropriate digital pin toggle system. More information about each individual measurement is presented later in section 3.2.2.

The next MATLAB section is "Constants", with all the constant values needed for the Simulink model, from the roles in the controller board to the message identification numbers. There are some special mentions to make. The hardware_granularity value is not actually selected from MATLAB, but directly in the Simulink model. In figure (5) it can be seen how to change the frequency of the hardware clock. More information about how the local time is set in the system is presented later in section 2.7.

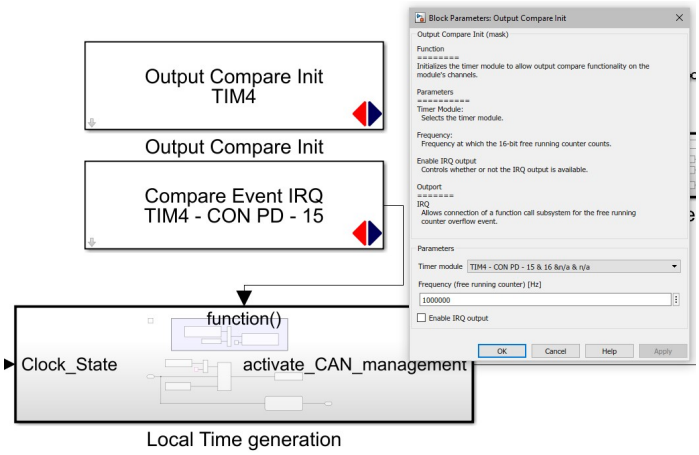


Figure 5: Output Compare Init block, where the frequency of the hardware clock can be set. A frequency of 1 MHz is set for this version of the software.

The communication tasks are defined by a series of parameters. These are defined this way in the MATLAB file so they can be modified during runtime. This allows for faster investigation of the system behaviour with different configurations. More information about the communication tasks and their parameters can be found in section 2.15.1. The maximum number of messages that can be sent during a communication task is seven (three bits), as defined in the coding and decoding systems from section 2.24.2 for the temporal information of the messages.

The communication delay of a message is the time from the moment a message is sent by a board until it is received by another. This time depends on the CAN baudrate. It has been measured that for a baudrate of 1 Mb/s the communication delay is 0.3 ms and for 250 kbit/s the communication delay is 0.7 ms. More information about the communication delay measurement is shown in section 3.2.2. In figure (6) it is presented how to change the baudrate of the CAN channels.

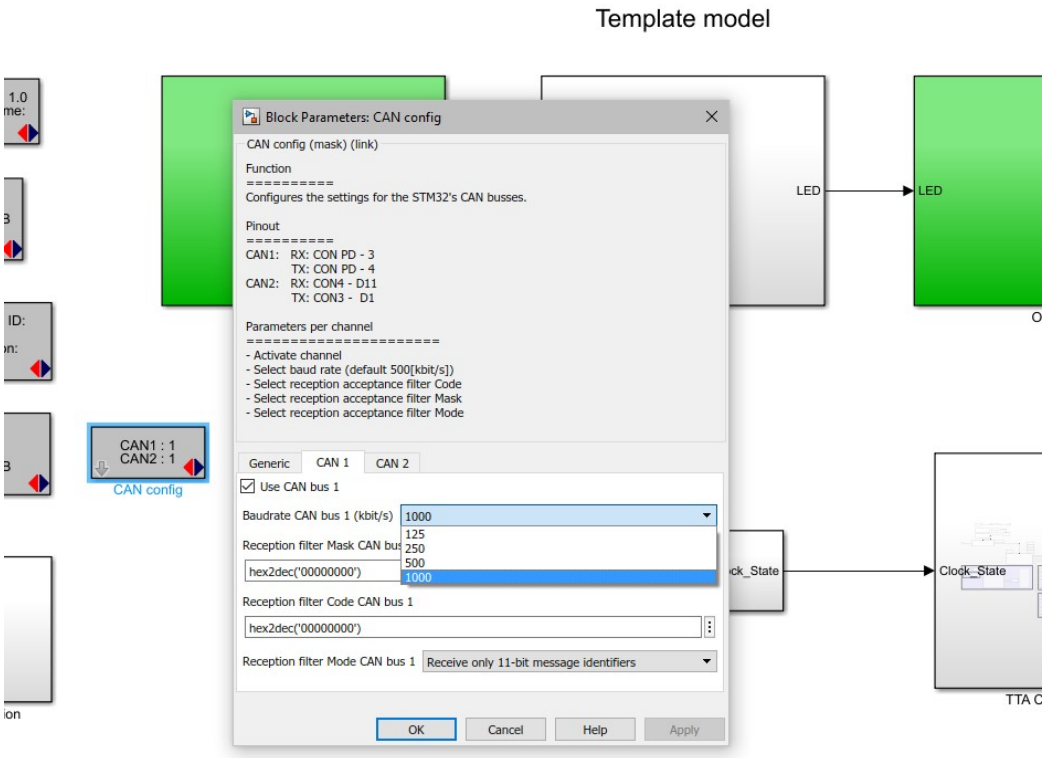


Figure 6: Baudrate selection for the CAN channels at the CAN config block.

The last interesting constant definition in the startup file is max_desync, a saturation limit for the desynchronization in the Desync_Ticks variable. It is set to ±15 ticks to prevent a too high desynchronization due to an unexpected too high communication delay in a message transmission. The Desync_Ticks register is presented in section 2.12. Its saturation is defined later in the desync calculation at the reference message check task, presented in figure (44).

The next section in the MATLAB file is the "Definition of Time marks", the moments in the schedule when a task starts. Every time mark is defined twice, one for the constant value and again inside the data array. This array together with the type array are used in the Positive_Desync system presented in section 2.12. Every task can either be oriented to communication (COMM) or computation (COMP). The file follows up with the "Value domain constants" defining the two-axle vehicle physical properties. Then, the parameters for the input generator signal and the controller gains are defined. Lastly, there are two type definitions, the vote_array for the vote count, explained in figure (56), and the message buffer, the type of all the messages for the CAN communication.

2.3 TTA_controller_v3 overview

The HANcoder code, with some help from the MATLAB startup file, defines the behaviour of the prototype. This is compounded of five different boards connected to each other with CAN transceivers. Each board has some more cables connecting the 5 V pin with some of its own D pins, defining its purpose in the ensemble. A board can either be part of the controller (IDs 1, 2 or 3), the input generator (ID 4) or the vehicle emulator (ID 5). These identification numbers are chosen using binary code in the digital input pins D2 to D5. Each part has its own matrix cycle defined in the TTA schedule, with specific tasks that have to be executed in order to control the vehicle emulator. During this schedule different messages are exchanged using the CAN channels ensuring that the boards remain synchronous and that the controller operations are held in the appropriate moments. The synchronicity of the boards is kept by the establishment of global time, having a time master share its own local time with the other boards. A board's operation can be stopped by freezing its own local time with the wake up button (WKUP) in the board. This document guides the user through the different sections of the code where all these mechanisms are developed.

The project software starts with the template page, where the two first important systems can already be spotted: the operation mode system and the TTA CAN system. The operation mode controls the activation of the TTA CAN system using the WKUP button from the STM32-E407 board. The main code is inside the TTA CAN system, and follows the structure in the scheme shown in figure (7).

The documentation starts introducing the software execution order before diving into the operation mode system, which shows how pressing the WKUP button is transformed into a boolean variable. Then the TTA CAN system is divided into its different sections, explaining them one by one, putting special attention in the TTA system, where most of the code is contained. Following the overview direction from figure (7), the TTA system is explored layer by layer, showing how the matrix cycle is initialized with the board's role, how the basic cycle is updated and looking into the mechanisms to debug the system with the logic analyzer. The documentation follows with the matrix cycle manager, which contains the matrix cycle for the different elements of the prototype: the controller, the input generator and the vehicle emulator. Each matrix cycle contains two basic cycles which furthermore contain the schedule's tasks. After studying each individual task of the schedule and how they are coded, including when the CAN subsystem should be activated for transmission or reception of messages, the documentation rises up again to the TTA CAN system layer to finish the code tour with the CAN interface.

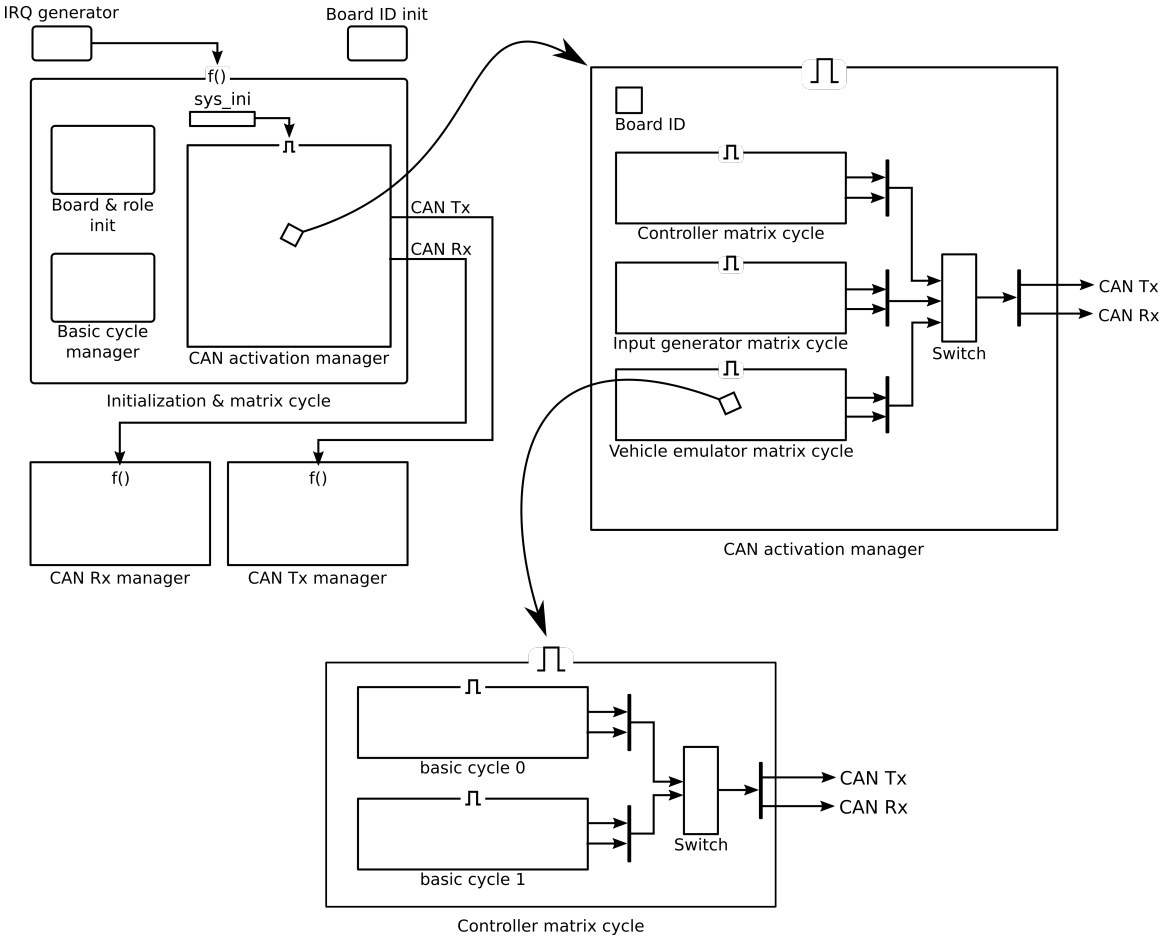


Figure 7: Software overview of the project. The TTA CAN is presented in the top left part of the figure. Some elements of the TTA system are visible inside. In the right side the Matrix manager is magnified, showing the matrix cycles. Finally, the bottom part of the picture shows the inside of a matrix cycle, with the basic cycles.

2.4 Software execution order

A normal programming code such as C, even though might seem cryptic for someone who does not know how to interpret it, defines a very straightforward order execution. Simulink blocks, however, are displayed in a "free void" in which unconnected systems could execute in the order Simulink finds more appropriate. Usually, blocks are executed from left to right and from top to bottom, also taking into account that any connected block is executed after the previous. More information about execution order and how to control it can be found in Simulink's documentation.

The main ideas used to be aware and manipulate the block's order in the prototype's code are the blocks colors and their priorities. In figure (8) it is shown how to view the block's colors and a legend showing what each of them means. The sample time distinguishing each of them is set by Simulink. However, most of the blocks employed in the prototype's software are run under triggered subsystems, governed by a hardware clock.

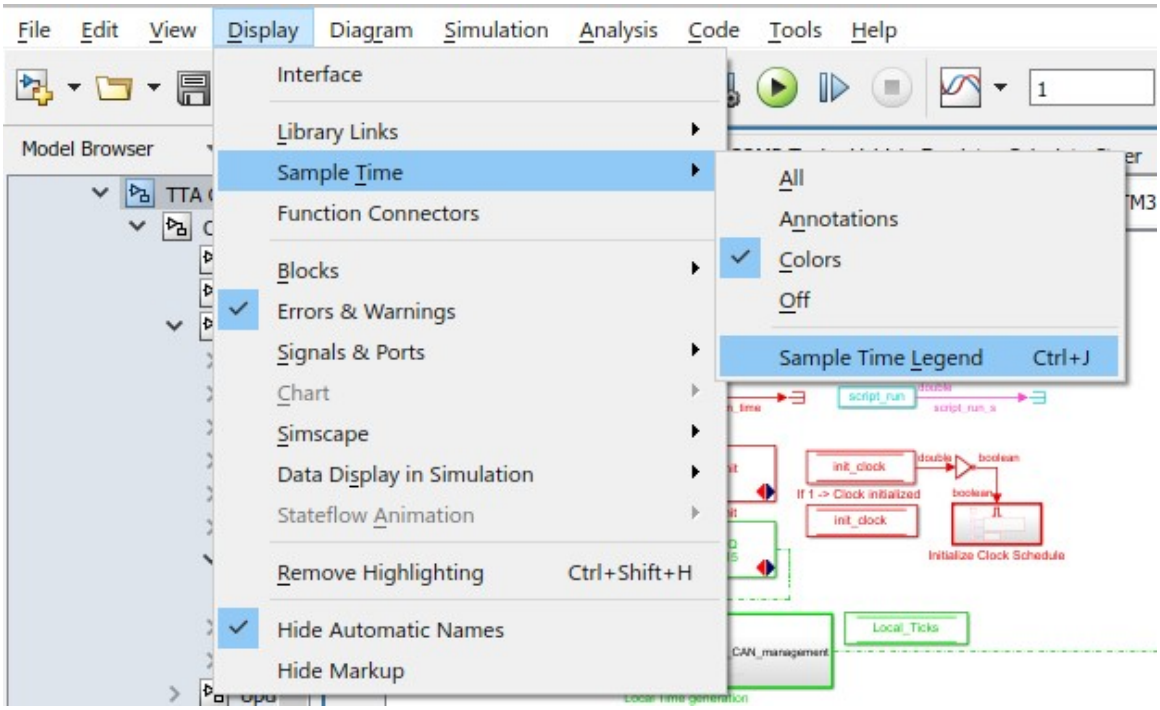


Figure 8: Menu to visualize sample time colors and their legend.

Triggered subsystems, such as function called subsystems, if subsystems or enabled subsystems (more information here), do not execute under Simulink's sample time, and each time they are executed, each block inside is invoked only once. The order in which each block is activated can be controlled using priorities. These can be accessed in the advanced tab of the properties in the property inspector, as presented in figure (9).

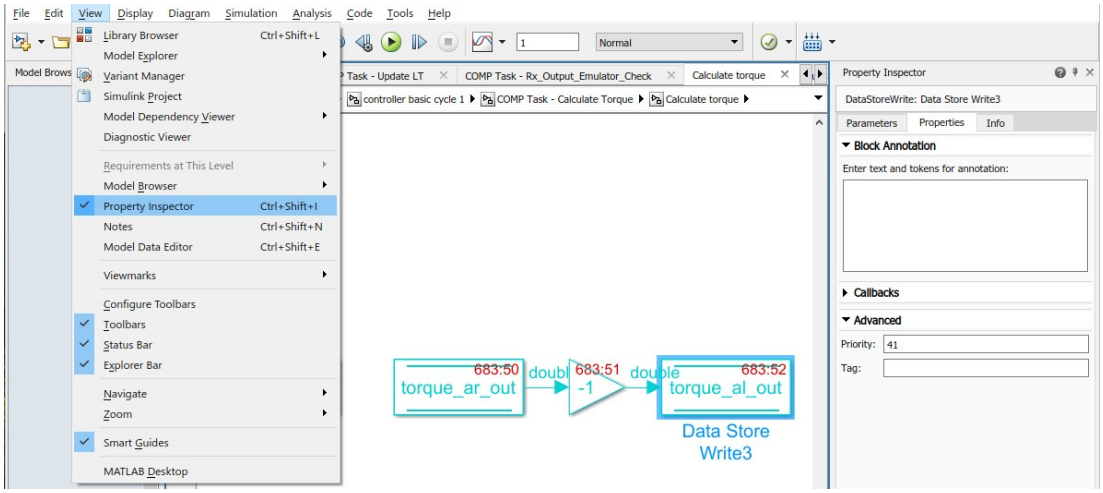


Figure 9: Menu to visualize the priority of a block.

Seeing the execution order of the blocks in the Simulink model requires the model to be compiled and the sorted execution order to be activated, as shown in figure (10). A block with a lower priority (a higher number in the priority parameter) will have a higher value in the execution order (it will be executed later). Blocks without a priority value will be sorted by Simulink disregarding the priorities relation with the other boards.

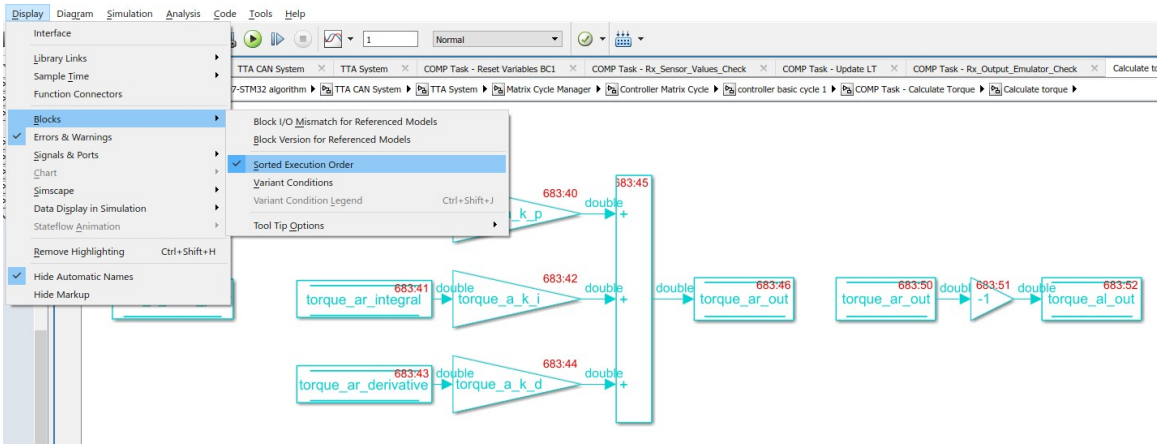


Figure 10: Menu to visualize the sorted execution order.

When working with these enabled systems it is important to pay special attention to the output configuration. Once there is a concrete order in the software controlled by the priorities we have set it is easy to forget that there are systems in the software that are not going to be activated with every run, and which could affect global variables they are connected to. In these cases we have to set the output of those systems to either hold or reset, depending on what do we want to happen with that specific system. If we want it to export a zero value when it is not activated, its outputs should be reset. If we want it to keep the last output it processed during its last activation, it should have its outputs configured as hold.

2.5 Wake-up button

The wake up button (WKUP) is one of the two buttons the user can press to interact with the board. The other button (RESET) resets the board. The code uses the WKUP button to freeze and resumes the operations of a board, allowing for some fault injection operations and code debugging. Figure (11) shows how the operation mode system looks, taking the Button_State as an input and transforming it into the Clock_State variable.

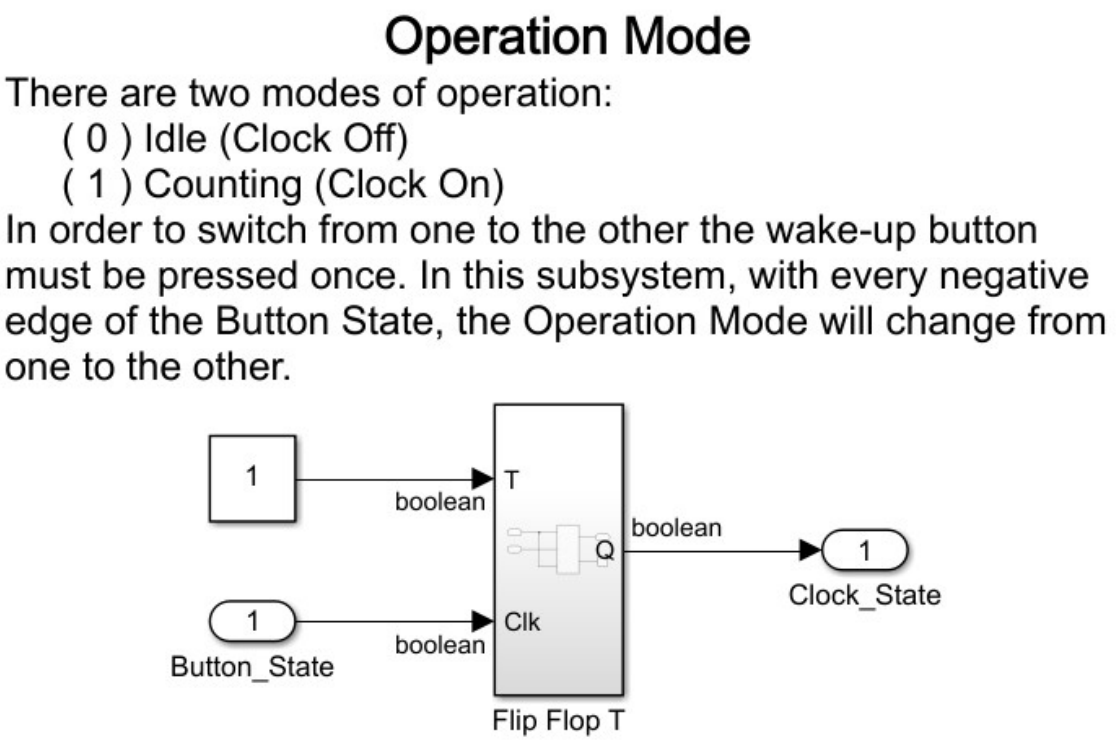


Figure 11: Operation mode system.

The Button_State responds to the button pressing. While the button is pressed the variable remains high, until the button is released. Pressing the button creates a rise edge in the Button_State signal that is used as the clock of the Flip Flop T to toggle the value of the Clock_State, so if it was True it changes to False and vice versa. The Clock_State boolean is later used in the time generation system to stop or resume the increment in the local time counter.

2.6 TTA CAN System

Inside the TTA CAN System there are different subsystems as presented in figure (12). These can be grouped in different parts:

1. Local time generation, at the top left side.
2. Board ID generation, at the bottom left part.
3. TTA system, at the left middle section.
4. CAN activation systems, at the rightmost side.

These are explored in the following sections.

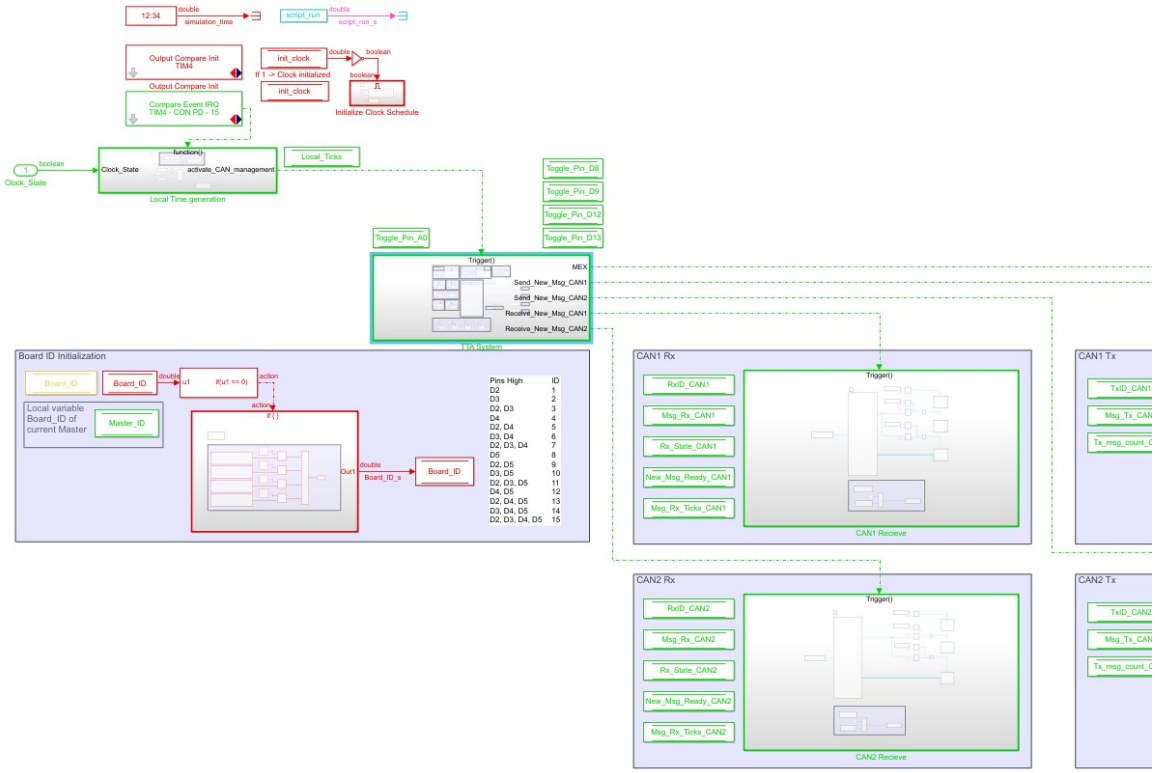


Figure 12: TTA CAN system. Systems in green are activated with function calls or IRQs, while the rest activate with the Simulink software loop.

The lines connecting the different systems can already give a sense of execution order. Those subsystems not connected to the others (initialize clock schedule and board ID initialization) are activated once when the board is switched on and never again. That is why those are activated with Simulink's software granularity. Everything in green is connected one to each other and follows an IRQ basis activation with the hardware clock. First the local time is increased in the local time generation system. Then the TTA system is activated, where the appropriate TTA schedule task is performed and it is decided if a CAN system shall be activated. Lastly, if a CAN transmission or reception should be held during this tick, the CAN systems are activated.

2.7 Local Time generation

Each board has its own local time, generated by a hardware clock. Using the clock frequency, it is possible to schedule interrupt requests (IRQs) that activate the other parts of the code. It is necessary to initialize the clock schedule once at the beginning of the execution. This happens in the Initialize Clock Schecule subsystem presented in figure (13). The Schedule Compare Event block allows choosing a clock from the board and schedule in how many clock ticks there should be an IRQ. One tick is selected (top input at Schedule Compare Event) to get an IRQ as soon as possible and the init_clock flag is set so this subsystem is not run again.

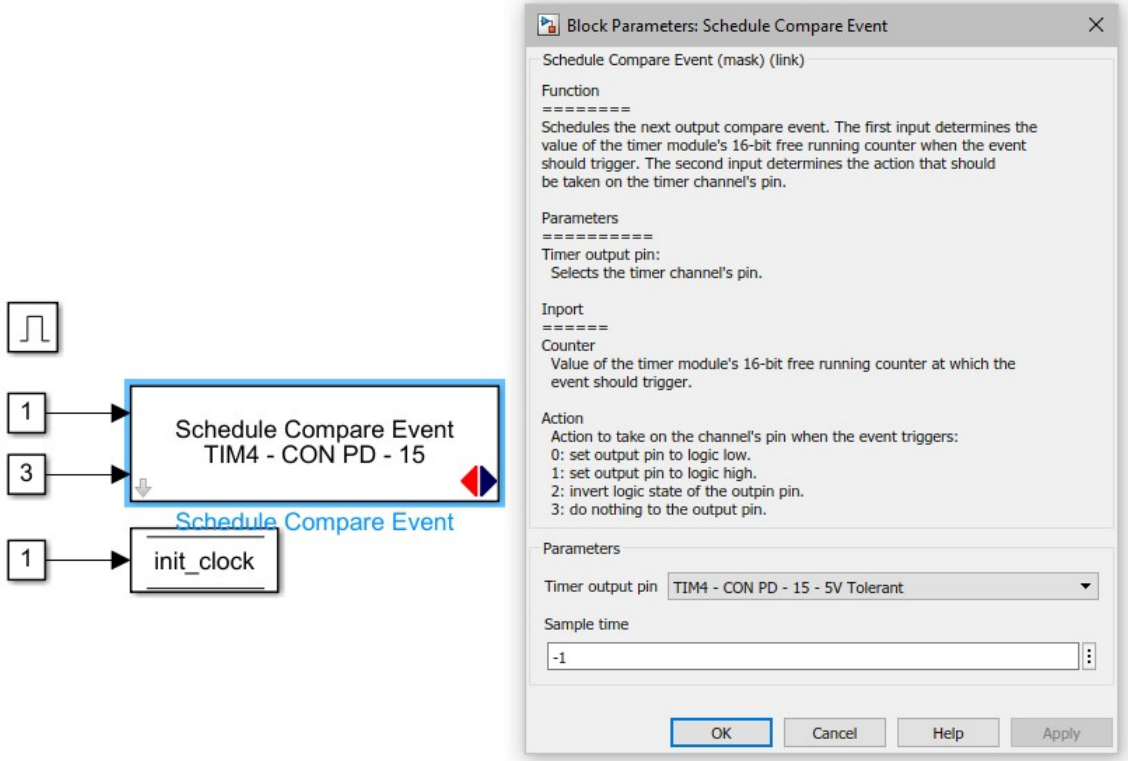


Figure 13: Initialize clock schedule.

When an IRQ from the hardware clock happens, the Compare Event IRQ block calls the Local Time generation subsystem. It is important that the Compare Event IRQ, Schedule Compare Event and Output Compare Init blocks refer to the same clock (TIM4 - CON PD - 15). The Output Compare Init block defines the tick granularity of the hardware clock. More information about the STM32-E407 clocks can be found in the board's user manual. These blocks and subsystems mentioned can be seen in figure (14).

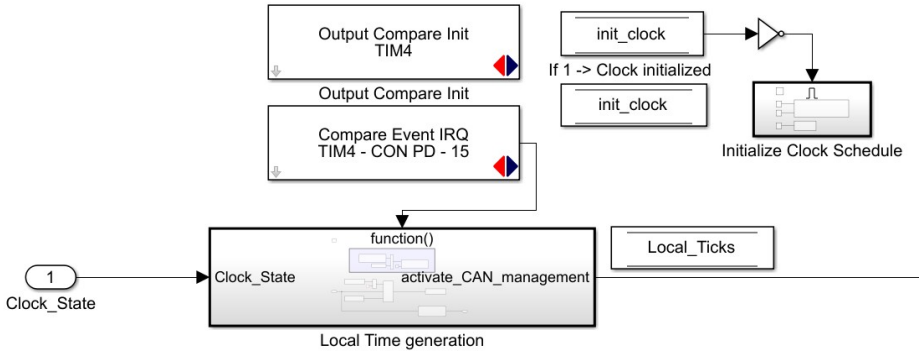


Figure 14: Local time generation group, with the necessary blocks and subsystems to make the local time increase.

Inside the Local Time generation subsystem the local time of the board is incremented and the IRQ is re-scheduled to keep the local time running. This is presented in figure (15). The variable frequency_IRQ is defined in the MATLAB startup file and defines the systems granularity. When frequency_IRQ = 100 it means that after 100 ticks of the hardware clock, there will be one IRQ, making the ticks in the local time to increase by one and activating the whole TTA system software. This frequency also depends on the speed of the hardware clock ticks. For example, with 1 MHz hardware clock frequency and frequency_IRQ = 100, the local time frequency is 10 kHz (0.1 ms).

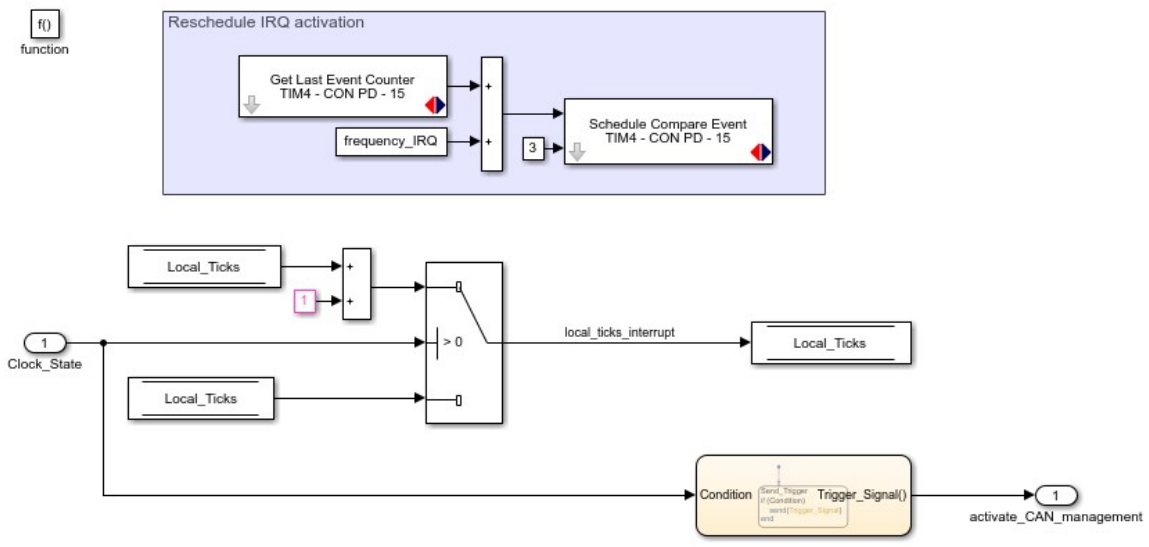


Figure 15: Local time generation subsystem.

When the Clock_State (governed by the WKUP button) is zero, neither the local time is incremented nor the function call for the TTA system is activated. More information about how the counter and the function call generator work can be found in section 2.24.

2.8 Board ID initialization

The board ID of a board is initialized at zero. If Board_ID = 0 the Board ID initialization subsystem from figure (16) is activated.

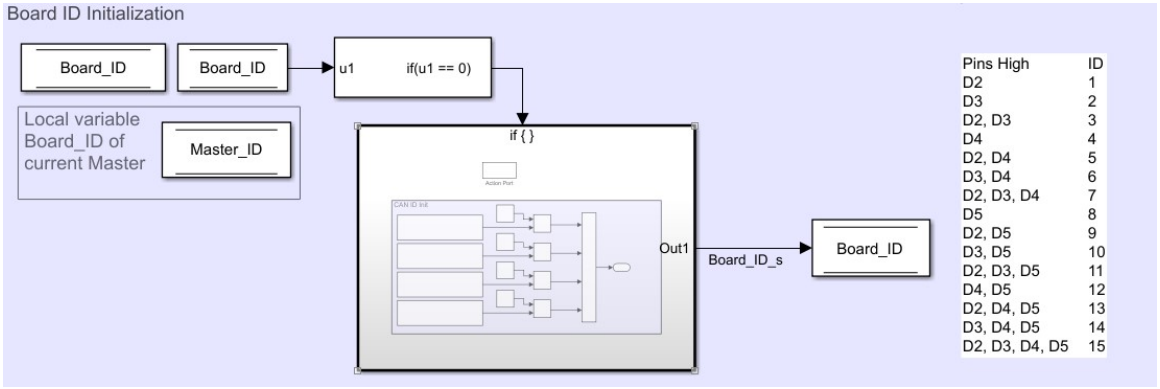


Figure 16: Board ID initialization group. This part of the code is run only once at the beginning of execution.

The Board_ID value is chosen inside the Board ID initialization subsystem, depending on the hardware configuration of the digital inputs. An input of 5 V is required in a combination of the D2 to D5 digital inputs, following binary logic, to choose the board's ID. The way in which the digital inputs are checked and processed is presented in figure (17).

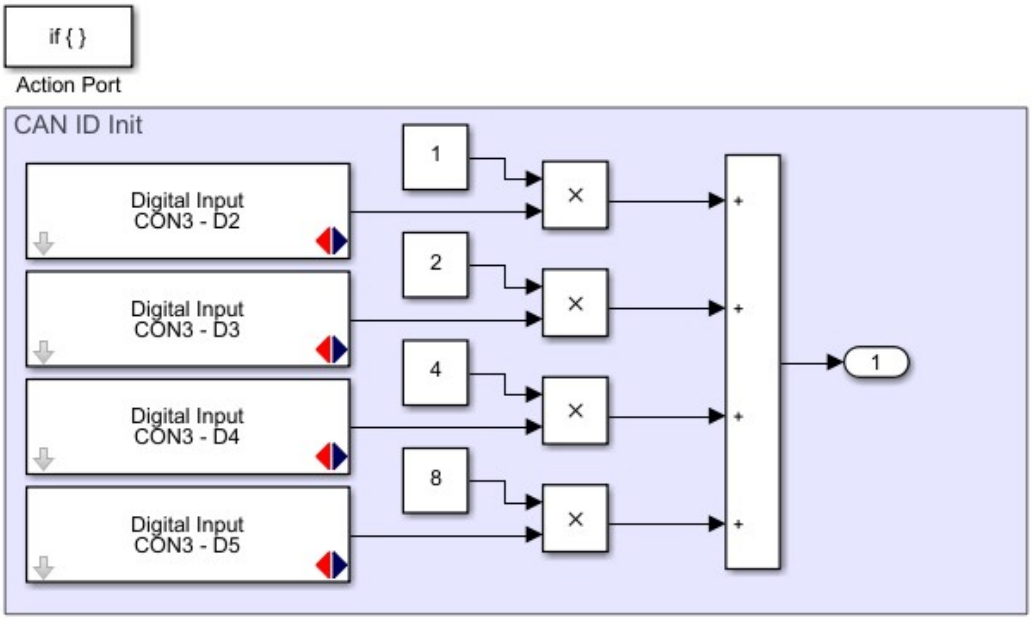


Figure 17: Board ID initialization subsystem.

2.9 TTA System

The TTA system contains different subsystems related to the matrix cycle manager, the subsystem in the middle of figure (18). This system includes the schedule initialization, the basic cycle update, the positive desync local time update case, the logic analyzer pin togglers and, most importantly, the matrix cycle manager. This last one outputs the boolean signals to activate the transmission or reception CAN systems. While the TTA schedule is still initializing, the CAN reception systems are activated every tick until a reference message arrives or, for the controller boards, enough time has passed waiting. More information about the initialization is included in the next section.

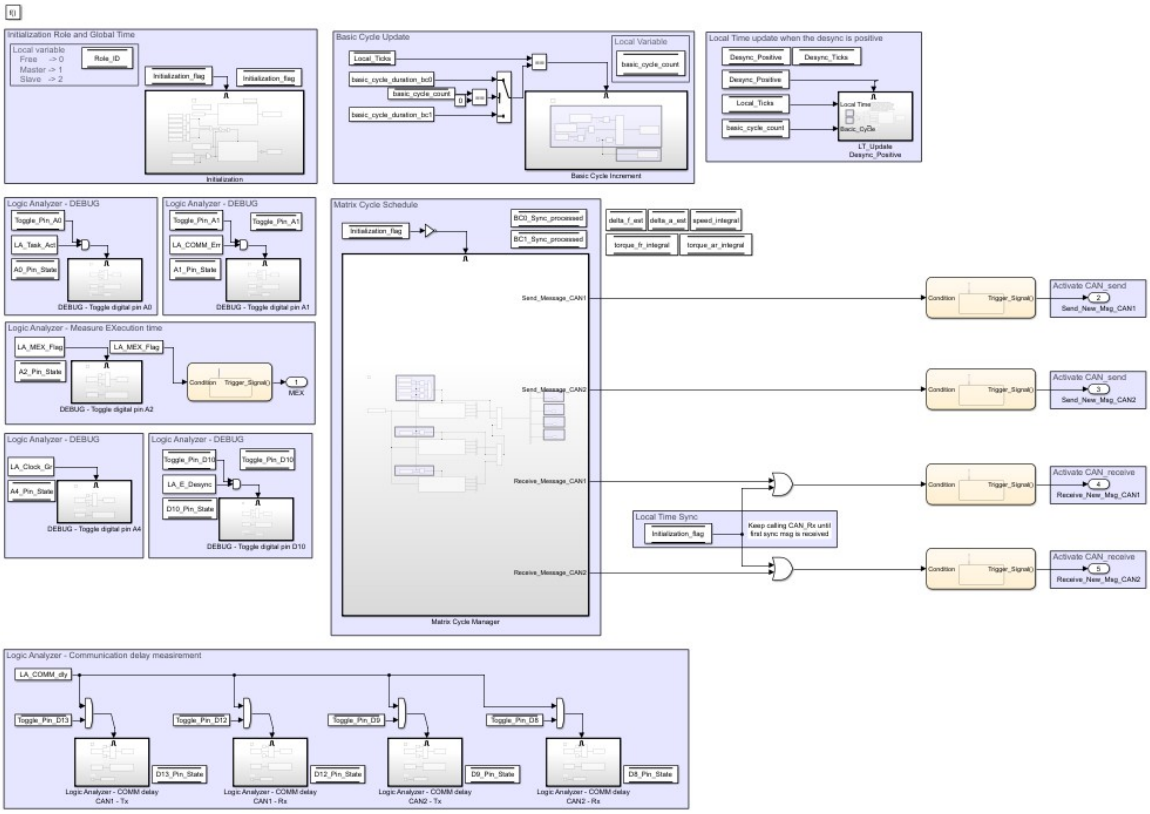


Figure 18: TTA system.

The TTA system runs once every time the local time generator creates a function call. The operations inside the TTA system must end before the next IRQ of the hardware clock happens. It is possible to measure both times, the granularity and the tasks execution time, using a logic analyzer. More information about how to use the logic analyzer debug is presented later in section 2.13.

2.10 Initialization

Before any of the operations in the matrix cycle can start, the time master of the ensemble must have already been selected. That is why it is necessary to make a initialization before activating the matrix cycle manager. The main overview of the initialization subsystem is presented in figure (19).

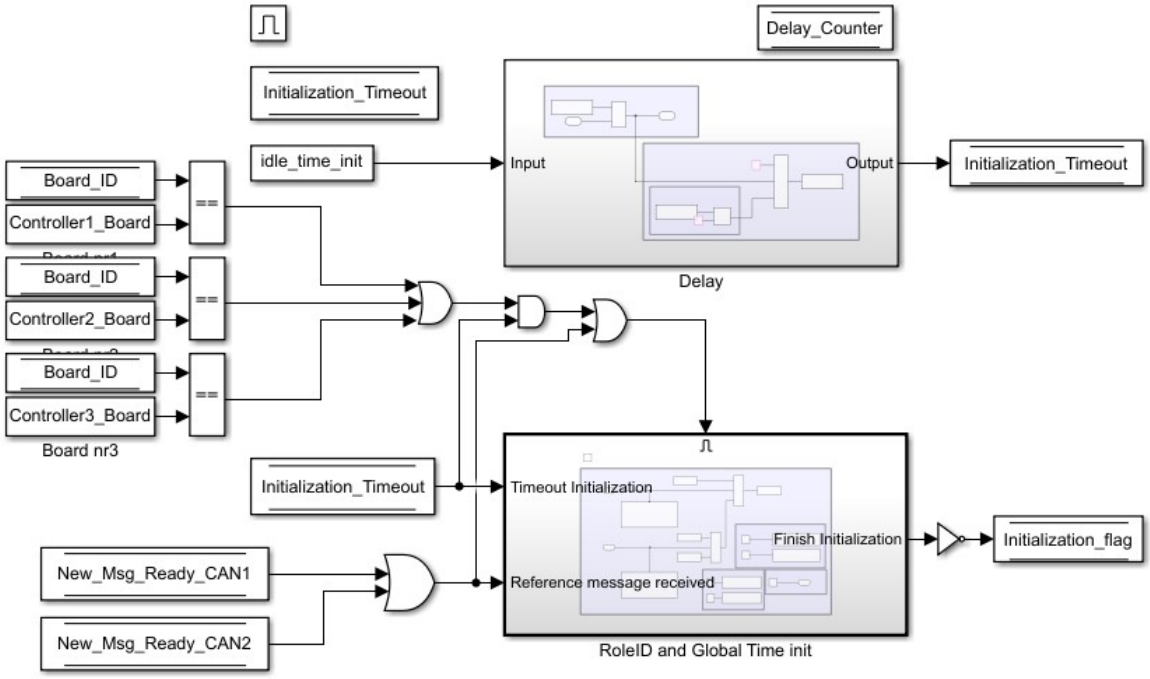


Figure 19: TTA initialization subsystem.

The subsystem contains two main parts, the delay and the role and global time initialization. If the board is part of the controller boards, the Delay subsystem will keep track of time for as much time as time defined by the idle_time_init variable, defined in the MATLAB start up file. The value for the delay is originally set at one matrix cycle duration, so if a controller board does not receive a reference message from other board, it takes the master role.

The delay is presented in figure (20), and is mainly built upon the counter idea. While the Delay_Counter has not reached the input value set by idle_time_init, it keeps increasing its value. The moment the Delay_Counter reaches the input period, the output is set to true, making the role and global time initialization system begin.
As it can be seen in figure (19), the role and global time initialization system is activated when one of the following condition is met:

1. The board is part of the controller boards and the delay reached its end, making the Initialization_Timeout true.

2. A message arrived at one of the CAN receiving interfaces. While the initialization happens the message expected is a reference message from the time master of the ensemble.

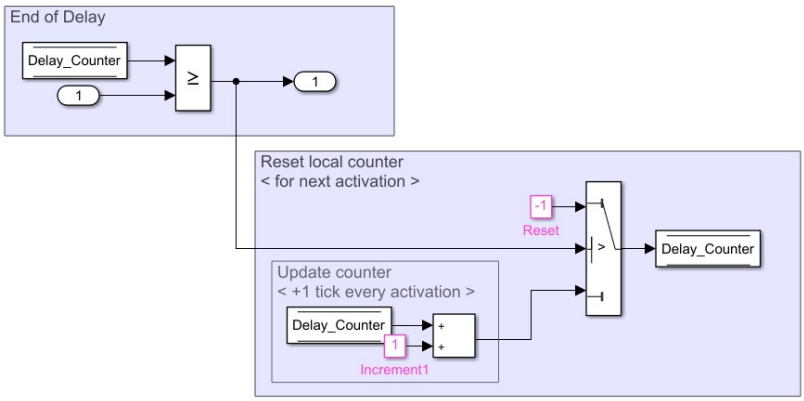


Figure 20: Delay subsystem in the TTA initialization.

In figure (21) the role and global time initialization subsystem is presented. Depending on how this system was activated, there are two different systems that can be activated. If the system was activated by the reference message arrival, the board initializes as a Slave, otherwise it initializes as a Master. Regardless of the role assigned, there are some variables that are prepared for the next time an initialization happens (Initialization_Timeout and Delay_Counter) and others that get reset to prepare for the first matrix manager activation (the new message ready variables). Also, the initialization is set as it has been completed successfully.

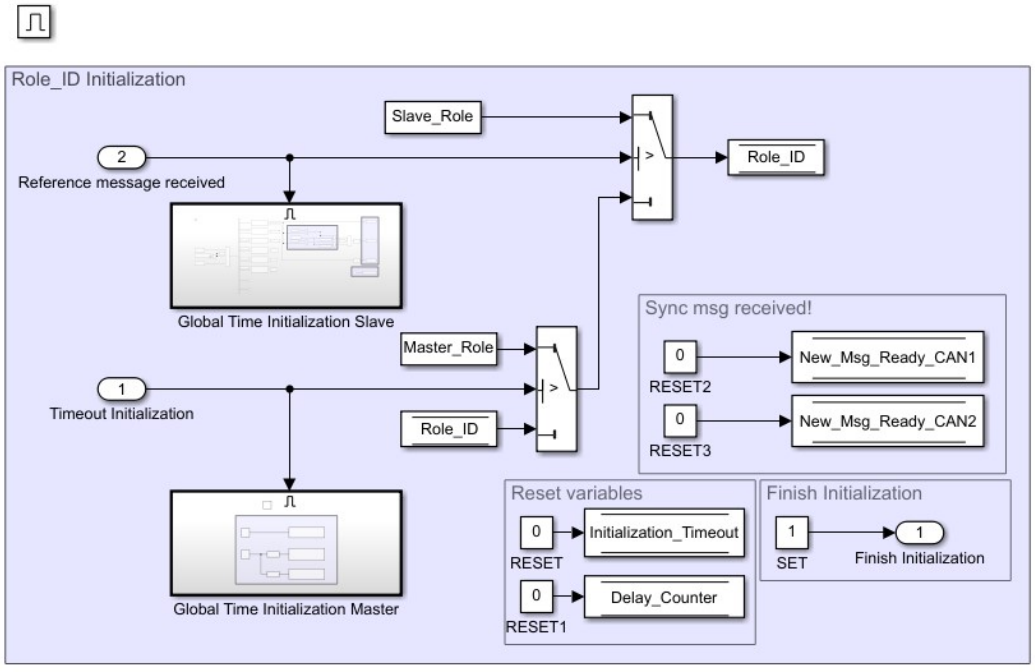


Figure 21: Role and global time initialization in the TTA initialization.

Before the code continues its operations with other systems in higher layers of the hierarchy, the code must run either the slave or the master initialization. The master initialization is presented in figure (22).

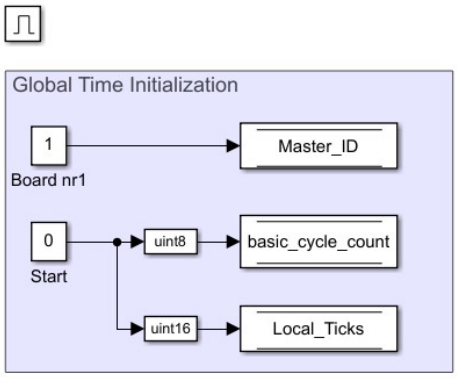


Figure 22: Master role initialization.

The slave initialization is presented in figure (23). In this subsystem the reference message received at the CAN system is processed. First, it is discerned if the message arrived at CAN channel 1 or at CAN channel 2. Then, the message received is divided into its components. Every message transmitted is compounded of a first byte of temporal information with the basic cycle in which the reference message was transmitted, the message counter of the communication task and the board ID of the board that sent the message. The reference message also carries value domain data with the values of the integrals used in the controller calculations. Every byte has to be decoded, the temporal information is compressed in a single byte and the integral values have to be transformed from unsigned eight values to floats. Lastly, the temporal domain information from the message is stored in the appropriate variables and the BC0_Sync_processed variable is set for the matrix manager calculations.

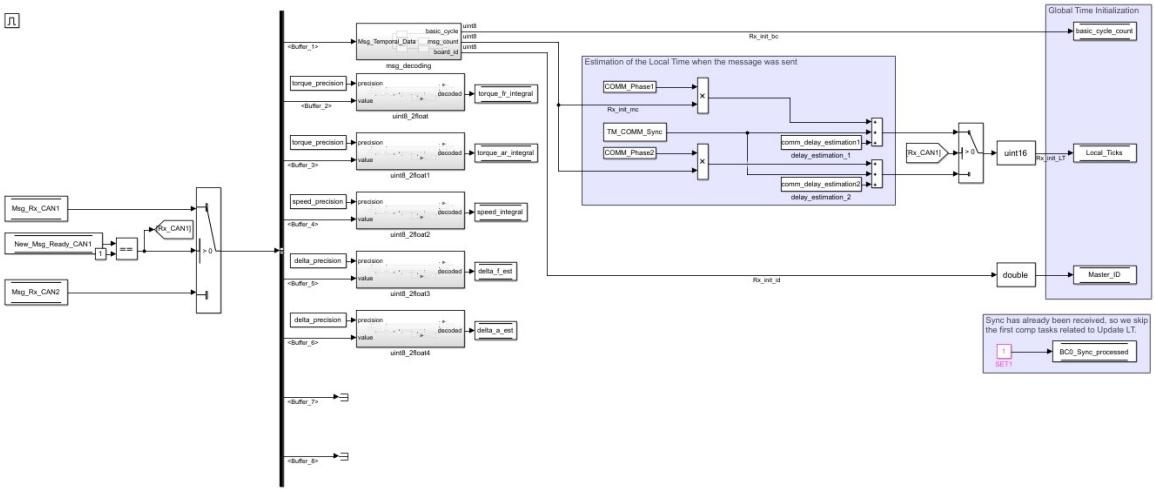


Figure 23: Slave role initialization.

2.11 Basic cycle update

The matrix cycles of the TTA schedule are compounded of two basic cycles. When the local time counter of the Local_Ticks variable reaches the basic cycle duration the basic cycle counter must be toggled. The basic cycle subsystem is presented in figure (24). Because the basic cycles from the proposed matrix cycle do not have the same length, both durations must be taken into account for the Basic Cycle Increment subsystem activation.

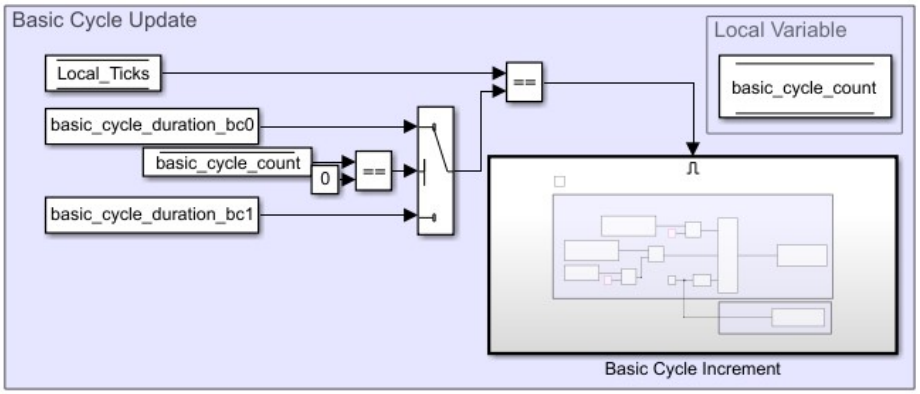


Figure 24: Basic cycle update group.

The interior of the Basic Cycle Increment system is presented in figure (25). When this system is activated the basic_cycle_count is increased or reset, in case the maximum number of basic cycles (matrix_rows defined in the MATLAB startup file) was reached. The moment the basic cycle is updated the local time is reset.

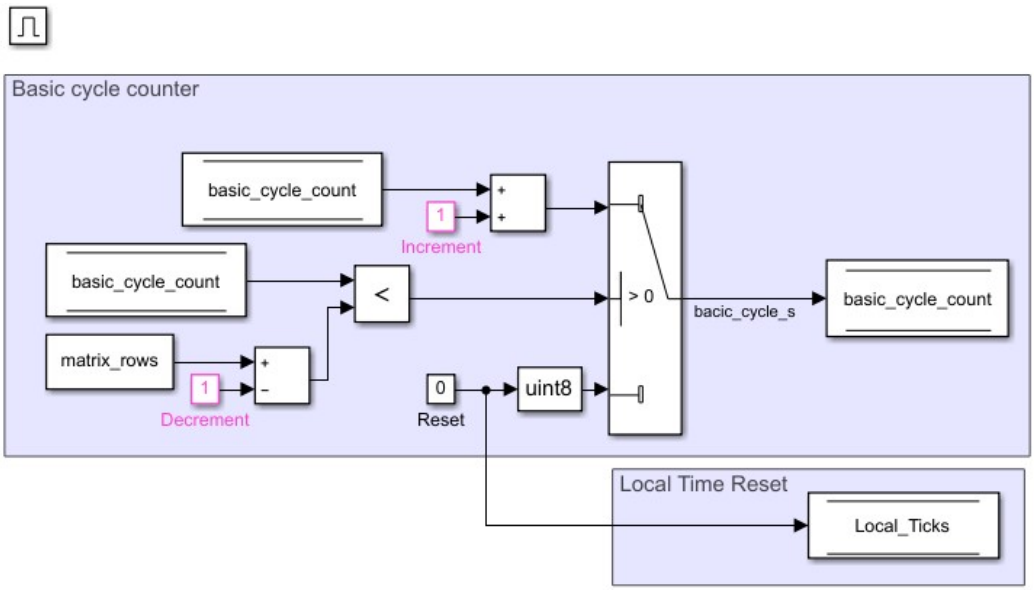


Figure 25: Basic cycle increment system.

2.12 Positive desync

The time slaves in the ensemble must update their own local time when receiving the reference message from the master and realizing that they are out of sync. This means that the local time counter (Local_Ticks variable) must be either decreased (negative desync) or increased (positive desync). This should not be done lightly, as the local time defines which task of the TTA schedule should be executed. On the one hand, reducing the local time is not very problematic, as it means "going back in time". Not repeating the tasks that have already been executed is enough to solve the problem. On the other hand, "going to the future" by increasing the local time could mean skipping some of the TTA Schedule. This is solved by increasing the local time in small steps, every time there is free time in the schedule in between tasks. The inside of the positive desync system is presented in figure (44). Depending on the current basic cycle, the appropriate basic cycle schedule with the task types are selected. These can be either computational (COMP) tasks or communication (COMM) tasks. A MATLAB script calculates with this information how many ticks are free before the next task activation.

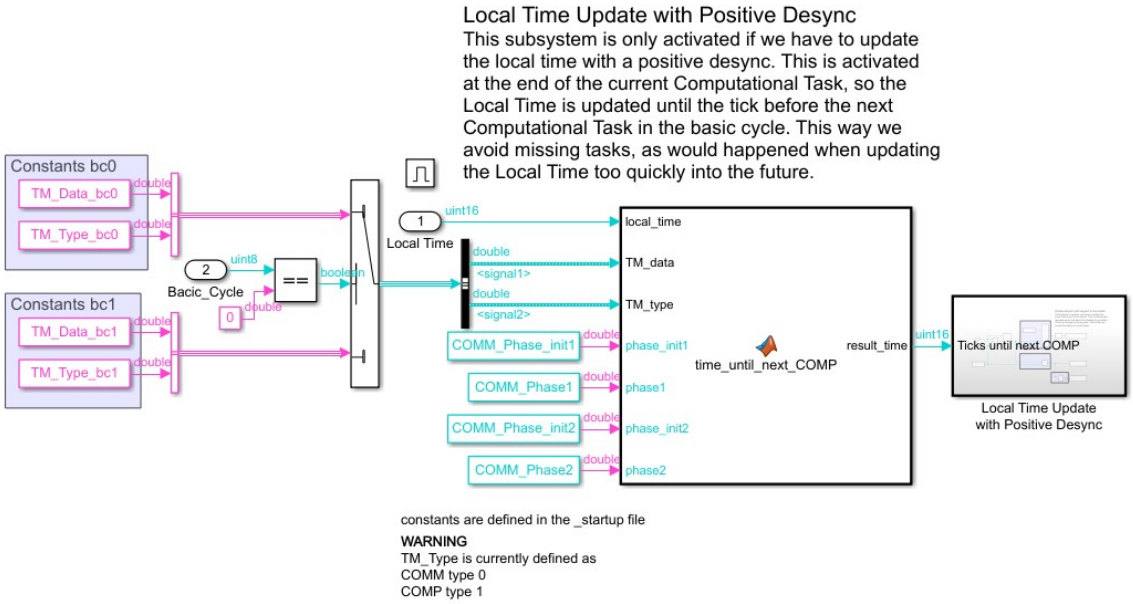


Figure 26: Positive desync system.

The local time and positive desync update is presented in figure (27). The local time is increased as many ticks as it was calculated by the MATLAB script. The positive desync is decreased the same amount of ticks, so if it is still not zero the operation keeps being performed while ensuring no tasks are skipped.

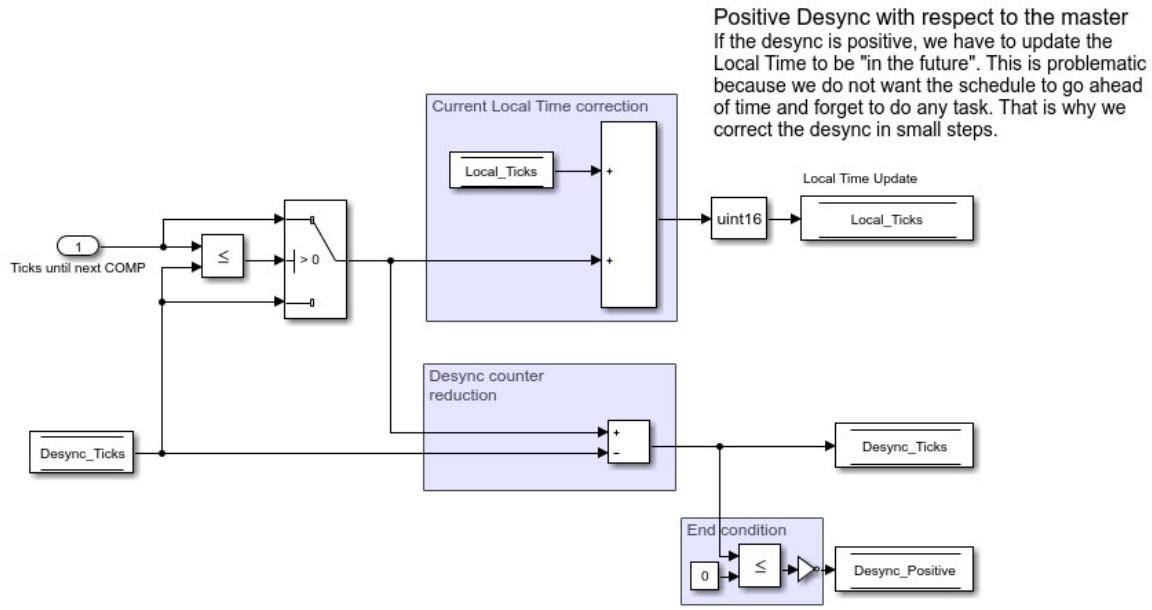


Figure 27: Desync and local time update.

2.13 Logic analyzer

There are two tools used to debug the software behaviour: HANTune and a logic analyzer. Because HANTune's sample frequency is limited (maximum 10 kHz) it is not able to show how the signal values change tick by tick. The logic analyzer is used to see this. There are six different logic analyzer's measurement coded:

1. Task activation.
2. Communication error.
3. Measure execution time.
4. Clock granularity.
5. Ensemble desynchronization.
6. Communication delay.

All of them except the clock granularity and execution time measurements require a toggle flag that is activated in a task inside the schedule. The communication delay is shown in figure (28) while the other five logic analyzer (LA) subsystems are presented in figure (29).

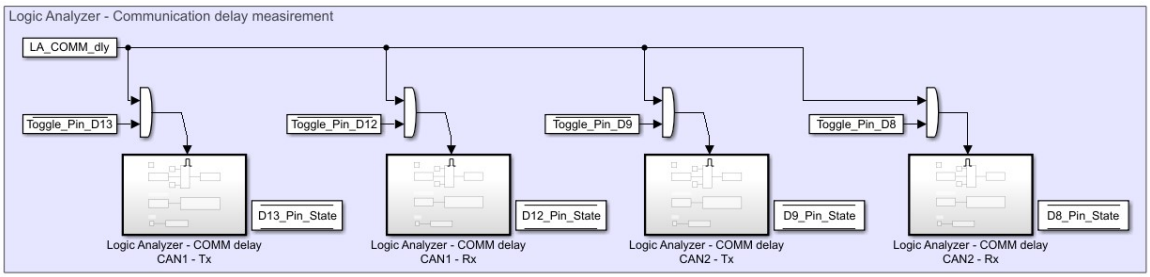


Figure 28: Logic analyzer communication delay subsystems.

The communication delay measurement counts with four different pins to account for the activation of the transmitter board and the receiver board in each channel. This subsystems are inside the TTA system and are presented in figure (28).

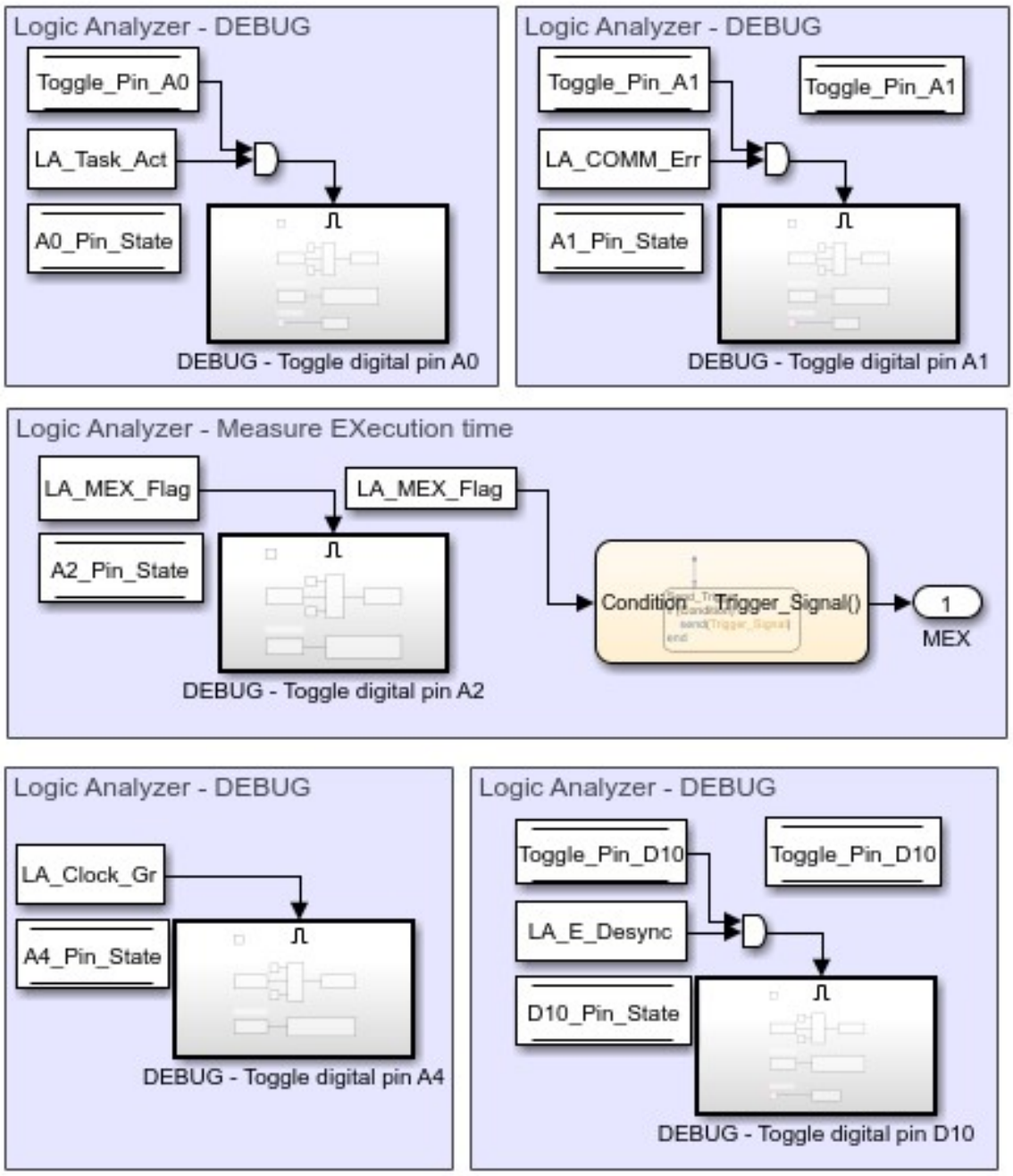


Figure 29: Logic analyzer subsystems.

The strategy followed for the measurements consists in toggling a digital output, so the time from one activation to the next can be measured with the logic analyzer. The measure execution time (MEX) measurement requires two different digital outputs to be activated, because the same digital output cannot be activated twice during the same run. The first MEX activation happens at the beginning of the TTA system activation, and the second happens at the end, and is located outside, with the CAN subsystems.

As an example, the inside of a logic analyzer subsystem is presented in figure (30). The pin state is toggled so the digital output changes between 0 V and 5 V each time the system is activated. The toggle pin flag is reset so during the next code activation it can be set again if necessary.

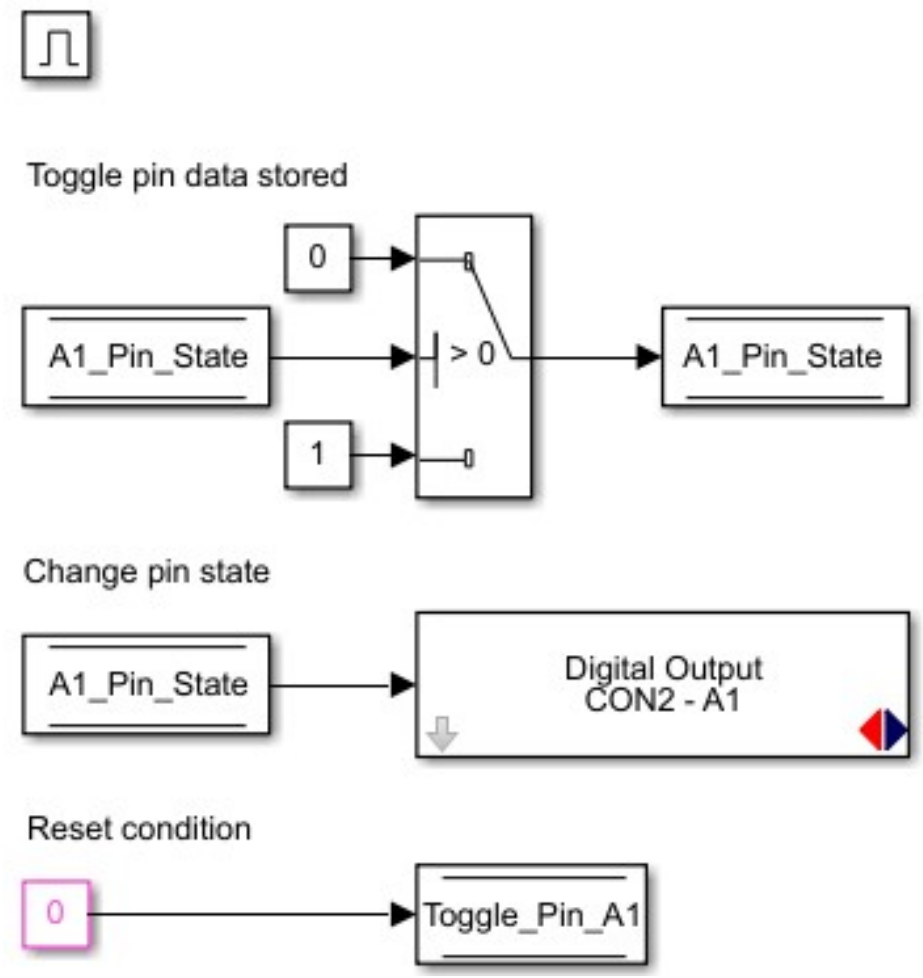


Figure 30: Example of a toggle of a pin state inside a logic analyzer subsystem.

2.14 Matrix cycle manager

The matrix cycle manager contains the matrix cycles for the controller, input generator and vehicle emulator. Here the correct matrix cycle is selected depending on the board ID value of the board. This number is unique, so each board will only execute one of these three matrix cycles. Inside the matrix cycles there are two basic cycles and each include the tasks from the TTA schedule. The decision on when to activate each CAN subsystem is made inside the basic cycles. The matrix cycle manager page is presented in figure (31).

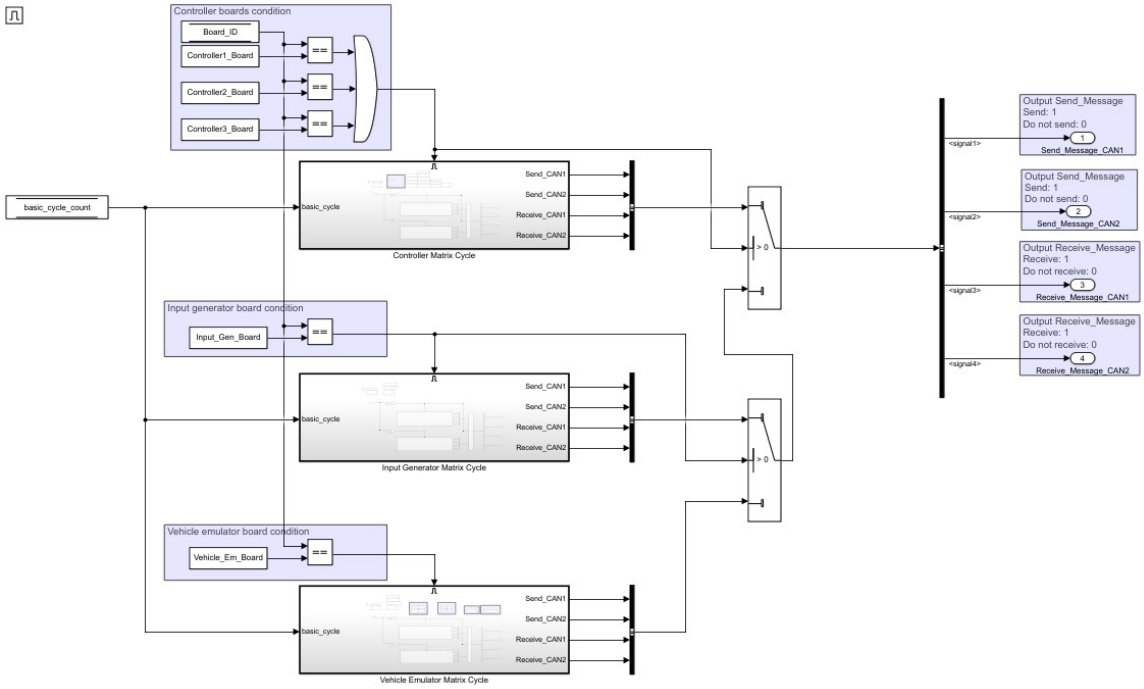


Figure 31: Matrix cycle manager subsystem.

2.15 Basic cycles

The TTA schedule of the prototype divides each matrix cycle in two basic cycles, the first one focused on board synchronization and the second dedicated to the controller operations. Both basic cycles in all the matrix cycles of this project have some operations in common. This section describes all these common tasks and leaves the concrete definition of each basic cycle for the next sections. The standard basic cycle selection window is shown in figure (32), depending on the current value of the basic cycle counter.

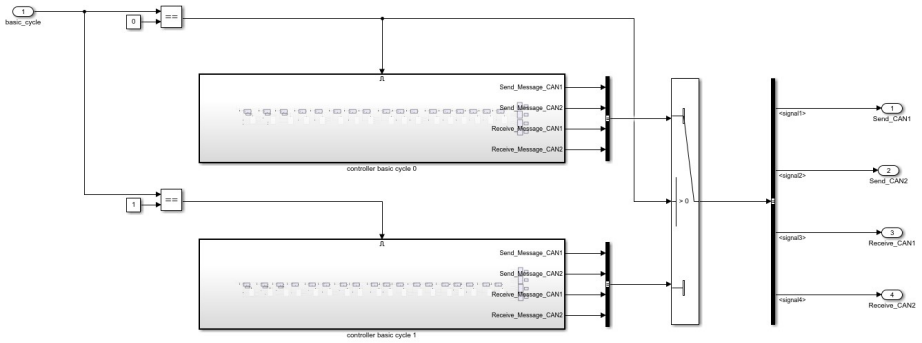


Figure 32: Matrix cycle system.

The basic cycles are divided into time windows in order to select the tasks. If a time window contains the current local time (Local_Ticks variable) when the basic cycle is activated, the task assigned to this local window is activated too. An example of time window is presented in figure (33). The time marks and COMM_Period values are defined in the MATLAB startup file.

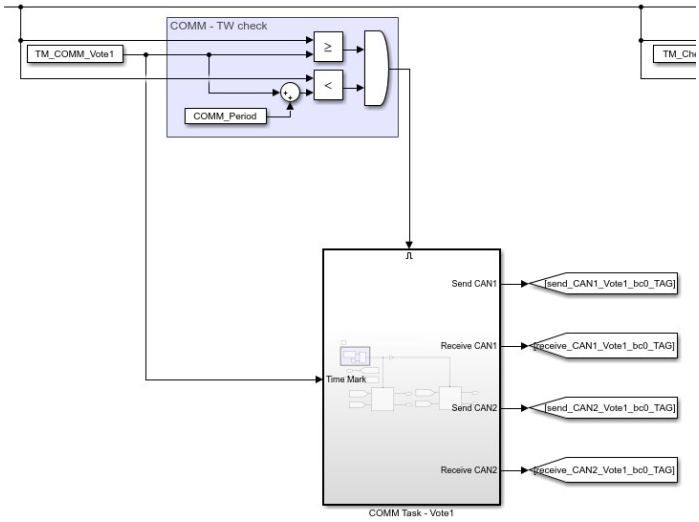


Figure 33: Time window example inside a basic cycle.

There are two main different kind of tasks, communication and computational tasks. All communication tasks are defined with the same characteristics, with only some special differences to decide what kind of message should be sent and what content the message shall include. Every computational task has a specific purpose. In the following subsections the common tasks to every basic cycle in the software are described.

2.15.1 Communication tasks - COMM

Every time a communication task appears in the TTA schedule this system is repeated. There are three differences in the communication subsystems that appear in the different communication time windows:

1. The transmission (Tx) condition.
2. The message ID.
3. The message value data content.

The first two elements are found in the first level of the hierarchy of the communication system, as presented in figure (34). The message value data content is part of the transmission subtasks.

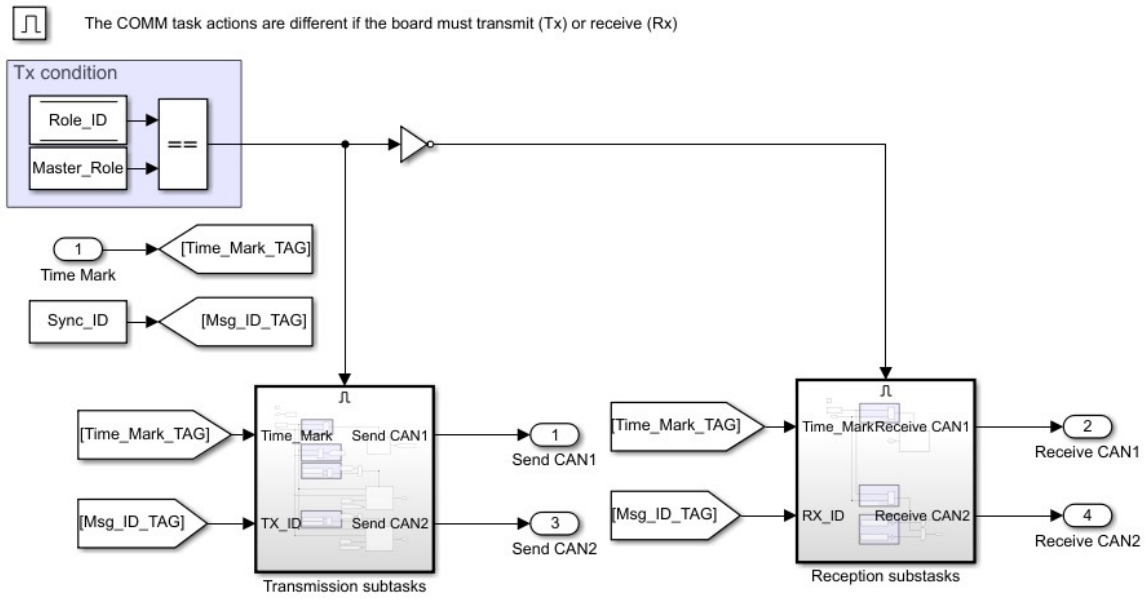


Figure 34: Communication system.

The tasks in communication are divided into reception and transmission subtasks. If the transmission condition is false, the board must listen for a message with the specified message ID during the communication time window. Otherwise, the message must transmit with the message ID and the appropriate data information.

The reception window is presented in figure (35). If a board has to listen to a message, at the beginning of the task it will update its reception buffers, so it gets to know to what ID it must listen to and the message buffers are reset. During the next ticks during the communication time window, if no message has been received yet in the corresponding channel, the receive CAN signals to activate the reception CAN systems are set to true.

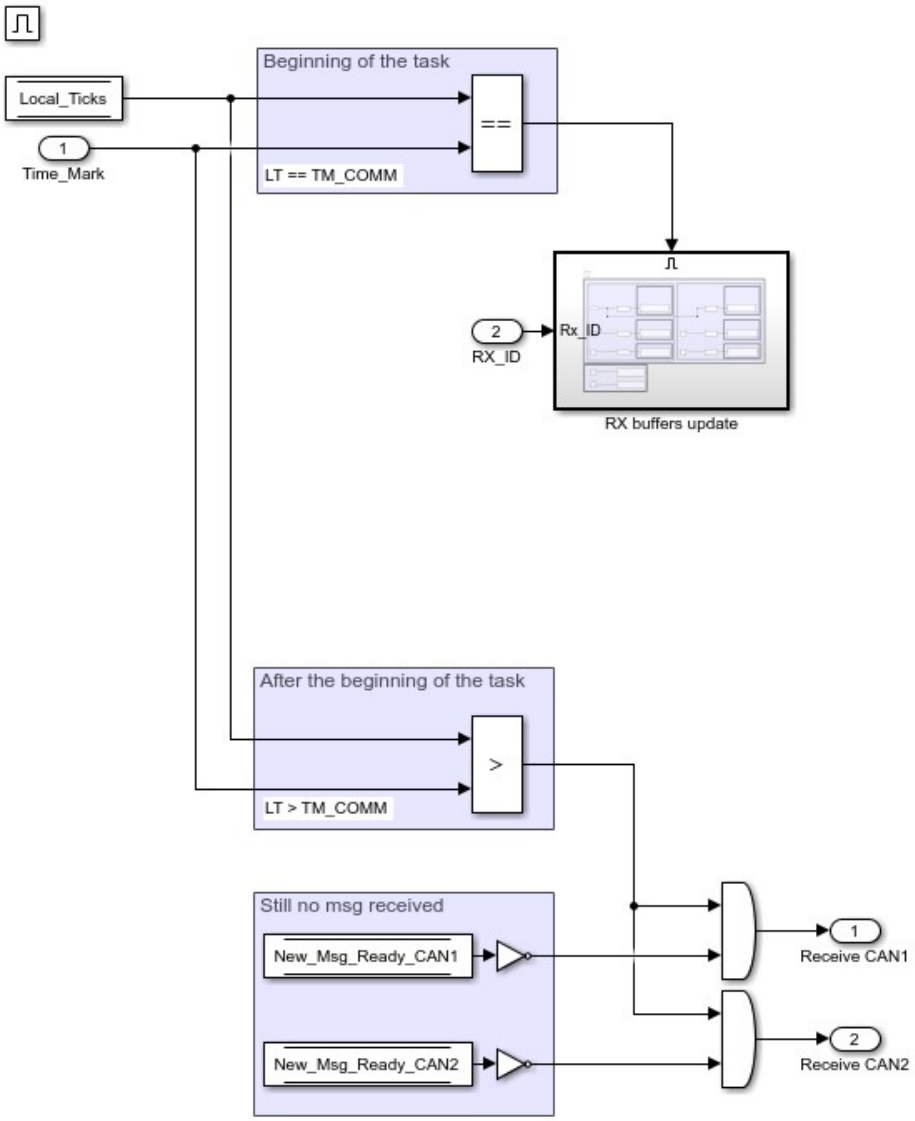


Figure 35: Reception subsystem in the communication tasks.

Every reception buffer updated is presented in figure (36). Both CAN channels get their buffers reset. New_Msg_Ready_CAN declares if a message has been received at any channel. The reception state machine is governed by the Rx_State_CAN variable. The message content is saved in Msg_Rx and new_msg_Rx is the flag setting up if a coherent message has been received.

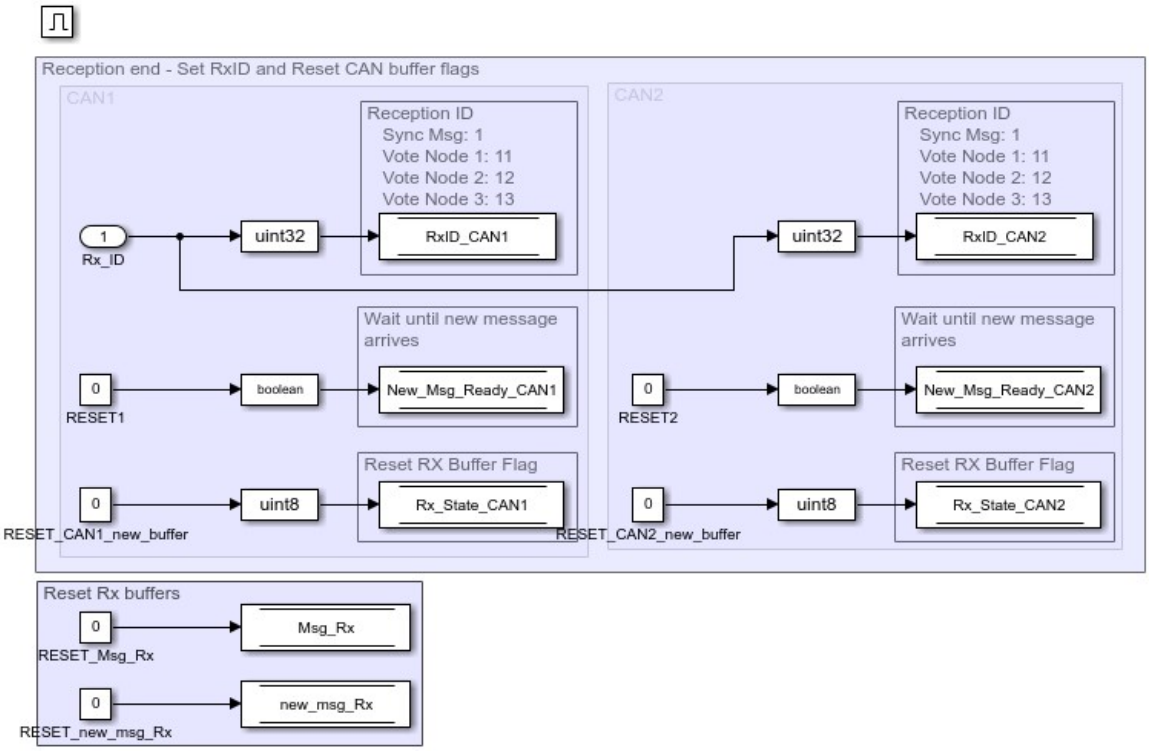


Figure 36: Receive buffers reset subsystem.

When the receiving CAN system is activated it starts at Rx_State_CAN equal to zero. The system must clean the CAN message buffer first, and then start listening without messages in the CAN buffer. When the first message with the specified ID is received, the New_Msg_Ready_CAN variable is set (New_Msg_Ready_CAN1 or New_Msg_Ready_CAN2, depending on where the message was received). From that moment on the CAN receive system for that channel is not activated again until the next communication task. The received message is stored at Msg_Rx_CAN1 or Msg_Rx_CAN2 and postprocessed in the next task of the schedule: the communication check task. There the received messages at CAN1 and CAN2 are compared and if they are coherent new_msg_Rx is set. Lastly, the coherent message is finally stored in Msg_Rx for further use in the next computational tasks.

The transmission task starts by updating the message value data variable and encoding the appropriate information. Then, during the next ticks the program checks if a new message must be sent or not. The parameters that state when a message is sent through each CAN channel during a communication task are the initial phase and phase, defined in the MATLAB startup file. The initial transmission task window is presented in figure (37).

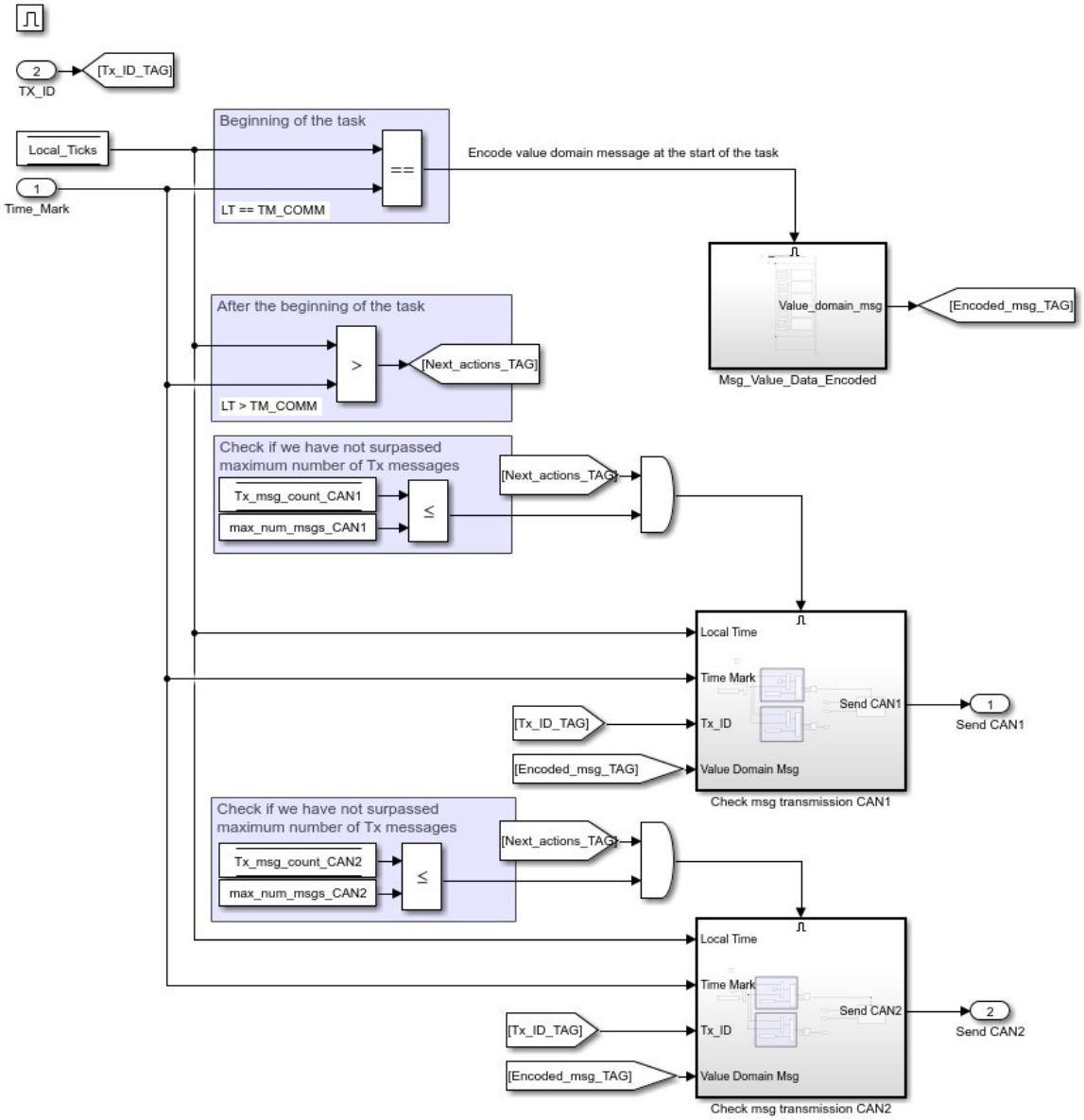


Figure 37: Communication transmission subtask.

Most of the data transmitted in messages are float variables. Because we can only send bytes of information in every message slot, the floats are encoded using some bits for the integer part and others for the decimals, declared by the precision value. The first message buffer is always reserved for temporal information of the message. An example of data encoding is presented in figure (38).

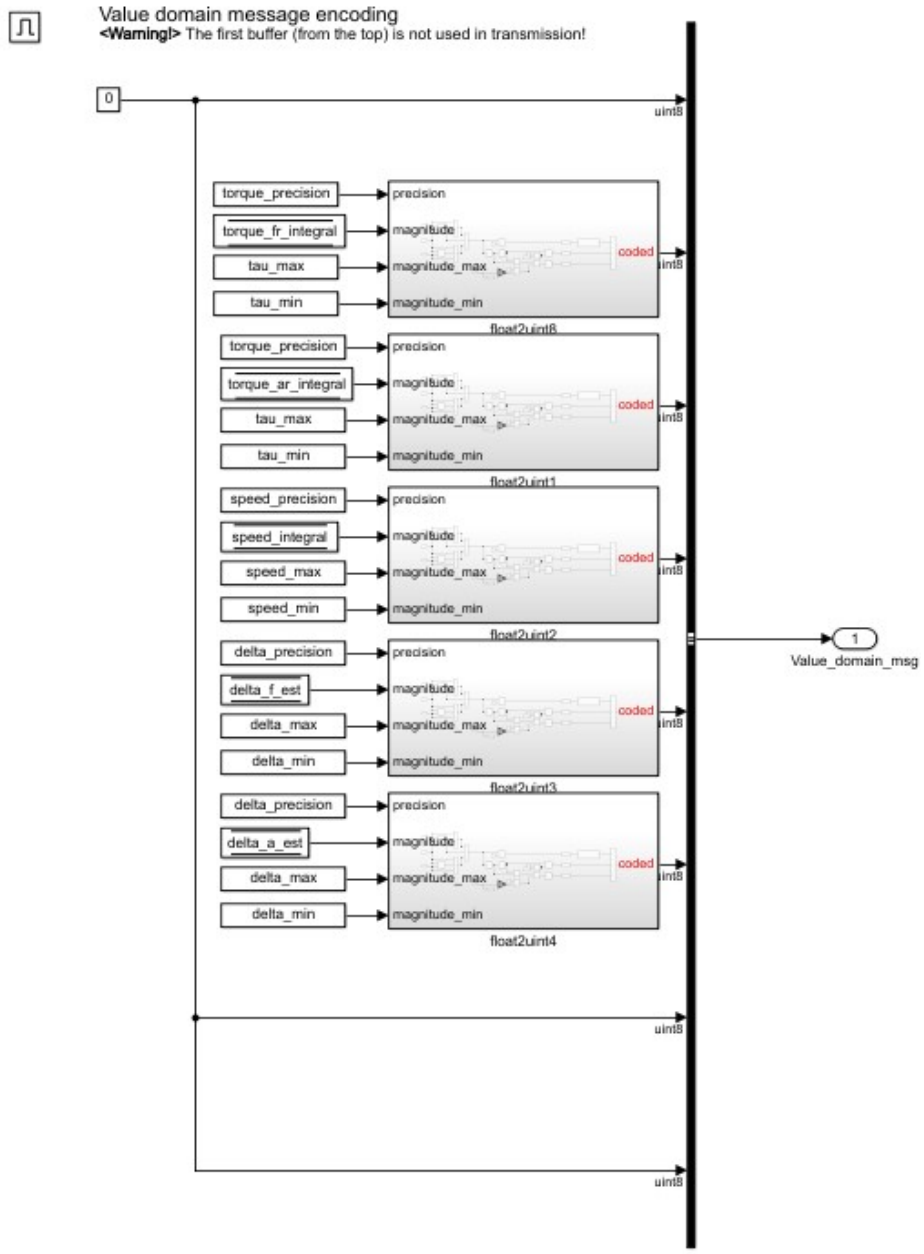


Figure 38: Example of message encoded for a transmission.

From the moment a communication task starts an extra COMM_Phase_init time in ticks is waited before sending the first message. From then on a message is sent every COMM_Phase ticks. This is checked as shown in figure (39), updating the transmission buffer the tick before the transmission happens and activating the CAN transmission systems in the appropriate tick.

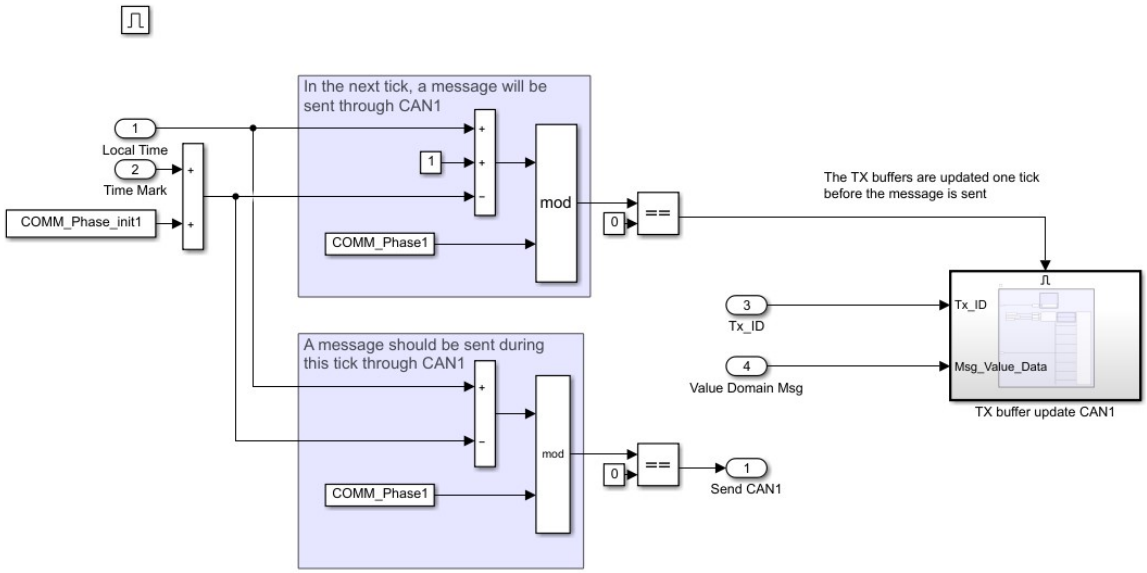


Figure 39: Check transmission in communication task subsystem.

The checks are done using the mod operation. If the amount of ticks that have already happened since the COMM_Phase_init tick is a multiple of COMM_Phase, then it is the appropriate time to activate the send CAN system.

The transmission buffers contain information about the ID of the message, and the message content. The most complex operation performed in this subsystem is the temporal information encoding. The first byte of every message contains the basic cycle counter at which the message was sent, the message count and the board ID of the transmitter. A maximum of seven messages can be sent during a communication task with the current build of the code. This sets one bit for the basic cycle (0 or 1), three bits for the message counter (0 to 7) and four bits for the board ID (0 to 15). The board ID was decided to count with four bits space to allow for further expansion of the system with more boards. The transmission buffer update is presented in figure (40).

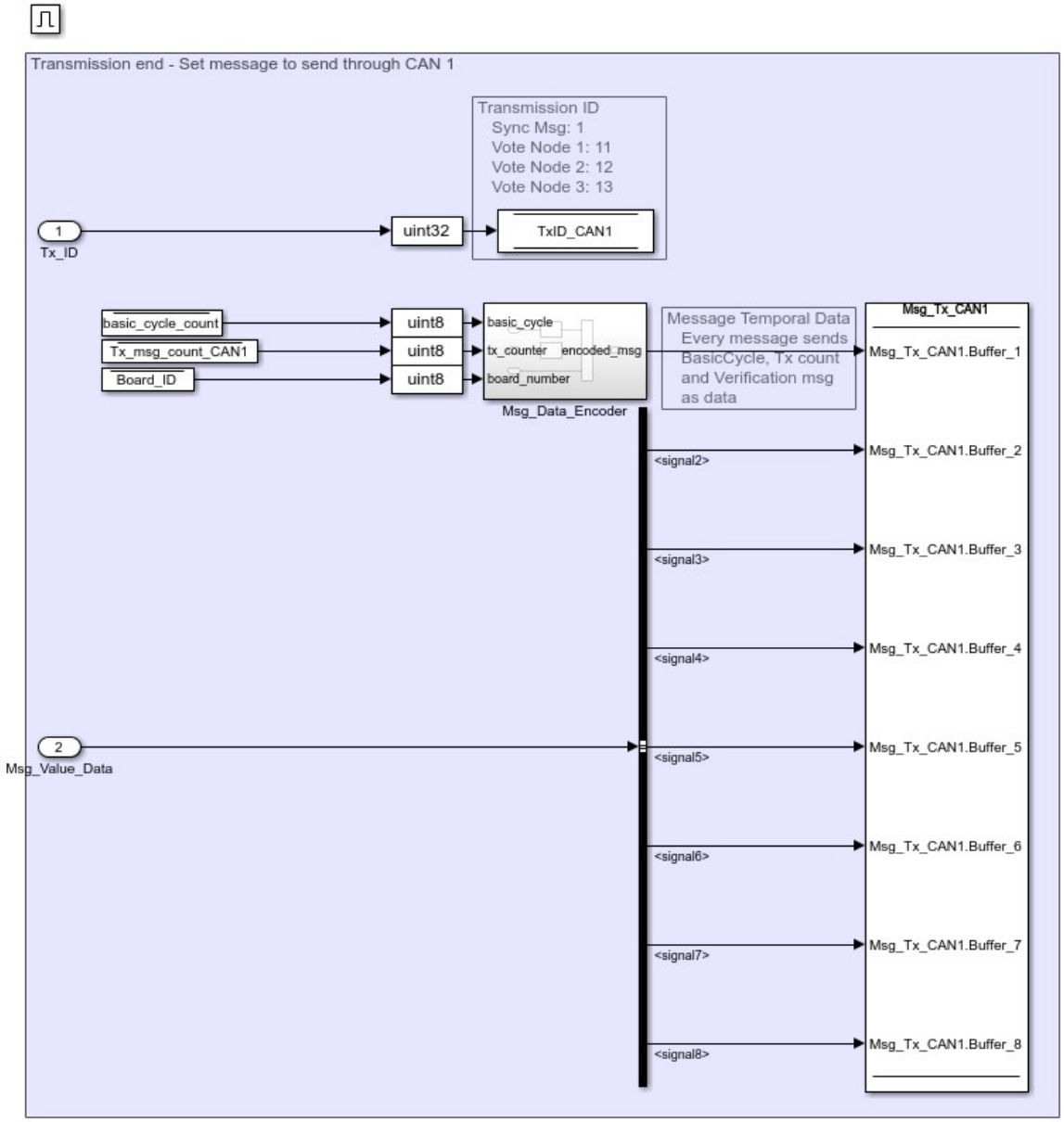


Figure 40: Update transmission buffers subsystem.

2.15.2 Communication check tasks

After every communication task comes a communication check. This task checks whether a message was received at any of the CAN channels. If so it prepared the message buffer so the information received can be further processed in later tasks. As it can be seen in figure (41), a board receiving a message during the previous communication task activates the process messages, while a board that transmitted a message resets its own message transmission counters.

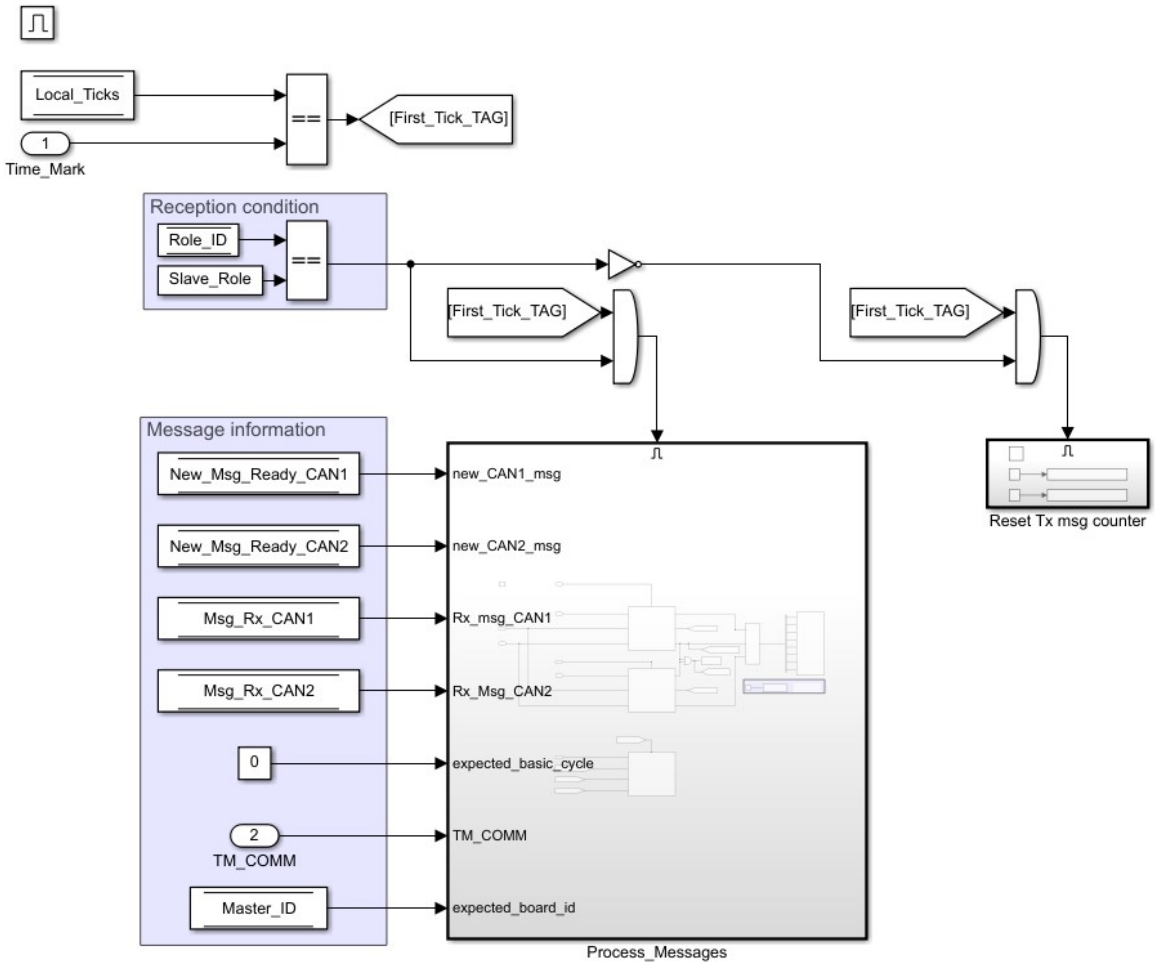


Figure 41: Communication check computational task system.

In the process message system presented in figure (42), the message buffers from CAN1 and CAN2 are checked. If any of the two contains information, the new_msg_Rx flag is set and the Msg_Rx buffer is filled with the received information.

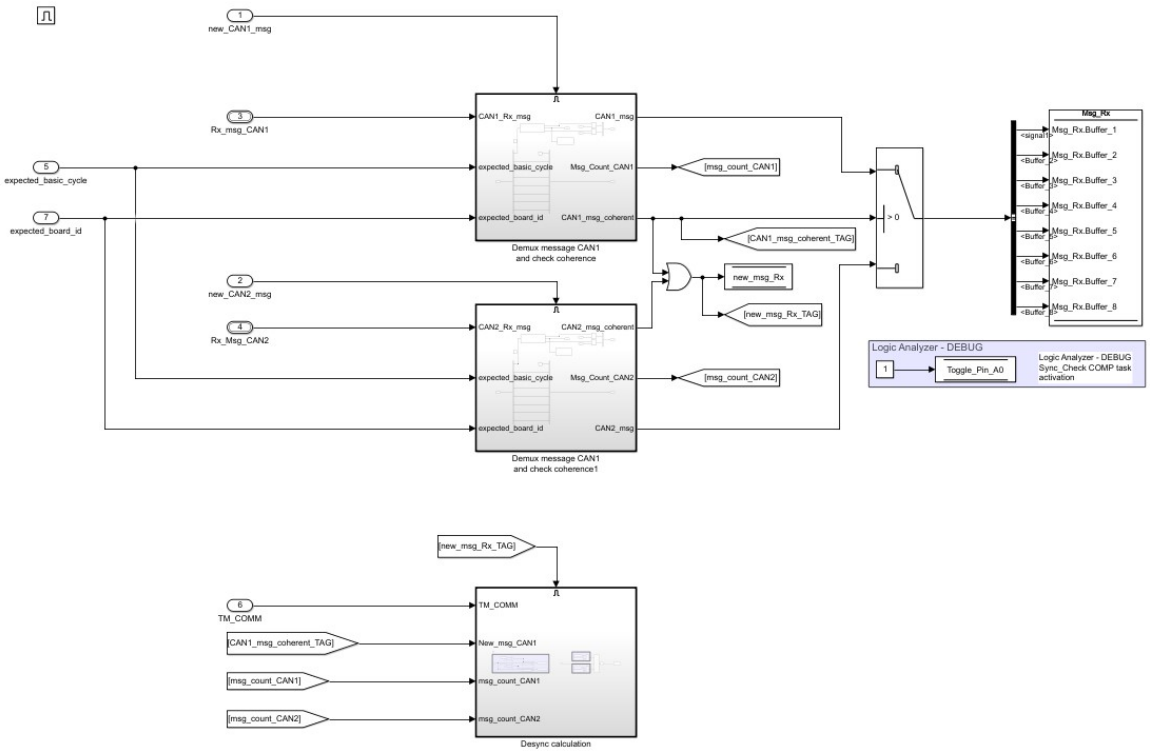


Figure 42: Process message system inside the communication check.

The reference messages at the beginning of every basic cycle contain an extra subsystem: the desync calculation. The difference between the time when the message was received and the time when the message was expected to be received is calculated using the message transmission counter. This desync is further processed later in the local time update task.

The demux coherence subsystems decode the temporal information of the message and check if the basic cycle and board ID received are the ones expected. It can be seen in figure (43).

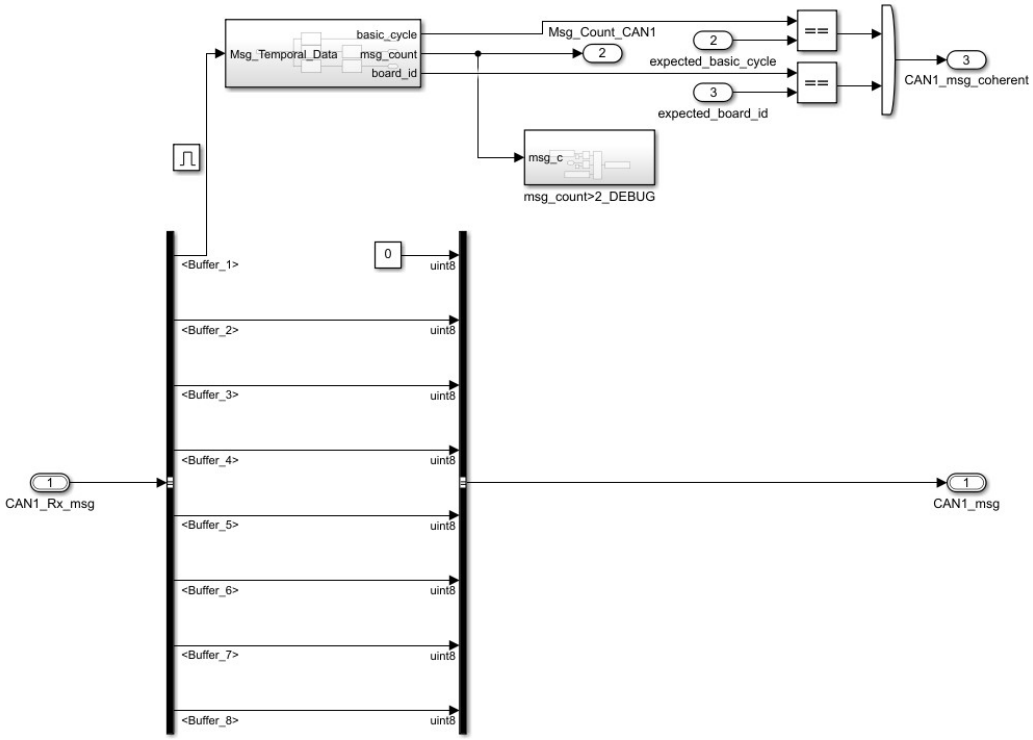


Figure 43: Demux coherence subsystem.

The desync calculation for the reference messages is presented in figure (44). Because every board is aware of the common TTA schedule in the system, they can calculate when the message was expected to be received. For this, the communication delay (which is a measurement done with the logic analyzer) and the message transmission counter (that comes with the temporal information of the message) are taken into account. This expected value is compared with the moment at which the message was actually received, recorded by the CAN receive system in the Msg_Rx_Ticks_CAN variables. The desynchronization value is saturated at 15 ticks, as a high desynchronization value caused by an unexpected too long communication delay could make the board to change its local time to a wrong local time too different to the rest of the ensemble, provoking the board to completely desynchronize.

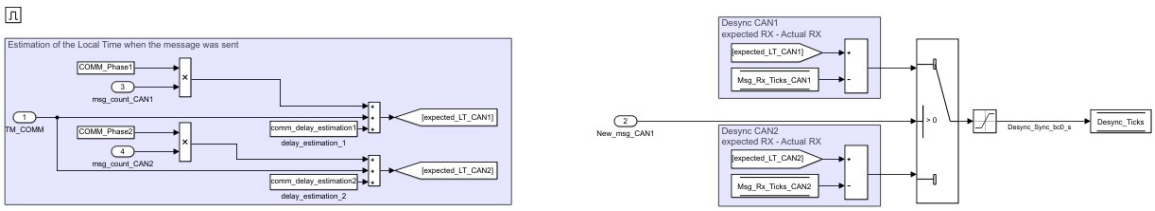


Figure 44: Desync calculation.

2.15.3 Update local time

All basic cycles start with a reference message, a check of the communication task and an update of the local time of the board. During the local time computation task the board corrects its own local time to make it closer to the master's local time using the information of the desync. The update local time system is presented in figure (45). Before the local time update a desync debug system is activated for a logic analyzer measurement.

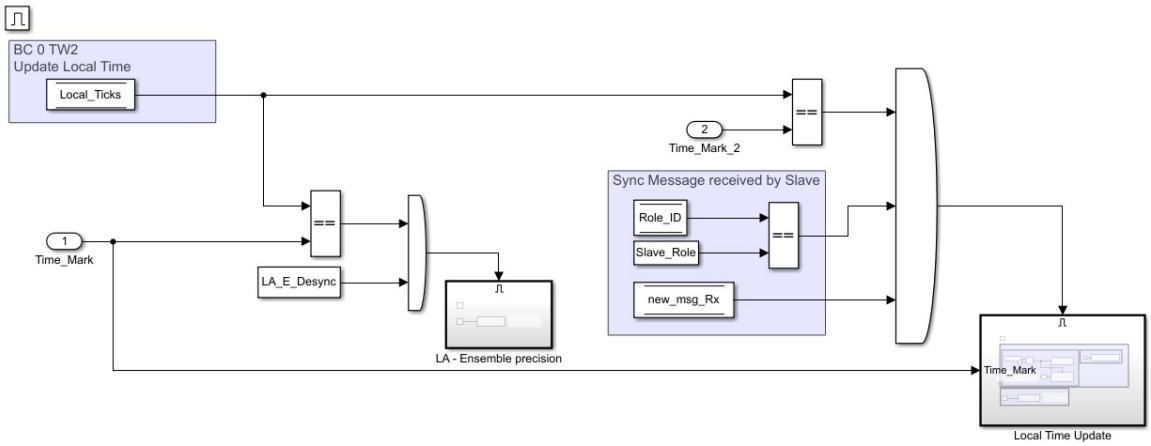


Figure 45: Update local time system.

The main update local time subsystem checks whether the desync calculated in the communication check is positive or negative. If the desync is positive the local time shall be updated in small steps in the desync positive system in the TTA System hierarchy, so no tasks are skipped by taking the local ticks into the future. When the desync is negative, the Desync_Negative subsystem is activated, as can be seen in figure (46). The variable BC0_Sync_processed is set to prevent that a previous already processed task is performed again when the local time is updated with a negative desync.

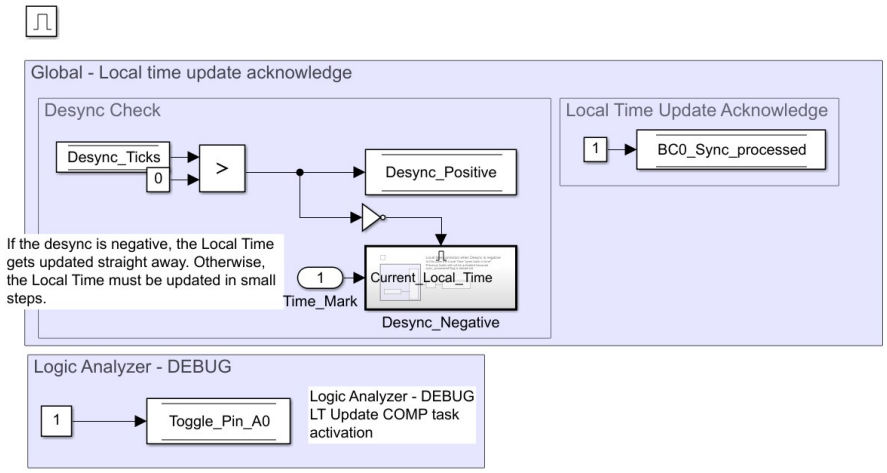


Figure 46: Desync sign check.

The local time is updated in the Desync_Negative subsystem. Here the local ticks are decreased to a "previous moment in time". Every task before this point has a Sync_processed flag controlling its activation, to make sure no task is repeated after the local time is updated to a lower value. Figure (47) shows one of the few places in the code where the local time is rewritten.

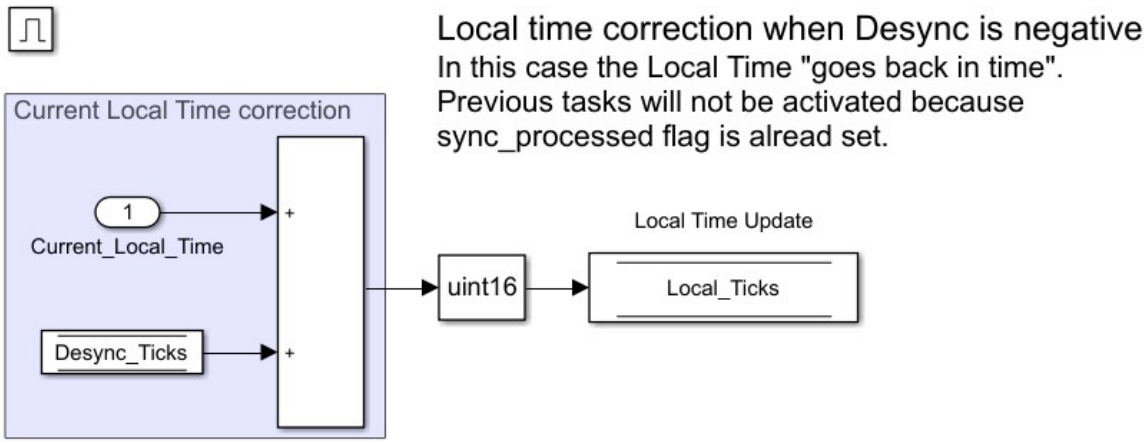


Figure 47: Local time update in the Desync_Negative subsystem.

2.15.4 Check timeouts

Before any of the boards is reset a check timeouts task checks if any of the communication tasks performed passed without receiving any message. In figure (48) an example with the missing message counter for vote 3 and sync 0 and the error flag for the current master is presented. The timeouts

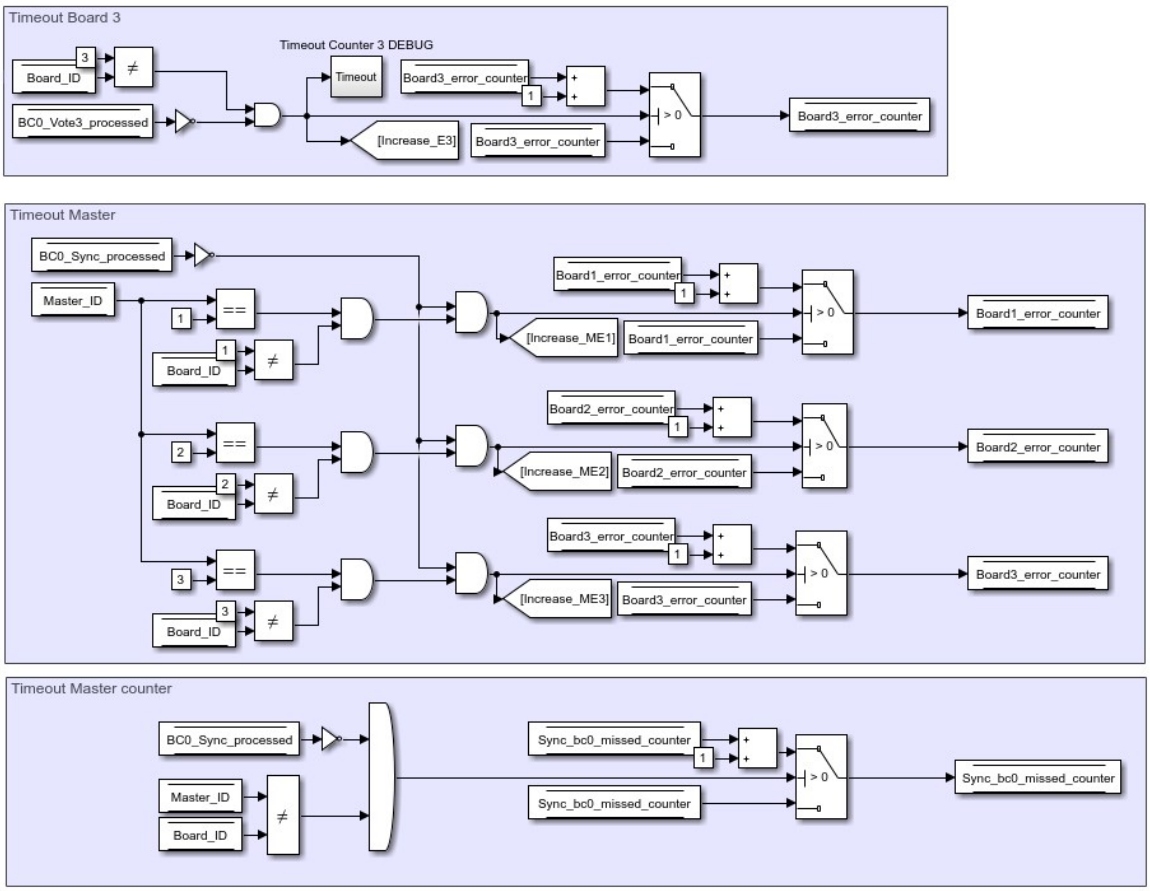


Figure 48: Timeouts processed example.

counters are used for two main purposes, check how many messages were missed to make the missing messages measurement and keep track of how many failures each board has made for the master decision at the New_Master task in the controller matrix cycle.

2.15.5 Reset variables

Other important computation task before starting the next basic cycle is the Reset_Variables system. As shown in figure (49) , here the main cycle variables required to make the schedule progress adequately are reset to be ready to start the cycle again later. Some basic cycles need to reset more variables than others.

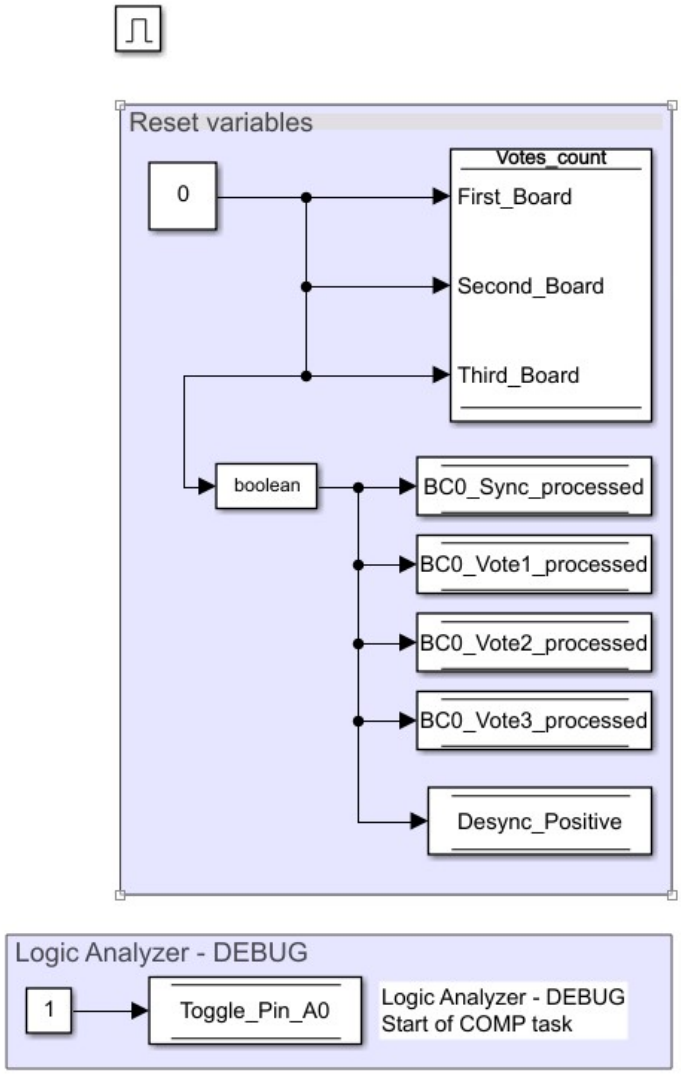


Figure 49: Reset variables system.

2.15.6 Reset board

If a board has not received any message during the different communication tasks of the schedule it performs an auto-reset. This means that it returns to a state in which no role is assigned yet, and the initialization needs to be done again. In figure (50) the reset variables to reset a controller board during the basic cycle 0 are presented.

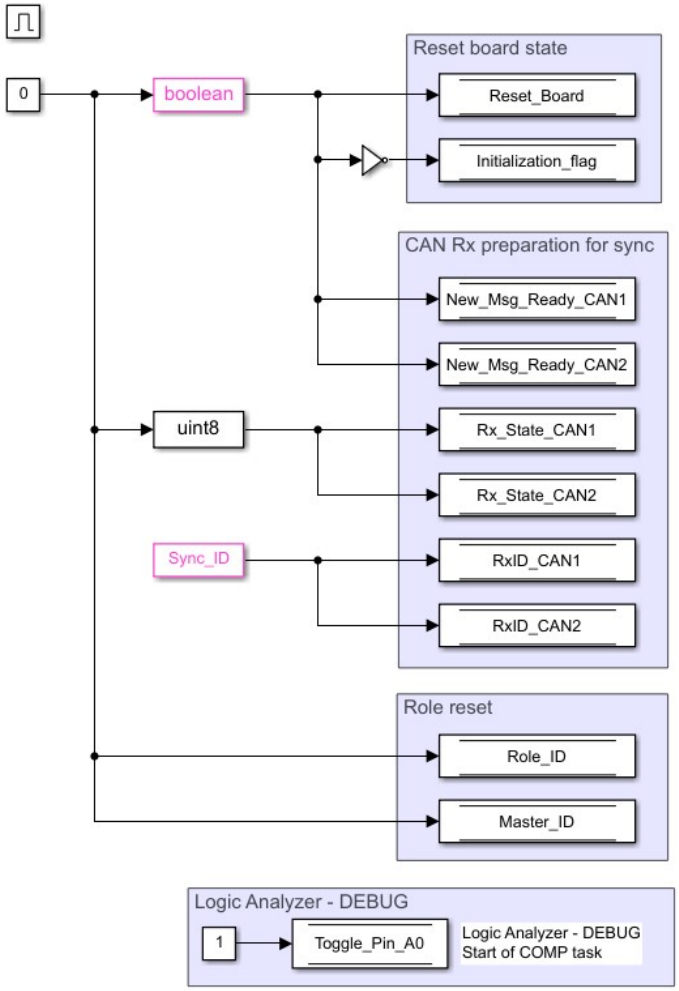


Figure 50: Reset board system.

2.16 Controller basic cycle 0

The first basic cycle in the controller boards is presented in figure (51). Most of the tasks that appear in the cycle have already been covered in the common tasks for all basic cycles. In this section the vote decision, the message value encoded in the vote communication tasks and the selection of a new master are explored in detail.

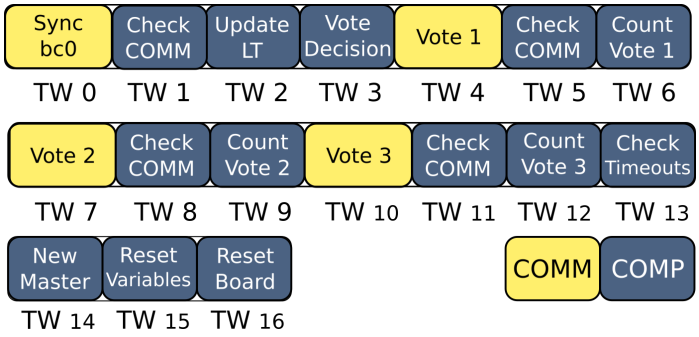


Figure 51: Basic cycle with the communication (COMM) and computation (COMP) tasks of the first controller basic cycle.

2.16.1 Vote decision

The first special task that is processed in the controller is the vote decision. The board must decide which board shall be the time master of the ensemble during the next basic cycle. In figure (52) the decision process is shown. If the reference message from the master has been successfully received (BC0_Sync_processed is set) then the current master is selected to continue being the master. Otherwise, a new board is selected taking into account the error counters from the timeouts system in section 2.15.4.

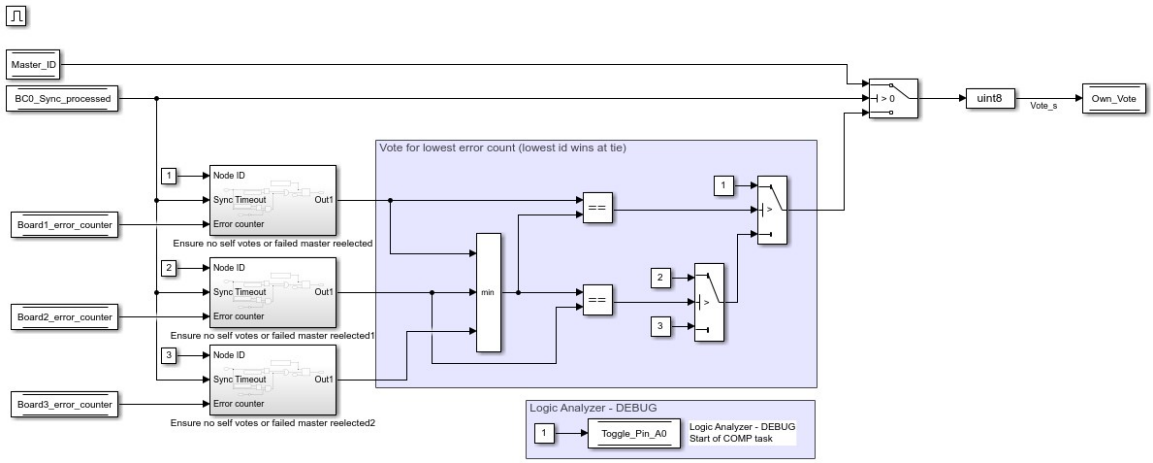


Figure 52: Vote decision system.

If a different board than the current master shall be selected as the new master, the vote decision conditions from figure (53) are taken into account.

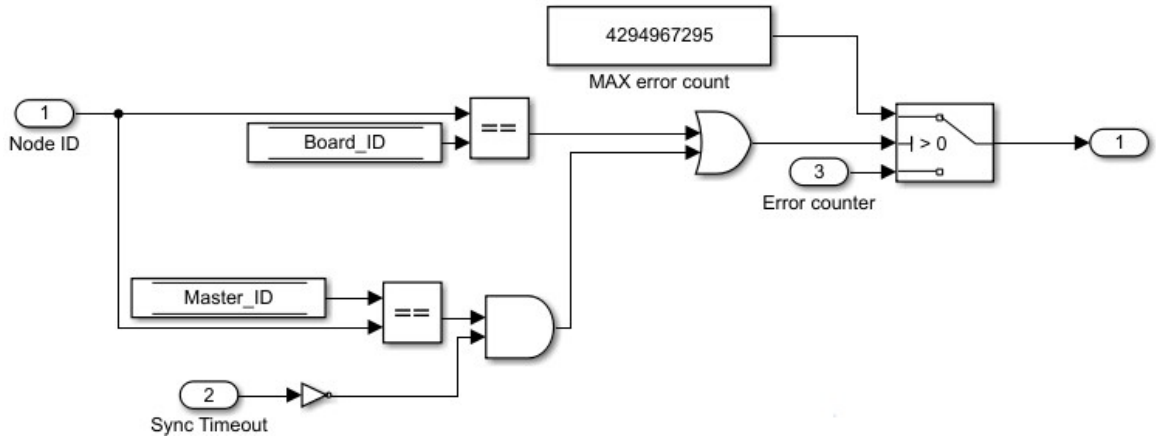


Figure 53: Conditions for vote decision.

The system in figure (53) is going to output a number from 0 to the maximum error count. If the Node ID that is being evaluated is either the current master or the board that is doing this operation, the maximum error count is selected as the output. Otherwise, the error count of the board associated to the Node ID is the output. This ensures that a board does not vote either for a master that has failed nor for itself. When looking at figure (52) again, it can be seen that the lowest output from the three filters evaluating the three controller board IDs is selected, choosing the associated controller ID as the new master.

As an overview, if the master reference message is received, the same board is again selected as the master. In other case, a board will vote for the board with the lowest error count, paying attention that it does not select the failed master nor itself as the new master.

2.16.2 Votes - COMM

The information sent in the communication task with the vote decision is the own vote of the board, previously selected at the Vote Decision task. The own vote is stored in a uint8 variable. so no encoding is needed, as can be seen in figure (54).

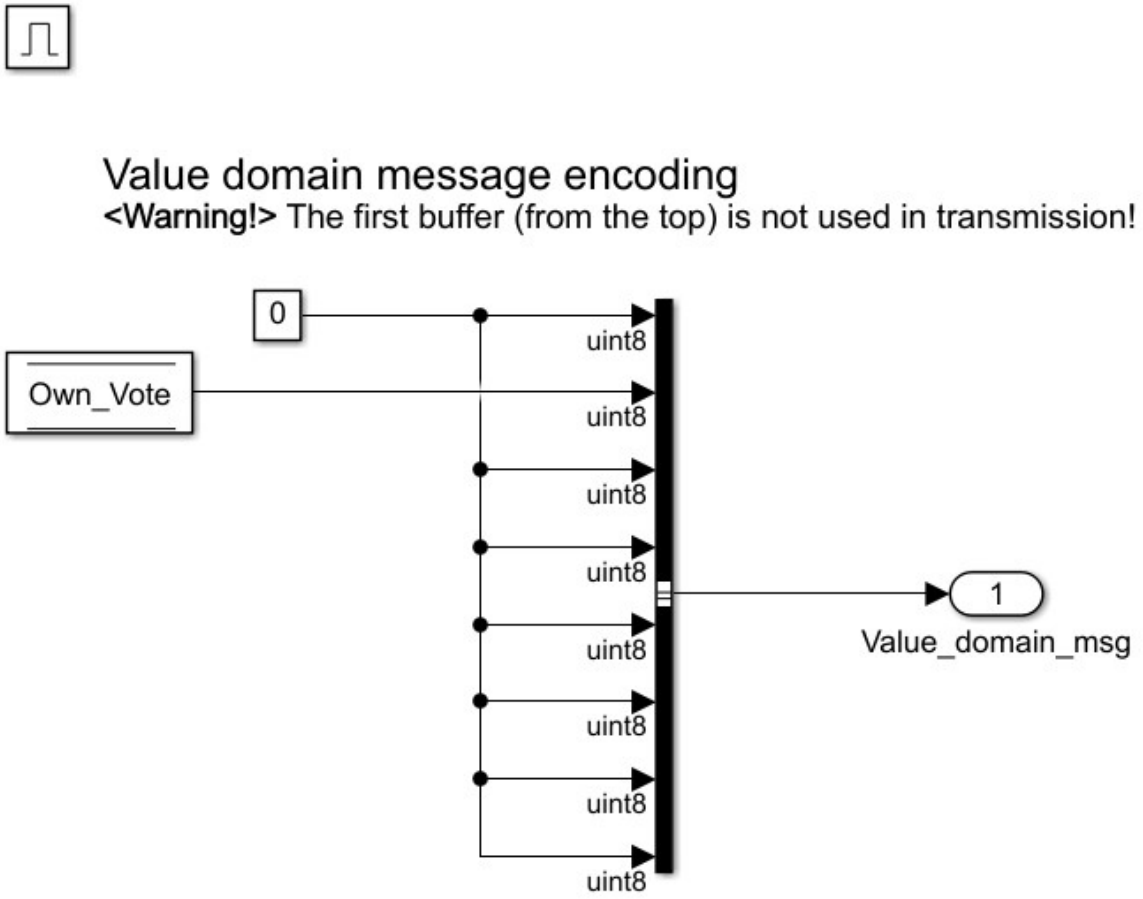


Figure 54: Message value data encoded subsystem in the vote communication tasks.

2.16.3 New master

Depending on the votes received by the other boards and the own vote of the board the time master ID is updated. Also, if the board that is evaluating this task is the new master, it changes its role. If it was master before and it is not any more, it changes its role too. A board that does not receive any message from any other board during the communication tasks of a basic cycle sets the Reset_Board flag. This is presented in figure (55).

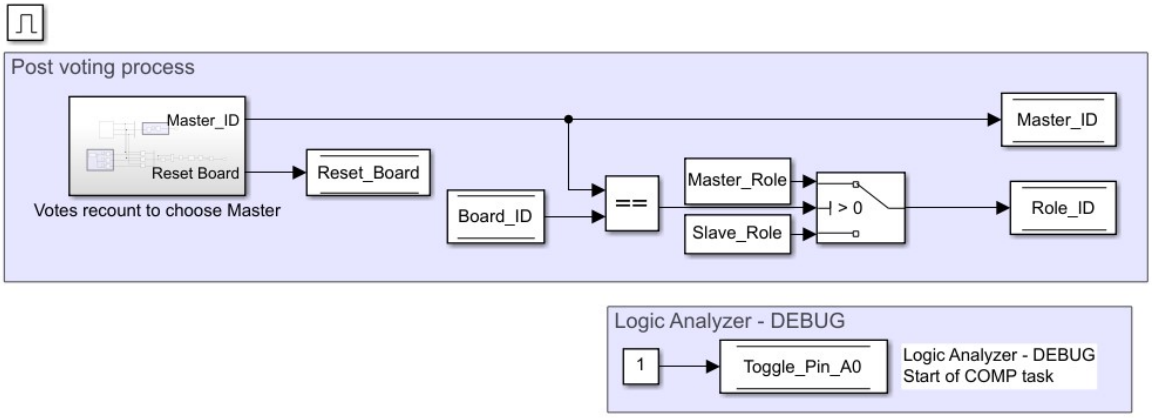


Figure 55: New master system.

The vote counting process is presented in figure (56). The top part checks if there was any message received, so the reset flag can be set otherwise. In the bottom part every vote is summed up, taking also the own vote into account. The results for each board are merged together in a bus, from which the maximum number is selected. In case of a draw, the board with the lowest ID is chosen.

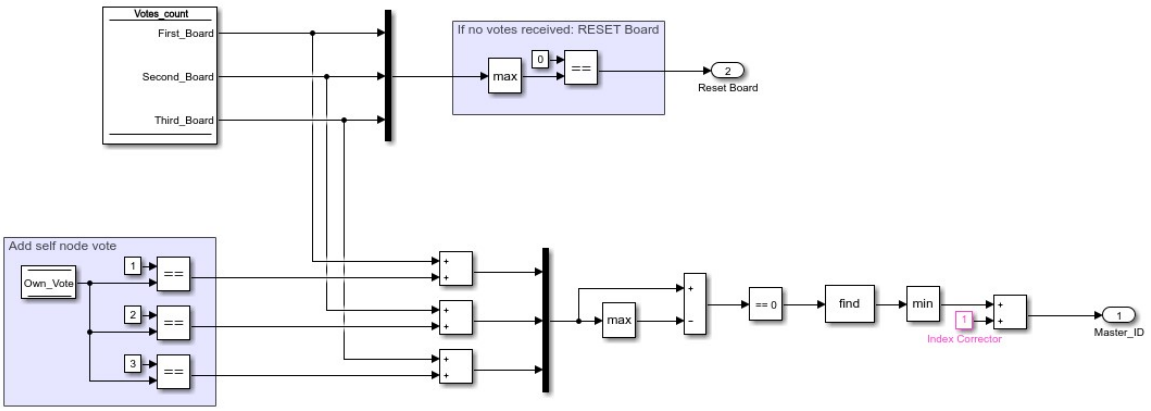


Figure 56: Vote count subsystem.

2.17 Controller basic cycle 1

The second controller basic cycle is in charge of the controller operations. All the tasks are presented in order in figure (57). The cycle starts again as it was presented before, with the reference message and the local time update. The set values from the input generator and the sensor values from the vehicle emulator are received. Then, the output torque to the vehicle actuators is calculated in three different tasks: steer, torque and velocity. Each slave board send their respective results and the time master performs the triple modular redundancy (TMR) operation. Lastly, the output value processed in the TMR is sent to the vehicle emulator and the cycle variables are reset.

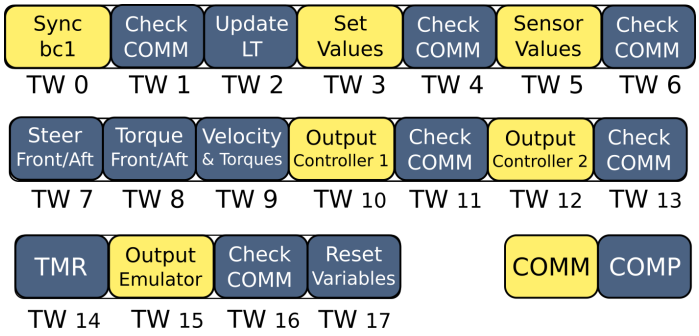


Figure 57: Basic cycle with the communication (COMM) and computation (COMP) tasks of the second controller basic cycle.

2.17.1 Set values - COMM

The set values are transmitted from the input generator. The controller board checks if the message was received and if it is, the data is stored in the Set Values Update subsystem from figure (58). If no message was received during the set communication task the controller calculations are not performed in the current version of the software. The flag variable Error_SetValues_NotRX is set when this happens to keep track of this issue.

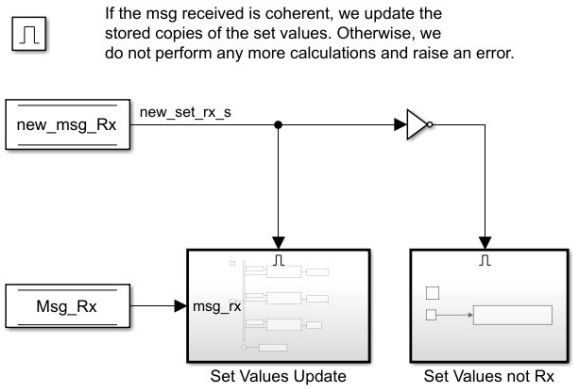


Figure 58: Set values check subsystem, with the update of the set buffers or the set error flag activation.

As presented in figure (59) the message received is decoded from uint8 information to floats. The steer angles for the front and aft axles are stored and processed, but in the current version of the software only the speed loop works. Further research in the two-axle vehicle controller is required to make the steering control work as well.

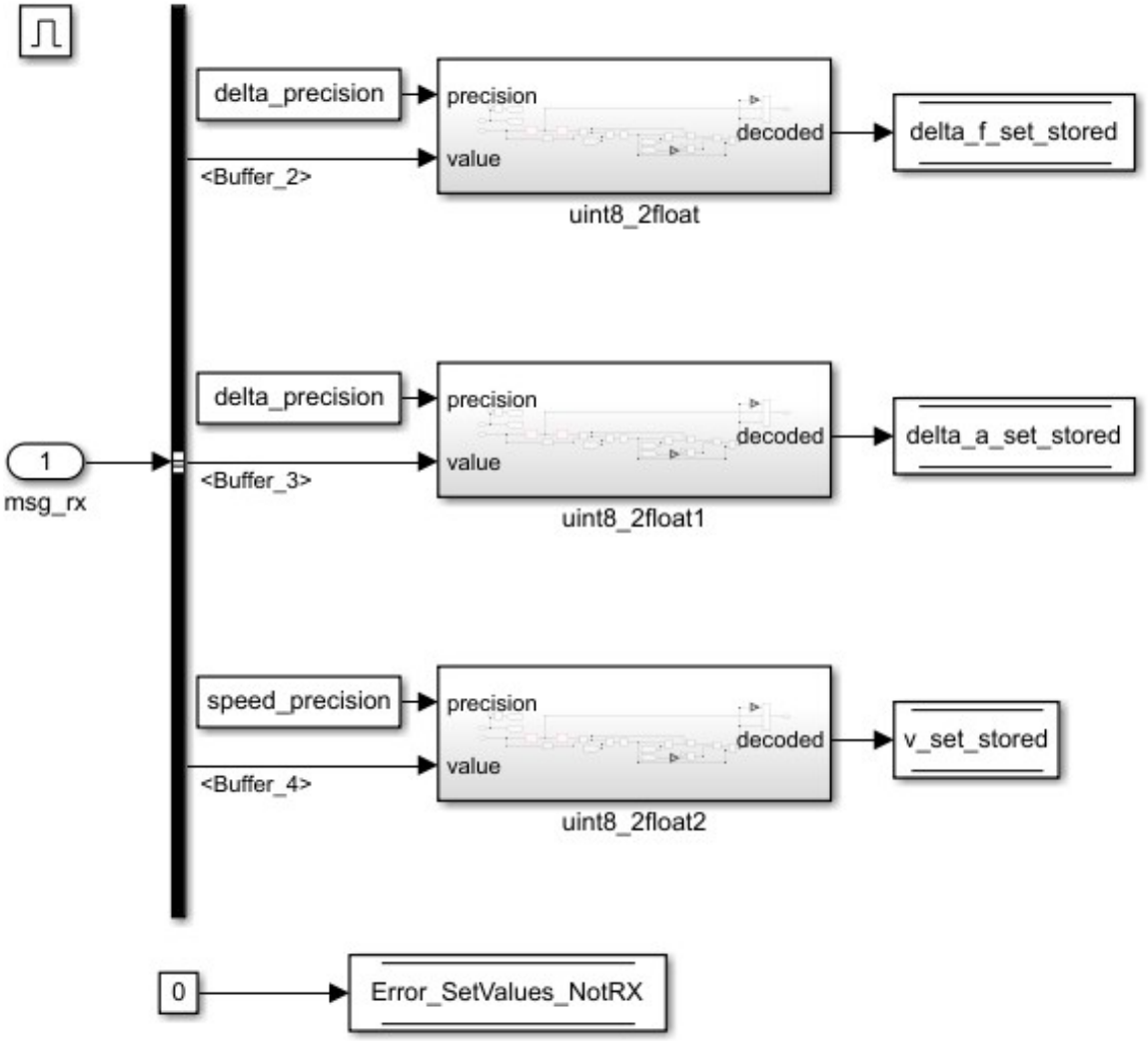


Figure 59: Set values decoded and stored.

2.17.2 Sensor values - COMM

If the sensor values are received in the sensor communication task, their data is decoded and stored. If the set values were also received, then everything was okay in the communication side and the controller calculations are later performed. The main page in the sensor values received is presented in figure (60).

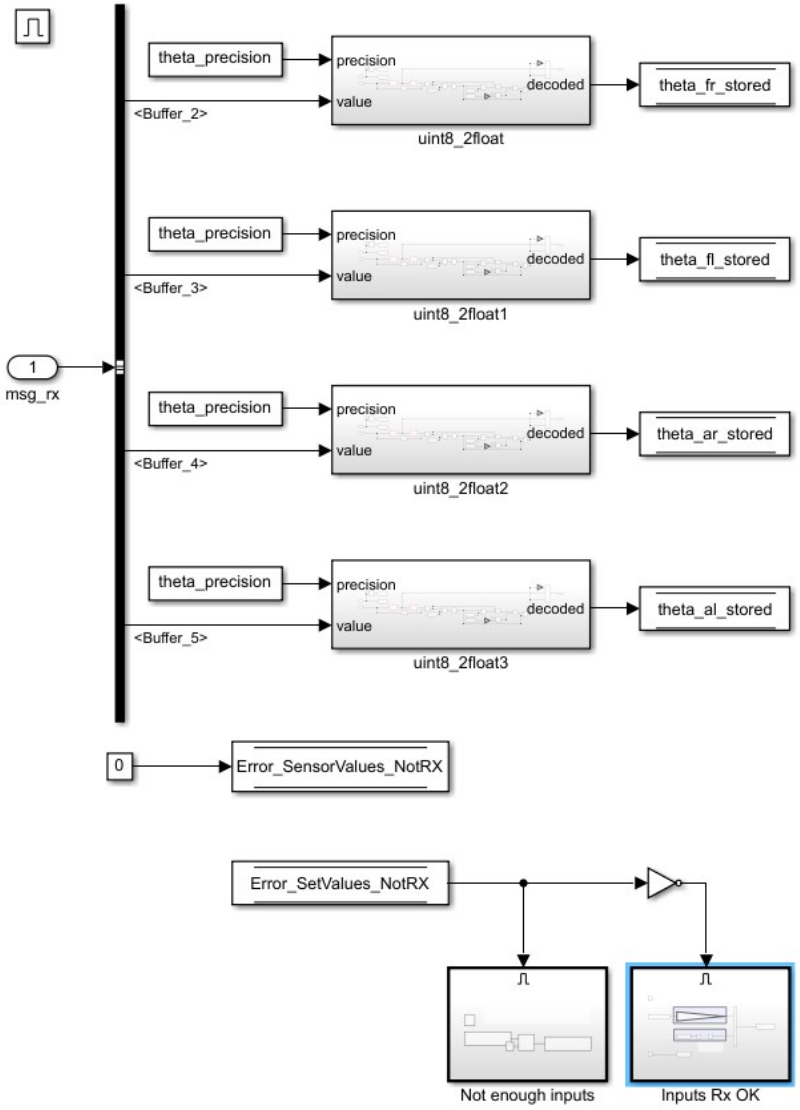


Figure 60: Sensor data decoded and stored.

When all of the input values are received in the controller the controller cycle time is stored in rx_data_incr_time. This is the time that has passed since the last time the controller calculations were performed. The mc_counter variable keeps track of every cycle in which the set or sensor values were not received. This variable is deprecated in the current version of the software, as the prototype operation design states that the system remains stationary when communication fails between the different parts of the system. This means that rx_data_incr_time always has the same value, the matrix cycle period.

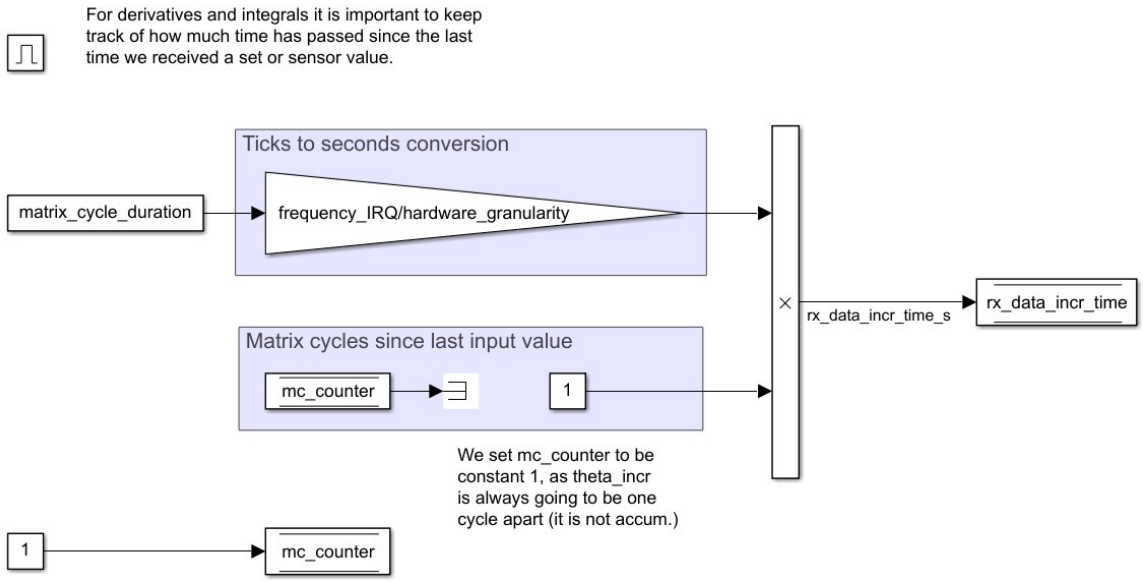


Figure 61: Inputs to the controller received subsystem, with the controller cycle time update.

2.17.3 Controller calculations

The controller calculations are divided upon three different computation tasks to avoid processor overload. They start estimating the steer front and aft angles from the wheel angles information provided by the sensors.

$$\begin{aligned} \delta_{f\,est} &= \int \frac{(\dot{\theta}_{fr\,act} R_{fr} - \dot{\theta}_{fl\,act} R_{fl})}{T_{lf}} \mathrm{d}t \\ \delta_{a\,est} &= \int \frac{(\dot{\theta}_{ar\,act} R_{ar} - \dot{\theta}_{al\,act} R_{al})}{T_{la}} \mathrm{d}t \end{aligned} \tag{1}$$

where $\delta_{f\,est}$ and $\delta_{a\,est}$ are the steering front and aft angles, respectively. Every $\dot{\theta}_{act}$ corresponds to the angle rate of one of the wheels and the R variables are the radii from each wheel to its axis. T_{lf} and T_{la} are the torques at the front and aft of the vehicle.

With the estimation of the steering angles it is already possible to calculate the torque at each wheel with the first PID controller.

$$\begin{aligned} \tau_{fr\,out} &= k_p(\delta_{f\,set} - \delta_{f\,est}) + k_i \int (\delta_{f\,set} - \delta_{f\,est})\mathrm{d}t + k_d \frac{\mathrm{d}(\delta_{f\,set} - \delta_{f\,est})}{\mathrm{d}t} \\ \tau_{fl\,out} &= -\tau_{fr\,out} \\ \tau_{ar\,out} &= k_p(\delta_{a\,set} - \delta_{a\,est}) + k_i \int (\delta_{a\,set} - \delta_{a\,est})\mathrm{d}t + k_d \frac{\mathrm{d}(\delta_{a\,set} - \delta_{a\,est})}{\mathrm{d}t} \\ \tau_{al\,out} &= -\tau_{ar\,out} \end{aligned} \tag{2}$$

where each τ_{out} variable is the torque at each corresponding wheel, k_p, k_i and k_d are the proportional, integral and derivative gains of each controller and δ_{set} is the steering angle set by the reference generator, either at the front or at the aft.

The inputs to the vehicle controller also allow it to estimate the speed of the vehicle v_{est}.

$$v_{est} = \frac{\dot{\theta}_{fr\,act} R_{fr} + \dot{\theta}_{fl\,act} R_{fl} + \dot{\theta}_{ar\,act} R_{ar} + \dot{\theta}_{al\,act} R_{al}}{4} \tag{3}$$

Using the second PID controller, it is possible to calculate the control effort speed $v_{ctr\,eff}$.

$$v_{ctr\,eff} = k_p(v_{set} - v_{est}) + k_i \int (v_{set} - v_{est})\mathrm{d}t + k_d \frac{\mathrm{d}(v_{set} - v_{est})}{\mathrm{d}t} \tag{4}$$

Lastly, the torque sent to the actuators is each of the τ_{set} variables, one for each wheel.

$$\begin{aligned} \tau_{fr\,set} &= \tau_{fr\,out} + v_{ctr\,eff} \\ \tau_{fl\,set} &= \tau_{fl\,out} + v_{ctr\,eff} \\ \tau_{ar\,set} &= \tau_{ar\,out} + v_{ctr\,eff} \\ \tau_{al\,set} &= \tau_{al\,out} + v_{ctr\,eff} \end{aligned} \tag{5}$$

The derivatives and integrals are handled with the systems presented in section 2.24. Due to different separate block operations it is specially important to pay attention to the priority values in these systems to ensure everything is performed in the appropriate order.

2.17.4 Output Control 1 and 2 - COMM

After performing the controller operations, the slave controller boards send their solutions to the master in the communication tasks Output Control 1 and Output Control 2. In figure (62) it can be seen the encoded values that are transmitted.

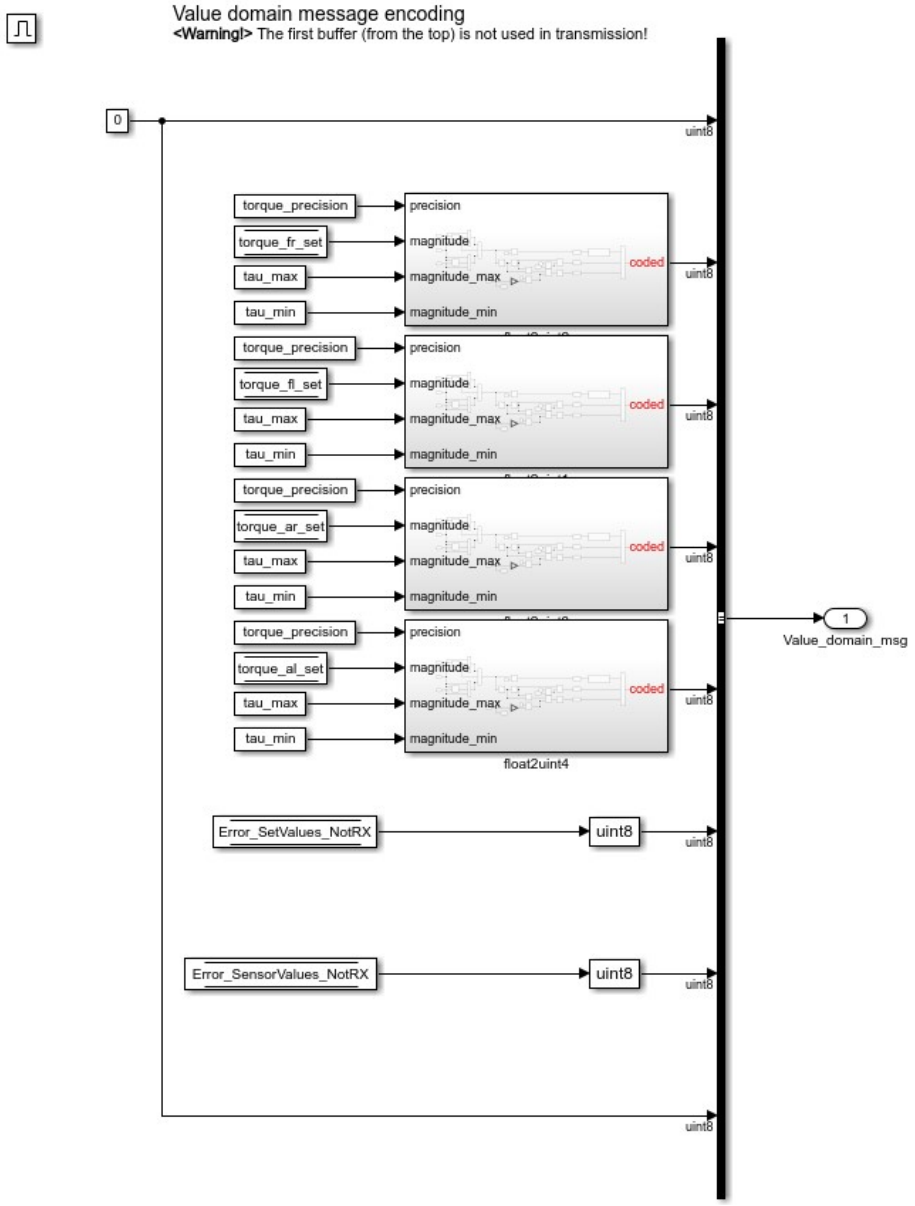


Figure 62: Message value data encoded subsystem in the output control communication tasks.

The slaves shall decide when they shall send their outputs, if during Output Control 1 or Output Control 2. The idea is that the lowest ID slave shall send its output during Output Control 1. The strategy followed is presented in figure (63), which is the transmission condition from the communication tasks. The transmission condition for Output Control 2 is analogous.

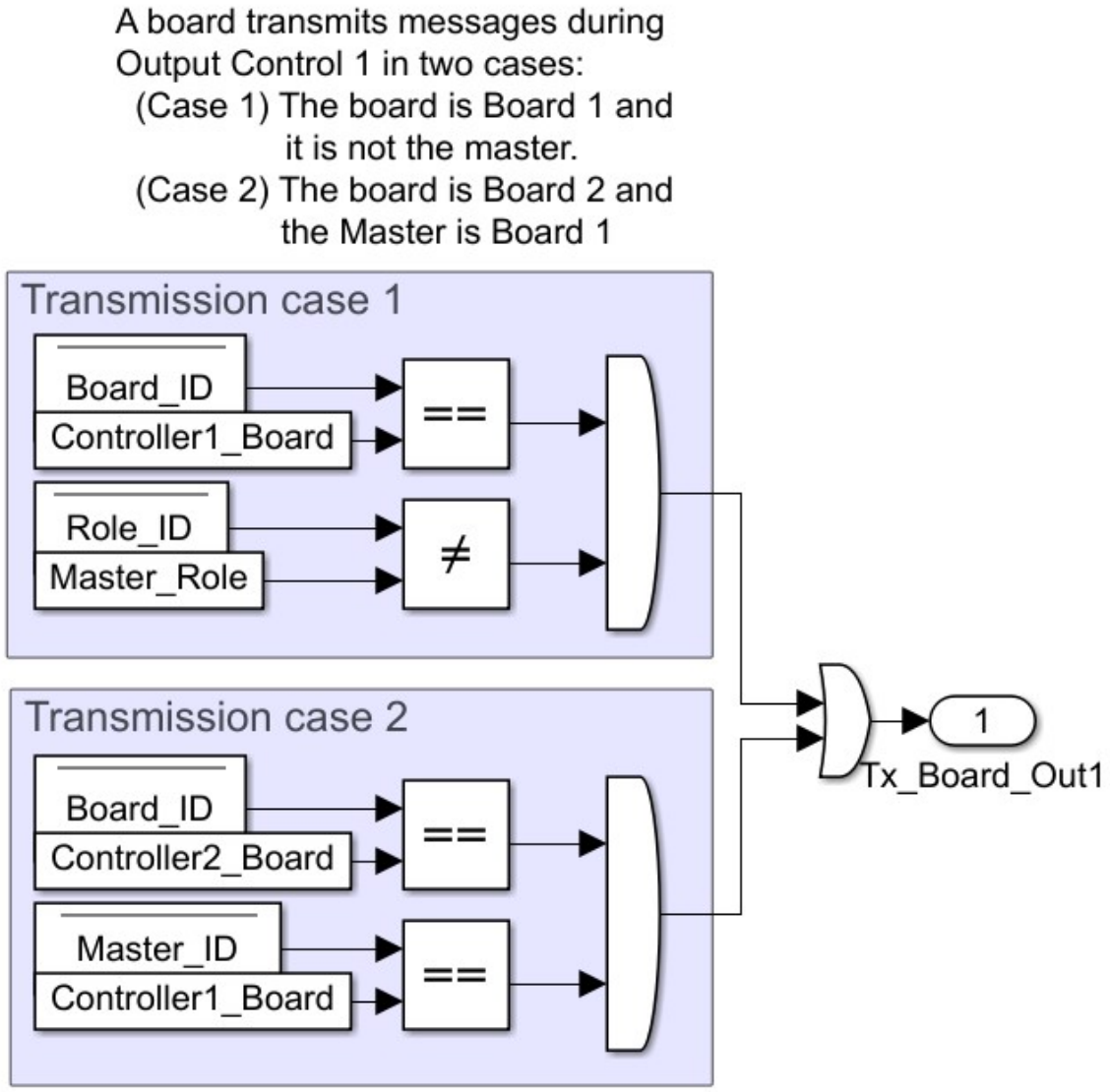


Figure 63: Output Control 1 transmission condition.

2.17.5 Triple Modular Redundancy

The triple modular redundancy (TMR) system is compounded of three subsystems. It has been observed that the encoding/decoding operation between uint8 and float types is quite computational intensive, so the loading up operation is divided in two, one per Output Control message. The main TMR system code page is presented in figure (64).

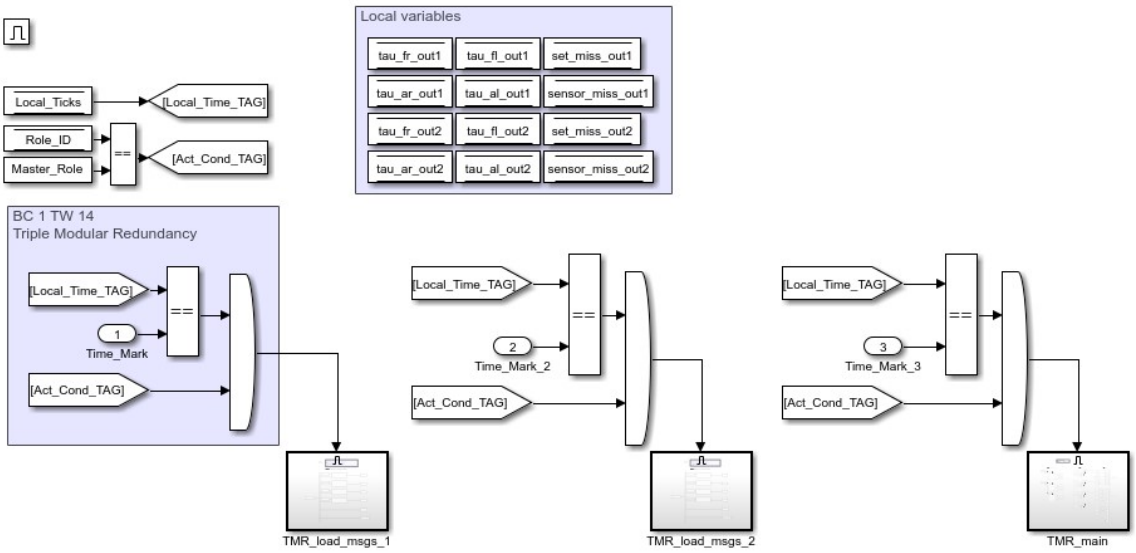


Figure 64: Triple modular redundancy system.

The subtasks activation is governed by different time marks defined in the MATLAB startup file. The different time marks must be at least one tick apart so the subtasks are not activated during the same tick. The current period set for the computational tasks is 4 ticks, and it can be as small as the maximum amount of subtasks that there are in the computational tasks of the matrix cycle.

As it can be seen in figure (65), the output control messages are stored in different variables. In case there were any communication error regarding the set or sensor values, the set_miss or sensor_miss flags would be set, informing the master that this slave did not perform the controller calculations.

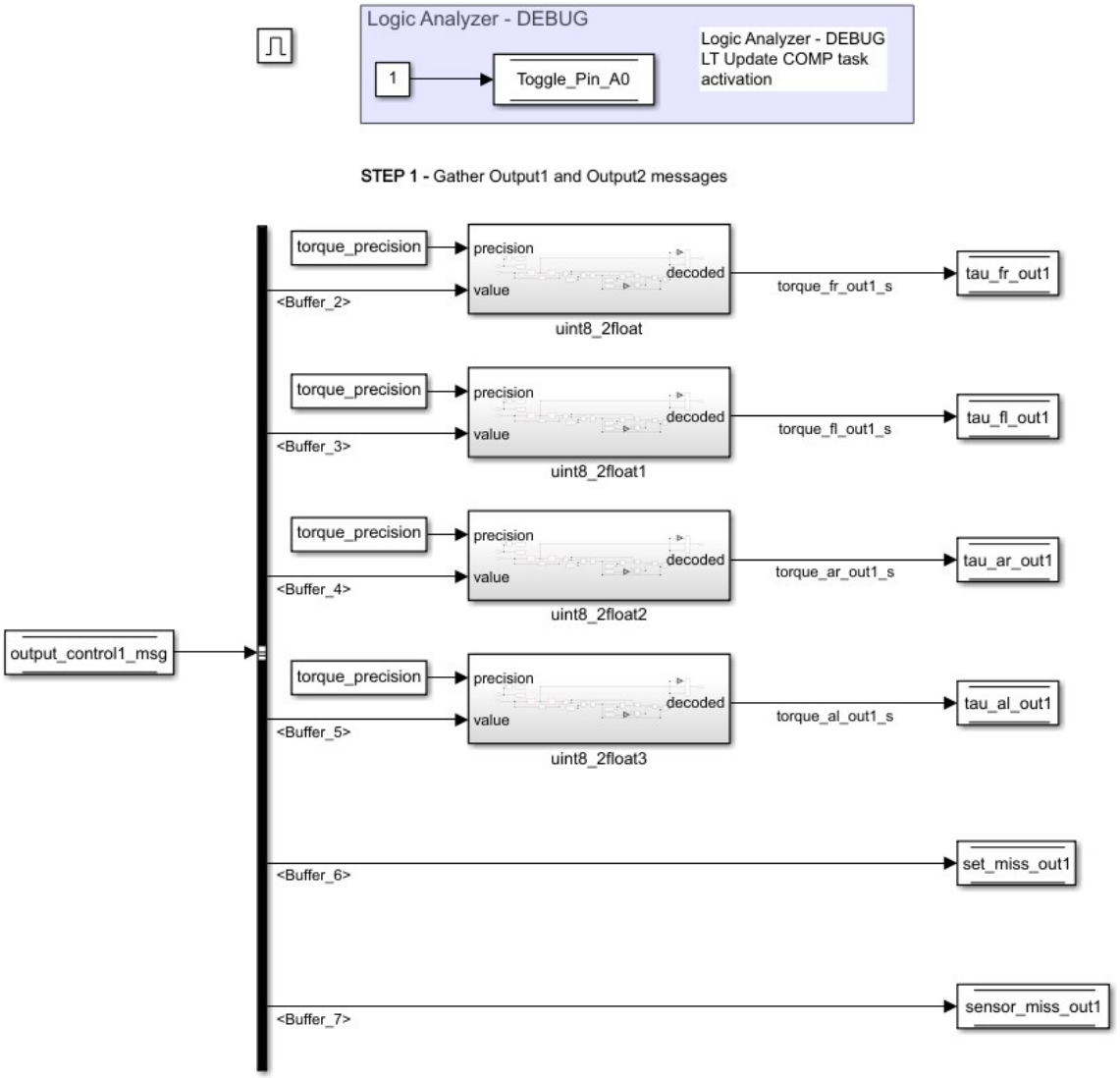


Figure 65: Output control data loaded into variables in the triple modular redundancy system.

The third subtask of the triple modular redundancy system is the one actually containing the main triple modular redundancy calculation. It is further divided in different sections, from left to right. Starting with the agreement check of the slaves' calculations with the controller and between each other, presented in figure (66). The calculations should be compatible within a certain error width defined in the MATLAB startup file with the variable e_width. The current error is set at 10% of the torque range.

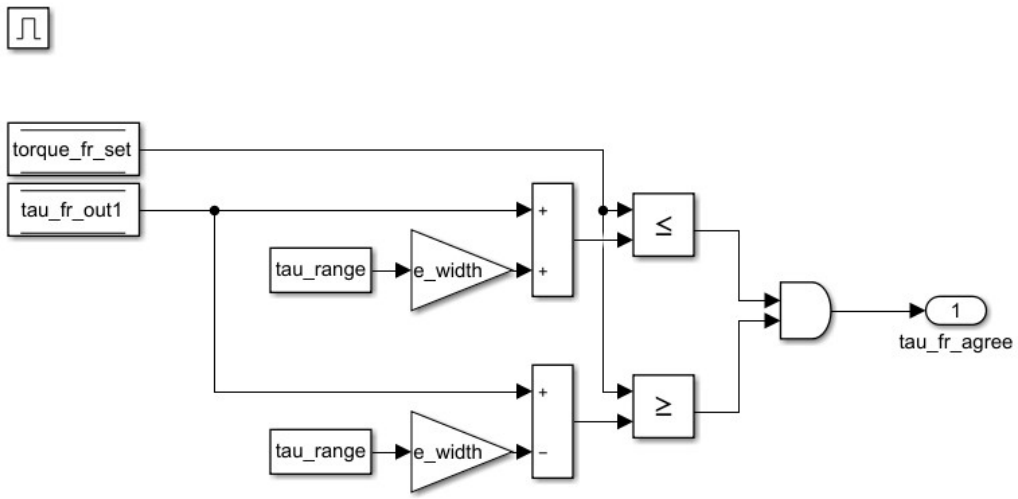


Figure 66: Agreement check between the controller and the first slave calculation of the front right wheel torque.

Once the agreements are checked it is possible to actually perform the triple modular redundancy. An example of a triple modular redundancy operation is presented in figure (67). With just one agreement (two operations being compatible within the error range) the torque value is filled with a calculation. If no agreements were found, the output is set to zero.

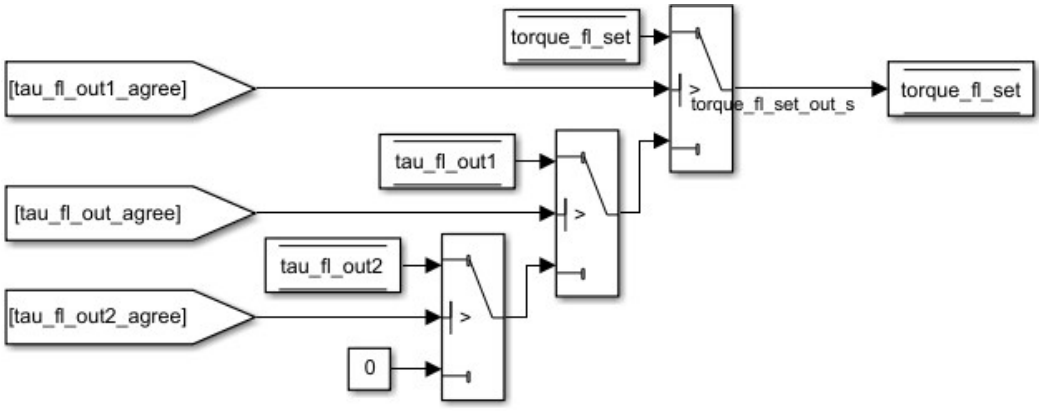


Figure 67: Example of the triple modular redundancy operation for the front left wheel torque.

The last operation performed in the main TMR subsystem is the error log constitution. There are three error log variables, compounded of several flags informing of the success in the communication during the cycle and the agreement of the calculations. Each bit of information is summed up in a different position of the data byte so the vehicle emulator can be informed of the state of the controller to some extent.

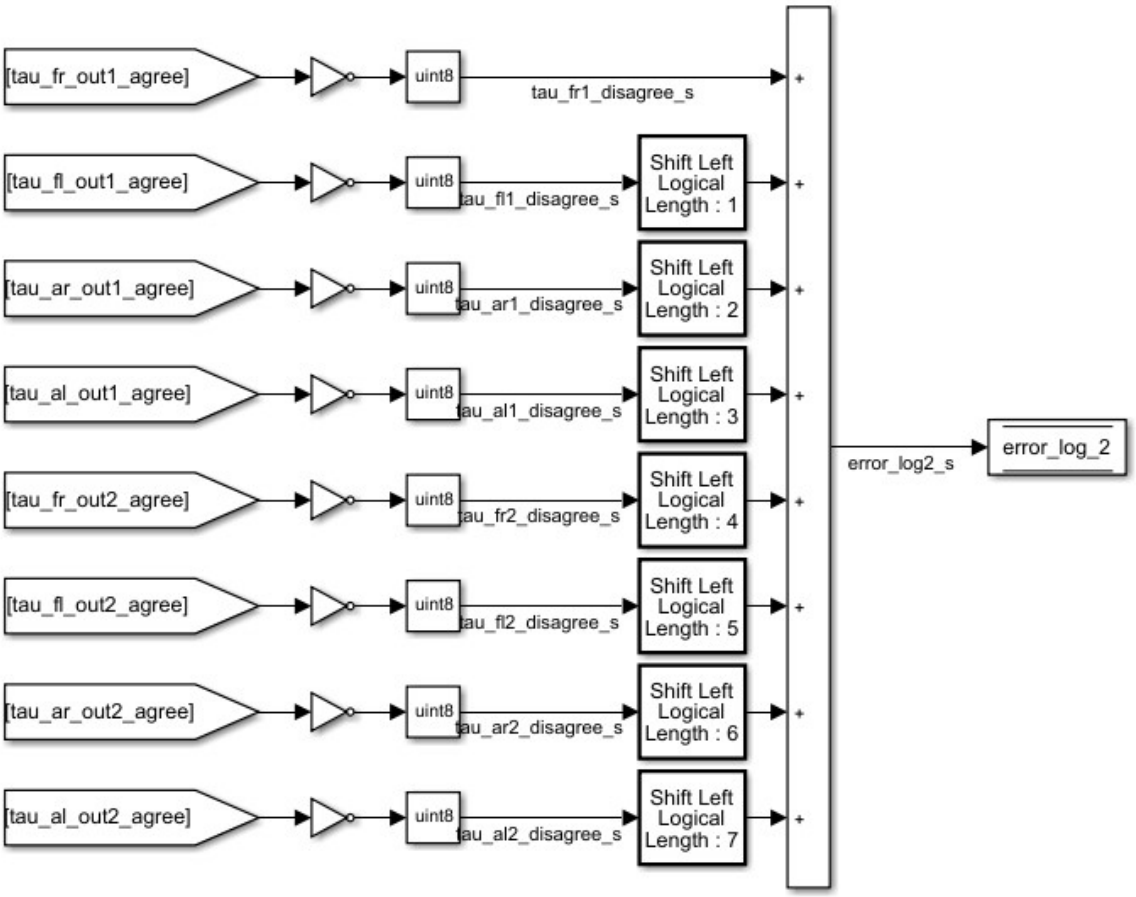


Figure 68: Example of error log compound.

2.17.6 Output emulator - COMM

In the last communication task of the controller matrix cycle, the output emulator, the outputs decided in the triple modular redundancy and the error log bytes are transmitted. The order of this data values and their coding is presented in figure (69).

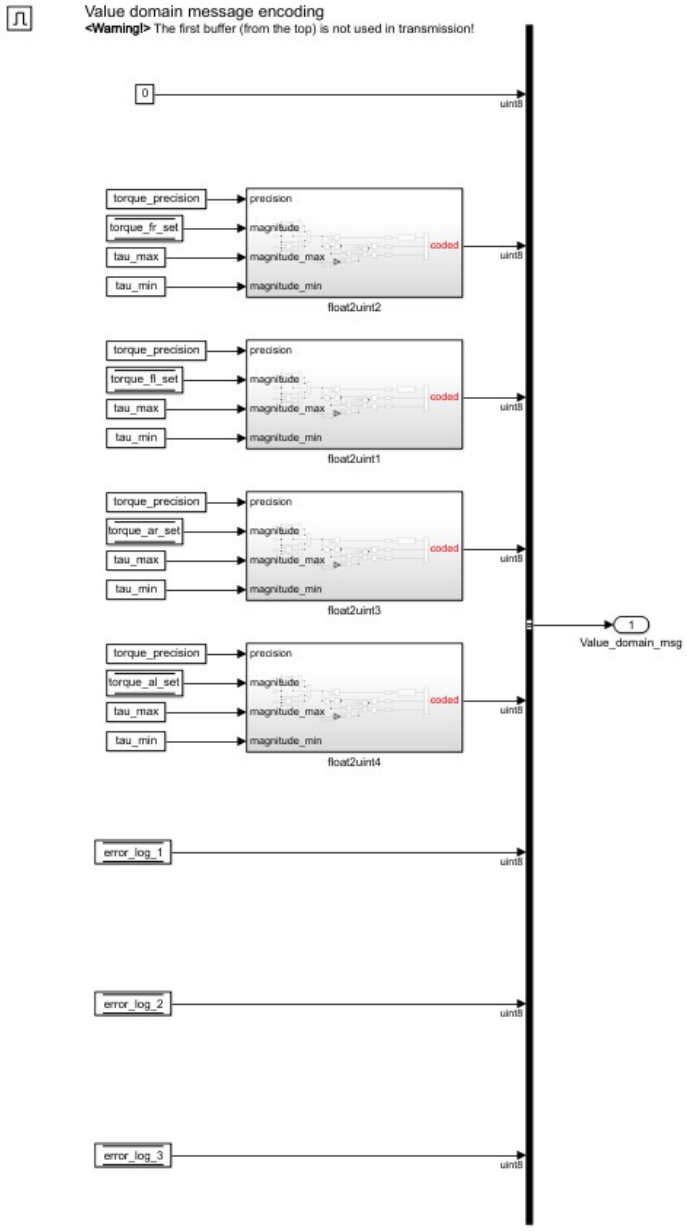


Figure 69: Message value data encoded subsystem in the output emulator communication task.

2.18 Input generator basic cycle 0

During its first basic cycle the input generator updates its local time with the controller master reference message. If no message was received it automatically resets to wait for the next reference message. If the reference message is successfully received, the cycle variables are reset by the end of the cycle. The input generator's first basic cycle is presented in figure (70).

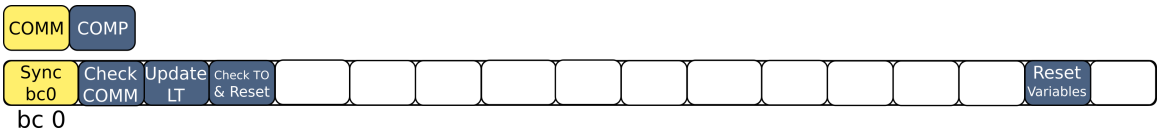


Figure 70: Input generator first basic cycle.

2.19 Input generator basic cycle 1

In its second basic cycle the input generator updates its local time again with the reference message. Then it sends the set values selected with HANTune and prepares itself for the next cycle. The second basic cycle in the input generator is presented in figure (71).

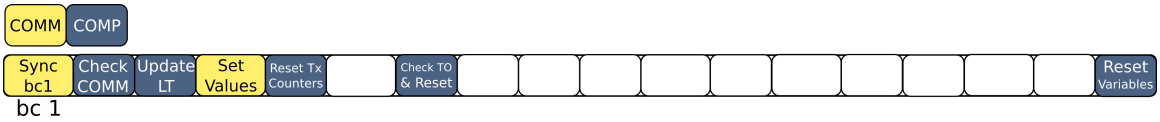


Figure 71: Input generator second basic cycle.

The input generator sends the set values to the controller at its only transmitting communication task. The set values sent are presented in figure (72). The variables delta_f_set, delta_a_set and v_set are initialized in the MATLAB startup file, and can be modified in HANTune during runtime.

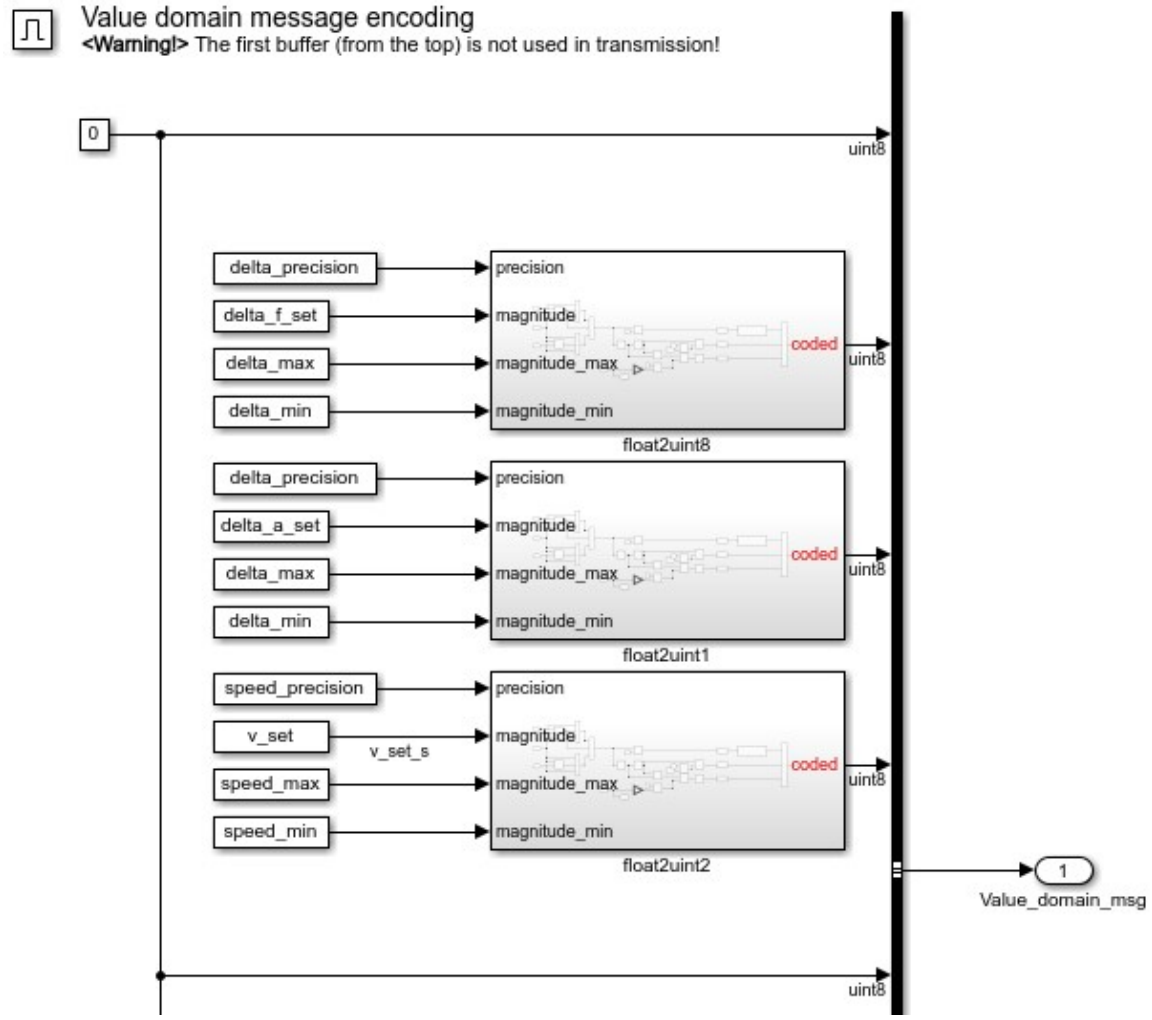


Figure 72: Message value data encoded subsystem in the set values communication task.

2.20 Vehicle emulator basic cycle 0

The first basic cycle in the vehicle emulator can be seen in figure (73). After updating its local time with the reference message from the controller master, the vehicle emulator makes the speed and steering calculations using the last received torques from the controller. If no reference message is received, the vehicle emulator reset once the calculations are finished.

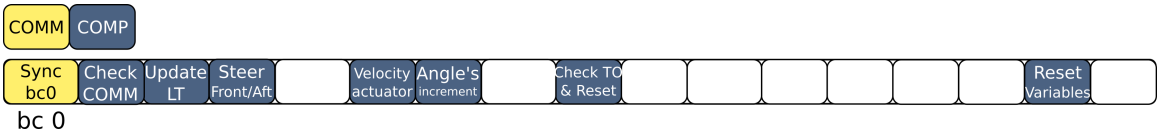


Figure 73: Vehicle emulator first basic cycle.

2.20.1 Vehicle emulator calculations

The vehicle emulator calculations are divided into three different computational tasks. These calculations include the steer angle rates for the front $\dot{\delta}_f$ and aft $\dot{\delta}_a$ parts of the vehicle and the velocity of the actuators v_{act}, which are needed for the calculations of the wheel angle displacement θ of each wheel. The steering rates, actuator velocity and wheel angle displacement are calculated using the following three equations:

$$\dot{\delta}_i = \int \left(\frac{\tau_{ir\,set}}{R_{ir}} - \frac{\tau_{il\,set}}{R_{il}} \right) \frac{T_{li}}{I_i} \mathrm{d}t \tag{6}$$

$$v_{act} = \int \left(\frac{\frac{\tau_{fr\,set}}{R_{fr}} + \frac{\tau_{fl\,set}}{R_{fl}} + \frac{\tau_{ar\,set}}{R_{ar}} + \frac{\tau_{al\,set}}{R_{al}}}{m_v} \right) \mathrm{d}t \tag{7}$$

$$\theta_{ij} = \int \left(v_{act} + \frac{\dot{\delta}_f T_{li}}{2} \right) \frac{1}{R_{ij}} \mathrm{d}t \tag{8}$$

where i refers to either front or aft and j to right or left. I is the moment of inertia at the front or aft part of the vehicle. Derivatives and integrals are handled with the systems presented in section 2.24.

2.21 Vehicle emulator basic cycle 1

The vehicle emulator second basic cycle, presented in figure (74), sends to the controller the sensor values previously calculated during the first basic cycle. At the end of the second basic cycle it receives the torque outputs from the controller to make the speed and torque calculations again during the next cycle.

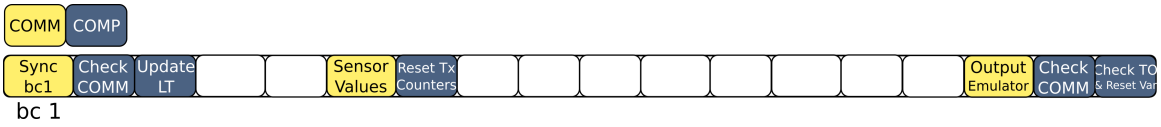


Figure 74: Vehicle emulator second basic cycle.

The vehicle emulator send its sensor values to the controller during the sensor values communication task. This values were previously obtained during the vehicle emulator calculations. They can be seen in figure (75).

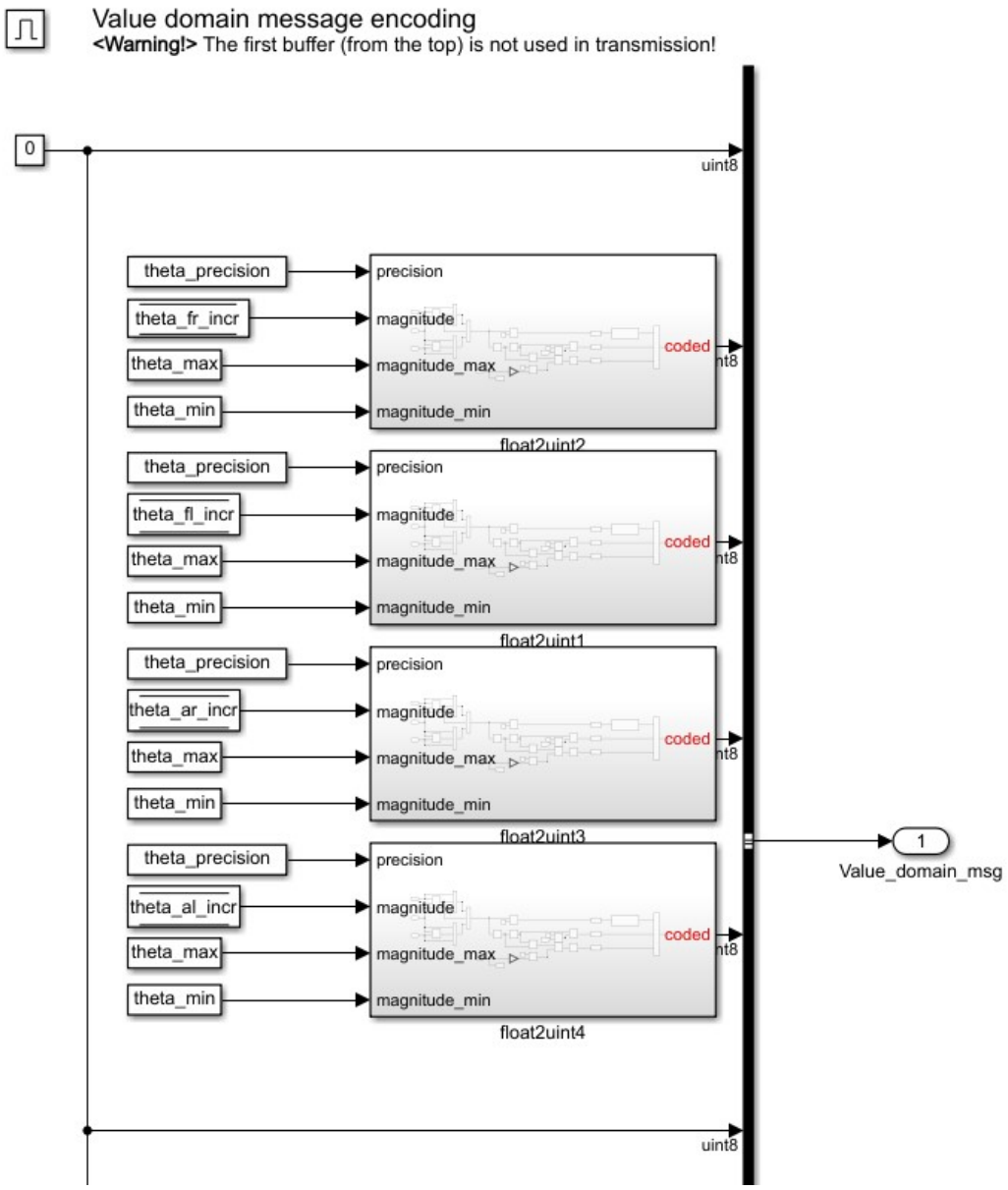


Figure 75: Message value data encoded subsystem in the sensor values communication task.

2.22 CAN Rx

The CAN reception (Rx) system is in the TTA CAN system level and is activated whenever the matrix cycle from the TTA system requires to listen to a message in a communication task. The Rx system is compounded of the RxID CAN, that contains the HANcoder CAN blocks, and the receiving state machine with three subsystems, one per state.

2.22.1 RxID CAN

The current version of the CAN HANcoder blocks cannot receive a Rx_ID input with a variable register. The Rx_ID input has to be a constant. Using a variable ID input makes the Simulink compilation crash or makes the CAN HANcoder block CAN_new output be constantly -1. This is why a selection system has been developed so different messages can be listened to while changing a single variable in the system, the Rx_ID. In figure (76) it can be seen how depending on the chosen Rx_ID a specific subsystem with the HANcoder CAN block is activated.

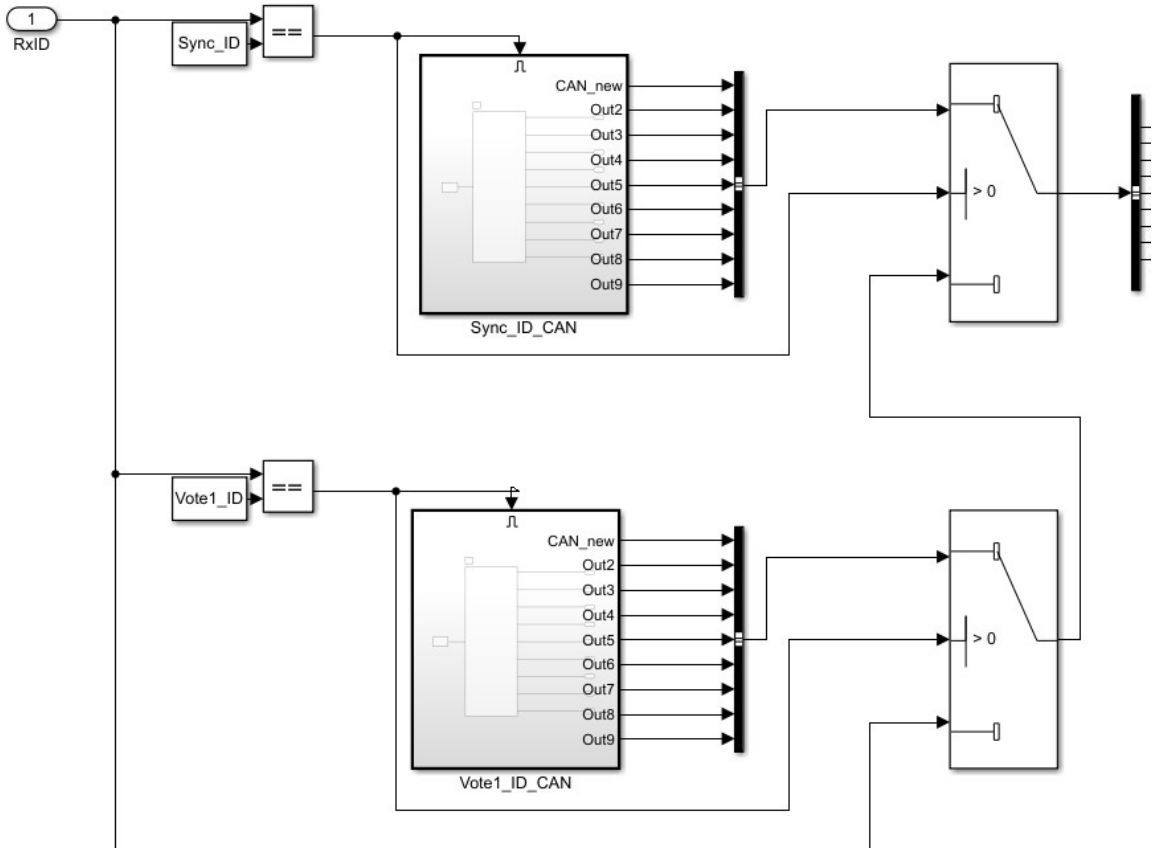


Figure 76: RxID CAN subsystem with two examples of CAN receive calling.

Every specific message Rx subsystem outputs the HANcoder CAN block information, which is also exported out to the main CAN Rx system. An example of how the HANcoder CAN block is handled with a constant ID input is presented in figure (77).

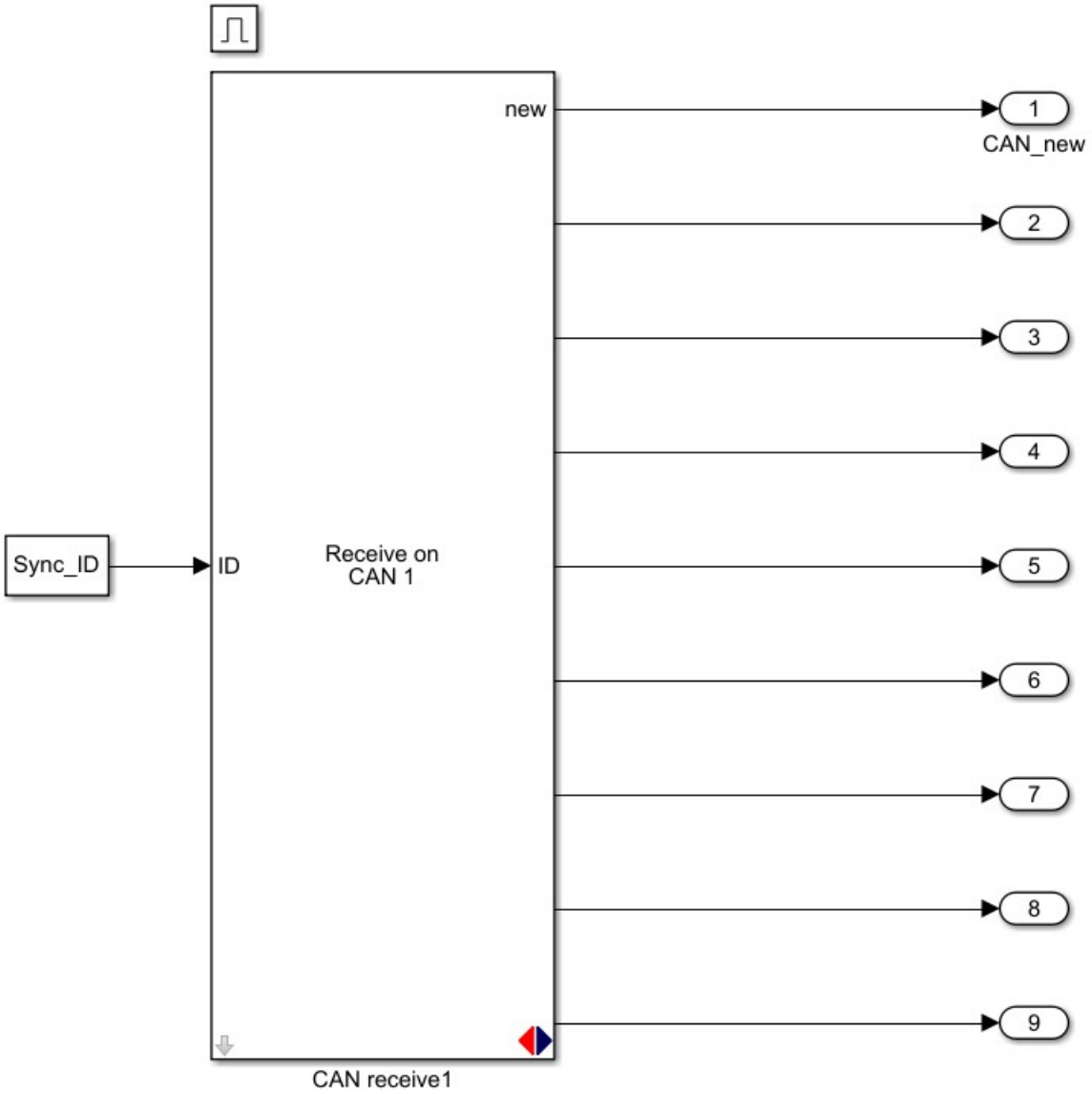


Figure 77: Reference message CAN receive block as an example of how the CAN HANcoder blocks are handled in the software.

2.22.2 Rx state machine

The reception of a message is divided into three phases coordinated by a state machine. This is presented in figure (78).

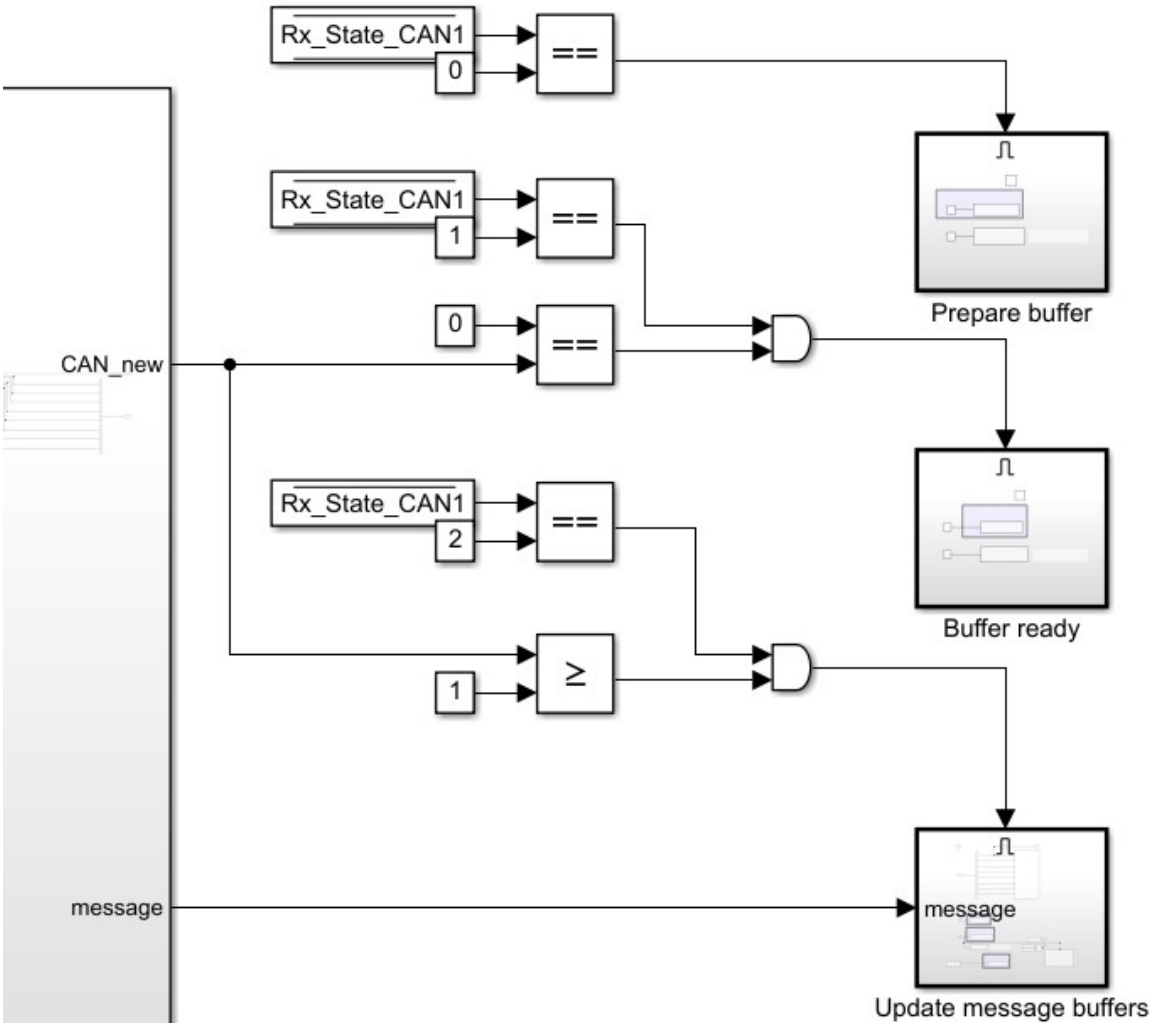


Figure 78: State machine in the CAN receiving system. Firstly, the CAN buffer is reset. Then, the system is called repeatedly until a message arrives. When a message arrives the message buffer is updated and a flag indicating a new message has arrived is set.

The Rx state is reset at the matrix cycle manager, at the beginning of the communication task. When this happens the receiving protocol starts by calling the appropriate HANcoder CAN block so the CAN buffer is reset. The state machine just changes to the next state: preparing the buffer.

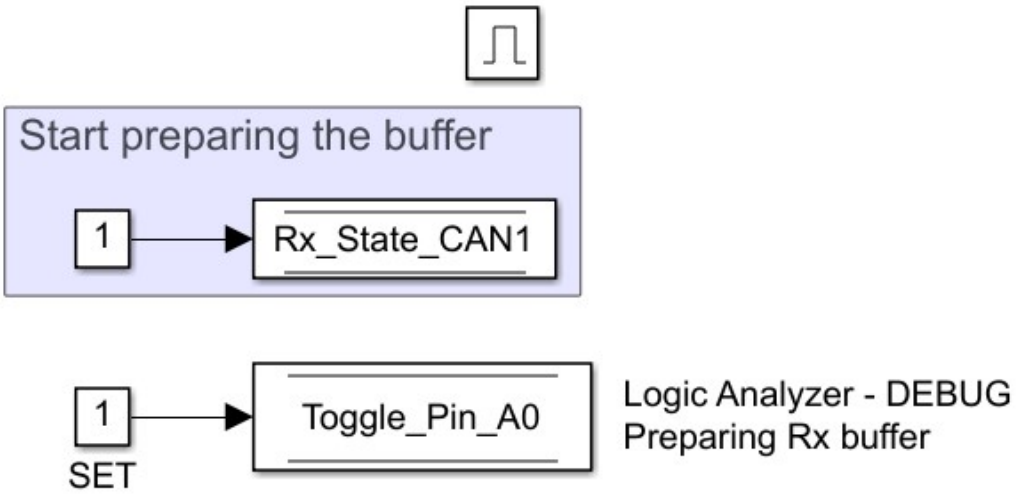


Figure 79: Preparing the buffer state in the CAN Rx system.

When the CAN buffer represented by the CAN_new output is zero the system transitions to the last part of the protocol: CAN buffer ready. This can be seen in figure (80).

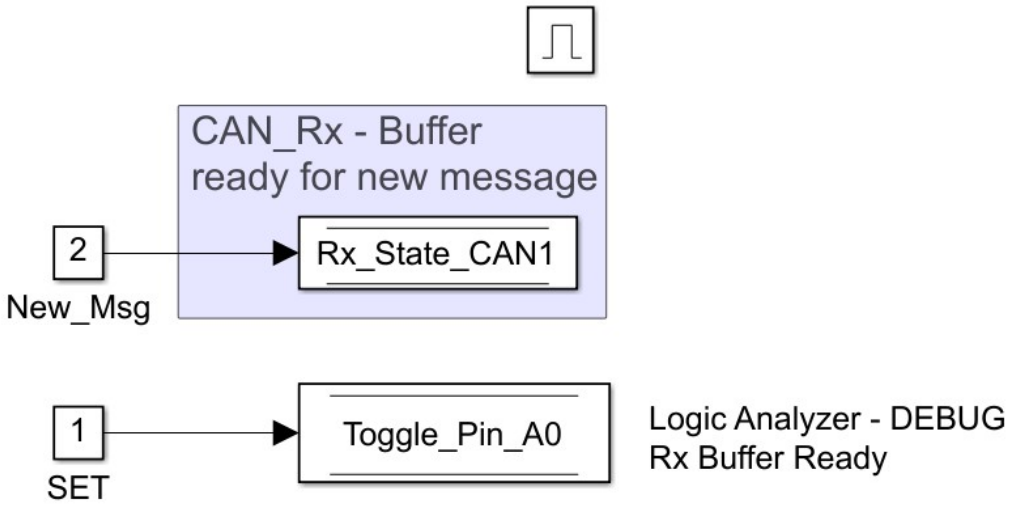


Figure 80: Buffer ready subsystem in CAN Rx system.

Lastly, when a message arrives while the reception system is ready, it is stored in the corresponding channel message variable Msg_Rx_CAN. Also, the message ready flag is set, a logic analyzer measurement system for the communication delay is activated and the current local time is stored so the desync can be calculated later. This is presented in figure (81).

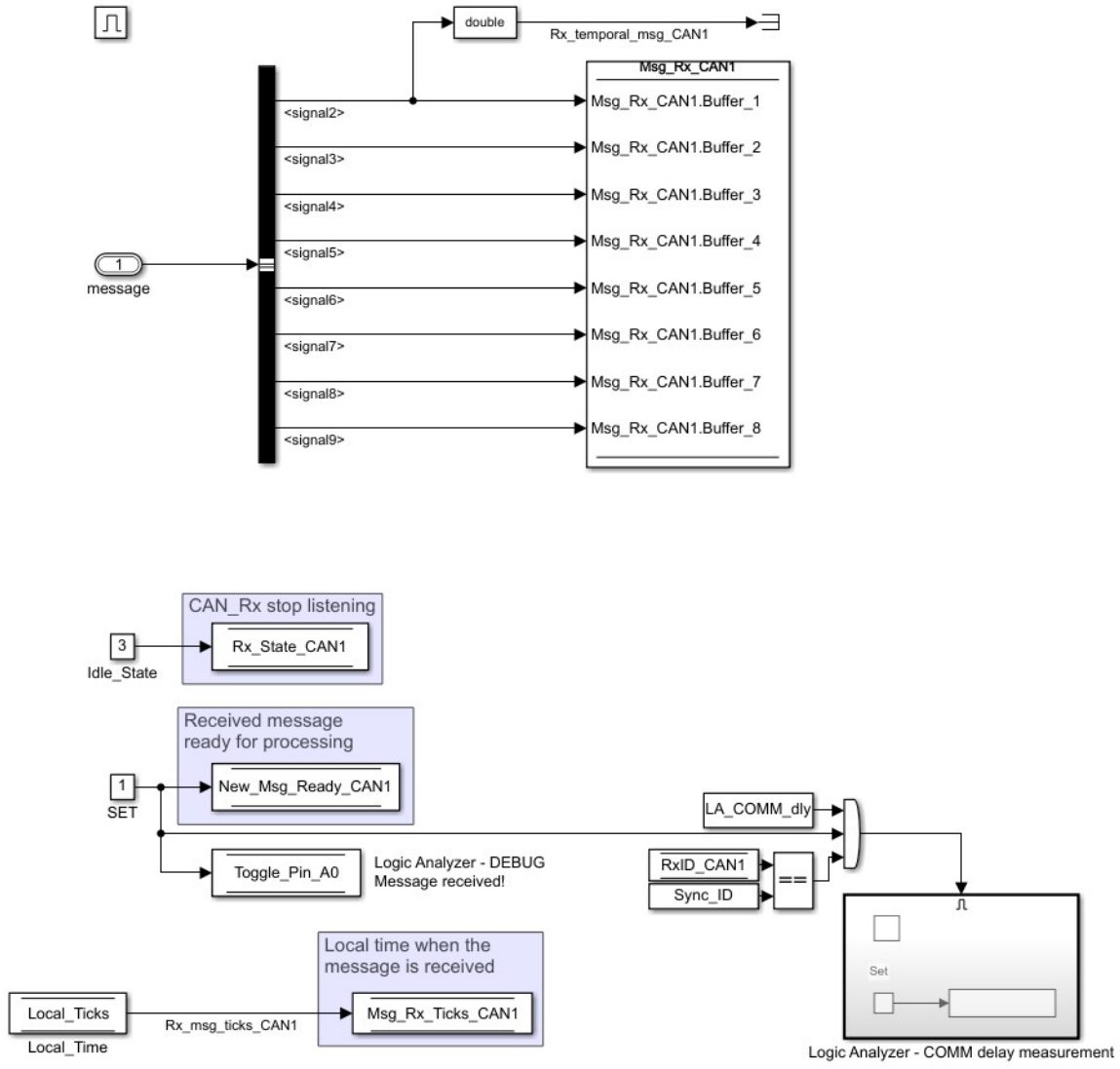


Figure 81: Update message buffers subsystem in CAN Rx system.

2.23 CAN Tx

The transmission CAN system is not as complex as the reception system presented in section 2.22. The system is presented in figure (82), and it is mainly constituted by the message being input to the HANcoder CAN transmitting block, the message counter update and a logic analyzer measurement system. The transmission CAN block does not require extra subsystems, as it is possible to give it a variable ID as an input.

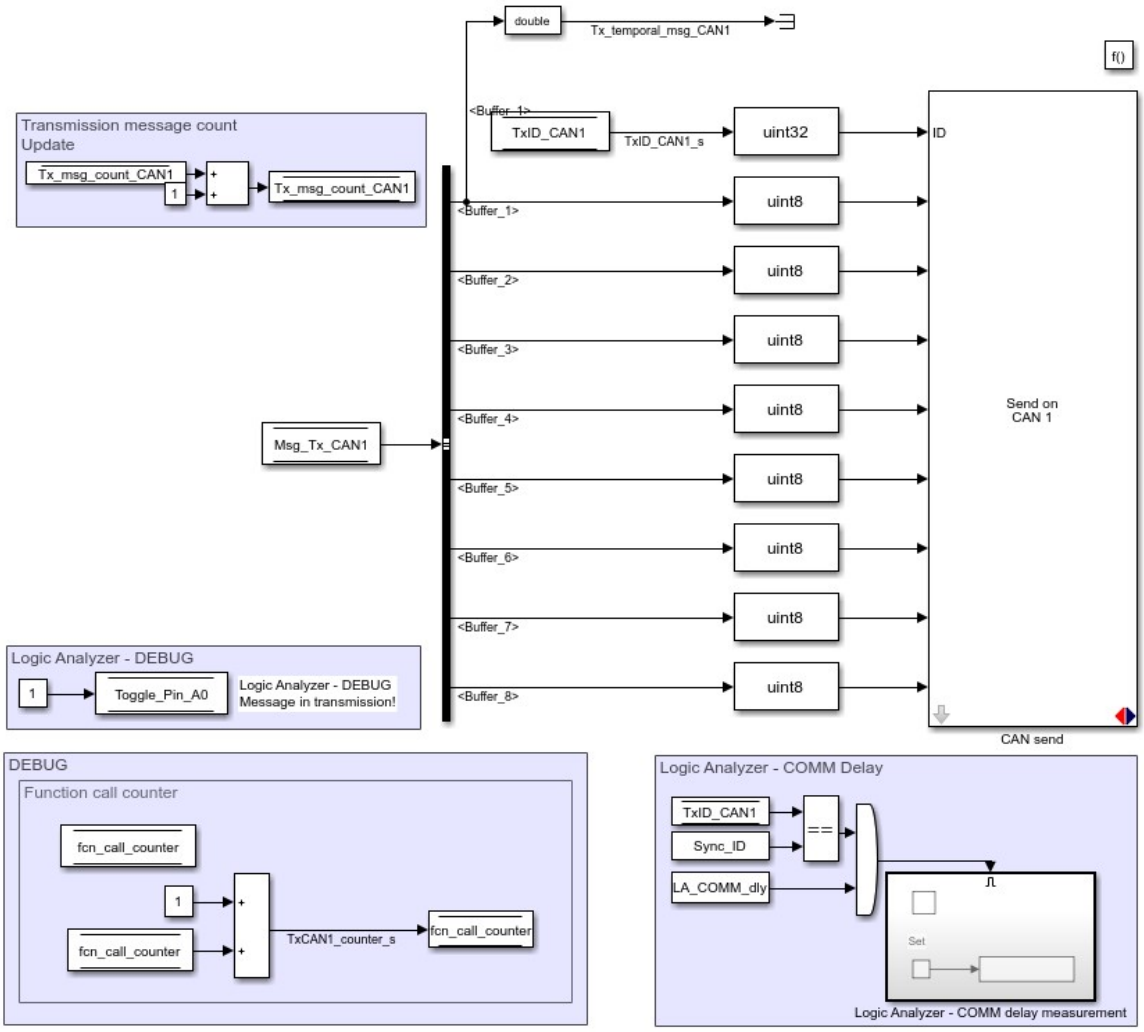


Figure 82: Transmission CAN system.

2.24 Recurrent subsystems

There are several block structures that are used several times throughout the software. These subsystems or block groups have a very specific purpose. This section makes a compilation with brief explanations for each recurrent subsystem.

2.24.1 Counter

A counter variable keeps track of a value that is progressively increased. The most important counter variable in the software is the local time stored in the Local_Ticks, and it is increased every time there is an interrupt request from the hardware clock as long as the Clock_State, governed by the wake up button, is true. The counter increase is based upon the switch block. If the condition (middle input) of the switch is true, the switch outputs the top input. Otherwise, the bottom input is output. The Local_Ticks counter update is presented in figure (83) as an example. Other places where counters are found in the software are the message transmission counter, for every message being transmitted during the same communication task, or the function activation counter, in the CAN systems, used to keep track of the number of times these were called by the matrix cycle.

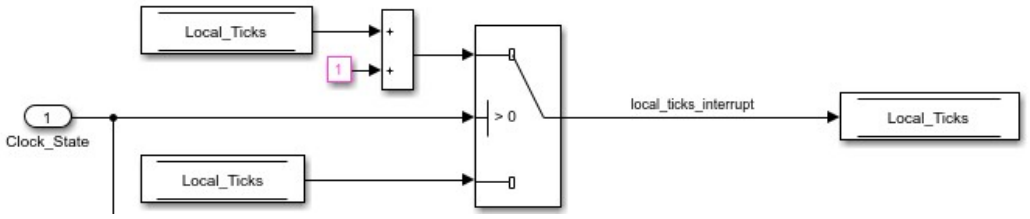


Figure 83: Counter group of blocks increasing the local time. This can be found in the Update Local Time subsystem, at the TTA CAN system.

2.24.2 Message coding and decoding

The temporal information of the messages is a byte compounded of the basic cycle (one bit), the message transmission counter (three bits) and the board number (four bits). It is required to merge the three parts together to code the message, as presented in figure (84), and decode them back to their original values at the receiver end, as presented in figure (85).

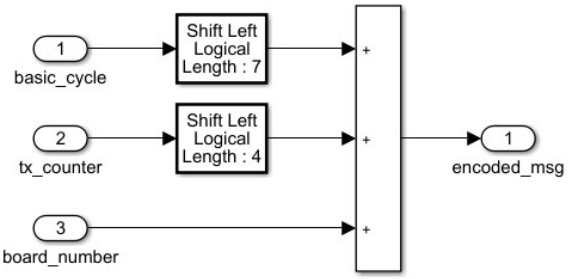


Figure 84: Coding example of the temporal information of a message.

The coding and decoding subsystems base their functionality in the shift right and left logical blocks. Coding requires shifting left the different bits so they occupy the appropriate space in the byte and sum everything together. Decoding is a little bit more complex, as recovering the original information requires two shifts. For example, the board ID requires shifting left first to remove the information from the basic cycle and the message count and shift right again so the result contains just what we were searching for. Coding the temporal information happens in the update of the transmission buffers, right before a message is transmitted. The decoding is done in the check communication systems, when looking for the message coherence.

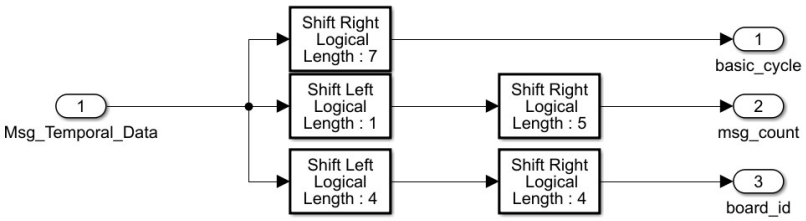


Figure 85: Decoding example of the temporal information of a message.

2.24.3 Float data encryption

CAN communication allows for the transmission of bytes of unsigned integer data. To transmit float numbers it is necessary to code the information before the transmission and decode it at the receiving end. An example of this encoding is presented in figure (86). The float encryption happens in the message value data encoded systems of the transmission part of a communication task. The decoding operation from uint8 to float to recover the information in the receiving end happens in different parts of the software, such as the slave initialization, some check communication tasks or at the beginning of the triple modular redundancy.

Figure 86: Eight-bit data binary encoding example. The most significant bit (red) corresponds to the sign of the number (0 is positive). The grey bits correspond to the integer part of the number and the blue bits to the decimal part. This number, 67 in binary, originally represents +4.1875.

To understand how the coding works it is possible to see the example from figure (86). This could be a data value from a torque variable. Torque variables in the controller have a range of [-5, 5]. The absolute value of the integer part of the float number requires a maximum of three bits to be represented in binary ($5_{10} = 101_2$). This means that using a bit for the sign and three bits to represent the integer part, the remaining four bits can be used to represent the decimal part of the number.

Using +4.2315 as an example, the sign (+) is considered as 0 and the integer part is 4. The remaining question is how to represent the decimal part 0.2315 with a precision of four bits. The resolution of the number with four bits decimal point is $2^{-4} = 0.0625$, so dividing 0.2315 over the resolution 0.0625, the fixed result is 3 (11 in binary base). Lastly, the three numbers are summed up together taken into account their weight in the byte value: $0 \times 2^7 + 4 \times 2^4 + 3 = 67$. Knowing the precision (bits for the decimal part) used to code the number it is easy to decode it back to its original value in the receiving end.

2.24.4 Function call generator

There are different kind of systems in the software: *function-call*, *if* and *enabled systems*. The if and enable conditions are processed within the same hierarchy level as the system they are calling. This is why we cannot use these to call a system with the hardware clock interruption request. The function-call execution order is always strictly afterwards the function call signal is generated, regardless of the hierarchy level. This means that if inside the matrix cycle we generate a function-call signal to call a CAN system, this CAN system is the next activation in the execution order of the software. This is how we are able to make our own granularity disregarding Simulink activation times, generating function call signals within the local time system activated with the hardware clock interrupt requests. The Stateflow charts do not have a priority option for execution order, their activation happens right after the activation of their input. More information about function-call subsystems can be found in the MATLAB documentation. Figure (87) shows the main Stateflow diagram window acting as a function-call generator. Every time this system is called and the input condition is true, it is going to generate a function-call signal that automatically activate the block it is connected to.

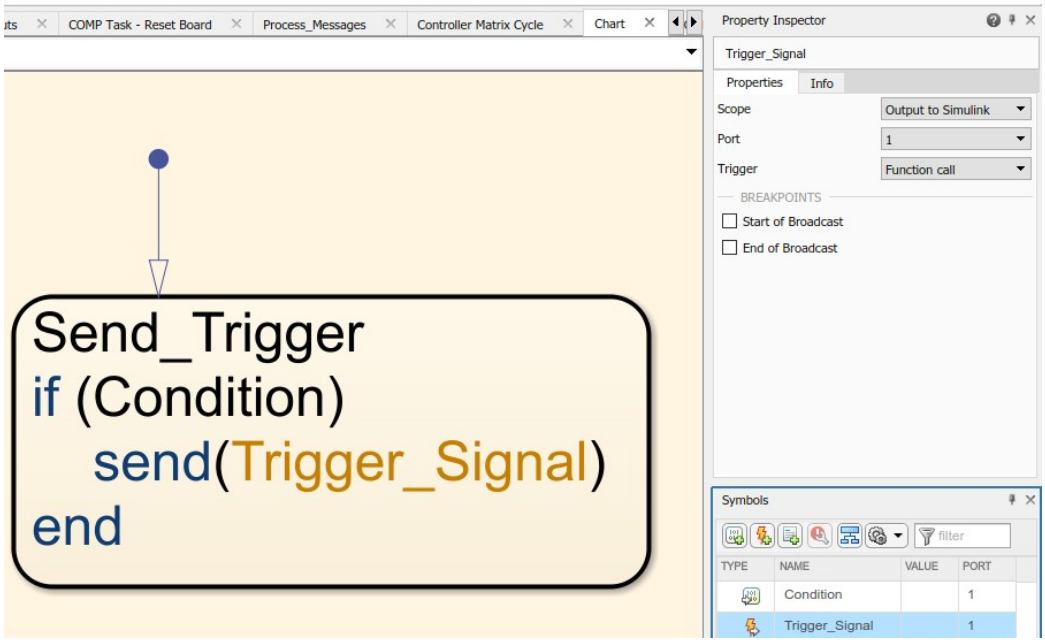


Figure 87: Stateflow system with a function-call generator.

There are two important panels with which to control the properties of the Stateflow signals: the symbols and the property inspector, shown in the right panel of figure (87). While inside the Stateflow system, click View and then you will see both options, symbols and property inspector as possible options. The symbols panel allows for creating new variables with different classes, such as data, messages or triggered signals. These can be chosen to be inputs, local data or outputs, among others. When selecting a triggered signal variable, the property inspector shows if the Trigger is function call or either edge. In order to create a function-call generator, it is important that this option is set at Function call. More information about Stateflow charts can be found in the MATLAB documentation.

2.24.5 Toggle value

There are some boolean registers in the software that require their values to be toggled from false to true and vice versa. The most common occurrence of this is seen in the logic analyzer systems, where a digital pin state is toggled to register when some kind of event happened. An example is shown in figure (88). The functionality of the toggle is based upon the switch block. If the State of the digital pin was true, the top input of the switch is output and the state changes its value to false. If the state was false, the bottom input of the switch is output and the state changes its value to true.

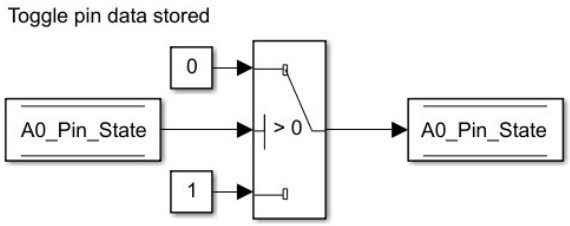


Figure 88: Toggle digital pin state example.

2.24.6 Integral

The integral subsystem uses the trapezoidal rule approximation, as presented in figure (89). It requires four different inputs, the current integral value (or previous value at the time it is being calculated, $integral_{i-1}$), the current value of the variable that is being integrated y_2, the previous value of the variable being evaluated y_1 and the increment in time since the last evaluation $t_2 - t_1$. The integral subsystems can be found in the controller calculations, in the second basic cycle of the controller matrix cycle, and in the vehicle emulator calculations, in the first basic cycle of the vehicle emulator matrix cycle.

Integral approximation:

$$integral_i = integral_{i-1} + \frac{y_2 + y_1}{2}(t_2 - t_1)$$

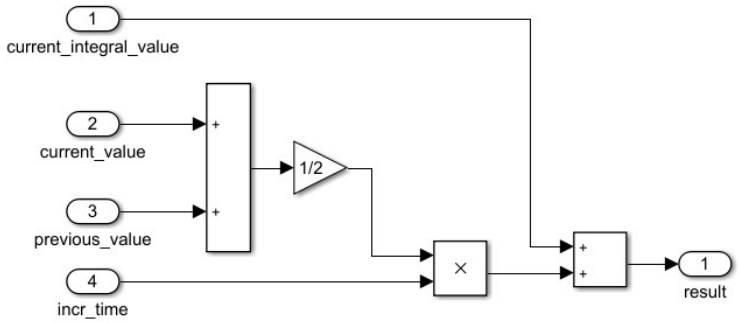


Figure 89: Integral subsystem.

2.24.7 Derivative

The derivative subsystem uses the linear approximation shown in figure (90). It requires as inputs the current and previous values of the variable being evaluated and the time since the last activation. It can be found in the same systems as the integral subsystem, within the controller and vehicle emulator calculations.

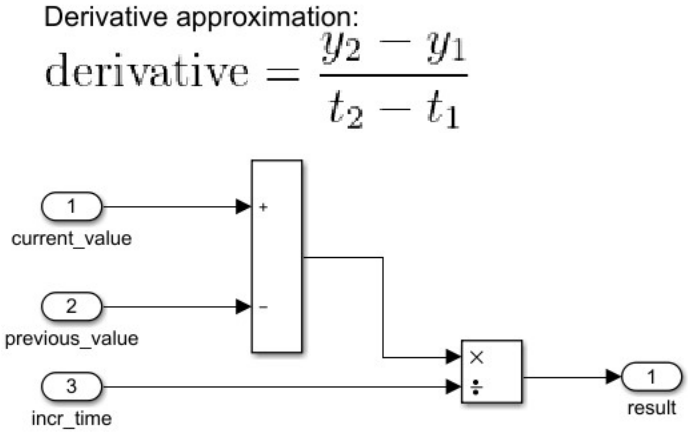


Figure 90: Derivative subsystem

2.25 List of variables

The following list of variables shows a brief description of the different data store memory blocks declared in the code. They are ordered from the top hierarchies to the deepest sections of the code. It is possible to see where each variable is used in the code by double clicking a data store memory block and reviewing its list of corresponding data store read/write blocks.

TTA CAN System

init_clock: flag to control Initialize Clock Schedule. True means the clock has been initialized.

Local_Ticks: local time counter.

Board_ID: unique board identification.

Master_ID: board ID of the controller board with the master role.

RxID_CAN1: receiving ID of the message currently being listened to at CAN channel 1.

Msg_Rx_CAN1: last message received at CAN channel 1.

Rx_State_CAN1: state machine state at the CAN1 Receive system.

New_Msg_Ready_CAN1: flag indicating if a new message has been received at CAN channel 1. True means that a message has been received

Msg_Rx_Ticks_CAN1: local time at which the last message was received at CAN channel 1.

RxID_CAN2: receiving ID of the message currently being listened to at CAN channel 2.

Msg_Rx_CAN2: last message received at CAN channel 2.

Rx_State_CAN2: state machine state at the CAN1 Receive system.

New_Msg_Ready_CAN2: flag indicating if a new message has been received at CAN channel 2. True means that a message has been received.

Msg_Rx_Ticks_CAN2: local time at which the last message was received at CAN channel 2.

TxID_CAN1: transmitting ID of the message currently being transmitted at CAN channel 1.

Msg_Tx_CAN1: last message transmitted at CAN1 Send.

Tx_msg_count_CAN1: counter with the message count in the communication task for CAN channel 1. It is currently limited to seven, as the temporal message information for message counter is only three bits.

TxID_CAN2: transmitting ID of the message currently being transmitted at CAN channel 2.

Msg_Tx_CAN2: last message transmitted at CAN2 Send.

Tx_msg_count_CAN2: counter with the message count in the communication task for CAN channel 2. It is currently limited to seven, as the temporal message information for message counter is only three bits.

Toggle_Pin_A0: flag for the logic analyzer measurement of task activation. True means digital pin A0 must be toggled.

Toggle_Pin_A8: flag for the logic analyzer measurement of communication delay, at the reception of CAN channel 2. True means digital pin A8 must be toggled.

Toggle_Pin_A9: flag for the logic analyzer measurement of communication delay, at the transmission of CAN channel 2. True means digital pin A9 must be toggled.

Toggle_Pin_A12: flag for the logic analyzer measurement of communication delay, at the reception of CAN channel 1. True means digital pin A12 must be toggled.

Toggle_Pin_A13: flag for the logic analyzer measurement of communication delay, at the transmission of CAN channel 1. True means digital pin A13 must be toggled.

TTA CAN System/ Measure EXecution time

A3_Pin_State: state of digital pin A3. One means it is high and zero means it is low.

TTA CAN System/ CAN1 and CAN2 Recieve and Send

fcn_call_counter: counter with the total number of times the system was activated.

TTA CAN System/ TTA System

Role_ID: controller role. It can be either 1 for master or 2 for slave.

Initialization_flag: flag controlling the activation of either the initialization or the matrix cycle manager systems. When true only the initialization system is activated.

basic_cycle_count: basic cycle counter. It can be either 0 or 1. Depending on its value the first or the second basic cycle of the matrix cycle is activated.

Desync_Positive: flag controlling the activation of the LT_Update Desync_Positive system. It is true when the Desync_Ticks value is positive.

Desync_Ticks: counter with the desynchronization value.

A0_Pin_State: state of digital pin A0. One means it is high and zero means it is low.

Toggle_Pin_A1: flag for the logic analyzer measurement of the communication failures. True means digital pin A1 must be toggled.

A1_Pin_State: state of digital pin A1. One means it is high and zero means it is low.

A2_Pin_State: state of digital pin A2. One means it is high and zero means it is low.

A4_Pin_State: state of digital pin A4. One means it is high and zero means it is low.

Toggle_Pin_D10: flag for the logic analyzer measurement of the ensemble desynchronization. True means digital pin D10 must be toggled.

D10_Pin_State: state of digital pin D10. One means it is high and zero means it is low.

D13_Pin_State: state of digital pin D13. One means it is high and zero means it is low.

D12_Pin_State: state of digital pin D12. One means it is high and zero means it is low.

D9_Pin_State: state of digital pin D9. One means it is high and zero means it is low.

D8_Pin_State: state of digital pin D8. One means it is high and zero means it is low.

BC0_Sync_processed: flag to ensure no task before local time update is repeated during basic cycle 0.

BC1_Sync_processed: flag to ensure no task before local time update is repeated during basic cycle 1.

delta_f_est: integral value with the estimation of the front axle steering value.

delta_a_est: integral value with the estimation of the aft axle steering value.

speed_integral: integral value with the speed of the vehicle.

torque_fr_integral: integral value with the front right wheel torque value.

torque_ar_integral: integral value with the aft right wheel torque value.

TTA CAN System/ TTA System/ Initialization

Initialization_Timeout: flag controlling if the delay counter has reached its limit. When true the RoleID and Global Time init system is activated.

Delay_Counter: counter keeping track of the number of activations of the Delay system.

TTA CAN System/ TTA System/ Matrix Cycle Manager/ Controller Matrix Cycle

msg_count_DEBUG: counter monitoring the message counter arriving at the receiving end.

Own_Vote: board ID representing the board which should be the next master.

Votes_count: dictionary data type with the separate count of every vote for each board. The type is defined in the MATLAB startup file.

Sync_bc0_missed_counter: counter of all reference messages missed in basic cycle 0.

Board1_error_counter: counter of all messages missed sent by the board with Board_ID = 1.

Board2_error_counter: counter of all messages missed sent by the board with Board_ID = 2.

Board3_error_counter: counter of all messages missed sent by the board with Board_ID = 3.

Sync_bc1_missed_counter: counter of all reference messages missed in basic cycle 1.

Set_missed_counter: counter of all set values messages missed.

Sensor_missed_counter: counter of all sensor values messages missed.

Out1_missed_counter: counter of all output controller 1 messages missed.

Out2_missed_counter: counter of all output controller 2 messages missed.

BC0_Vote1_processed: flag indicating if Vote1 message was received. True means the message was received.

BC0_Vote2_processed: flag indicating if Vote2 message was received. True means the message was received.

BC0_Vote3_processed: flag indicating if Vote3 message was received. True means the message was received.

TTA CAN System/ TTA System/ Matrix Cycle Manager/ Controller Matrix Cycle and Input Generator Matrix Cycle/ controller basic cycle 0 and Input Generator Matrix Cycle/ controller basic cycle 1

new_msg_Rx: flag indicating is a new message was received during the last communication task. True means the message was received.

Msg_Rx: buffer with the message received during the last communication task.

Reset_Board: flag controlling if a board shall be reset or not. True means it shall be reset.

TTA CAN System/ TTA System/ Matrix Cycle Manager/ Controller Matrix Cycle/ controller basic cycle 1

new_msg_Rx: flag indicating is a new message was received during the last communication task. True means the message was received.

Msg_Rx: buffer with the message received during the last communication task.

delta_f_set_stored: value received from the input generator for the steering of the front axle.

delta_a_set_stored: value received from the input generator for the steering of the aft axle.

v_set_stored: value received from the input generator for the vehicle speed.

Error_SetValues_NotRX: flag indicating if the set values message was not received. True means the message was not received.

theta_fl_stored: value received from the vehicle emulator for the angle increment in the front left wheel.

theta_fr_stored: value received from the vehicle emulator for the angle increment in the front right wheel.

theta_al_stored: value received from the vehicle emulator for the angle increment in the aft left wheel.

theta_ar_stored: value received from the vehicle emulator for the angle increment in the aft right wheel.

rx_data_incr_time: time in ticks since the last controller cycle was performed.

Error_SensorValues_NotRX: flag indicating if the sensor values message was not received. True means the message was not received.

theta_dot_fr: estimated value of the front right wheel angular velocity.

theta_dot_fl: estimated value of the front left wheel angular velocity.

theta_dot_ar: estimated value of the aft right wheel angular velocity.

theta_dot_al: estimated value of the aft left wheel angular velocity.

torque_fr_out: intermediate calculation of the torque that should be applied on the front right wheel without taking the vehicle speed into account.

torque_fl_out: intermediate calculation of the torque that should be applied on the front left wheel without taking the vehicle speed into account.

torque_ar_out: intermediate calculation of the torque that should be applied on the aft right wheel without taking the vehicle speed into account.

torque_al_out: intermediate calculation of the torque that should be applied on the aft left wheel without taking the vehicle speed into account.

v_est: estimated velocity of the vehicle.

torque_fr_set: torque value that is going to be sent for the front right wheel.

torque_fl_set: torque value that is going to be sent for the front left wheel.

torque_ar_set: torque value that is going to be sent for the aft right wheel.

torque_al_set: torque value that is going to be sent for the aft left wheel.

output_control1_msg: buffer with the message received at the output controller 1 communication task.

output_control1_rx: flag indicating if the output controller 1 was received. True means the message was received.

output_control2_msg: buffer with the message received at the output controller 2 communication task.

output_control2_rx: flag indicating if the output controller 2 was received. True means the message was received.

error_log_1: buffer containing the first part of the error log information.

error_log_2: buffer containing the second part of the error log information.

error_log_3: buffer containing the third part of the error log information.

TTA CAN System/ TTA System/ Matrix Cycle Manager/ Controller Matrix Cycle/ controller basic cycle 1/ Calculate steering

delta_dot_f_prev: stored value of the previous calculation of the angular velocity at the front axle.

delta_dot_a_prev: stored value of the previous calculation of the angular velocity at the aft axle.

TTA CAN System/ TTA System/ Matrix Cycle Manager/ Controller Matrix Cycle/ controller basic cycle 1/ Calculate torque

delta_f_error_prev: stored value of the previous calculation of the steering error at the front axle.

delta_a_error_prev: stored value of the previous calculation of the steering error at the aft axle.

torque_fr_derivative: derivative part in the PID of the torque at the front right wheel.

torque_ar_derivative: derivative part in the PID of the torque at the aft right wheel.

TTA CAN System/ TTA System/ Matrix Cycle Manager/ Controller Matrix Cycle/ controller basic cycle 1/ Calculate speed

speed_error_prev: stored value of the previous calculation of the vehicle speed error.

speed_derivative: derivative part in the PID of the speed calculation.

v_ctreff: control effort velocity, as the result of the speed PID.

TTA CAN System/ TTA System/ Matrix Cycle Manager/ Controller Matrix Cycle/ controller basic cycle 1/ COMP Task - TMR

tau_fr_out1: front right value of the torque received at the output controller 1 communication task.

tau_fl_out1: front left value of the torque received at the output controller 1 communication task.

tau_ar_out1: aft right value of the torque received at the output controller 1 communication task.

tau_al_out1: aft left value of the torque received at the output controller 1 communication task.

tau_fr_out2: front right value of the torque received at the output controller 2 communication task.

tau_fl_out2: front left value of the torque received at the output controller 2 communication task.

tau_ar_out2: aft right value of the torque received at the output controller 2 communication task.

tau_al_out2: aft left value of the torque received at the output controller 2 communication task.

set_miss_out1: flag received at output controller 1 indicating if the set values message was missed at that board. True means the message was missed.

sensor_miss_out1: flag received at output controller 1 indicating if the sensor values message was missed at that board. True means the message was missed.

set_miss_out2: flag received at output controller 2 indicating if the set values message was missed at that board. True means the message was missed.

sensor_miss_out2: flag received at output controller 2 indicating if the sensor values message was missed at that board. True means the message was missed.

TTA CAN System/ TTA System/ Matrix Cycle Manager/ Input Generator Matrix Cycle

msg_count_DEBUG: counter monitoring the message counter arriving at the receiving end.

Sync_bc0_missed_counter: counter of all reference messages missed in basic cycle 0.

Sync_bc1_missed_counter: counter of all reference messages missed in basic cycle 1.

TTA CAN System/ TTA System/ Matrix Cycle Manager/ Vehicle Emulator Matrix Cycle

msg_count_DEBUG: counter monitoring the message counter arriving at the receiving end.

Sync_bc0_missed_counter: counter of all reference messages missed in basic cycle 0.

Sync_bc1_missed_counter: counter of all reference messages missed in basic cycle 1.

OutController_missed_counter: counter of all output emulator messages missed.

theta_fr_incr: front right wheel displacement sensor value sent to the controller.

theta_fl_incr: front left wheel displacement sensor value sent to the controller.

theta_ar_incr: aft right wheel displacement sensor value sent to the controller.

theta_al_incr: aft left wheel displacement sensor value sent to the controller.

torque_fr: front right torque value received from the controller.

torque_fl: front left torque value received from the controller.

torque_ar: aft right torque value received from the controller.

torque_al: aft left torque value received from the controller.

rx_data_incr_time: time in ticks since the last controller cycle was performed.

controller_msg_missed: flag indicating if the controller output message was missed. True means the message was not received.

TTA CAN System/ TTA System/ Matrix Cycle Manager/ Vehicle Emulator Matrix Cycle/ vehicle emulator basic cycle 0

new_msg_Rx: flag indicating is a new message was received during the last communication task. True means the message was received.

Msg_Rx: buffer with the message received during the last communication task.

Reset_Board: flag controlling if a board shall be reset or not. True means it shall be reset.

delta_dot_f: angular velocity at the front axle.

delta_dot_a: angular velocity at the aft axle.

speed_act: vehicle speed.

TTA CAN System/ TTA System/ Matrix Cycle Manager/ Vehicle Emulator Matrix Cycle/ vehicle emulator basic cycle 0/ COMP Task - Vehicle Emulator Calculate Steer

delta_double_dot_f_prev: stored value of the previous calculation of the angular acceleration at the front axle.

delta_double_dot_a_prev: stored value of the previous calculation of the angular acceleration at the aft axle.

TTA CAN System/ TTA System/ Matrix Cycle Manager/ Vehicle Emulator Matrix Cycle/ vehicle emulator basic cycle 0/ COMP Task - Vehicle Emulator Calculate Velocity

avg_force: average force of the actuators.

avg_force_prev: stored value of the previous calculation of the average force of the actuators.

TTA CAN System/ TTA System/ Matrix Cycle Manager/ Vehicle Emulator Matrix Cycle/ vehicle emulator basic cycle 0/ COMP Task - Vehicle Emulator Calculate Angles

theta_dot_fr_ve: angular velocity at the front right wheel.

theta_dot_fl_ve: angular velocity at the front left wheel.

theta_dot_ar_ve: angular velocity at the aft right wheel.

theta_dot_al_ve: angular velocity at the aft left wheel.

theta_dot_fr_prev_ve: stored value of the previous calculation of the angular velocity at the front right wheel.

theta_dot_fl_prev_ve: stored value of the previous calculation of the angular velocity at the front left wheel.

theta_dot_ar_prev_ve: stored value of the previous calculation of the angular velocity at the aft right wheel.

theta_dot_al_prev_ve: stored value of the previous calculation of the angular velocity at the aft left wheel.

theta_fr: angular position at the front right wheel.

theta_fl: angular position at the front left wheel.

theta_ar: angular position at the aft right wheel.

theta_al: angular position at the aft left wheel.

theta_fr_prev: stored value of the previous calculation of the angular position at the front right wheel.

theta_fl_prev: stored value of the previous calculation of the angular position at the front left wheel.

theta_ar_prev: stored value of the previous calculation of the angular position at the aft right wheel.

theta_al_prev: stored value of the previous calculation of the angular position at the aft left wheel.

TTA CAN System/ TTA System/ Matrix Cycle Manager/ Vehicle Emulator Matrix Cycle/ vehicle emulator basic cycle 1

new_msg_Rx: flag indicating is a new message was received during the last communication task. True means the message was received.

Msg_Rx: buffer with the message received during the last communication task.

2.26 List of signals and parameters

Some connection lines joining together two blocks have associated a name. This represents a signal in the system that can be read by HANTune during runtime. Parameters are defined in the MATLAB startup file and can be overwritten in HANTune during runtime. The following list of signals and parameters can be found in the DAQ default list of the HANcoder_E407_TTA_Controller_v3.a2l ASAP2 file HANTune when loading up the DS_TTA_Monitor.hml file of the project. **Parameters**

COMM_Phase1: time in ticks from one message to the next in CAN channel 1 during a communication task.

COMM_Phase2: time in ticks from one message to the next in CAN channel 2 during a communication task.

COMM_Phase_init1: time in ticks from the beginning of a communication task until the first message is cast in CAN channel 1.

COMM_Phase_init2: time in ticks from the beginning of a communication task until the first message is cast in CAN channel 2.

HANtuneOverride: test parameter from the HANcoder template.

delta_a_set: steering angle set at the aft axle for the input generator.

delta_f_set: steering angle set at the front axle for the input generator.

script_run: flag to start and stop the execution of python scripts within HANTune.

speed_k_d: derivative gain for the speed loop in the controller.

speed_k_i: integral gain for the speed loop in the controller.

speed_k_p: proportional gain for the speed loop in the controller.

torque_a_k_d: derivative gain for the aft in the torque loop of the controller.

torque_a_k_i: integral gain for the aft in the torque loop of the controller.

torque_a_k_p: proportional gain for the aft in the torque loop of the controller.

torque_f_k_d: derivative gain for the front in the torque loop of the controller.

torque_f_k_i: integral gain for the front in the torque loop of the controller.

torque_f_k_p: proportional gain for the front in the torque loop of the controller.

v_set: vehicle velocity set for the input generator.

Signals

BC0_sync_processed_ing_s: missed reference messages in the basic cycle 0 of the input generator.

BC0_sync_processed_s: missed reference messages in the basic cycle 0 of the controller.

BC0_sync_processed_vem__s: missed reference messages in the basic cycle 0 of the vehicle emulator.

BC0_vote1_processed_s: boolean indicating if vote1 message was received in the controller. True means the message was received.

BC0_vote2_processed_s: boolean indicating if vote2 message was received in the controller. True means the message was received.

BC0_vote3_processed_s: boolean indicating if vote3 message was received in the controller. True means the message was received.

BC1_sync_processed_ing_s: missed reference messages in the basic cycle 1 of the input generator.

BC1_sync_processed_s: missed reference messages in the basic cycle 1 of the controller.

BC1_sync_processed_vem__s: missed reference messages in the basic cycle 1 of the vehicle emulator.

Board_1_error_counter_s: complete number of missed messages from board 1.

Board_2_error_counter_s: complete number of missed messages from board 2.

Board_3_error_counter_s: complete number of missed messages from board 3.

Board_ID_s: board ID value of the board.

Desync_Sync_bc0_ing_s: desynchronization value at the basic cycle 0 of the input generator.

Desync_Sync_bc0_s: desynchronization value at the basic cycle 0 of the controller.

Desync_Sync_bc0_vem_s: desynchronization value at the basic cycle 0 of the vehicle emulator.

Desync_Sync_bc1_ing_s: desynchronization value at the basic cycle 1 of the input generator.

Desync_Sync_bc1_s: desynchronization value at the basic cycle 1 of the controller.

Desync_Sync_bc1_vem_s: desynchronization value at the basic cycle 1 of the vehicle emulator.

LedValue: HANcoder template's led value.

Master_ID_ing_s: board ID of the master in the controller from the input generator board.

Master_ID_s: board ID of the master in the controller.

Master_ID_vem_s: board ID of the master in the controller from the vehicle emulator board.

Role_ID_ing_s: role of the input generator. It is set to 0.

Role_ID_s: role of the controller board. It is set to 1 for the master and 2 for the slaves.

Role_ID_vem_s: role of the vehicle emulator. It is set to 0.

RxCAN1_counter_s: number of times the CAN1 receive system has been called.

RxCAN2_counter_s: number of times the CAN2 receive system has been called.

RxID_CAN1_s: ID value of the message being listened to in CAN channel 1. The different IDs for the messages are defined in the MATLAB startup file.

RxID_CAN2_s: ID value of the message being listened to in CAN channel 2. The different IDs for the messages are defined in the MATLAB startup file.

Rx_init_LT: local time received at the initialization.

Rx_init_bc: basic cycle received at the initialization.

Rx_init_id: board ID received at the initialization.

Rx_init_mc: message counter received at the initialization.

Rx_msg_ticks_CAN1: time in ticks at which the message was received in CAN channel 1.

Rx_msg_ticks_CAN2: time in ticks at which the message was received in CAN channel 2.

Rx_temporal_msg_CAN1: temporal message received in CAN channel 1.

Rx_temporal_msg_CAN2: temporal message received in CAN channel 2.

SI_CPUload: HANcoder's template CPU load.

SI_FreeHeap: HANcoder's template free heap.

SI_FreeStack: HANcoder's template free stack.

TM1_timeout_counter_s: counter with the number of misses of vote1.

TM2_timeout_counter_s: counter with the number of misses of vote2.

TM3_timeout_counter_s: counter with the number of misses of vote3.

TMR_miss1_sensor: boolean received at the master at output control 1 indicating if the sensor message was missed by the slave that sent the message.

TMR_miss1_set: boolean received at the master at output control 1 indicating if the set message was missed by the slave that sent the message.

TMR_miss2_sensor: boolean received at the master at output control 2 indicating if the sensor message was missed by the slave that sent the message.

TMR_miss2_set: boolean received at the master at output control 2 indicating if the set message was missed by the slave that sent the message.

TMR_miss_out1: boolean indicating if out controller 1 message was missed.

TMR_miss_out2: boolean indicating if out controller 2 message was missed.

TMR_miss_sensor: boolean indicating if the sensor message was missed by the master.

TMR_miss_set: boolean indicating if the set message was missed by the master.

TxCAN1_counter_s: number of times the CAN1 transmit system has been called.

TxCAN2_counter_s: number of times the CAN2 transmit system has been called.

TxID_CAN1_s: ID value of the message being transmitted through CAN channel 1. The different IDs for the messages are defined in the MATLAB startup file.

TxID_CAN2_s: ID value of the message being transmitted through CAN channel 2. The different IDs for the messages are defined in the MATLAB startup file.

Tx_temporal_msg_CAN1: temporal message transmitted through CAN channel 1.

Tx_temporal_msg_CAN2: temporal message transmitted through CAN channel 2.

Vote_s: Board ID of the board that shall be the master.

basic_cycle_s: basic cycle counter.

error_log1_s: first part of the error log.

error_log2_s: second part of the error log.

error_log3_s: third part of the error log.

local_ticks_interrupt: local time of the board in ticks.

mc_counter_s: matrix cycle counter with the amount of matrix cycles that have passed since the last controller calculations. This is deprecated in the current version of the software.

msg_count_DEBUG_ing_s: number of times a message with a message counter of 2 or more has arrived at a communication task in the input generator.

msg_count_DEBUG_s: number of times a message with a message counter of 2 or more has arrived at a communication task in the controller.

msg_count_DEBUG_vem_s: number of times a message with a message counter of 2 or more has arrived at a communication task in the vehicle emulator.

new_sensor_rx_s: boolean indicating if the sensor values message was received at the controller. When true the message was received.

new_sensor_rx_vem_s: boolean indicating if the output emulator message was received at the vehicle emulator. When true the message was received.

new_set_rx_s: boolean indicating if the set values message was received at the controller. When true the message was received.

out1_miss_counter_s: number of output controller 1 messages missed at the master.

out1_rx_s: boolean indicating if the output controller 1 message was received at the master. When true the message was received.

out2_miss_counter_s: number of output controller 2 messages missed at the master.

out2_rx_s: boolean indicating if the output controller 2 message was received at the master. When true the message was received.

outcont_miss_counter_vem: number of output emulator messages missed at the vehicle emulator.

reset_ing_s: boolean indicating if the input generator shall reset. When true the board shall reset.

reset_s: boolean indicating if the controller board shall reset. When true the board shall reset.

reset_vem_s: boolean indicating if the vehicle emulator shall reset. When true the board shall reset.

rx_data_incr_time_s: time in ticks since the last controller operations.

script_run_s: value of the script_run parameter. It is necessary to have the parameter as a signal too so the python script can read its value.

sensor_miss_counter_s: number of missed sensor values messages missed.

set_miss_counter_s: number of missed set values messages missed.

simulation_time: time in seconds the board has been running (regardless of the matrix cycle running).

sync_bc0_miss_counter_ing: number of missed reference messages in basic cycle 0 missed by the input generator.

sync_bc0_miss_counter_s: number of missed reference messages in basic cycle 0 missed by the controller.

sync_bc0_miss_counter_vem: number of missed reference messages in basic cycle 0 missed by the vehicle emulator.

sync_bc1_miss_counter_ing: number of missed reference messages in basic cycle 1 missed by the input generator.

sync_bc1_miss_counter_s: number of missed reference messages in basic cycle 1 missed by the controller.

sync_bc1_miss_counter_vem: number of missed reference messages in basic cycle 1 missed by the vehicle emulator.

tau_al1_disagree_s: boolean indicating disagreement between the master and output controller 1 aft left wheel torque output calculations. True means they disagree.

tau_al2_disagree_s: boolean indicating disagreement between the master and output controller 2 aft left wheel torque output calculations. True means they disagree.

tau_al_disagree_s: boolean indicating disagreement between the output controller 1 and output controller 2 aft left wheel torque output calculations. True means they disagree.

tau_ar1_disagree_s: boolean indicating disagreement between the master and output controller 1 aft right wheel torque output calculations. True means they disagree.

tau_ar2_disagree_s: boolean indicating disagreement between the master and output controller 2 aft right wheel torque output calculations. True means they disagree.

tau_ar_disagree_s: boolean indicating disagreement between the output controller 1 and output controller 2 aft right wheel torque output calculations. True means they disagree.

tau_fl1_disagree_s: boolean indicating disagreement between the master and output controller 1 front left wheel torque output calculations. True means they disagree.

tau_fl2_disagree_s: boolean indicating disagreement between the master and output controller 2 front left wheel torque output calculations. True means they disagree.

tau_fl_disagree_s: boolean indicating disagreement between the output controller 1 and output controller 2 front left wheel torque output calculations. True means they disagree.

tau_fr1_disagree_s: boolean indicating disagreement between the master and output controller 1 front right wheel torque output calculations. True means they disagree.

tau_fr2_disagree_s: boolean indicating disagreement between the master and output controller 2 front right wheel torque output calculations. True means they disagree.

tau_fr_disagree_s: boolean indicating disagreement between the output controller 1 and output controller 2 front right wheel torque output calculations. True means they disagree.

theta_al_s: angular position of the aft left wheel calculated at the vehicle emulator.

theta_al_stored_s: angular position increment of the aft left wheel received at the controller from the vehicle emulator.

theta_ar_s: angular position of the aft right wheel calculated at the vehicle emulator.

theta_ar_stored_s: angular position increment of the aft right wheel received at the controller from the vehicle emulator.

theta_fl_s: angular position of the front left wheel calculated at the vehicle emulator.

theta_fl_stored_s: angular position increment of the front left wheel received at the controller from the vehicle emulator.

theta_fr_s: angular position of the front right wheel calculated at the vehicle emulator.

theta_fr_stored_s: angular position increment of the front right wheel received at the controller from the vehicle emulator.

torque_al_out1_s: aft left wheel torque received at the master from the output controller 1 message.

torque_al_out2_s: aft left wheel torque received at the master from the output controller 2 message.

torque_al_set_out_s: aft left wheel torque result from the triple modular redundancy at the master.

torque_al_set_s: aft left wheel torque result at the master before the triple modular redundancy.

torque_ar_out1_s: aft right wheel torque received at the master from the output controller 1 message.

torque_ar_out2_s: aft right wheel torque received at the master from the output controller 2 message.

torque_ar_set_out_s: aft right wheel torque result from the triple modular redundancy at the master.

torque_ar_set_s: aft right wheel torque result at the master before the triple modular redundancy.

torque_fl_out1_s: front left wheel torque received at the master from the output controller 1 message.

torque_fl_out2_s: front left wheel torque received at the master from the output controller 2 message.

torque_fl_set_out_s: front left wheel torque result from the triple modular redundancy at the master.

torque_fl_set_s: front left wheel torque result at the master before the triple modular redundancy.

torque_fr_out1_s: front right wheel torque received at the master from the output controller 1 message.

torque_fr_out2_s: front right wheel torque received at the master from the output controller 2 message.

torque_fr_set_out_s: front right wheel torque result from the triple modular redundancy at the master.

torque_fr_set_s: front right wheel torque result at the master before the triple modular redundancy.

v_ctreff_s: velocity control effort result of the controller PID velocity loop.

v_est_s: estimated velocity of the vehicle in the controller.

v_est_vem_s: estimated velocity of the vehicle in the vehicle emulator.

v_integral_s: velocity integral component of the PID at the controller.

v_set_s: velocity set by the input generator.

votes_Board_1_DEBUG: number of votes received to make board 1 the master.

votes_Board_2_DEBUG: number of votes received to make board 2 the master.

votes_Board_3_DEBUG: number of votes received to make board 3 the master.

3 System debug and prototype measurements

There are different mechanisms that have been developed to ensure the system is going to behave as expected or, otherwise, monitor it to identify what might be wrong. This section starts with the test programs, that help in the diagnosis of the CAN connections, controller calculations, message encryption and message reception. Afterwards, the monitor programs used, HANTune and the logic analyzer, are explained focusing on the different measurements proposed in the prototype.

3.1 Test programs

There are four test programs that help in the code development before introducing it into the TTA software. The first, the double CAN communication **CAN_2x_Test**, helps identifying if there is some problem in the CAN communication between the boards. It differentiates between a sender board and a receiver board using the digital inputs D2 and D3. When pressing the WKUP button the sender sends a message with a counter value. If the connections are fine the receiver will get the counter value from the sender. To actually verify that the system is working HANTune is used to monitor the different signals. As shown in figure (91), it is possible to see if a board has been connected as a sender or receiver. This can be changed during runtime making a different configuration with the 5 V source and the digital inputs. It is also possible to make a board both sender and receiver. If the connections are right, when pressing the WKUP button on a sender board the same number that appears on its TX_MSG_1 and TX_MSG_2 windows will appear in the RX_MSG_1 and RX_MSG_2 windows of the receiver boards. Disconnecting the CAN channels 1 or 2 will make the corresponding number not to be transmitted.

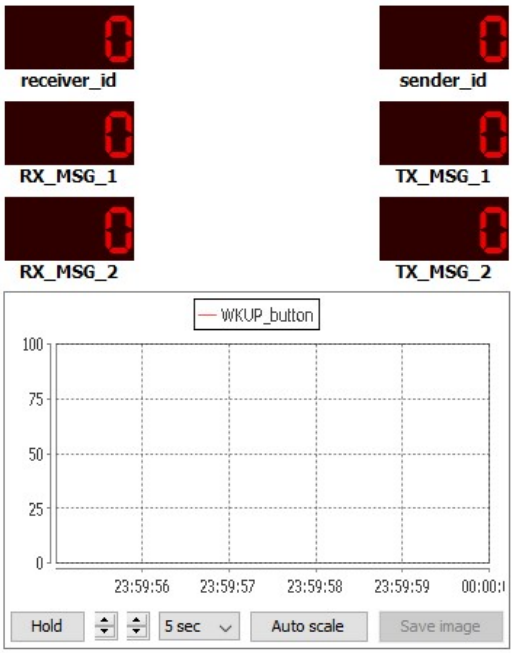


Figure 91: Double CAN communication test window in HANTune.

The **Controller_ValueDomain_TEST** checks within a single board if the controller calculations are correct. The same board acts as input generator, controller and vehicle emulator, making the operations from basic cycle 1 in the controller and basic cycle 0 in the vehicle emulator. No CAN communication is performed to simplify the system, and HANTune monitoring is possible to see all required signals within the same HANTune instance using USB connection. The current version works fine for the speed control loop, but the steer loop requires further investigation.

Because CAN communication sends eight data bytes of information, it is required to transform float values into unsigned integers of eight bits (uint8). The **float2uint8_code_decode_TEST** program performs a code/decode conversion of a float number, checking if the coding operation works fine. In figure (92) it can be seen how the operations can be monitored with the display blocks in Simulink. This program is run within a Simulink simulation, without deployment. More information about how the float data is encypted can be found in section 2.24.3.

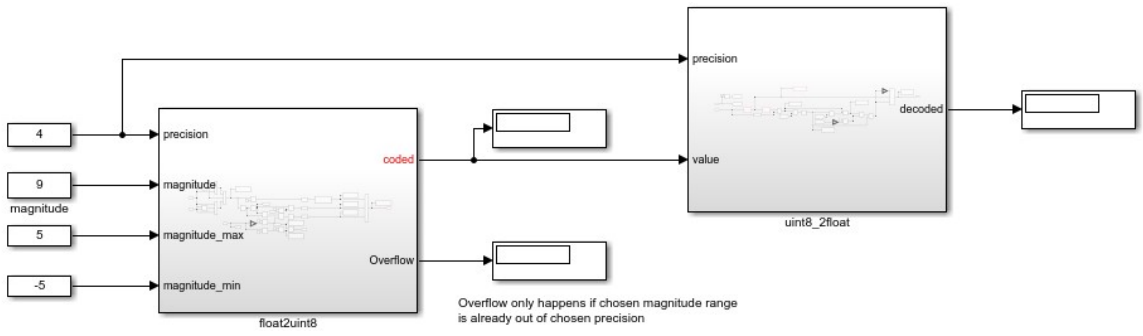


Figure 92: Program coding a float value into an unsigned8 integer and back to float.

The last test puts together all the previous software to check if the whole communication and encryption is possible at the same time. With **CAN_COMM_Test** different float values can be transmitted through CAN channels 1 and 2. This test has a high degree of customization, so, as it can be seen in figure (93), it can be troublesome to set all the different parameters at the beginning. However, once the parameters are set for the test, it allows for a lot of different tests in both communication and data encryption. This test requires deployment of the program onto at least two boards connected through CAN, like the 2xCAN test. The messages are also sent by pressing the WKUP button at the sender board and sender and receiver are distinguished by 5 V connections at the D2 and D3 digital pins.

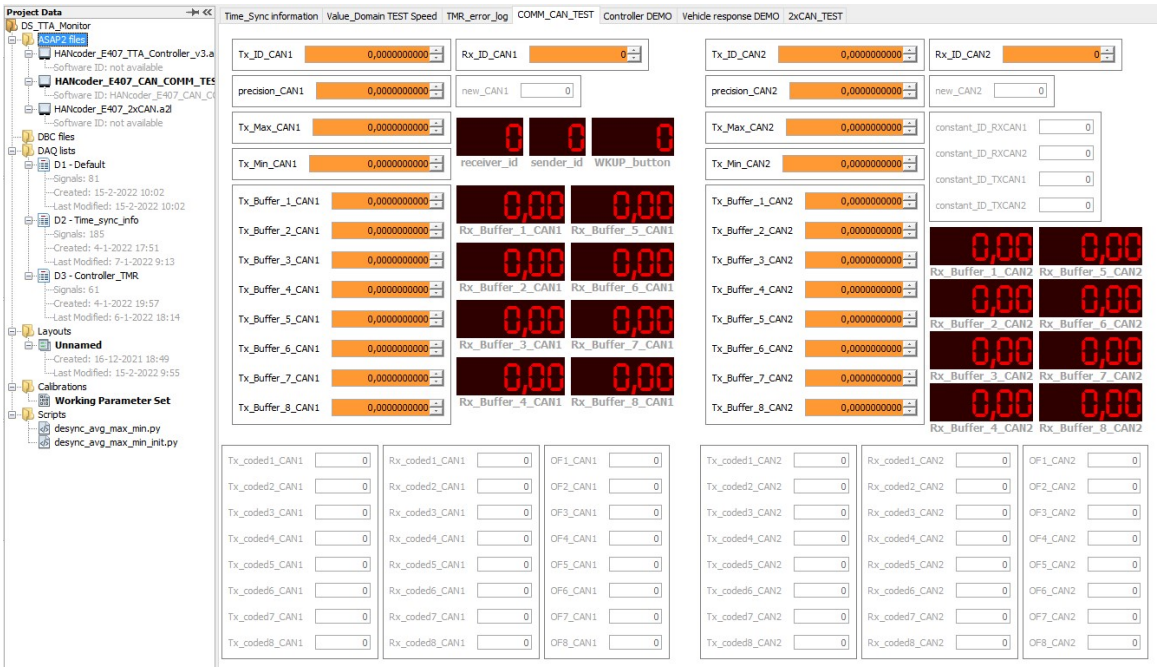


Figure 93: HANTune window to check the CAN communication float exchange.

3.2 Prototype measurements

There are seven different measurements that have been performed to the prototype during the master thesis project. The first two are made with HANTune, while the others require the logic analyzer and postprocessing with Python. The next sections present the steps required to reproduce each measurement.

3.2.1 HANTune measurements

HANTune monitors the software signals during runtime. Its fastest sample frequency is 100 Hz so it is not able to record the changes in the software that happen every tick. However, there are some important measurements that we can do with it: the missing messages and the desynchronization.

When connecting an ensemble of boards to HANTune there are some things to take into account. Every board can be connected to a single instance of HANTune. The current version of the program is prepared to connect each board via USB. When connecting a board to a computer, the operating system assigns a COM port to the board connection. It is possible to select to which COM port the HANTune instance must connect by going to the Communication tab and choosing Communication Settings. In the USB/UART port you can choose the UART port. Using the Device Manager application in your computer you can see which COM ports have been assigned to the board connections.
The HANTune file with the information required to perform the measurements presented in this section is DS_TTA_Monitor, included in the GitHub repository mentioned in section 2.1. This file counts with three ASAP2 files, one from a different Simulink project build: the TTA_Controller, the CAN_COMM_TEST and the 2xCAN_TEST. The HANTune measurements explained next use the TTA_Controller ASAP2 file. Every ASAP2 file counts with different signals that can be checked in the DAQ lists from the ASAP2 elements tab. The Default DAQ list contains all the signals from the project, but if there are too many signals only some of them are presented. It is possible to create custom DAQ lists with a reduced number of signals. This is important to do when there could be too much CPU load, as the higher the number of signals being monitored by HANTune, the higher the impact on the board processor.

Missing messages
There are times when a message sent from a board is not received by the other board. Different causes on why this happens are explored in the master project report. This is a communication problem that must be assessed, as there is not yet an explanation on why there are messages being missed. It is important to at least reduce the misses significantly to ensure the system is safe. To do this it is required to investigate the source of the error exploring the CAN traffic. The current version of the software and the tools used for the master thesis could not make this investigation possible, but HANTune can show the number of messages that are missed every matrix cycle looking at the appropriate signals. When looking at the top right panel in the Time_Sync information of DS_TTA_Monitor, the different missing messages counters are displayed. These are presented in figure (94).

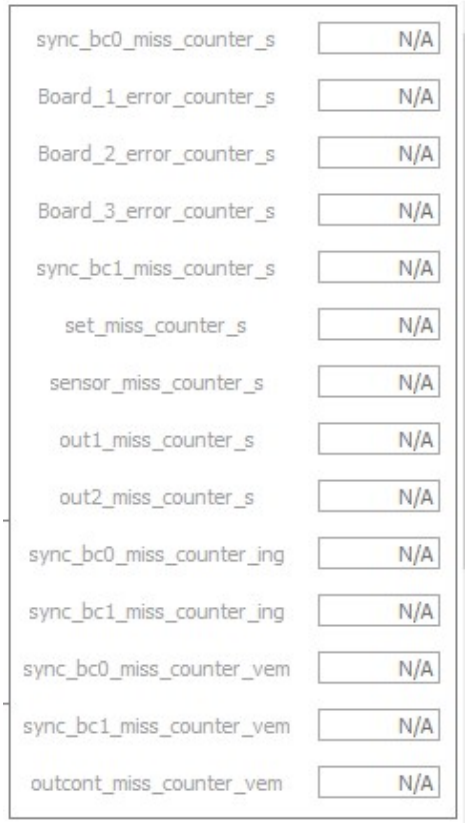


Figure 94: HANTune window with the missed messages signals.

The measurements presented in the results and missed messages appendix of the thesis report correspond to the values presented by these signals after fifteen minutes of ensemble operation.

Desynchronization
When the master board in the ensemble sends the reference message, the slave board receiving it compares its own local time with the moment when this reference message was expected to be received. The difference between these two times is the desynchronization between the slave and the master. The highest desynchronization in the ensemble defines the ensemble precision. Recording the desynchronization values of the boards over a certain amount of time and taking the mean, maximum and minimum value can give us an estimation of the ensemble precision. HANTune only offers the signal values, but it is possible to perform operations with them, such as the average over time, using Python scripts. In the Project data tab of HANTune there is a Script folder with two scripts, desync_avg_max_min.py and its initialization. The initialization script create the script signals in the Script elements tab. It is necessary to run this script after making the connection with the boards to create these elements before starting the measurement. Then, once the boards are already connected and their local time is running, the script_run parameter shall be set and the desync_avg_max_min.py must be run to start the measurement.

The script_run parameter controls the execution of the script, so when it is reset the script stops. This is done this way because when forcing the script to stop within HANTune, the instance freezes and its task shall be killed manually by the user with the Task Manager. The current desync signals in the HANTune window are not usable, as those are the scripted elements from the last measurement. These signals should be replaced by the new desync signals created by the initialization script. The desynchronization measurements from the project report correspond to the values shown by these signals after fifteen minutes operation. More information about HANTune scripts can be found in its documentation page.

3.2.2 Logic Analyzer measurements

The local time of the boards in the ensemble run at 10 kHz frequency, so a tool with a higher sample frequency than that is required to monitor most of the system behaviour. A logic analyzer was chosen for its simplicity and cost. There are different systems and variables in the software in charge of toggling digital outputs to extract information of the behaviour. Every time something new happens, such as a task activation, a communication failure or a message arrive, the digital output toggles giving temporal information of what is going on.

The software used to connect with the logic analyzer was Saleae Logic. This software counts with guides on how to use it and an active community that solves specific doubts. When exporting the digital data collected to a .csv file just the times when an event happened are recorded to minimize the file size. This requires postprocessing in Python to separate the signals and get the differences in time from one event to the next.

Task activation
The first measurement for the logic analyzer toggles the digital pin A0 every time a task is activated. This helps visualizing when each different task starts one after another, see if there is any task activation missing and check if communication and computation tasks last as long as expected. It is also possible to see when a message is sent in a communication task of a board and when it is received in another board. The variable LA_Task_Act in the MATLAB startup document must be set to true for this measurement to happen. This measurement is very limited though, as it does not provide information about which task has been activated. It is possible to extract a temporal flow of what activates when, but activating every task with programs as big as the TTA_Controller make things messy very quickly. This measurement was very useful at the beginning of development when the amount of task activating was lower. It is possible to add or remove A0 toggle flags in the code to change the tasks being activated for a specific check, but having an internal log showing the evolution of different variables in the system would probably be more useful to extract information of the task activation.

Communication failure
Setting the LA_COMM_Err to true makes digital pin A1 to toggle during the Check_Timeouts task every time a message has been missed in a communication task during that basic cycle. This, along with the task activation measurement or the communication delay measurement, can be useful to try to find the cause of the missing messages problem.

Execution time
The execution time is measured toggling the digital pin A2 when the TTA_CAN system is activated and then toggling the digital pin A3 later after the CAN systems have finished their operations. The execution time of the matrix cycle during that tick (activation) is the time since it started (A2) until it finished operation (A3). This means that two pins per board are required to make this measurement. It is not possible to do it with a single pin because the same digital pin block cannot be called twice during the same code run. This measurement requires the variable LA_MEX_Flag to be set to true in the MATLAB file. The python script wrote to handle the execution times measured is execution_times.py. It takes the information from three boards (six channels from the logic analyzer) and returns the average, maximum and minimum values of the execution time for each individual task in the matrix cycle. It also counts with a write_data procedure to manually write .txt files with the execution times of a single board. For example, to write the execution times of the input generator collected in the first two channels of the logic analyzer, the write_data procedure should be invoked this way:

```
>> write_data('data_board4.txt',execution_times['b1']['general']['ET'])
```

Once all .txt files have been written down, the postprocess_execution_times.py can be invoked to plot all the execution times together.

Clock granularity
The clock granularity is measured by taking the difference in time since the TTA_CAN is activated until it is activated again in the next IRQ. It uses digital pin A4 and needs LA_Clock_Gr to be true to activate. The process of the data acquired by the logic analyzer is done for each individual board, so a .csv file for each board should be exported. In the gr_processing.py Python script, it is possible to change the name of the .csv file that shall be processed to get a .txt file with the granularity values of that board. When there is a .txt file for each board, the postprocess() procedure from gr_processing.py plots all granularity data together.

In the GitHub repository it is also possible to find two more files, the logic analyzer file gr_0_05ms and the image clock_lag_activation.jpg. They show how when speeding up the granularity of the IRQs faster than the shortest execution time, the software activation starts to become erratic.

Communication delay
When a message is sent from one board it requires some time until it arrives to its destination and is acknowledge by the receiver board. This is the communication delay. It is measured by toggling pins D13 and D9 for the transmitting board in CAN channels 1 and 2 respectively, and toggling pins D12 and D8, for CAN 1 and 2 respectively again, in the receiving board. It requires LA_COMM_dly flag to be true. The measurement is done by making the difference of the moment when the message arrives minus the time when the message is sent. However, in the software every message is transmitted several times from the transmitter during every communication task, and it is not guaranteed that the first message will arrive at its destination (that is why several messages are sent). To simplify the analysis this measurement is done sending a single message per transmission and in case it is not received the Python program filters the communication failure. This might have an impact on the communication delay, as the network contains a less amount of messages. It is recommended to measure this communication delay by analyzing the CAN network instead.

The communication delay can be processed with the python script process_COMM_delay.py. It takes the information recorded from CAN channels 1 and 2 and automatically produces the average, maximum and minimum values of the communication delay.

Ensemble precision
The ensemble precision has already been measured by looking at the desynchronization values with HANTune as presented before. However, a measurement with the logic analyzer is proposed to double check the results. Setting the LA_E_Desync flag digital pin D10 will toggle right before the local time is updated. Measuring the difference in time between the local time update and the reference message sent gives the desynchronization between each board and the Master. It is possible to add a toggle D10 flag activation in the CAN Sent systems when the Tx_ID is Sync0 or Sync1, or combine this measurement with the task activation measurement, making A0 activate just at the transmission of the reference message.