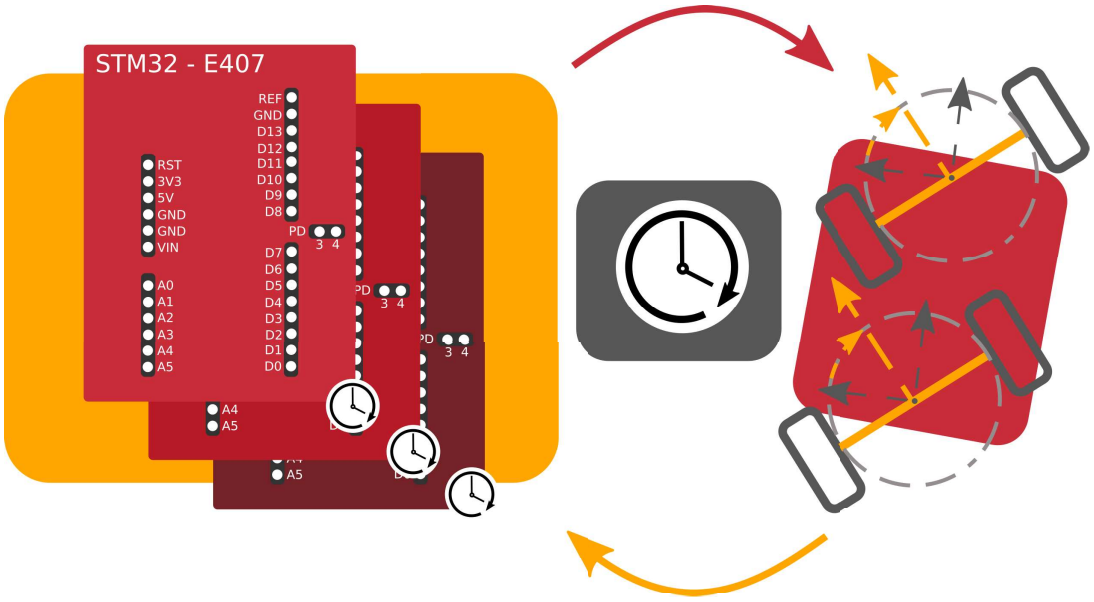
HAN_UNIVERSITY OF APPLIED SCIENCES

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).

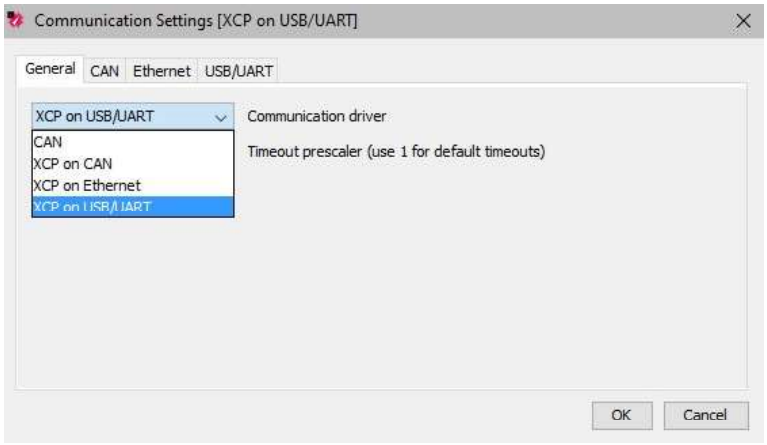


Figure 1: HANtune communication settings.

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.

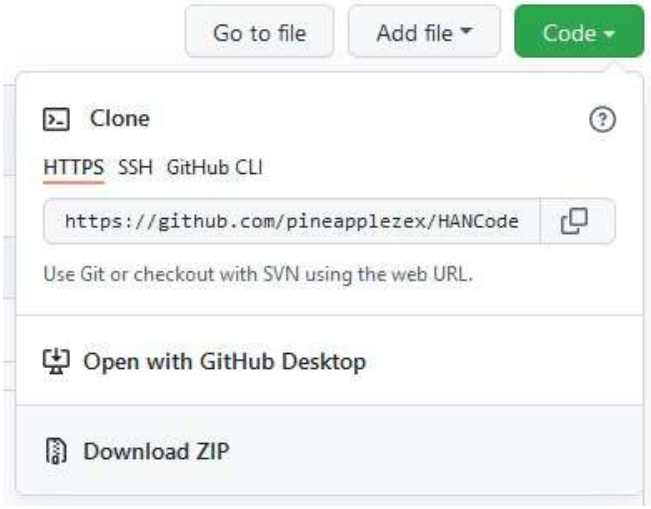


Figure 2: GitHub code pannel with the Download ZIP option at the bottom.

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.

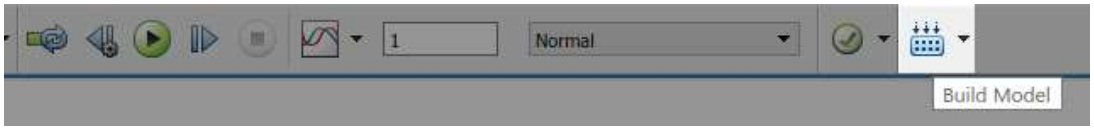


Figure 3: Simulink's build button.

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).

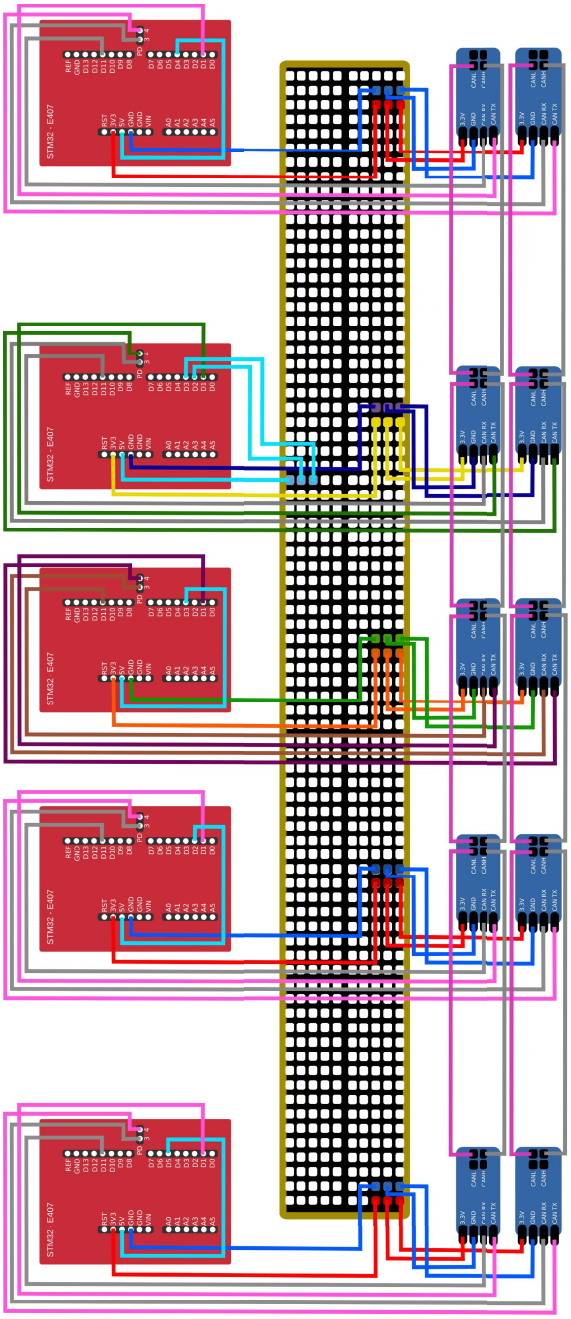


Figure 4: Electric scheme for the TTA_Controller prototype. The red squares represent the STM32-E407 boards and the blue rectangles the CAN transcievers.

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.

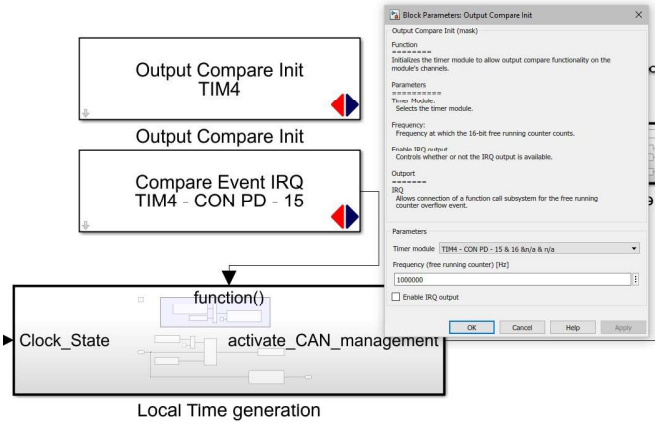


Figure 5: Output Compare Init block, where the frequency of the hardware clock can be set. A frequency of 1 MHz is set for this version of the software.

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.

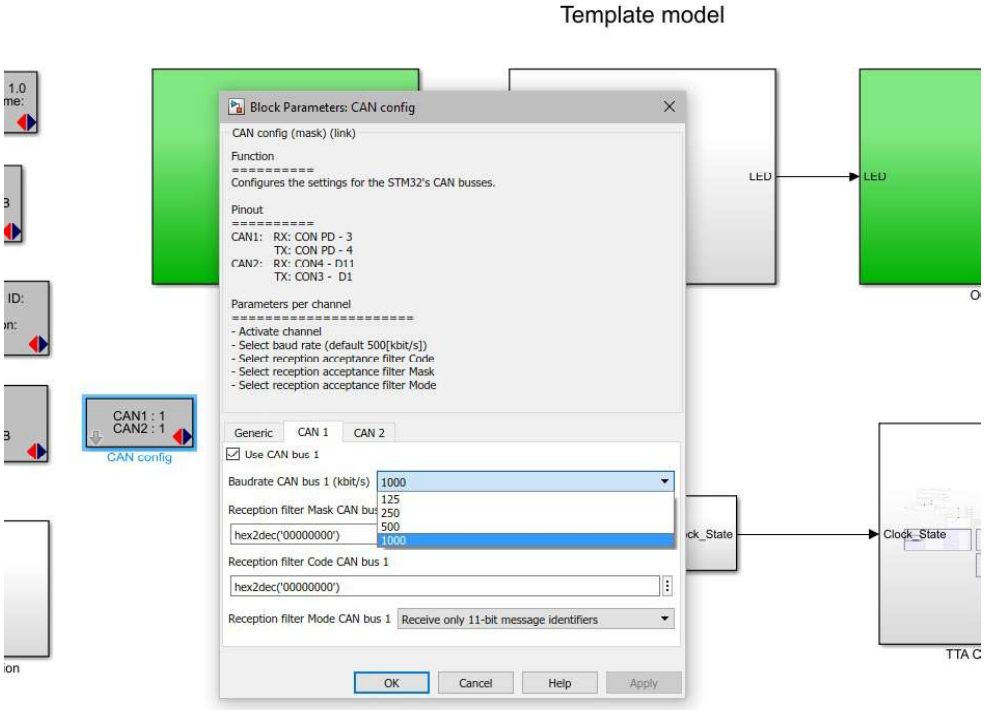


Figure 6: Baudrate selection for the CAN channels at the CAN config block.

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.

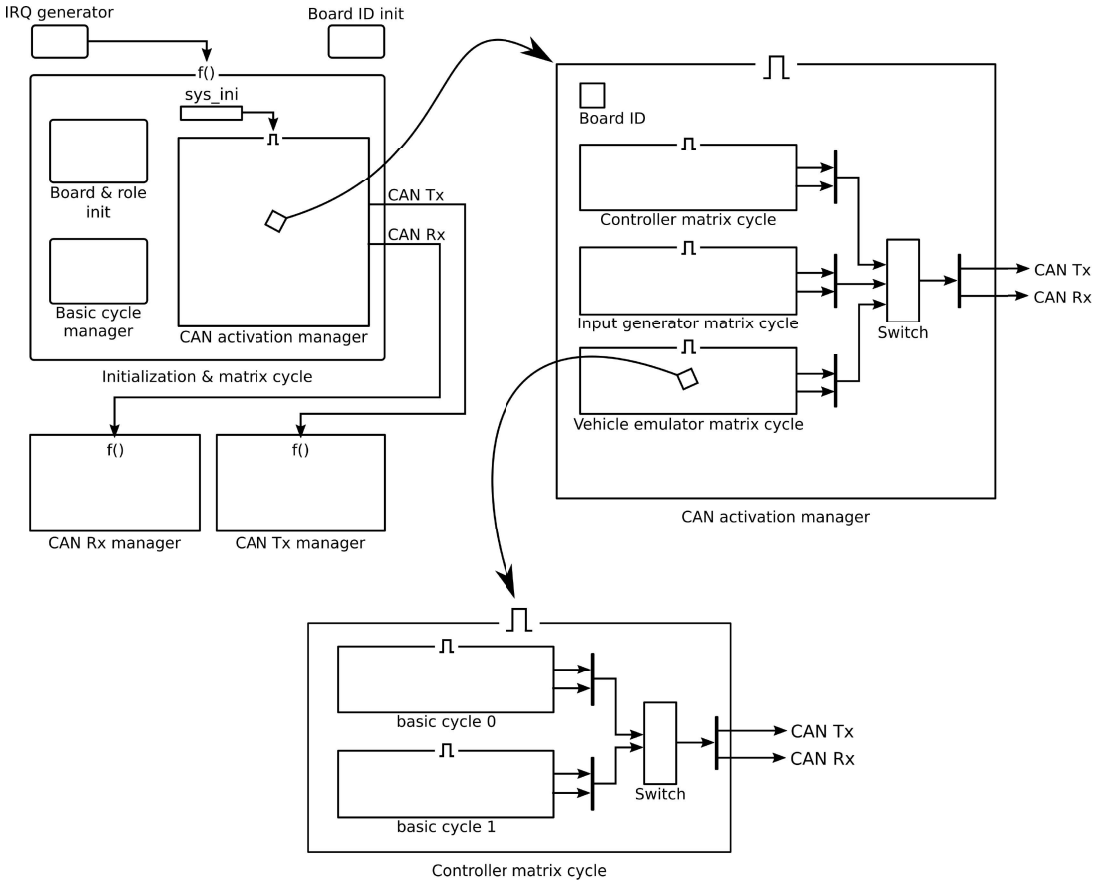


Figure 7: Software overview of the project. The TTA CAN is presented in the top left part of the figure. Some elements of the TTA system are visible inside. In the right side the Matrix manager is magnified, showing the matrix cycles. Finally, the bottom part of the picture shows the inside of a matrix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

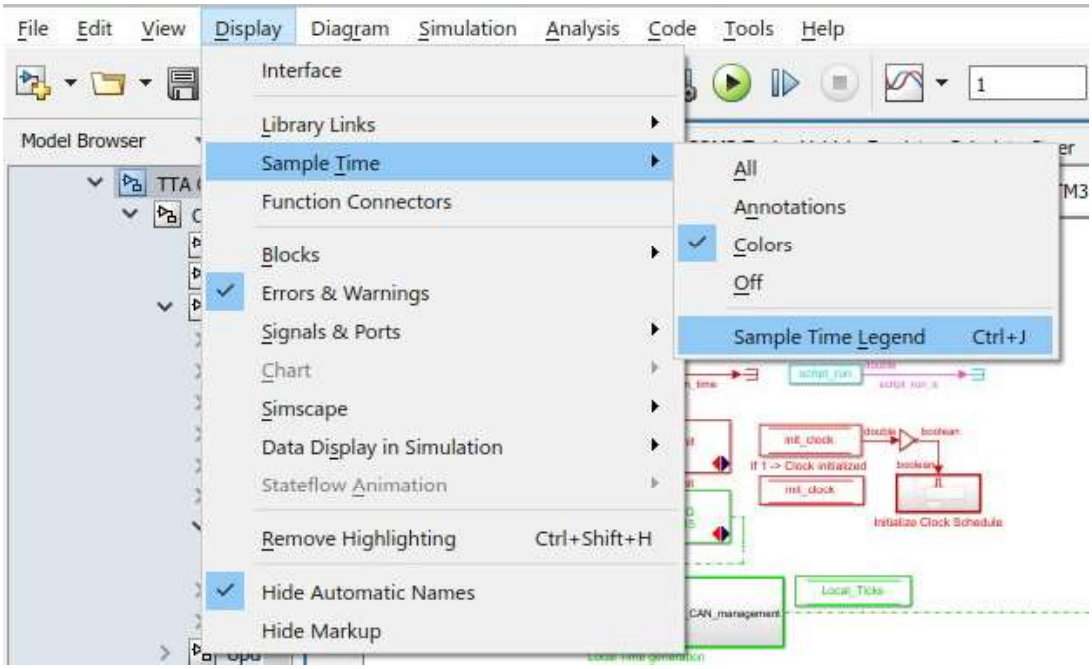


Figure 8: Menu to visualize sample time colors and their legend.

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).

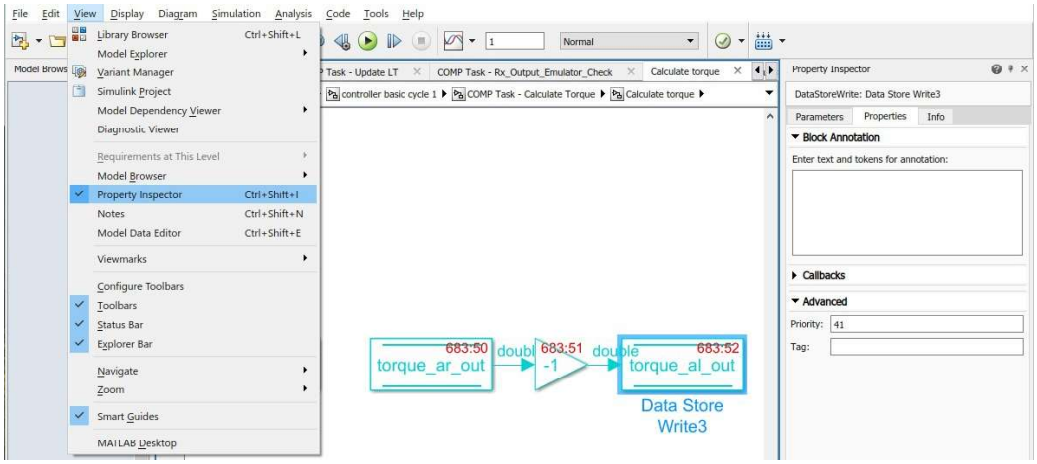


Figure 9: Menu to visualize the priority of a block.

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.

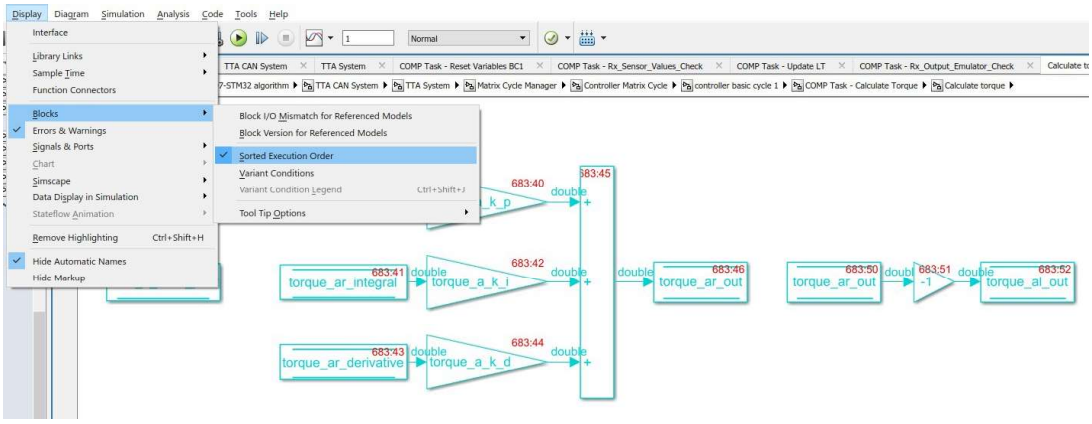


Figure 10: Menu to visualize the sorted execution order.

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.

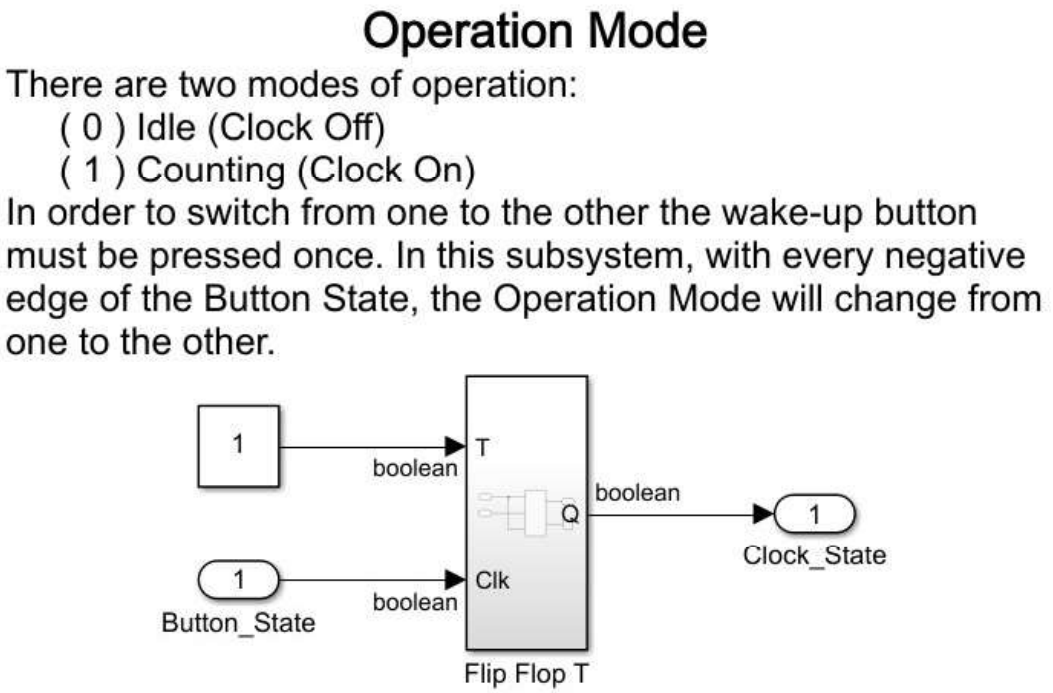


Figure 11: Operation mode system.

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.

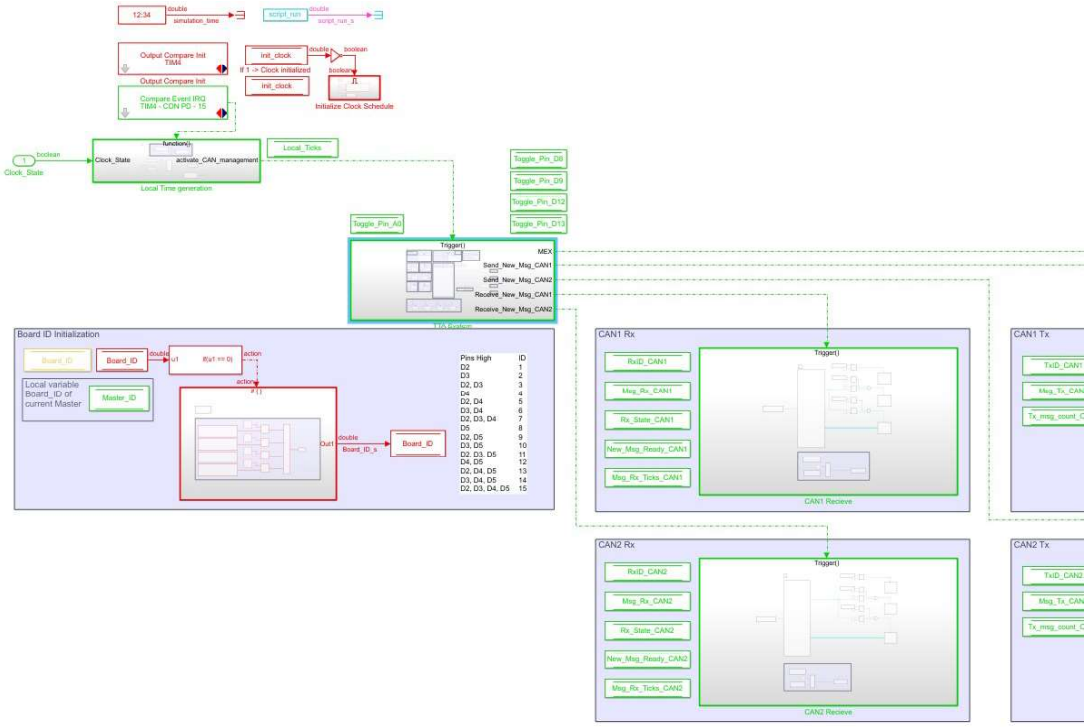


Figure 12: TTA CAN system. Systems in green are activated with function calls or IRQs, while the rest activate with the Simulink software loop.

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.

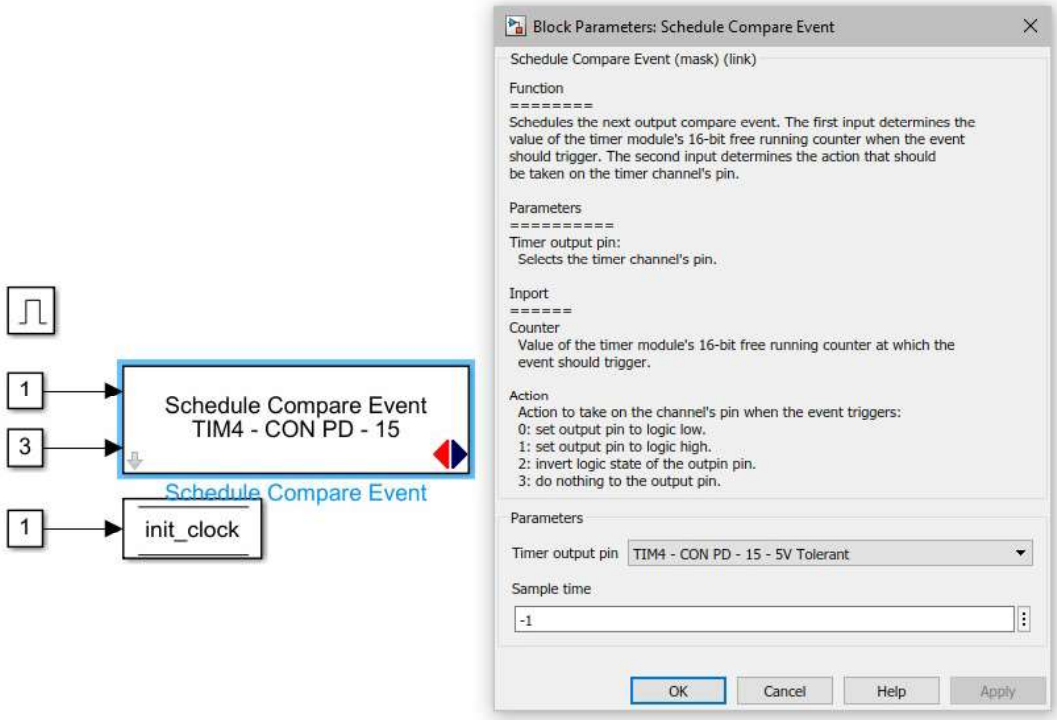


Figure 13: Initialize clock schedule.

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).

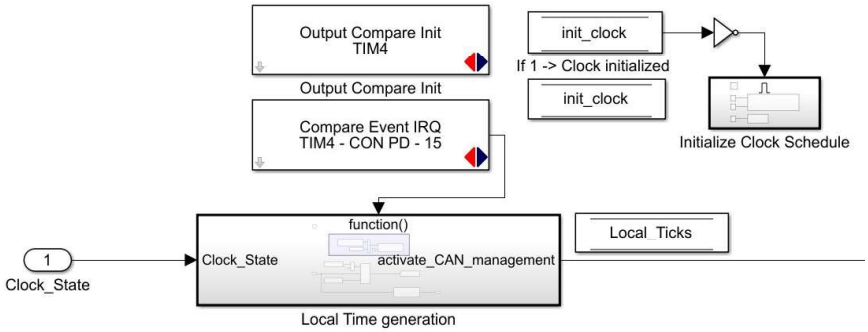


Figure 14: Local time generation group, with the necessary blocks and subsystems to make the local time increase.

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).

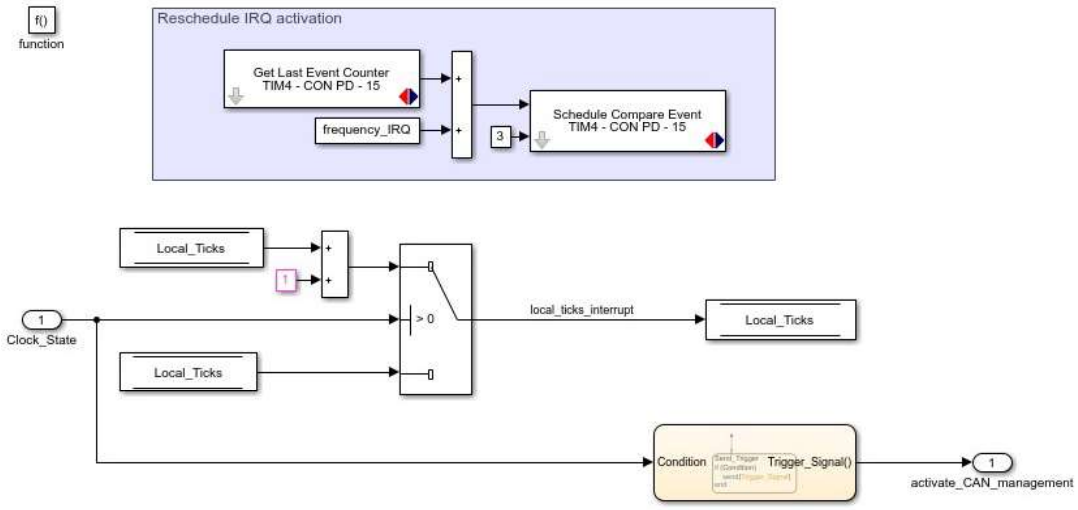


Figure 15: Local time generation subsystem.

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.

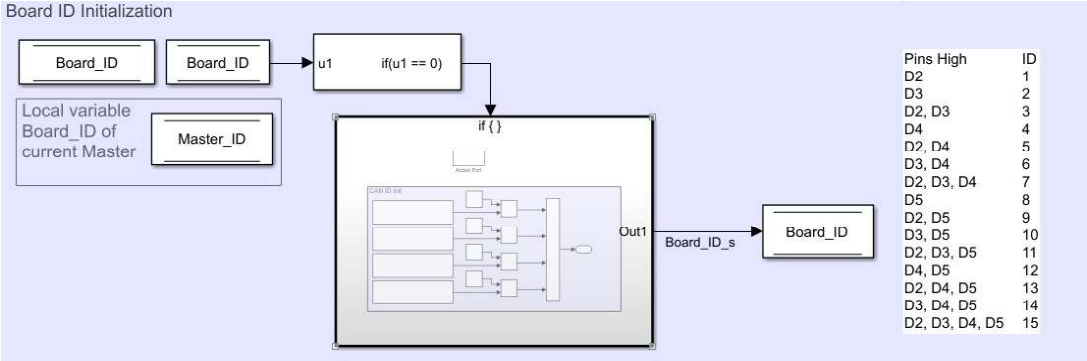


Figure 16: Board ID initialization group. This part of the code is run only once at the beginning of execution.

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).

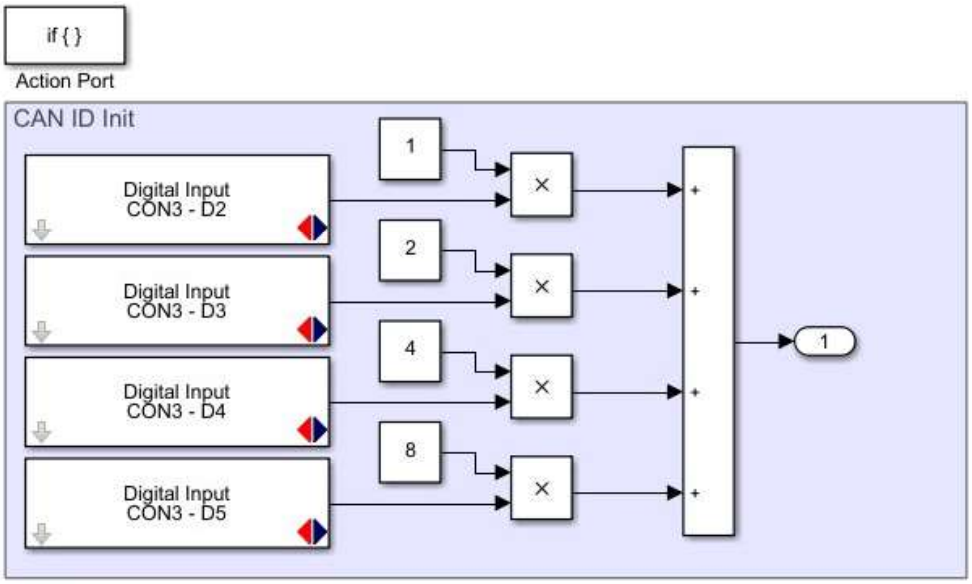


Figure 17: Board ID initialization subsystem.

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.

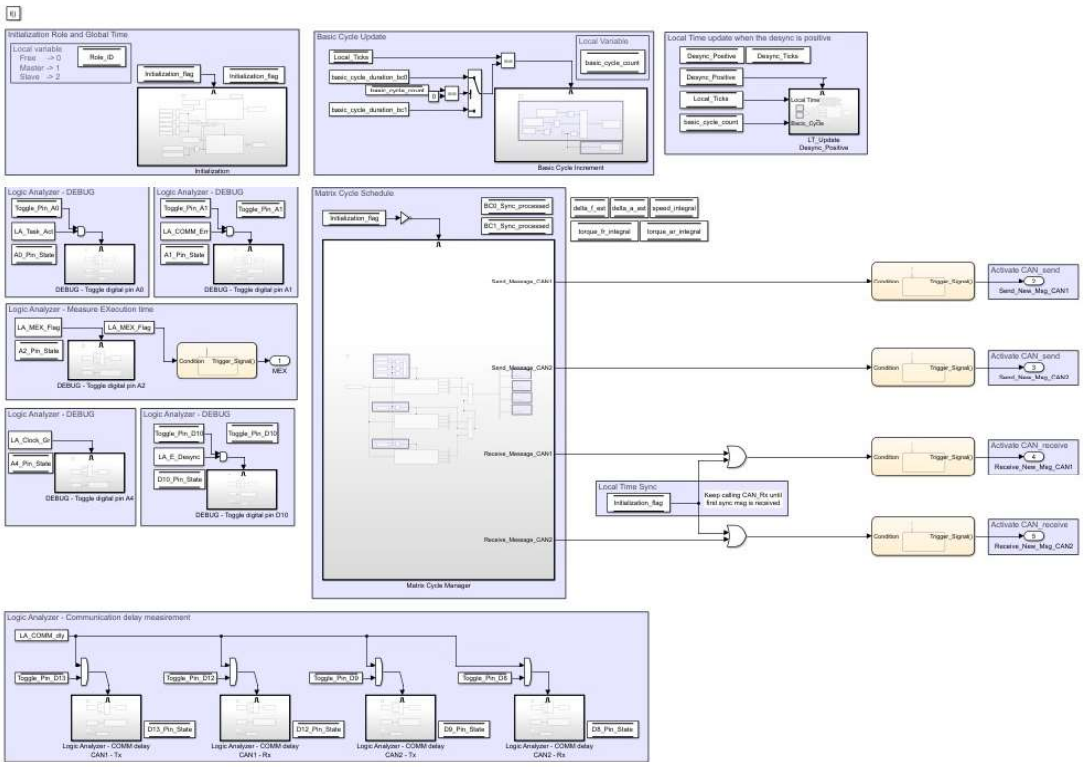


Figure 18: TTA system.

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).

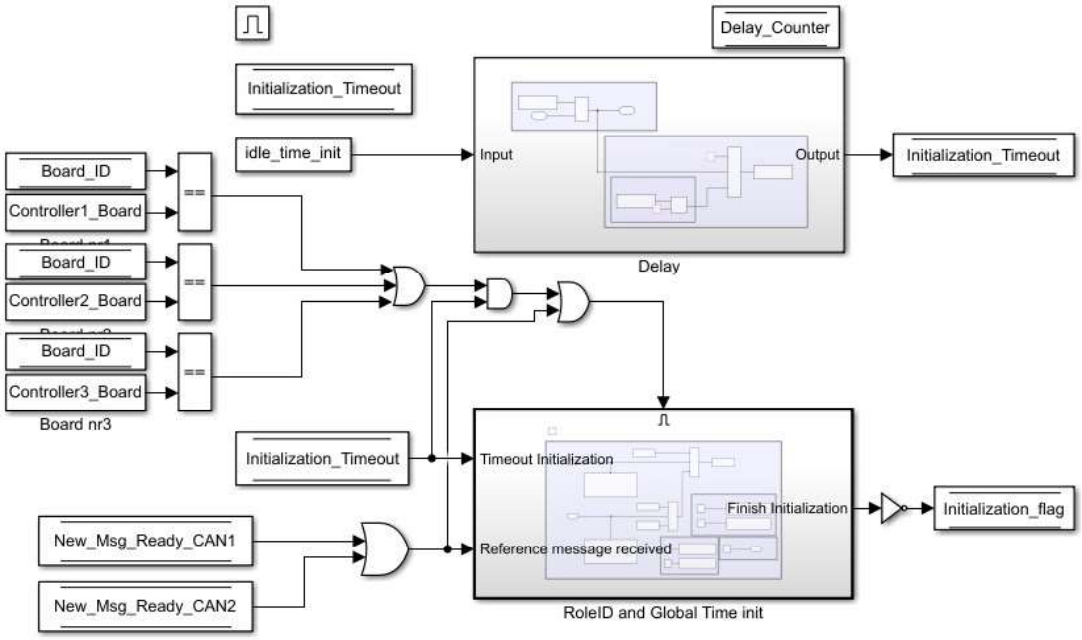


Figure 19: TTA initialization subsystem.

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.

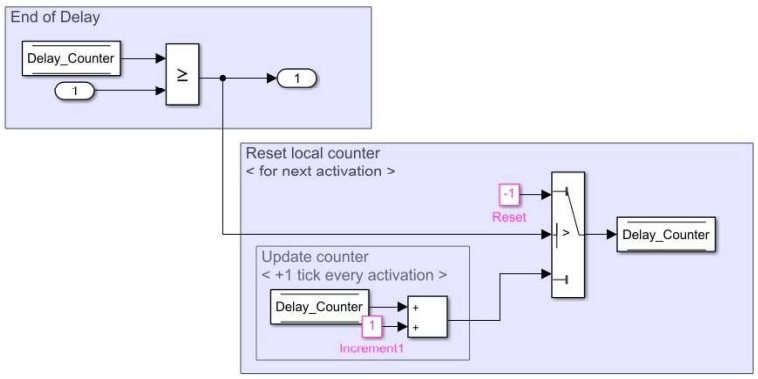


Figure 20: Delay subsystem in the TTA initialization.

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.

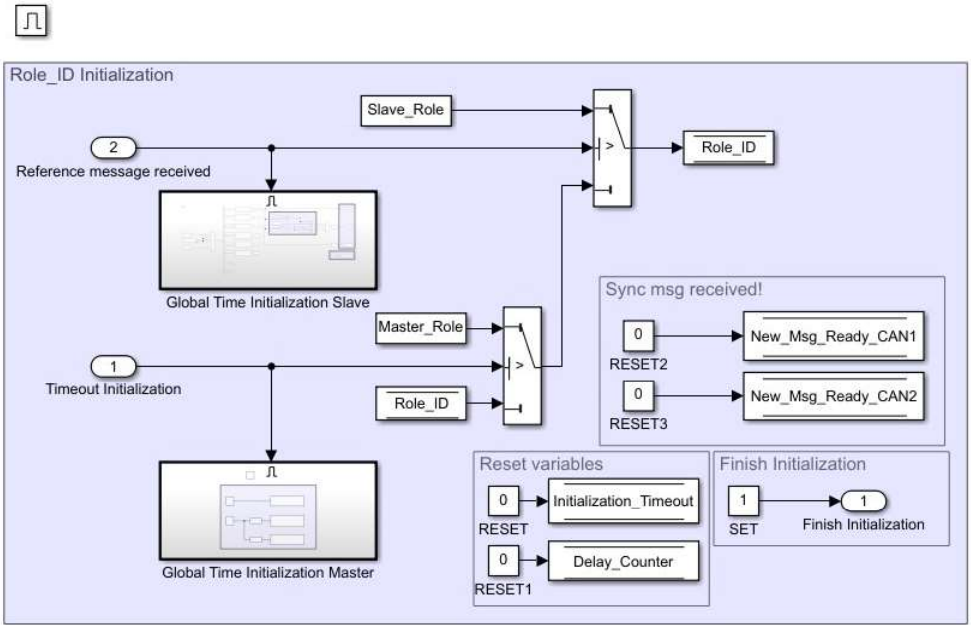


Figure 21: Role and global time initialization in the TTA initialization.

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).

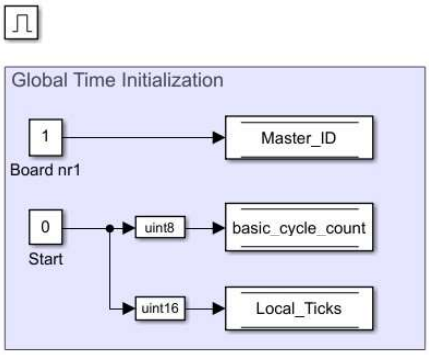


Figure 22: Master role initialization.

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.

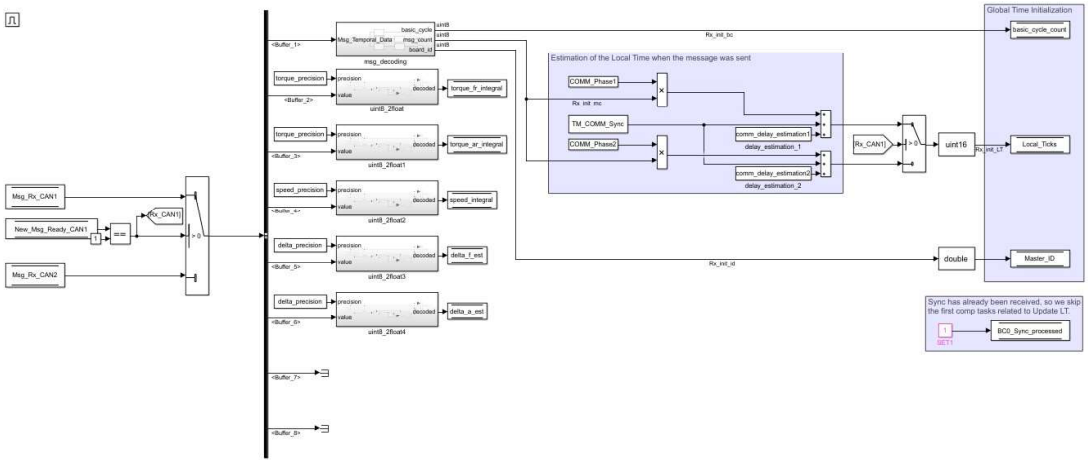


Figure 23: Slave role initialization.

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.

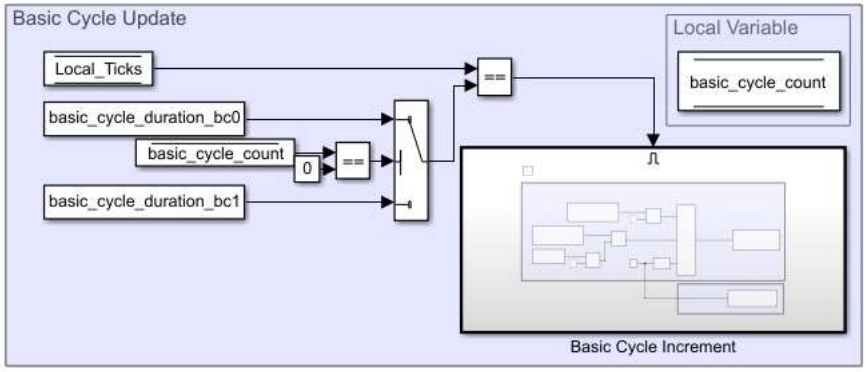


Figure 24: Basic cycle update group.

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.

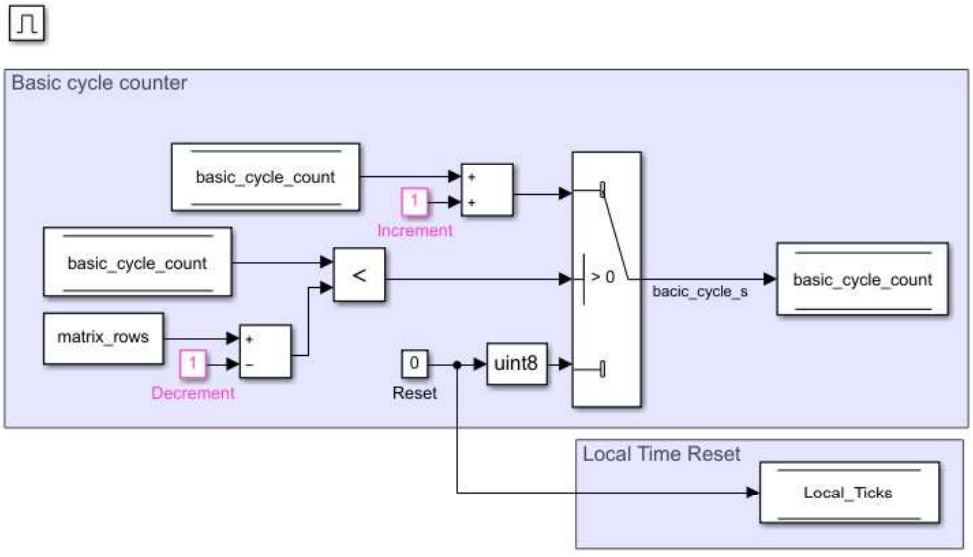


Figure 25: Basic cycle increment system.

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.

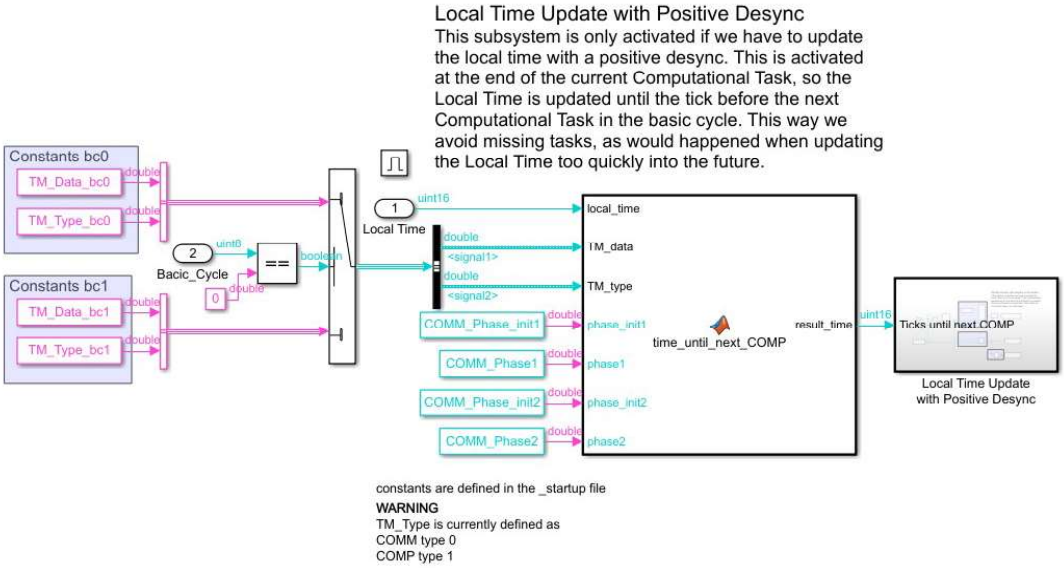


Figure 26: Positive desync system.

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.

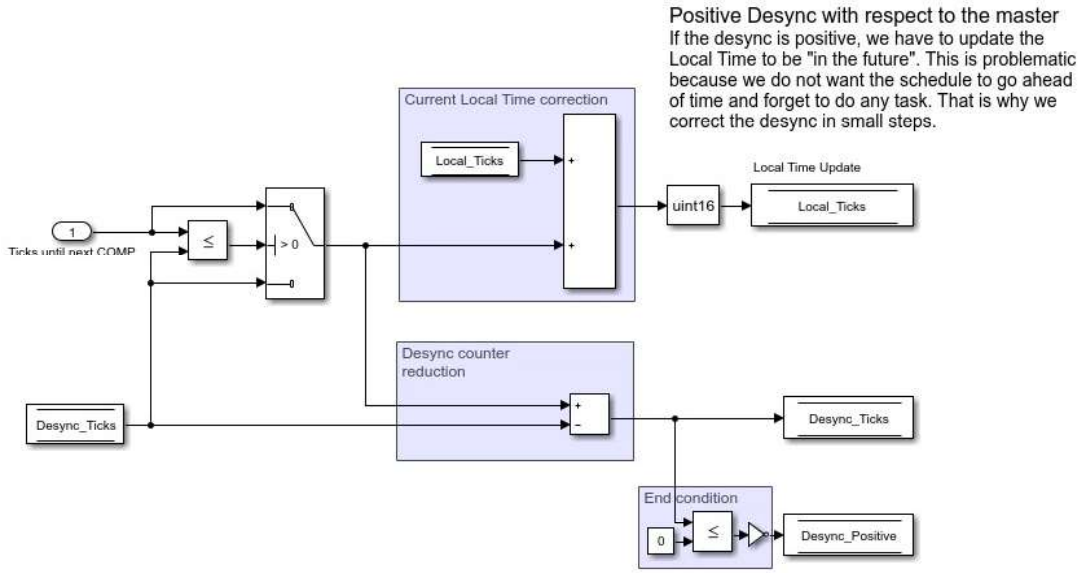


Figure 27: Desync and local time update.

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).

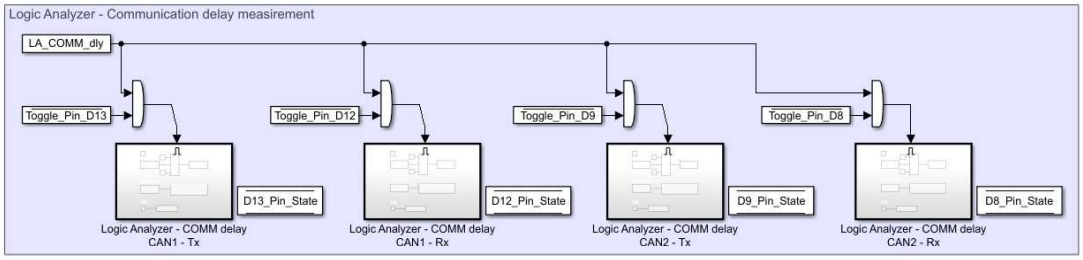


Figure 28: Logic analyzer communication delay subsystems.

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).

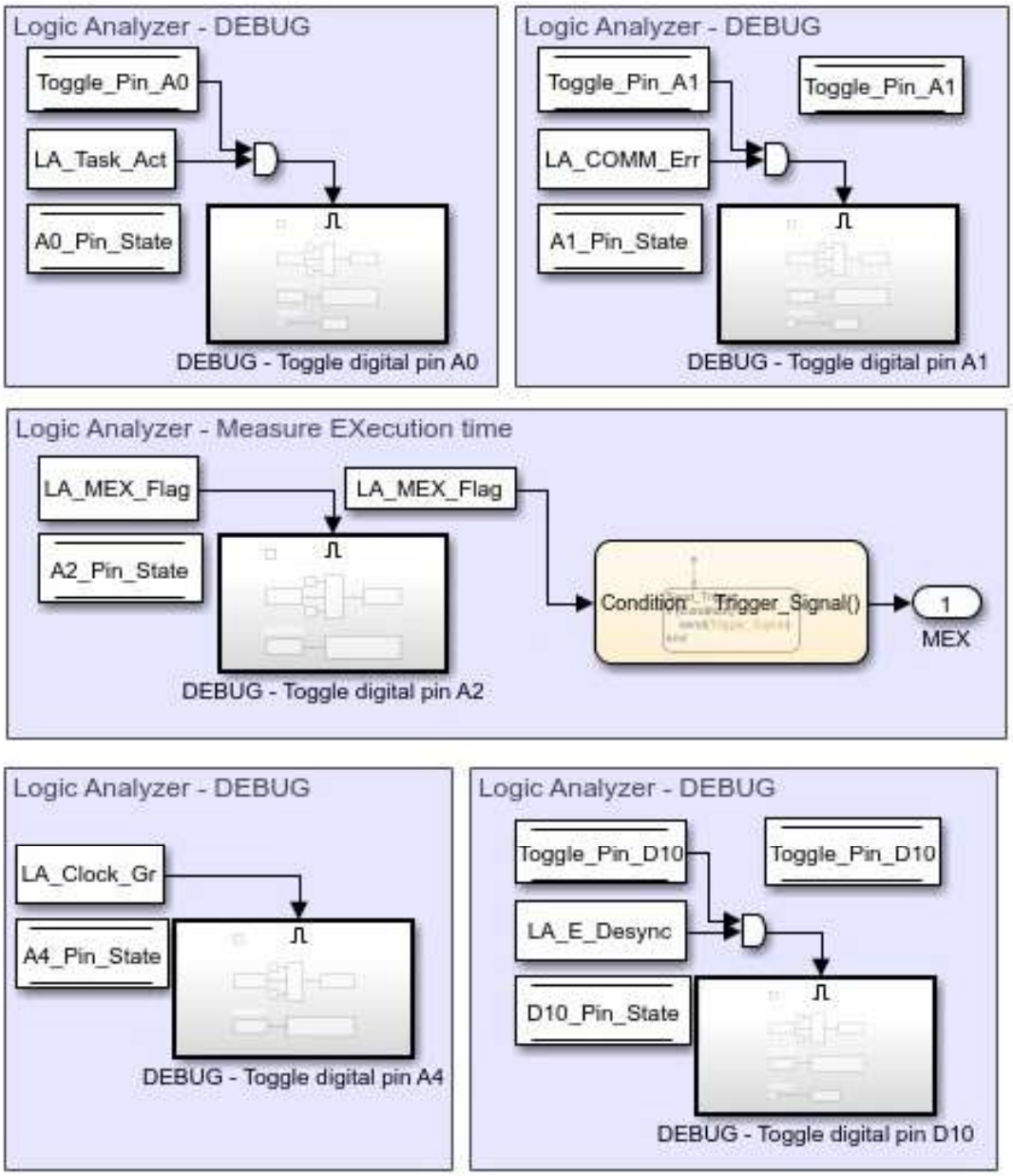


Figure 29: Logic analyzer subsystems.

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.

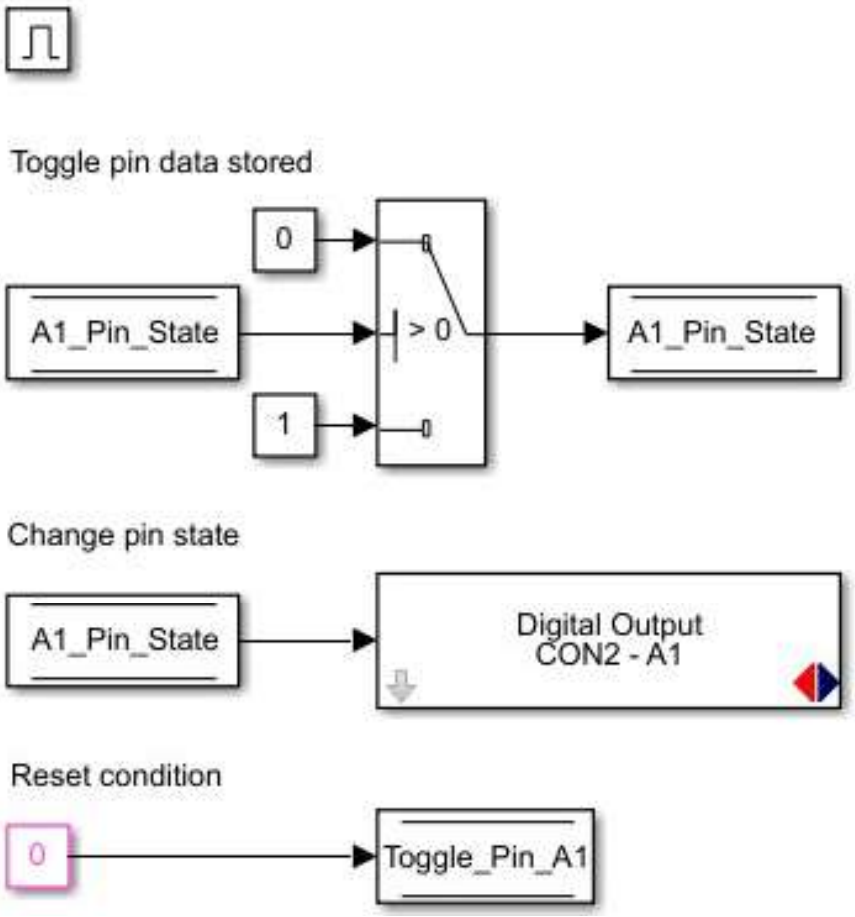


Figure 30: Example of a toggle of a pin state inside a logic analyzer subsystem.

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).

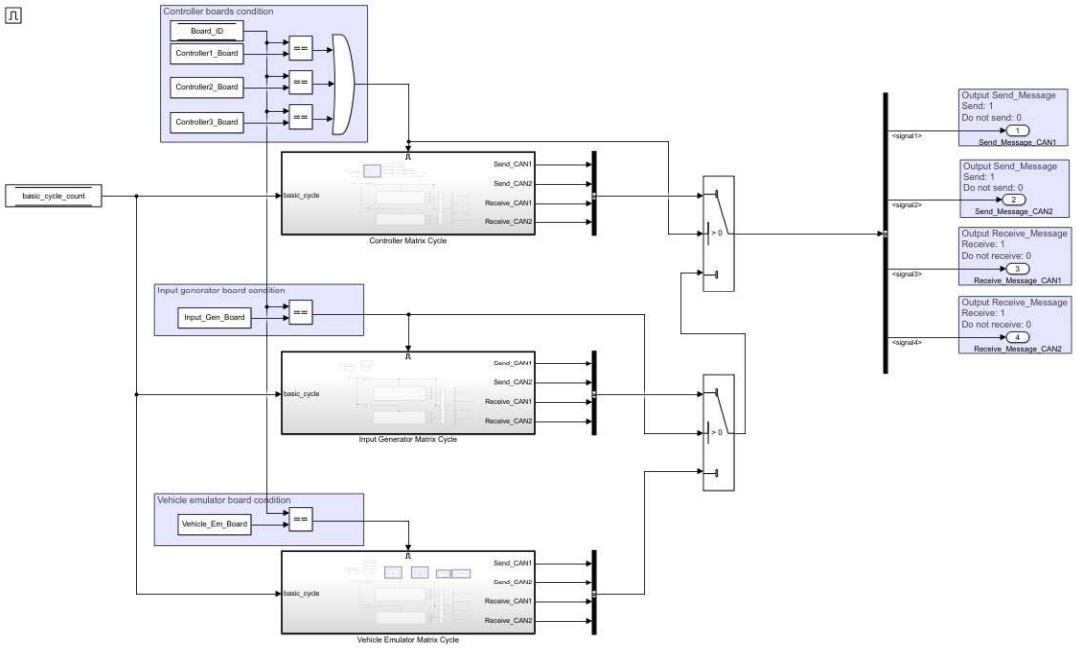


Figure 31: Matrix cycle manager subsystem.

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.

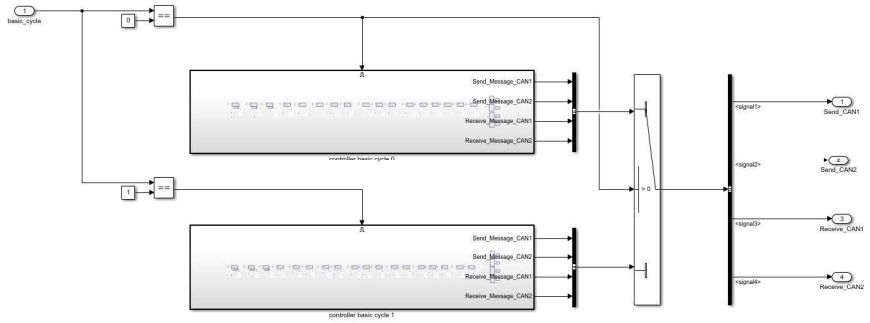


Figure 32: Matrix cycle system.

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.

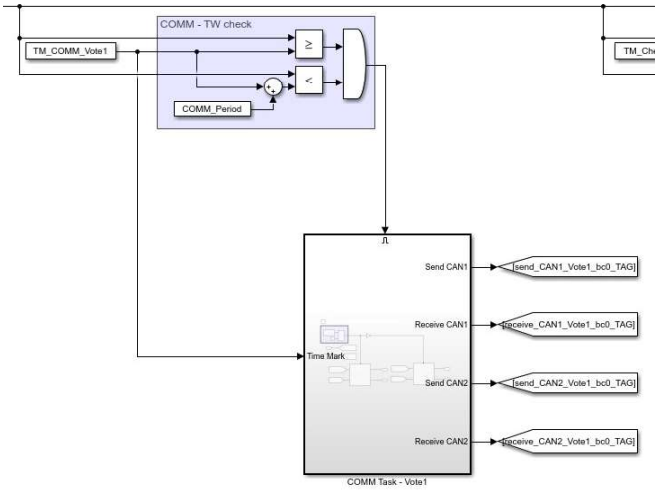


Figure 33: Time window example inside a basic cycle.

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.

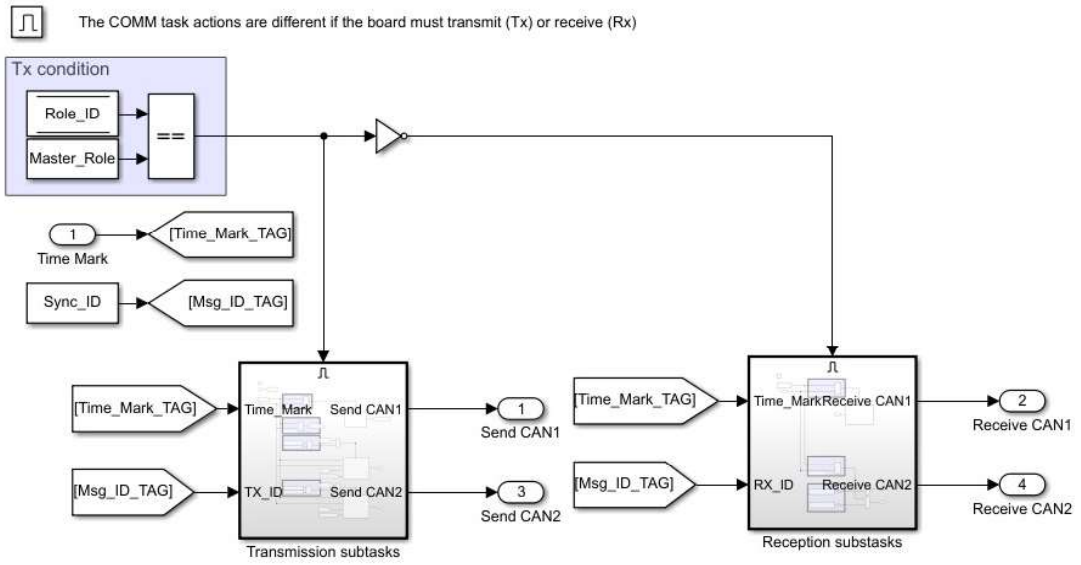


Figure 34: Communication system.

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.

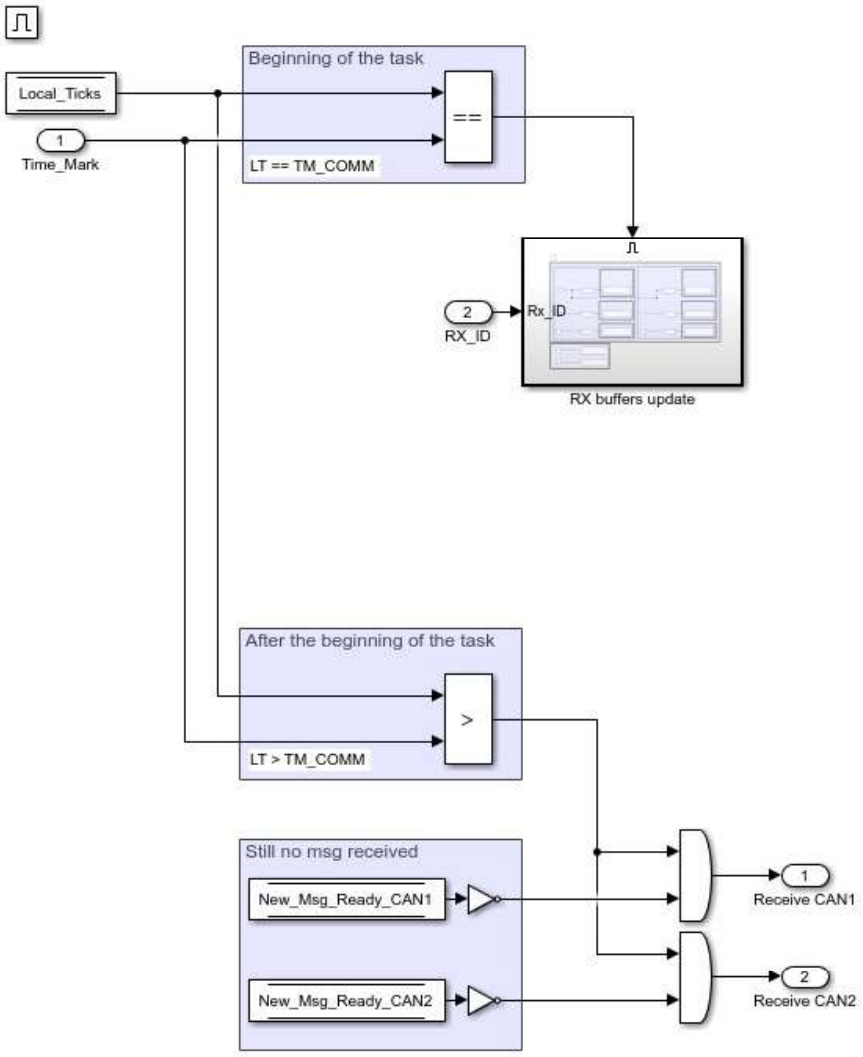


Figure 35: Reception subsystem in the communication tasks.

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.

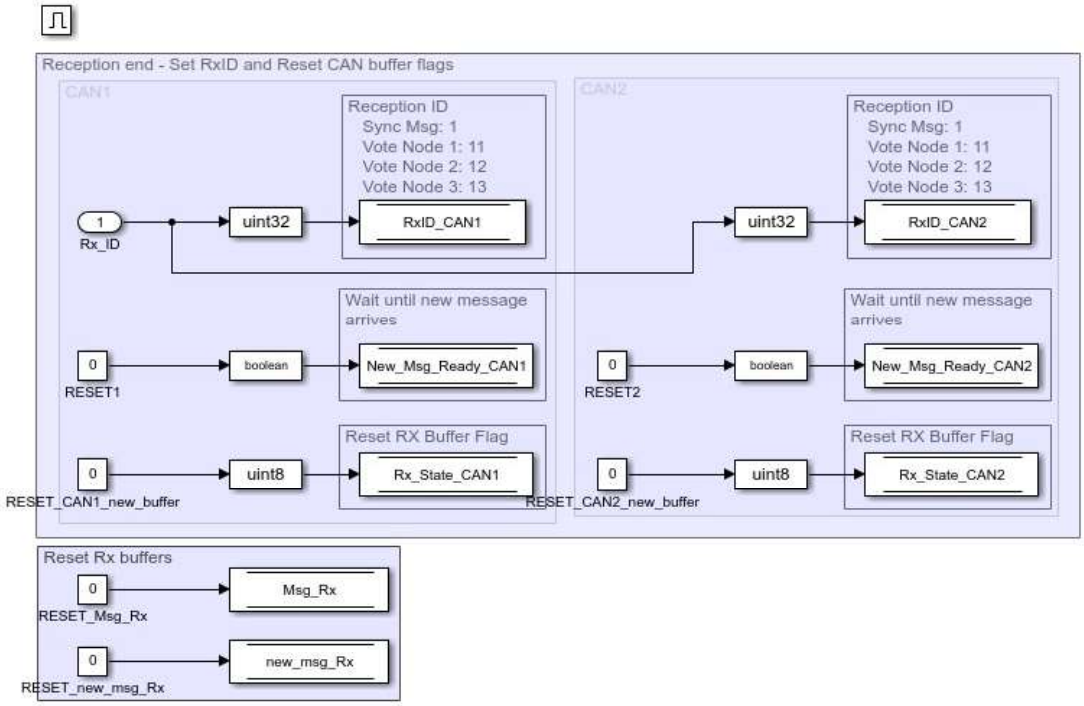


Figure 36: Receive buffers reset subsystem.

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).

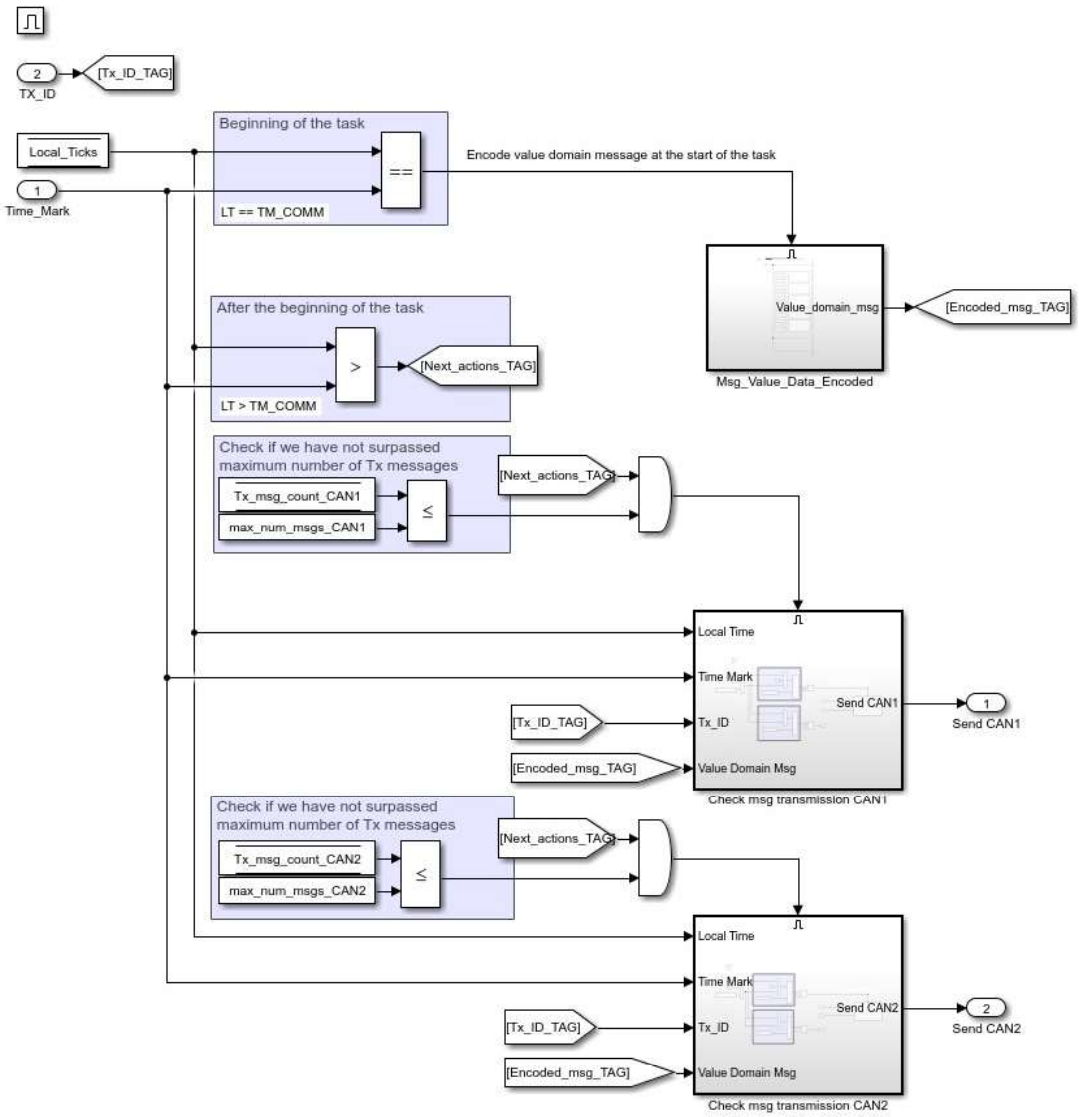


Figure 37: Communication transmission subtask.

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).

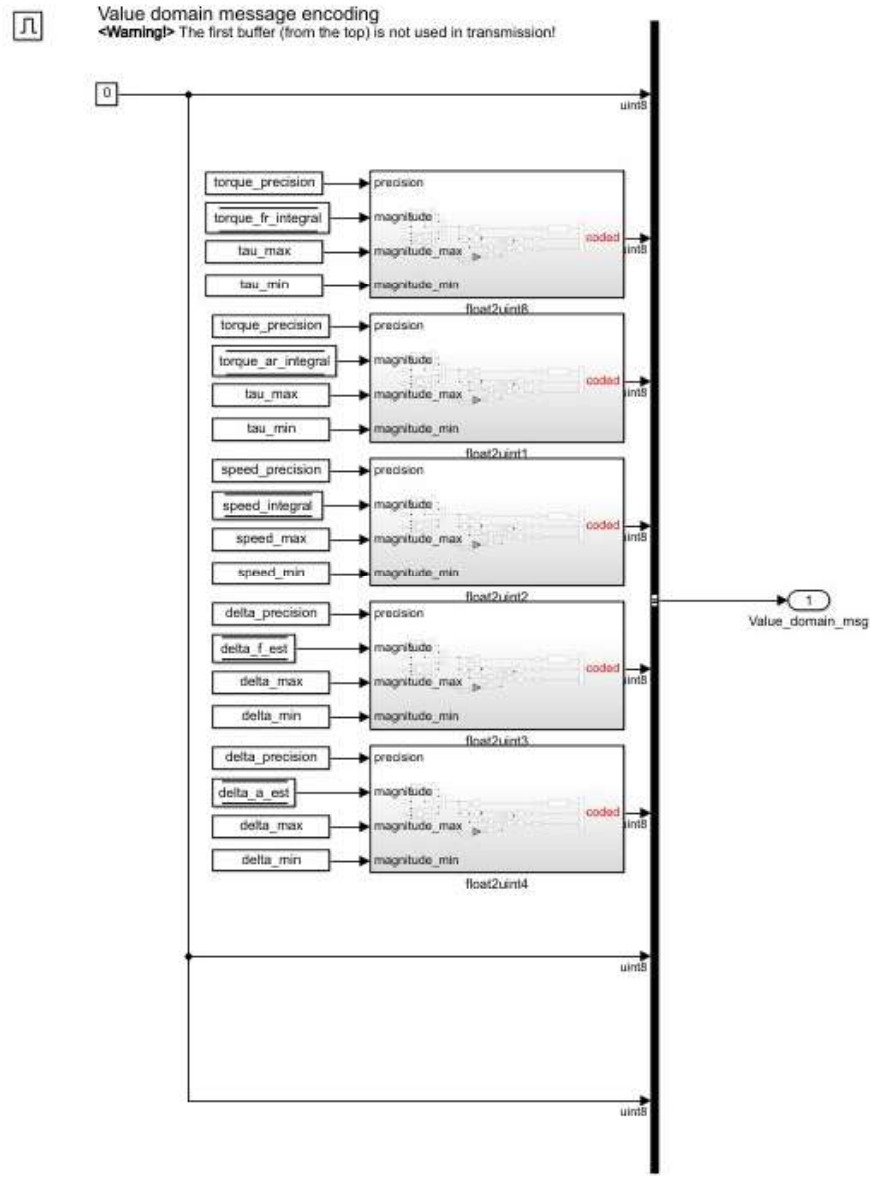


Figure 38: Example of message encoded for a transmission.

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.

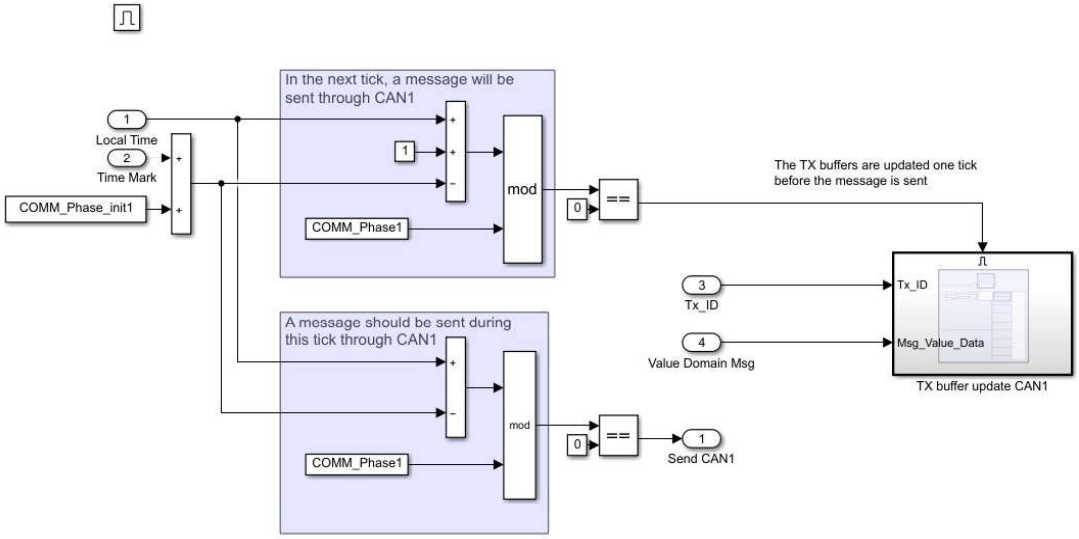


Figure 39: Check transmission in communication task subsystem.

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).

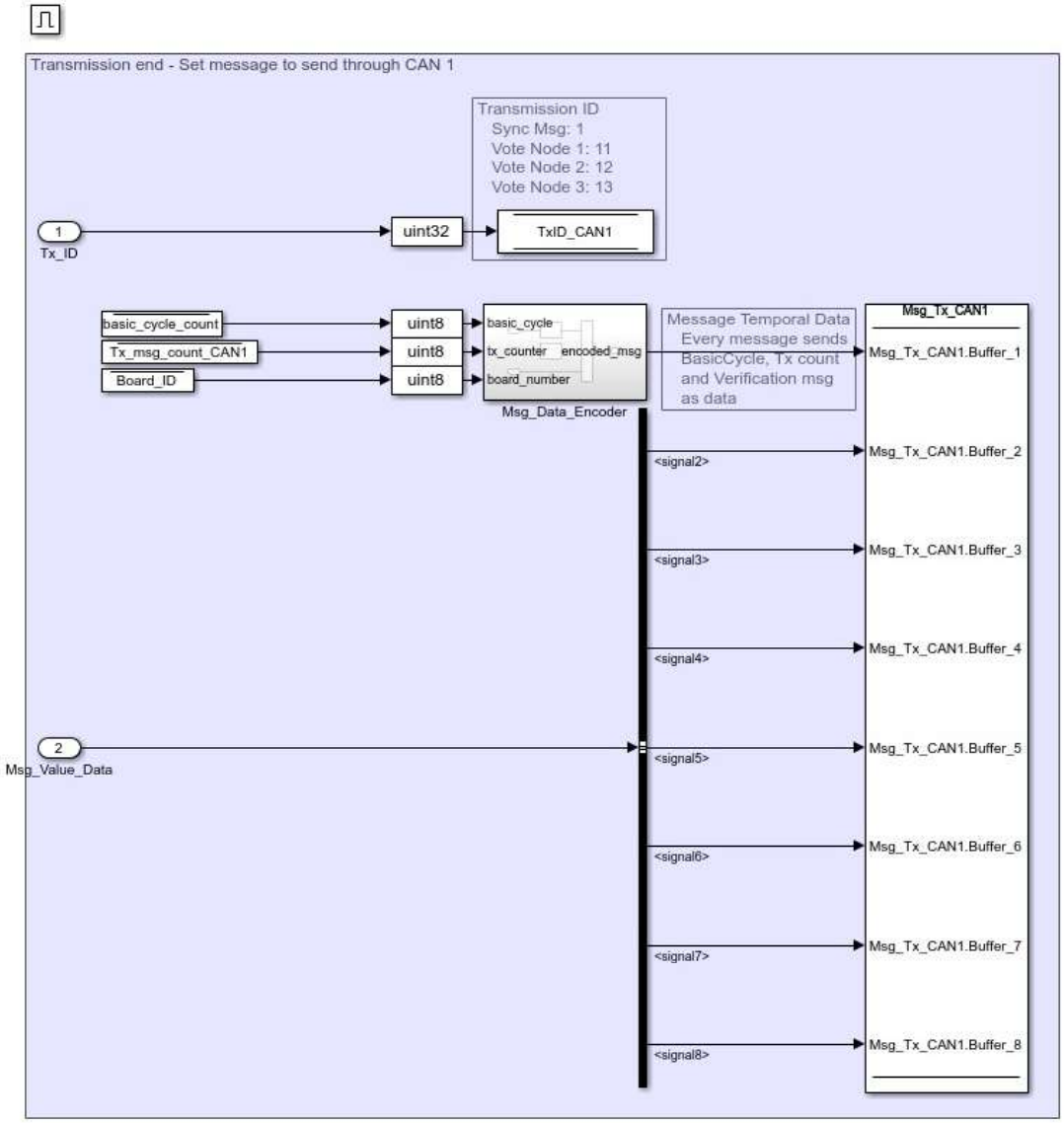


Figure 40: Update transmission buffers subsystem.

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.

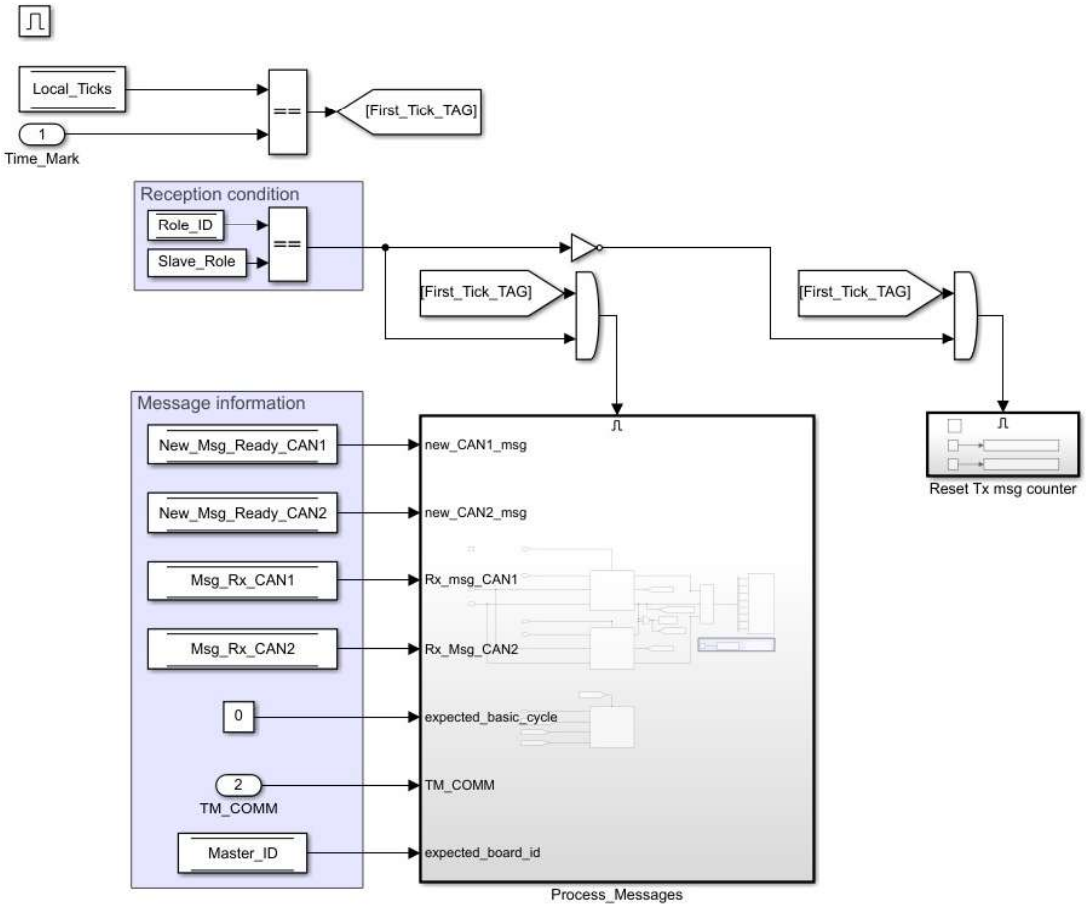


Figure 41: Communication check computational task system.

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.

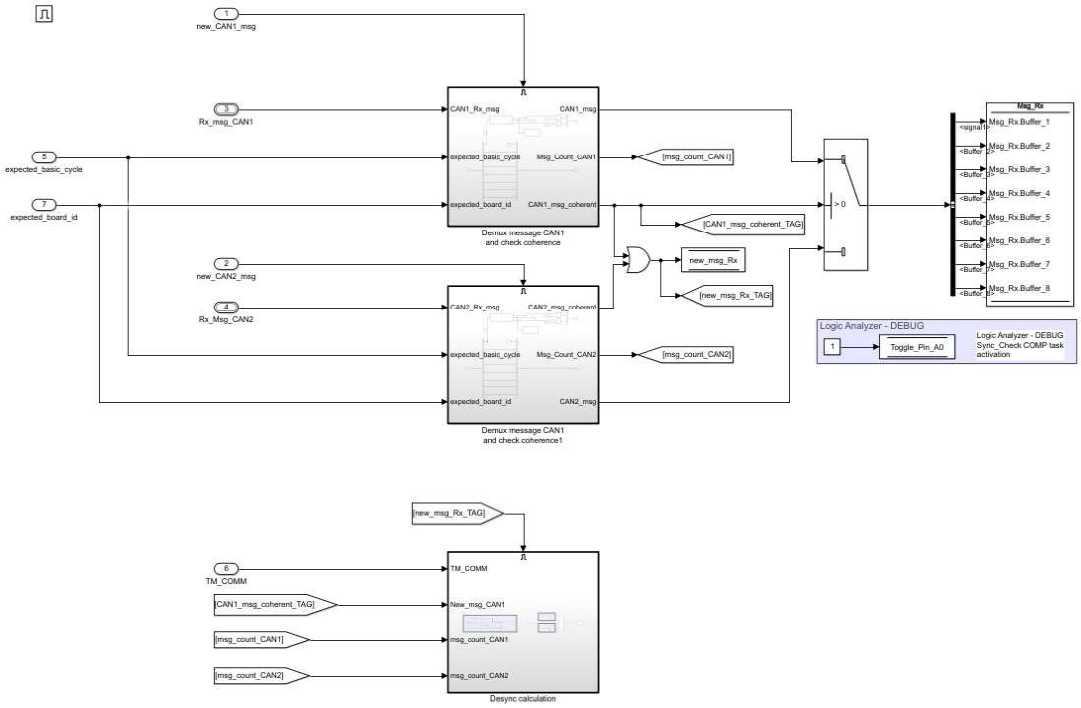


Figure 42: Process message system inside the communication check.

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).

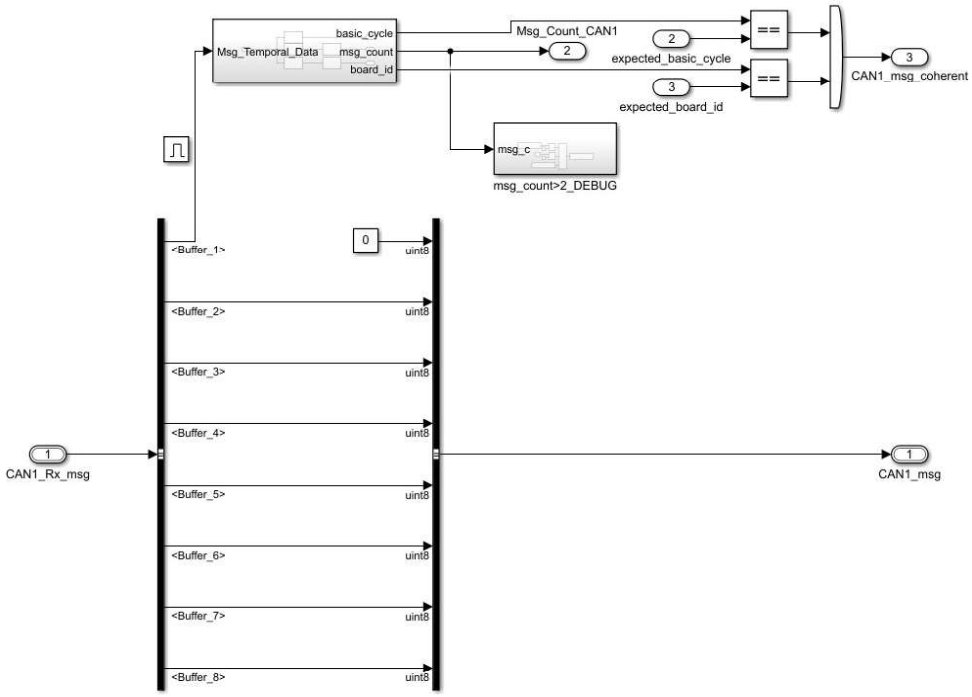


Figure 43: Demux coherence subsystem.

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.

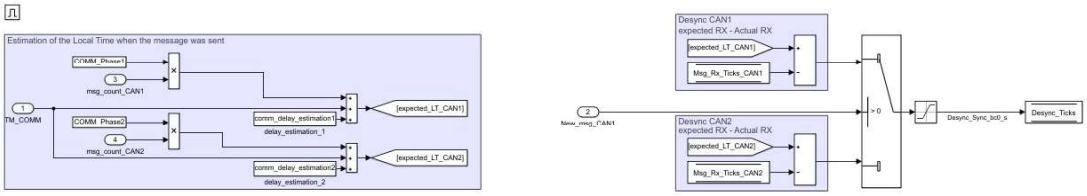


Figure 44: Desync calculation.

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.

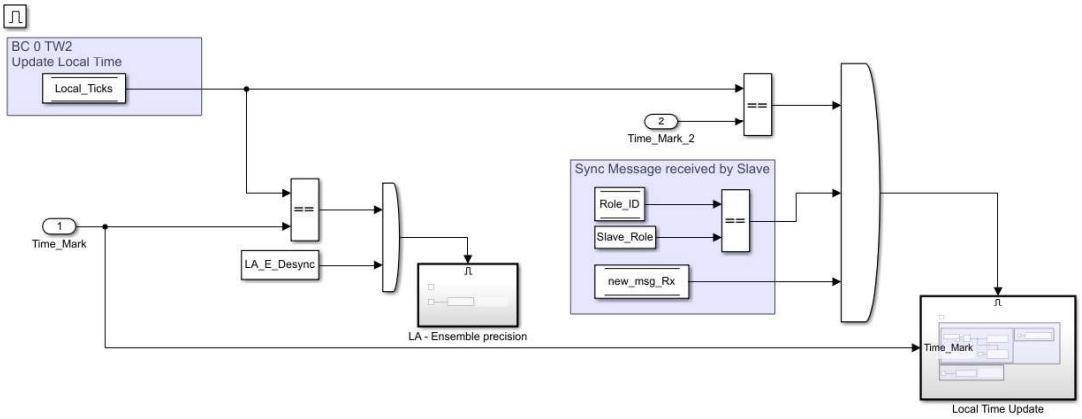


Figure 45: Update local time system.

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.

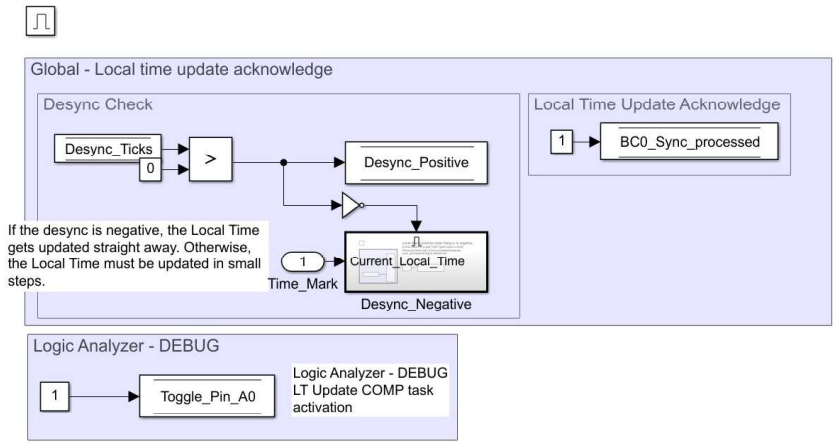


Figure 46: Desync sign check.

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.

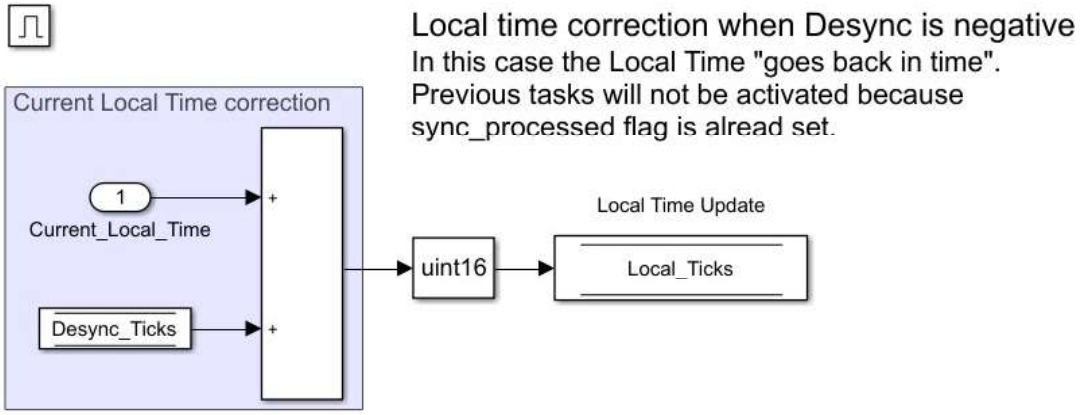


Figure 47: Local time update in the Desync_Negative subsystem.

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

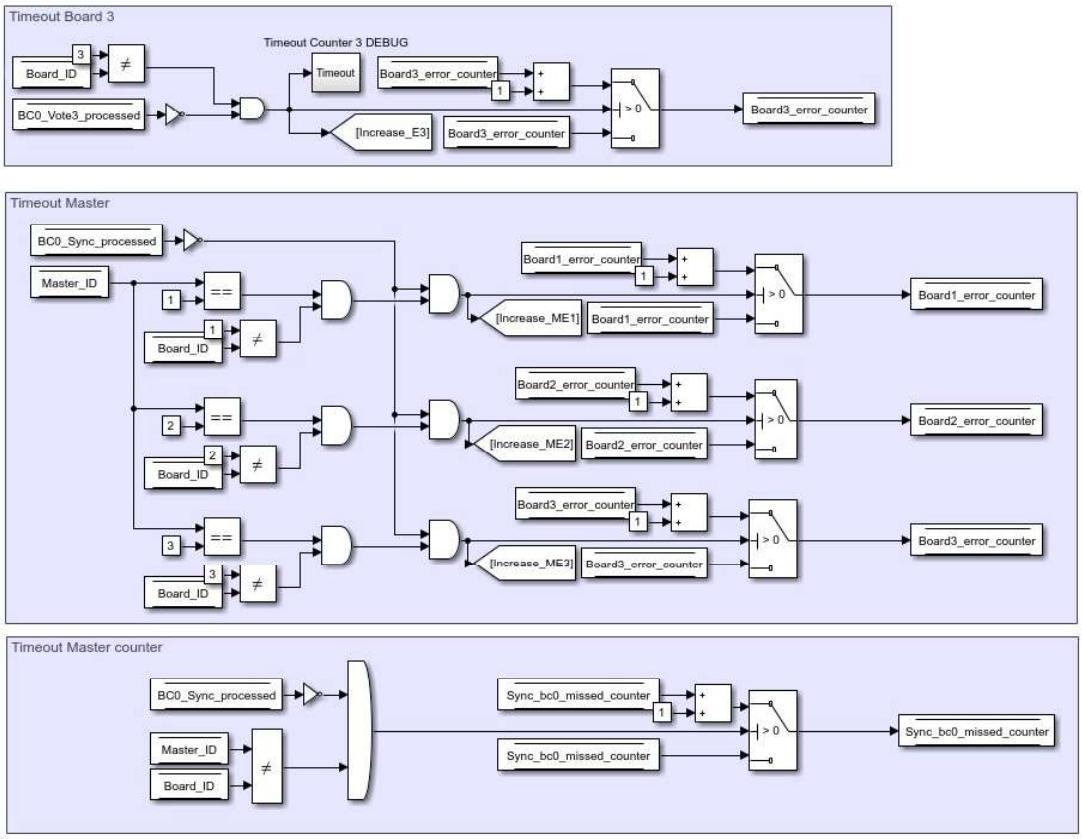


Figure 48: Timeouts processed example.

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.

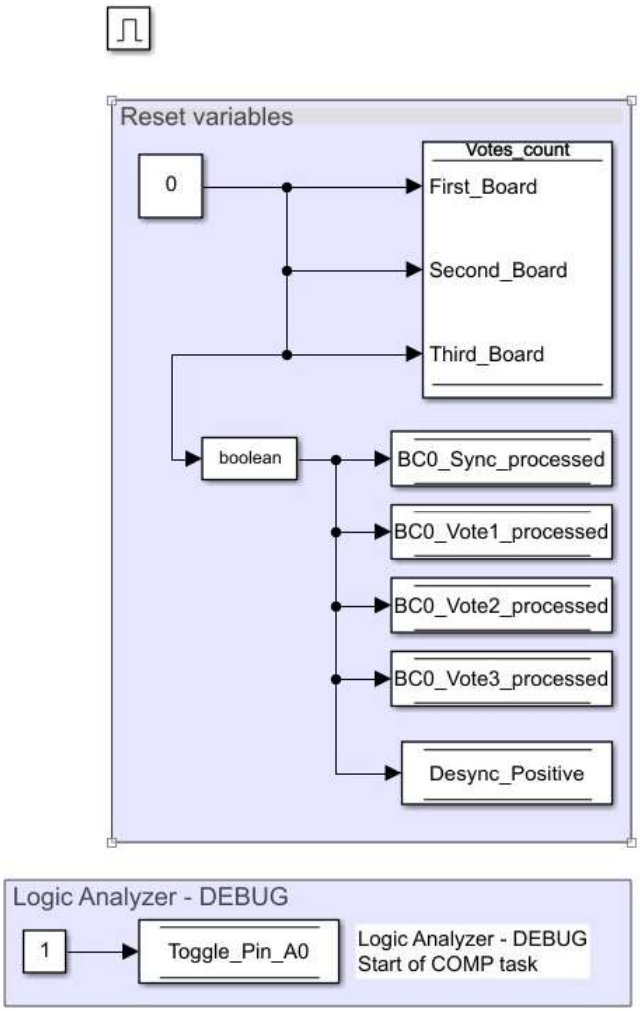


Figure 49: Reset variables system.

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.

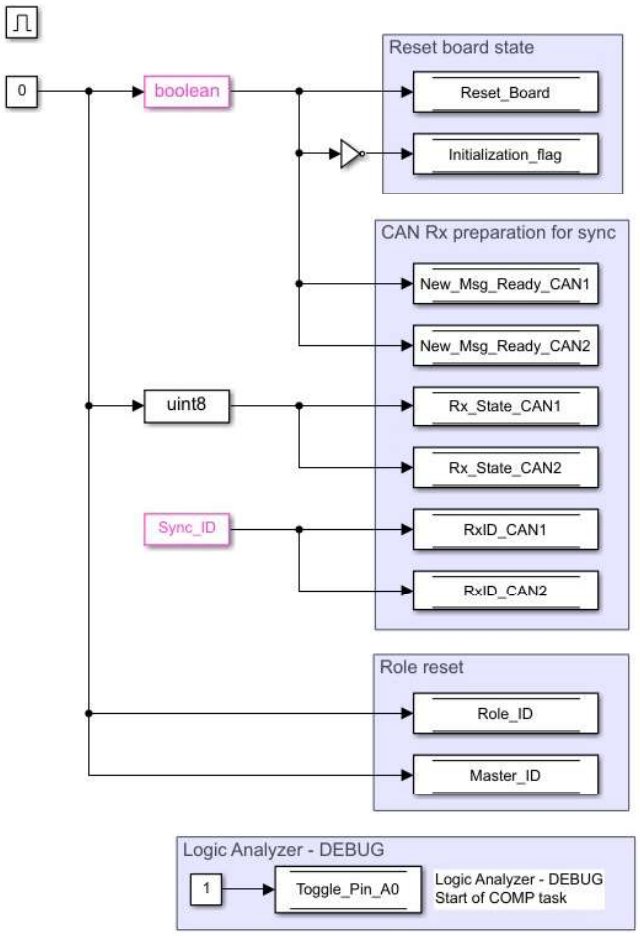


Figure 50: Reset board system.