MASTER

Modeling and implementation of a CAN controller

Driessen, J.G.

Award date:
1995

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
Modeling and implementation of a CAN controller

By Joachim Driessen

Coach : dr. ir. A.C. Verschueren
Supervisor : prof. ir. M.P.J. Stevens
Period : Februari 1995 - December 1995
Abstract

CAN is an abbreviation of Controller Area Network. It is a field bus developed by Bosch for use in cars. In this master thesis report, the modeling of the CAN controller is described. The method that was used for creating this model was the Ward and Mellor method. After having created a model, the CAN controller was implemented in IDaSS.

The model of the CAN controller is completed. Some state transition diagrams are very large. These should be divided into smaller parts in order to make the model even more clear.

The implementation of the CAN controller will be mapped on the SDLC controller that can be found in the IDaSS library. This controller is composed of a bitprocessor and a byte processor. The byte processor can be parametrized. Part of the can protocol will be programmed on the byte processor. A separate filtering unit is available for the filtering that is needed.

The main part of the CAN protocol will be mapped on the bitprocessor. For this purpose I chose to redesign the bitprocessor. The new bitprocessor is composed of eight different units:

- PLS (Physical Layer Signaling)
- Transmit
- Receive
- Bitstuff
- BitDestuff
- CRC
- Management
- BackEnd

The PLS unit takes care of the bit representation and synchronization. This unit is working properly with the exception of the resynchronization.

The Transmit unit is responsible for the transmission of frames. As far as I can judge at this moment, it is working according to specification. The receive unit, the Bitstuff unit, the BitDestuff unit, and the CRC unit are also working according to specification.

The Management unit is composed of three sub units: the Compare unit, the Control unit, and the fault confinement entity (FCE). The compare unit works according to specifications. The Control unit is almost complete. Something for the interface with the BackEnd unit has to be added. Also the handling of the suspend transmission state has to be adjusted. The FCE unit needs two adjustments. The first adjustment is that the error frame handling for an error passive receiver has to be changed. The second adjustment is that the error counters have to be implemented.
The BackEnd functionality has to be defined completely. This is a minor problem though.

In general, the complete bitprocessor has to be checked exhaustively.
Contents

1 Introduction ........................................................................................................................................................................ 1

2 The Controller Area Network ................................................................................................................................................ 3
   2.1 The OSI layer architecture of CAN .......................................................................................................................... 4
   2.2 The LLC sublayer .................................................................................................................................................... 4
   2.3 The MAC sublayer .................................................................................................................................................. 5
   2.4 The physical layer .................................................................................................................................................. 13
   2.5 The fault confinement entity .................................................................................................................................. 17

3 A brief introduction to the Ward and Mellor modeling method ............................................................................................... 21
   3.1 Description of the Ward and Mellor tools .................................................................................................................. 21
      3.1.1 Terminators .................................................................................................................................................... 22
      3.1.2 Data transformations .................................................................................................................................... 22
      3.1.3 Control transformations ................................................................................................................................ 22
      3.1.4 Discrete flows .................................................................................................................................................. 22
      3.1.5 Continuous flows ............................................................................................................................................ 22
      3.1.6 Event flows .................................................................................................................................................... 22
      3.1.7 Data stores ..................................................................................................................................................... 23
      3.1.8 Control stores .................................................................................................................................................. 23
      3.1.9 Ward and Mellor state transition diagrams .................................................................................................... 23
   3.2 An example: sample and hold ....................................................................................................................................... 23
   3.3 Deviations from the Ward and Mellor method .............................................................................................................. 25

4 Presentation of the model of CAN .............................................................................................................................................. 27
   4.1 First iteration ............................................................................................................................................................ 27
   4.2 Second iteration .......................................................................................................................................................... 28
      4.2.1 The context diagram ........................................................................................................................................... 28
      4.2.2 The CAN\{PMA,MDI\} transformation ............................................................................................................. 29
      4.2.3 The LLC transformation .................................................................................................................................. 31
      4.2.4 The MAC transformation .................................................................................................................................. 31
   4.3 Third iteration ............................................................................................................................................................ 32
      4.3.1 The context diagram ........................................................................................................................................... 33
      4.3.2 The CAN transformation .................................................................................................................................. 34
      4.3.3 The LLC transformation .................................................................................................................................. 35
      4.3.4 The MAC transformation .................................................................................................................................. 40
      4.3.5 The PLS transformation .................................................................................................................................. 60

5 Global hardware mapping ......................................................................................................................................................... 63
   5.1 The bitprocessor ......................................................................................................................................................... 63
   5.2 The byteprocessor ....................................................................................................................................................... 64

6 An implementation of the bitprocessor .................................................................................................................................. 65
   6.1 The toplevel schematic ................................................................................................................................................. 65
   6.2 Composition of the bitprocessor .................................................................................................................................. 66
      6.2.1 The PLS schematic .............................................................................................................................................. 67
      6.2.2 The Transmit schematic (Tx) ............................................................................................................................. 68
      6.2.3 The Receive schematic (Rx) ............................................................................................................................... 69
      6.2.4 The bit stuffing schematic .................................................................................................................................. 71
      6.2.5 The bit de stuffing schematic ........................................................................................................................... 72
1 Introduction

In an association between several European companies, initiated by the European Union, a field bus standard is being developed. It is intended to be used in combination with equipment for disabled people. Mainly the use of it in a wheelchair was considered. This field bus carries the name M3S (Multiple Master, Multiple Slave). It is build on top of the CAN (Controller Area Network) bus. The CAN bus is developed by Bosch for use in cars. The purpose was to reduce the amount of cables in the car. The choice to take CAN as a basis for M3S is made because CAN already has extensive safety features. For applications involving equipment for disabled people, extra safety measures have to be taken. M3S introduces these extra safety measures, so the safety of the system is not only depending on the safety of the CAN bus.

For this master thesis, I am going to design an implementation in IDaSS of the CAN part of M3S. A description of the CAN bus is given in chapter 2. Before I can start with the implementation, a dissection of the communication protocol is in order. Therefore, I will first make a model of the CAN bus. The modeling method used is the Ward and Mellor method. A brief description on the Ward and Mellor modeling method is given in chapter 3, and the model is presented in chapter 4. After the model is finished, it is mapped on hardware. For this, the SDLC (Synchronous Data Link Control) controller from the IDaSS library is used. For data involving bit streams, this controller will make use of its bitprocessor. Data involving byte oriented manipulations are carried out by its byteprocessor. The global hardware mapping is described in chapter 5. We will see that the bitprocessor of the SDLC controller has to be changed severely. The IDaSS implementation of the resulting bitprocessor is presented in chapter 6. Chapter 7 gives some conclusions.
2 The Controller Area Network

As was said in the introduction, M3S is built on top of the CAN bus. In this chapter, an overview is given on the functionality of the CAN bus. This information is abstracted from the CAN draft standard [3].

Information on the bus is sent in fixed format frames of different, but limited lengths. When the bus is free, any connected node may start to transmit a new frame. If two or more nodes start to transmit a frame at the same time, the bus access conflict is resolved by contention-based arbitration using the identifier. The transmitter that is sending the frame with the highest priority gains bus access. CAN makes use of recessive and dominant bits. When one node puts a recessive bit on the bus and another node simultaneously puts a dominant bit on the bus, the eventual bus value will be dominant. So during arbitration, when a node puts a recessive bit on the bus and detects a dominant bit, it knows it has lost arbitration and it will immediately refrain from sending other bits.

The frame does not incorporate an address. A node passes a received frame to its user by means of a mechanism called ‘frame acceptance filtering’. This mechanism decides whether the received information is relevant for the node or not. There is no need for receivers to know the transmitter of the information and vice versa. The identifier actually describes the information of the frame. If the information is needed by a receiving node, the node accepts the frame and passes it to its user. Within the CAN network it is guaranteed that a frame is simultaneously accepted either by all nodes or by no node at all.

A transmitter can either send a data frame (if it wins the arbitration), or it can request a frame from another node. The latter is called a remote data request.

There are methods provided for detecting errors. These are: Monitoring (transmitters compare the bit levels they transmitted with the bit levels detected on the bus), 15 bit cyclic redundancy check, bit stuff errors, acknowledgement errors, and frame format check. All receivers that are not bus off (see below) check the consistency of the received frame and will acknowledge consistent frames. Corrupted frames are flagged by any transmitting node and normally operating receiving node. These frames are aborted and will be retransmitted according to the implemented recovery procedure, when the bus becomes idle again. They do participate in the arbitration process in order to gain bus access again. CAN nodes distinguish short disturbances from permanent failures. Defective transmitting nodes are switched off, meaning that the node is logically disconnected from the bus. When we look at a CAN node, it can be in one of three states: error active, error passive, and bus off. An error active node can normally take part in bus communication. It can send an active error flag when it detects an error. An error passive node must not send an active error flag. It takes part in the bus communication, but after transmitting a frame, the node will wait some additional time before initiating another transmission. This is called suspend transmission (see below). When a node is in the bus off state, it can not send or receive frames. It has to wait a long time before it may leave the bus off state again. We will soon see how this exactly works.
2.1 The OSI layer architecture of CAN

The OSI reference model consists of seven layers (see [1]):

1. The Physical Layer
2. The Data Link Layer
3. The Network Layer
4. The Transport Layer
5. The Session Layer
6. The Presentation Layer
7. The Application Layer

The CAN architecture represents two of these layers: The Physical Layer, and the Data Link Layer. In figure 1 we can find these two layers. We can see that the Physical Layer is divided into three sublayers: The Physical Signaling (PLS), the Physical Medium Attachment (PMA), and the Medium Dependent Interface (MDI). The data link layer is divided into two sublayers: The Logic Link Control (LLC), and the Medium Access Control (MAC).

Figure 1. Layered architecture of CAN

2.2 The LLC sublayer

The LLC sublayer describes the upper part of the OSI data link layer. It is concerned with those protocol issues that are independent of the type of medium access method. The LLC provides two types of connectionless transmission services: Unacknowledged data transfer service, and unacknowledged remote data request service. In appendix A the primitives and messages of this sublayer can be found.

The structure of an LLC data frame is depicted in figure 2. The identifier field is composed of 11 bits. The seven most significant bits may not be all '1'. The DLC field indicates the number of bytes in the data field. This Data Length Code consists of 4 bits. The data field can be zero. The admissible number of data bytes for a data frame ranges from 0 to 8. Values 9 to 15 may not be used. The data field consists of the data to be transferred within a data frame. It can contain from 0 to 8 bytes as specified by the DLC field. The structure of an LLC remote frame is depicted in figure 3.
The LLC has to perform three functions: Frame acceptance filtering, overload notification, and recovery management. A frame transaction initiated at the LLC sublayer is a single, self-contained operation, independent of previous frame transactions. The content of the frame is named by its identifier. The identifier does not indicate the destination of the frame but describes the meaning of the data. Each receiver decides by a frame acceptance filtering whether the frame is relevant for it or not. The transmission of an overload frame will be initiated by the LLC sublayer if internal conditions of a receiver require delay of the next LLC data or LLC remote frame. To delay the next data or remote frame, at most two overload frames may be generated. The LLC sublayer provides means for automatic retransmission of frames that have lost arbitration or that have been disturbed by errors during transmission. The frame transmission service will not be confirmed to the user before the transmission is successfully completed.

2.3 The MAC sublayer

The MAC sublayer represents the lower part of the OSI data link layer. It services the interface to the LLC sublayer and to the Physical Layer. Furthermore it comprises the functions and rules that are related to encapsulation and decapsulation of the transmit data or receive data, the error detection and signaling, and the management of the transmit or receive medium access. The MAC sublayer provides services to the local LLC for acknowledged transfer of LLC frames and for transmission of overload frames.

The acknowledged data transfer service provides a means by which LLC entities can exchange information without the establishment of a data link connection. The data transfer can be point-to-point, multicast, or broadcast. The acknowledged remote data request service provides means by which an LLC entity can request another node to transmit information without the establishment of a data link connection. The remote LLC entity uses the MAC service 'acknowledged data transfer' for the transmission of the requested data. In both cases, acknowledgement of a service is generated by the MAC sublayer(s) of the remote node(s). Acknowledgement does not contain any data of the remote node's user. The overload frame transfer service provides means by which an LLC entity can initiate the transmission of an overload frame. This causes to delay the next data or remote frame. Appendix A gives the primitives and messages of this sublayer.
The functionality of the MAC sublayer is divided into two fully independently operating parts: The transmitter part and the receiver part. In table I the functionality of the transmit-part of the MAC sublayer is given. The functionality of the receive part can be found in table II.

<table>
<thead>
<tr>
<th>Frame transmission</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Transmit Data Encapsulation</strong></td>
<td></td>
</tr>
<tr>
<td>Acceptance of LLC frames and interface control information</td>
<td></td>
</tr>
<tr>
<td>CRC sequence calculation</td>
<td></td>
</tr>
<tr>
<td>Construction of MAC frame by adding SOF, RTR bit, reserved bits, CRC, ACK, and EOF to the LLC frame</td>
<td></td>
</tr>
<tr>
<td><strong>Transmit Media Access Management</strong></td>
<td></td>
</tr>
<tr>
<td>Initialization of the transmission process after recognizing bus idle (compliant with interframe space)</td>
<td></td>
</tr>
<tr>
<td>Serialization of the MAC frame</td>
<td></td>
</tr>
<tr>
<td>Insertion of stuffbits (bit stuffing)</td>
<td></td>
</tr>
<tr>
<td>Arbitration and passing into receive mode in case of loss of arbitration</td>
<td></td>
</tr>
<tr>
<td>Error detection (monitoring, format check)</td>
<td></td>
</tr>
<tr>
<td>Acknowledgement check</td>
<td></td>
</tr>
<tr>
<td>Recognition of an overload condition</td>
<td></td>
</tr>
<tr>
<td>Overload frame construction and initiation of transmission</td>
<td></td>
</tr>
<tr>
<td>Error frame construction and initiation of transmission</td>
<td></td>
</tr>
<tr>
<td>Presentation of a serial bit stream to the physical layer for transmission</td>
<td></td>
</tr>
</tbody>
</table>
Table II. The functionality of the MAC sublayer (Receive part)

<table>
<thead>
<tr>
<th>Frame reception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive Media Access Management</td>
</tr>
<tr>
<td>Reception of a serial bit stream from the physical layer</td>
</tr>
<tr>
<td>Deserialization and recompiling of the frame structure</td>
</tr>
<tr>
<td>Deletion of stuffbits (bit destuffing)</td>
</tr>
<tr>
<td>Error detection (CRC, format check, stuff rule check)</td>
</tr>
<tr>
<td>Transmission of acknowledgement</td>
</tr>
<tr>
<td>Error frame construction and initiation of transmission</td>
</tr>
<tr>
<td>Recognition of an overload condition</td>
</tr>
<tr>
<td>Reactive overload frame construction and initiation of transmission</td>
</tr>
<tr>
<td>Receive Data Decapsulation</td>
</tr>
<tr>
<td>Removing the MAC specific information from the received frame</td>
</tr>
<tr>
<td>Presenting the LLC frame and interface control information to the LLC sublayer</td>
</tr>
</tbody>
</table>

Data transmission and reception between nodes in a CAN system is manifested and controlled by four different frame types: A data frame, a remote frame, an error frame, and an overload frame. The MAC data frame is composed of seven different bit fields, as is depicted in figure 4. The Start of frame (SOF) field marks the beginning of data and remote frames. It consists of a single dominant bit. A node is only allowed to start transmission when the bus is idle. All nodes have to synchronize to the leading edge caused by start of frame of the node starting transmission first. The arbitration field is composed of the identifier field passed from the LLC sublayer, and the RTR (Remote Transmission Request) bit. The value of the RTR bit is dominant in a MAC data frame, and recessive in a MAC remote frame. The control field consists of six bits. It includes two bits reserved for future expansion followed by the Data Length Code. Receivers have to accept dominant as well as recessive bits as reserved bits in all combinations. Until the function of the reserved bits is defined, the transmitter will
only send dominant bits. The data field is equivalent to the LLC data field. The CRC field contains the CRC sequence followed by a CRC delimiter. The polynomial that is used in the CRC calculation is composed of the following fields (of the destuffed bit stream): The start of frame, the arbitration field, the control field, the data field (if present), and for the lowest coefficients by 15 dominant bits. This polynomial is divided (modulo 2) by the generator polynomial \((X^{15} + X^{14} + X^{10} + X^8 + X^7 + X^4 + X^3 + 1)\). The remainder of this polynomial division is the CRC sequence transmitted over the bus. The CRC delimiter consists of a single recessive bit. The Acknowledgement field contains the acknowledge slot and the acknowledgement delimiter, both consisting of one bit. The transmitting node sends a recessive bit in the acknowledgement slot. All nodes that have received the matching CRC sequence send a dominant bit in the Acknowledgement slot thus overwriting the recessive bit. The acknowledgement delimiter is the second bit of the acknowledgement field. It must have the value recessive. The end of frame (EOF) is a flag sequence consisting of seven consecutive recessive bits. It is used to delimit each data or remote frame.

![Figure 4. MAC data frame](image)

Figure 4. MAC data frame

Figure 5 reveals the structure of the MAC remote frame. Comparing it with the structure of the MAC data frame, we see that there are two differences. The first difference is the missing data field and the second difference is that the RTR bit that is part of the arbitration field (not visible in figures 4 and 5) is recessive in stead of dominant.

![Figure 5. CAN remote frame](image)

Figure 5. CAN remote frame

In figure 6 we can find the structure of the error frame. It consists of two different fields. The first field is given by the superposition of error flags contributed from different nodes. There are two different kinds of error flags: The active error flag and the passive error flag. The active error flag consists of six consecutive dominant bits. The passive error flag consists of six consecutive recessive bits. Some or all bits of the passive error flag may be overwritten by dominant bits from other nodes. An error active node detecting an error condition signals this by transmission of an error active flag. The error flag's form violates the rule of bit stuffing or destroys the bit field requiring fixed form. As a consequence, all other nodes detect an error condition too and on their part start transmission of an error flag. So the sequence of dominant bits which actually can be monitored on the bus, results from a superposition of different error flags transmitted by individual nodes. The total length of this sequence varies between a minimum of six and a maximum of twelve bits. Passive error flags initiated by a transmitter cause error flags at the receivers when they start in a frame field which is encoded by the method of bit stuffing. The receivers will detect stuff errors in that case. This requires, however, that such an error flag does not start during arbitration and another node continues transmitting, or that it starts very few bits.
before the end of the CRC sequence and the last bits of the CRC sequence happen to be all recessive. Passive error flags, initiated by receivers, are not able to prevail any activity on the bus line. Therefore, error passive receivers always have to wait for six subsequent equal bits after detecting an error condition, until they have completed their error flag. The error delimiter consists of eight recessive bits. After transmission of an error flag, each node sends recessive bits and monitors the bus until it detects a recessive bit. Afterwards it starts transmitting seven more recessive bits.

There are two types of overload frames: LLC-requested overload frames, and reactive overload frames. Both overload frames have the same format. In figure 7, the structure of an overload frame is given. From this figure, we can see that an overload frame consists of two bit fields: The overload flag and the overload delimiter. The overload flag consists of six consecutive dominant bits. It destroys the fixed form of the intermission field (see below). As a consequence, all other nodes also detect an overload condition and on their part start transmission of an overload flag. The overload delimiter consists of eight consecutive recessive bits. After transmission of an overload flag, every node monitors the bus until it detects a recessive bit. At this point of time, every node has finished sending its overload flag, and all nodes start transmission of seven more recessive bits simultaneously, to complete the eight bit long overload delimiter.

There are two types of overload frames: LLC-requested overload frames, and reactive overload frames. Both overload frames have the same format. In figure 7, the structure of an overload frame is given. From this figure, we can see that an overload frame consists of two bit fields: The overload flag and the overload delimiter. The overload flag consists of six consecutive dominant bits. It destroys the fixed form of the intermission field (see below). As a consequence, all other nodes also detect an overload condition and on their part start transmission of an overload flag. The overload delimiter consists of eight consecutive recessive bits. After transmission of an overload flag, every node monitors the bus until it detects a recessive bit. At this point of time, every node has finished sending its overload flag, and all nodes start transmission of seven more recessive bits simultaneously, to complete the eight bit long overload delimiter.

Data frames and remote frames are separated from preceding frames whatever type they are (data frame, remote frame, error frame, overload frame) by a bit field called interframe space. In contrast to this, overload frames and error frames are not preceded by an interframe space as well as multiple overload frames are not separated by an interframe space. Depending on whether a node is error active or error passive, the interframe space has two different forms of appearance. Figure 8 gives the
interframe space for error active nodes, and figure 9 gives the interframe space for error passive nodes. In case of error active nodes, the interframe space consists of two fields: Intermission, and bus idle. The intermission consists of three consecutive recessive bits. During intermission, no node is allowed to start transmission of a data or remote frame. The only action to be taken is signaling an overload condition. The period of bus idle may be of arbitrary length. The bus is recognized to be free and any node can access the bus in order to transmit. A frame which is pending for transmission during the transmission of another frame is started in the first bit following intermission. The detection of a dominant bit on the bus during bus idle is interpreted as start of frame. In case of an error passive node, the bus idle period is preceded by a period called suspend transmission. After an error passive node has transmitted a frame, it sends eight recessive bits following intermission before it is allowed to transmit another frame. When a transmission (caused by another node) starts, the node will become receiver of this frame.

![Figure 8. Interframe space for nodes which are not error passive or have been receiver of the previous frame](image)

![Figure 9. Interframe space for error passive nodes which have been transmitter of the previous frame](image)

The frame segments start of frame, arbitration field, control field, data field, and CRC sequence are coded by the method of bit stuffing. Whenever a transmitter detects five consecutive bits (including stuff bits) of identical value in the bit stream to be transmitted, it automatically inserts a complementary bit in the actual transmitted bit stream. Table III gives an example. The remaining bit fields of the data frame or remote frame (CRC delimiter, ACK field, and end of frame) are of fixed form and are not stuffed. The error frame and the overload frame are of fixed form as well and are not coded by the method of bit stuffing. The bit stream in a frame is coded according to the Non-Return to Zero (NRZ) method. This means that the generated bit level is constant during the total bit time.
A frame shall be transferred bit field by bit field, starting with its SOF field. Within a field, the most significant bit shall be transmitted first.

The point in time at which a frame is taken to be valid is different for the transmitter and the receiver of the frame. The frame is valid for the transmitter if there is no error until the end of end of frame. If a frame is corrupted, recovery is processed as described earlier. The frame is valid for the receiver if there is no error until the next to the last bit of end of frame.

Every node transmitting a data frame or a remote frame is bus master during transmission. The bus is considered free by any node after having detected that the bit field intermission has not been interrupted by a dominant bit. An error active node may access the bus as soon as the bus is free. An error passive node, which is the transmitter of the current frame or has been transmitter of the previous frame, may access the bus as soon as suspend transmission is finished, provided that no other node has started transmission meanwhile. Whenever more than one node starts transmitting simultaneously, the node which is transmitting the frame with the highest priority will become the only bus master. The mechanism to resolve the resulting bus access conflict is contention-based arbitration.

MAC data frames and MAC remote frames may be started when the node is allowed to access the bus according the description in the previous paragraph. A MAC error frame is transmitted as specified later. A MAC overload frame is transmitted as specified later.

During arbitration, every transmitter compares the level of the bit transmitted with the level that is monitored on the bus. If these levels are equal, the node may continue to send. When a recessive level is sent and a dominant level is monitored, the node has lost arbitration and must withdraw without sending one more bit. When a dominant level is sent and a recessive level is monitored, the node detects a bit error.

Contention based arbitration is performed on the identifier and on the RTR bit. Among two frames with different identifiers, the higher priority is assigned to the frame the identifier of which has the lower binary value. If a data frame and a remote frame with the same identifier are initiated at the same time, the data frame has the higher priority than the remote frame. This is achieved by assigning according values to the RTR bits.

### Table III. Bit stuffing

<table>
<thead>
<tr>
<th>Destuffed bit stream</th>
<th>Stuffed bit stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000abc</td>
<td>1000001abc</td>
</tr>
<tr>
<td>011111abc</td>
<td>0111110abc</td>
</tr>
</tbody>
</table>

a,b,c ∈ {0,1}
Modeling and implementation of a CAN controller

Besides the principle that transmission may be initiated only when the bus is free, there exist further principles for the resolution of collision. These principles are:

- Within one system, each information must be assigned by a unique identifier.
- A data frame with a given identifier and a non-zero data length code may only be initiated by one node.
- Remote frames may only be transmitted with a system-wide determined data length code, which is the data length code of the corresponding data frame. Simultaneous transmission of remote frames with identical identifiers and different data length codes lead to unresolvable collisions.

The MAC sublayer provides the following mechanisms for error detection: Monitoring, stuff rule check, frame check, 15 bit cyclic redundancy check, and acknowledgement check. There are five different error types (which are not mutually exclusive): Bit error, stuff error, CRC error, form error, and acknowledgement error.

A node that is sending a bit on the bus also monitors the bus. A bit error is detected at that bit time when the bit value that is monitored differs from the bit value sent. There are two exceptions. First, a dominant bit does not lead to a bit error when a recessive information bit is sent during arbitration, or a recessive bit is sent during the ACK slot. Secondly, a node sending a passive error flag and detecting a dominant bit does not interpret this as a bit error.

A stuff error is detected at the bit time of the sixth consecutive equal bit level in a frame field that should be coded by the method of bit stuffing.

The CRC sequence consists of the result of the CRC calculation of the transmitter. The receiver calculates the CRC in the same way as the transmitter. A CRC error is detected when the calculated CRC sequence is not equal to the received one.

A form error is detected when a fixed-form bit field contains one or more illegal bits. There is one exception on this. A receiver monitoring a dominant bit at the last bit of end of frame does not interpret this as a form error.

An acknowledgement error is detected by a transmitter whenever it does not monitor a dominant bit during the ACK slot.

Whenever one of these errors is detected, the LLC sublayer will be informed. As a consequence, the MAC sublayer initiates the transmission of an error flag.

Whenever a bit error, stuff error, form error, or acknowledgement error is detected by any node, transmission of an error flag is started by the respective node at the next bit. Whenever a CRC error is detected, transmission of an error frame starts at the bit following the ACK delimiter, unless an error frame for another error condition has already been started.
The following conditions lead to the transmission of an overload frame:

1. LLC-requested overload frame (initiated by the LLC sublayer); Internal conditions of a receiver, which requires a delay of the next MAC data frame or MAC remote frame.
2. Reactive overload frame (initiated by the MAC sublayer):
   • Detection of a dominant bit during intermission
   • Detection of a dominant bit in the last bit of end of frame by a receiver.

An LLC requested overload frame is only allowed to be started at the first bit of an expected intermission, whereas reactive overload frames start one bit after detecting the dominant bit due to condition 2. The start of a reactive overload frame due to condition 2 is allowed but not required to be implemented. At most two LLC overload frames may be generated to delay the next MAC data frame or MAC remote frame.

### 2.4 The physical layer

The physical layer is an electrical circuit realisation that connects an ECU (Electronic Control Unit) to a bus. The total number of ECUs will be limited by the electric loads on the bus line. The physical layer is specified for high speed applications (up to 1 Mbit/s).

The physical layer is modelled according to the LAN standard specification as in ISO 8802-3 (see figure 10). The physical layer is divided into three parts: the Physical Signaling (PLS), the Physical Medium Attachment (PMA), and the Medium Dependent Interface (MDI). The PLS encompasses those functions related to bit representation, timing and synchronization. The Medium Access Unit (MAU) denotes the functional part of the physical layer used to couple the node to the transmission medium. The MAU consists of the PMA and the MDI. The PMA sublayer encompasses the functional circuitry for bus line transmission or reception and may provide means for bus failure detection. The MDI encompasses the mechanical and electrical interface between the physical medium and the PMA. In appendix A, the services and primitives of the physical layer are given.
The PLS sublayer

The bit time, $t_b$, is defined as the duration of one bit. Bus management functions executed within the bit time frame, such as ECU synchronization behaviour, network transmission delay compensation, and sample point positioning, are defined by the programmable bit timing logic of the CAN protocol IC (Integrated Circuit).

First we will give some definitions:

**Nominal bit rate:**
The nominal bit rate gives the number of bits per second transmitted in the absence of resynchronization by an ideal transmitter.

**Nominal bit time:**
The nominal bit time is the reciprocal value of the nominal bit rate, i.e. Nominal bit time = $1$/Nominal bit rate. The nominal bit time can be thought of as being divided into separate, non overlapping time segments. These segments form the bit time as shown in figure 11. In this figure we can distinguish four time segments: The synchronization segment (Sync_Seg), The propagation segment (Prop_Seg), The phase buffer segment 1 (Phase_Seg1), and the phase buffer segment 2 (Phase_Seg2). The synchronization segment is used to synchronize the various ECU's on the bus. An edge is expected within this segment. The propagation segment is used to compensate physical delay times within the network. These delay times consist of the signal propagation time on the bus line and the internal delay time of the ECU's. The phase segment one and two are used to compensate for edge phase errors. These segments can be lengthened or shortened by resynchronization.
Modeling and implementation of a CAN controller

Sample point:
The sample point is the point of time at which the bus level is read and interpreted as the value of that respective bit. Its location is at the end of phase segment 1.

Information processing time:
The information processing time is the time segment starting with the sample point, reserved for calculation of the subsequent bit level.

Time quantum:
The time quantum is a fixed unit of time derived from the oscillator period. There exists a programmable prescaler, with integral values ranging at least from 1 to 32. Starting with the minimum time quantum, the time quantum can have a length of:
\[
\text{Time quantum} = m \times \text{minimum time quantum}
\]
Here is \( m \) the value of the prescaler.

Programming of the bit time

We will now look at the length the individual time segments can have. The synchronization segment is one time quantum long. The propagation segment and the phase segment 1 are programmable to be 1 up to and including 8 time quanta long. The phase segment 2 is the maximum of phase segment 1 and the information processing time. The information processing time is less than or equal to two time quanta long. The total number of time quanta in a bit time has to be programmable at least from 8 to 25. The frequencies of the oscillators in the different ECU's must be co-ordinated in order to provide a system-wide specified time quantum.

Let us look at the synchronization for a moment. There are two types of synchronization: Hard synchronization and resynchronization. They obey the following rules:

1. Only one synchronization within one bit time is allowed.
2. An edge will be used for synchronization only if the value detected at the previous sample point differs from the bus value immediately after the edge.
3. Hard synchronization is performed during bus idle whenever there is a recessive to dominant edge.
4. All other recessive to dominant edges (and optional dominant to recessive edges in case of low bit rates) fulfilling the rules 1 and 2 will be used for resynchronization. There is one exception on this rule: If only recessive to dominant edges are used, a
transmitter will not perform resynchronization as result of a recessive to dominant edge with a positive phase error (see below).

As a result of resynchronization, phase segment 1 may be lengthened or phase segment 2 may be shortened. The amount of lengthening and shortening the phase buffer segments has an upper limit given by the resynchronization jump width. The resynchronization jump width shall be programmable between 1 and the minimum of 4 and the value of phase segment 1. Clocking information is derived from transitions from one bit value to the other. The property that (due to the bit stuffing) only a fixed maximum number of successive bits have the same value, provides the possibility of resynchronizing a bus unit to the bit stream during a frame. The maximum length between two transitions which can be used for resynchronization is 29 bit times.

**Phase error of a resynchronization edge:**
The phase error, \( e \), of an edge is given by the positioning of the edge relative to the synchronization segment, measured in time quanta. Table IV gives the definition of the sign of the phase error.

<table>
<thead>
<tr>
<th>Phase error ( e )</th>
<th>Position of the edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e = 0 )</td>
<td>The edge lies within the synchronization segment.</td>
</tr>
<tr>
<td>( e &gt; 0 )</td>
<td>The edge lies before the sample point.</td>
</tr>
<tr>
<td>( e &lt; 0 )</td>
<td>The edge lies after the sample point.</td>
</tr>
</tbody>
</table>

**Hard synchronization**

After a hard synchronization, the bit time is restarted by each bit timing logic unit with the synchronization segment. Thus hard synchronization forces the edge which has caused the hard synchronization to lie within the synchronization segment of the restarted bit time.

**Resynchronization**

When the magnitude phase error of the edge which causes resynchronization is less than or equal to the programmed value of the resynchronization jump width, the effect of resynchronization is the same as that of hard synchronization. When the magnitude of the phase error is larger than the resynchronization jump width, and:

- if the phase error \( e \) is positive, then phase segment 1 is lengthened by an amount equal to the resynchronization jump width;
- if the phase error \( e \) is negative, then phase segment 2 is shortened by an amount equal to the resynchronization jump width.
The PMA and MDI sublayers

There will not be much said about the PMA and MDI sublayers. The implementation of these two layers will be off-chip, so there is no need for us to model these layers. The interested user can read about them in the CAN specifications [2]. The messages and primitives that are passed from the PLS to the PMA and vice versa can be found in appendix A.

2.5 The fault confinement entity

The objective of fault confinement is to preserve a high availability of data transmission, even if there is a defective node somewhere in the system. Therefore, the fault confinement strategies have to provide reliability on distinction between temporary errors and permanent failures, and on localisation and switching off faulty nodes.

Any node is supplied with a transmit error counter and a receive error counter. The first one registers the number of errors during the transmission and the second one registers errors during the reception of frames. If frames are sent or received correctly, the counters are decreased. In case of errors, the counters are increased more than they are decreased in case of absence of errors. The ratio in which the counters are increased or decreased depends on the acceptable ratio of invalid and valid frames on the bus. At any time, levels of the error counters reflect the relative frequency of previous disturbances. Depending on predetermined counter values, the behaviour of nodes in respect to errors is modified. This ranges from a prohibition of sending error flags in order to cancel frames, up to switching off nodes which often would send invalid frames.

Figure 12 gives the fault confinement interface. In appendix A, the description of the messages and primitives is given.
Rules of fault confinement

With respect to fault confinement, a node may be in one of the three states, depending on the level of the error counters. These states are: error active, error passive, and bus off. The error counters are modified according to the following rules. More than one rule may apply during a given frame transfer:
1. When a receiver detects an error, the receive error counter will be increased by 1, except when the error was a bit error during sending of an error flag or an overload flag.

2. When a receiver detects a dominant bit as the first bit after sending an error flag, the receive error counter will be increased by 8.

3. When a transmitter sends an error flag, the transmit error counter is increased by 8. There are two exceptions:
   1. If the transmitter is error passive and it detects an acknowledgement error because of not detecting a dominant ACK and does not detect a dominant bit while sending its passive error flag.
   2. If the transmitter sends an error flag because of a stuff error occurred during arbitration whereby the stuff bit is located before the RTR bit, and should have been recessive, and has been sent as recessive but monitored as dominant.

   In case of exception 1 and 2, the transmit error counter remains unchanged.

4. When an error active transmitter detects a bit error while sending an active error flag or when any transmitter detects a bit error while sending an overload flag, the transmit error counter is increased by 8. If an error passive transmitter detects a dominant bit while sending a passive error flag, the transmitter does not interpret this as an error, hence does not start a new error flag.

5. If an error active receiver detects a bit error while sending an active error flag, or any receiver detects a bit error while sending an overload flag, the receive error counter is increased by 8. If an error passive receiver detects a dominant bit while sending a passive error flag, the receiver does not interpret this as an error, hence does not start a new error flag.

6. Any node tolerates up to seven consecutive dominant bits after sending an active error flag or a passive error flag. After detecting the eighth consecutive dominant bit and after each sequence of additional eight consecutive dominant bits, each transmitter increases its transmit error counter by 8 and every receiver increases its receive error counter by 8.

7. After the successful transmission of a frame (getting ACK and no error has been detected until end of frame is finished), the transmit error counter is decreased by 1, unless it was already 0.

8. After the successful reception of a frame (reception without error up to the ACK slot and successful sending of the ACK bit), the receive error counter is decreased by 1, if it was between 1 and 127. If the receive error counter was 0, it stays 0, and if it was greater than 127, then it will be set to a value between 119 and 127.

Start up

If during system start-up only one node is on line and if this node transmits some frame, it will get no acknowledgement, detect an error and repeat the frame. It can become error passive but not bus off due to this reason. A node which is switched off or bus off has to run through a start-up routine in order to:

- synchronize with already available nodes before starting to transmit.

Synchronization is achieved when 11 recessive bits equivalent to ACK delimiter,
end of frame, and intermission, or equivalent to error or overload delimiter and intermission, have been detected.

- Wait for other nodes without becoming bus off if there is no other node available at the moment.

Now we will look at the relationship between the states error active, error passive, and bus off. Refer to figure 13 for a visual overview.

If the transmit error counter or the receive error counter of a node exceeds 127, the supervisor requests the MAC sublayer to set the corresponding node into the error passive state. An error condition letting a node become error passive causes the node to send an active error flag. An error passive node becomes error active again, when both the transmit error counter and the receive error counter are less than or equal to 127.

If the transmit error counter of a node is greater than 255, the supervisor requests the physical layer to set the node into the bus off state. A bus off state node is not allowed to have any influence on the bus. It must not send any frames nor send acknowledgement, error frames, or overload frames. A node which is bus off is permitted to become error active (no longer bus off) with its error counters both set to zero after having monitored 128 occurrences of 11 consecutive recessive bits on the bus.

Figure 13. Node status transition diagram

TEC=Transmit Error Counter
REC=Receive Error Counter
3 A brief introduction to the Ward and Mellor modeling method

When designing real time systems, it is often necessary to make some kind of formal description first. Such a formal description gives us the opportunity to get acquainted with the functionalities of the system without being bothered with implementation details. For this, Paul T. Ward and Stephen J. Mellor have designed a method for structured development of real time systems. In this chapter, a brief overview of the method is given. For interested readers I refer to [2].

3.1 Description of the Ward and Mellor tools

In the Ward and Mellor method, the formal description is called an essential model. In order to depict this model, tools are used. These tools are:

- Terminators
- Data transformations
- Control transformations
- Discrete flows
- Continuous flows
- Event flows
- Data stores
- Control stores

Now we will discuss the different tools. Figure 14 gives an overview of how the tools are depicted. In figure 15 up to and including figure 19, a simple example can be found of the use of (some of) these tools.

![Diagram of modeling tools](image)

**Figure 14.** Modeling tools
3.1.1 Terminators

A system is always connected to an environment. The connection to the environment is indicated by making use of terminators. A terminator is depicted by a rectangle (see figure 14). Terminators are only used in the context diagram. The context diagram is the upper most transformation schema in the hierarchy.

3.1.2 Data transformations

Data is processed within a data transformation. A data transformation is depicted by a circle with a solid line (see figure 14). A hierarchy can be created by defining the content of the data transformation as another transformation schema. How this works, will be explained in the example.

3.1.3 Control transformations

Control transformations are used to control functions carried out by data transformations. Within these control transformations, only processing carried out on event flows (see below) is taken care of. A control transformation is depicted as a circle with a dashed line (see figure 14). It is also possible to create a hierarchy with control transformations. The only tools used in such a hierarchy are other control transformations and event flows.

3.1.4 Discrete flows

Discrete flows are used to depict data that is not continuously available. A discrete flow is depicted by an arrow with a solid line and one arrow point (see figure 14). Discrete flows can be combined and can be split into multiple discrete flows.

3.1.5 Continuous flows

Continuous flows are used when data is continuously available. A continuous flow is depicted by an arrow with a solid line and two arrow points (see figure 14). Continuous flows can also be combined and split into multiple continuous flows.

3.1.6 Event flows

Events flows are used when something happened. Only the knowledge that something happened is important. If, for example, a clock generator generates a pulse every 10 seconds, the value of the pulse is not important. Only the occurrence of the pulse is important. An event flow is depicted by an arrow with a dashed line and a single
arrow point (see figure 14). Event flows may not be combined, but it is allowed to split one into multiple event flows.

3.1.7 Data stores

Data stores are used to store data. They are depicted by two parallel solid lines (see figure 14). Data can be written into them, and read out of them.

3.1.8 Control stores

Control stores are used to remember the occurrence of an event. They are depicted by two parallel dashed lines (see figure 14).

3.1.9 Ward and Mellor state transition diagrams

To describe the content of a control transformation at the lowest level in the hierarchy, state transition diagrams are used. In contrast to states in 'normal' state transition diagrams, Ward and Mellor states are depicted by rectangles. If circles would be used in Ward and Mellor state transition diagrams, they could easily be mistaken for a data transformation diagram. An arrow is drawn from one state to another (or to itself) to indicate a state transition. The text next to the arrow indicates when something has to be done. It is followed by a horizontal line, and beneath that line, the action to be taken is described. See section 3.2 for an example.

3.2 An example: sample and hold

We will briefly discuss an example, so the usage of the tools described in section 3.1 will become more clear. The example is a sample and hold unit. Incoming data is sampled every second. The sampled value will be presented to the output during the whole sample period. After a second, the new sample is presented to the output.

Figure 15 contains the context diagram of this system. We can see one terminator (world). This terminator represents the connection to the world. We also can see one data transformation (sample&hold). This data transformation takes care of the complete functionality of the sample and hold system. Furthermore, two continuous flows are present in figure 15: in and out. The in flow represents the continuous value of the signal that has to be sampled. The out flow represents the continuous value of the sampled input value. This value is hold constant during the whole sample period.
Fig. 15. Context diagram of the sample and hold example

In figure 16, the content of the sample&hold transformation is given. This way, a lower level in the hierarchy is created.

Fig. 16. Content of the sample&hold transformation

Every time the event sample occurs, the sample transformation puts the value of the continuous flow in to the discrete flow sampled value. Figure 17 gives the state transition diagram of this transformation. This is a strange way to use state transition diagrams.

Fig. 17. State transition diagram of the sample transformation

The timer transformation makes sure that the trigger event flow is asserted every second. Figure 18 gives the state transition diagram of this transformation. An internal event (1 second passed) is the reason for executing the sample event.
Every time a sampled value is presented to the hold transformation, this transformation stores this value in the sampled value store. Furthermore, the hold transformation puts the value stored in the hold value store to the continuous flow out. Figure 19 presents the state transition diagram of the hold transformation. The right arrow does not contain text on when the action has to be taken care of. This means that it should be done all the time.

3.3 Deviations from the Ward and Mellor method

The Ward and Mellor method describes several rules which have to be followed. We do not follow all these rules. First of all, we do not use any control transformations in the final model. In the Ward and Mellor method, these transformations are used to control the data transformations. The reason that we do not use these transformations in this way, is that it would complicate our model extensively. As the main reason for making a model is getting insight in the problem, extra complication introduced by certain rules is in contradiction with our objectives. Another rule I did not obey, is the rule for the naming of the transformations. According to the Ward and Mellor method, the naming of a transformation should be a verb (or a sentence with a verb in it), which indicates the action that is taking place in the transformation.
4 Presentation of the model of CAN

In this chapter, the model of the CAN protocol is presented. This is done in three iterations. The first two iterations are used to present additions to the primitives and messages defined in the CAN standard. The third iteration represents the final model. Without distinction, control transformations and data transformations are addressed as 'transformations'.

4.1 First iteration

After examining the ISO/DIS 11898 draft international standard [3] (Further referred to as ‘the CAN standard’), I created a context diagram as shown in figure 20. The accompanying data dictionary is given in appendix B. In figure 20 all the primitives specified in the CAN standard can be found. The data transformation is called CAN\{PMA,MDI\} and this means CAN without the Physical Medium Attachment and without the Medium Dependent Interface. According to the CAN standard, the PMA sublayer ‘encompasses the functional circuitry for bus line transmission and reception and may provide means for bus failure detection’. The MDI sublayer ‘encompasses the mechanical and electrical interface between the physical medium and the MAU’. As we have seen earlier they are off-chip, so they do not have to be modelled.

Figure 20. The context diagram of the first iteration

During discussion of this context diagram, we decided to make some major adjustments. First of all, some extensions are needed. The first extension is the Filter_Params primitive. The CAN standard states that one of the tasks of the Logic Link Control is frame acceptance filtering. Incoming frames are filtered on their identifier. To do so, the node has to know the identifiers of the frames it has to accept (or the identifiers of the frames it has to reject). Therefore, a primitive called
Filter_Params is added to the primitives in the CAN standard. The Filter_Params primitive consists of one or more parameters that have to be added to or deleted from the filter parameters list.

The second extension are the L_ABORT.request and L_ABORT.confirm primitives. When an L_DATA.request or L_REMOTE.request is initiated, the node tries to transmit the frame associated with the request. If the frame has lost arbitration or is disturbed by errors during transmission, it is automatically retransmitted. This process continues until the frame is transmitted without encountering any error or when the node turns to node off mode due to transmit error counter overflow. Suppose that a frame has lost arbitration. Also suppose that another request (with a higher priority) has to be send. The request with the higher priority should be send before the request that is already being processed. One way to achieve this is to make use of a priority queue. We chose a solution with the abort primitives. When an L_ABORT.request is initiated and the frame in the LLC sublayer is being sent, the LLC awaits the arrival of a L_DATA.confirm or L_REMOTE.confirm. If this confirm is COMPLETE, the L_ABORT.confirm will also send COMPLETE to the user to indicate the successful transmission of the frame. If this confirm is NOT_COMPLETE, the L_ABORT.confirm will be ABORTED and the LLC will refrain from further retransmission of the frame.

We will not dig any deeper into the CAN\{PMA,MDI\} transformation. In the next section, the changes discussed here, will be used.

4.2 Second iteration

4.2.1 The context diagram

In figure 21 we can find the combination of the primitives as stated in the previous section. The data dictionary can be found in appendix C. The L_DRA.request primitive is a combination of the L_DATA.request primitive, the L_REMOTE.request primitive, and the L_ABORT.request primitive. The L_DRA.confirm primitive is composed similarly from L_DATA.confirm, L_REMOTE.confirm, and L_ABORT.confirm. The L_DR.indication primitive is composed of L_DATA.indication and L_REMOTE.indication. We can also see the newly introduced Filter_Params flow.
4.2.2 The CAN\{PMA,MDI\} transformation

In figure 22, the content of the CAN\{PMA,MDI\} transformation of figure 21 is given. The data dictionary can be found in appendix C. The transformations that are found in this figure are deduced from the CAN standard. The Logic Link Control (LLC) transformation takes care of the frame acceptance filtering, the overload
notification, and the recovery management. The Medium Access Control (MAC) transformation

**Figure 22.** The content of the CAN\{(PMA,MDI) transformation of figure 21
takes care of the transmission and reception of a frame. The Physical Layer Signaling (PLS) transformation takes care of the bit representation, timing, and synchronization.

4.2.3 The LLC transformation

In figure 23, the content of the LLC transformation of figure 22 is given. The data dictionary can be found in appendix C. The buffer requests transformation takes care of the buffering of L_DRA.request primitives. The frame acceptance filtering transformation makes sure that frames are passed to the user only when their identifiers match one of the filter parameters. The node status is determined by the determine Node_Status transformation. Finally, the reset is handled by the Reset transformation. The reset_LLCC event flow is split into the reset_LLCC_faf and reset_LLCC_br event flows. These event flows represent the same event as the reset_LLCC event flow. The difference in names is due to the use of the tool 'system architect'.

![Diagram](image)

**Figure 23.** The content of the LLC transformation of figure 22

4.2.4 The MAC transformation

In figure 24, the content of the MAC transformation of figure 22 is given. The data dictionary can be found in appendix C. The Transmit Data Encapsulation builds a MAC frame from an LLC frame. The Transmit Media Access Management takes care of the initialization of the transmit process, serialization of the MAC frame, bit stuffing, arbitration, error detection, acknowledgement check, recognition of an overload condition, overload frame construction and initiation of transmission, error frame construction and initiation of transmission, and presentation of a serial bit stream to the physical layer. The Receive Data Decapsulation extracts the LLC frame.
from the MAC frame. The Receive Media Access Management takes care of the
reception of a serial bit stream from the physical layer, deserialization, bit destuffing,
error detection, transmission of acknowledgement, error frame construction and
initiation of transmission, recognition of an overload condition, and reactive overload
frame construction and initiation of transmission. The control transformation makes
sure that everything the other transformations have to do will be done.

Figure 24. The content of the MAC transformation of figure 22

4.3 Third iteration

After discussing the results of the second iteration and after studying the physical
layer signaling more carefully, it turned out that another additional primitive was
needed. In the CAN standard, several parameters of the PLS have to be
programmable. Therefore, we introduced the primitive Timing_Params. Another
major change has been made. The USER terminator is divided into the M3S extension
& application terminator and the M3S management terminator, hereby splitting the
management task from the tasks concerning data. The transformations on the lowest
level are described by state transition diagrams. These diagrams can be found in
appendix E. The state transition diagrams of the TMAMcontrol, SFcontrol,
RMAMcontrol, and DFcontrol transformations are too big to print on one sheet.
Therefore they will not be present in appendix E.
4.3.1 The context diagram

In figure 25, the context diagram that depicts the changes described earlier is given. The data dictionary can be found in appendix D. We can see if we compare the context diagram of figure 25 with the context diagram of figure 21 that the Reset_Request, Reset_Response, Filter_Params, and Node_Status primitives are passed to the M3S management terminator at the right side instead of to the USER terminator on the upper side. The Timing_Params primitive is also passed to the M3S management terminator. All these primitives are management related. They deal with those functions of the node that do not concern the data transfer of the node.

![Figure 25. The context diagram of the third iteration](image-url)
4.3.2 The CAN transformation

In figure 26, the content of the CAN transformation of figure 25 is given. The data

![Diagram of CAN transformation]

Figure 26. The content of the CAN transformation of figure 25
dictionary can be found in appendix D. Comparing this diagram with the diagram of figure 22 we can see that the LLC transformation, the MAC transformation, and the PLS transformation are still present although their content may have been changed. Instead of the FCE transformation of figure 22 we have introduced a MANAGEMENT transformation. The reasons for doing this are the same as they were for introducing the M3S management terminator in figure 21. We will now have a discussion on the differences between the FCE and the MANAGEMENT.

The FCE takes care of the fault confinement. It contains and manages the transmit and receive error counters, and takes care of the bus status the node is in. The MANAGEMENT transformation also takes care of the bus status. It does not contain the transmit and receive error counters though; they have been moved to the MAC transformation. So the complexity that is present within the FCE can not be found in the MANAGEMENT transformation. For a considerable part, this complexity has been moved to the MAC transformation.

Comparing this LLC transformation with the one from figure 22 we see that the Reset_Request, the Reset_Response, and the Node_Status primitives are not connected to the LLC anymore, but to the MANAGEMENT. This is done because these messages involve typical management functions.

The MAC transformation has changed considerably because of the extra complexity imposed by moving a great amount of the FCE complexity into this transformation. We will come back on this later.

The PLS transformation has not changed except for the extra message concerning the timing parameters.

4.3.3 The LLC transformation

In figure 27 we can see the content of the LLC transformation of figure 26. The data dictionary can be found in appendix D. When we compare the content of the LLC of this figure with the LLC of figure 23 we can see that the determine node status and Reset transformations have been removed. Their tasks are taken care of by the MANAGEMENT transformation. The other two transformations are the same as in figure 23.
The BUFFER REQUESTS transformation

In figure 28 we can see the BUFFER REQUESTS transformation of figure 27. The data dictionary can be found in appendix D. The BUFFER REQUESTS transformation takes care of the buffering of frames in order to be able to retransmit them if necessary. If an L_DATA.request or L_REMOTE.request arrives, it is passed to the Add Request transformation and the Retrieve Request transformation. The Add Request transformation stores the request in the REQUEST store. The Remove Request transformation removes the request from the REQUEST store after successful transmission. The Retrieve Request transformation passes the request to MA_DR.request_LLc when it first arrives. In case of a retransmission it gets the request from the REQUEST store and passes it to MA_DR.request_LLc. The br_control transformation controls the process of retransmission.
Figure 28. The content of the BUFFER REQUESTS transformation of figure 27
The FRAME ACCEPTANCE FILTERING transformation

In figure 29 we can see the content of the FRAME ACCEPTANCE FILTERING transformation of figure 27. The data dictionary can be found in appendix D. There is a store that contains the filter parameters list. This list is composed of an enumeration of the identifiers which have to be passed to the M3S extension & application. The M3S management can change those filter parameters by means of the Filter_Params primitive. If a change is executed, the change filter params transformation takes care of the addition of the parameter(s) to or removal from the list.

The LLC is able to queue incoming frames. In this model, the queue can contain one frame. Surely this can easily be expanded to a larger queue, but this is not important for the essential model. This queuing is controlled by the queue control transformation. If the queue is full, an event is sent to the overload control transformation. On reception, the overload control initiates an MA_OVLD.request_LLCLC. After reception of an MA_OVLD.confirm_LLCLC, the overload control transformation can initiate a second overload request when needed.

The actual filtering is achieved by the compare ID transformation. This transformation compares the identifier of the L_DR_frame with the identifiers stored in the filter params list. If a match occurs, the L_DR_frame is passed to the L_DR.indication. If there is no match, the frame is discarded.
Figure 29. The content of the FRAME ACCEPTANCE FILTERING transformation of figure 27
4.3.4 The MAC transformation

In figure 30 we can see the content of the MAC transformation of figure 26. The data dictionary can be found in appendix D. We can distinguish three main parts in figure 30: The transmit part, the receive part, and the management part. The transmit part consists of the TRANSMIT transformation and the TRANSMIT ERROR COUNTER transformation. It takes care of the complete transmit process. The receive part consists of the RECEIVE transformation and the RECEIVE ERROR COUNTER transformation. It takes care of the complete receive process. The management is covered by the MAC MANAGEMENT transformation and takes care of all management related tasks.

![Diagram of MAC transformation](image)

**Figure 30.** The content of the MAC transformation of figure 26

The TRANSMIT transformation

In figure 31 we can see the content of the TRANSMIT transformation of figure 30. The data dictionary can be found in appendix D. The Transmit Data Encapsulation takes care of the creation of a MAC frame from an LLC frame. It therefore calculates the CRC value and adds all heading information to the LLC frame. The Transmit Media Access Management takes care of the serialization of the frame, the insertion of stuff bits, the arbitration, error detection, overload recognition, overload frame construction and initiation of transmission, and error frame construction and initiation of transmission.
Figure 31. The content of the TRANSMIT transformation of figure 30
The Transmit Data Encapsulation transformation

In figure 32 the content of the Transmit Data Encapsulation transformation of figure 31 is given. The data dictionary can be found in appendix D. An incoming \texttt{MA\_DR.request\_MAC\_TR} primitive is passed to the construct polynomial transformation and to the generate MAC frame transformation. The construct polynomial transformation takes care of the construction of the polynomial that is used to perform the CRC calculation. This polynomial is a composition of the following fields: the SOF field, the arbitration field, the control field, and the data field. The lowest fifteen coefficients are zero. The resulting polynomial is stored in the polynomial\_T store. The generator polynomial is kept in the generator\_T store. When the construction of the polynomial is completed, the divide modulo 2 transformation takes care of the actual computation of the CRC value. Finally, the generate MAC frame transformation builds a MAC frame from the CRC value, and the \texttt{MA\_DR.request\_MAC\_TR\_GMF} primitive by adding the header and trailer information.
Figure 32. The content of the Transmit Data Encapsulation transformation of figure 31
The Transmit Media Access Management transformation

In figure 33, the content of transmit media access management transformation of figure 31 is given. The data dictionary can be found in appendix D. The Serialize frame transformation serializes the incoming frame. The bit stuffing transformation, when enabled, adds stuff bits to the serialized bit stream when necessary. If the send bit transformation is enabled, it passes the stuffed bit stream to the PLS_TD.request_sb primitive. The Tcompare transformation compares the transmitted bit with the received bit. The TMAMcontrol transformation controls the complete transmit process and passes status and errors to the outside of the Transmit Media Access Management transformation.
Figure 33. The content of the Transmit Media Access Management transformation of figure 31
The Serialize frame transformation

The content of the Serialize frame transformation of figure 33 is given in figure 34. The data dictionary can be found in appendix D. The manage Tbuffer transformation controls the content of the Tbuffer store. If a Transmit MAC frame message arrives, it puts this message into the Tbuffer store. If a TMAMerror or TMAMoverload event takes place, it puts the appropriate error frame or an overload frame into the Tbuffer store. The next bit from Tbuffer transformation passes the next bit from the Tbuffer to the transmit bit primitive when a next Tbit event arrives. The SFcontrol transformation keeps track of the place within the frame and controls the complete serialize frame process.
Figure 34. The content of the Serialize frame transformation of figure 33
The bit stuffing transformation

In figure 35 the content of the bit stuffing transformation of figure 33 is given. The data dictionary can be found in appendix D. An incoming bit (transmit bit) is compared with the bit stored in the Tprevious store. If both bits match, the Tcounter is increased by one. If they do not match, the Tcounter is set to zero. In both cases the bit is passed to the Tbit primitive. The add Tprevious transformation stores the Tbit_ap in the Tprevious store. The detect Tstuff bit transformation triggers the Tstuff bit_dsb event flow every time the Tcounter has reached the value five. If this happens, the Tcounter is set to zero. When the stuff bit_send event flow is triggered, the Tsend transformation passes the complementary value of the Tbit_send primitive to the stuffed bit primitive, thus inserting a stuff bit. Then it sends the original bit (still stored in the Tprevious store) to the stuffed bit primitive.
Modeling and implementation of a CAN controller

Figure 35. The content of the bit stuffing transformation of figure 33
The RECEIVE transformation

In figure 36 the content of the RECEIVE transformation of figure 30 is given. The data dictionary can be found in appendix D. The Receive Data Decapsulation takes care of the construction of an LLC frame from the incoming MAC frame. The Receive Media Access Management transformation takes care of the following tasks: The deserialization of the incoming bit stream, the deletion of stuff bits, error detection, transmission of acknowledgement, error frame construction and initiation of transmission, recognition of an overload condition, and reactive overload frame construction and initiation of transmission.
Figure 36. The content of the RECEIVE transformation of figure 30
The Receive Data Decapsulation transformation

In figure 37 the content of the Receive Date Decapsulation transformation of figure 36

Figure 37. The content of the Receive Data Decapsulation transformation of figure 36
is given. The data dictionary can be found in appendix D. The Decapsulate MAC
frame transformation divides the received MAC frame into four fields: The identifier
(ID), the data length code (DLC), the DATA, and the RTR bit. The construct LLC
frame constructs an LLC frame out of these four fields. The RTR bit determines
whether the MA_DR.indication is a MA_DATA.indication or a
MA_REMOTE.indication.

**The Receive Media Access Management transformation**

In figure 38 the content of the Receive Media Access Management transformation of
figure 36 is given. The data dictionary can be found in appendix D. The bit destuffing
transformation takes care of the deletion of stuff bits. The deserialize frame
transformation transforms the serial bit stream into a parallel bit stream. The Rsend
transformation is responsible for sending the error or overload frame, when necessary.
The Rcompare transformation compares the outgoing bits with the incoming bits in
case an error frame or an overload frame is being send. The RMAMcontrol controls
the Receive Media Access Management transformation.
Figure 38. The content of the Receive Media Access Management transformation of figure 36

Rsend

In figure 39 the content of the Rsend transformation of figure 38 is given. The data dictionary can be found in appendix D. If the node turns from error active mode into
error passive mode or from error passive mode into error active mode, the error frame stored in Rerror frame has to be adjusted. This is taken care of by the add Rerror frame transformation. The send Rbit transformation takes care of the actual sending of an error frame or an overload frame.

Figure 39. The content of the Rsend transformation of figure 38

The bit destuffing transformation

In figure 40, the content of the bit destuffing transformation of figure 38 is given. The data dictionary can be found in appendix D. When a bit arrives, the Compare and update Rcounter transformation compares that bit with the bit stored in the Rprevious
store. If they match, the Rcounter is increased by one. If the Rcounter has reached the value five, the bit is not passed to the Rbit discrete flow; it was a stuff bit and thus has

![Diagram](image)

**Figure 40.** The content of the bit destuffing transformation of figure 38
to be deleted. If the Rcounter has not reached the value five yet, the bit is passed to Rbit. The add Rprevious transformation adds the Rbit_ap to the Rprevious store.

**The deserialize frame transformation**

In figure 41, the content of the deserialize frame transformation of figure 38 is given. The data dictionary can be found in appendix D. On arrival of a new bit, the next Rbit to Rbuffer transformation puts this bit into the Rbuffer. The DFcontrol transformation takes care of the control of the deserialize frame transformation. After the fields necessary for the CRC calculation are received, the CRC calculation transformation receives the calculate event and starts calculating the CRC value. When this CRC value is available, the CRC calculation stores it in the CRC store. The send Rframe transformation passes the complete frame to the Receive Data Decapsulation transformation upon reception of a last bit received event.
Figure 41. The content of the deserialize frame transformation of figure 38
The TRANSMIT ERROR COUNTER transformation

In figure 42 the content of the TRANSMIT ERROR COUNTER transformation of figure 30 is given. The data dictionary can be found in appendix D. The te_control transformation takes care of the changes the transmit error counter (TEC) is submitted to. Depending on the error indicated by Terror it updates the TEC store. The Tbound crossing message generation transformation detects if the TEC has reached a value on which the node has to change its mode. For example, if the TEC has reached the value of 128, the Tbound crossing message generation transformation gives the message error passive.

![Diagram](image_url)

**Figure 42.** The content of the TRANSMIT ERROR COUNTER transformation of figure 30

The RECEIVE ERROR COUNTER transformation

In figure 43 the content of the RECEIVE ERROR COUNTER transformation of figure 30 is given. The data dictionary can be found in appendix D. The re_control transformation takes care of the changes the receive error counter (REC) is submitted to. Depending on the error indicated by Rerror it updates the REC store. The Rbound crossing message generation transformation detects if the REC has reached a value on which the node has to change its mode. For example, if the REC has reached the value of 128, the Rbound crossing message generation transformation gives the message error passive.
4.3.5 The PLS transformation

In figure 44 the content of the PLS transformation of figure 26 is given. The data dictionary can be found in appendix D. The Time Quantum GENERATION transformation triggers TQ every time quantum. The time quantum is derived from a minimum value called the minimum time quantum. This minimum time quantum is multiplied by the prescaler value stored in the PARAMS store to get the time quantum.
Figure 44. The content of the PLS transformation of figure 26

The SEND transformation triggers the request event flow each time a PLS_DATA.request is received. The PLS_DATA.request is passed to the output on a sync event. The RCV (receive) transformation passes the value on the input flow to the PLS_DATA.indication on a sample event. The SYNC CONTROL transformation keeps track of the place within the bit time. It takes care of resynchronization and indicates the sample point and the point at which a request has to be put on the bus.

The SYNC CONTROL transformation

In figure 45 the content of the SYNC CONTROL transformation of figure 44 is given. The data dictionary can be found in appendix D. The edge detection transformation triggers the edge event flow every time it detects an edge on the bus. When the node is operating at slow bit rates (indicated by a bit in the PARAMS store) both edges (dominant to recessive as well as recessive to dominant) are considered. On the other hand, if the node is operating at fast bit rates, only the recessive to dominant edges are involved.

Figure 45. The content of the SYNC CONTROL transformation of figure 44

The phase error detection transformation determines the phase error. This phase error is needed for resynchronization. The (re)sync transformation takes care of the hard synchronization and resynchronization.
5 Global hardware mapping

In this chapter, the global mapping of the model on hardware is discussed. As mentioned earlier, the IDaSS library has an SDLC controller which we will (partly) use to implement the M3S protocol. Figure 46 shows the rough outline of this controller. We can see a bitprocessor and a byteprocessor. The bitprocessor will take care of the handling of bit stream data. The byteprocessor on the other hand deals with byte-oriented data.

Figure 46. Rough outline of the SDLC controller

5.1 The bitprocessor

In this section we will determine what functionality will be mapped on the bitprocessor. Therefore we will look to our model of the CAN protocol. In figure 21, the content of the CAN transformation of our model is shown. From this we can see that there are four transformations: LLC, MAC, PLS, and MANAGE.

The LLC transformation takes care of frame acceptance filtering, Overload notification, and recovery management. The frame acceptance filtering is evidently byte-oriented, so it has to be implemented on the byte processor. Overload notification is initiated from the LLC in case its internal identifier buffer is full. This also indicates that this functionality has to be mapped on the byteprocessor. Finally, recovery management makes sure a retransmission is initiated in case of loss of arbitration or in case of an error. Therefore it has to state a new request to the MAC sublayer. This is also byte-oriented because the complete request (including request data) has to be passed to the MAC sublayer. Summarizing: All LLC functionality will be mapped on the byteprocessor.

Now we will look at the functionalities of the MAC sublayer. The are summarized in table I and II. When studying these tables we have to draw the conclusion that all these functionalities have to be mapped on the bitprocessor, because they are all involving bit-oriented data.

The PLS transformation encompasses those functions related to bit representation, timing and synchronisation. They clearly are bit-oriented, so these functionalities are mapped on the bitprocessor.

For the other parts of the physical layer, we have already stated that those parts will be off-chip, so no mapping is needed.
The MANAGE transformation involves bus status generation and reset handling. The bus status generation will be implemented on the bitprocessor and the reset handling will be implemented on the byteprocessor.

The bitprocessor of the SDLC controller takes care of a full-duplex protocol. This is not necessary for the CAN controller. Except for the arbitration phase, the CAN controller is transmitting or receiving. This was one of the reasons for me to develop the bitprocessor from scratch. There were two other reasons which led to this decision. The first reason is, that the documentation that was available for the bitprocessor was very brief; the information I needed to be able to adapt the design was not in it. Also several changes were made to the design which were not documented. The second reason is, that the communication protocols were too different to get any advantage when using the bitprocessor of the SDLC controller. The implementation of the bitprocessor will be discussed extensively in chapter 6.

5.2 The byteprocessor

The byteprocessor of the SDLC controller is a parametrizable processor. It has 7 timers on board. There is a DMA controller present. We can also find a parameter storage in the byteprocessor and a master processor interface is present. This master processor interface is implemented as a dual ported register file. Because the byteprocessor is completely parametrizable, it can be kept as small as necessary. For example, when only 2 timers are needed, the remaining timers will be left out.

As we have seen earlier, the (generic part of the) M3S protocol and the LLC part of CAN will be programmed on the byteprocessor. The programming of the M3S protocol will not be discussed here. The LLC part of CAN would put a great burden on the computing power of the byteprocessor. Especially the frame acceptance filtering would become very power consuming. Therefore a special unit is designed to take care of this filtering (see figure 47). The overload notification and the recovery management do not occupy much of the processing power, so they will be carried out by the byteprocessor.

![Figure 47. Byteprocessor with filtering unit](image-url)
6 An implementation of the bitprocessor

In this chapter we will see how the bitprocessor was implemented. This implementation is done making use of the design tool IDaSS. IDaSS is an abbreviation of Interactive Design and Simulation System. For a further contemplation on IDaSS, I refer to the user manual of IDaSS [11].

Before we are going to look at the actual implementation, the internal control of signals is going to be discussed. Almost every data path within the controller is one bit larger than the width of the data. This is because the most significant bit indicates the validity of the data. We will elaborate on this a little bit by using figure 48. In this figure, two functional units (A and B) can be found. They are connected to each other with a bus. Let us suppose that this bus is meant for moving data from unit A to unit B. The data itself can be found on the lowest part of the bus. The validity bit is the most significant bit of the bus. This is illustrated in figure 48.

![Figure 48. Use of a validity bit on the data busses](image)

Now we are ready to visit the implementation. In section 6.1, the toplevel is found. The lower level schematics can be found in section 6.2.

6.1 The toplevel schematic

In figure 49, the toplevel schematic of the bitprocessor can be found. There are five connectors. The in connector is the input from the PMA. The out connector is the output to the PMA. The meaning of the Data connector, the Control connector and the Status connector are not clear at this moment.
6.2 Composition of the bitprocessor

Figure 50 shows the composition of the bitprocessor. From this figure we can see how the bitprocessor is divided into several schematics.

As we can see, there are eight different schematics:

- The PLS schematic
- The transmit schematic (Tx)
- The receive schematic (Rx)
- The bit stuffing schematic (BitStuff)
- The bit destuffing schematic (BitDestuff)
- The CRC schematic
- The Management schematic
- The BackEnd schematic
Before we are going to visit these schematics one by one, we will explain the relation of the model presented in chapter 4 with the implementation found in figure 50. The PLS schematic contains the functionality of the PLS transformation of figure 26. The BitStuff schematic contains the functionality of the bit stuffing transformation of figure 33. The BitDestuff schematic contains the functionality of the bit destuffing transformation of figure 38.

The Tx schematic contains all functionality concerning the transmission of frames or bits (e.g. acknowledgement). The Rx schematic contains all functionality concerning the reception of a frame. The CRC schematic contains all functionality used to compute, compare, and generate the CRC sequence. In stead of the two CRC generators of the model, we used only one CRC schematic to take care of the complete functionality. By doing this, we reduced the amount of hardware used. The Management schematic contains all functionality that is used for comparison, control and error handling. Finally, the BackEnd schematic is used to provide an interface to the byteprocessor. There is no explicit functionality found in the presented model.

The schematics if figure 50 will now be visited one by one.

6.2.1 The PLS schematic

In figure 51 we can find the content of the PLS schematic of figure 50.
In figure 51 we can find three operators: TQgen, ParamsControl and SyncControl. For a description of these operators, see appendix F.

The TQgen operator takes care of the generation of a signal, indicating a time quantum (see chapter 2). The TQcnt register is used as a counter. It counts clock cycles. Every time the TQcntI value equals the value on the m input (prescaler), the TQ output gets the value 1. A 1 on the reset input resets the TQcnt register. This is used for hard synchronization.

The ParamsControl operator creates the value for the timing parameters. These are: the prescaler value, the length of the propagation segment (prop output), the length of the phase segment 1 (phase1 output), the length of the phase segment 2 (phase2 output), the value of the resynchronization jump width (RJW output), and finally the speed indication. These parameters are stored in the paramsREG register. A modification of these parameters can be achieved by means of the params input. This is a 10 bit wide input; the 8 least significant bits carry the data and the 2 most significant bits control what part of the parameter space is changed.

The SyncControl operator takes care of the synchronization and resynchronization. The TQ input is connected to the TQ output of the TQgen operator.

During a 0 on the TQ input, the operator waits. No activity is done during this phase. A 1 on the TQ input indicates a new time quantum. On this event the operator makes sure that a request is put on the bus at the right moment, and that the bus is sampled at the right moment. It should also take care of the resynchronization, but this has not been implemented yet.

The HardSync register is used to distinguish between hard synchronization and resynchronization. The cnt register is used as a time quantum counter. The State register indicates the place within the bit time. The PrevBit register is used for the edge detection.

When a request is initiated (on the request input), this request is stored in the req register. During the next synchronization segment, the requested bit value is put on the out output. To keep the bus value stable during the complete bit time (NRZ coding), the outBit register is used.

At the appropriate moment, the bus is sampled (in input). This value is passed to the indication output. The most significant bit of this output becomes 1 for one clock cycle, indicating the validity of the indication.

6.2.2 The Transmit schematic (Tx)

In figure 52 we can find the content of the Tx schematic of figure 50. The Tx schematic takes care of the transmission of data or remote frames. It also takes care of transmission of the acknowledgement bit. Furthermore, it transmits error and overload frames.
The state connector is connected to the control input of the Control operator. The value on this input controls the function that the operator carries out. The operator can execute sixteen functions. For a description of the operator, see appendix F.

The so is the serial output. It width is two bits where the most significant bit is the valid bit. A dominant bit is represented as 0 and a recessive bit is represented as 1.

The out register is an eight bit register used to shift parts of the frame for transmission. In this way, the serialization is realized. Every time the NextBit input is equal to 1, the next bit is transmitted.

The pi input is the parallel input. Its width is 8 bits. It is used to get parts of the frame that has to be send. The value on this input is valid if the NextByte input is equal to 1. This is an exception to the implementation shown in figure 48. The reason for this is, that the NextByte value is necessary in other places where the pi value is not needed.

As can be seen from figure 50, the bitprocessor has only one CRC unit. This unit is used during transmission as well as during reception of a frame. In case of a transmission of a frame, the CRC unit computes the CRC. The CRC input is used to get the CRC value bit by bit from the CRC unit during the transmission of the CRC field of the frame.

The cBit register is a one bit register. It is used during arbitration to distinguish between the upper part of the arbitration field and its lower part.

### 6.2.3 The Receive schematic (Rx)

In figure 53 we can find the content of the Rx schematic of figure 50.
The Rx schematic takes care of all receive functionality. It is active during reception of a frame as well as during transmission. The state connector is connected to the control input of the Control operator. The value on this input controls the function that the operator carries out. The operator can execute fifteen functions. For a description of the operator, see appendix F.

All the bit counting during the reception or transmission of a frame (data, remote, error, or overload) is done within the Rx schematic. For this, the cnt register is used. This is a 3 bit register. In case of an error or overload, this register is reset to zero. This is done by means of asserting a 1 to the control input of this register.

The si input is the serial input. It is 2 bits wide where the most significant bit is the valid bit. In the least significant bit, the received bit can be found.

The in register is an eight bit wide register. It is used as a shift register during reception of a frame.

When the in register is full, the received bits are passed to the output. They are presented to the outO output. This output is connected to the po (parallel out) connector. The validity of the value on the outO output is indicated by the NextByte output. If the data on the outO output are valid, the NextByte output carries the value 1. Otherwise, it carries the value 0.

The DLC register is used to store the DLC. This value is needed to know the number of data bytes of the frame in case of a data frame. After reception of the DLC, this value is stored into this register. This happens during the reception of the control field. During the data field, this register is used as a down counter. During the arbitration field and the CRC field it is also used to distinguish between the lower and upper part of the field.
The RTR register is used to store the RTR bit. This bit is used to distinguish between a data frame and a remote frame. It is also necessary to know if there is a data field in this frame.

If the NoData output is equal to 1, this indicates that the frame is a data frame with \( \text{DLC}=0 \) or it is a remote frame. This value is asserted after reception of the complete Control field.

The busy signal is asserted when the receiver is within a frame. It indicates that no state transition has to be made. When this signal becomes 0, the state has to be changed.

### 6.2.4 The bit stuffing schematic

In figure 54 we can find the content of the bit stuffing schematic of figure 50.

![Diagram of the BitStuff schematic](image)

**Figure 54.** The content of the BitStuff schematic

The control connector is connected to the control input of the Control operator and to the ctrl input. The value on the control input controls the function that the operator carries out. The operator can execute two functions. For a description of the operator, see appendix F.

During the DoNotStuff phase, the values on the in input are passed to the out output. Nothing else is done.

We will now see what happens during the Stuff phase. The PreviousBit register stores the previous bit. This is done to compare it with the current bit. If these bits are equal, the counter is increased. If these bits are different, the counter is set to 1 (one equal bit counted so far). The cnt register is a bitcounter. It is used to count the number of consecutive bits with equal values. As long as no five consecutive equal bits have been detected, the value from the in input is passed to the out output. When five consecutive bits have been detected and a new bit arrives, no matter what value, the
out output will be provided with the complementary value of the Previous bit store (the stuff bit). On this event, the stuff register is set to 1 to indicate the insertion of the stuff bit. The bit that originally arrived is stored in the PreviousBit register. Now the BitStuff schematic waits until the lowest bit of the ctrl input becomes equal to 1. If this happens, the original bit is send.

6.2.5 The bit destuffing schematic

In figure 55 we can find the content of the bit destuffing schematic of figure 50.
6.2.6 The CRC schematic

In figure 56 we can find the content of the CRC schematic of figure 50.
The control connector is connected to the control input of the Control operator. The value on the control input controls the function that the operator carries out. The operator can execute four functions. For a description of the operator, see appendix F.

During the ResetCRC state, the CRCsequence register is given the value 0. During the ComputeCRC state, the CRC sequence is computed from the in input. During the CompareCRC state, the received CRC is compared to the computed CRC in the CRCsequence register. In case the bit from the in input does not equal the bit from the computed CRC sequence, the error output goes to 1, indicating a CRC error. During the CRCout state, the CRC is put on the CRC output bit by bit. This is done to provide the Tx schematic with the CRC sequence it has to transmit.

6.2.7 The Management schematic

In figure 57 we can find the content of the Management schematic of figure 50.
The Management schematic is divided into three subschematics: Compare, Control, and FCE. The contents of these subschematics are now going to be discussed.

The Compare schematic

In figure 58 we can find the content of the Compare schematic of figure 57.

![Figure 58. The content of the Compare schematic](image)

The description of the Control operator can be found in appendix F. The FirstValue register is used in case that a bit is transmitted. This bit is monitored on the in1 input. When a bit is received (on the in2 input), there are two possibilities: The value in the FirstValue register is valid or not. If the value stored in the FirstValue register is valid, this value is compared to the value seen on the bus. In this case, the out output carries the result of the comparison (i.e. match, no match [recessive bit found in stead of expected dominant], or no match [dominant bit found in stead of expected recessive]). In case there is no first value, the bitprocessor is receiving. When a bit is received, the value of the bit is passed to the out output.

The Control schematic

In figure 59 we can find the content of the Control schematic of figure 57.
A description of the Control operator can be found in appendix F.

The request input is used to initiate a request for transmission of a data or remote frame. When this request is initiated, and the request can not be granted immediately, the reqFF bit is set to remember the request.

The error input reflects if the bitprocessor is operating normally, or if an error or overload has occurred. The mode input indicates the mode the transmitter is in. This can be power up, bus off, error active, or error passive.

The ctrl input is used to get the result from the comparison unit. This indicates the bit that is received, or that a match or no match situation has occurred.

The Tbit register is used to distinguish between transmission of a frame and reception of a frame. If the bitprocessor is transmitting a frame, this bit is set to 1. Otherwise, it carries the value 0.

The NextBit output is asserted to indicate that the next bit can be sent. It is delayed because it is needed one clock cycle later than generated.

If the busy input carries a 1, the Rx unit is busy receiving a frame field. When this bit becomes zero, a new state has to be entered.
The NoData input is asserted if the frame does not have a data field. When no data field is expected, the state after the Control state will be CRC instead of Data.

The FrameState register represents the state within the frame. The Rxstate output carries the same value as the FrameState. This is because the bitprocessor is always receiving. The Txstate output carries the same value as the FrameState during the transmission of a frame. During reception of a frame, this output only indicates the transmission of the acknowledgement during the ACK slot. The CRCctrl output controls the CRC unit.

The BitStuff unit is controlled by the Sctrl output. The width of this output is 2. The most significant bit indicates whether or not the bitstream has to be stuffed. The least significant bit indicates the transmission of the original bit after insertion of a stuff bit (refer to section 6.2.4). This signal is delayed (delay2) because it is needed one clock cycle later than generated. The status of the BitStuff unit is reflected on the Sstatus input. A 1 on this input indicates that a stuff bit has been inserted.

The BitDestuff unit is controlled by the DSctrl output. A 1 on this output indicates that stuff bits have to removed from the incoming bitstream. A detection of a stuff bit is indicated by a 1 on the Sstatus input.

The FCE schematic

In figure 60 we can find the content of the FCE schematic of figure 57.
Modeling and implementation of a CAN controller

The cnt1 register is used as a counter. It is 8 bits wide. If the mode is bus off or power up, it counts the number of successive recessive bits. During the error active or error passive phase, it is used as Transmit Error Counter. The implementation of the TEC is not ready yet.

The cnt2 register is used as a counter. It is 8 bits wide. During power up mode, it is not used. During bus off mode it should be used to count the sequences of 11 successive recessive bits. This is not implemented yet. In case the mode is error active or error passive, this counter should be used as Receive Error Counter. This also is not implemented yet.

The transmission of an error flag in case of a CRC error is delayed until the bit following the ACK delimiter. The WaitCRCerror register is used to remember the fact that a CRC error has occurred.

The NoData input of the Management unit (see figure 57) is connected to the Wait input. This signal is used during the superposition phase to indicate the absence of a recessive bit.

The ErrorMode register is used to indicate the error mode (error active, error passive, bus off, or power up).

Figure 60. The content of the FCE schematic
The CRCerror input is 1 in case of a CRC error. The DSerror input is 1 in case of a stuff error.

The state input carries the state the bitprocessor is in. The T input reflects the value of the Tbit register from the Control unit.

The ctrl output carries the information of the comp input. The comp input reflects the result of the comparison form the value that is put on the bus with the value that is monitored on the bus. This result is translated to the proper form for the ctrl output.

The error output indicates if an error or overload or arbitration loss has occurred. The errorO output is used to reset the bitcounter of the Rx unit. It is asserted in case of an error or overload.

After a discussion with my coach it became clear to me that the transmit and receive error counter should be transfered to the byte processor. When a CAN node is operating normally, the error counters will not need to be updated often. So the updating of the error counters would not introduce a lot of overhead for the byteprocessor. By transfering them to the byteprocessor, an enormous hardware reduction is achieved. So in the future design, these counters should be transfered to the byteprocessor.

6.2.8 The BackEnd schematic

In figure 61 we can find the content of the BackEnd schematic of figure 50.

![BackEnd schematic diagram]

Figure 61. The content of the BackEnd schematic

The description of the BEcontrol operator can be found in appendix F.
At this moment, the BEcontrol operator has no functionality. It is used to provide the interface with the byteprocessor. This interface could be build around the three connectors Data, Control, and Status. The data1 and data2 registers are 8 bit registers and they could be used as temporary storage.

The data to be transmitted are passed to the po output. Assertion of the NextByteO signal indicates the validity of the po output. The indication that there is data ready to send a frame is done by asserting the request output.

The data that has been received is found on the pi input. The validity of this input is indicated by the nextByteI input.

The params output is used to pass the timing parameters to the PLS unit.

After a discussion with my coach, I will make some remarks regarding the actual interface to the byteprocessor. First of all there will be two separate busses for input of data and output of data. These busses will both be 8 bits wide. There will have to be an event bus that is used to interrupt the byteprocessor on certain events. The bitc bus of the SDLC controller is used to transfer static setup data. An example of static setup data is the parameters of the PLS schematic.
7 Conclusions

In section 7.1, some recommendations on future expansion will be given. Section 7.2 gives the final conclusions.

7.1 Recommendations on future work

First I will give some general remarks on what has to be done in the future. A thorough check on implementation details has to be carried out. The CAN protocol is filled with little exceptions. As far as I know, I have implemented them all except the things mentioned below. Also an exhaustive testing has to be submitted to the complete bitprocessor.

Now, a contemplation is given on the adjustments of the different units of the implementation:

As far as I can see, the Tx unit does not need major adjustments. The same is true for the Rx unit. The BitStuff, BitDestuff, and CRC units are also working properly. Maybe some small extensions might be needed in the future, but I can not think of any at this moment.

The Compare unit provides all information needed, so it will probably not be needing any adjustments. The Control unit is almost complete. There has to be added something for the interface with the BackEnd unit. Also the transition to the suspend transmission state has to be adjusted. At this moment, the interframe space of error passive nodes which have been receiver of the previous frame is depicted in figure 9, but it should be as depicted in figure 8. The FCE unit needs several adjustments. In case a receiving node that is operating in the error passive mode wants to send an error frame, it has to wait for a sequence of six consecutive recessive bits until it has ended the error flag. This is not yet implemented. The main adjustment in the FCE unit will be moving the error counters to the byteprocessor.

The BackEnd functionality has to be completely defined. A recommendation on how this can be done is given. The PLS unit functions properly with the exception of the phase error correction.

7.2 Final conclusions

The model of CAN is completed. Some state transition diagrams are very large. That is why they were omitted in this report. They should be subdivided to make the model more clear.

The Tx, Rx, BitStuff, BitDestuff, CRC, and Compare units do not need any major adjustments. They should be submitted to an exhaustive test though. The Control unit needs some minor adjustments and the FCE unit needs several adjustments. The
BackEnd unit is not implemented. It does not contain much functionality, so the implementation of this unit should not be a big problem. The PLS unit works properly with the exception of the error correction.

The bitprocessor can transmit and receive frames. It is also able to initiate error and overload frames. Therefore it should not be too big a problem to complete the design. The design is kept small. Wherever possible, registers are shared. After completion of the design, one might be able to reduce even more registers. Just finding out if there are registers that can be used concurrently should suffice to make this reduction. The number of interconnections between the several blocks is kept as small as possible. By not making use of IDaSS state controllers, we have a greater insight in the amount of busses needed.
Literature

Computer networks, 2nd ed.
Englewood Cliffs: Prentice-Hall, 1988

Structured Development for Real-time Systems
Vol. 1: Introduction & Tools
Vol. 2: Essential Modeling Techniques
Vol. 3: Implementation Modeling Techniques
London: Yourdon, 1985-1986

[3] Road vehicles - Interchange of digital information - Controller area network (CAN) for high speed communication
ISO/DIS 11898
ISO, 1992


A Single World Standard - Forget It!
Assembly Automation, Vol. 14 No. 1, 1994, p. 3

Auto's Carry the CAN - Controller Area Network (CAN) for Industrial Applications

The Evolution of Control Architectures for Automated Manufacturing Systems

[8] Smith, R.G.
The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver

TIDE, 16 April 1993

M3S - a standard communication architecture for rehabilitation applications
IDaSS for ULSI - Interactive Design and Simulation System for Ultra Large Scale Integration
Eindhoven University of Technology, 1990
Appendix A - CAN service primitives and messages

In this appendix the primitives and messages defined in the CAN standard are presented.

Primitives and messages between user and LLC

Messages sent from LLC user to LLC sublayer

Reset_Request:
  Request to set the node into an initial state

Messages sent from LLC sublayer to LLC user

Reset_Response:
  Response to the Reset_Request

Node_Status:
  Indicates the current status of the node. (I.e. it signals whether or not the node is in the bus off state)

Unacknowledged data transfer service primitives

L_DATA.request:
  This primitive provides a request to send data. It is provided with the parameters IDENTIFIER (identifies the data and its priority), DLC (Data Length Code), and DATA (data the user wants to transmit). The effect on receipt of this primitive causes the LLC sublayer to initiate the transfer of a LLC data frame by use of the data transfer service provided by the MAC sublayer.

L_DATA.indication:
  This primitive indicates the arrival of data. It is provided with the parameters IDENTIFIER (identifies the data and its priority), DLC (Data Length Code), and DATA (data the user wants to transmit). The effect on receipt of this primitive by the LLC user is unspecified.

L_DATA.confirm:
  This primitive provides a confirmation of the L_DATA.request primitive. It is provided with the parameter TRANSFER_STATUS. The TRANSFER_STATUS can have two values: COMPLETE or NOT_COMPLETE. The effect on receipt of this primitive by the LLC user is unspecified.
Unacknowledged remote data request service primitives

L_REMOTE.request:
This primitive provides a request to get data from a remote entity. It is provided with the parameters IDENTIFIER (identifies the data and its priority) and DLC (Data Length Code). Receipt of this primitive causes the LLC sublayer to initiate the transfer of a data unit by use of the remote date transfer service provided by the MAC sublayer.

L_REMOTE.indication:
This primitive indicates the arrival of a request for data from a remote entity. It is provided with the parameters IDENTIFIER (identifies the data and its priority) and DLC (Data Length Code). The effect on receipt of this primitive by the LLC user is unspecified.

L_REMOTE.confirm:
This primitive provides a confirmation of the L_REMOTE.request primitive. It is provided with the parameter TRANSFER_STATUS. The TRANSFER_STATUS can have two values: COMPLETE or NOT_COMPLETE. The effect on receipt of this primitive by the LLC user is unspecified.

Primitives between LLC and MAC

Acknowledged data transfer service primitives

MA_DATA.request:
This primitive provides a request to send data. It is provided with the parameters IDENTIFIER (identifies the data and its priority), DLC (Data Length Code), and DATA (data the user wants to transmit). Receipt of this primitive causes the MAC sublayer to prepare a protocol data unit by adding all MAC specific control information (SOF, RTR bit, reserved bits, CRC, recessive bit during ACK slot, EOF) to the data unit coming from the LLC sublayer. The MAC data unit will be serialized and passed bit by bit to the physical layer for transfer to the peer MAC sublayer entity or entities.

MA_DATA.indication:
This primitive indicates the arrival of data from a remote entity. It is provided with the parameters IDENTIFIER (identifies the data and its priority), DLC (Data Length Code), and DATA (data the user wants to transmit). The effect on receipt of this primitive by the LLC sublayer is unspecified.

MA_DATA.confirm:
This primitive provides a confirmation of the MA_DATA.request. It is provided with the parameter TRANSMISSION_STATUS. This parameter can have one of two different values: SUCCESS or NO_SUCCESS. The effect on receipt of this primitive by the LLC sublayer is unspecified.
Acknowledged remote data transfer service primitives

MA_REMOTE.request:
This primitive provides a request to get data from a remote entity. It is provided with the parameters IDENTIFIER (identifies the data and its priority) and DLC (Data Length Code). Receipt of this primitive causes the MAC sublayer to prepare a protocol data unit by adding all MAC specific control information (SOF, RTR bit, reserved bits, CRC, recessive bit during ACK slot, EOF) to the data unit coming from the LLC sublayer. The MAC data unit will be serialized and passed bit by bit to the physical layer for transfer to the peer MAC sublayer entity or entities.

MA_REMOTE.indication:
This primitive indicates the arrival of a request for data from a remote entity. It is provided with the parameters IDENTIFIER (identifies the data and its priority) and DLC (Data Length Code). The effect on receipt of this primitive by the LLC sublayer is unspecified.

MA_REMOTE.confirm:
This primitive provides a confirmation of the MA_REMOTE.request. It is provided with the parameter TRANSMISSION_STATUS. This parameter can have one of two different values: SUCCESS or NO_SUCCESS. The effect on receipt of this primitive by the LLC sublayer is unspecified.

Overload frame transfer service primitives

MA_OVLD.request:
This primitive provides a request to send an overload frame. It is not provided with any parameter. Receipt of this primitive causes the MAC sublayer to form an overload frame. The overload frame will be passed to the lower layers for transfer to the peer MAC sublayer entities.

MA_OVLD.indication:
This primitive indicates that an overload frame has been received. The effect on receipt of this primitive by the LLC sublayer is unspecified.

MA_OVLD.confirm:
This primitive provides a confirmation of the MA_OVLD.request primitive. It is provided with the parameter TRANSMISSION_STATUS. This parameter can have one of two different values: SUCCESS or NO_SUCCESS. The effect on receipt of this primitive by the LLC sublayer is unspecified.

Primitives between MAC and PLS

PLS_DATA.request:
Request for a dominant or recessive bit. This primitive provides one parameter: OUTPUT_UNIT. OUTPUT_UNIT can have one of the following two values: DOMINANT or RECESSIVE.

PLS_DATA.indication:
This is an indication of the arrival of a dominant or recessive bit. This primitive provides one parameter: INPUT_UNIT. INPUT_UNIT can have one of the following two values: DOMINANT or RECESSIVE.

Messages between PLS and PMA

Messages sent from PLS sublayer to PMA sublayer

Output:
The PLS sublayer sends an output message to the PMA sublayer whenever it receives an OUTPUT_UNIT from the MAC sublayer. The output message causes the PMA to send a dominant or recessive bit.

Bus_off:
The PLS sublayer sends a bus_off message to the PMA sublayer whenever it receives a Bus_Off_Request from the supervisor.

Bus_off_release:
The PLS sublayer sends a bus_off_release message to the PMA sublayer whenever it receives a bus_off_release_request from the supervisor.

Messages sent from PMA sublayer to PLS sublayer

Input:
The PMA sublayer sends an input message to the PLS sublayer whenever the MAU has received a bit from the medium. The input signal indicates the arrival of a dominant or recessive bit to the PLS.

Messages between LLC and FCE

Messages sent from LLC sublayer to FCE sublayer

Normal_Mode_Request:
Resets the FCE to initial state, thereby resetting the values of the transmit and receive error counters to zero.

Messages sent from FCE sublayer to LLC sublayer

Normal_Mode_Response:
Response to the Normal_Mode_Request

Bus_Off:
Indicates that the node is in the bus off state.
Messages between MAC and FCE

Messages sent from MAC sublayer to FCE sublayer

Transmit/Receive:
   Indicates the node's current transfer mode.

Error:
   Indicates that the MAC sublayer has detected an error (bit error, stuff error, CRC error, form error, acknowledgement error).

Primary_Error:
   Signals that the MAC sublayer has detected a dominant bit after sending an error flag (indicates that the MAC sublayer has detected a primary error and not an error that is caused by the error flag of another node).

Error/overload flag:
   Indicates that the MAC sublayer is sending an error flag or an overload flag.

Counters_unchanged:
   Indicates that the FCE counters remain unchanged (due to special cases).

Error_delimiter-too-late:
   Indicates that the MAC sublayer is waiting too long for the error delimiter. This signal is set each time after a sequence of eight consecutive dominant bits have been received after sending an error flag.

Successful_transfer:
   Indicates that transmission or reception was successfully completed.

Error_passive_response:
   Indicates that the node was set into the error passive state.

Error_active_response:
   Indicates that the node was set into the error active state.

Messages sent from FCE sublayer to MAC sublayer

Error_passive_request:
   Request to set the node into the error passive state.

Error_active_request:
   Request to set the node into the error active state.

Messages between physical layer and FCE

Messages sent from physical layer to FCE sublayer

Bus_off_response:
   Response to the bus_off_request

Bus_off_release_response:
   Response to the bus_off_release_request.
Messages sent from FCE sublayer to physical layer

**Bus_off_request:**
- Request to switch off the node from the bus.

**Bus_off_release_request:**
- Request to set the node into the normal transmit and receive mode.
Appendix B - Data Dictionary of the first iteration

Here, the data dictionary of the first iteration will be presented. It is divided into the sections terminators, transformations, discrete flows, continuous flows, and event flows.

Terminators

USER
This terminator symbolizes the connection to the user of the CAN protocol.

PMA
This terminator symbolizes the connection to the PMA (Physical Medium Attachment). The PMA is connected to the MDI (Medium Dependent Interface) and the MDI is connected to the medium.

Data transformations

CAN\{PMA,MDI\}
This transformation reflects the CAN functionality as described in the CAN specifications: Road vehicles - Interchange of digital information - Controller area network (CAN) for high-speed communication; ISO/DIS 11898. Nevertheless, the PMA and MDI parts are not incorporated in this transformation.

Discrete flows

L_DATA.confirm
This primitive provides a confirmation of the L_DATA.request primitive. It is provided with the parameter TRANSFER_STATUS. The TRANSFER_STATUS can have two values: COMPLETE or NOT_COMPLETE.

L_DATA.indication
This primitive indicates the arrival of data. It is provided with the parameters IDENTIFIER (identifies the data and its priority), DLC (Data Length Code), and DATA (data the user wants to transmit).

L_DATA.request
This primitive provides a request to send data. It is provided with the parameters IDENTIFIER (identifies the data and its priority), DLC (Data Length Code), and DATA (data the user wants to transmit).
L_REMOTE.confirm
This primitive provides a confirmation of the L_REMOTE.request primitive. It is provided with the parameter TRANSFER_STATUS. The TRANSFER_STATUS can have two values: COMPLETE or NOT_COMPLETE.

L_REMOTE.indication
This primitive indicates the arrival of a request for data from a remote entity. It is provided with the parameters IDENTIFIER (identifies the data and its priority) and DLC (Data Length Code).

L_REMOTE.request
This primitive provides a request to get data from a remote entity. It is provided with the parameters IDENTIFIER (identifies the data and its priority) and DLC (Data Length Code).

Node_Status
Indicates the current status of the node (i.e. it signals whether or not the node is in the bus off state).

Continuous flows
input
The input message represents the current value of the CAN bus.

output
The output message is the value that has to be put on the CAN bus (when the node is transmitting).

Event flows
Bus_off
The Bus_off message tells the Physical Medium Attachment to transfer to the bus off state.

Bus_off_release
The Bus_off_release message tells the Physical Medium Attachment to transfer to the bus on state.

Reset_Request
Request to set the node into an initial state.

Reset_Response
Response to the Reset_Request.
Appendix C - Data Dictionary of the second iteration

Here, the data dictionary of the second iteration will be presented. It is divided into the sections terminators, transformations, discrete flows, continuous flows, and event flows.

Terminators

**USER**
This terminator symbolizes the connection to the user of the CAN protocol.

**PMA**
This terminator symbolizes the connection to the PMA (Physical Medium Attachment). The PMA is connected to the MDI (Medium Dependent Interface) and the MDI is connected to the medium.

Data transformations

**buffer requests**
This transformation buffers the requests for a possible retransmission.

**CAN\{PMA,MDI\}**
This transformation reflects the CAN functionality as described in the CAN specifications: Road vehicles - Interchange of digital information - Controller area network (CAN) for high-speed communication; ISO/DIS 11898. Nevertheless, the PMA and MDI parts are not incorporated in this transformation.

**Control**
This transformation controls the MAC. It provides the FCE with the necessary information for the fault confinement.

**Determine Node_Status**
This transformation generates the Node_status.

**FCE**
The Fault Confinement Entity takes care of the error management. It inhabits the error counters. The rules of confinement can be found in section 2.5.

**frame acceptance filtering**
This transformation filters the incoming frames. At this place the decision is made whether an incoming frame is passed to the M3S extension or not. This decision is made according to the filter parameters. It also generates an overload request in case it is not able to filter the messages quickly enough.
LLC
The Logic Link Control has three tasks:
• Frame acceptance filtering
• Overload notification
• Recovery management.

MAC
The Medium Access Control sublayer is divided into two fully independently operating parts: the transmitter part and the receiver part. The functions of these two parts are described below:

Transmission part:
Transmit data encapsulation:
• Acceptance of LLC frames and interface control information
• CRC sequence calculation
• Construction of MAC frame by adding SOF, RTR bit, reserve bits, CRC, ACK, and EOF to the LLC frame
Transmit media access management:
• Initiation of the transmission process after recognizing bus idle (compliance with interframe space)
• Serialization of the MAC frame
• Insertion of stuffbits (bit stuffing)
• Arbitration and passing into receive mode in case of loss of arbitration
• Error detection (monitoring, format check)
• Acknowledgement check
• Recognition of an overload condition
• Overload frame construction and initiation of transmission
• Error frame construction and initiation of transmission
• Presentation of a serial bit stream to the physical layer for transmission

Receive part:
Receive media access management:
• Reception of a serial bit stream from the physical layer
• Deserialization and recompiling of the frame structure
• Deletion of stuffbits (bit destuffing)
• Error detection (CRC, format check, stuff rule check, monitoring)
• Transmission of acknowledgement
• Error frame construction and initiation of transmission
• Recognition of an overload condition
• Reactive overload frame construction and initiation of transmission
Receive data decapsulation:
• Removing the MAC specific information from the received frame
• Presenting the LLC frame and interface control information to the LLC sublayer.
The Physical signaling (PLS) transformation encompasses those functions related to bit representation, timing, and synchronization.

Receive Date Decapsulation
The tasks of this transformation are:
• Removing the MAC specific information from the received frame
• Presenting the LLC frame and interface control information to the LLC sublayer.

Receive Media Access Management
The tasks of this transformation are:
• Reception of a serial bit stream from the physical layer
• Deserialization and recompiling of the frame structure
• Deletion of stuffbits (bit destuffing)
• Error detection (CRC, format check, stuff rule check, monitoring)
• Transmission of acknowledgement
• Error frame construction and initiation of transmission
• Recognition of an overload condition
• Reactive overload frame construction and initiation of transmission

Reset
This transformation resets the LLC.

Transmit Data Encapsulation
The tasks of this transformation are:
• Acceptance of LLC frames and interface control information
• CRC sequence calculation
• Construction of MAC frame by adding SOF, RTR bit, reserve bits, CRC, ACK, and EOF to the LLC frame

Transmit Media Access Management
The task of this transformation are:
• Initiation of the transmission process after recognizing bus idle (compliance with interframe space)
• Serialization of the MAC frame
• Insertion of stuffbits (bit stuffing)
• Arbitration and passing into receive mode in case of loss of arbitration
• Error detection (monitoring, format check)
• Acknowledgement check
• Recognition of an overload condition
• Overload frame construction and initiation of transmission
• Error frame construction and initiation of transmission
• Presentation of a serial bit stream to the physical layer for transmission

Discrete flows

Error_Request
The Error_Request primitive is used to send error messages to from the FCE to the MAC.

Error_Response
The Error_Response primitive is a response to the Error_Request primitive.

Filter_Params
The filter parameters consist of one or more parameters that have to be added to or deleted from the parameter filter memory.

L_DRA.confirm
This primitive is a composition of three primitives: L_DATA.confirm, L_REMOTE.confirm, and L_ABORT.confirm. These primitives provide a confirmation of the L_DATA.request, L_REMOTE.request, and L_ABORT.request primitives, respectively. They are provided with the parameter TRANSFER_STATUS. In case of the DATA and REMOTE primitives, the TRANSFER_STATUS can have two values: COMPLETE or NOT_COMPLETE. In case of the ABORT primitive, the TRANSFER_STATUS can also have two values: COMPLETE or ABORTED.

L_DR.indication
This primitive is a composition of two primitives: L_DATA.indication and L_REMOTE.indication. These primitives indicate the arrival of data or the arrival of a request for data from a remote entity. The L_DATA.indication and L_REMOTE.indication primitives are both provided with the parameters IDENTIFIER (identifies the data and its priority) and DLC (Data Length Code). The L_DATA.indication primitive is additionally provided with the parameter DATA (data the user wants to transmit).

L_DRA.request
This primitive is a composition of three primitives: L_DATA.request, L_REMOTE.request, and L_ABORT.request. These primitives provide a request to send data, to get data from a remote entity, or to abort retransmission of a previously initiated data or remote request, respectively. The L_DATA.request and L_REMOTE.request primitives are both provided with the parameters IDENTIFIER (identifies the data and its priority) and DLC (Data Length Code). The L_DATA.request primitive is additionally provided with the parameter DATA (data the user wants to transmit). The L_ABORT.request primitive is not provided with any parameter.

MA_DRO.confirm
This primitive is a composition of three primitives: MA_DATA.confirm, MA_REMOTE.confirm, and MA_OVLD.confirm. These primitives provide a confirmation of the MA_DATA.request, MA_REMOTE.request, and MA_OVLD.request primitives, respectively. They are provided with the parameter TRANSMISSION_STATUS. This parameter can have one of two different values: SUCCESS or NO_SUCCESS.
MA_DRO.request
This primitive is a composition of three primitives: MA_DATA.request, MA_REMOTE.request, and MA_OVLD.request. These primitives provide a request to send data, to get data from a remote entity, or to send an overload frame, respectively. The MA_DATA.request and the MA_REMOTE.request primitives are both provided with the parameters IDENTIFIER (identifies the data and its priority) and DLC (Data Length Code). The MA_DATA.request primitive is additionally provided with the parameter DATA (data the user wants to transmit). The MA_OVLD.request primitive is not provided with any parameter.

MA_DR.confirm
This primitive is a composition of two primitives: MA_DATA.confirm, and MA_REMOTE.confirm. These primitives provide a confirmation of the MA_DATA.request, and MA_REMOTE.request, respectively. They are provided with the parameter TRANSMISSION_STATUS. This parameter can have one of two different values: SUCCESS or NO_SUCCESS.

MA_DR.indication
This primitive is a composition of two primitives: MA_DATA.indication and MA_REMOTE.indication. These primitives indicate the arrival of data or the arrival of a request for data from a remote entity. The MA_DATA.indication and MA_REMOTE.indication primitives are both provided with the parameters IDENTIFIER (identifies the data and its priority) and DLC (Data Length Code). The MA_DATA.indication primitive is additionally provided with the parameter DATA (data the user wants to transmit).

MA_DR.request
This primitive is a composition of two primitives: MA_DATA.request, and MA_REMOTE.request. These primitives provide a request to send data, and to get data from a remote entity, respectively. The MA_DATA.request and the MA_REMOTE.request primitives are both provided with the parameters IDENTIFIER (identifies the data and its priority) and DLC (Data Length Code). The MA_DATA.request primitive is additionally provided with the parameter DATA (data the user wants to transmit).

MA_DR.request_TDE
This primitive is a composition of two primitives: MA_DATA.request, and MA_REMOTE.request. These primitives provide a request to send data, and to get data from a remote entity, respectively. The MA_DATA.request and the MA_REMOTE.request primitives are both provided with the parameters IDENTIFIER (identifies the data and its priority) and DLC (Data Length Code). The MA_DATA.request primitive is additionally provided with the parameter DATA (data the user wants to transmit).

MA_OVLD.confirm
This primitive provides a confirmation of the MA_OVLD.request primitive. It is provided with the parameter TRANSMISSION_STATUS. This parameter can have one of two different values: SUCCESS or NO_SUCCESS.
MA_OVLD.request
This primitive provides a request to send an overload frame. It is not provided with any parameter.

MA_OVLD.request_TMAM
This primitive provides a request to send an overload frame. It is not provided with any parameter.

Node_Status
Indicates the current status of the node (i.e. it signals whether or not the node is in the bus off state).

PLS_DATA.indication
This is an indication of the arrival of a dominant or recessive bit. This primitive provides one parameter: INPUT_UNIT. INPUT_UNIT can have one of the following two values: DOMINANT or RECESSIVE.

PLS_DATA.indication_RMAM
This is an indication of the arrival of a dominant or recessive bit. This primitive provides one parameter: INPUT_UNIT. INPUT_UNIT can have one of the following two values: DOMINANT or RECESSIVE.

PLS_DATA.indication_TMAM
This is an indication of the arrival of a dominant or recessive bit. This primitive provides one parameter: INPUT_UNIT. INPUT_UNIT can have one of the following two values: DOMINANT or RECESSIVE.

PLS_DATA.request
Request for a dominant or recessive bit. This primitive provides one parameter: OUTPUT_UNIT. OUTPUT_UNIT can have one of the following two values: DOMINANT or RECESSIVE.

PLS_DATA.request_RMAM
Request for a dominant or recessive bit. This primitive provides one parameter: OUTPUT_UNIT. OUTPUT_UNIT can have one of the following two values: DOMINANT or RECESSIVE.

PLS_DATA.request_TMAM
Request for a dominant or recessive bit. This primitive provides one parameter: OUTPUT_UNIT. OUTPUT_UNIT can have one of the following two values: DOMINANT or RECESSIVE.

Receive MAC frame
This flow indicates a MAC frame.

Rmode
This flow indicates a mode change to the receiver.
Rstatus
This flow provides the status of the receiver to the Control transformation.

Status
This primitive is used to reveal the status of the MAC to the FCE.

Tmode
This flow indicates a mode change to the transmitter.

Transmit MAC frame
This flow indicates a MAC frame.

Tstatus
This flow provides the status of the transmitter to the Control transformation.

Continuous flows

input
The input message represents the current value of the CAN bus.

output
The output message is the value that has to be put on the CAN bus (when the node is transmitting).

Event flows

Bus Off
The Bus Off message indicates to the LLC that the node is set to bus off state. The LLC needs this information to produce the Node_Status.

Bus On
The Bus On message indicates to the LLC that the node is set to bus on state. The LLC needs this information to produce the Node_Status.

Bus_off
The Bus_off message tells the Physical Medium Attachement to transfer to the bus off state.

Bus_Off_Request
Request to switch off the node from the bus.

Bus_Off_Response
Response to Bus_Off_Request

Bus_off_release
The Bus_off_release message tells the Physical Medium Attachement to transfer to the bus on state.
Bus_Off_Release_Request
   Request to set the node into the normal transmit/receive mode.

Bus_Off_Release_Response
   Response to Bus_Off_Release_Request

Normal_Mode_Request
   Resets FCE to initial state.

Normal_Mode_Response
   Response to Normal_Mode_Request.

Reset LLC
   The internal reset signal

Reset LLC_br
   The internal reset signal

Reset LLC_dns
   The internal reset signal

Reset LLC_faf
   The internal reset signal

Reset_Request
   Request to set the node into an initial state.

Reset_Response
   Response to the Reset_Request.
Appendix D - Data Dictionary of the third iteration

Here, the data dictionary of the third iteration will be presented. It is divided into the sections terminators, transformations, discrete flows, continuous flows, event flows, data stores, and control stores.

**Terminators**

**MAU & medium**

This terminator symbols the connection to the PMA (Physical Medium Attachment). The PMA is connected to the MDI (Medium Dependent Interface) and the MDI is connected to the medium.

**M3S extension & application**

This terminator symbols a connection to the data part of the M3S extension

**M3S management**

This terminator reflects the connection to the M3S management. The M3S management takes care of all the 'management' tasks as there are parameter changes and reset handling. It also receives the node status from the CAN node.

**Data transformations**

**Add Rerror frame**

This transformation adds the appropriate error frame to the Rerror frame store.

**Add Request**

This transformation adds a request to the REQUEST store.

**add Rprevious**

This transformation adds the Rbit to the Rprevious store.

**add Tprevious**

This transformation adds the Tbit to the Tprevious store.

**bit stuffing**

This transformation takes care of the insertion of a stuff bit when necessary.

**bit destuffing**

This transformation takes care of the deletion of a stuff bit when necessary.

**br_control**
This transformation is the buffer requests control. It controls the buffering of the request. When necessary, it initiates a retransmission of the request. After successful transmission it removes the request from the buffer.

**BUFFER REQUESTS**

This transformation buffers the requests for a possible retransmission.

**CAN**

This transformation reflects the CAN functionality as described in the CAN specifications: Road vehicles - Interchange of digital information - Controller area network (CAN) for high-speed communication; ISO/DIS 11898. Nevertheless, the MDI part is not incorporated in this transformation, because according to the CAN specifications, it merely encompasses the mechanical and electrical interface between the physical medium and the higher layers.

**change filter params**

This transformation takes care of storing changes in filter parameters to the filter params list. On reset, it flushes the filter params list.

**Compare and update Rcounter**

This transformation compares the incoming bit with the previous bit (stored in Rprevious). If a match occurs, the Rcounter is increased by one. If the bits do not match, the Rcounter is set to zero.

**compare and update Tcounter**

This transformation compares the incoming bit with the previous bit (stored in Tprevious). If a match occurs, the Tcounter is increased by one. If the bits do not match, the Tcounter is set to zero.

**compare ID**

This transformation compares the identifier of the incoming frame with the identifiers stored in the filter params list. When the identifier of the incoming frame matches one of the identifiers in the filter params list, the frame is passed to the M3S extension. When no identifier from the filter params list matches the incoming frame's identifier, the frame is rejected. In both cases the ready signal is triggered.

**Construct LLC frame**

This transformation constructs an LLC frame from the components ID, DLC, and DATA. Depending on whether the RTR bit is set or not, it creates a MA_REMOTE.indication or a MA_DATA.indication, respectively.

**construct polynomial**

This transformation constructs the polynomial that takes part in the CRC calculation.

**CRC calculation**

This transformation performs the calculation of the CRC sequence.
Decapsulate MAC frame
This transformation decapsulates the received MAC frame into the components ID, DLC, DATA, and RTR.

deserialize frame
This transformation takes care of the deserialization of the received frame.

detect Tstuff bit
This transformation compares the Tcounter with 5. If there is a match, the Tcounter is set to zero, and the Tstuff bit_dsb event is triggered.

DFcontrol
This transformation keeps track of the frame reception. It also initiates an overload or error frame, and hands out the status it is in.

divide modulo 2
This transformation divides the polynomial T and the generator T modulo 2. The result is the desired CRC value.

edge detection
When this transformation detects an edge it triggers the edge event. If the bit rate is fast, it only reacts on recessive to dominant edges. If the bit rate is slow, it reacts on both edges.

FRAME ACCEPTANCE FILTERING
This transformation filters the incoming frames. At this place the decision is made whether an incoming frame is passed to the M3S extension or not. This decision is made according to the filter parameters. It also generates an overload request in case it is not able to filter the messages quick enough.

generate MAC frame
This transformation composes the MAC frame for transmission.

LLC
The Logic Link Control has three tasks:
• Frame acceptance filtering
• Overload notification
• Recovery management.

MAC
The Medium Access Control sublayer is divided into two fully independently operating parts: the transmitter part and the receiver part. The functions of these two parts are described below:

Transmission part:
  Transmit data encapsulation:
  • Acceptance of LLC frames and interface control information
  • CRC sequence calculation
Modeling and implementation of a CAN controller

- Construction of MAC frame by adding SOF, RTR bit, reserve bits, CRC, ACK, and EOF to the LLC frame
- Transmit media access management:
  - Initiation of the transmission process after recognizing bus idle (compliance with interframe space)
  - Serialization of the MAC frame
  - Insertion of stuffbits (bit stuffing)
  - Arbitration and passing into receive mode in case of loss of arbitration
  - Error detection (monitoring, format check)
  - Acknowledgement check
  - Recognition of an overload condition
  - Overload frame construction and initiation of transmission
  - Error frame construction and initiation of transmission
  - Presentation of a serial bit stream to the physical layer for transmission

Receive part:
- Receive media access management:
  - Reception of a serial bit stream from the physical layer
  - Deserialization and recompiling of the frame structure
  - Deletion of stuffbits (bit destuffing)
  - Error detection (CRC, format check, stuff rule check, monitoring)
  - Transmission of acknowledgement
  - Error frame construction and initiation of transmission
  - Recognition of an overload condition
  - Reactive overload frame construction and initiation of transmission
- Receive data decapsulation:
  - Removing the MAC specific information from the received frame
  - Presenting the LLC frame and interface control information to the LLC sublayer.

MAC MANAGEMENT
This transformation takes care of the management functions of the MAC sublayer. These are: mode setting (error active, error passive, or bus off, receiving, transmitting); generate bus status.

MANAGEMENT
The management transformation handles the reset and reacts on a change in bus status, which is reflected in the Node_Status.

manage Tbuffer
This transformation puts a new frame, an error frame, or an overload frame into the Tbuffer.

next bit from Tbuffer
When next Tbit is triggered, the next bit from the Tbuffer is passed to transmit bit.
next Rbit to Rbuffer
The next bit from destuffed bit_nbtb is put in the right (is next) place in the
Rbuffer.

overload control
This transformation submits a MA_OVLD.request to the MAC sublayer on a
queue full event. If no not queue full event is encountered when the overload
confirmation is received, a second overload is requested.

phase error detection
This transformation detects the phase error.

PLS
The Physical signaling (PLS) transformation encompasses those functions
related to bit representation, timing, and synchronization.

queue control
This transformation takes care of the queueing of one data or remote frame. It
also indicates if an overload request has to be submitted.

Rbounce crossing message generation
This transformation takes care of the generation of the Rbounce crossing
message.

Rcompare
This transformation compares the incoming bit with the bit that has been
send.

RCV
This transformation extracts the indication of a received bit from the
analogous input. It samples the input at the correct place within the bit time.

RECEIVE
This transformation handles the functionality of the receive process.

Receive Date Decapsulation
The tasks of this transformation are:
• Removing the MAC specific information from the received frame
• Presenting the LLC frame and interface control information to the LLC
  sublayer.

RECEIVE ERROR COUNTER
This transformation inhabits the error counter of the receiver. It updates the
counter when necessary. It also detects boundary crossings of the counter.

Receive Media Access Management
The tasks of this transformation are:
• Reception of a serial bit stream from the physical layer
• Deserialization and recompiling of the frame structure
• Deletion of stuffbits (bit destuffing)
• Error detection (CRC, format check, stuff rule check, monitoring)
• Transmission of acknowledgement
• Error frame construction and initiation of transmission
• Recognition of an overload condition
• Reactive overload frame construction and initiation of transmission

Remove Request
This transformation removes a request from the REQUEST store.

Retrieve Request
This transformation retrieves a request from the REQUEST store for retransmission.

re_control
This transformation takes care of the control of the receive error counter.

RMAMcontrol
This transformation takes care of the control of all receive functionalities. It also keeps track of the status of the node.

Rsend
This transformation takes care of the sending of an error or overload frame.

Rsend_db
This transformation takes care of sending the destuffed bit.

SEND
This transformation creates the analogous output to a request. It places the possible edge in the correct place within the bit time.

send bit
This transformation takes care of the initiation of the PLS_TD.request. When the node is in receive status, it blocks incoming calls.

send Rbit
This transformation takes care of sending the next bit from an error or overload frame.

send Rframe
This transformation sends the received frame on a last bit received event.

Serialize frame
This transformation takes care of the serialization of the MAC frame.

SFcontrol
This transformation keeps track of the frame transmission. It also initiates an overload or error frame, and reveals the status it is in.
SYNC CONTROL
This transformation keeps track of the timing within the bit time. It also takes care of the synchronization (i.e. hard synchronization and resynchronization).

Tbound crossing message generation
This transformation takes care of the generation of the Tbound crossing message.

Tcompare
This transformation compares the input bit with the output bit.

tcontrol
This transformation takes care of the control of the transmit error counter.

TQ generation
This transformation generates an event after every time quantum. This event is derived from the system clock. According to the CAN specification, it has to be scalable by means of a prescaler (m), which is part of the timing parameters that are stored in the PARAMS store.

TMAM control
This transformation takes care of the control of all transmit functionalities. It also keeps track of the status of the node.

TRANSMIT
This transformation takes care of the functionality of the transmit process.

Transmit Data Encapsulation
The tasks of this transformation are:
• Acceptance of LLC frames and interface control information
• CRC sequence calculation
• Construction of MAC frame by adding SOF, RTR bit, reserve bits, CRC, ACK, and EOF to the LLC frame

TRANSMIT ERROR COUNTER
This transformation contains the error counter of the transmitter. It updates the counter when necessary. It also detects boundary crossings of the counter.

Transmit Media Access Management
The task of this transformation are:
• Initiation of the transmission process after recognizing bus idle (compliance with interframe space)
• Serialization of the MAC frame
• Insertion of stuffbits (bit stuffing)
• Arbitration and passing into receive mode in case of loss of arbitration
• Error detection (monitoring, format check)
• Acknowledgement check
• Recognition of an overload condition
• Overload frame construction and initiation of transmission
Modeling and implementation of a CAN controller

- Error frame construction and initiation of transmission
- Presentation of a serial bit stream to the physical layer for transmission

Tsend
This transformation moves the Tbit_send to stuffed bit or, in case a stuff bit has to be inserted, moves the Tprevious bit to stuffed bit. In the latter case, the next step is to send the complementary value of the Tprevious bit to stuffed bit.

(re)sync
This transformation takes care of the hard and resynchronization of. It also indicates via the status where the bus is within the bit time.

Discrete flows

Bus_Status
The bus status reflects the state of the bus. It can have several different values:
- Bus_idle_mode
- Bus_off_mode
- Error_active_mode
- Error_passive_mode

CRC_T
This flow indicates the computed CRC value of the transmitter.

DATA
This flow indicates the (possible) data of the frame.

destuffed bit
This flow represents the destuffed bit stream.

destuffed bit_ctrl
This flow represents the destuffed bit stream.

destuffed bit_nbth
This flow represents the destuffed bit stream.

DLC
This flow indicates the Data Length Code.

e
This flow gives the phase error. It can have the following values
- e=0 if the edge lies within Sync_seg
- e>0 if the edge lies before the sample point
- e<0 if the edge lies after the sample point

Filter_Params
The filter parameters consist of one or more parameters that have to be added to or deleted from the parameter filter memory.

**ID**

This flow indicates the identifier.

**L_DRA.confirm**

This primitive is a composition of three primitives: L_DATA.confirm, L_REMOTE.confirm, and L_ABORT.confirm. These primitives provide a confirmation of the L_DATA.request, L_REMOTE.request, and L_ABORT.request primitives, respectively. They are provided with the parameter TRANSFER_STATUS. In case of the DATA and REMOTE primitives, the TRANSFER_STATUS can have two values: COMPLETE or NOT_COMPLETE. In case of the ABORT primitive, the TRANSFER_STATUS can also have two values: COMPLETE or ABORTED.

**L_DRA.request**

This primitive is a composition of three primitives: L_DATA.request, L_REMOTE.request, and L_ABORT.request. These primitives provide a request to send data, to get data from a remote entity, or to abort retransmission of a previously initiated data or remote request, respectively. The L_DATA.request and L_REMOTE.request primitives are both provided with the parameters IDENTIFIER (identifies the data and its priority) and DLC (Data Length Code). The L_DATA.request primitive is additionally provided with the parameter DATA (data the user wants to transmit). The L_ABORT.request primitive is not provided with any parameter.

**L_DRA.request LLC_AR**

This primitive is a composition of three primitives: L_DATA.request, L_REMOTE.request, and L_ABORT.request. These primitives provide a request to send data, to get data from a remote entity, or to abort retransmission of a previously initiated data or remote request, respectively. The L_DATA.request and L_REMOTE.request primitives are both provided with the parameters IDENTIFIER (identifies the data and its priority) and DLC (Data Length Code). The L_DATA.request primitive is additionally provided with the parameter DATA (data the user wants to transmit). The L_ABORT.request primitive is not provided with any parameter.

**L_DR.indication**

This primitive is a composition of two primitives: L_DATA.indication and L_REMOTE.indication. These primitives indicate the arrival of data or the arrival of a request for data from a remote entity. The L_DATA.indication and L_REMOTE.indication primitives are both provided with the parameters IDENTIFIER (identifies the data and its priority) and DLC (Data Length Code). The L_DATA.indication primitive is additionally provided with the parameter DATA (data the user wants to transmit).

**L_DR.request**
This primitive is a composition of two primitives: L_DATA.request, and L_REMOTE.request. These primitives provide a request to send data, or to get data from a remote entity, respectively. The L_DATA.request and L_REMOTE.request primitives are both provided with the parameters IDENTIFIER (identifies the data and its priority) and DLC (Data Length Code). The L_DATA.request primitive is additionally provided with the parameter DATA (data the user wants to transmit).

L_DR_frame
An L_DATA frame or L_REMOTE frame

MA_DRO.confirm
This primitive is a composition of three primitives: MA_DATA.confirm, MA_REMOTE.confirm, and MA_OVLD.confirm. These primitives provide a confirmation of the MA_DATA.request, MA_REMOTE.request, and MA_OVLD.request primitives, respectively. They are provided with the parameter TRANSMISSION_STATUS. This parameter can have one of two different values: SUCCESS or NO_SUCCESS.

MA_DRO.request
This primitive is a composition of three primitives: MA_DATA.request, MA_REMOTE.request, and MA_OVLD.request. These primitives provide a request to send data, to get data from a remote entity, or to send an overload frame, respectively. The MA_DATA.request and the MAREMOTE.request primitives are both provided with the parameters IDENTIFIER (identifies the data and its priority) and DLC (Data Length Code). The MA_DATA.request primitive is additionally provided with the parameter DATA (data the user wants to transmit). The MA_OVLD.request primitive is not provided with any parameter.

MA_DR.confirm_LLCC
This primitive is a composition of two primitives: MA_DATA.confirm, and MA_REMOTE.confirm. These primitives provide a confirmation of the MA_DATA.request, and MA_REMOTE.request, respectively. They are provided with the parameter TRANSMISSION_STATUS. This parameter can have one of two different values: SUCCESS or NO_SUCCESS.

MA_DR.indication
This primitive is a composition of two primitives: MA_DATA.indication and MA_REMOTE.indication. These primitives indicate the arrival of data or the arrival of a request for data from a remote entity. The MA_DATA.indication and MA_REMOTE.indication primitives are both provided with the parameters IDENTIFIER (identifies the data and its priority) and DLC (Data Length Code). The MA_DATA.indication primitive is additionally provided with the parameter DATA (data the user wants to transmit).

MA_DR-request_LLCC
This primitive is a composition of two primitives: MA_DATA.request, and MA_REMOTE.request. These primitives provide a request to send data, and
Modeling and implementation of a CAN controller

to get data from a remote entity, respectively. The MA_DATA.request and the
MA_REMOTE.request primitives are both provided with the parameters
IDENTIFIER (identifies the data and its priority) and DLC (Data Length
Code). The MA_DATA.request primitive is additionally provided with the
parameter DATA (data the user wants to transmit).

MA_DR.request_MAC_TR
This primitive is a composition of two primitives: MA_DATA.request, and
MA_REMOTE.request. These primitives provide a request to send data, or to
get data from a remote entity, respectively. The MA_DATA.request and the
MA_REMOTE.request primitives are both provided with the parameters
IDENTIFIER (identifies the data and its priority) and DLC (Data Length
Code). The MA_DATA.request primitive is additionally provided with the
parameter DATA (data the user wants to transmit).

MA_DR.request_MAC_TR_CP
This primitive is a composition of two primitives: MA_DATA.request, and
MA_REMOTE.request. These primitives provide a request to send data, or to
get data from a remote entity, respectively. The MA_DATA.request and the
MA_REMOTE.request primitives are both provided with the parameters
IDENTIFIER (identifies the data and its priority) and DLC (Data Length
Code). The MA_DATA.request primitive is additionally provided with the
parameter DATA (data the user wants to transmit).

MA_DR.request_MAC_TR_GMF
This primitive is a composition of two primitives: MA_DATA.request, and
MA_REMOTE.request. These primitives provide a request to send data, or to
get data from a remote entity, respectively. The MA_DATA.request and the
MA_REMOTE.request primitives are both provided with the parameters
IDENTIFIER (identifies the data and its priority) and DLC (Data Length
Code). The MA_DATA.request primitive is additionally provided with the
parameter DATA (data the user wants to transmit).

MA_OVLD.confirm_LLC
This primitive provides a confirmation of the MA_OVLD.request primitive. It
is provided with the parameter TRANSMISSION_STATUS. This parameter
can have one of two different values: SUCCESS or NO_SUCCESS.

Node_Status
Indicates the current status of the node (i.e. it signals whether or not the node
is in the bus off state).

PLS_DATA.indication
This is an indication of the arrival of a dominant or recessive bit. This
primitive provides one parameter: INPUT_UNIT. INPUT_UNIT can have one
of the following two values: DOMINANT or RECESSIVE.

PLS_DATA.request
Request for a dominant or recessive bit. This primitive provides one parameter: OUTPUT_UNIT. OUTPUT_UNIT can have one of the following two values: DOMINANT or RECESSIVE.

**PLS_RD.indication**

This is an indication of the arrival of a dominant or recessive bit. This primitive provides one parameter: INPUT_UNIT. INPUT_UNIT can have one of the following two values: DOMINANT or RECESSIVE.

**PLS_RD.indication_bd**

This is an indication of the arrival of a dominant or recessive bit. This primitive provides one parameter: INPUT_UNIT. INPUT_UNIT can have one of the following two values: DOMINANT or RECESSIVE.

**PLS_RD.indication_Rcomp**

This is an indication of the arrival of a dominant or recessive bit. This primitive provides one parameter: INPUT_UNIT. INPUT_UNIT can have one of the following two values: DOMINANT or RECESSIVE.

**PLS_RD.request**

Request for a dominant or recessive bit. This primitive provides one parameter: OUTPUT_UNIT. OUTPUT_UNIT can have one of the following two values: DOMINANT or RECESSIVE.

**PLS_RD.request_Rcomp**

Request for a dominant or recessive bit. This primitive provides one parameter: OUTPUT_UNIT. OUTPUT_UNIT can have one of the following two values: DOMINANT or RECESSIVE.

**PLS_RD.request_s**

Request for a dominant or recessive bit. This primitive provides one parameter: OUTPUT_UNIT. OUTPUT_UNIT can have one of the following two values: DOMINANT or RECESSIVE.

**PLS_TD.indication**

This is an indication of the arrival of a dominant or recessive bit. This primitive provides one parameter: INPUT_UNIT. INPUT_UNIT can have one of the following two values: DOMINANT or RECESSIVE.

**PLS_TD.request**

Request for a dominant or recessive bit. This primitive provides one parameter: OUTPUT_UNIT. OUTPUT_UNIT can have one of the following two values: DOMINANT or RECESSIVE.

**PLS_TD.request_sb**

Request for a dominant or recessive bit. This primitive provides one parameter: OUTPUT_UNIT. OUTPUT_UNIT can have one of the following two values: DOMINANT or RECESSIVE.
PLS_TD.request_Tcomp
Request for a dominant or recessive bit. This primitive provides one parameter: OUTPUT_UNIT. OUTPUT_UNIT can have one of the following two values: DOMINANT or RECESSIVE.

Rbit
This flow represents the bit in turn.

Rbit_ap
This flow represents the bit in turn.

Rbit_send
This flow represents the bit in turn.

Rbound crossing
This flow indicates that the receive error counter (REC) has reached one of four values: ep_warning, error passive, bus off, or error active. The ep_warning message is sent when REC exceeds 96. The error passive message is sent when REC exceeds 128. The bus off message is sent when REC exceeds 256. The error active message is sent when REC falls below 128 again.

Rcommands
This flow takes care of passing commands from the RMAMcontrol transformation to the Rsend transformation. It can have the values erro, overload, and next bit.

Receive MAC frame
This flow indicates a MAC frame.

Rerror
This flow indicates that an error situation has occurred, which has a reflection on the receive error counter.

result_RMAM
This flow can take one of three values: match, no_match{DioR}, and no_match{RioD}. DioR means dominant instead of recessive and RioD means recessive instead of dominant.

result_TMAM
This flow can take one of three values: match, no_match{DioR}, and no_match{RioD}. DioR means dominant instead of recessive and RioD means recessive instead of dominant.

RMAMcommands
This flow is used to control the deserialized frame transformation. It can have the values next bit, error, and EO_del.

RMAMstatus
This flow reflects the status of the receiver. It can have the following values: SOF, RTR, control, data, CRC, CRC_del, ACK, ACK_del/EOF, intermission, suspend transmission, arbitration loss, frame received, frame send, overload, form error, bit error, CRC error, and 11recessive.

Rmode
This flow indicates a mode change to the receiver. It is used to enable/disable the receiver and to reset it.

Rmode_ctrl
This flow indicates a mode change to the receiver. It is used to enable/disable the receiver and to reset it.

Rmode_s
This flow indicates a mode change to the receiver. It is used to enable/disable the receiver and to reset it.

Rstatus
This flow reflects the status of the receiver. It can have the following values: SOF, RTR, control, data, CRC, CRC_del, ACK, ACK_del/EOF, intermission, suspend transmission, arbitration loss, frame received, frame send, overload, form error, bit error, CRC error, and 11recessive.

Rstatus_ctrl
This flow reflects the status of the receiver. It can have the following values: SOF, RTR, control, data, CRC, CRC_del, ACK, ACK_del/EOF, intermission, suspend transmission, arbitration loss, frame received, frame send, overload, form error, bit error, CRC error, and 11recessive.

RTR
This flow indicates the RTR bit.

status
This flow indicates the status within the bit time. It can have the values sync_seg, prop_seg, phase_seg1, and phase_seg2.

stuffed bit
This flow encompasses the stuffed bit stream

Tbit
This flow represents the bit in turn.

Tbit_ap
This flow represents the bit in turn.

Tbit_send
This flow represents the bit in turn.

Tbound crossing
This flow indicates that the transmit error counter (TEC) has reached one of four values: ep_warning, error passive, bus off, or error active. The ep_warning message is sent when TEC exceeds 96. The error passive message is send when TEC exceeds 128. The bus off message is send when TEC exceeds 256. The error active message is send when TEC falls below 128 again.

Tcommands
This flow is used to give commands from the TMAMcontrol transformation to the Serialize frame transformation. It can take the following values: next bit, error, overload, send bit, and EO_del.

Terror
This flow indicates that an error situation has occurred, which has a reflection on the transmit error counter.

Timing_Params
The timing parameters consist of the number of time quanta of the time segments (PROP_SEGMENT_TQ, PHASE_SEGMENT1_TQ, and PHASE_SEGMENT2_TQ), of the prescaler value (m), and of the resynchronization jump width value (RJW_TQ).

TMAMmode
This flow indicates a mode change to the transmitter. It is used to enable/disable the transmitter and to reset it. It can have the following values: error passive, error active, disable, and reset.

TMAMstatus
This flow reflects the status of the transmitter. It indicates in what position the transmitter is within the frame. It also indicates the arrival of a new data or remote frame. It can have the following values: DATA_F, REMOTE_F, SOF, identifier, RTR, control, Data, CRC, CRC_del, ACK, intermission, suspend transmission, and frame send.

Tmode
This flow indicates a mode change to the transmitter. It is used to enable/disable the transmitter and to reset it. It can have the following values: error passive, error active, disable, and reset.

Tmode_ctrl
This flow indicates a mode change to the transmitter. It is used to enable/disable the transmitter and to reset it. It can have the following values: error passive, error active, disable, and reset.

transmit bit
This flow depicts the serialized bit stream.

Transmit MAC frame
This flow indicates a MAC frame.
Tstatus
This flow reflects the status of the transmitter. It indicates in what position
the transmitter is within the frame. It also indicates the arrival of a new data or
remote frame. It can have the following values: DATA_F, REMOTE_F, SOF,
identifier, RTR, control, Data, CRC, CRC_del, ACK, intermission, suspend
transmission, and frame send.

Tstatus_ctrl
This flow reflects the status of the transmitter. It indicates in what position the
transmitter is within the frame. It also indicates the arrival of a new data or
remote frame. It can have the following values: DATA_F, REMOTE_F, SOF,
identifier, RTR, control, Data, CRC, CRC_del, ACK, intermission, suspend
transmission, and frame send.

Continuous flows

input
The input message represents the current value of the CAN bus.

input_1
The input message represents the current value of the CAN bus.

input_2
The input message represents the current value of the CAN bus.

output
The output message is the value that has to be put on the CAN bus (when the
node is transmitting).

Event flows

Bus_off
The Bus_off message tells the Physical Medium Attachment to transfer to the
bus off state.

Bus_off_release
The Bus_off_release message tells the Physical Medium Attachment to
transfer to the bus on state.

delete
This event tells the Remove Request transformation to remove a request

disable send bit
This event tells the send bit transformation to stop sending.

disable_bd
This event tells the bit destuffing transformation to stop destuffing
disable_bs
   This event tells the bit stuffing transformation to stop stuffing.

edge
   This event indicates that an edge has been detected.

enable_send bit
   This event tells the send bit transformation to start sending.

enable_bd
   This event tells the bit destuffing transformation to start destuffing.

enable_bs
   This event tells the bit stuffing transformation to start stuffing.

last bit received
   This event indicates that the last bit was received.

last Tbit send
   This event indicates that the last bit was send.

L_ABORT.request LLC_st
   This primitive provides a request to abort retransmission of a previously
   initiated data or remote request. It is not provided with any parameter.

MAC frame arrival
   This event indicates the arrival of a MAC frame.

MA_OVLD.request LLC
   This primitive provides a request to send an overload frame. It is not provided
   with any parameter.

MA_OVLD.request MAC_TR
   This primitive provides a request to send an overload frame. It is not provided
   with any parameter.

next Tbit
   This transformation indicates that the next bit from the Tbuffer has to be send.

not queue full
   This event flow indicates that the LLC can store an incoming frame. It is
   initiated after the queue has been full.

queue full
   This event flow indicates that no incoming frame can be buffered at this
   moment and that a overload frame has to be send.

ready
This event indicates that the compare ID transformation is ready to accept another frame.

ready_MAC_TR
This event flow indicates that all information necessary to compute the CRC value is present.

request
This event indicates the arrival of a data request. It is used to achieve the 'hard synchronization' of the transmitter.

reset
The reset event flow is used to reset the LLC and the MAC transformation. It therefore split into two different reset event flow: reset_LL and reset_MAC.

reset_LL
This event flow is used to reset the LLC transformation.

reset_LL_BR
This event flow is used to reset the buffer request transformation of hte LLC.

reset_LL_FAF
This event flow is used to reset the frame acceptance filtering transformation of the LLC.

reset_LL_FAF_2
This event flow is used to reset the frame acceptance filtering transformation of the LLC.

reset_LL_FAF_CFP
This event flow is used to reset the change filter params transformation of the frame acceptance filtering.

reset_LL_FAF_CID
This event flow is used to reset the compare ID transformation of the frame acceptance filtering.

reset_LL_FAF_OC
This event flow is used to reset the overload control transformation of the frame acceptance filtering.

reset_LL_FAF_QC
This event flow is used to reset the queue control transformation of the frame acceptance filtering.

reset_MAC
This event flow is used to reset the MAC transformation.

Reset_Request
Request to set the node into an initial state.

retrieve
This event indicates to the Retrieve Request transformation that it has to retrieve a request for retransmission.

Reset_Response
Response to the Reset_Request.

Reset
When this event is triggered, the receive error counter is reset (= set to 0).

Rstuff bit
This event indicates the deletion of a stuff bit to the RMAMcobelrol transformation.

Rstuff error
This event indicates that a stuff error has occurred.

sample
This event indicates the sample point to the receiver. When the node is in the receive state, it has to sample the input at this point within the bit time.

sync
This event indicates the point in time on which transmitter has to put the edge (if necessary).

TMAMerror
This event indicates that an error frame has to be send.

TMAMoverload
This event indicates that an overload frame has to be send.

TQ
This event is triggered every time quantum.

Treset
When this event is triggered, the transmit error counter is reset (= set to 0).

Tstuff bit
This event indicates that a stuff bit is added to the bit stream.

Tstuff bit_dsb
This event indicates that a stuff bit is added to the bit stream.

Tstuff bit_send
This event indicates that a stuff bit is added to the bit stream.
Data stores

CRC
This store holds the result of the CRC calculation.

DorR
This store indicates whether the frame being transmitted is a data frame or a remote frame.

error
This store contains the phase error number specified in time quanta.

error frame
This store contains the (passive or active) error frame.

filter params list
This store contains a list of identifiers. If the identifier of an incoming frame matches one of the identifiers in this list, the frame has to be passed to the M3S extension. If no identifier from the list matches the identifier from the incoming frame, this frame is rejected.

generator_T
This store contains the generator polynomial for the CRC calculation.

number of data bits
This store contains the number of data bits of the frame.

PARAMS
This store contains the timing parameters. They consist of the number of time quanta of the time segments (PROP_SEG_#TQ, PHASE_SEG1_#TQ, and PHASE_SEG2_#TQ), of the prescaler value (m), and of the resynchronization jump width value (RJW_#TQ). It also contains a flag that indicates if the bit rate is low or high.

polynomial_T
This store contains the part of the MAC frame on which the CRC calculation has to take place.

queue
In this store the incoming frames are queued. It can contain one data or remote frame.

Rbit cnt
This store contains a counter to keep track of the place within the error or overload frame.

Rbit count
This store is used as a counter to keep track of where in the frame we are.
Rbuffer
This store is used to store the received bits and create the MAC frame.

Rcounter
This store counts the number of consecutive matching bits.

REC
This store contains the Receive Error Counter.

Reframe
This store contains the error frame that in case of an error has to be send. It can have the value error-active or error-passive.

Rerror frame
This store contains the error frame that in case of an error has to be send. It can have the value error-active or error-passive.

REQUEST
This store contains a request that is being send. It is used to retrieve the request when a retransmission has to be carried out.

Rprevious
This store contains the value of the previous received bit.

Tbit count
This store is used as a counter to keep track of where in the frame we are.

Tbuffer
This store buffers the MAC frame. It is read from one bit at a time.

Tcounter
This store contains the number of matching consecutive bits.

TEC
This store contains the Transmit Error Counter.

Temode
This store indicates if the error mode is error active or error passive.

Tprevious
This store remembers the previous bit.

#data bits
This store contains the number of data bits of the frame.
Control stores

ABORT
This store remembers if an L_ABORT.request_LLCC_st event has taken place in the past.

OVLD_MAC_TR
This store remembers if an MO_OVLD.request_MAC_TR event has taken place in the past.

SEND_RQ
This store remembers if a request event has taken place.
Appendix E - State transition diagrams

In this appendix, the state transition diagrams of the model of CAN are presented.

BUFFER REQUESTS

Figure E1. State transition diagram of the Add Request transformation
Figure E2. State transition diagram of the br_control transformation

Figure E3. State transition diagram of the Remove Request transformation
Modeling and implementation of a CAN controller

Figure E4. State transition diagram of the Retrieve Request transformation

FRAME ACCEPTANCE FILTERING

Figure E5. State transition diagram of the change filter params transformation
Figure E6. State transition diagram of the compare ID transformation
Figure E7. State transition diagram of the overload control transformation
Figure E8. State transition diagram of the queue control transformation
MAC MANAGEMENT

Figure E9. State transition diagram of the MAC MANAGEMENT transformation

Transmit Data Encapsulation

Figure E10. State transition diagram of the construct polynomial transformation
Modeling and implementation of a CAN controller

Figure E11. State transition diagram of the divide modulo 2 transformation

Figure E12. State transition diagram of the generate MAC frame transformation
Transmit Media Access Management

Figure E13. State transition diagram of the send bit transformation

Figure E14. State transition diagram of the Tcompare transformation
bit stuffing

Figure E15. State transition diagram of the add Tprevious transformation

Figure E16. State transition diagram of the compare and update Tcounter transformation
Idlem

\[ \text{Tcounter} = 6 \]
\[ \text{trigger Tstuff bit Ldsb} \]
\[ \text{Tcounter} := 0 \]

**Figure E17.** State transition diagram of the detect Tstuff bit transformation
Figure E18. State transition diagram of the Tsend transformation
Serialize frame

Figure E19. State transition diagram of the manage Tbuffer transformation
Modeling and implementation of a CAN controller

Figure E20. State transition diagram of the next bit from Tbuffer transformation

Receive Data Decapsulation

Figure E21. State transition diagram of the Construct LLC frame transformation
Modeling and implementation of a CAN controller

Figure E22. State transition diagram of the Decapsulate MAC frame transformation

Receive Media Access Management

Figure E23. State transition diagram of the Rcompare transformation
Rsend

Figure E24. State transition diagram of the add Rerror frame transformation

Figure E25. State transition diagram of the send Rbit transformation
bit destuffing

Figure E26. State transition diagram of the add Rprevious transformation
Figure E27. State transition diagram of the Compare and update Rcounter transformation
Figure E28. State transition diagram of the Rsend_db transformation

deserialize frame

Figure E29. State transition diagram of the CRC calculation transformation
Figure E30. State transition diagram of the next Rbit to Rbuffer transformation

Figure E31. State transition diagram of the send Rframe transformation
TRANSMIT ERROR COUNTER

Figure E32. State transition diagram of the Tbound crossing message generation transformation

Figure E33. State transition diagram of the te_control transformation
Modeling and implementation of a CAN controller

RECEIVE ERROR COUNTER

Figure E34. State transition diagram of the Rbound crossing message generation transformation

Figure E35. State transition diagram of the re_control transformation
PLS

Figure E36. State transition diagram of the RCV transformation

Figure E37. State transition diagram of the SEND transformation
Modeling and implementation of a CAN controller

Figure E38. State transition diagram of the Time Quantum Generation transformation

SYNC CONTROL

Figure E39. State transition diagram of the edge detection transformation
Figure E40. State transition diagram of the phase error detection transformation
Figure E41. State transition diagram of the (re)sync transformation
Appendix F - Description of the used operators

In this appendix, a description of the used operators of the bitprocessor is given. The operators descriptions contain nested IF-THEN-ELSE constructions. The brackets that make up such construction are accompanied by comments of the following form:

\[
\text{(guard)}
\]
\[
\text{if 0: "BEGIN\{name\}" (}
\]
\[
\text{...}
\]
\[
\text{) "END\{name\}"}
\]

There is a comment before an operator expression, if necessary. So if NextBit is an output of the operator, and this output has a certain task, then this task is described in the comment preceding the output name. This comment is started with the output name. This appears in the following form:

"NextBit: if this output equals 1, the next bit has to be transmitted."

NextBit:= ....

Operators of the PLS schematic

The TQgen operator

Description of the function Always

"cnt0: controls the TQcnt register. This operator is used to generate the TQ clock from the internal clock cycle."

cnt0:=
(\text{reset})
\text{if 0: (}
\text{(cntI=m)}
\text{if 0:}
\text{cntI+1}
\text{if 1:}
\text{1}
\text{)}
\text{if 1:}
\text{0.}

"TQ: this output carries a one every time quantum."

TQ:=(cntI=m)
The ParamsControl operator

Description of the function Always

"PRO: controls the paramsREG register. If new parameters arrive, they are stored in this register. The two most significant bits indicate which part of the parameters is loaded."

\[\text{PRO:= ((params from:8 to:9)=\text{\%00}) if0: "BEGIN\{Load params\}" ( ((params from:8 to:9)=\text{\%01}) if0: "BEGIN\{Not prescaler and prop\}" ( (params at:0),(PRI from:0 to:15) if1: "Load phase1, phase2, RJW" (PRI at:1),(params from:0 to:7),(PRI from:0 to:7) ) \"END\{Not prescaler and prop\}" if1: "Load prescaler and prop" (PRI from:8 to:16),(params from:0 to:7) ) \"END\{Load Params\}" if1: "Hold" ) \"

The outputs prescaler, prop, phase1, phase2, RJW and speed are used to carry the appropriate values."


The SyncControl operator

Description of the function Always

"StateO: controls the State register. This register can have four values: %00 Synchronisation state %01 Propagation state %10 Phase segment 1 state %11 Phase segment 2 state."

\[\text{StateO:= (TQ) if0: "Hold" StateI if1: "BEGIN\{Advance\}" ( (StateI=0) \(\lor\) (cntI=prop)) \(\lor\) (StateI=2) \(\lor\) (cntI=phase1))} \]
if0: (  
  ((StateI=3) \ (cntI=phase2))  
  if0:  
    StateI  
  if1: (  
    0 width:2  
  )  
  )  
if1:  
StateI+1  
) "END(Advance)".

"cntO: used as a TQ counter"

cntO:=  
  (TQ)  
  if0: "Hold"  
  cntI  
  if1: "BEGIN(Advance)" (  
    ((StateI=0) \  
      ((StateI=1) \ (cntI=prop)) \  
      ((StateI=2) \ (cntI=phase1)) \  
      ((StateI=3) \ (cntI=phase2))  
    )  
    if0:  
      cntI+1  
    if1:  
      %000  
  ) "END(Advance)".

PrevBitO:=in.  
HSO:=HSI.  
reset:=%0.  

_sync:=  
  (TQ) \ (StateI=3) \ (cntI=phase2).  

reqO:=  
  (_sync)  
  if0: (  
    (request at:1)  
    if0:  
      reqI  
      if1:  
        request  
    )  
  if1:  
    %00.  

oBitO:=  
  (_sync)  
  if0:  
    oBitI  
  if1: (  
    (request at:1)  
    if0: (  
      (reqI at:1)  
      if0:  
        %1  
      if1:  
        (reqI at:0)  
    )  
  if1:  
    (request at:0)
Modeling and implementation of a CAN controller

\[ \text{out} := \text{oBitI}. \]

\[ \_\text{sample} := \text{(TQ)}/(\text{StateI}=2)/(\text{cntI} = \text{phaseI}). \]

\[ \text{indication} := \text{(\_\text{sample})} \]

\[ \begin{align*}
\text{if 0:} & \quad \%00 \\
\text{if 1:} & \quad (\%1 \text{ width:1}),\text{in} \\
\end{align*} \]

\[ \text{Operator of the Tx schematic} \]

The control specification of the Control operator of the Tx schematic is given below:

\[ \begin{align*}
\&0000 \text{ Idle.} \\
\&0001 \text{ SOF.} \\
\&0010 \text{ Arbitration.} \\
\&0011 \text{ Control.} \\
\&0100 \text{ Data.} \\
\&0101 \text{ CRC.} \\
\&0110 \text{ CRCdelimiter.} \\
\&0111 \text{ ACKTx.} \\
\&1000 \text{ ACKRx.} \\
\&1001 \text{ ACKdelimiter.} \\
\&1010 \text{ EOF.} \\
\&1011 \text{ Intermission.} \\
\&1100 \text{ SuspendTransmission.} \\
\&1101 \text{ ActiveErrorOverload.} \\
\&1110 \text{ PassiveError.} \\
\&1111 \text{ Superposition} \\
\end{align*} \]

\[ \text{Description of the function ACKdelimiter} \]

"This is the state the transmitter is in during the ACK delimiter slot."

"so: serial out. Transmit a recessive bit when NextBit is equal to 1."

\[ \text{so} := \text{(NextBit)} \]

\[ \begin{align*}
\text{if 0:} & \quad (\%00 \text{ width:2}) \\
\text{if 1:} & \quad \%11. \\
\end{align*} \]

\[ \text{outO} := \text{outI}. \]

\[ \text{cO} := \text{cI} \]

\[ \text{Description of the function ACKRx} \]

"This is the state the transmitter is in during the ACK slot when receiving."
"so: serial out. Transmit a dominant bit when NextBit is equal to 1."

\[
\text{so: } (\text{NextBit}) \\
\quad \text{if 0:} \\
\quad \quad (\%00 \text{ width:2}) \\
\quad \quad \text{if 1:} \\
\quad \quad \%10. \\
\text{outO:=outI.} \\
\text{cO:=cI}
\]

Description of the function ACKTx

"This is the state the transmitter is in during the ACK slot when receiving."

"so: serial out. Transmit a recessive bit when NextBit is equal to 1."

\[
\text{so:=} \\
\quad (\text{NextBit}) \\
\quad \text{if 0:} \\
\quad \quad (\%00 \text{ width:2}) \\
\quad \quad \text{if 1:} \\
\quad \quad \%11. \\
\text{outO:=outI.} \\
\text{cO:=cI}
\]

Description of the function ActiveErrorOverload

"This is the state the transmitter is in during an active error or overload flag."

"so: serial out. Transmit a dominant bit when NextBit equals 1."

\[
\text{so:=} \\
\quad (\text{NextBit}) \\
\quad \text{if 0:} \\
\quad \quad (\%00 \text{ width:2}) \\
\quad \quad \text{if 1:} \\
\quad \quad \%10. \\
\text{outO:=outI.} \\
\text{cO:=cI}
\]

Description of the function Arbitration

"This is the state the transmitter is in during the arbitration phase."

"so: serial out. Transmit the next bit from the out register when NextBit is 1."

\[
\text{so:=} \\
\quad (\text{NextBit})
\]
Modeling and implementation of a CAN controller

if0:
  %00
if1:
  (%1 width:1), (outI at:7).

"outo: controls the out register. It is used as a shift register to output the successive bits of the arbitration."

out0:=
  (NextByte)
  if0: "BEGIN(No next byte)" (NextBit)
       if0:
         outI
       if1:
         (outI rol:1)
   ) "END(No next byte)"
if1: "BEGIN(Next byte)" (cI)
  if0: 
    (pi shl:4)
  if1:
    (pi shl:2)
  ) "END(Next byte)".

"c0: controls the cBit. It is used to distinguish the upper and lower part of the arbitration."

c0:=
  (NextByte)
  if0: 
    cI
  if1:
    (cI not)

Description of the function Control

"This is the state the transmitter is in during the Control phase."

"s0: serial out. Transmit the control information bit by bit."

s0:=
  (NextBit)
  if0:
    %00
  if1:
    (%1 width:1), (outI at:7).

"out0: controls the out register. It is used as a shift register to output the control information."

out0:=
  (NextByte)
  if0: "BEGIN(No next byte)" (NextBit)
       if0:
Modeling and implementation of a CAN controller

outI
if1:
  (outI rol:1)
) "END(No next byte)"
if1: "Next byte"
p

cO:=cI

Description of the function CRC

"This is the state the transmitter is in during the CRC phase."

"so: serial out. Transmit the CRC sequence bit by bit when NextBit equals 1."

so:=
  (NextBit)
  if0:
    %00
  if1:
    (%1 width:1),CRC.

outO:=outI.
cO:=cI

Description of the function CRCdelimiter

"This is the state the transmitter is in during the CRC delimiter slot."

"so: serial out. Transmit a recessive bit when NextBit equals 1."

so:=
  (NextBit)
  if0:
    (%00 width:2)
  if1:
    %11.

outO:=outI.
cO:=cI

Description of the function Data

"This is the state the transmitter is in during the data phase."

"so: serial out. Transmit the next data bit when NextBit equals 1."

so:=
  (NextBit)
  if0:
    %00
  if1:
Modeling and implementation of a CAN controller

\[
\text{\texttt{outO:=}}
\begin{cases}
\text{\texttt{(NextByte)}} \\
\quad \text{if} \, 0: \text{\texttt{"BEGIN\!(No next byte)\!" (}}} \\
\quad \quad \text{\texttt{(NextBit)}} \\
\quad \quad \quad \text{if} \, 0: \text{\texttt{outI}} \\
\quad \quad \quad \quad \text{if} \, 1: \text{\texttt{(outI rol:1)}} \\
\quad \quad \text{\texttt{) \texttt{"END\!(No next byte)\!"}}} \\
\quad \text{if} \, 1: \text{\texttt{"Next byte"}} \\
\end{cases}
\]
\[\text{\texttt{pi.}}\]
\[\text{\texttt{c0:=c1}}\]

Description of the function EOF

"This is the state the transmitter is in during the EOF phase."

"so: serial out. Transmit a recessive bit every time Nextbit equals 1."

\[
\text{\texttt{so:=}}
\begin{cases}
\text{\texttt{(NextBit)}} \\
\quad \text{if} \, 0: \text{\texttt{(%00 width:2)}} \\
\quad \quad \text{if} \, 1: \text{\texttt{%11.}} \\
\end{cases}
\]
\[\text{\texttt{outO:=outI.}}\]
\[\text{\texttt{c0:=c1}}\]

Description of the function Idle

"This is the state the transmitter is in during the Idle phase."

"outO: when NextByte equals 1, the first byte of the arbitration is loaded into the out register."

\[
\text{\texttt{outO:=}}
\begin{cases}
\text{\texttt{(NextByte)}} \\
\quad \text{if} \, 0: \text{\texttt{outI}} \\
\quad \quad \text{if} \, 1: \text{\texttt{pi.}} \\
\end{cases}
\]
\[\text{\texttt{so:=%00.}}\]
\[\text{\texttt{c0:=c1}}\]

Description of the function Intermission
"This is the state the transmitter is in during the Intermission phase."

"so: serial out. If NextBit equals 1, a recessive bit is transmitted."

\[ so:=(\text{NextBit}) \]
\[ \text{if 0:} \]
\[ \%00 \]
\[ \text{if 1:} \]
\[ \%11. \]

\[ \text{outO:=outI.} \]
\[ \text{cO:=cI} \]

Description of the function PassiveError

"This is the state the transmitter is in during the transmission of a passive error flag."

"so: serial out. Transmit a recessive bit every time when NextBit equals 1."

\[ so:=(\text{NextBit}) \]
\[ \text{if 0:} \]
\[ \%00 \]
\[ \text{if 1:} \]
\[ \%11. \]

\[ \text{outO:=outI.} \]
\[ \text{cO:=cI} \]

Description of the function SOF

"This is the state the transmitter is in during the SOF phase."

"so: serial out. When NextBit equals 1, send a dominant bit."

\[ so:=(\text{NextBit}=1),(%0 \text{ width:1}). \]

"out0: When NextByte equals 1, load the first byte of the arbitration field into the out register."

\[ \text{out0:=} \]
\[ (\text{NextByte}) \]
\[ \text{if 0:} \]
\[ \text{outI} \]
\[ \text{if 1:} \]
\[ \text{pi.} \]

\[ \text{c0:=cI} \]
Description of the function Superposition

"This is the state the transmitter is in during the Superposition phase of an error or overload frame."

"so: serial out. Transmit a recessive bit every time NextBit equals 1."

so :=
    (NextBit)
    if 0:
        (%00 width:2)
    if 1:
        %11.

outO := outI.
cO := cI

Description of the function SuspendTransmission

"This is the state the transmitter is in during the suspend transmission phase."

"so: serial out. Transmit a recessive bit every time NextBit equals 1."

so :=
    (NextBit)
    if 0:
        (%00 width:2)
    if 1:
        %11.

outO := outI.
cO := cI

Operator of the Rx schematic

The control specification of the Control operator of the Rx schematic is given below:

%0000 Idle.
%0001 SOF.
%0010 Arbitration.
%0011 Control.
%0100 Data.
%0101 CRC.
%0110 CRCdelimiter.
%0111 ACK.
%1000 ACK.
%1001 ACKdelimiter.
%1010 EOF.
%1011 Intermission.
%1100 SuspendTransmission.
%1101 ActiveErrorOverload.
%1110 PassiveError.
%1111 Superposition
Description of the function ACK

"This function is active during the ACK slot"

"busy: indicates that the receiver is busy"

\[
\begin{align*}
\text{busy} & := ((\text{si at:1})=0). \\
\text{cntO} & := 0. \\
\text{inO} & := \%00000000. \\
\text{DLCO} & := \%000000. \\
\text{RTRO} & := \%0. \\
\text{outO} & := \%00000000. \\
\text{NextByte} & := \%0. \\
\text{NoData} & := \%0
\end{align*}
\]

Description of the function ACKdelimiter

"This function is active during the ACK delimiter."

"busy: indicates that the receiver is busy."

\[
\begin{align*}
\text{busy} & := ((\text{si at:1})=0). \\
\text{cntO} & := 0. \\
\text{inO} & := \%00000000. \\
\text{DLCO} & := \%000000. \\
\text{RTRO} & := \%0. \\
\text{outO} & := \%00000000. \\
\text{NextByte} & := \%0. \\
\text{NoData} & := \%0
\end{align*}
\]

Description of the function ActiveErrorOverload

"This function is active during the transmission of an error or overload frame."

"cntO: counts the number of bits during the frame."

\[
\begin{align*}
\text{cntO} & :=
\begin{cases}
(\text{si at:1}) & \\
\text{if 0: "Hold"} & \\
\text{cntI} & \\
\text{ifi: "BEGIN(Advance cnt)"} & \\
\text{(cntI<5)} & \\
\text{if 0: "cnt=0"} & \\
\text{0} & \\
\text{ifi: "cnt++"} & \\
\text{cntI+1} & \\
\text{) "END(Advance cnt)".}
\end{cases}
\end{align*}
\]

"busy: indicates that the receiver is busy."

\[
\begin{align*}
\text{busy} & := (\text{cntI}<5) \\
& \quad \text{\{ \text{cntI}=5 \} \backslash \{ ((\text{si at:1})=0) \}.}
\end{align*}
\]

\[
\text{inO} := \%00000000.
\]
Description of the function Arbitration

"This state is during the arbitration phase."

"DLCO: controls the DLC register. This register is used to distinguish between the upper part of the arbitration field and the lower part. It is equal to 1 during the upper part of the arbitration field."

\[
\text{DLCO} := \begin{cases} 
0 & \text{if } 0: \text{"Hold"} \\
1 & \text{if } 1: \text{"Low part of arbitration"} \\
\text{DLCI}-(1 \text{ width:6}) & \text{if } 1: \text{"Increment count"} \\
\text{cntI}+1 & \text{if } 1: \text{"Reset count"} \\
0 & \text{else} 
\end{cases}
\]

"RTRO: controls the RTR register. The RTR bit is stored in this register."

\[
\text{RTRO} := \begin{cases} 
0 & \text{if } 0: \text{"Hold"} \\
1 & \text{if } 1: \text{"Load RTR"} \\
\text{si at:0} & \text{if } 1: \text{"Increment count"} \\
\text{cntI}+1 & \text{if } 1: \text{"Reset count"} \\
0 & \text{else} 
\end{cases}
\]

"cnt0: bit counter during the arbitration field"

\[
\text{cnt0} := \begin{cases} 
0 & \text{if } 0: \text{"Hold"} \\
\text{cntI} & \text{if } 1: \text{"Increment count"} \\
\text{cntI}+1 & \text{if } 1: \text{"Reset count"} \\
0 & \text{else} 
\end{cases}
\]

"in0: used as a shift register to collect the incoming bits."

\[
in0 := \begin{cases} 
0 & \text{if } 0: \text{"Hold"} \\
\text{inI} & \text{if } 1: \text{"Increment count"} \\
\text{cntI}+1 & \text{if } 1: \text{"Reset count"} \\
0 & \text{else} 
\end{cases}
\]
if0: "Insert first bit"
(0 width:7),(si at:0)
if1: "Shift left and insert bit"
((inI shl:1) from:1 to:7),(si at:0)
) "END(Shift)".

"outO: used to pass the completed arbitration data from the in register. If the first eight bits of the arbitration field are received, this byte is passed to this output. After the last four bits are received, these too are passed to this output."

outO:=
((si at:1)=1)/
((cntI=7)/(DLCI=1))/
((cntI=3)/(DLCI=0)))
if0: "No new output byte"
0
if1: "Load value"
((inI shl:1) from:1 to:7),(si at:0).

"NextByte: indicates when the value that the outO output carries, is valid."

NextByte:=
((si at:1)=1)/
((cntI=7)/(DLCI=1))/
((cntI=3)/(DLCI=0))).

"busy: indicates the receiver is busy."

busy:=
((DLCI=1)/
((cntI<3)/
((cntI=3)/\((si at:1)=0))))).
NoData:=%O

Description of the function Control

"This function is active during the reception or transmission of the control."

"cntO: bit counter during the Control field."

cntO:=
(si at:1)
if0: "Hold cnt"
cntI
if1: "BEGIN(Advance cnt)" (cntI<5)
if0: "cnt=0"
0
if1: "cnt++"
cntI+1
) "END(Advance cnt)".

"inO: shift register during the Control field."
Modeling and implementation of a CAN controller

\[
\text{inO:}= \\
\text{(si at:1)} \\
\text{if0: "Hold"} \\
\text{inI} \\
\text{if1: "BEGIN(Shift in)" (} \\
\text{(cntI-0)} \\
\text{if0: "Insert first bit"} \\
\text{0 width:7),(si at:0)} \\
\text{if1: "Shift left and insert bit"} \\
\text{((inI shl:1) from:1 to:7),(si at:0)} \\
\text{)}"\text{END(Shift in)}."
\]

"outO: used to pass the completed Control data from the in register. If the six bits of the Control field are received, they are passed to this output."

\[
\text{outO:}= \\
\text{(((si at:1)=1)}/(cntI=5)) \\
\text{if0: "Hold"} \\
\text{0} \\
\text{if1: "Load value"} \\
\text{((inI shl:1) from:1 to:7),(si at:0)}."
\]

"NextByte: indicates the validity of outO."

\[
\text{NextByte:}=(((si at:1)=1)/(cntI=5)).
\]

"busy: indicates that the receiver is busy."

\[
\text{busy:}=(cntI<5)/(cntI=5)/(si at:1=0)).
\]

"DLCO: controls the DLC register. After reception of the control field, the DLC is loaded into this register."

\[
\text{DLCO:}= \\
\text{(((si at:1)=1)/(cntI=5))} \\
\text{if0:} \\
\text{DLCO} \\
\text{if1: (} \\
\text{((RTRI=\%0)}/(\%(inI shl:1) from:1 to:3),(si at:0))=\%0000)) \\
\text{if0:} \\
\text{000001} \\
\text{if1:} \\
\text{00 width:2},((inI shl:1) from:1 to:3),(si at:0)} \\
\text{).}
\]

"NoData: indicates that this frame does not have a data field."

\[
\text{NoData:}= \\
\text{(((si at:1)=1)/(cntI=5))} \\
\text{if0:} \\
\text{0} \\
\text{if1:} \\
\text{(RTRI=1))/( ((inI shl:1) from:1 to:5),(si at:0))=0).}
\]
RTRO:=RTRI

Description of the function CRC

"The receiver is in this state during the CRC sequence."

"DLCO: controls the DLC register. This register is used to distinguish between the upper part of the CRC field and the lower part. It is equal to 1 during the upper part of the CRC field."

\[
\text{DLCO}:= \begin{cases} 
(\text{si at:1}=1)\land (\text{cntI}=7)\land (\text{DLCI}=1) & \text{if0: "Hold" } \\
\text{DLCI} & \text{if1: "Low part of CRC" } \\
\text{DLCI}=(1 \text{ width:6}) & 
\end{cases}
\]

"cntO: used as bitcounter during CRC sequence."

\[
\text{cntO}:= 
\begin{cases} 
(\text{si at:1}) & \text{if0: "Hold" } \\
\text{cntI} & \text{if1: "BEGIN{Advance count}" } \\
( (\text{DLCI}=1)\land (\text{cntI}<7) \lor 
( (\text{DLCI}=0)\land (\text{cntI}<6) ) & \\
0 & \text{if0: "Reset count" } \\
\text{cntI}+1 & \text{if1: "Increment count" } \\
\end{cases}
\]

"busy: indicates that the receiver is busy."

\[
\text{busy}:= (\text{si at:1})
\]

inO:=%00000000.
RTRO:=%0.
outO:=%00000000.
NextByte:=%0.
NoData:=%0

Description of the function CRCdelimiter

"The receiver is in this state during the CRC delimiter slot."

"busy: indicates that the receiver is busy."

\[
\text{busy}:= (\text{si at:1}=0).
\]

cntO:=0.
Modeling and implementation of a CAN controller

inO:=%00000000.
DLCO:=%000000.
RTRO:=%0.
outO:=%00000000.
NextByte:=%0.
NoData:=%0

Description of the function Data

"This is the state the receiver is in during the Data field."

"DLCO: controls the DLC register. It is used as a down counter to count the number of data bytes."

DLCO:= (((si at:1)=1)\(cntI=7))
   if0: "Hold"
   DLCI
   if1: "BEGIN(New byte?)" ( (DLCI=1)
        if0: DLCI-(1 width:6)
        if1: DLCI
   ) "END(New byte?)".

"cntO: used during the Data field as a bitcounter."

cntO:= (si at:1) 
   if0: "Hold"
   cntI
   if1: "BEGIN(Advance cnt)" ( (cntI<7)
        if0: "cnt=0"
        0
        if1: "cnt++"
        cntI+1
   ) "END(Advance cnt)".

"inO: used as a shift register during the Data field."

inO:= (si at:1) 
   if0: "Hold"
   inI
   if1: "BEGIN(Shift in)" ( (cntI=0)
        if0: "Insert first bit"
        (0 width:7),(si at:0)
        if1: "Shift left and insert bit"
        ((inI shl:1 from:1 to:7),(si at:0)
        ) "END(Shift in)".

"outO: every time a data byte has been received it is passed to this output."
Modeling and implementation of a CAN controller

out0:=
('''((si at:1)=1)/\ 
(cntl=7))
  if0: "Hold"
  0
  if1: "Load value"
    ((intn shl:1) from:1 to:7),si at:0).

"NextByte: indicates the validity of the out0 output."

NextByte:='((si at:1)=1)/\(cntI=7)).

"busy: indicates that the receiver is busy."

busy:=(DLCI>1)/\ 
  (cntI<7)/\ 
  ((cntI=7)/\((si at:1)=0)).

RTRO:=RTRI.
NoData:=0

Description of the function EOF

"The receiver is in this state during the EOF field."

"cnt0: used as a bitcounter during the EOF field."

cnt0:=
  (si at:1)
  if0: "Hold"
  cntI
  if1: "BEGIN(Advance cnt)" /
    (cntI<6)
      if0: "cnt=0"
      0
      if1: "cnt++"
        cntI+1
    ) "END(Advance cnt)".

"busy: indicates that the receiver is busy."

busy:=(cntI<6)/\ 
    ((cntI=6)/\((si at:1)=0)).

in0:=%00000000.
DLCO:=%0000000.
RTRO:=%0.
out0:=%00000000.
NextByte:=%0.
NoData:=%0

Description of the function Idle

"The receiver is in this state when the bus is Idle."

cnt0:=0.
Description of the function Intermission

"The receiver is in this state during the intermission field."

"cntO: used during the intermission field as a bit counter."

cntO:=
   (si at:1)
   if0: "Hold"
   cntI
   if1: "BEGIN(Advance cnt)" (cntI<2)
      if0: "cnt=0"
      0
      if1: "cnt++"
      cntI+1
   ) "END(Advance cnt)".

"busy: indicates that the receiver is busy."

busy:=(cntI<2)\/
   ((cntI=2)\/(si at:1)=0)).

Description of the function PassiveError

"The receiver is in this state during the transmission of the passive error flag."

"cntO: used as a bit counter during the transmission of an passive error flag."

cntO:=
   (si at:1)
   if0: "Hold"
   cntI
   if1: "BEGIN(Advance cnt)" (cntI<5)
      if0: "cnt=0"
      0
      if1: "cnt++"
      cntI+1
   ) "END(Advance cnt)".
"busy: indicates that the receiver is busy."

\[ \text{busy} := (\text{cntI} < 5) \lor ((\text{cntI} = 5) \land (\text{si at:1} = 0)) \]

\begin{align*}
\text{inO} &:= 00000000. \\
\text{DLCO} &:= 000000. \\
\text{RTRO} &:= 0. \\
\text{outO} &:= 00000000. \\
\text{NextByte} &:= 0. \\
\text{NoData} &:= 0
\end{align*}

**Description of the function SOF**

"When the bitprocessor is transmitting, the receiver is in this state during the SOF slot."

"busy: indicates that the receiver is busy during the SOF slot."

\[ \text{busy} := (\text{si at:1} = 0) \]

\begin{align*}
\text{cntO} &:= 0. \\
\text{inO} &:= 00000000. \\
\text{DLCO} &:= 000001. \\
\text{RTRO} &:= 0. \\
\text{outO} &:= 00000000. \\
\text{NextByte} &:= 0. \\
\text{NoData} &:= 0
\end{align*}

**Description of the function Superposition**

"The receiver is in this state during the Superposition field."

"cntO: used as a bitcounter during the Superposition state of an error or overload frame."

\[ \text{cntO} := (\text{cntI} > 0) \]

\begin{align*}
\text{if0} &:\ "\text{Wait for first recessive bit}" \\
&:\ (\text{si = 11}) \lor (\text{width:3}) \\
\text{if1} &:\ "\text{BEGIN(First recessive bit detected)}" \\
&:\ (\text{si = 11}) \\
\text{if0} &:\ "\text{Hold}" \\
&:\ \text{cntI} \\
\text{if1} &:\ "\text{BEGIN(Advance counter)}" \\
&:\ (\text{cntI = 7}) \\
\text{if0} &:\ \text{cntI} + 1 \\
\text{if1} &:\ 0 \\
&:\ "\text{END(Advance counter)}" \\
&:\ "\text{END(First recessive bit detected)}".
\end{align*}

"busy: indicates that the receiver is busy."
Modeling and implementation of a CAN controller

busy := (cntI<7)/
  ((cntI=7)/\((si \ at:1)=0))

"NoData: indicates that a sequence of eight
dominant bits has been detected."

NoData :=
  (cntI=0)/\(si\-=\%11).

in0 := \%00000000.
DLCO := \%000000.
RTRO := \%0.
out0 := \%00000000.
NextByte := \%0.

Description of the function SuspendTransmission

"The receiver is in this state during
the suspend transmission phase."

"cntO: used as a bitcounter during the
suspend transmission phase."

cntO :=
  (si \ at:1)
  if0: "Hold"
    cntI
  if1: "BEGIN(Advance cnt)" (  
    (cntI<7)
    if0: "cnt=0"
      0
    if1: "cnt++"
      cntI+1
  ) "END(Advance cnt)".

"busy: indicates that the receiver is busy"

busy := (cntI<7)/
  ((cntI=7)/\((si \ at:1)=0)).

in0 := \%00000000.
DLCO := \%000000.
RTRO := \%0.
out0 := \%00000000.
NextByte := \%0.
NoData := \%0.
Operator of the BitStuff schematic

The control specification of the Control operator of the Bitstuff schematic is given below:

%0x DoNotStuff.
%1x Stuff

Description of the function DoNotStuff

"When this function is active, no bitstuffing is submitted to the bit stream of the 'in' input."

PBO:=0.
cnt0:=0.
stuff0:=0.
out:=in

Description of the function Stuff

"When this function is active, stuff bits are inserted if necessary. When a stuff bit is inserted, the original 6th bit (stored in the PreviousBit store) is send when the ctrl input equals 1."

"PBO: control the PreviousBit register. It stores the previous bit for comparison with the current bit."

PBO:=
  (in at:1)
  if0: "Hold"
  PBI
  if1: "Store bit"
  (in at:0).

"cnt0: counts the occurences of successive bits of the same value."

cnt0:=
  (stuffI=1)
  if0: "BEGIN(No stuff bit inserted)" ( (in at:1)
    if0: "Hold"
    cntI
    if1: "BEGIN(New bit arrived)" ( (cntI<5)
      if0: "BEGIN(Stuff bit needed)" ( ((in at:0)=(PBI not))
        if0:
          (1 width:3)
          if1:
            2
      ) "END(Stuff bit needed)"
    if1: "BEGIN(Advance counter)" ( (in at:0)=PBI)"
if0: 1
if1:
    cntI+1
} "END(Advance counter)"
} "END(New bit arrived)"
} "END(No stuff bit inserted)"
if1: cntI.

"out: if no stuff bit is inserted, the value of the 'in' input is passed to this output. If a stuff bit is inserted, the inserted stuff bit is passed to this output."

out:=
(stuffI=1) if0: "BEGIN(No stuff bit inserted)" (cntI<5)
    if0: "BEGIN(Stuff bit needed)" (in at:1)
        if0: %00
        if1: (%1 width:1),(PBI not)
    } "END(Stuff bit needed)"
    if1: in
} "END(No stuff bit inserted)"
if1: "BEGIN(Stuff bit inserted)" (
  ((ctrl at:0)=1)
  if0: %00
  if1: (%1 width:1),PBI
} "END(Stuff bit inserted)".

"stuff0: controls the stuff register. If a stuff bit is detected, this register gets the value 1. If the stuff bit is send, this register is resets to 0 again."

stuff0:=
(stuffI=1)
if0: (((in at:1)=1) /
  ((cntI=5))
if1: ((ctrl at:0)=0)
Operator of the BitDestuff schematic

The control specification of the Control operator of the BitDestuff schematic is given below:

%0 DoNotDestuff.
%1 Destuff

Description of the function Destuff

"When this function is active, stuff bits are removed from the bit stream presented to the 'in' input."

"PBO: controls the PreviousBit register. This register is used to remember the previous bit so it can be compared with the current bit."

PBO:=
  (in at:1)
  if0: "No new bit arrived"
  PB1
  if1: "New bit arrived"
  (in at:0).

"cntO: counts the number of consecutive bits with the same value."

cntO:=
  (in at:1)
  if0: "Hold"
  cntI
  if1: "BEGIN(New bit arrived)" (cntI=5)
    if0: "BEGIN(Count less than 5)" ((in at:0)=PB1)
    if0:
      cntI+1
      if1:
        1
    ) "END(Count less than five)"
    if1: "Count equals 5" %1
  ) "END(New bit arrived)".

"out: if a stuff bit is expected, this bit is not passed to this output. If no stuff bit is expected, the 'in' input is passed to this output."

out:=
  ((in at:1)=%1)
  if0: "Hold"
  in
  if1: "BEGIN(New bit arrived)" (cntI=5)
    if0: "No stuff bit"
    in
    if1: "Stuff bit expected" %00
"destuff: indicates if a stuff bit has been detected."

\[
\text{destuff} := \begin{cases} 
\text{if 0: "No new bit"} & \text{cntI=5} \\
\text{if 1: "New bit"} & \text{cntI=5}/\text{in at:0=PBI}) 
\end{cases}
\]

"error: indicates if an error has been detected."

\[
\text{error} := \begin{cases} 
\text{if 0: "No new bit"} & \text{in at:1=1} \\
\text{if 1: "New bit arrived"} & \text{cntI=5}/\text{in at:0=PBI}) 
\end{cases}
\]

Description of the function DoNotDestuff

"No destuffing is done when this function is active. The input is directly passed to the output."

\[
PBO := \text{in} \text{ at:0=0}.
\]
\[
cnt0 := \text{0}.
\]
\[
\text{destuff} := \text{0}.
\]
\[
\text{error} := \text{0}.
\]
\[
\text{out} := \text{in}
\]

Operator of the CRC schematic

The control specification of the Control operator of the CRC schematic is given below:

\[
\text{%00 ResetCRC.}
\]
\[
\text{%01 ComputeCRC.}
\]
\[
\text{%10 CompareCRC.}
\]
\[
\text{%11 CRCout}
\]

Description of the function CompareCRC

"This function compares the received CRC sequence with the computed CRC sequence. If a mismatch occurs, this is indicated by the 'error' bit."

\[
\text{CRCO} := \begin{cases} 
\text{if 0: "Hold"} & \text{CRCI} \\
\text{if 1: "Rotate"} & \text{CRCI rol:1).} 
\end{cases}
\]

Modeling and implementation of a CAN controller

Description of the function ComputeCRC

"This function computes the CRC sequence according to the in the CAN draft standard presented implementation program."

CRC:=%0

Description of the function CRCout

"This function will rotate the content of the CRC register, most significant bit first, so the transmitter can send the calculated CRC sequence."

CRC:=CRCI at:14.

Description of the function ResetCRC

"This state reset the CRC register."

CRC:=%0

Error:=
(in at:1)
if 0: "No error"
%0
if 1: "Determine error"
((CRCI at:14)=-(in at:0)).
Operator of the Compare schematic

Description of the function Always

"The control waits till the bus has been sampled. There are two possibilities: A comparison with the value put on the bus has to be made, or the value monitored on the bus has to be passed to the output. In the first case, the value of the bus is compared to the value stored in the FirstValue register (the value put on the bus). The result is passed to the output:

\%100 = no match (dominant in stead of recessive)
\%101 = no match (recessive in stead of dominant)
\%11x = match.

In the second case, there is no value stored in the FirstValue register. The sampled bus value is passed to the output:

\%00x = can not occur
\%010 = the bus sample is dominant
\%011 = the bus sample is recessive.
"

out:= ((in2 at:1)=1)
  if0: "Bus sample not available yet"
  %000
  if1: "BEGIN(Bus sample available)" ((FVI at:1)=1)
    if0: "No value put on bus"
    (%0 width:1),in2
    if1: "Compare"
    (1 width:1),((in2 at:0)=(FVI at:0)),(in2 at:0)
  ) "END(Bus sample available)".

FVO:=
  ((in1 at:1)=1)
  if0: "BEGIN(No value put)" ((in2 at:1)=1)
    if0: "No sample"
    FVI
    if1: "Sample value available"
    %00
  ) "END(No value put)"
  if1: "Value put on bus"
in1

Operator of the Control schematic

Description of the function Always

"_Tbit: This expression is needed in several places"

_Tbit:=
  (((mode=%10)
    

174
Modeling and implementation of a CAN controller

"TbitO: Control of the T bit register. If T=1, the bitprocessor is transmitting. If T=0, the bitprocessor is receiving."

TbitO:

\( TbitO = \_Tbit / (\text{request} / \text{reqFFI}) \)

if 0: "BEGIN(No start of transmission)"

if 0:

\( \begin{cases} 
\text{error} = 0 & \text{if } (\text{mode} = 10) \land (\text{FSI} = 0011) \land (\text{busy} = 0) \\
\text{mode} = 11 & \text{if } (\text{FSI} = 1100) \land (\text{busy} = 0) \\
\end{cases} \)

if 0:

\( \begin{cases} 
\text{TbitI} & \text{if } 1 \\
\%0 & \text{if } 0 \\
\end{cases} \)

"END(No start of transmission)"

if 1:

\%1.

"reqFF0: Control of the request flipflop. When a request to send is initiated, this bit is set to one. When the bitprocessor is starting the transmission, this bit becomes zero again."

reqFF0:

\( \_Tbit \)

if 0: "reqFF:=1 if necessary" (request / reqFFI)

if 1: "reqFF:=0"

\%0.

"FSO: Control of the Frame State. This FrameState register keeps track of the place the frame is in."

FSO:

\( \begin{cases} 
(\text{mode} = 10) \lor (\text{mode} = 11) & \text{if } 0 \\
\%000 & \text{if } 1 \\
\end{cases} \)

\( \begin{cases} 
\text{FSI} = 0000 & \text{if } 0 \\
\text{FSI} = 0001 & \text{if } 0 \\
\text{FSI} = 0010 & \text{if } 0 \\
\text{FSI} = 0011 & \text{if } 0 \\
\text{FSI} = 0100 & \text{if } 0 \\
\text{FSI} = 0101 & \text{if } 0 \\
\text{FSI} = 0110 & \text{if } 0 \\
\text{FSI} = 0111 & \text{if } 0 \\
\end{cases} \)

\( \begin{cases} 
\text{FSI} = 1000 & \text{if } 0 \\
\text{FSI} = 1001 & \text{if } 0 \\
\text{FSI} = 1010 & \text{if } 0 \\
\text{FSI} = 1011 & \text{if } 0 \\
\end{cases} \)
if0: (  
(FSI=%1100)  
if0: (  
(FSI=%1101)  
if0: (  
(FSI=%1110)  
if0: "BEGIN(Superposition)" (  
(error=%10)  
if0: "BEGIN(No error)" (  
(busy)  
if0: "Not busy"  
%0000  
if1: "Busy"  
FSI  
) "END(No error)"
if1: "BEGIN(Error)" (  
(mode=%10)  
if0: "Passive error"  
(%1110 width:4)  
if1: "Active error"  
%1101  
) "END(Error)"
) "END(Superposition)"
if1: "BEGIN(Passive error)" (  
(error=%10)  
if0: "BEGIN(No error)" (  
(busy)  
if0: "Not busy"  
%1111  
if1: "Busy"  
FSI  
) "END(No error)"
if1: "BEGIN(Error)" (  
(mode=%10)  
if0: "Passive error"  
(%1110 width:4)  
if1: "Active error"  
%1101  
) "END(Error)"
) "END(Passive error)"
)
if1: "BEGIN(Active error or overload)" (  
(error=%10)  
if0: "BEGIN(No error)" (  
(busy)  
if0: "Not busy"  
%1111  
if1: "Busy"  
FSI  
) "END(No error)"
if1: "BEGIN(Error)" (  
(mode=%10)
if0: "Passive error" (%1110 width:4)  
if1: "Active error"  
%1101  
) "END(Error)"

) "END(Active error or overload)"

)  
if1: "BEGIN(Suspend transmission)" (  
(error=%10)  
if0: "BEGIN(No error)" (  
(busy)  
if0: "Not busy"  
%0000  
if1: "Busy"  
FSI  
) "END(No error)"

if1: "Error->SOF"  
%0001  
) "END(Suspend transmission)"

)  
if1: "BEGIN(Intermission)" (  
(error=%11)  
if0: "BEGIN(No overload)" (  
(busy)  
if0: "BEGIN(Not busy)" (  
(mode=%10)  
if0: "Error passive"  
FSI+1  
if1: "Error active"  
%0000  
) "END(Not busy)"

if1: "Busy"  
FSI  
) "END(No overload)"

if1: "Overload"  
%1101  
) "END(Intermission)"

)  
if1: "BEGIN(E0F)" (  
(error=%10)  
if0: "BEGIN(No error)" (  
(busy)  
if0: "Not busy"  
FSI+1  
if1: "Busy"  
FSI  
) "END(No error)"

if1: "BEGIN(Error)" (  
(mode=%10)  
if0: "Passive error"  
(%1110 width:4)
if1: "Active error"  
    %1101  
    ) "END(Error)"
)
"END(EOF)"
)
if1: "BEGIN(ACK delimiter)" (  
    (error=%10)
if0: "BEGIN(No error)" (  
        (busy)
        if0: "Not busy"
        FSI+1
        if1: "Busy"
        FSI
    ) "END(No error)"
if1: "BEGIN(Error)" (  
        (mode=%10)
        if0: "Passive error"  
            (%1110 width:4)
        if1: "Active error"  
            %1101
    ) "END(Error)"
)
"END(ACK delimiter)"
)
if1: "BEGIN(ACKRx)" (  
    (error%10)
if0: "BEGIN(No error)" (  
        (busy)
        if0: "Not busy"
        FSI+1
        if1: "Busy"
        FSI
    ) "END(No error)"
if1: "BEGIN(Error)" (  
        (mode=%10)
        if0: "Passive error"  
            (%1110 width:4)
        if1: "Active error"  
            %1101
    ) "END(Error)"
)
"END(ACKRx)"
)
if1: "BEGIN(ACKTx)" (  
    (error%10)
if0: "BEGIN(No error)" (  
        (busy)
        if0: "Not busy"
        FSI+2
        if1: "Busy"
        FSI
    ) "END(No error)"
if1: "BEGIN(Error)" (
(mode=%10)
  if 0: "Passive error"
    (%1110 width:4)
  if 1: "Active error"
    %1101
  ) "END(Error)"
)
"END(ACTx)"
)
if 1: "BEGIN(CRC delimiter)" (TbitI)

if 0: "BEGIN(Receive)" (error=%10)

  if 0: "BEGIN(No error)" (busy)
    if 0: "Not busy"
      FSI+2
    if 1: "Busy"
      FSI
  ) "END(No error)"

  if 1: "BEGIN(Error)" (mode=%10)
    if 0: "Passive error"
      (%1110 width:4)
    if 1: "Active error"
      %1101
  ) "END(Error)"
)
"END(Receive)"

if 1: "BEGIN(Transmit)" (error=%10)

  if 0: "BEGIN(No error)" (busy)
    if 0: "Not busy"
      FSI+1
    if 1: "Busy"
      FSI
  ) "END(No error)"

  if 1: "BEGIN(Error)" (mode=%10)
    if 0: "Passive error"
      (%1110 width:4)
    if 1: "Active error"
      %1101
  ) "END(Error)"
)
"END(Transmit)"
)
"END(CRC delimiter)"
)
if 1: "BEGIN(CRC)" (error=%10)

  if 0: "BEGIN(No error)" (busy)
if0: "Not busy"
  FSI+1
if1: "Busy"
  FSI
) "END(No error)"

if1: "BEGIN(Error)" (mode=\%10)
  if0: "Passive error" (%1110 width:4)
  if1: "Active error" %1101
) "END(Error)"

) "END(CRC)"

)
if1: "BEGIN(Data)" (error=\%10)

  if0: "BEGIN(No error)" (busy)
    if0: "Not busy"
      FSI+1
    if1: "Busy"
      FSI
  ) "END(No error)"

  if1: "BEGIN(Error)" (mode=\%10)
    if0: "Passive error" (%1110 width:4)
    if1: "Active error" %1101
  ) "END(Error)"

) "END(Data)"

)
if1: "BEGIN(Control)" (error=\%10)
  if0: "BEGIN(No error)" (busy)
    if0: "BEGIN(Not busy)" (NoData)
      if0: FSI+1
    if1: FSI+2
  ) "END(No error)"

  if1: "BEGIN(Error)" (mode=\%10)
    if0: "Passive error" (%1110 width:4)
    if1: "Active error" %1101
  ) "END(Error)"

) "END(Control)"
if1: "BEGIN(Arbitration)" ( ((error=%01)\/(error=%10))
  if0: "BEGIN(No error)" ( (busy)
    if0: "Not busy"
    FSI+1
    if1: "Busy"
    FSI
  ) "END(No error)"
if1: "BEGIN(Error)" ( (error=%01)
  if0: "BEGIN(No arbitration loss)" ( (mode=%10)
    if0: "Passive error"
    (%1110 width:4)
    if1: "Active error"
    %1101
  ) "END(No arbitration loss)"
  if1: "Arbitration loss"
  FSI
  ) "END(Error)"
) "END(Arbitration)"

if1: "BEGIN(SOF)" ( (error=%10)
  if0: "BEGIN(No error)" ( (busy)
    if0: "Not busy"
    FSI+1
    if1: "Busy"
    FSI
  ) "END(No error)"
if1: "BEGIN(Error)" ( (mode=%10)
  if0: "Passive error"
  (%1110 width:4)
  if1: "Active error"
  %1101
  ) "END(Error)"
) "END(SOF)"

if1: "BEGIN(Idles)" ( (ctrl at:1)
  if0: FSI
  if1: ( (ctrl at:0)
    if0: %0010
    if1: (%000 width:3),
    (request\/reqFFI)
  )
)
Modeling and implementation of a CAN controller

```
) "END(Idle)"
)
) "END(Active or passive error mode)".

"Rxstate: controls the state of the receiver.
The bitprocessor is ALWAYS receiving!"

Rxstate:=FSI.

"Txstate: Controls the state of the transmitter.
If the bitprocessor is transmitting, it is the
same as FSI. If the bitprocessor is receiving,
it is only needed for acknowledgement."

Txstate:=
(TbitI)
  if0: "BEGIN(Receive)"
    (FSI=1000)
      if0: %0000
      if1:
        FSI
    ) "END(Receive)"
  if1: "Transmit"
    FSI.

"CRCctrl: controls the state of the CRC schematic.
During SOF, arbitration, control, and data, the
CRC value is computed. During CRC sequence, a
transmitter gets the CRC value, bit by bit (CRCout)
and a receiver compares the CRC value, bit by bit
(CompareCRC). The CRC is being reset during all
other states."

CRCctrl:=
((FSI=0001)\(FSI=0010)\(FSI=0011)\(FSI=0100))
  if0: "BEGIN(Not compute CRC)"
    (FSI=0101)
  if0: "Idle->ResetCRC"
    %00
  if1: "BEGIN(Not resetCRC)"
    (TbitI)
      if0: "CompareCRC"
        (%10 width:2)
      if1: "CRCout"
        %11
    ) "END(Not resetCRC)"
  ) "END(Not compute CRC)"
  if1: "Compute CRC"
    %01.

"_Stuffing: indicates if bitstuffing has
to be carried out."

_stuffing:=
(FSI=0001)\(FSI=0010)\(FSI=0011)\(FSI=0100)\(FSI=0101).

"d20: In case the previous bit was a stuffbit
this signal sends the actual bit (the bit after
the stuff bit (delayed)"

\[ d_{20} = (T_{bit I} = 1) \land (DS_{status}) \land (\text{ctrl at:1}). \]

"Sctrl: bit stuffing control. The most significant
bit indicates whether or not the output bitstream
has to be stuffed. The least significant bit
indicates that the next bit has to be send after
a stuff bit has been inserted."

\[ S_{ctrl} := \text{Stuffing},d_{21}. \]

"DSctrl: destuffing control. Indicates if the input
bitstream has to be destuffed."

\[ DS_{ctrl} := \text{Stuffing}. \]

"NextBit: if the bitprocessor is transmitting, this
output signals when the next bit has to be send."

\[ \text{NextBit} :=
  (T_{bit I} = 0)
  \text{if0:}
    (DS_{status} = 0) \land (\text{ctrl at:1})
  \text{if1:}
    (\text{ctrl at:1}) \land
    (FSI = 0110) \land
    (\text{request} \land \text{reqFFI}) \land (\text{error at:1})\]

Operator of the FCE schematic

Description of the function Always

"_Overload: indicates an overload condition"

\[ _{\text{Overload}} := (\text{state} = 1011) \land (\text{comp} = 010). \]

"_ArbitrationLoss: indicates that the bitprocessor
has lost arbitration."

\[ _{\text{ArbitrationLoss}} :=
  (T) \land (\text{state} = 0010) \land (\text{comp} = 100). \]

"_ErrorDetected: indicates that an error has been
detected."

\[ _{\text{ErrorDetected}} :=
  (T)
  \text{if0:} \ "\text{BEGIN(Receive)}" \ (D_{error} = 1) \land
    (\text{state} = 0110) \land (\text{comp} = 100) \land
    (\text{state} = 1000) \land (\text{comp} = 101) \land
    (\text{state} = 1001) \land (\text{comp} = 100) \land
    (\text{state} = 1010) \land (\text{comp} = 100) \land
    (\text{state} = 1001) \land (\text{comp} = 111) \land \text{WCEI})
  \ "\text{END(Receive)}"
  \text{if1:} \ "\text{BEGIN(Transmit)}" \ (}
Modeling and implementation of a CAN controller

"error: indicates the error status. It can have the following values:
%00 no error
%01 arbitration loss
%10 error
%11 overload."

error:=
  (_ArbitrationLoss)
  if 0: "BEGIN(Not arbitration loss)"
  (_Overload)
  if 0: "Error or no error"
    _ErrorDetected,%0 width:1
  if 1: "Overload"
    %11
  ) "END(Not arbitration loss)"
  if 1: "Arbitration loss"
    %01.

"error0: this bit indicates that an error or an overload condition has occurred. It is used to reset the bitcounter in the receiver."

error0:=
  (_ArbitrationLoss)
  if 0: "BEGIN(Not arbitration loss)"
  (_Overload)
  if 0: "Error or no error"
    _ErrorDetected
  if 1: "Overload"
    %1
  ) "END(Not arbitration loss)"
  if 1: "Arbitration loss"
    %0.

"cnt10: counter 1. If the mode is bus off, it counts the number successive recessive bits. If the mode is power up, it count the number of recessive bits too. In case the mode is error active or error passive it is used as Transmit error counter."

cnt10:=
  (EMI=%00)
  if 0: 
    (EMI=%01)
    if 0: 
      (EMI=%10)
      if 0: "BEGIN(Error mode: error passive)" 
        cnt11

184
"END{Error mode: error passive}"

if1: "BEGIN(Error mode: error active)" (cnt1I
) "END(Error mode: error active)"

if1: "BEGIN(Error mode: bus off)" (cnt1I
) "END(Error mode: bus off)"

if1: "BEGIN(Error mode: power up)" (comp at:1)
if0: (cnt1I
if1: (((comp at:0)=%0)\/
(cnt1I=10))
if0: (cnt1I+1
if1: )
0
)
) "END(Error mode: power up)".

"cnt20: counter 2. If the mode is bus off, it counts the number sequences of 11 successive recessive bits. If the mode is power up, this counter is not used. In case the mode is error active or error passive it is used as Receive error counter."

cnt20:=
(EMI=%00)
if0: (EMI=%01)
if0: (EMI=%10)

if0: "BEGIN(Error mode: error passive)" (cnt2I
) "END(Error mode: error passive)"

if1: "BEGIN(Error mode: error active)" (cnt2I
) "END(Error mode: error active)"

if1: "BEGIN(Error mode: bus off)" (cnt2I
) "END(Error mode: bus off)"

if1: "BEGIN(Error mode: power up)" (cnt2I
) "END(Error mode: power up)".

"WCEO: controls the WaitCRCerror register. When a CRC error is detected, transmission of an error frame starts at the bit following the ACK delimiter, unless an error frame for another error condition has already been started."
WCEO:=
(CRCerror)
  if0: "BEGIN(No CRC error)" {
    (state=%1010)
    if0:
      WCEI
    if1:
      %0
  } "END(No CRC error)"
  if1: "CRC error" %1.

"EMO: controls the ErrorMode register. It can have the following values:
%00 power up
%01 bus off
%10 error active
%11 error passive."

EMO:=
(EMI=%00)
  if0: (
    (EMI=%01)
    if0: (
      (EMI=%10)
      if0: "BEGIN(Error mode: error passive)" ( EMII 
        ) "END(Error mode: error passive)"
      if1: "BEGIN(Error mode: error active)" ( EMII 
        ) "END(Error mode: error active)"
    )
  if1: "BEGIN(Error mode: bus off)" ( EMII 
  ) "END(Error mode: bus off)"
}
if1: "BEGIN(Error mode: power up)" ( ((cntII=10)/
  (comp=%011))
  if0:
    EMI
  if1: "Goto error active mode" %10
  ) "END(Error mode: power up)".

"ctrl: used to pass the result of the comparison to the Control schematic in the proper form."

ctrl:=
(EMI=%10)
  if0:
    %00
  if1: "BEGIN(Mode not error active)" ( comp at:2)
    if0:
      (comp from:0 to:1)
    if1:
      (comp from:1 to:2)
  ) "END(Mode not error active)"
Operator of the BackEnd schematic

Description of the function Always

\[
\text{NextByte0} := 0. \\
\text{pi} := 00000000. \\
\text{request} := 0. \\
\text{params} := 00000000. \\
\text{dataI0} := \text{dataI}. \\
\text{data20} := \text{data2I}.
\]