Fachhochschule
Dortmund
University of Applied Sciences and Arts

Documentation of the Canakari CAN Controller

by

Lucas Schreiter
Prof. Dr. Michael A. Karagounis
Alexander Walsemann
Philipp Ledüc
Aaron Beer

Table of Contents

1. Introduction

The Canakari CAN Controller is a greatly tested ISO 11898 compliant CAN controller which is implemented in VHDL and Verilog. Canakari was originally developed by Micheal A. Karagounis at the Cologne University of Applied Sciences as part of their diploma thesis in the year 2000. Over the years, numerous corrections and changes were made to the design, which were elaborated in various papers, to improve performance and compliance with the CAN ISO 11898 standard. This document is mainly based on the original thesis and is updated to meet the current implementation of the Canakari. The aim of this documentation is to present the current status and to provide the necessary information to gain deep understanding of the Canakari CAN controller. In addition, information is given on the configuration and use of the Canakari.

2. CAN Protocol

The CAN protocol (Controller Area Network) defines communication on a serial BUS system and is documented in all details by Bosch in the CAN2.0A or CAN2.0B standard (http://www.can.bosch.com). The Bosch specification was later taken up by the international standardization authority with minimal modifications to the ISO 11898 standard.

The CAN-BUS can have two states, either dominant or recessive. However, the CAN standard does not assign any physical states to these attributes, such as voltage level. It does not even specify an assignment to the logical values '1' or '0'. This gives the developer a lot of freedom in choosing the bus architecture (Wired- OR or Wired-AND) and the transmission medium. However, it must be ensured that if two units write to the bus simultaneously, the unit sending a dominant bit overwrites the recessive bit of the other unit. In most CAN implementations, the logical '0' overwrites the logical '1', which corresponds to an AND operation of all the outputs of the transmitting CAN units connected to the bus.

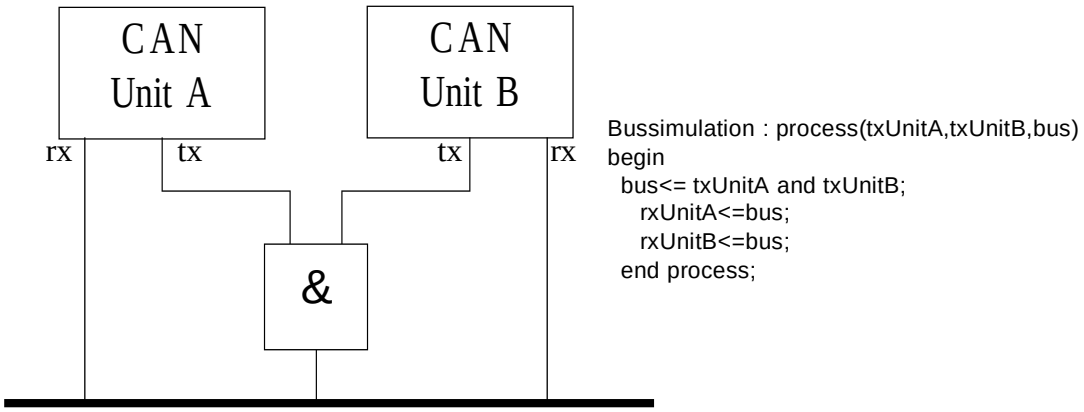


Figure 1: CAN Wired AND bus architecture and simulation of a bus in VHDL

The CAN-BUS also takes into account the finite propagation speed of a signal on the transmission medium by adequately defining the length of the bus and the data transmission speed. This ensures that all units connected to the bus can read the same value simultaneously within a certain tolerance range. Thus enabling the CAN devices to be synchronized with each other. They only have to listen to the communication on the bus and react to the data streams accordingly. Especially during the arbitration phase it is important that the competing CAN units are synchronized with each other. If several controllers want to use the bus for transmission at the same time, the CAN units can compare their transmitted bit patterns with the received ones and recognize whether a recessive bit has been overwritten by a dominant bit of another participant or not. All CAN modules for which this is the case, withdraw from the bus and become recipients of the message sent by the unit with the highest priority until the next arbitration phase. The CAN unit that has won the arbitration does not register the competitor's attempt to obtain the bus, as it sends its data without an error by overwriting the recessive bits of the competitors with dominant ones. The message sent by the unit with the highest priority is preserved and is not destroyed as it would be the case with the CSMA/CD arbitration approach used by Ethernet. Because of this arbitration mechanism, unnecessary collisions are avoided and the bandwidth is used effectively and distributed to the bus participants according to their priority. Due to these properties, the CAN protocol is particularly well suited for applications that require deterministic behavior and high data integrity.

2.1 CAN Frame Format

There are two types of CAN frames. On the one hand, there are frames for sending data, called "Data Frame", and on the other, frames for requesting data, called "Remote Frame". In rough terms, a CAN frame consists of up to seven segments. The frame starts with a mandatatory dominant bit, which in Figure 2 is called "Start of Frame" (SOF) from the CAN standard. The SOF bit marks the start of new transmission and is used by all receiving units of the bus to synchronize with the sending unit. Following the "SOF" bit, the bits of the arbitration field are sent. The values of these bits are decisive for the probability of a successful arbitration for the CAN unit. A high-priority message must contain more dominant bits at the bit positions of higher significance of the arbitration field than other participants since all frame segments are sent with the significant bits first. The exact number of bits used for arbitration depends on whether the CAN protocol 2.0A or 2.0B is used. In the control field, the size of the data field is set and in addition the number of bits in the arbitration field and frame type, i.e. whether it is a data or remote frame, are defined by flags. The data field can contain up to eight bytes of data. However, it can also be empty if a remote frame is sent. The data field is followed by the CRC field, which contains the checksum calculated from all previous fields including the SOF bit. In the ACK field, all receiving units on the bus acknowledge (or not) that the message has been received without error. Each CAN frame is terminated with the eight recessive bits of the "End of Frame" frame.

Figure 2: CAN Frame Format [MotorolaCan]

2.1.1 Arbitration Field

The exact structure of the arbitration field is dependent on the version of the protocol. Controllers designed according to the CAN2.0A standard use 11 bits in the arbitration field to identify the frame. CAN2.0B uses 29 bits to identify a CAN message. The CAN standard specifies that CAN2.0B controllers must also be able to receive and transmit CAN2.0A frames. However, CAN2.0A controllers are not required to be able to receive and transmit CAN2.0B frames, although they must be able to distinguish CAN2.0B messages from transmission errors. In addition to the frame identifier, the arbitration field also includes the RTR bit (Remote Request). The RTR bit informs the receiver whether it is receiving a "data frame" or a "remote frame". "Remote frames" are identified by a recessive bit and "data frames" by a dominant bit. This conveniently gives data frames a higher priority than remote frames that carry the same identifier.

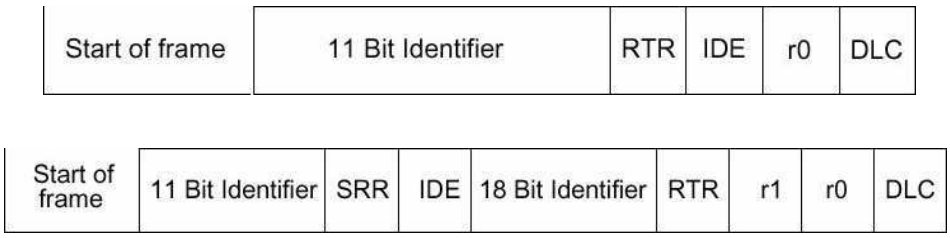


Figure 3: Arbitration and Control Field [MotorolaCan]

The IDE (Identifier Extension) bit is part of the control field and is located in CAN2.0B frames between the basic and extended part of the identifier. An extended frame is marked by a recessive IDE bit. Standard frames have a dominant IDE bit and thus a higher priority than extended frames, which have the same base identifier. In extended frames, the SRR bit (Substitute Remote Request) takes the place of the RTR bit between the base identifier and the IDE bit. The SRR bit is always recessive and has no other function than to replace the RTR bit, as the name suggests. When interpreting the data stream, it is necessary to have received the IDE bit to reliably and accurately detect whether it is an RTR frame in the standard format and an RTR bit has been read as the 13th bit, or whether it is an extended frame and the 13th bit is merely the SRR bit. In a CAN2.0B frame, the RTR bit is only sent after the extended identifier as the 33rd bit.

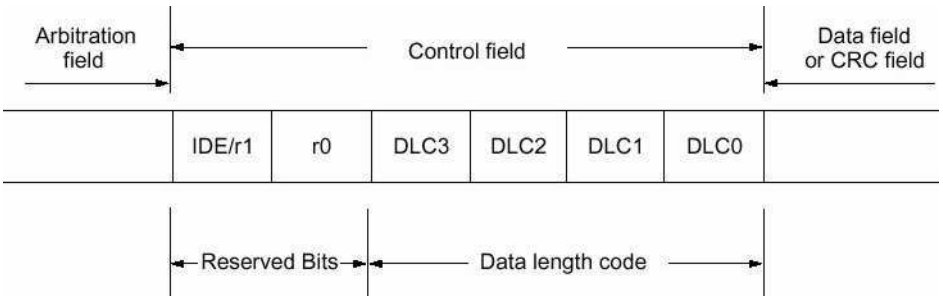


Figure 4: Control Field [MotorolaCan]

2.1.2 Control Field

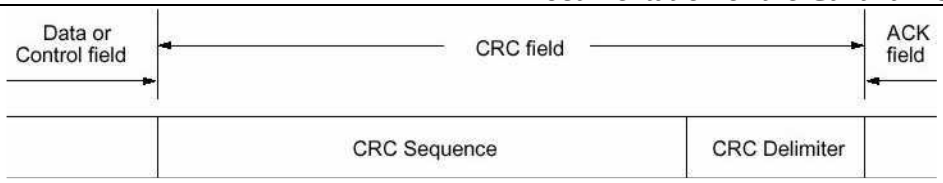
The control field consists out of 6 bits. For standard frames, the first bit (IDE) indicates by its dominant value that it is not an extended frame. The second bit is reserved and is currently not used. For the extended frame, the first bit is also reserved because the IDE bit has already been used after the base identifier to initiate the transmission of the extended identifier. The last 4 bits of the control field specify the number of data bytes in BCD code. Since a maximum of eight bytes of data can be transmitted, numbers greater than eight are ignored and not treated as errors. Remote frames can also contain numbers in the DLC (Data Length Code) part of the control field, even though they do not contain any data. It is left to the higher OSI layers of the receiver in which way this information is interpreted.

2.1.3 Data Field

The data field can consist of zero to eight bytes of data, all sent with the most significant bit first. The number of bytes in the data field should match the number in the DLC part of the control field.

2.1.4 CRC Field (Cyclical Redundancy Check)

The CRC field stores the CRC sequence calculated from the Start of Frame bit, the Arbitration Field, the Control Field, and the Data Field. The CRC field consists of 15 bits of CRC sum and the recessive CRC delimiter. The bits of the above fields are interpreted as a polynomial which have "1" or "0" as coefficients and is divided by the polynomial given below. The numerator of the fractional rational part resulting from the division is put into the first 15 bits of the CRC field and sent over the bus.



$$x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1$$

Figure 5: CRC Field and CRC Polynom

To illustrate how an implementation of the CRC algorithm in VHDL could look like, we consider an exemplary division of the polynomial formed from the bits "010111001" by the generator polynomial with the bit pattern "111".

```
Step 1  010111001 : 111 = 0111110
        000
Step 2   101
         111
Step 3    101
          111
Step 4     101
           111
Step 5      100
            111
Step 6       110
             111
Step 7        011
              000
               11
```

Figure 6: Polynomial division to illustrate the CRC algorithm

The polynomial division of bit patterns is very similar to an ordinary polynomial division. The only difference is that it is calculated with modulo 2 and thus an addition gives the same result as a subtraction. As is seen in Figure 7, the bits of the generator polynomial is anded with the most significant bit of the data polynomial and written under the bits of the data polynomial. If the most significant bit of the data polynomial is a logical "1", the generator polynomial is written under the bits of the data polynomial, otherwise, as in step 1, only the corresponding number of zeros is placed under the generator polynomial. The usual subtraction of the coefficients can be replaced by an XOR operation due to the modulo 2 calculation. Afterwards, a shift operation takes place. This calculation is continued until the last bit has been reached. In addition a number of zeros equal to the amount of bits reserved for the CRC sum is handeled in the same way as described above. The remaining division equates to the CRC sum. The functionality of the shifting and the XOR operation can easily be mapped in hardware.

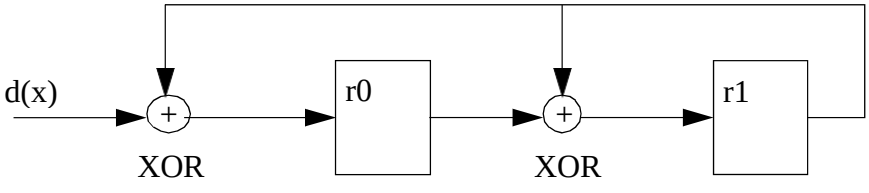


Figure 7: Realization of a 2 bit wide CRC register

```
entity rcrc is
 port( bitin : in bit;
         activ : in bit;
        reset : in bit;
         rest  : out bit_vector(14 downto 0));
end rcrc;

architecture behv of rcrc is
begin
 process(activ,reset)
  variable polynom : bit_vector(14 downto 0);
  variable crcreg : bit_vector(14 downto 0);
  variable buf1,buf2 : bit;
 begin
  if reset= '1' then
   polynom:="100010110011001";
   crcreg :="000000000000000";
   buf1:='0';
   buf2:='0';
  elsif activ'event and activ='1' then
   buf1:=bitin;
   for i in 0 to 14 loop
    buf2:=crcreg(i);
    crcreg(i):=buf1 xor (crcreg(14) and polynom(i));
    buf1:=buf2;
   end loop;
  rest<=crcreg;
  end if;
 end process;
end;
```

Figure 8: CRC algorithm in VHDL with CAN polynomial

The VHDL code shown in Figure 8 implements the CRC algorithm and is used in the CAN model presented in this work. As can be seen, the generator polynomial is stored in the variable polynom and is anded with the most significant bit of the CRC register. With each new run, an XOR operation is performed on all bits of the CRC register, including the new bit which is to be shifted into the CRC register. According to the CAN standard, the CRC register contains the CRC checksum after the last data bit has been sent. In case of the reception the CRC sum should become zero when all bits have been received.

2.1.5 Acknowledge Field

The acknowledge field consists of two recessive bits. All CAN nodes that have received a message without an error, overwrite the first recessive bit of the acknowledge field with a dominant bit. When a transmitting CAN node does not register a dominant bit during the acknowledge slot, this means that either no CAN unit is connected to the bus which has received the message without error or that there is no second CAN unit on the bus that cannot acknowledge the message with an ACK bit. If the transmitting unit receives the dominant bit, it sends the "End of Frame" flag after the ACK delimiter, which comprises eight recessive bits.

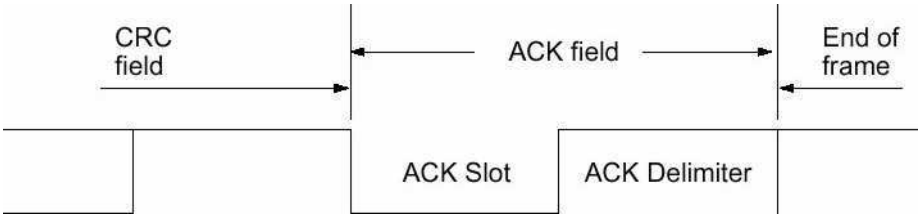


Figure 9: Acknowledge Field [MotorolaCAN]

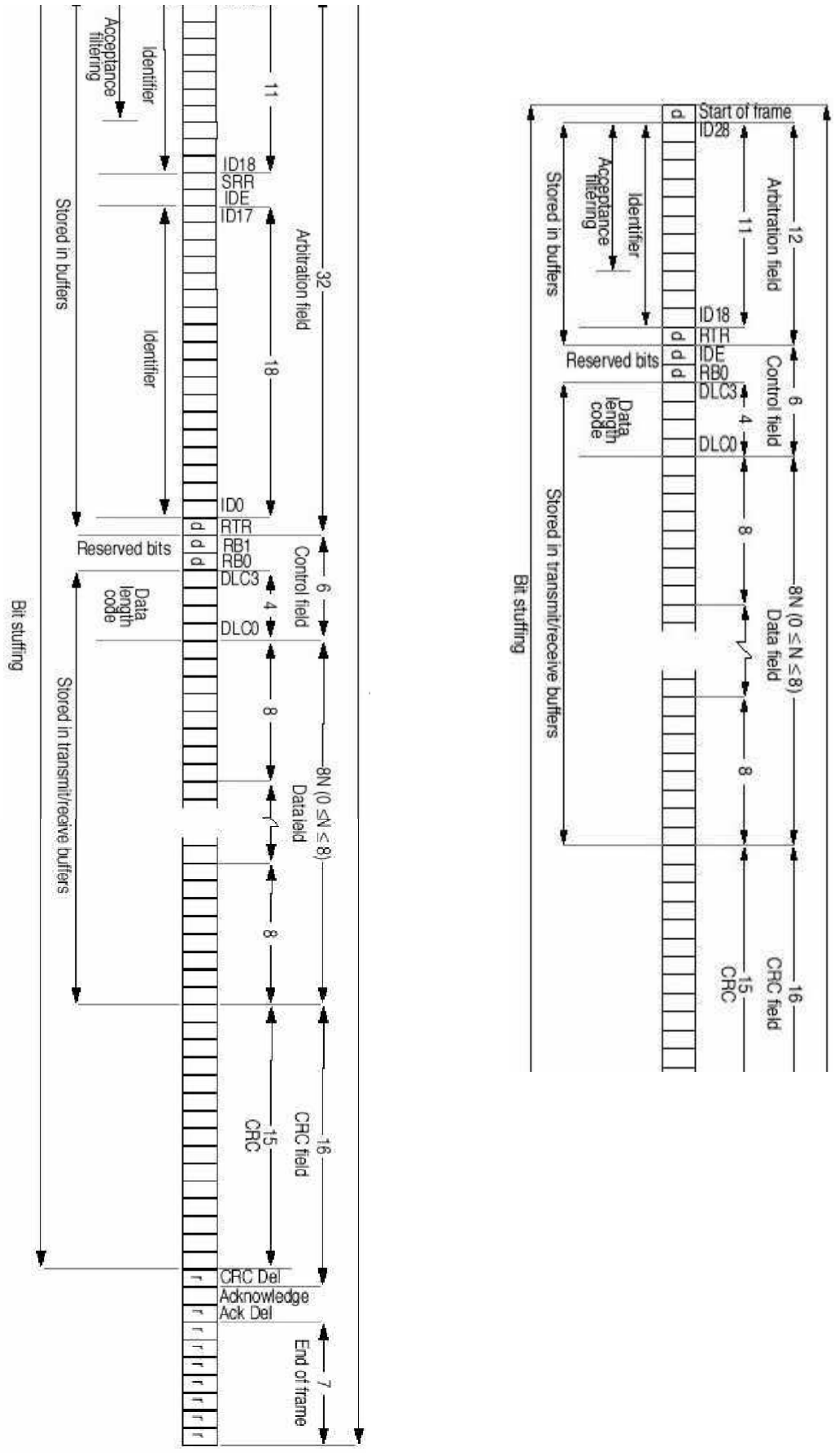


Figure 10: Overview of the CAN frame format

```
entity encapsulation is
   port(identifier                                              :  in bit_vector( 28 downto 0);
        data1,data2,data3,data4,data5,data6,data7,data8 : in bit_vector( 7 downto 0);
        extended,remote,activ,reset                             :  in bit;
        datalen                                                  : in bit_vector(  3 downto 0);
        message                                                 : out bit_vector(102 downto 0);
        meslen                                                   : out integer range 0 to 64);
end encapsulation;

architecture behv of encapsulation is
begin
  idres : process(activ,reset)
  variable state : bit_vector( 1 downto 0);
  variable mesbuf: bit_vector(102 downto 0);
  variable len: integer range 0 to 64;
 begin
  if reset='1' then
   state :="00";
   mesbuf :=(others => '0');
   len  :=0;
   message<=(others => '0');
   meslen <=0;
  elsif activ'event and activ='1'then
   mesbuf(102 downto 102) := "0";    -- start of frame
   mesbuf(101 downto  91) := identifier (28 downto 18);
                             -- standard identifier
   len := bitv_to_int(datalen);
   len := len * 8;          -- Datenbitanzahl
   state := extended & remote;
   case state is
                             -- standard data frame
   when "00" => mesbuf(90 downto 88) := "000";
                             -- RTR=IDE=R0=d
                mesbuf(87 downto 84) := datalen;
                             -- DLC = datalen
                mesbuf(83 downto 20) := data1 & data2 & data3 & data4 &
                             data5 & data6 & data7 & data8;
                             -- standard remote frame
   when "01" => mesbuf(90 downto 88) := "100";
                             -- RTR=r; IDE=R0=d
                mesbuf(87 downto 84) := "0000";
                             -- DLC = datalen
                len:=0;              -- independent of the value of datalen
                             -- extended data frame
   when "10" => mesbuf(90 downto 89) := "11";
                             -- SRR=r IDE=r
                mesbuf(88 downto 71) := identifier(17 downto 0);
                             -- extended identifier
                mesbuf(70 downto 68) := "000";
                             -- RTR=r1=r0=d
                mesbuf(67 downto 64) := datalen;
                             -- DLC= datalen
                mesbuf(63 downto 0) := data1 & data2 & data3 & data4 &
                             data5 & data6 & data7 & data8;
                             -- extended remote frame
   when "11" => mesbuf(90 downto 89) := "11";
                             -- SRR=r IDE=r
                mesbuf(88 downto 71) := identifier(17 downto 0);
                             -- extended identifier
                mesbuf(70 downto 68) := "100";
                             -- RTR=r r1=r0=d
                mesbuf(67 downto 64) := "0000";
                             -- DLC= datalen
                len:=0;              -- independent of the value of datalen
  when others =>
  end case;
  message <= mesbuf;
  meslen <= len;
 end if;
 end process;
end;
```

Figure 11: Encapsulation of the transmission data into the CAN frame format

A possible realization for the encapsulation of the transmission data into the CAN frame format would be to instruct the protocol unit to fetch the required data, such as message identifiers and data bytes, one by one from the registers which are used to communicate with the CPU and to place them into the shift register along with control data to transmit them.

However, constant access to the registers is costly and may endanger the integrity of the data if the CPU changes the register contents before a message has been completely sent. For this reason, it can be beneficial to put a logic unit between the data registers and the transmit shift register, which takes over the encapsulation of the transmit data and loads the complete CAN message with all control data and protocol overhead into the shift register at once. The VHDL source code shown in Figure 11 describes the function of such a unit. The encapsulation unit receives as input signals all necessary data of the message to be sent and outputs the complete CAN message. Depending on the frame type, the program jumps to a branch of the case tree and places the data into the correct position. The branching decision in the case statement is performed based on the IDE and RTR flags set by the user. Four cases must be considered. If neither the extended nor the remote flag is set, a standard data frame is set. If the extended flag is not set, but the remote flag is set, a standard remote frame is constructed. Similarly, a distinction is made between Extended Data Frame and Extended Remote Frame. Since all frames that are sent must contain at least the SOF bit and base portion of the identifier, they get assigned right at the beginning, regardless of which branch of the case statement is executed.

Furthermore, the encapsulation unit also provides the protocol unit with the number of data bits to be sent. This number is used in the protocol unit to stop data transmission in time and continuing with the CRC checksum. For this reason, all eight bytes fed to the encapsulation unit are inserted into the message frame, even if the actual number of bytes containing meaningful data is less. The risk that data garbage of the remaining unused bytes is sent over the bus is nonexistent because the counter of the data bits has already run out by then. The decapsulation of the data is again done in a module explicitly designed for this process.

However, the protocol unit must be able to determine the frame type and the length of the payload portion of the received message while the current bitstream is still being received to calculate at what time the CRC sequence will be placed on the bus and when the acknowledge bit must be sent. Therefore, the protocol unit cannot wait for the final reception and decapsulation of the object and must extract bits of the control data and flags from the bitstream synchronously.

The data obtained by the protocol supports the decapsulation process. For example, from the number of user data bytes it can be concluded, on which position in the shift register the control field and the identifier are located. Figure 33 shows the VHDL source code of the decapsulation unit. According to the number of available user data, the corresponding branch of the case instruction is executed, in which the contents of the receive shift register are read out, starting with low-order bits and ascending to high-order bits. The information whether it is an extended or a standard frame, which is also provided by the protocol unit, is used to decide whether only the basic part of the identifier has to be read or whether the extended identifier range has to be considered as well. The obtained data is fed tot he corresponding signals and forwarded to the memory registers to enable access by the CPU and thus the user's application

```
entity decapsulation is
   port(
    activ,reset: in bit;
    message: in bit_vector(102 downto 0);
    meslen :  in integer range 0 to 64;
    ext,rtr       : in bit;
    identifier : out bit_vector( 28 downto 0);
    data1  :  out  bit_vector(  7  downto  0);
    data2  :  out  bit_vector(  7  downto  0);
    data3  :  out  bit_vector(  7  downto  0);
    data4  :  out  bit_vector(  7  downto  0);
    data5  :  out  bit_vector(  7  downto  0);
    data6  :  out  bit_vector(  7  downto  0);
    data7  :  out  bit_vector(  7  downto  0);
    data8  :  out  bit_vector(  7  downto  0);
    extended : out bit;
    remote    : out bit;
    datalen : out bit_vector( 3 downto 0));
end decapsulation ;
architecture behv of decapsulation is
begin
 idres : process(activ,reset)
  variable state : bit_vector( 1 downto 0);
 variable mesbuf: bit_vector(127 downto 0);
 begin
  if activ'event and activ='1'then
   case meslen is
   when  0 => data1<="00000000";
              data2<="00000000";
              data3<="00000000";
              data4<="00000000";
              data5<="00000000";
              data6<="00000000";
              data7<="00000000";
              data8<="00000000";
              datalen<=message(3 downto 0);
              if ext='0' then
              identifier(28 downto 18)<=message( 17 downto  7);
               identifier(17 downto  0)<=(others => '0');
              else
               identifier(17 downto  0)<=message( 24 downto   7);
             identifier(28 downto 18)<=message( 37 downto 27);
              end if;
   when  8 => data1<=message( 7 downto  0);
              data2<="00000000";
              data3<="00000000";
              data4<="00000000";
              data5<="00000000";
              data6<="00000000";
              data7<="00000000";
              data8<="00000000";
              datalen<=message( 11 downto  8);
              if ext='0' then
             identifier(28 downto 18)<=message( 25 downto 15);
               identifier(17 downto  0)<=(others => '0');
              else
              identifier(17 downto  0)<=message( 32 downto  15);
             identifier(28 downto 18)<=message( 45 downto 35);
              end if;
   when 16 => data1<=message(15 downto  8);
              data2<=message( 7 downto  0);
              data3<="00000000";
              data4<="00000000";
              data5<="00000000";
              data6<="00000000";
              data7<="00000000";
              data8<="00000000";
              datalen<=message( 19 downto  16);
              if ext='0' then
             identifier(28 downto 18)<=message( 33 downto 23);
               identifier(17 downto  0)<=(others => '0');
              else
              identifier(17 downto  0)<=message( 40 downto  23);
             identifier(28 downto 18)<=message( 47 downto 37);
              end if;
   when 32 => data1<=message(31 downto 24);
              data2<=message(23 downto 16);
              data3<=message(15 downto  8);
              data4<=message( 7 downto  0);
              data5<="00000000";
              data6<="00000000";
              data7<="00000000";
              data8<="00000000";
              datalen<=message(35 downto 32);
              if ext='0' then
              identifier(28 downto 18)<=message( 49 downto  39);
              identifier(17 downto  0)<=(others => '0');
             else
              identifier(17 downto  0)<=message( 56 downto  39);
              identifier(28 downto 18)<=message( 69 downto  59);
             end if;
  when 40 => data1<=message(39 downto 32);
             data2<=message(31 downto 24);
             data3<=message(23 downto 16);
             data4<=message(15 downto  8);
             data5<=message( 7 downto  0);
             data6<="00000000";
             data7<="00000000";
             data8<="00000000";
             datalen<=message(43 downto 40);
             if ext='0' then
              identifier(28 downto 18)<=message( 57 downto  47);
              identifier(17 downto  0)<=(others => '0');
             else
              identifier(17 downto  0)<=message( 64 downto 47);
              identifier(28 downto 18)<=message( 77 downto 67);
             end if;
  when 48 => data1<=message(47 downto 40);
             data2<=message(39 downto 32);
             data3<=message(31 downto 24);
             data4<=message(23 downto 16);
             data5<=message(15 downto  8);
             data6<=message( 7 downto  0);
             data7<="00000000";
             data8<="00000000";
             datalen<=message( 51 downto  48);
             if ext='0' then
              identifier(28 downto 18)<=message( 65 downto  55);
              identifier(17 downto  0)<=(others => '0');
             else
              identifier(17 downto  0)<=message( 72 downto  55);
              identifier(28 downto 18)<=message( 85 downto  75);
             end if;
  when 56 => data1<=message(55 downto 48);
             data2<=message(47 downto 40);
             data3<=message(39 downto 32);
             data4<=message(31 downto 24);
             data5<=message(23 downto 16);
             data6<=message(15 downto  8);
             data7<=message( 7 downto  0);
             data8<="00000000";
             datalen<=message( 59 downto  56);
             if ext='0' then
              identifier(28 downto 18)<=message( 73 downto  63);
              identifier(17 downto  0)<=(others => '0');
             else
              identifier(17 downto  0)<=message( 80 downto  63);
              identifier(28 downto 18)<=message( 93 downto  83);
             end if;
  when 64 => data1<=message(63 downto 56);
             data2<=message(55 downto 48);
             data3<=message(47 downto 40);
             data4<=message(39 downto 32);
             data5<=message(31 downto 24);
             data6<=message(23 downto 16);
             data7<=message(15 downto  8);
             data8<=message( 7 downto  0);
             datalen<=message( 67 downto  64);
             if ext='0' then
              identifier(28 downto 18)<=message( 81 downto  71);
              identifier(17 downto  0)<=(others => '0');
             else
              identifier(17 downto 0)<=message( 88 downto 71);
             identifier(28 downto 18)<=message(101 downto 91);
             end if;
   when others =>
  end case;
  extended<=ext;
  remote<=rtr;
  end if;
 end process;
end;
```

Figure 12: Decapsulation of the transmission data out of the CAN frame format

2.2 Error Handling

The CAN protocol has a very detailed error handling, which is designed to keep the synchronization of the CAN unit on the bus despite the occurrence of errors. Special measures are taken to prevent a single defective CAN node from paralyzing the entire network communication. The CAN protocol distinguishes between five different types of error: bit error, stuffing error, CRC error, form error, and acknowledgment error. These errors can occur at the same time and are not mutually exclusive.

2.2.1 Error Types

Stuffing Error
The SOF, arbitration, control, data, and CRC fields are coded using the stuffing method. This means that after a certain number of equal bits, in this case, five, a stuffing bit with the reversed bit value is inserted into the bitstream. The receiver examines the received bits and, after receiving five sequential bit of same values, removes the next bit from the bitstream, checking whether it is of opposite value to the five previous bits. If this is not the case, it detects a stuffing error and triggers error handling.

Bit Error
A transmitting CAN unit listens to the data traffic on the bus and compares it with the bits it sends. If the controller notices a difference in the bitstream an error gets registered. An exception to this, however, is the intentional or tolerated overwriting of a transmitted recessive bit by a dominant bit of another CAN unit, which can occur during the arbitration process or in the acknowledge field. In the arbitration field, an error bit indicates that the CAN node has lost arbitration and must withdraw from the bus to become the recipient of a message. In the acknowledge field, all CAN controllers connected to the bus confirm error-free reception of the transmitted message by overwriting the recessive bit of the transmitting unit with a dominant one during the acknowledge slot.

CRC Error
A CRC error occurs when the CRC sum calculated by the receiving unit differs from the CRC sum received.

Form Error
A form error occurs when bits that have a fixed or static structure in the CAN frame deviate from it. If, for example, a dominant bit is received in place of the recessive ACK delimiter or the recessive CRC delimiter, a form error has happeneds. This also applies to the bits of the End of Frame field, except for the last bit, which receives special treatment and is discussed in the section on the Overload case.

Acknowledge Error
If during the acknowledge slot the recessive bit of the sending CAN controller is not overwritten by any other unit, then this is an indication that no CAN node was able to read the message without errors or that no other CAN unit is connected to the bus to acknowledge the message. Both will result in an acknowledge error. Each time an error occurs, the receive or transmit counter will increment. If the counters reach certain limits, the controller switches to another operating state. As long as both counters remain below 128, the controller is in the "Error Active" state.

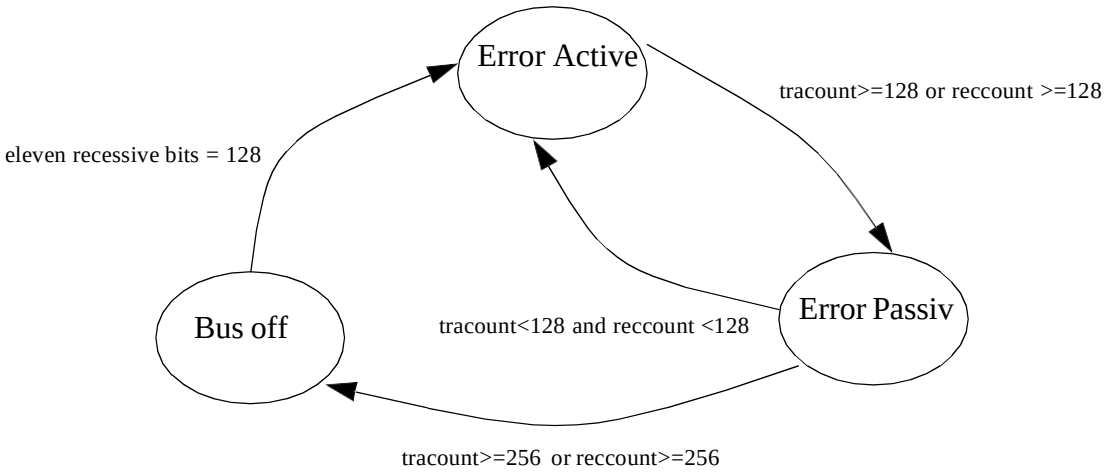


Figure 13: Error States

If one of the two counters reaches the value 96, a warning signal is sent to alert the user of a severely disturbed bus. If one of the two counters reaches or exceeds the value 128, the controller enters the "Error Passive" state, which affects handling. Each time a message is successfully sent or received, the counters are decremented again so that a controller can return to the Error Active state when both counters have fallen below the limit of 128. However, if the bus still does not allow error-free transmission, the error counters are incremented further until one of the two counters reaches the value 256. The controller then stops transmitting on the bus until it has read 128 times eleven consecutive recessive bits.

2.2.2 Error Frame

If a controller detects one of the five errors mentioned above, it sends out an Error Frame. An error frame consists of two parts, the error flag, and the error delimiter. The structure of the error flag depends on the error state of the CAN controller. An Active Error Flag consists of six dominant bits and a Passive Error Flag consists of 6 recessive bits. If a controller sends an Active Error Flag due to an error, then it is possible that the error was only noticed by this controller and the other CAN units are only made aware of the error by the violation of the Stuff rule or the causing of a FORM error by the Error Flag itself. The other bus nodes then also start sending an error flag, so that the dominant active error flag, which was originally 6 bits wide, can be doubled, up to 12 bits, by superpositioning several such flags. A CAN controller in Error Passive state tries to send a Passive Error Flag consisting of six recessive bits until it can detect six consecutive bits of the same value on the bus, regardless of whether they are dominant or recessive bits.

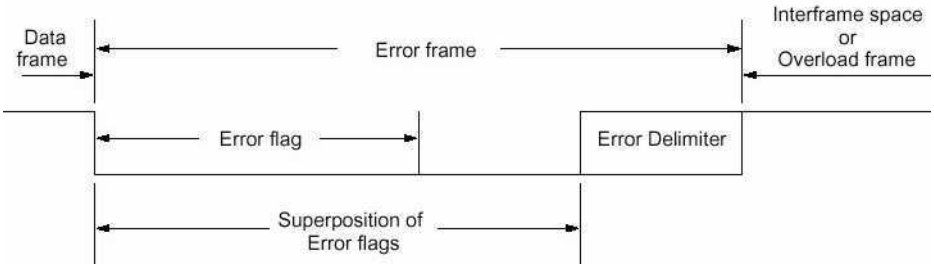


Figure 14: Error Frame

If, for example, the CAN unit that has won the arbitration and is therefore allowed to transmit, detects an error and is furthermore in the Error Passive state, then it can generally transmit six recessive bits and thus send its Error Passive flag. The other instances detect a stuffing rule violation or a form error and send error flags as well. If an Error Passive instance is the recipient of a message and it detects an error, it again attempts to write six recessive bits to the bus. If the other units have also detected the error and at least one of these instances is in the Error Active state, the recessive bits of the Error Passive Controller are overwritten by the Error Active instance. However, the Error Passive unit does not interpret this as an error but rather waits until it has read six same value bits (dominant in this case) from the bus and then terminates the Error Passive flag. However, if the Error Passive unit is the only receiver which has detected an error because the error has a local cause, e.g. a defect at the receiving part of the controller, the recessive bits of the Error Passive flag is overwritten by the data stream of the emitting controller and the Error Passive unit can complete its Error Passive flag at the End of Frame of the current message at the earliest, because only at this point six recessive bits are on the bus. An active error flag or passive error flag is followed by an error delimiter consisting of eight recessive bits. After a controller has sent an error flag, it writes recessive bits to the bus until it can read a recessive bit from the bus, i.e. until no other controller overwrites the recessive bit with a dominant one. If a recessive bit was detected, seven more are written to the bus and the error delimiter, and thus the complete error frame is terminated.

2.2.3 Overload Frame

In special cases, an overload frame is sent instead of an error frame when a form error occurs. If for example the last bit of the End Of Frame field is overwritten by a dominant bit of another unit, this is not interpreted as an error case, but as an overload situation. An overload situation can also be artificially induced by a higher OSI layer, e.g. to bypass bottlenecks in the processing of received data. The transmission of an Overload frame can only ever happen at certain times. If a controller is forced by an internal cause to delay the reception of further messages by sending overload frames, it must wait until the current message has been sent or received and only emit the overload frame after the last bit of the EOF field, or if an error active or error passive frame has been received or sent at the time when the overload frame should be sent, the overload flag may only start after the last bit of the error delimiter. According to the CAN standard, a maximum of two such overload frames may be sent to delay bus traffic. The Overload frame has the same appearance as an Error Active frame. It consists of the Overload Flag and the Overload Delimiter. The overload flag is composed of six dominant bits and the overload delimiter is generated by stringing together eight recessive bits. Since overload frames can overlap just like error frames, recessive bits are sent after transmission of the overload flag until the first recessive bit on the bus has been detected. Then seven more recessive bits are sent after them.

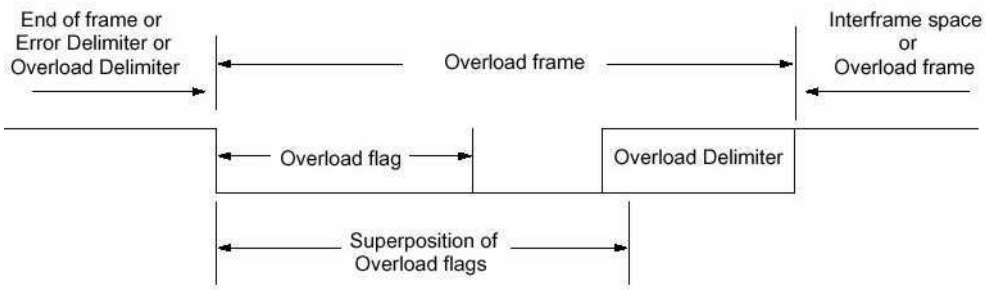


Figure 15: Overload Frame

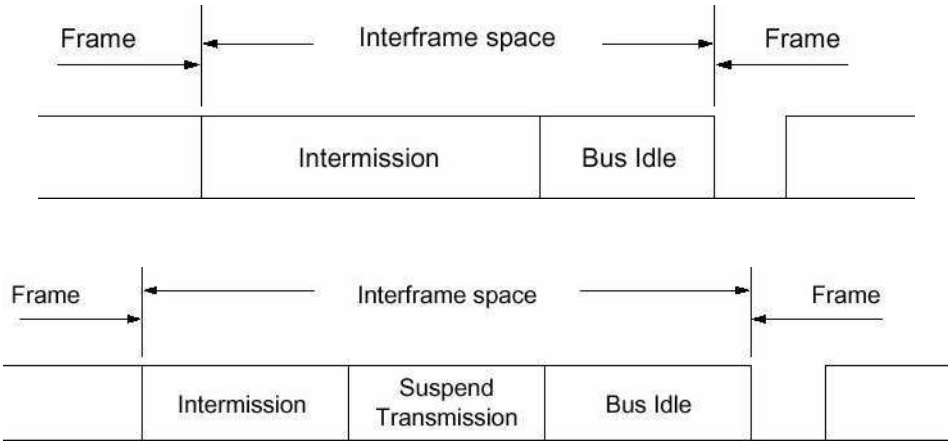


Figure 16: Intermission Field

2.3 Interframe Space

Data and remote frames are inserted through a special interframe space. This means that after a frame of any type, such as data, remote, error, or overload frame, the CAN controller must first send an intermediate field before another data or remote frame can be sent. Error frames do not require such a space and generally begin one bit after an error condition is detected. Overload frames are also sent without a previous intermediate field and are not separated by such a field when a controller sends two overload frames in succession. The intermediate field consists of a recessive bit triple. Only after these bits, a new message may be initiated by a SOF bit. If no message could be sent, the bus remains free (Bus Idle). If a CAN node is in the Error Passive state, it must place another eight recessive bits in the form of the Suspend Transmission Flag on the bus after the actual Intermission intermediate field before it can start sending a message. If the controller detects a dominant bit on the bus while it is transmitting the suspend transmission field, it enters receive mode, even though it might have a transmit request of its own.

As a result, Error Active units are given priority over Error Passive units in the distribution of bus resources. The intention behind this approach is to minimize the probability of disturbing units blocking or slowing down bus communication by multiple erroneous transmission attempts. If a dominant bit is observed on the bus during the intermission field, an overload frame is sent to prevent the early transmission attempt of a badly synchronized CAN unit. An exception is made for the third and last bit of the Intermission field. If this bit is overwritten by a bus unit, all controllers interpret this bit as Start of Frame and either enter the arbitration sequence to participate in the competition for the bus if they wish to send a message, or become recipients of the message from the node that sent the premature SOF bit if there is no send request of their own.

2.4 Error Rules

As described above, two error counters are used for the transmit and receive side to determine the error susceptibility of the bus. The error counters are incremented after each detected error case, whereby the rules given below must be observed.

Rule 1
If a CAN node detects an error while receiving a message, it increments its receive error counter by one. This does not apply if it is a bit error that occurred while receiving an Error Active Flag or Overload Flag. In this case, rule 5 comes into effect.

Rule 2
If a receiver receives a dominant bit as the first bit after sending an Error Flag, it increments its receive error counter by eight. The receipt of a dominant bit as the first bit after the transmission of an Error Flag is an indication that the error detected by the CAN unit has not been registered by the other bus participants and that only the transmission of the Error Flag has led the communication partners into the error state. This is an indication of a local error of the unit sending the error flag.

Rule 3
If an instance detects an error during the transmission of its message and starts sending an error flag, the send error counter is incremented by eight. Two exceptions exist for this rule:

> **Exception 1**
> When a transmit controller is in the Error Passive state and notices an Acknowledge Fault, it does not increment the error counter unless it reads a dominant bit from the bus that overwrites its recessive bits while sending the Passive Error Flag.
>
> **Exception 2**
> The transmit controller does not increment its receive counter if a stuff error occurs during the arbitration phase because a bit that must be recessive due to the stuffing rule has also been sent recessively but has been overwritten by a dominant bit.

Rule 4
A transmitter increments its transmit error counter by eight if it registers a bit error during an Active Error or Overload flag.

Rule 5
A receiver increments its receive error counter by eight if it registers a bit error during an Active Error or Overload Flag (see Rule 1).

Rule 6
By overlaying several Error Flags an Error Flag can widen and exceed the original number of six bits. For this reason, each CAN node receiver and transmitter tolerates up to 7 dominant bits after the transmission of an Error Active, Error Passive, or Overload Flag. After receiving these eight dominant bits, the error counters are incremented by eight. After every eight further dominant bits, the error counters are incremented again.

Rule 7
After each successful transmission of a message, the transmit error counter is decremented by one, as long as the counter is not already at zero.

Rule 8
After each successful reception, the receive error counter is decremented by one if the counter value is between 1 and 127. If the counter has a value greater than 127, it is set to a value between 127 and 119. If the counter is already at zero, no change is made.

2.5 Bittiming

The Bosch CAN specification divides a bit time unit into four different segments: the synchronization segment, the propagation segment, the phase buffer segment 1, and the phase buffer segment 2. Each segment consists of specifically programmable basic time units tq. The length of that is determined by the system clock. The synchronization segment Sync_Seg is part of the bit time in which falling or rising edges should be observed on the bus. The distance between edges that occurred outside the synchronization segment is called a phase error. The Propagation Segment is intended to compensate for transmission delays due to the finite transmission speed of the signal. Phase Buffer segments 1 and 2 surround the sample point time. The Synchronization Jump Width (SJW) defines how far the sample timing may be shifted within the Phase Buffer segments to compensate for phase errors. Though due to unavoidable transmission delays, any CAN unit that has synchronized to the bitstream of the current transmitter will in reality have a phase error to the transmitting unit.

However, it is necessary that also the receiving units can write synchronously dominant bits to the bus to ensure that an arbitration sequence is performed according to the rules or to enable confirmation of error-free reception by the acknowledge bit. For this very reason, transmission speed and bus length must be taken into account to determine an adequate value for the propagation segment. If an unsuitable value is selected for this segment, sporadic synchronization errors will occur, which will lead to transmission disturbances. Another feature of the CAN protocol besides the propagation segment is the ability to synchronize to edges in the data stream. This synchronization is performed on falling edges from recessive bits to dominant bits. The goal is to guarantee a certain time interval between the edge and the sampling time.

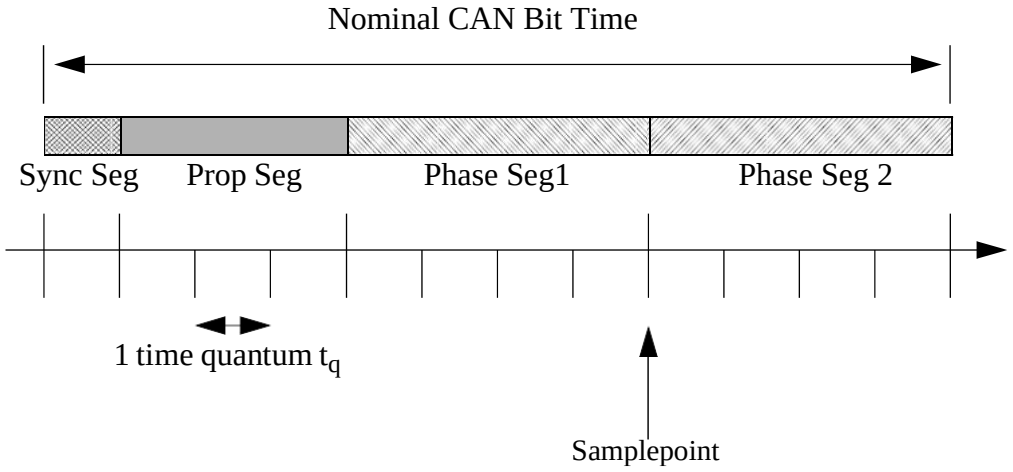


Figure 17: Structure of a basic time unit

An edge is determined by sampling the value of the bus at each basic time unit and comparing it with the previous bus value. A synchronization is executed if the previous bus level was recessive and the current bus level is dominant. An edge appears synchronous to the CAN unit if it occurs within a synchronization segment. Otherwise, the distance between the synchronization segment and edge is the phase error. If the edge occurred before the synchronization segment, the phase error is negative, otherwise positive.

There are two types of synchronization operations. Hard synchronization and resynchronization. Hard synchronization takes place only at the beginning of transmissions at the Start of Frame bit. Within a frame, only resynchronization is performed.

Hard Synchronization
If a hard synchronization is performed, the bit time after the falling edge starts with the end of the synchronization block regardless of the error. This means that hard synchronization "forces" the edge into the synchronization segment by making the bit time unit in which the falling edge occurred the synchronization segment and restarting the bit time unit count.

Bit Resynchronization
Resynchronization leads to an extension or shortening of the bit time so that the position of the sample time is moved with reference to the falling edge. If the phase error that triggers the resynchronization process is positive, Phase Buffer 1 segment is then extended. If the phase error is smaller than the maximum synchronization jump width SJW, the phase buffer 1 segment is extended according to the phase error, otherwise the phase buffer 1 segment is only extended by SJW. If the phase error that triggers the resynchronization process is negative, the Phase Buffer 2 segment is shortened. If the amount of the phase error is smaller than the maximum SJW, the phase buffer 2 segment is shortened according to the phase error, otherwise the phase buffer 2 segment is only shortened by SJW. In case the amount of phase error is less than or equal to the maximum synchronization jump width, hard synchronization and resynchronization have the same results. However, if the magnitude is larger than the maximum synchronization jump width, the synchronization process cannot compensate for the full phase error. A discrepancy (phase error - SJW) remains. Buffer segments are only extended or shortened temporarily. At the beginning of the next bit time, they take the preset values again. Most often, resynchronization processes are triggered during the arbitration phase. The bus nodes synchronize to a "leading" transmitter, which started to transmit first, but due to the transmission delay, they are not completely synchronized to each other. However, the leading transmitter does not necessarily have to win the arbitration. For this reason, the receivers have to synchronize to other CAN units that temporarily take the lead and are not equally synchronized due to the different geographical distance to the previous transmitter. The same happens with the acknowledge bit since the transmitter and some of the receivers have to synchronize to the leading CAN node. Resynchronizations that occur after the end of the arbitration sequence are due to inaccuracies of the clock oscillator. Over time, the differences in the period of the oscillators add up, so that a correction becomes necessary.

3. Documentation of the Canakari design

3.1 Structural organization of the top hierarchy level

The Canakari implementation of the CAN specification can be roughly divided into five main components that interact with each other to fulfill the required functionality. As shown in Figure 18, these components are named CPU Interface Logic, Logical Link Control, Medium Access Controller, Timing Logic, and Fault Confinement.
The CPU Interface Logic controls the communication between the CPU and the controller. It allows the CPU to load control and user data into the controller's registers or to read the information received from the controller.
The Logical Link Control evaluates control information stored in the registers of the Logical Link Control to initiate send operations, to check whether a received message should be stored or discarded, and to reset the CAN controller if this is desired by the CPU.
The Medium Access Controller (MAC) is the actual protocol unit. It encapsulates or decapsulates received data or data to be sent. If required, it takes part in the competition for the bus during the arbitration phase. It transmits and receives CAN frames. It detects errors and initiates error handling depending on the current error state.
The current error state is determined by the Fault Confinement instance. It counts the number of errors that have occurred. It takes into account the weighting of the errors generated by the MAC unit and informs the other units of the controller of the current error status.
The Timing Logic is responsible for managing the timing bit length. It reports to the MAC unit the times at which it can place a new bit on the bus or read a new bit from the bus. It is also responsible for hard synchronization or resynchronization to the bitstream.

Several signals are used for communication between the individual components. The CPU interface logic notifies the LLC unit via the signal **initreqr** that the bit has been set in the general register by the CPU, which was agreed for signaling the request for initialization of the controller. If the CPU has placed transmit data in the controller's register and wants to initiate the transmit process, the CPU interface logic sets the **traregbit**. If a CAN message has been successfully transmitted or received, the LLC unit sets the corresponding bits in the controller registers by using the signals **sucftrao**, **sucfreco**, **recindico** (successful transmission, successful reception, reception indication) and by activating the writing of the registers by the signals **actvtreg**, **actvrreg** and **actvgreg** (activate transmission, reception, general control register). The CPU interface logic component provides the LLC unit with the current value of the bits in the control registers through the signals **sucfrecvr**, **sucftranr,** and **recindicr**. With these values, the LLC unit can check whether the CPU has read and reset the control information from the bits.

If the LLC unit e.g. wants to inform the CPU about the successful reception of a CAN message by setting the corresponding bit in the control register using the signal **sucfreco**, it can first detect by the **sucrecvr** signal whether the CPU has noticed the previous signaling of successful reception and set the **overlowo** bit if necessary, so that the CPU recognizes that a new CAN message has overwritten the data from a previous message in the data registers before they were read by the CPU. If the CPU notifies the controller that it wishes to send a CAN message, the **trareg** signal is activated by writing the corresponding bit in the transmission register.

Subsequently, the LLC unit resets the transmit part of the MAC by the signal **resettra** (reset transmission) and instructs the MAC unit one clock cycle later by the signal **actvtcap** (activate capsulation) to encapsulate the transmit data from the transmit registers present at the signals **data1r-data8r**, **identifierr**, **datalenr**, **remoter** and **extendedr** into the CAN frame format.

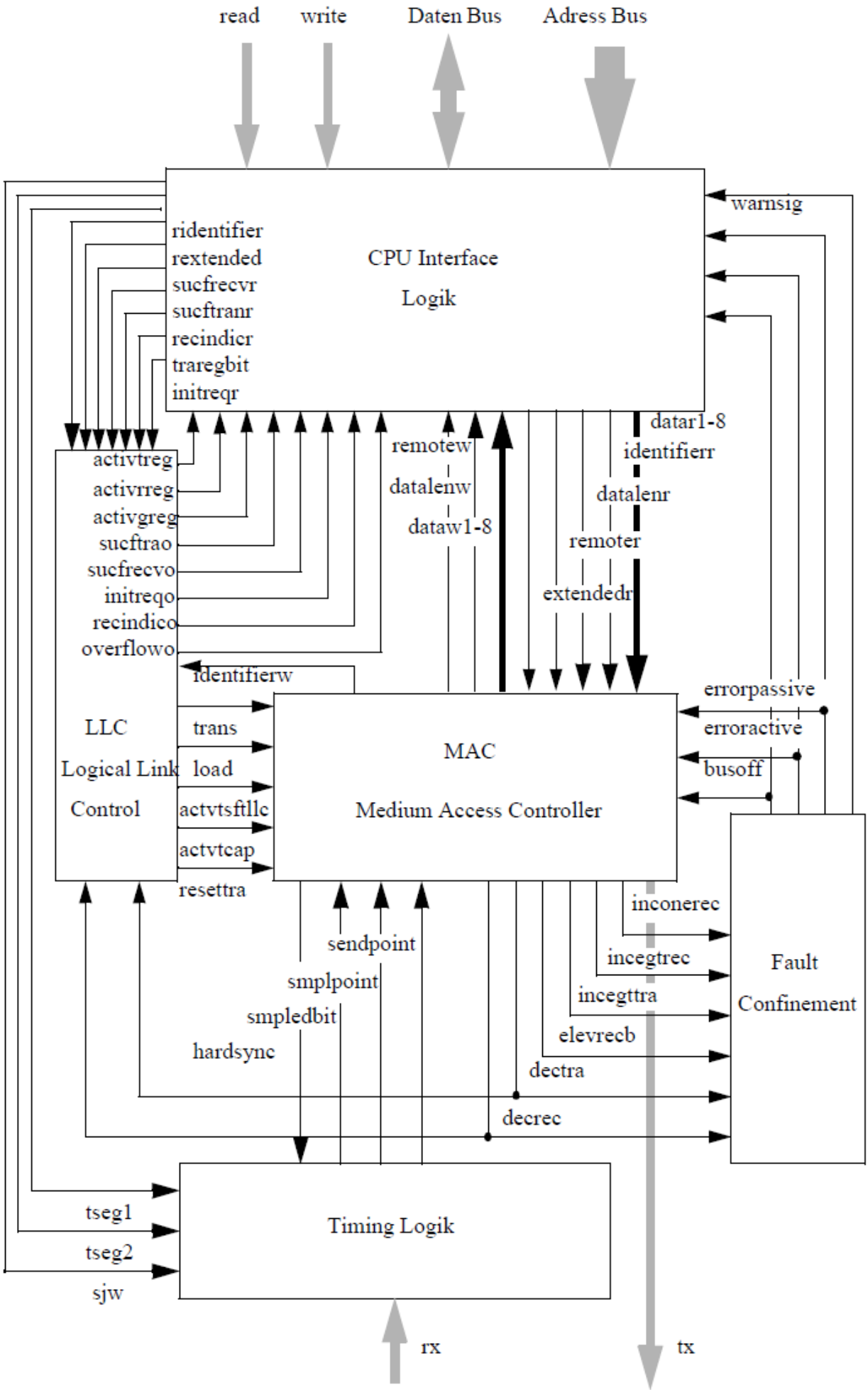


Figure 18: Structure of the top hierachy level of the Canakari Controller

identifierr contains the identifier of the CAN message to be sent. **datalenr** specifies the number of user data bytes. **remoter** and **extendedr** provide information about whether the frame is an extended or a standard frame and whether a data or a remote frame is to be sent. One clock cycle later, the LLC unit loads the encapsulated data from the encapsulation unit into the transmit shift register of the MAC instance by setting the signals **load** and **actvtsftllc** (activate shift register). Finally, the LLC unit sets the signal **trans** (transmission). This informs the MAC unit that a transmission request has been received and that the data is ready in the transmission shift register. If a message has been successfully transmitted or received by the medium access controller, it informs the LLC unit and the fault confinement component by the signals **dectra** and **decrec** (decrement transmission/reception error counter). In addition, the user and control data decapsulated from the received CAN frames are provided at the signal outputs **data1w**-**data8w**, **identifierw**, **datalenw**, **remotew** and **extendedw**. The LLC unit now compares the identifier **indentifierw** and the extended bit of the CAN frame received from the MAC instance with the identifier and the extended bit written by the CPU into the receive register of the CPU interface logic device and passed on to the LLC unit by the signals **ridentifier** and **rextended**. If the masked identifiers match, the received data is backed up by the LLC unit activating the receive registers by the **actvrreg** signal. The medium access controller informs the fault confinement unit by the signals **incegttra**, **incegtrec** and **inconerec** (increment one/eight receive/transmit error counter) about the detection of an error during the reception or transmission of data and the number by which the corresponding error counter is incremented.

After each successful transmission or reception of a CAN message, the MAC instance instructs the Fault Confinement unit to decrement the receive error counter (**decrec**) or the transmit error counter (**dectra**) by one. The fault confinement unit monitors the error counters and sets the signals **erroractive** , **errorpassive** and **busoff** to provide information about the current error status. When the error counters reach such a high level that the controller must enter the **busoff** state, the MAC unit scans the bus for eleven consecutive recessive bits. Each time this occurs, the MAC unit transmits this event to the fault confinement unit by the **elevrecb** signal (eleven recessive bits). When one of the two fault counters reaches 96, the FC unit sets the warning signal, which the CPU interface logic feeds into the general register to inform the CPU that faults are accumulating on the bus. The timing logic evaluates the timing of the signals **tseg1** (propagation segment + phase segment 2), **tseg2** (phase segment 2) and **sjw** (synchronization jump width) previously set by the CPU in the general register of the CPU Interface Logic Unit and uses the signals **sendpoint** and **smplpoint** (sample point) to provide the Medium Access Controller with timing points for sending and receiving a new bit. The medium access unit informs the timing logic by the signal **hardsync** (hard synchronization) whether it should hard synchronize on falling edges or just resynchronize.

3.2 Logical Link Control

The Logical Link Control unit links the actual protocol unit, the Medium Access Controller with the CPU. As shown in Figure 19, the LLC unit is in the idle state **waitoact** (wait on action) as long as no actions are triggered by the CPU or the protocol unit. For example, if the CPU has set the **initqr** signal by writing the corresponding bit in the general register, the LLC unit enters the reset state in which the complete CAN controller is reset by the **resetall** signal.

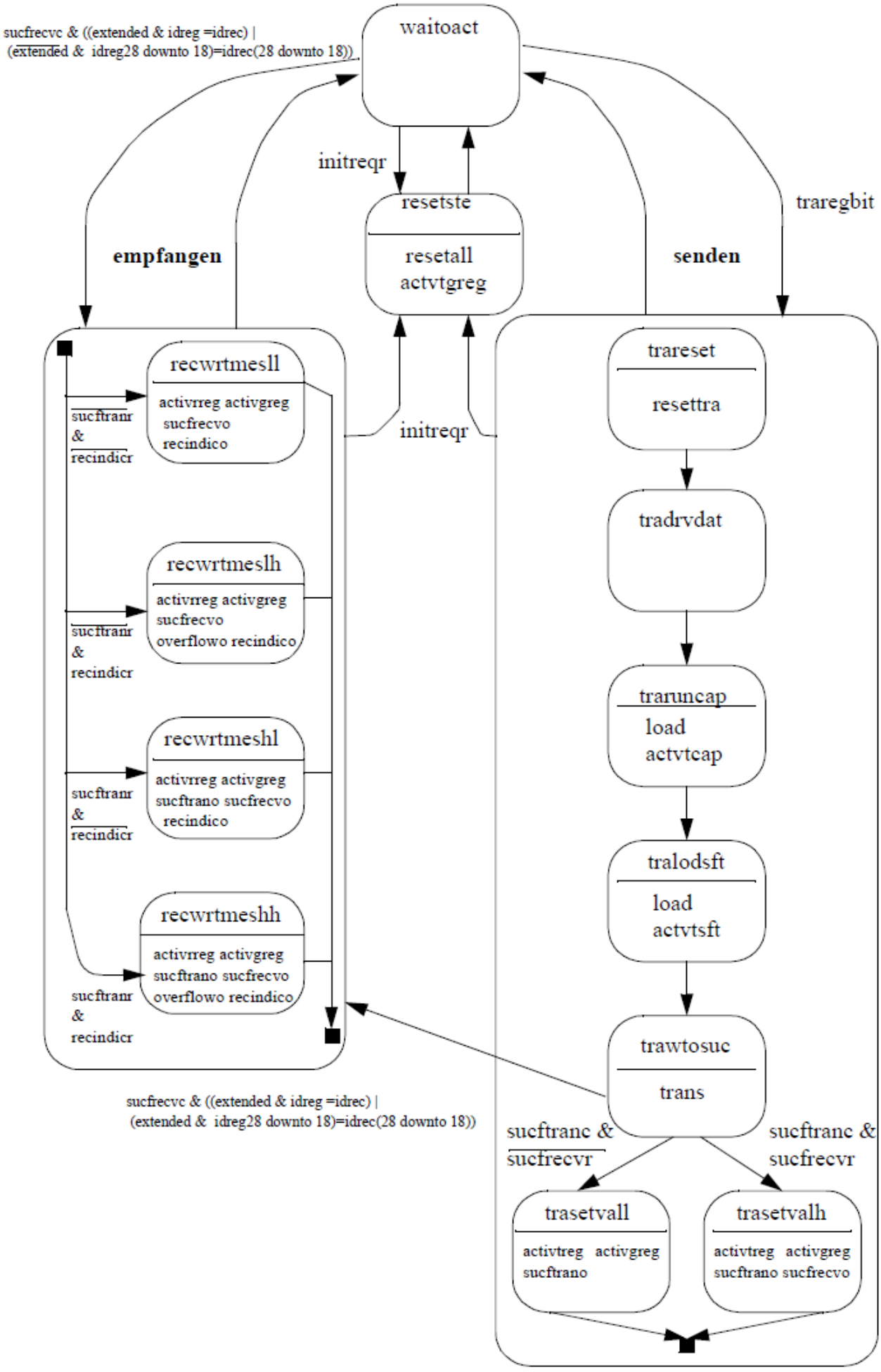


Figure 19: Logical Link Control state transition diagram

The transition to the **resetste** state is always performed when the **initqr** signal appears, regardless of the state in which the LLC is currently located since the initreqr signal (initialization request register bit) is checked in every state. When the protocol unit receives a new CAN frame, it is up to the LLC to check whether this message is stored in the receive registers or gets discarded. The MAC signals the reception of a new message to the LLC unit by the signal **sucfrecvc** (successful received can bit). However, the transition to receiving is only completed if the identifier **idreg** (identifier register) stored by the CPU into the receive control registers matches the received identifier **idrec** (identifier received), where the signal extended is used by the CPU to determine whether the received frame may be extended or not. If the addresses match, the LLC address activates the receive registers by the signal **activrreg** (activate receive register) and sets bits in the receive control register and in the general register, which inform the CPU that a new CAN frame has been successfully received and stored. The LLC unit has to consider the previous contents in the registers. It must know whether the last stored message has already been read by the CPU which is signaled by the CPU by resetting the **recindicr** bit. If the **recindicr** is not reset, it means that the old CAN message has been overwritten by the new one. The LLC unit clarifies this to the CPU by setting the **overflowo** bit. Furthermore, the LLC unit takes care of the current value of the **sucftranr** (successful transmission bit) of the general register, in order not to falsify the value of the **sucfrectranr** bit when writing the **sucfrecvo** bit (successful received bit) into the general register. This is necessary because only the complete register can be accessed and individual bits cannot be manipulated without affecting the other register contents. So all combinations of **sucftranr** and **sucfrecvr** bits must be considered, which gives a state number of $2^2 = 4$.
For this reason, four states are starting with the name **recwrtmes** (write received message). The last two letters ll, lh, hl and hh indicate the values of the **sucftranr** and **sucfrecvr** bits, where l stands for low and h for high. If the CPU requests to send a CAN frame by setting the traregbit (transmit register bit) signal, the LLC first resets the transmission report of the MAC by the use of the **resettra** signal in the **trareset** (transmission reset) state. Then the LLC unit waits for one clock cycle in the **tradrvdat** state to ensure the readiness of the transmit unit of the MAC after the reset. In the **traruncap** state (transmission run capsulation), the **actvtcap** signal activates the MAC's encapsulation unit. In addition, the signal load is set to inform the logic of the register that a load operation and not a shift operation has to be conducted when the transmission shift register is activated in the next state **tralodsft** (transmission load shift register).
The activation of the register is done by the signal **actvtsft** (activate shift registers). When all preliminary actions are completed, the LLC unit activates the **trans** signal in the **trawtosuc** (transmission wait on success) command. The MAC protocol unit takes the signal into account when it reaches an intermission field and participates in the arbitration phase in the competition for the bus. The LLC unit remains in the **trawtosuc** state until the successful transmission of the frame is reported by **sucftranc** (successful transmission controller bit). In case errors occur during transmission and the protocol unit had to start a new transmission attempt, the transmission is delayed. However, the LLC instance remains in the **trawtosuc** state until said signal arrives. If the controller loses arbitration and instead successfully receives a message carrying the identifier requested by the CPU, the LLC exits the **trawtosuc** state to initiate data storage in the receive registers. However, if the controller wins the arbitration and the CAN frame is sent successfully, the LLC sets the successful send bit in the general and transmission registers in the state **trasetval** (transmission set value), whereas on the receive side, the old values of the general register must be respected. I.e. if the successful received bit in the general register is high, the LLC branches to the state **trasetvalh** to set the successful received bit as well as the successful send bit. If it has the value low, only the successful send bit is set in the state trasetvall.

4. Medium Access Controller

The Medium Access Controller forms the actual protocol unit. It receives and transmits CAN frames, checks the bitstream for errors and reacts according to the current error state. The heart of the Medium Access Controller is the MAC FSM. This state machine controls the remaining components in a way that the CAN 2.0B specification is met. Essentially, the components can be assigned to the receive or transmit side.
On the transmit side, whose components are located in the upper half of the page in Figure 20, the transmit data is fed from the transmit registers to the encapsulation unit. When a transmit request is present, the LLC activates the encapsulation unit as described above and fills the encapsulated data into the transmit shift register. If the state machine detects that the end of the intermission field has been reached and if the **trans** signal indicates to the LLC unit that there is a send request, the MAC FSM starts sending the frame by letting the stuffing unit write the SOF bit to the bus by **actvtstf**. The stuffing unit obtains the information (**bittosend**) from the transmit shift register which bit is to be transmitted next. It checks whether a bit of the same value is to be sent for the sixth time in succession, which would violate the stuffing rule and would be interpreted as an error by the other bus nodes. If a sith bit with the identical value gets detected, it sets the **tstuff** signal and places a bit with the opposite value to the previous bits on the bus.
The MAC FSM recognizes the stuffing case by **tstuff** signal and therefore does not shift the current bit from the shift register, instead, the same bit gets sends again at the next starting bit time. If there is no stuffing case, the MAC FSM causes the shift register to shift the next bit into the transmit position by setting the signal **tshift** and activating the shift register with **actvtsft**. Whenever there is no stuffing case, the CRC register is also activated by **actvtcrc**, which is preloaded with the 15 least significant bits of the transmit shift register, thus containing the valid Crcsum after the last data bit has been sent, which is loaded into the transmit shift register by the signal **tcrc** when it is activated the next time.
However, it is often necessary to bypass the functionality of the stuffing unit to send fields that intentionally violate the stuffing rule. For example, if the end of **frame** of a CAN frame or an error flag is sent, the MAC FSM can instruct the stuffing unit to send either only recessive or only dominant bits by setting the **setbdom** (set bit dominant) or **setbrec** (set bit recessive) signals. It may however also be instructed via the **actvtdct** signal (activate direct) to send the shift register contents without applying the stuffing rule. The MAC FSM is notified by the timing logic of the send and receive time by the signals **sendpoint** and **smplpoint**. The signal **sendpoint** triggers the activation of the stuffing unit in the transmit state. When the signal **smplpoint** is set, the shift register is activated and the polarity of the transmitted bit is compared with the polarity of the received bit by the bit error detection unit to detect a bit error if necessary. Bit error detection is activated by the signal **actvtbed** (activate bit error detection) and sets the signal **biterror** in case of an error. The MAC FSM uses the counter component to determine the number of bits sent or received. These numbers are necessary to ensure protocol compliance. For example, it is necessary to determine when the three bits of the intermission field have been received to start the arbitration phase at the correct time. The counter and the **extendedr** signal enable the FSM to detect when the arbitration phase has ended and when the user data is sent. Since the FSM has been informed of the number of user data bits by the **tmlen** signal, it can use the counter to determine at what time the last data bit was sent and when the CRC sum can be started. The counter is incremented by the signal **inccount** (increment counter) or reset by the signal **rescount** (reset counter).

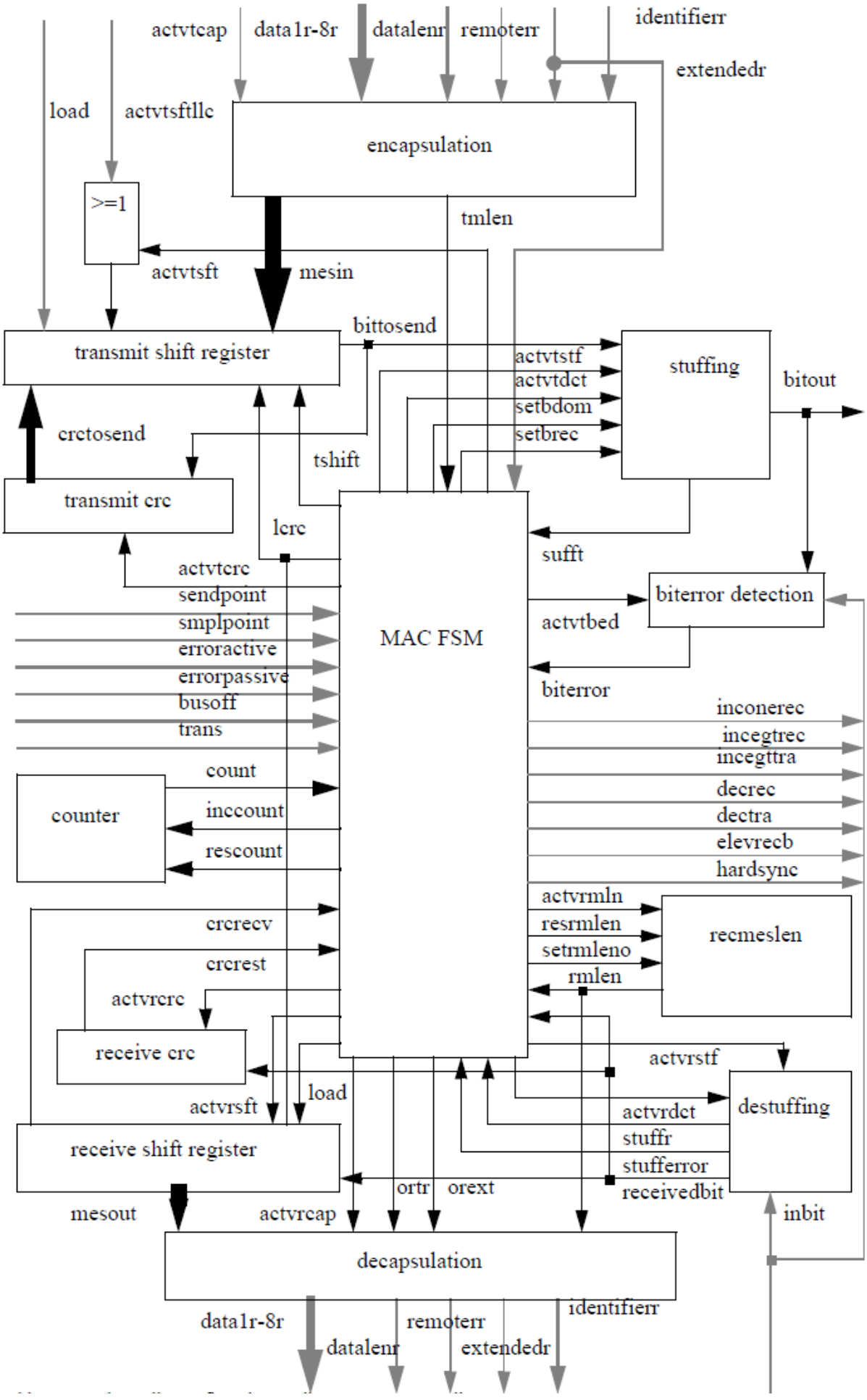


Figure 20: Architectural design of the Medium Access Controller

The receive side is identical to the transmit side. A stuffing unit, a shift register and a crcre register can be found there as well. If the controller has lost arbitration or if the controller had no data to transmit, the controller becomes the recipient of a message sent by another CAN node. Each time the **smplpoint** signal is set by the time logic, the MAC FSM activates the destuffing module by the signal **actvrstf**. Similar to the transmit side, the unit checks whether six bits of the same polarity have been transmitted on the bus. If this is the case, it is a violation of the stuffing rule and the MAC FSM is informed about this error by the signal **stufferror**. If a bit of opposite value follows after five bits of the same value, the destuffing unit detects compliance with the stuffing rule and removes the stuffing bit from the bitstream. The MAC FSM is informed by the signal **stuffr** that the current bit is a stuffing bit that must not be taken into account in the calculation of the receive crcsum and should not be included in the receive shift register. Therefore the MAC FSM does not set the signals **actvrcrc** and actvrsft in the stuff case. The MAC FSM must extract control data already during reception of the bitstream to be able to recognize the length and the type of the CAN frame. For example, it must check whether the IDE (Identifier Extended) bit has been set to determine whether a standard or an extended frame is being received. To be able to consider whether and how many user data bytes the current CAN message will contain, the MAC FSM must extract the RTR (Remote Request) bit and the DLC (Data Length Code) from the bitstream. The recmeslen unit (receive message length) calculates the number of user data bits from the DLC bits. The MAC FSM passes the weight of the current DLC bit by the signal **setrmleno** and activates the recmeslen module by the signal **actvrmeln** (activate receive message length). When all user data bits have been received, the MAC FSM sets the signal **lcrc** to trigger the receive shift register to store the incoming bits in a separate area intended for the CRC sum transmitted by the transmitting CAN unit. If the CRC sum has been received, it is compared by the MAC FSM with the CRC sum calculated during reception. The received and self-calculated CRC sum is fed to the MAC FSM via the signals **crcrecv** (crc received) and **crcrest**. When the validity of the CRC sum has been confirmed, the data of the receive shift register is decapsulated by the decapsulation unit. The flags that have been filtered out of the bitstream during the reception of the CAN frame are helpful here. They spare the decapsulation unit additional logic to determine the type and length of the received frame. The flags are fed to the decapsulation module by the signals **ortr** and **oext**. The number of payload bytes is passed by the signal **rmlen**. After the decapsulation unit has been activated by the MAC FSM through **actvrcaps**, the received data is available at the outputs for storage in the receive registers.

4.1 Overview of the MAC State Machine

The implementation of the MAC state machine is an incredibly complex task since all details of the CAN protocol specification have to be considered. The state machine of the Canakari design consists of 126 states. The design does not claim to be optimally developed, but due to the complexity of the required functionality, it will be very difficult to reduce the number of necessary states significantly. As shown in Figure 20, the states of the MAC FSM can be roughly divided into seven groups. After the controller has been switched on, the MAC FSM enters the synchronization state. There it waits for the appearance of eight successive recessive bits on the bus before it takes part in the further communication action. This may be an End of Frame or an Error Delimiter. It may also be that the bus nodes are in the bus idle state since none of them has messages to send.

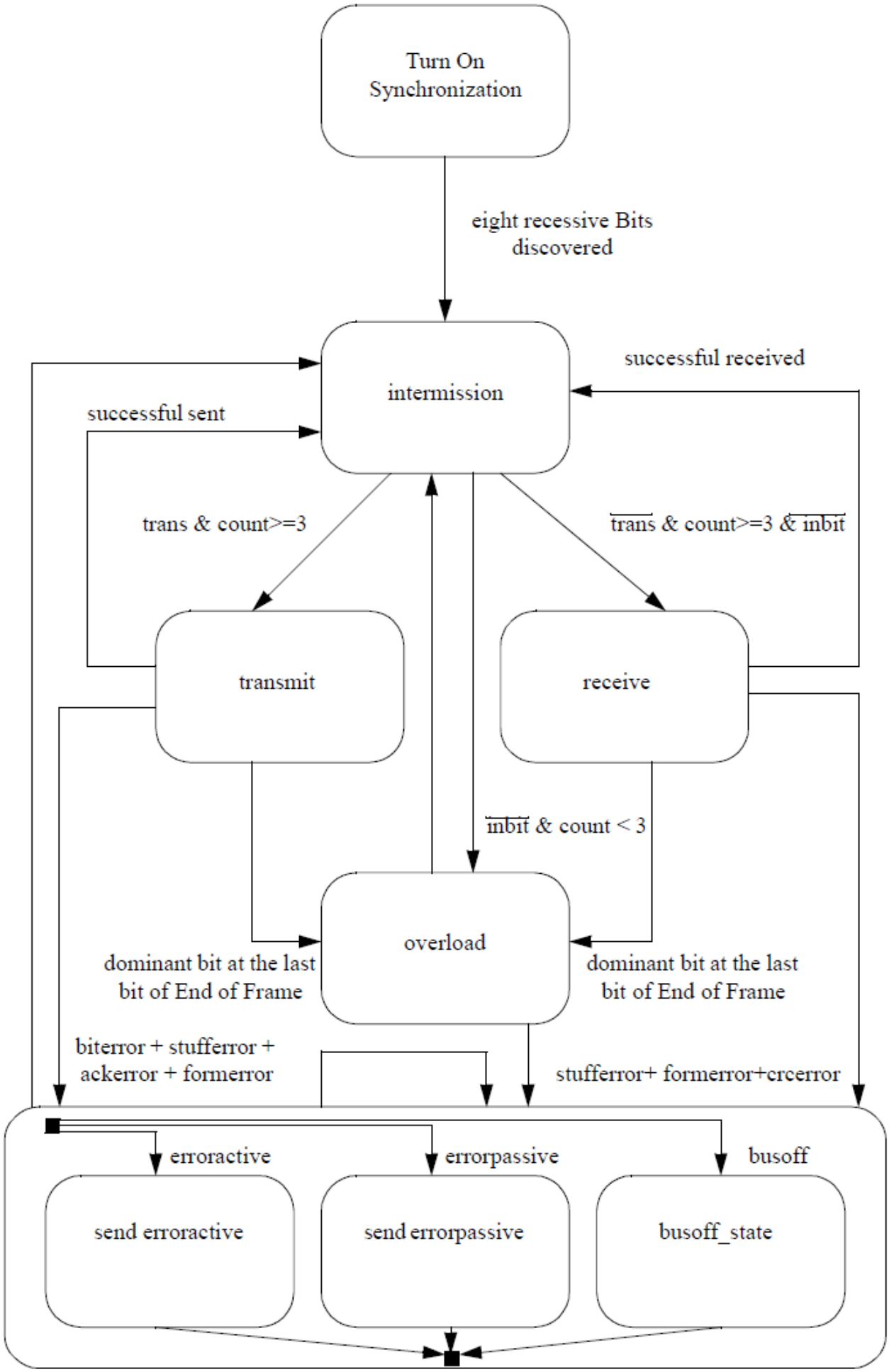


Figure 21: Rough outline of the Medium Access Controller State Machine

This synchronization process reduces, although does not fully eliminate, the likelihood that the new CAN controller entering the communication flow will interfere with any ongoing exchange of information. The controller then enters the intermission state. The intermission field consists of three recessive bits. Only after these three bits, the controller can enter the transmit or receive state. However, the transition conditions are much more complicated than shown in figure 20. For now, however, it is sufficient to know that the controller enters the overload state when any dominant bit is detected on the bus during the intermission period. If a dominant bit is received, an overload frame must be generated according to the CAN specification. In the regular case, a controller wishing to transmit, indicated by the trans signal from the LLC, may enter the transmit mode after the third bit of the intermission field, which initiates an arbitration phase. If the controller has no message to transmit, it waits in the intermission state until such a message is received or another CAN instance writes a SOF bit to the bus. If a message was successfully sent or received, the CAN controller returns to the intermission state. If a dominant bit is detected during reception or transmission of the last end of frame bit, an overload frame is transmitted. If an error occurs during reception or transmission of the frame, the controller starts immediate handling of the error case depending on the current operating state of the controller. After transmitting errors or overload frames, the MAC FSM returns to the intermission state. If errors occur during the transmission of an overload or error frame, the transmission of a new error frame is started.

4.1.1 Synchronize State

In the synchronization state, the current bus state is read at each sample time by activating the destuffing unit via the signal actvrstf. Since the stuffing rule should not be observed, the signal actvrdct is set. Furthermore, the signal hardsync is set, because the controller synchronizes hard to falling edges during the synchronization state. If a recessive bit is detected on the bus, the state sync_sum is set and the counter is incremented by one. If a dominant bit is read from the bus, the counter is reset in the sync_start state. If the eighth recessive bit has been read in sequence, the MAC FSM jumps to the end of the synchronize state. The counter is reset again to make it ready for use in the intermission state.

4.1.2 Intermission State

The intermission state is rendered much more complicated than it would otherwise be by the requirement to interpret a dominant bit in the third bit of the intermission field as SOF. The intermission procedure starts in the inter_sample state. If the first or second bit of the intermission field is overwritten by a dominant bit, an overload flag is sent. If the CAN controller wants to send a CAN message, it can do so after the third bit of the intermission field, if the controller is in the Error Active state. If the controller is in Error Passive state, it must wait for the other eight bits of the Suspend field. Subsequently, in the inter_goregtran state (go to regular transmission), it sends the SOF bit, activates the CRC register and resets the counter. Before finally jumping to the transmission state, the counter is incremented in the inter_regtrancnt state so that the transmission of the SOF bit is taken into account. If a transmitting unit anticipates the request of the controller to transmit by overwriting the third bit of the intermission field with a dominant bit, the controller interprets this bit as SOF and participates in the arbitration if it is not in Error Passive state. At the same bit time, the controller switches to the inter_react state to catch up with the SOF bit and the initial activation of the CRC register. Before the controller can participate regularly in the arbitration process starting from the second bit, it must first shift the first bit out of the transmit register in the inter_transhift state. If the controller has no message to transmit, the controller waits after the third bit until a dominant SOF bit has beend sampled. If this happens, the counter is reset in the inter_preprec (prepare for reception) state and the reception process is then triggered

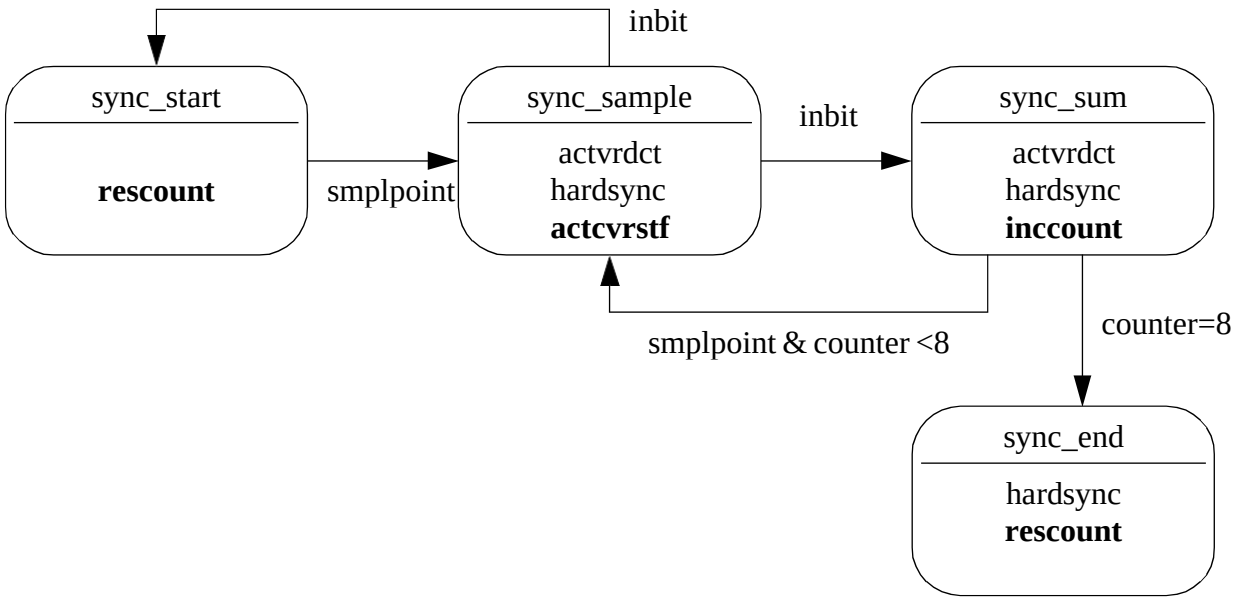


Figure 22: State transition diagram of the synchronization state

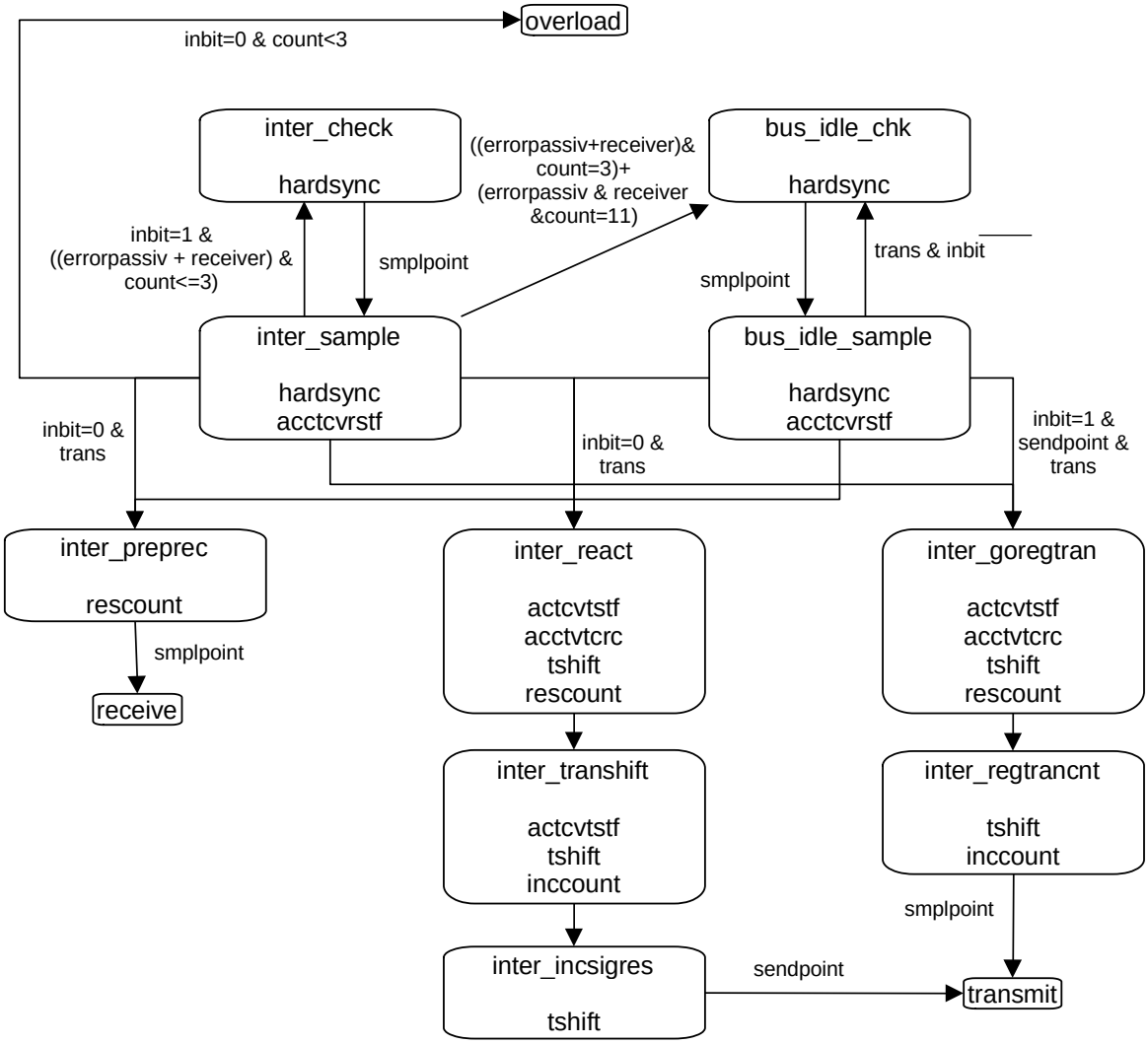


Figure 23: State transition diagram of the intermission state

4.2 Overview of the send state

The transmission process starts with the arbitration sequence. If a bit error is detected during arbitration, it is highly probable that this error has been caused by the overwriting of a recessive bit by a dominant bit of a CAN controller of a higher priority. In this case, the CAN controller withdraws from the bus and becomes the receiver of the message. The arbitration ends with the reception of the RTR bit, which is located in the 13th bit for standard frames and the 33rd bit for extended frames (see Figure 9). If the controller won the arbitration, it sends the 6 bits of the control field and the user data, if any, in the Send Data state (tmlen!=0). If the last user data bit has been sent or the last bit of the control field if no user data is present, the 15 bit wide CRC sum calculated during transmission is transmitted. After the CRC delimiter has been sent in the CRC state, the controller switches to the ACK state. A recessive bit is sent by the MAC FSM while the other communication participants are expected to verify that the message has been received without errors by overwriting the recessive bit with a dominant bit. If this does not happen, an acknowledge error has occurred. The controller sends the acknowledge delimiter and enters the error state for error handling. If a communication partner sets the acknowledge bit, the controller enters the end of frame state in which the acknowledge delimiter and the EOF field and therefore eight recessive bits, are sent. After the error-free transmission of the CAN message, the MAC FSM returns to the intermission state.

4.2.1 Arbitration phase

During arbitration, data is not only sent but also received, i.e. the destuffing unit (**actvrstf**), the CRC receive register (**actvrcrc**) and the receive shift register (**actvtrsft**) are activated in addition to the components of the transmission side. Furthermore, the information about the frame type is extracted from the data stream. This is done by setting the **rrtr** (received rtr bit) and the **rext** (received extended id bit) as required at the given time. Thus, in case of bus loss, the reception oft he frame can be performed according tot he rules. The arbitration state is entered either in the **tra_arbit_tsftrsmpl** state (transmit side shifting, receive side sampling) when the controller starts transmitting during or at the beginning of the bus idle state, or in the **tra_arbit_tactrsftn** state (transmit side CRC activation, receive side shifting, normal type) when the third bit of the intermission field has been overwritten. There are two states in which the destuffing unit is activated and thus the bus value is sampled. In the **tra_arbit_tsftrsmpl** state, the receive register is also shifted so that the next bit to be transmitted is available at the time of transmission. The **tra_arbit_tnsftrsmpl** state (transmit side no shift) is only entered by the MAC FSM if the stuffing unit has previously signaled the application of the stuffing rule by the tstuff signal when sending the last bit. In the stuffing case, the current bit is ignored and a bit with opposite polarity to the previous bits is inserted into the bitstream. For this reason, a new transmission attempt must be started. Therefore in the stuffing case in the transmit register is not shifted and only the destuffing unit is activated in the **tra_arbit_tnsftrsmpl** state. The destuffing unit also recognizes the stuffing case and sets the signal rstuff.

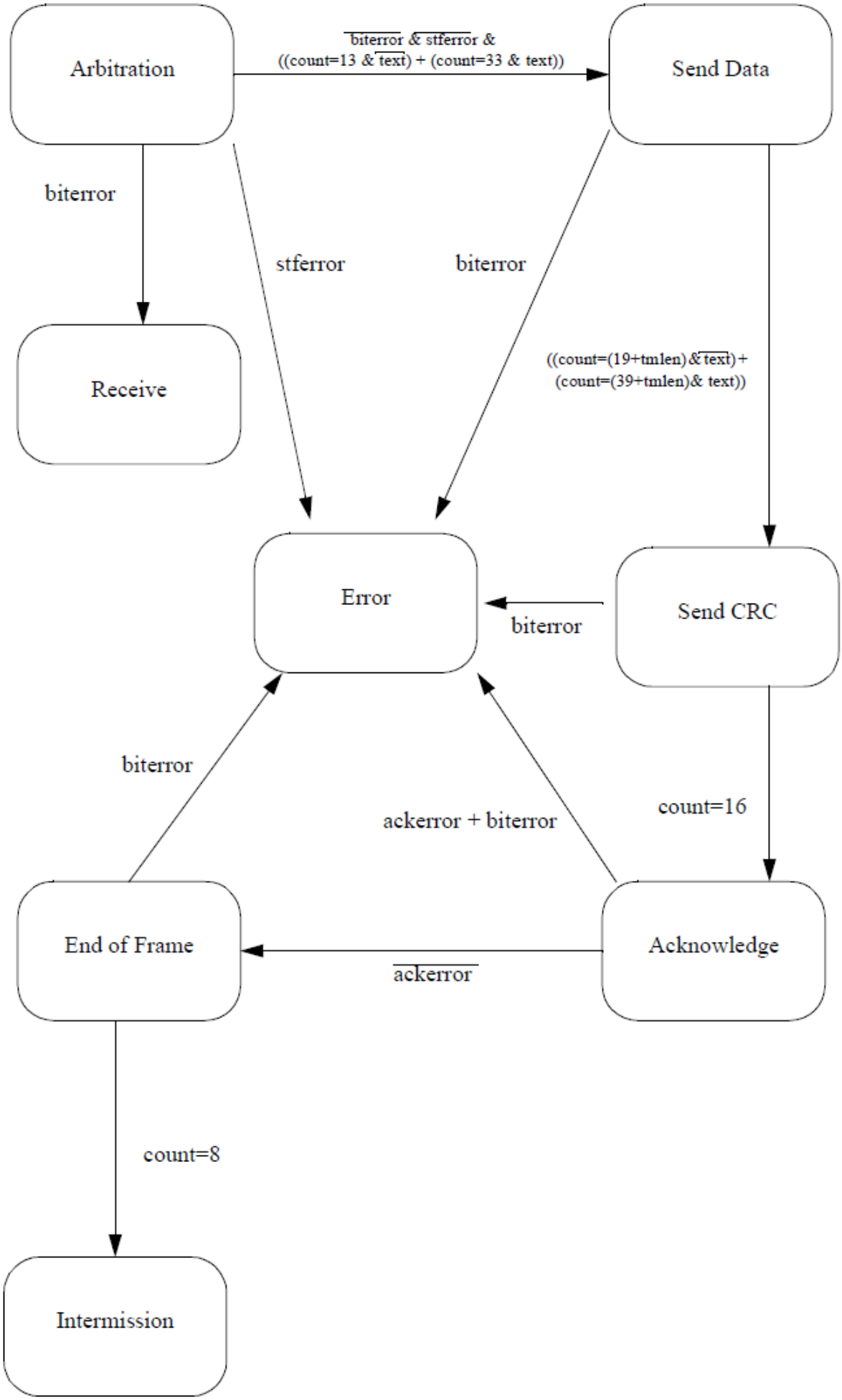


Figure 24: State transition digram of the transmitting state

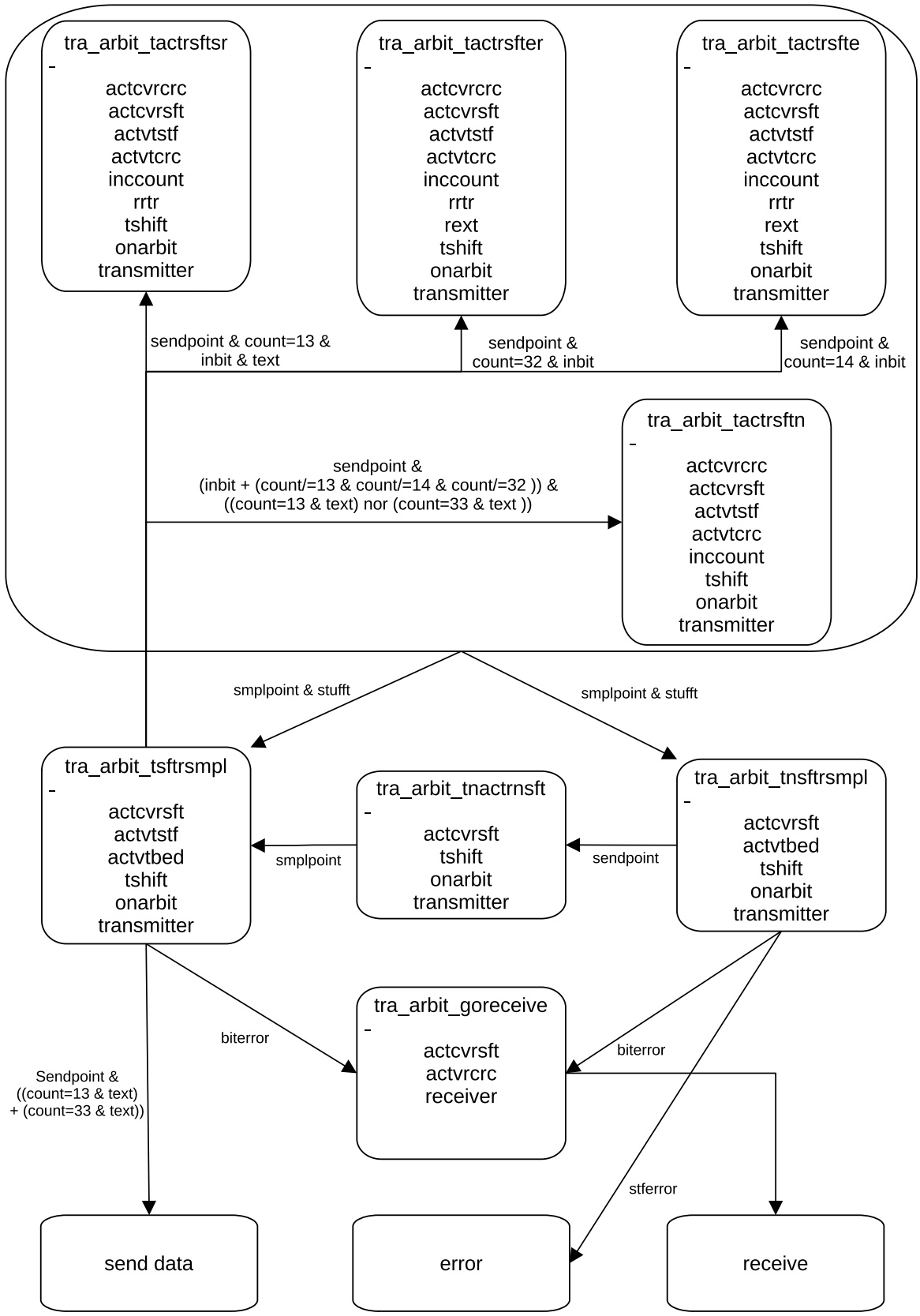


Figure 25: Arbitration state transition diagram

However, since the MAC FSM has already been informed of the stuffing case by the stuffing unit when the bit was sent, the stuffr signal does not need to be taken into account here. The MAC FSM then enters the **tra_arbit_tnactrnsft** state to start the retransmission attempt. During this transmission attempt, it is not necessary to activate the transmit CRC register, since this has already been done during the first transmission attempt. In addition, the bit that the destuffing unit has read from the bus must not be fed to the receive shift register and the receive CRC register, since it is only a stuffing bit. For the regular, non-stuffing case, the sampled bit is stored with actvrcrc and actvrsft, as can be seen in all four states (**tra_arbit_tactrsftsr**, **tra_arbit_tactrsfter**, **tra_arbit_tactrsfte**, **tra_arbit_tactrsftn**) in the upper rectangle of Figure 25. In addition, the bit present at the transmit position of the transmit shift register is processed in the stuffing and transmit crce unit (actvtcrc, actvtstf) and the counter is incremented (**inccount**). With the help of the counter, the IDE and RTR bit can be extracted from the received bitstream and provided to the receiving unit. For example, if the 13th bit received by the controller is recessive, the rrtr (received rtr bit) is set because it may be a remote frame. After all, standard frames have their RTR bit at the 13th-bit position. However, if the 14th bit is also recessive (count=14 & inbit), it is an extended frame, which is marked by setting the rext signal. For extended frames, the 13th position is not the RTR bit but the SRR bit (Substitute Remote Request) therefore the rrtr signal is reset and only reactivated if a recessive bit is received at the bit position of the RTR flag in the extended frame (count=32). For all other cases the MAC FSM branches to the tra_arbit_tacrsftn state and leaves the signals rrtr and rext unchanged. The endings of the names of the four states mentioned above indicate which case is being processed. (tactsft**sr** => standard remote, tactsft**e** => extended, tactsft**er**=>extended remote, tactsft**n** => normal). The arbitration phase of a standard frame ends after the 13 bit. The arbitration phase of an extended frame ends after 32 bits. If a bit error is detected during transmission, the controller has lost arbitration and switches to the receive state. The signal **tshift** informs the transmit shift register that when ist is activated it should only shift its register contents and not new register contents. The signal transmitter is necessary during error handling to increment the correct error counter. The onarbit signal is set in the error state, to recognize in case of an error, that a fault has occurred during arbitration and therefore the second exception of the third error rule has to be considered.

4.2.2 User data transmission state

The transmission of the control field and the transmission of user data starts in the tra_data_activtecrc state. In this state the data bits are put on the bus by the stuffing unit (actvtstf), the crcre register is activated (actvtcrc) and the counter is incremented (inccount). If neither a stuffing case is present nor the last data bit has been sent, the contents of the send shift register (actvtsft) are shifted in the tra_data_shifting state and then the next bit is sent in the tra_data_activtecrc state. If a violation of the stuffing rule occurs, the stuffing unit sets an opposite polarity bit on the bus line as well as the signal stufft. The MAC FSM enters the tra_data_noshift state in which no shifting of the register contents takes place since the last bit must be sent again. Subsequently, in the tra_data_activatncrc state, the last bit is passed into the stuffing unit again. However, the CRC register is not activated, since this bit has already been taken into account. If the last data bit has been sent, the lcrc signal is set in the tra_data_lastshift state so that the CRC sum from the CRC register is placed in the send shift register the next time the shift register is activated in the tra_data_loadcrc state. The MAC FSM then switches to the Crcsend state. If the Bit Error Detection Unit detects a bit error, the MAC FSM switches to error handling.

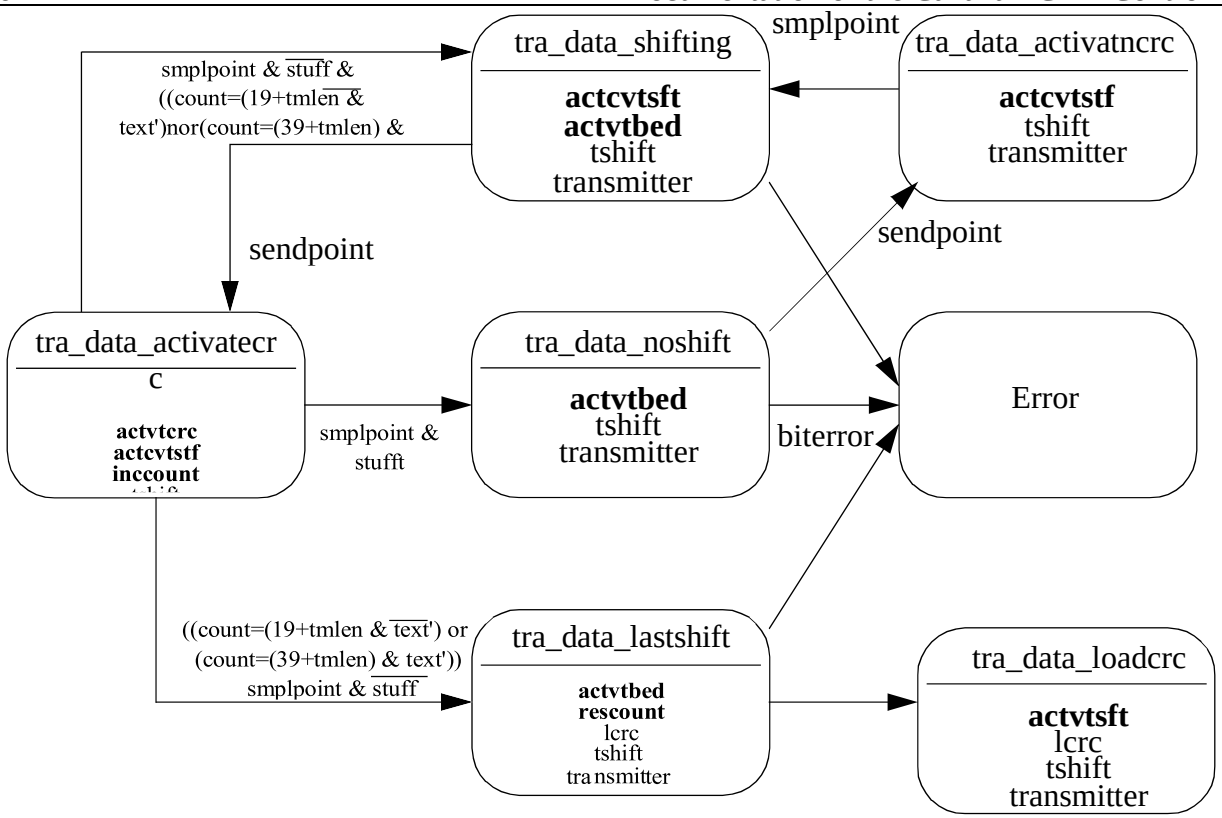


Figure 26: State transition diagram of user data transmission

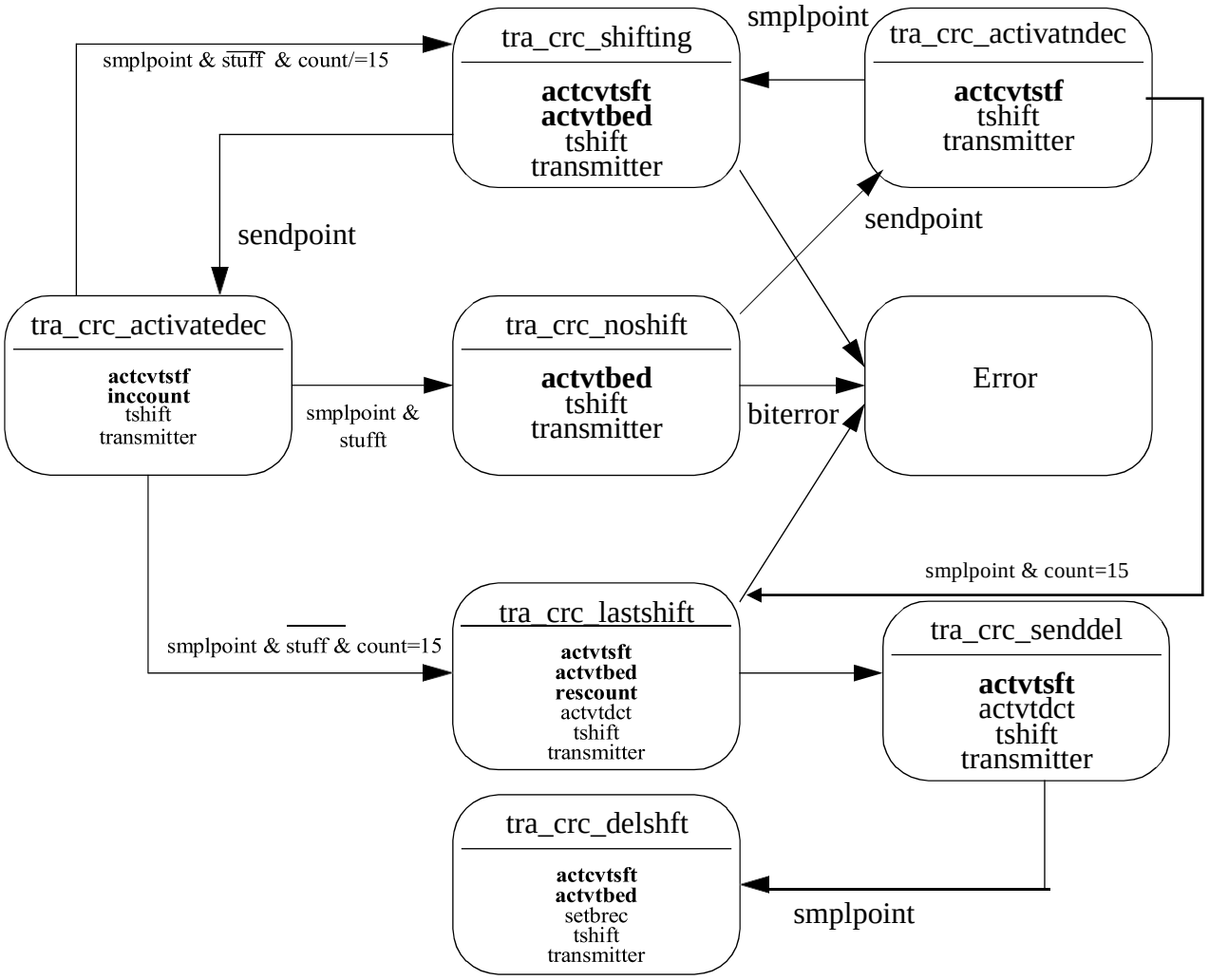


Figure 27: State transition diagram of the CRC sum transmission

4.2.3 CRC-Sum transmission

In principle, the CRC sum is sent in the same way as the user data. The only difference is that the CRC register does not need to be activated again because the CRC sum has already been calculated. The CRC data is also sent in compliance with the stuffing rule. If a stuffing case is detected, neither the next bit is moved to the send position of the shift register nor the counter is incremented. If all 15 CRC bits have been sent, the CRC delimiter is sent afterward in the tra_crc_senddel state. Since the CRC delimeter is not coded according to the stuffing rule, the stuffing unit is informed by the signal actvtdct that the rule do not need to be respected. In the last state of the CRC transmission, the setbrec (set bit recessive) signal is already set in preparation for the next state, which informs the stuffing unit that it may only place recessive bits on the bus when activated.

4.2.4 Acknowledge reception

In the tra_ack_sendack state, the MAC FSM places a recessive bit on the bus and checks in the tra_ack_shifting state whether the bit has been overwritten by another CAN unit, thus signaling that it has received the previously transmitted data without error. If the MAC FSM registers the overwrite of the Acknowledge Bit, it jumps to the End of Frame state to transmit the Acknowledge Bit and the End of Frame field, so a total of eight recessive bits. If the recessive bit in the acknowledge slot is not overwritten, the CAN controller transmits the acknowledge delimiter and sets the ackerror signal in the tra_ack_stopack state, which is used during error handling.

4.2.5 End of Frame transmission

The acknowledge delimiter and the end of frame field are sent. If the recessive bits are overwritten by dominant bits, the MAC FSM enters the error state. After the 7th bit of the End of Frame field is sent, the MAC FSM sets the signal **dectra**(decrement transmission error counter) in the **tra_edof_lastshift** state to notify the Fault Confinement Entity and the LLC that a message has been successfully transmitted. If the last bit of the End of Frame field is overwritten by a dominant bit, the MAC FSM enters the Overload state.

4.3 Overview of the reception state

The reception of a message is processed at the beginning in the flag and length state (Figure 30). The controller receives the identifier and the control field. The frame type and the number of user data bytes, in particular, are determined by checking the corresponding bits. This information is used later to detect the time when the reception of the CRC field starts and when the Acknowledge bit must be set. If the controller detects an RTR bit, it switches to the receive CRC state after receiving the control field, since remote frames do not contain user data. For standard frames, the number of data bits specified in the control field is received in the receive data state. When the last user data bit has been received, the controller starts receiving the 15-bit wide CRC sum and compares this with the calculated checksum in the send acknowledge state. If the calculated CRC sum matches the received one, the controller sets the dominant acknowledge bit. After the controller has received the Acknowledge Delimiter and the EOF field, it returns to the Intermission state. If the check sums do not match, the controller has received faulty data. As a result, no acknowledge bit is sent, but error handling is started after the ACK delimiter has been received. If the destuffing unit detects a violation of the stuffing rule or if the prescribed frame shape is not adhered due to a fault, e.g. the CRC or the acknowledged delimiter is overwritten by a dominant bit, the FSM also enters error handling.

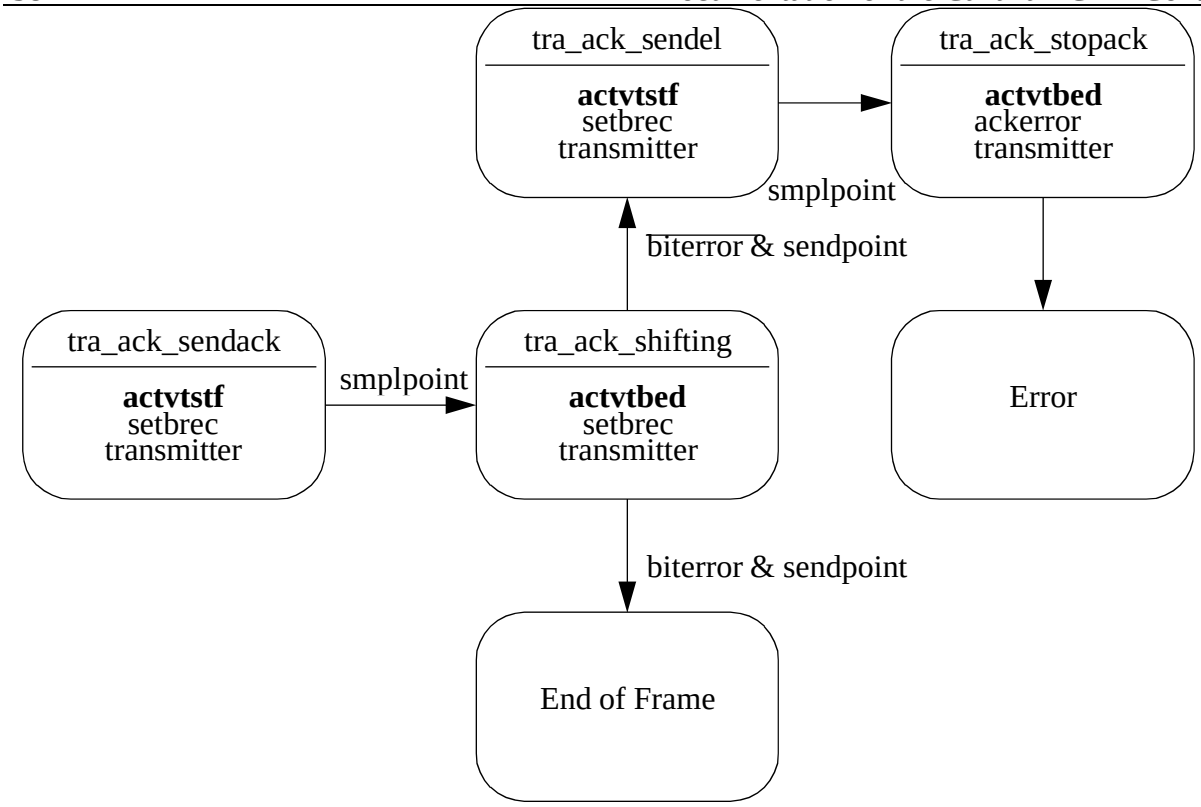


Figure 28: Acknowledge handling state transition diagram

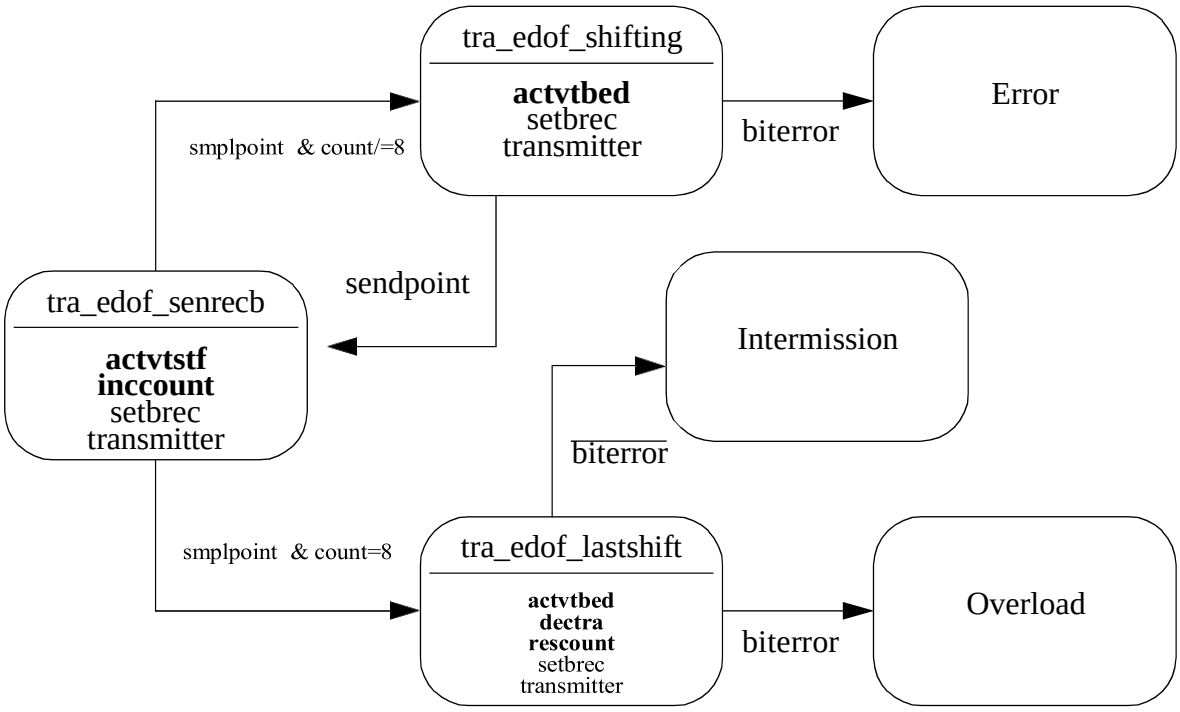


Figure 29: State transition diagram of the End of Frame frame

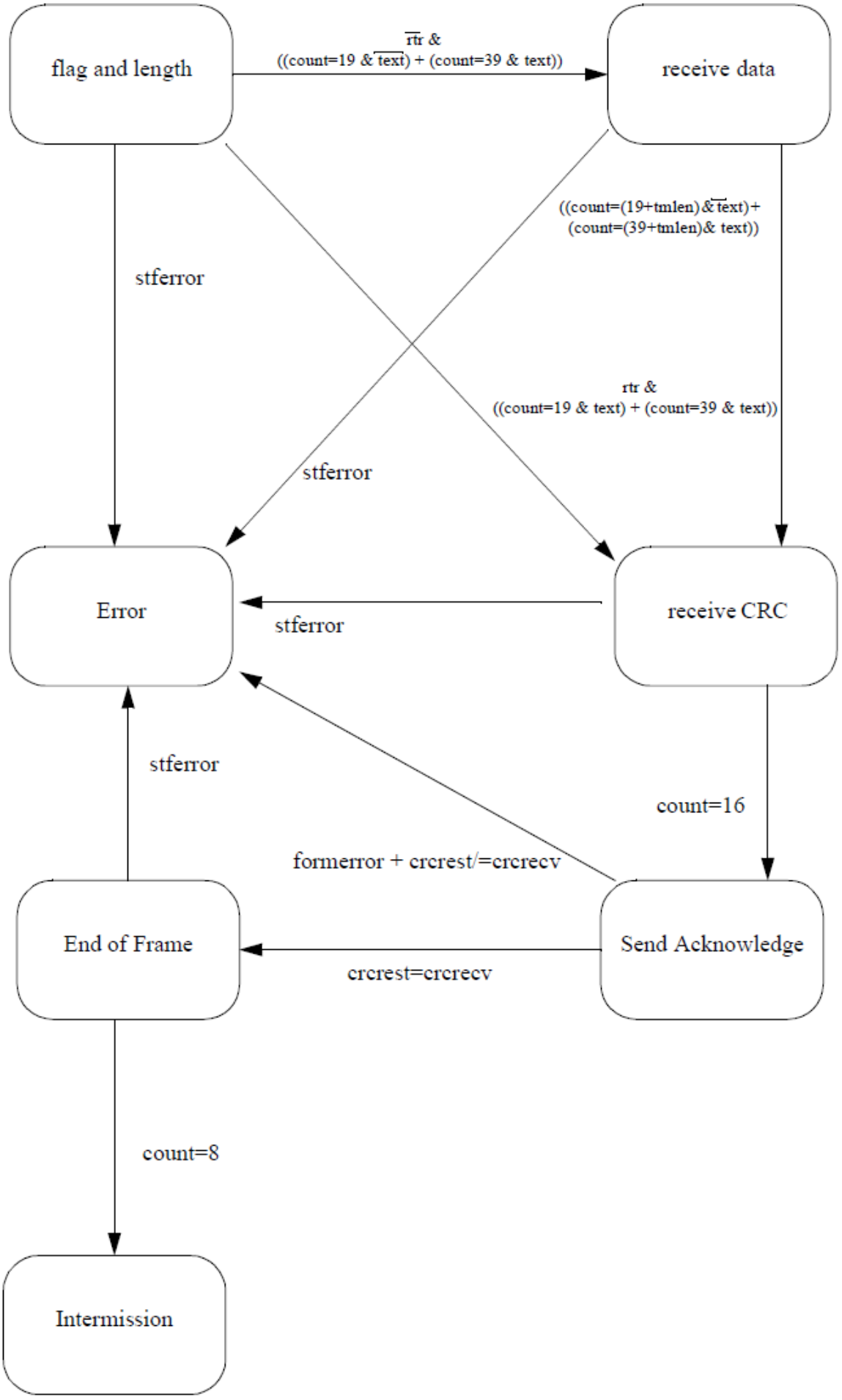


Figure 30: State transition diagram of the receive state

4.3.1 Flag and Length

As soon as a dominant bit is sampled in bus idle mode, the MAC FSM jumps to the **rec_flglen_shifting** state (Figure 31). The sampled bit is stored in the receive shift register (**actvrsft**) and fed to the CRC calculation (**actvrcrc**). The next bit is sampled at the next sample time in the state **rec_flglen_sample** (**actvrstf**). When a bit is sampled that contains important control information such as a frame type flag or a bit of the data length code, the FSM jumps to a state that is responsible for processing this particular bit. The frame type flags are handled by the states **shiftstdext**, **shiftextnor**, **shiftextrt**. If e.g. the 14th received bit is recessive, the frame is an extended frame. At the sampling time of the 14th bit, however, the counter has the value of 13 and only when the EXT bit is handled, the counter is incremented and thus receives the correct value of 14. For this reason, the transitions from the sample state to the states that deal with the control bits must be executed at a counter value that corresponds to the respective previous bit position. The incrementation of the counter is not executed in the sample state, so that stuffing bits are not counted. In the stuffing case, the FSM jumps to the **noshift** state, which leaves the counter unchanged and also does not activate the shift register and the crcre register. If during the reception of the control field a recessive bit is sampled that can be assigned to a bit of the DLC (Data Length Code), the FSM branches into one of the states **shiftdlc8**, **shiftdlc16**, **shiftdlc32** and **shiftdlc64**. In these states, the signal **setrmlen** is set according to the valence of the currently received bit with a value between one and four. In the **setdlc** state, the unit recmeslen is replaced by the signal **actvrmlen**, which reads the signal setrmlen and stores the data length. If a stuffing error is detected during sampling, the FSM branches to error handling. After the last bit of the control field is received, the flag and length state is exited. If a remote frame has been detected, receiption of the CRC sum is initiated. In the case of a standard frame, the user data reception is handled beforehand.

4.3.2 User data reception

The reception of data is quite straight forward. In **rec_acptdat_sample** the current bus value is read. In the stuffing case, the FSM goes into the state **rec_acptdat_noshift**. In that state, the FSM does not perform any actions because the stuffing bit is ignored. In the regular case, the received bit is written to the shift register and fed to the CRC unit. When the last data bit has been read, the FSM enters the lastshift state and resets the counter (Figure 32).

4.3.3 CRC sum reception

The reception of the CRC sum is handled similarly. The current bit is read (Figure 33) in the **rec_crc_sample** state. In the stuff case the bit is ignored. In the regular case, the bit is written to a separate area of the shift register. The signal lcrc (load CRC) tells the shift register that the bit does not belong to the rest of the data in the conventional shift register, but is part of the CRC sum, which is stored separately. If all 15 bits of the CRC field are received, the FSM enters the acknowledge state.

4.3.4 Acknowledge transmission

In the first state of the acknowledge handling rec_ack_recdelim the CRC delimiter is received. If a dominant bit is received instead of the recessive delimiter bit, the FSM starts error handling. In this state, the calculated CRC checksum is also compared with the received CRC sum. If the checksum matches, the controller acknowledges. Otherwise, the FSM sends a recessive bit and then switches to the error state. After the FSM has read the dominant acknowledge bit and checked the bus value in the next state, it must ensure that the dominant bit is taken back from the bus at the start of the next bit time. For this reason, the FSM sends a recessive bit in the rec_ack_stopack state. (setbrec, actvtstf).

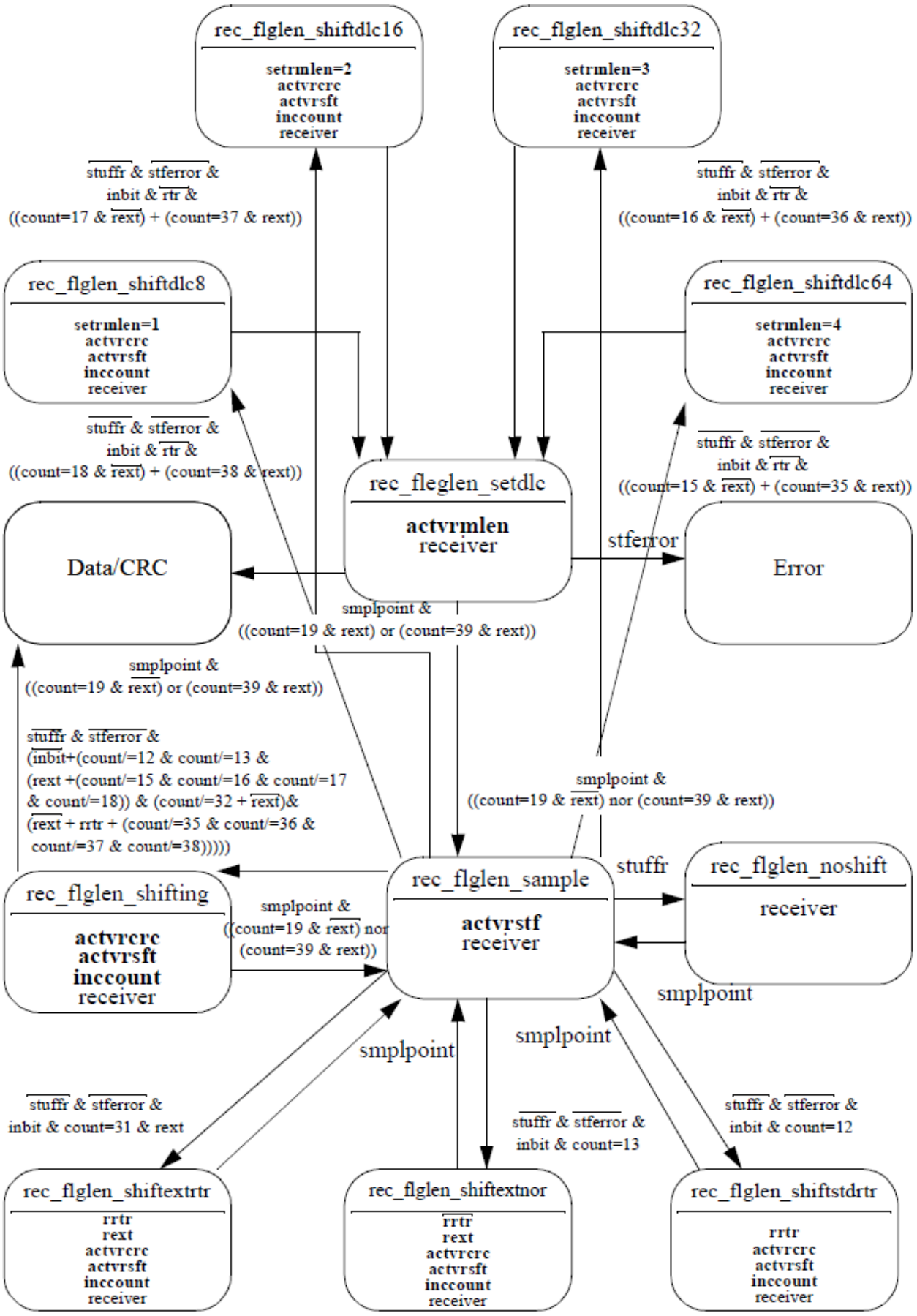


Figure 31: Flag and Length state transition diagram

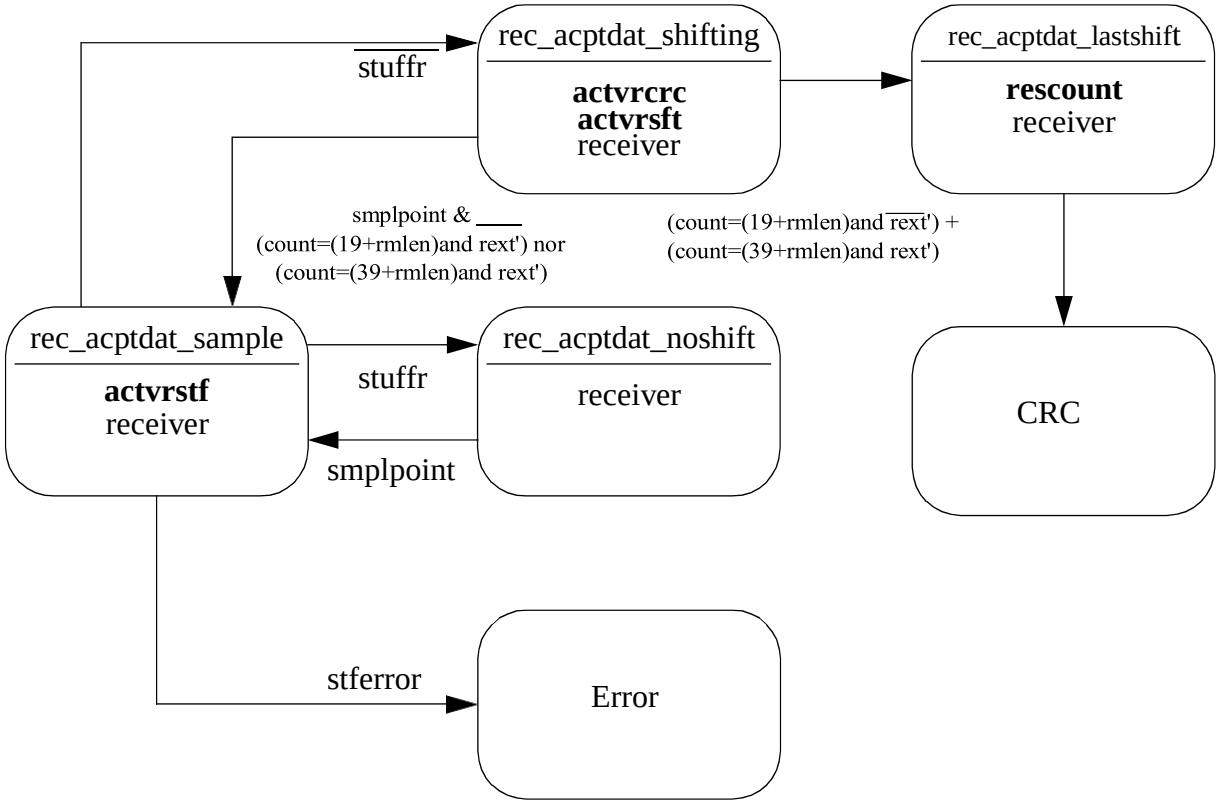


Figure 32: State transition diagram of the user data reception

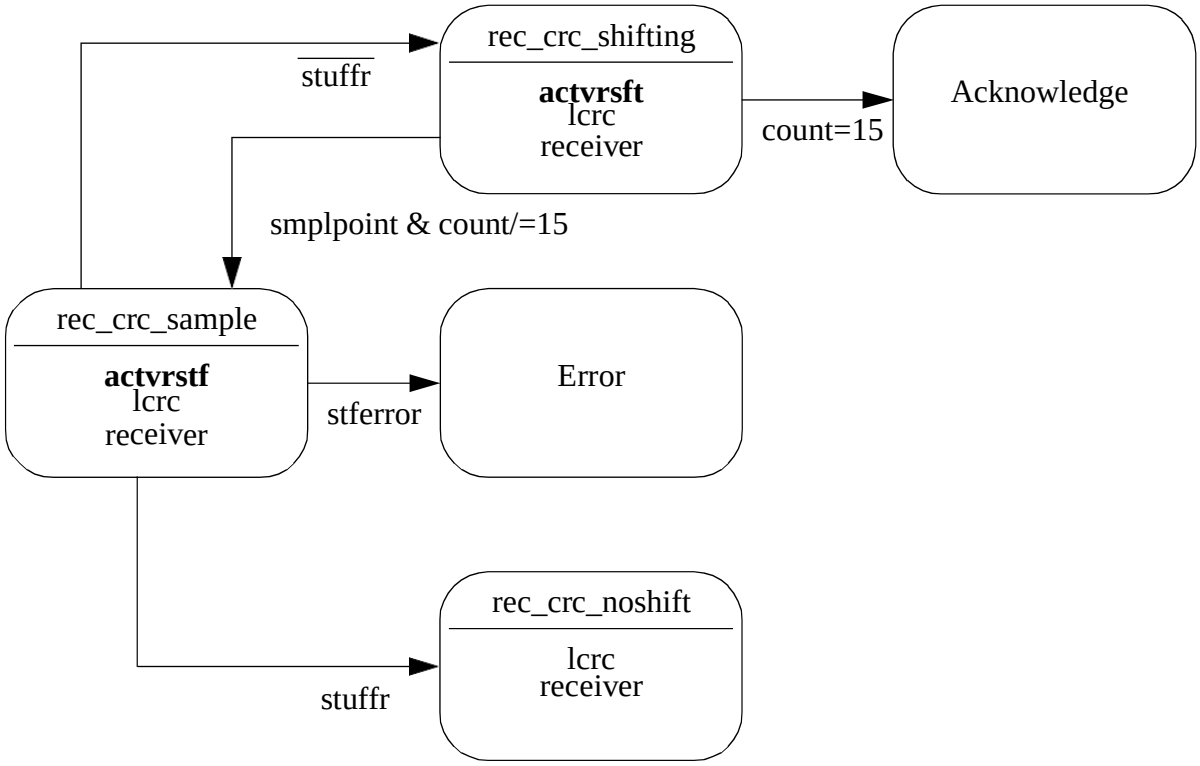


Figure 33: State transition diagram of the CRC reception

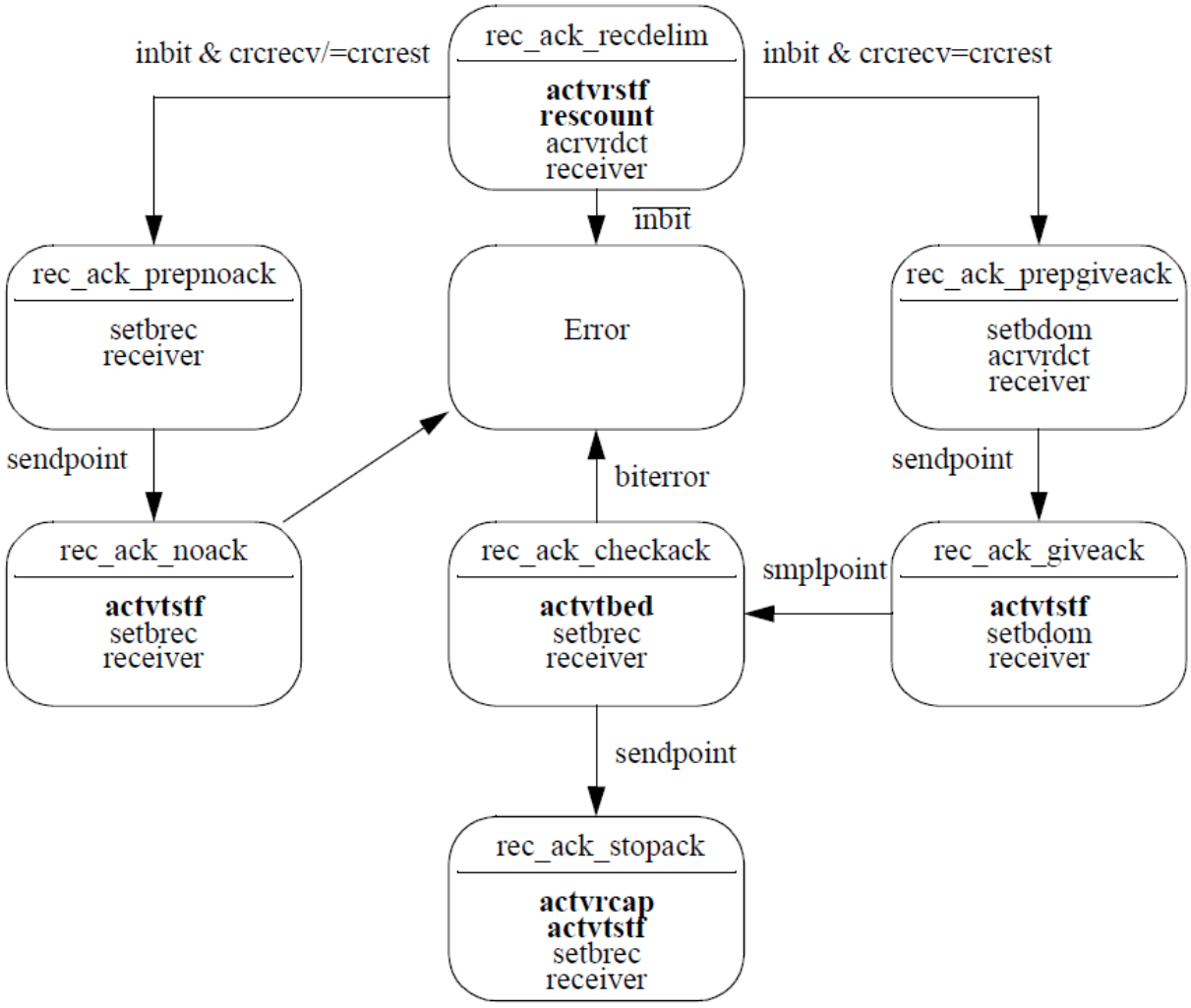


Figure 34: State transition diagram of the acknowledge transmit state

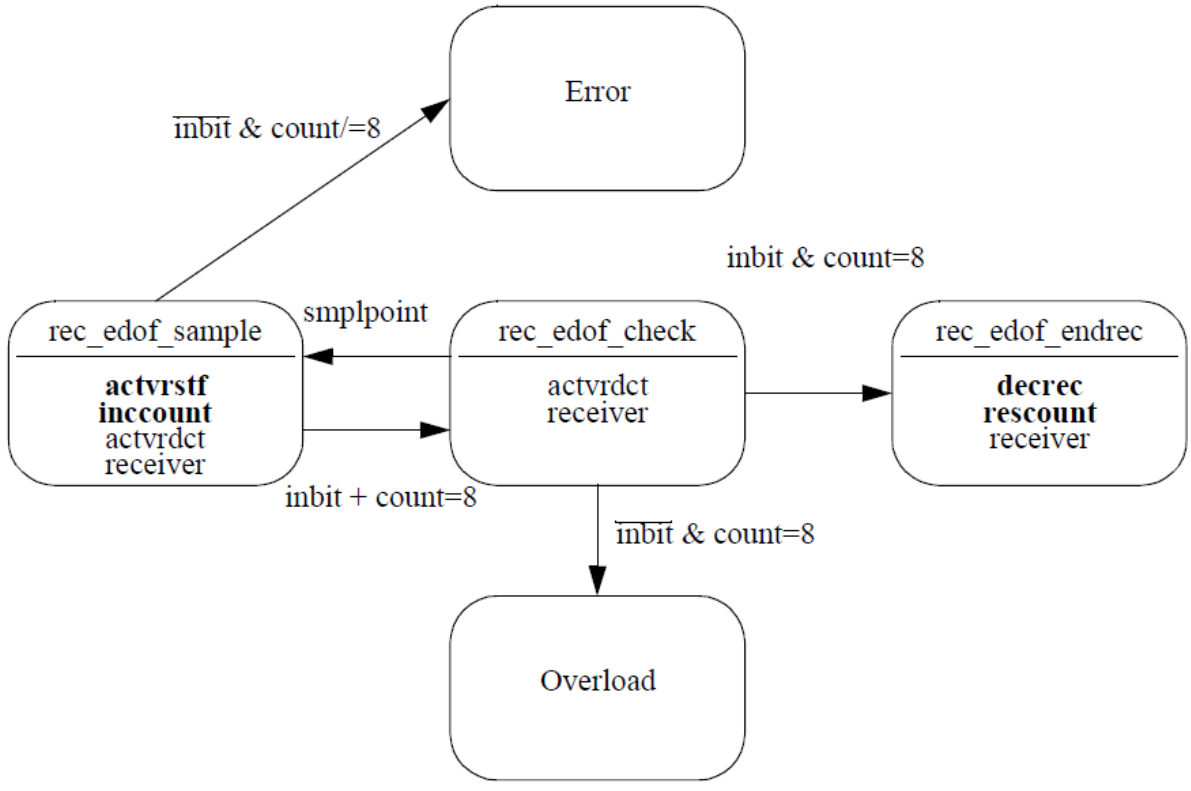


Figure 35: State transition diagram of end of frame reception

4.3.5 End of Frame reception

When handling the EOF frame, the FSM expects eight recessive bits, the acknowledge delimiter bit, and the 7th bit of the EOF flag. If a dominant bit is sampled during this flag, error handling is triggered. Except in the case of the last bit of the EOF field. A dominant bit in the last bit of the EOF field triggers an overload case. The FSM sets the actvrdct bit to notify the destuffing unit that the stuffing rule is not required to be obeyed. After receiving the 6th EOF bit, the FSM sends the decrec signal (decrement recessive error counter) to notify the LLC and Error Confinement Entity that a CAN frame has been successfully received.

5.4 Transmission of an Overload Frame

The Overload, the Error Active and the Error Active frame have the same structure. However, the Overload frame is the least complex, since the incrementation of the error counters does not have to be taken into account, at least at the beginning. It is advisable to cover the handling of the Overload and Error Active frames first, to easily understand the handling of the more complicated Error Passive frame. The transmission of an overload frame starts with the reset of the counter in the state **over_firstdom**. Furthermore, since the overload flag consists of six dominant bits, the stuffing unit is aligned to send dominant bits by setbdom (set bit dominant). In case of an error, the error signal is set to inform the error handling that the error occurred during the transmission of an overload or error flag. This makes it possible to fulfill error rules four and five. In the **over_sendomb** state, the bits are set on the bus. In the **over_check1** state, the level of the bus is checked and the number of transmitted bits is counted (inccount) (Figure 36). If six dominant bits were transmitted without errors, the controller is prepared for the transmission of the overload delimiters in the state **over_preprecb** (prepare for recessive bits). The counter is reset (rescount) and the stuffing unit is set to send recessive bits (setbrec). The controller now sends recessive bits and waits to detect a recessive bit on the bus. This is not necessarily the case at the first attempt, because as already described above, overload and error flags can be broadened by the overlapping of several such flags on the bus. In the state **overload_wtonrecb,** the controller sets a recessive bit on the bus. In the state **overload_check2,** the FSM checks whether the recessive bit has asserted itself on the bus. If a bit error has occurred, it transmits a recessive bit again. If the recessive bit was not overwritten, it retransmits seven more recessive bits in the **over_sendrecb** and **over_check3** states, thus terminating the overload delimiter and switching to the intermission sequence. According to the sixth error rule, the FSM must tolerate the overwriting of seven recessive bits in the check2 state. At the appearance of the eighth dominant bit and every eight further dominant bits, the error counters are incremented by eight. This happens in the states **over_increccounter** and **over_inctracounter** depending on whether the FSM was in the receive or transmit state when the overload condition was detected.

5.5 Transmission of an Error Active Frame

The handling of the Error Active frame starts in the **erroractiv_firstdom** state, where the counter is reset first so that it becomes available for later applications. In addition to the functions that the input state has when handling the overload frame, the first state of the Error Active and Error Passive frame handling decides whether the receive error counter or the transmit error counter must be incremented.

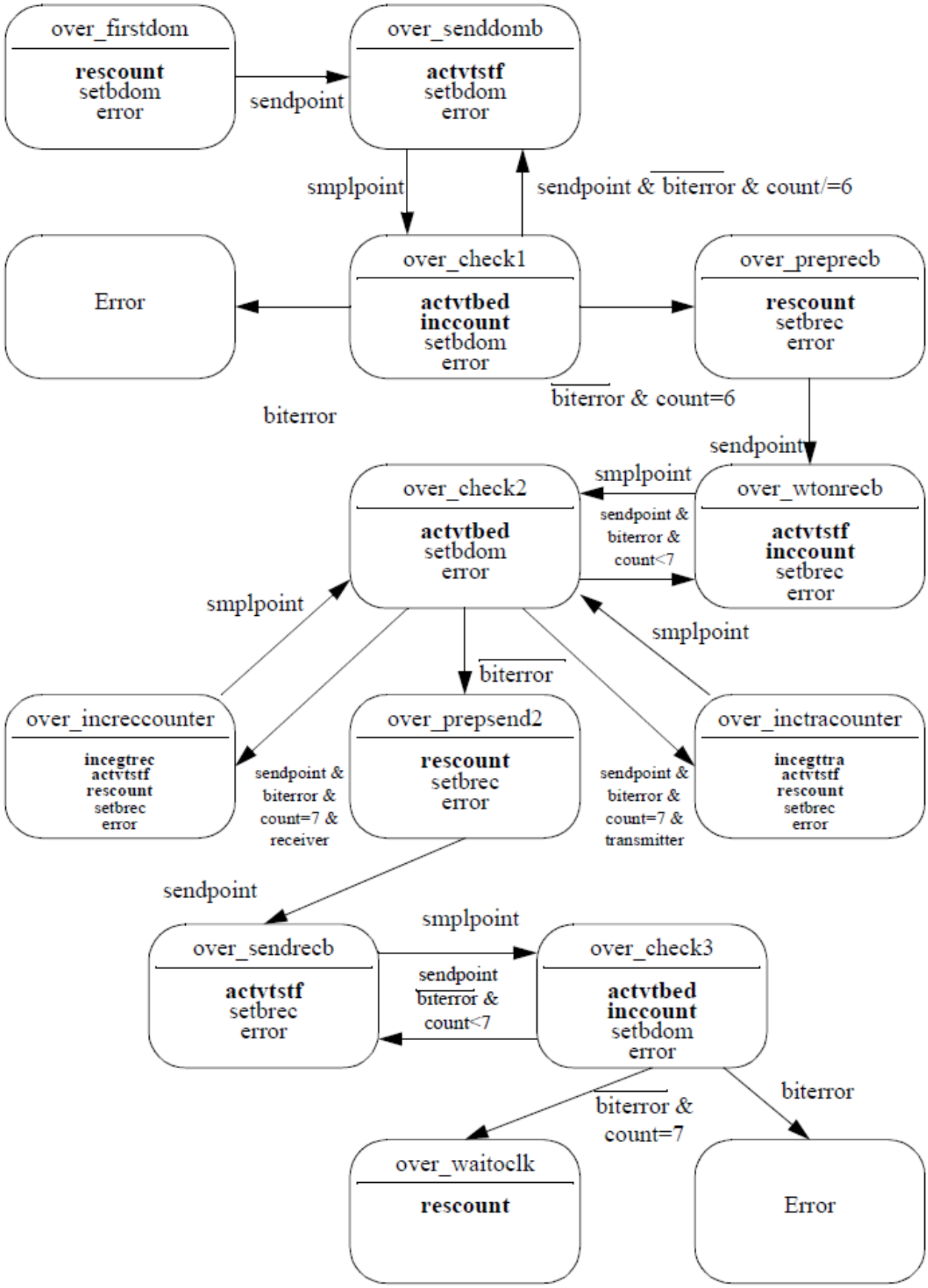


Figure 36: State transition diagram of the overload state

Thereby the signals transmitter, receiver and error are used. The signals transmitter and receiver are set in the send and receive state respectively. In addition, the error signal is set when an error or an overload condition occurs. Thus the MAC FSM can decide whether the transmit error counter must be incremented by eight in the **erroractiv_incachttra** state or whether the receive error counter must be incremented by one or eight in the **erroractiv_inceinsrec** and **erroractiv_incachtrec** states. According to the first error rule, the receive error counter is incremented by one if an error occurred during the reception of a message. If the recipient then starts sending an error frame and detects an error again, the receive error counter is incremented by eight. According to the second exception of the third error rule, the error counter of a transmitter is not incremented if a stuff error occurred during arbitration. This case is marked by the signal onarbit. However, there is the problem that the signals transmitter, receiver error and onarbit are overwritten with new values in the first state of the error handling erroractiv_firstdom before the branch to the corresponding increment case has been executed. One possibility would be to not include the signals transmitter, receiver error and onarbit in the erroractiv_firstdom at all and to assign neither a logical one nor a logical zero to them.

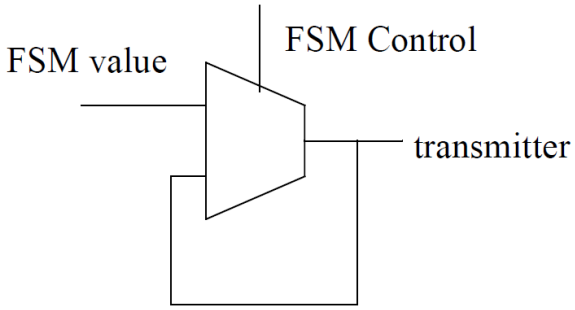


Figure 37: Multiplex with feedback

However, this would lead to the use of latches during synthesis. An alternative to this approach is to assign the previous value to these signals in the erroractiv_firstdom state (error<=error, transmitter<=transmitter, receiver<=receiver). Thus, instead of a latch, a feedback multiplexer is used in the synthesis. If a new value is explicitly assigned to the signals in a state, the FSM switches the new value to the output via the control input of the multiplexer, otherwise, the value is stored by the feedback of the output value into an input of the multiplexer. Now that the correct counter has been incremented, the six dominant bits of the Error Active flag are sent in the erroractiv_senddom and erroractiv_check1 states. This grey marked are in Figure 38 to make clear which area of the Error Active handling differs from the Error Passive state. Similar to the overload frame, the controller now places recessive bits on the bus and waits until the first recessive bit on the bus is no longer overwritten. However, the second error rule must be observed here. If a receiver receives a dominant bit as the first bit after sending an error flag, it must increment its receive error counter by eight. For this reason, the FSM enters the **erroractiv_dombitdct** state (dominant bit detection) if a bit error is registered at a counter value of one, i.e. at the first bit, and error handling has been initiated by a receiver. As before in the overload frame, seven dominant bits are tolerated before the error counters are incremented at the eighth dominant bit in the **erroractiv_increccounter** and **erroractiv_inctracounter** states, respectively. In these states also the counter is reset to be able to increment the error counters again when eight dominant bits appear again. However, it is obvious that a counter reading of one can be reached again but in this case the FSM should not branch to the **erroractiv_dombitdct** state as before.

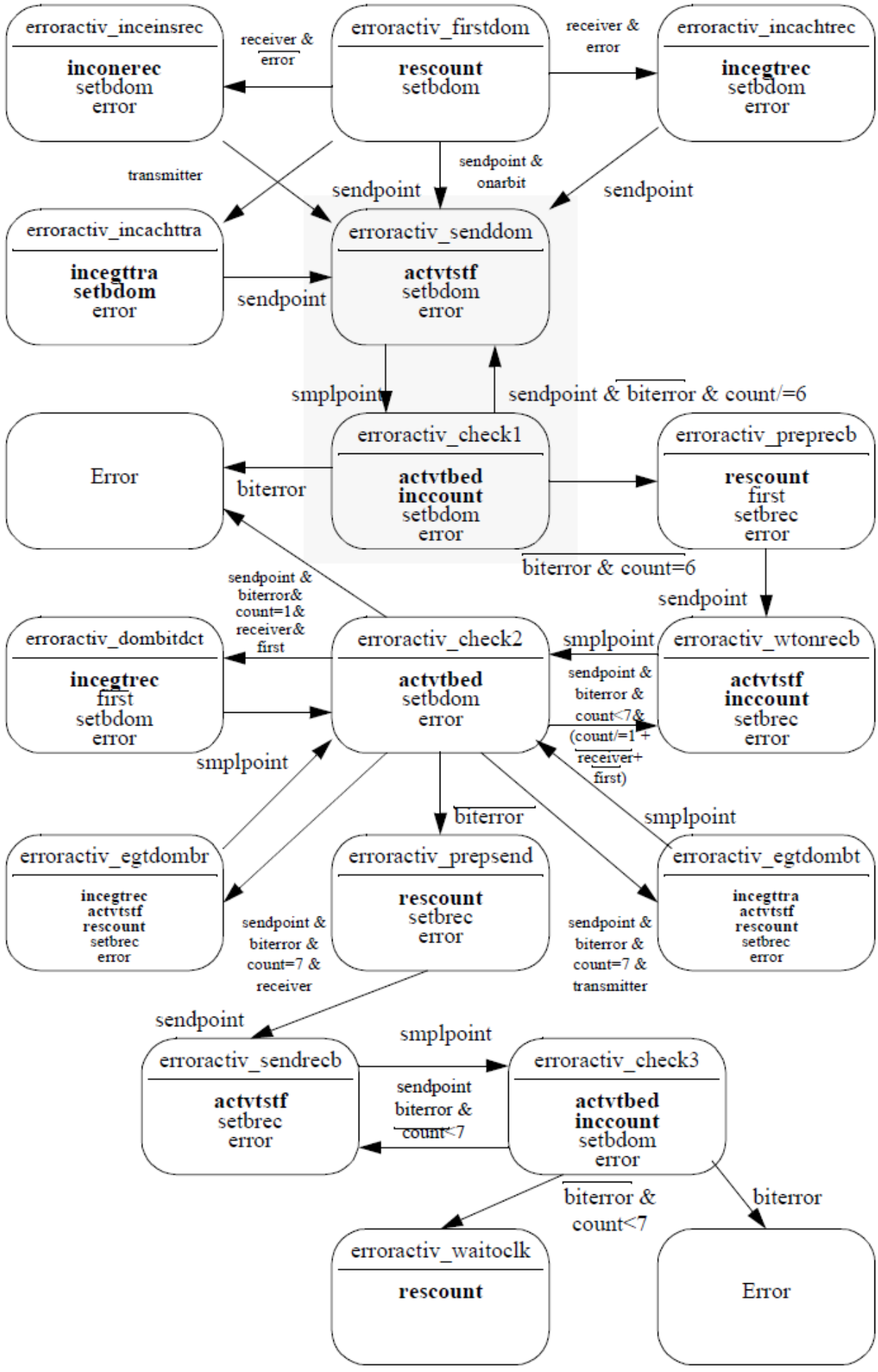


Figure 38: State tranistion diagram of the Error Active State

For this reason, another signal first is used, which provides information whether it is indeed the first bit after sending the Error Active flag which has been overwritten by a dominant bit and not the 9., 17., 25., etc. After the first pass, the first signal is reset and thus the transition to the erroractiv_dombitdct state is prevented. If a recessive bit was detected, the FSM sends another seven recessive bits similar to an overload frame, thus terminating the error delimiter and entering the Intermission state.

4.6 Transmission of an Error Passive Frame

The difference between Error Active and Error Passive frames is that Error Active flags are dominant and Error Passive flags are recessive. A controller must consider the transmission of its recessive Error Passive Flag as completed when it detects eight consecutive bits of the same value on the bus. It is irrelevant whether there are eight dominant bits or eight recessive bits. However, it is important that as soon as a bit is detected which is then inverted to its predecessor, the counting of the bits starts again. The handling of the Error Passive frame starts in the **errorpassiv_firstrec** state. Similar to the Error Active case, a decision is made as to which counter is to be incremented. In the Error Passive mode, however, there is another exception to the third error rule. This exception states that a transmitter in the Error Passive state will not increment its error counter if an acknowledge error has been detected and no dominant bit is detected during the transmission of the Passive Error Flag. For this reason, the FSM immediately goes to transmitting the recessive bits when the signal transmitter and the signal ackerror are set. This error rule prevents a controller from switching to bus off mode whenever no participants are on the CAN bus and thus cannot receive an acknowledge from a transmitting partner. In the errorpassiv_incsrecb state, the first recessive bit is placed on the bus. Subsequently, in the errorpassiv_fillpuffer state, the bus value is sampled and, depending on it, the buffer is filled, which enables the FSM to compare the currently received value with that of the previous bitycle. If a recessive bit is received from the bus, the FSM branches to the errorpassiv_pufferrec (buffer recessive) state to assign a recessive value to the buffer signal. If a dominant bit is detected, the buffer signal is set to a dominant value in the errorpassiv_pufferdom state. However, if the second exception of the third error rule was previously applied, in which the transmit counter was not incremented because an acknowledge error was detected, this must be done when a dominant bit is detected for the first time. For this reason, the FSM goes into the errorpassiv_pufferdomi state to increment the transmit counter and to withdraw the ackerror signal so that the transmit error counter is not incremented again if a dominant bit is again detected. After the buffer has been set for the first time, the FSM now tries to transmit the Error Passive flag until it detects five more consecutive bits with the polarity of the buffer signal. However, if a bit with a different value is received, the buffer value must be reset and the counting restarted. A bit with the same value as the buffer signal is present if the signal biterror is not equal to the signal puffer. It must be considered that the controller sends recessive bits and thus a bit error is set equal to one if there is a dominant bit on the bus. If a recessive bit is stored in buffer and a recessive bit is read from the bus, biterror has the value zero and puffer the value one and is thus unequal to bit error. If a dominant bit is stored in puffer and a dominant bit is read from the bus, biterror has the value one and puffer the value zero and is therefore again unequal to bit error. If a bit is received that has a different value than the signal stored in the buffer, the signals biterror and puffer are the same. In this case the FSM branches into one of the three states errorpassiv_zersrecbo, errorpassiv_zersrecbz and errorpassiv_zersrecbi. If a recessive bit was received, the buffer signal is set to one in the errorpassiv_zersrecbo state. If a dominant bit was received, the buffer signal is set to zero in the errorpassiv_zersrecbz state. When a dominant bit has been detected for the first time and the second exception of error rule three discussed above has been previously applied before, the increment of the transmit error counter must be made up, which is done in the erropassiv_zersrecbi state. If at some point six bits with the same polarity are detected on the bus, the error delimiter must be sent.

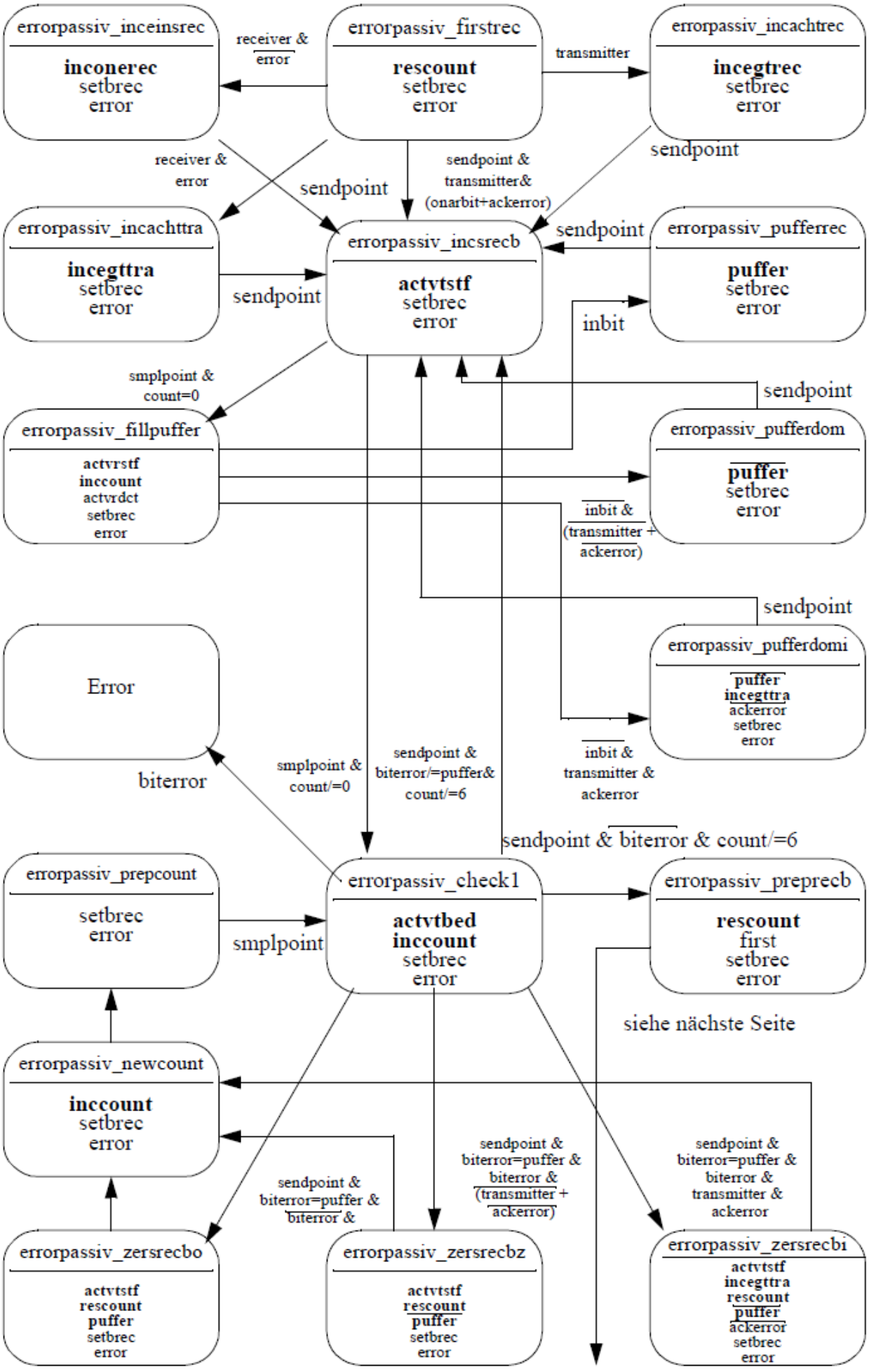
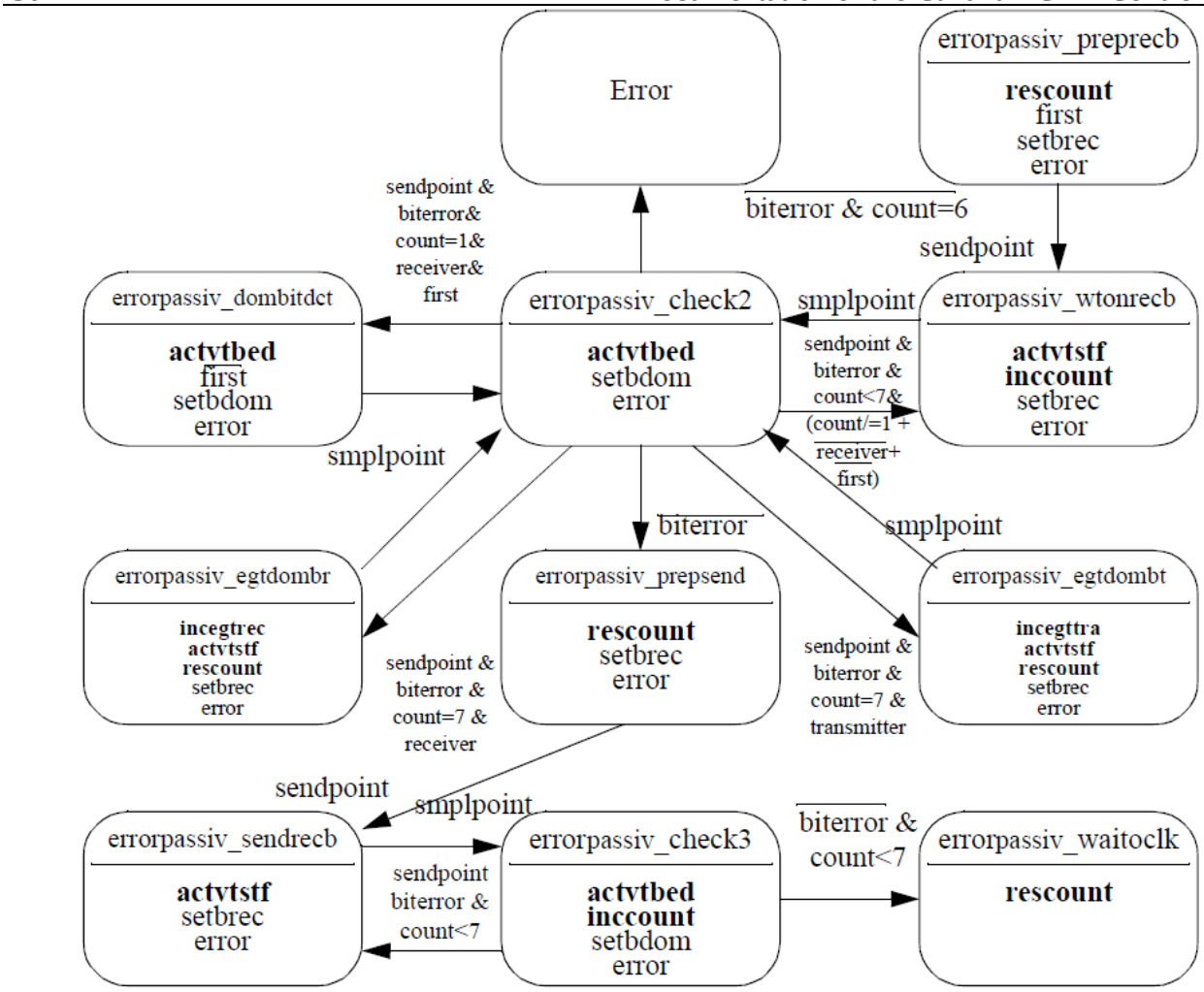


Figure 39: State transition diagram of the Error Passive state



Continuation Fig. 39: State transition diagram of the Error Passive state

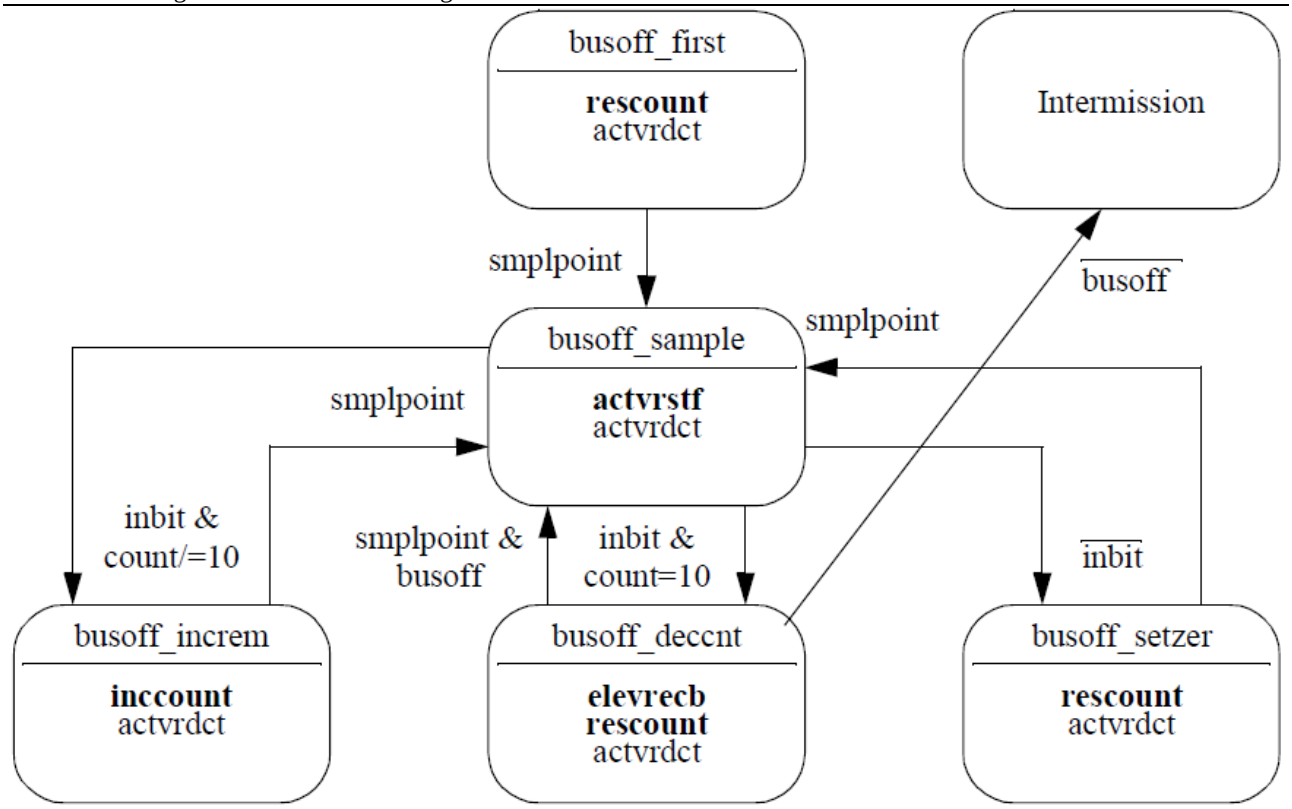


Figure 40: State transition diagram of the Bus Off state

During this process, the same steps are carried out as in the Error Active and Overload case. A recessive bit is placed on the bus until at some point a recessive bit can be read from the bus. Then seven more recessive bits are sent, ending the transmission of the error delimiter. After the error delimiter has been sent, the FSM returns to the intermission sequence.

4.7 Handling of the Bus Off mode

When the bus off error mode is reached, the FSM jumps to the busoff_first state. The counter then is reset and the destuffing unit is prepared with the signal actvrdct to disregard the stuffing rule. In the busoff_sample state, the current bit is read from the bus. If a recessive bit was sampled, the counter is incremented in the busoff_increm (increment) state. On the other hand, if a dominant bit was read from the bus, the counter is reset in the busoff_setzer (set zero) state. Whenever eleven recessive bits have been read from the bus, the Fault Confinement Unit is informed of this by the signal elevrecb (eleven recessive bits). If eleven consecutive recessive bits have been read on the bus 128 times, the Fault Confinement Unit resets the busoff signal and the FSM initiates the intermission sequence. If the busoff signal is still set, the FSM remains in bus-of mode and continues to search for eleven recessive bits.

5. Documentation of the remaining design components

5.1 Fault Confinement Unit

The Fault Confinement Unit consists of two components (Figure 41). The Error Counter takes the signals set by the Medium Access Controller and increments the corresponding counters. The counter readings are fed to a state machine that sets the corresponding fault state signals. The signal **inconerec** increments the receive error counter by one. The signal **incegtrec** increments the receive error counter by eight. The signal **incegttra** increments the transmit error counter by eight. The signal **elevrecb** increments the counter that counts the detection of eleven consecutive bits in the bus off state. The signal **decrec** decrements the receive error counter by one. The signal **dectra** decrements the transmit error counter by one. The error counter transfers the current error states to the FSM via the signals **tracounter** for the transmit error counter, **reccounter** for the receive error counter and **erbcounter** for the counter used in bus off state. The Fault Confinement FSM evaluates the error states and sets the signal corresponding to the current error state.

5.2 Timing Logic

The timing logic consists of a state machine that monitors the bus for falling edges, a counter which counts the current time in bit time quanta, an arithmetic component that processes the counter reading and the length of the bit segments and supplies the state machine with signals and values that it needs to make protocol-compliant synchronization decisions in the event of falling edges. The timing logic is fed with the signals tseg1, tseg2 and sjw (time segment 1, time segment 2, synchronization jump width). The time segment tseg1 comprises the propagation segment and phase segment 1. The time segment tseg2 corresponds to phase segment 2.

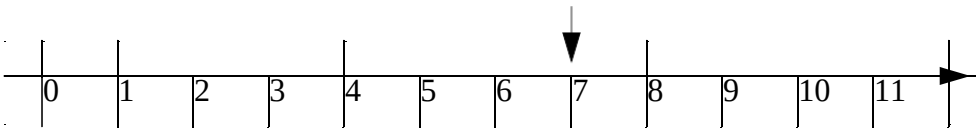


Figure 41: Number of basic time units at tseg1=5 and tseg2=4

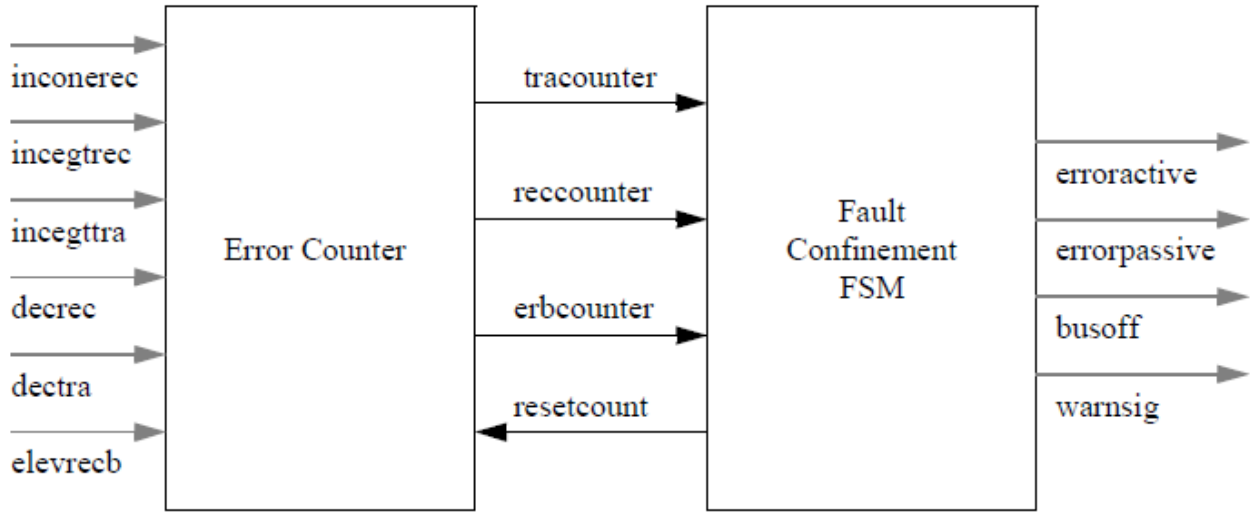


Figure 42: Structural design of the Fault Confinement Unit

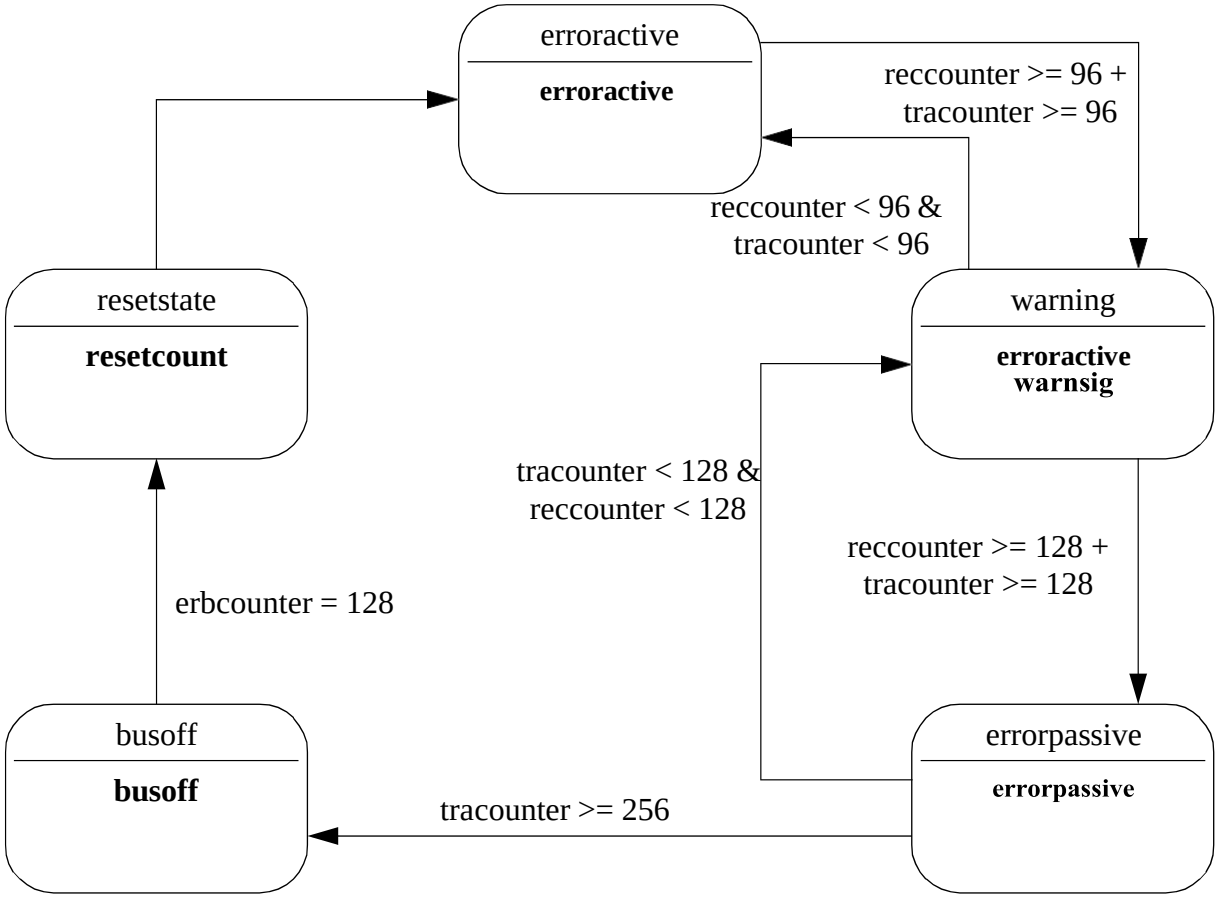


Figure 43: State transition diagram of the fault state machine

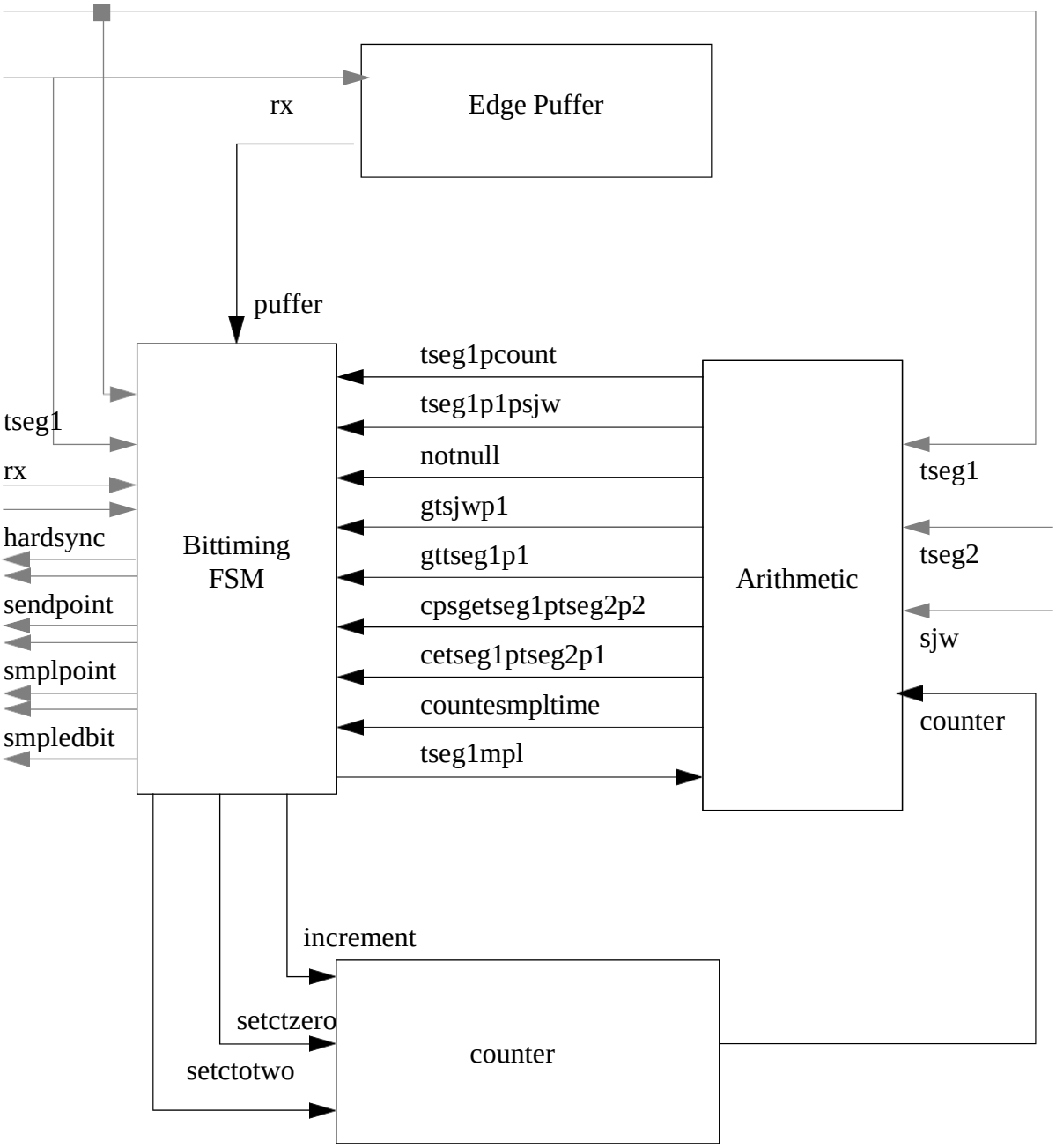


Figure 44: Structural design of the timing logic

The values of tseg1, tseg2 and sjw are written to the corresponding areas of the general register decremented by one for space reasons. I.e. if the time segment tseg1 consists of six basic time units, the number 5 is written into the register. The arithmetic unit sets some signals depending on the counter value. The signal **notnull** is always true if the counter reading has a value different from zero. This allows the Bittiming FSM to detect whether a falling edge has been detected on the bus outside the Synchronization Segment. The signal gtsjwp1 (greater than [sjw plus 1]) is set by the arithmetic unit if the counter reading is greater than the value of sjw plus one. The addition of one is necessary because the value of sjw is stored in the general register decremented by one. If the FSM detects a falling edge at a counter value that is smaller than the Synchronization Jump Width, the time segment tseg1 can simply be increased by the current counter value. Otherwise, the time segment may only be extended by the value of sjw. tseg1 is extended by the FSM assigning the values of the signals **tseg1pcount** at a counter reading smaller than the sjw or **tseg1p1psjw** at a counter reading larger than sjw.

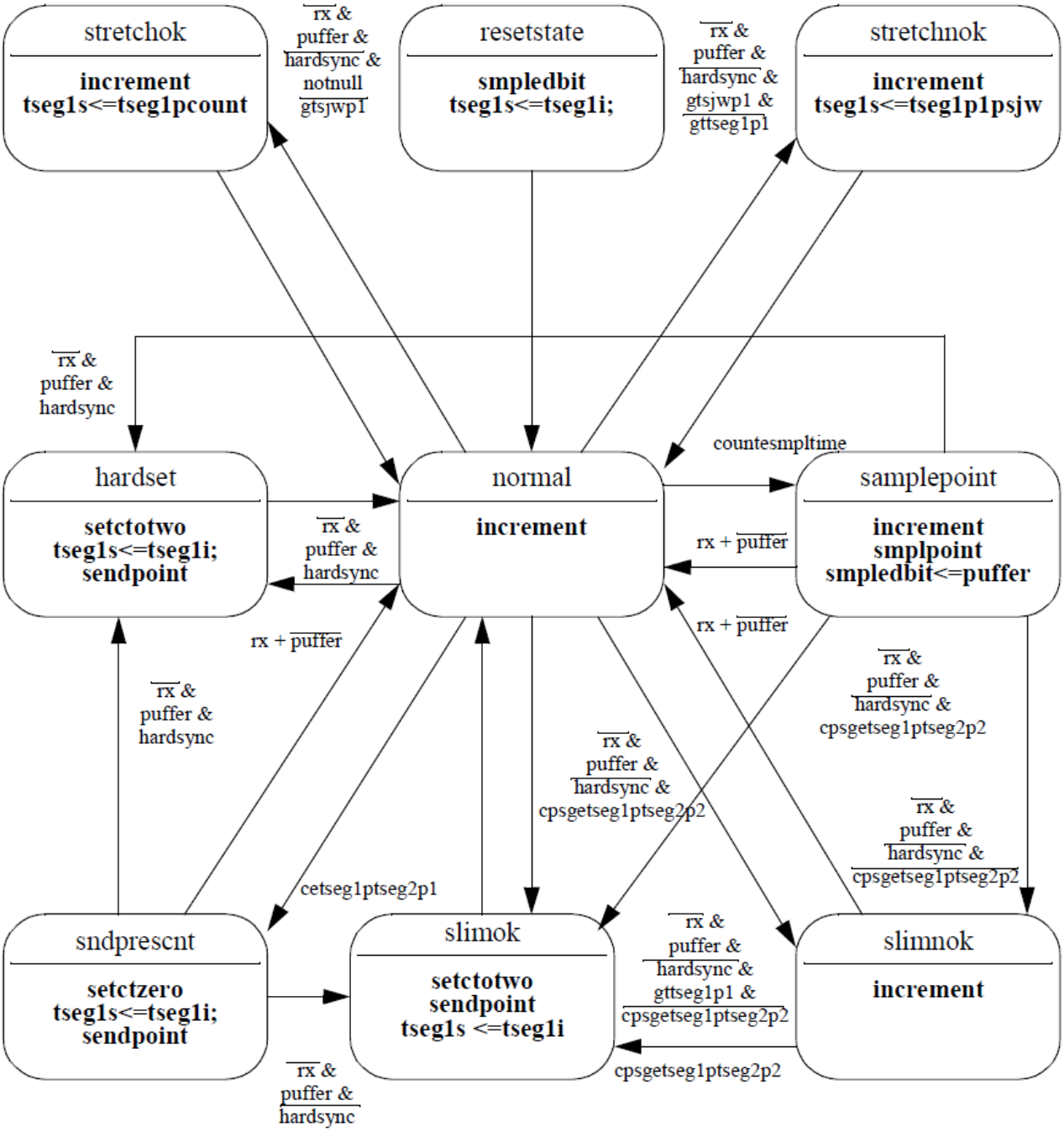


Figure 45: State transition diagram of the timing state machine

The new value of tseg1 is passed to the arithmetic unit via the signal **tseg1mpl** (tseg1 manipulated). The signal **countesmpltime** becomes true if the value of the counter corresponds to the sample time. The sample time is reached when the end of the tseg1 segment is reached (counter=tseg1+1). At this time the FSM sets the **smplpoint** signal and passes on the current bus level to the medium access controller. If the tseg1 segment has previously been extended due to a delayed falling edge, the sample time is also delayed by the same period. The signal **gttseg1p1** becomes true when the counter reading has exceeded the sample time and is therefore in the time segment tseg2. If the FSM now detects a falling edge, it shortens the segment tseg2. If the current counter reading is sjw or fewer time units away from the end of the time segment tseg2 when the falling edge is detected, the FSM can interpret the current position as a synchronization segment and thus jump to the beginning of the tseg1 segment.

If the current counter reading is more than sjw time units away from the end of time segment tseg2 when the falling edge is detected, it must not initiate the start of time segment tseg1 until the corresponding distance has been reached. For this purpose the signal **cpsgetseg1ptseg2p2** (counter+ sjw greater or equal tseg1 + tseg2 +2) is used. The signal becomes true if the distance to the end of the time segment tseg2 is sjw or fewer time units. If the end of the time segment tseg2 is reached regularly, i.e. without receiving an early falling edge, the FSM jumps to a special state in the last basic time unit, in which the counter is reset via the **setctzero** signal and the medium access controller is requested to place the next bit on the bus via the sendpoint signal. The state transition diagram shown in Figure 45 illustrates the use of the signals described. After a reset of the controller, the state machine changes from the state **resetstate** to the state **normal**. In this state, the counter is incremented for each clock pulse and waits for falling edges. A falling edge manifests itself by the current bus value Rx having the value zero and the bus value stored in the buffer at the beginning of the basic time unit having the value one (puffer='1' and rx='0'). If such a falling edge is detected while the **hardsync** signal is activated, the hard synchronization is performed in the hardset state. With a hard synchronization, the basic time segment in which the falling edge occurred is regarded as the Synchronization Segment. The synchronization segment is normally assigned to the counter reading zero. However, the counter status can only be influenced one clock unit later in the **hardset** state. At this time the counter should already contain the value one. For this reason, the FSM sets the signal setctotwo (set counter to two), which sets the counter with the next clock pulse to the value 2. If the hardsync signal is not set and a falling edge occurs, the time segment tseg1 is extended (stretchok, stretchnok) or the time segment tseg2 is shortened depending on the signals discussed above. If the error edge occurs during a non-zero (notzero) counter reading and is less than or equal to the sjw (gtsjwp1=0), the time segment tseg1 is extended according to the counter value in the **stretchok** state. However, if the counter value is greater than the value of the sjw (gtsjwp1=1) and the sample time has not yet been exceeded (gttseg1p1=0), then the time segment tseg1 in the **stretchchnok** state is extended by the value of the sjw. At sample time, the FSM enters the **samplepoint** state, sets the smplpoint signal and forwards the current bus value to the medium access controller (smpledbit<=buffer). At the last but one basic time unit of the time segment tseg2, the **cetseg1ptseg2p1** signal is activated by the arithmetic unit so that the FSM is promoted to the **sndprescnt** state at the next clock pulse. If the FSM has entered this state, the last basic time unit of the time segment tseg2 has already started. For this reason, this is the right time to set the signal **setctzero**, which resets the counter value to zero at the next clock pulse. Furthermore, in this state, the medium access controller must be informed via the signal **sendpoint** that it can put the next bit on the bus. However, if an early falling edge occurs, the state machine responds to the edge depending on the distance to the end of time segment tseg2. If the end of the time segment is sjw or less basic time units away (cpsgetseg1ptseg2p2), the FSM jumps to the slimok state. The basic time unit in which the falling edge was registered is subsequently assigned to the counter reading zero. As a result, the counter reading should have the value one while the FSM is in the slimok state. However, since the counter reading does not correspond to these values, the FSM sets the signal setctotwo in the **slimok** state, whereby the counter adopts the value two at the next clock pulse and the fault is thus corrected. If the end of the time segment tsge2 is more than sjw basic time units away, but the sample time has already passed (cpsgetseg1ptseg2p2=0 and gttseg1p1=1), then the FSM waits in the **slimnok** state until the correct distance to the end of the time segment has been reached (cpsgetseg1ptseg2p2=1) and then goes into the slimok state to initiate the shortening of the time segment tseg2. Since it cannot be ruled out that a falling edge occurs during the stay in state sndprescnt state, i.e. when the counter reading corresponds to the last basic time unit of the time segment tseg2, care must be taken to ensure that it is possible to jump to the hardset state if the hardsync signal is set, or to the slimok state if the hard-sync signal is not set. The same applies to the samplepoint state, which can also contain a falling edge. This would lead to the shortening of the segment tseg2.

5.3 CPU Interface Logik

The CPU interface logic consists of a unit that controls the bus protocol and fifteen registers. Figure 46 shows the structure of the CPU interface logic using the transmit control register as an example. The unit called multiplexer reacts to the signals swrite and sread by reading the address on the address bus and by writing the data from the corresponding register to the computer bus or by writing the information read from the data bus to the addressed register. In the particular case of the transmit control register, the register contents can be read from the multiplexer by the signal **traconr**. (The r at the end of the signal name stands for read, the w for write and the a stands for activate). Via the signal **traconw** the values read from the bus are applied to the inputs of the register. By the signal **tracona**, the data from the signal traconw are taken over into the register. Since also the LLC unit must have the possibility to write into the transmission control register, e.g. to mark the successful transmission of a CAN frame, there are specific signals, which carry the data which is stored in the register and can enable data take over in the register. In addition, the controller can read the contents of the register via the signal traconr. In the same way, all fourteen further registers are connected with the multiplexer unit and the controller, so that an interface between controller and CPU is established.

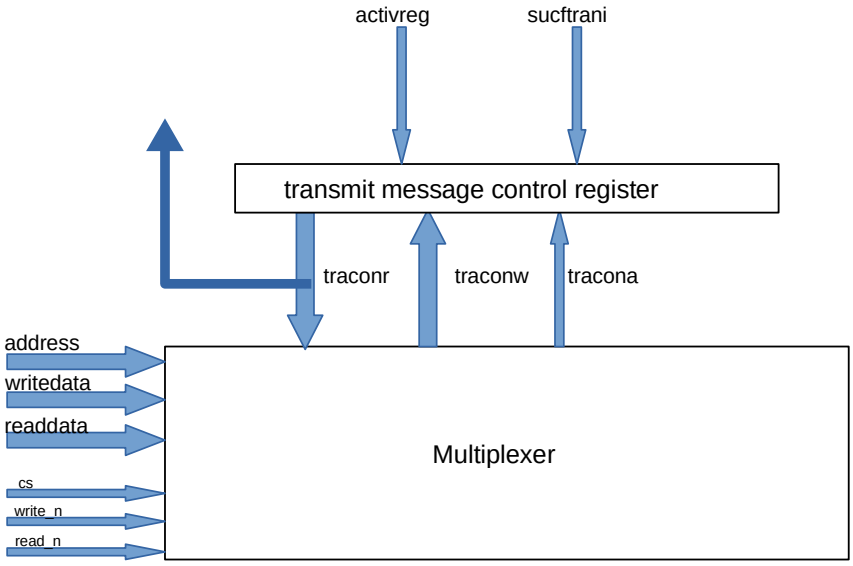


Figure 46: Sketch of the structure of the IO Control Unit using the example of the transmit control register

5.3.1 Avalon Interface

The Avalon Interface is a simple parallel interface that can be used to directly write to and read from the registers of the controller. The interface consists of the control signal cs and the data signals address, readdata, writedata, read_n and write_n. To read or write to a specific register the related data signals must be set to the correct values before cs gets pulled up. Pulling up the cs signal will result in the controller reading in the provided data on the other ports. With exception of the readdata port. This port is read only and will provide you with the data of the requested register address in a read command. In the following table an overview of the Avalon Interface ports is shown.

Name	Type	Bits	Description
cs	input	1	Chip Select. Set to 1 to read in the data on the other ports.
write_n	input	1	If write command, set to 0 (low active).
read_n	input	1	If read command, set to 0 (low active).
address	input	5	Register address of the target register.
writedata	input	16	Data to be written to the target register.
readdata	output	16	Requested data of the target register.

When a register of the CANakari is accessed, either a write or a read command can be applied. To specify the command type the related low active signal write_n or read_n is set to 0. Only one of them should be set to 0 at the same time . To write data into a register also the address and the writedata bus is set. If a read command is used the address is set and the writedata signal can driven with random data, because it is not used. In general it is set to zero. The CANakari design does not use the full available register address space. So caution has to be applied to not accidently access a non-existing address. The following two figures show the signal flow for a write and a read command. From the figures it can be seen that at first, the necessary data is set (t=2) before the chip select line is pulled up to read in the data by the controller (t=3). Using a read command, it can be expected that the register data to be valid by the next rising clock edge (t=4). At this point of time the input data signals should be set to a default value like zero and should be kept at least an intermission time of one clock cycle before starting next the transmission at t=5. Data is read and set data on rising edges of the clk. In the figures it is expected that the CANakari CAN-Controller and the chip on the user side (µC, FPGA, ...) share the same system clock (clk) signal.

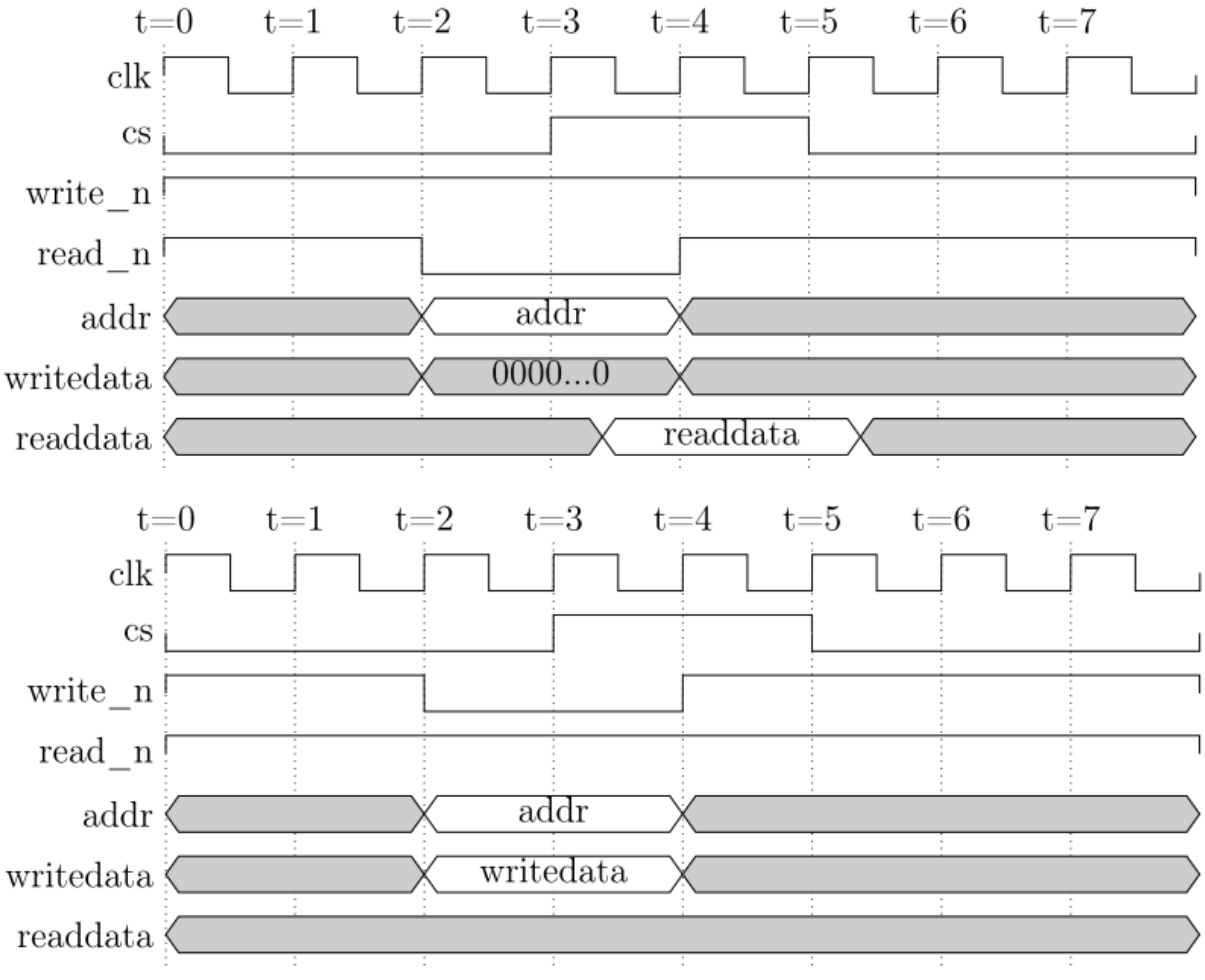


Figure 47: Avalon Interface Read/Write

5.3.2 Register set

The register set consists of the general register, which contains information that is of global importance for the function of the controller, seven transmit data registers and seven receive data registers. A comprehensive list of the registers is given in Figure 48 and more details are given in the following sections. Through the registers it is possible to configure the controller and send/receive CAN-messages. The registers 0x13 and 0x14 are read only. The first one contains the current TEC and REC values combined in one register (0x13) and the second one contains the System ID that is set internally in the top level of the CANakari and is meant to identify the chip, but is not necessary for the communication by any means.

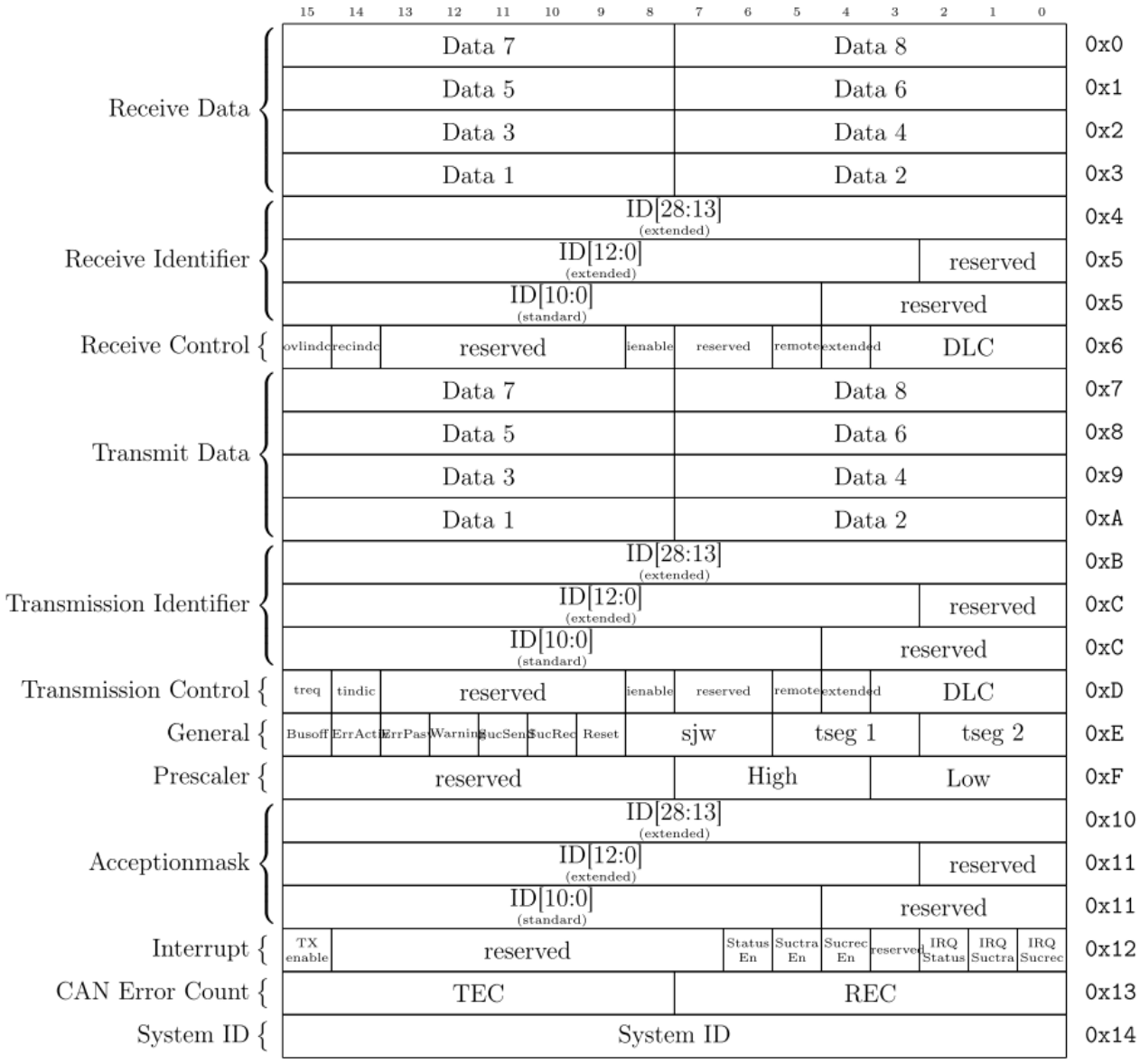
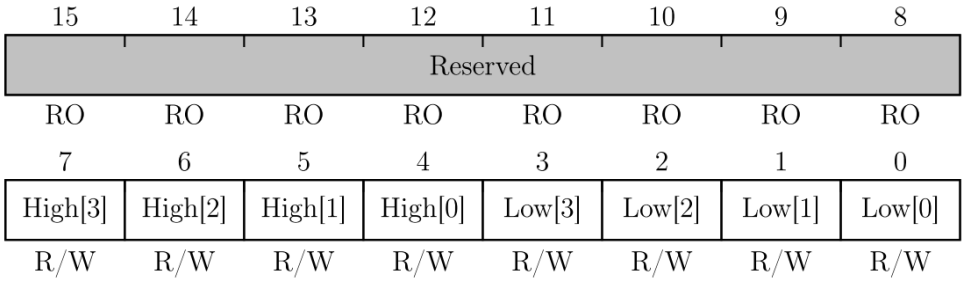


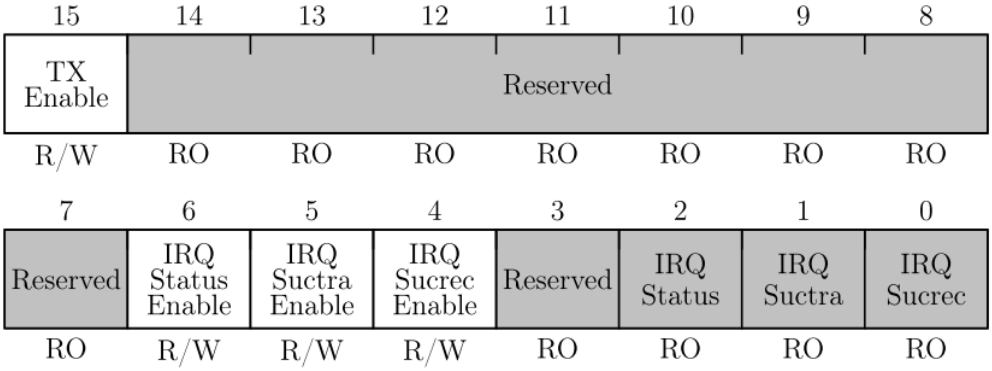
Figure 48: Registers of the CANakari CAN Controller

5.4 Prescaler Register



- **High**: Prescale value related to the high-level part of the clock.
- **Low**: Prescale value related to the low-level part of the clock.

5.5 IRQ-Register / Interrupt Register



- **TX Enable**: Enable flag to activate transmission functionality for the physical layer.
- **IRQ Status Enable**: Enables the interrupt to inform the user about a change in the status of the controller.
- **IRQ Suctra Enable**: Enables the interrupt to inform the user about a successful transmission.
- **IRQ Sucrec Enable**: Enables the interrupt to inform the user about a received can message.
- **IRQ Status**: Bit is set by the controller if the can controller changes its state. This field is connected to the corresponding interrupt port and must be reset after every interrupt to recognize the next one.
- **IRQ Suctra**: Bit is set by the controller if a transmission has been successfully executed. This field is connected to the corresponding interrupt port and must be reset after every interrupt to recognize the next one.
- **IRQ Sucrec**: Bit is set by the controller if the can controller receives a can message. This field is connected to the corresponding interrupt port and must be reset after every interrupt to recognize the next one.

5.6 Acception Mask Register

F	E	D	C	B	A	9	8
idMask(28)	idMask(27)	idMask(26)	idMask(25)	idMask(24)	idMask(23)	idMask(22)	idMask(21)
7	6	5	4	3	2	1	0
idMask(20)	idMask(19)	idMask(18)	idMask(17)	idMask(16)	idMask(15)	idMask(14)	idMask(13)

- Contains the stored masked identifier bits.

F	E	D	C	B	A	9	8
idMask(12)	idMask(11)	idMask(10)	idMask(9)	idMask(8)	idMask(7)	idMask(6)	idMask(5)
7	6	5	4	3	2	1	0
idMask(4)	idMask(3)	idMask(2	idMask(1)	idMask(0)	reserved	reserved	reserved

- Contains the stored masked identifier bits.

5.7 General Register

F	E	D	C	B	A	9	8
Busoff	ErrActiv	ErrPasv	Warning	SucSend	SucRec	Reset	sjw(2)
7	6	5	4	3	2	1	0
sjw(1)	sjw(0)	tseg1(2)	tseg1(1)	tseg1(0)	tseg2(2)	tseg2(1)	tseg2(0)

- **Busoff, ErrActv, ErrPasv**: These bits correspond to the possible error states of the can controller whereby the current state field contains a 1. Fields are updated on every clock cycle.
- **Warning**: This field is set by the can controller if one of the error counters (rec, tec) has reached a value of 96.
- **SucSend**: This field states that since the last reset of this bit a successful transmission has been executed.
- **SucRec**: Indicates that at least one message has been successfully received since the last reset of this bit.
- **Reset**: If this field is set to 1 the can controller will start an internal reset (soft reset).
- **sjw:** Used to configure the synchronization jump width for the can bit-timing.
- **tseg1**: Used to configure the timing segment 1 (Caution, value has to be set as tseg-1).
- **tseg2**: Used to configure the timing segment 2 (Caution, value has to be set as tseg-1).

5.8 Transmission Control / Transmit Message Control Register

F	E	D	C	B	A	9	8
treq	tindic	reserved	reserved	reserved	reserved	reserved	ienable
7	6	5	4	3	2	1	0
reserved	reserved	remote	extended	DLC3	DLC2	DLC1	DLC0

- **treq:** Set to initiate the transfer of the data in the transmit arbitration and transmit data register.
- **tindic:** Indicates that since the last reset of this bit a successful transmission has been executed.
- **ienable**: This field was supposed to enable an interrupt in case of successful transmission (not supported).
- **Remote, extended**: Specifies the frame type.
- **DLC3-DLC0**: Contains data length information, i.e. the number of user data bytes to be sent.

5.9 Transmission Identifier Register 1: Bits 28 – 13

F	E	D	C	B	A	9	8
ident(28)	ident(27)	ident(26)	ident(25)	ident(24)	ident(23)	ident(22)	ident(21)
7	6	5	4	3	2	1	0
ident(20)	ident(19)	ident(18)	ident(17)	ident(16)	ident(15)	ident(14)	ident(13)

- Contains the 16 most significant bits of the identifier of the frame that is to be sent.

5.10 Transmission Identifier Register 2: Bits 12 – 0

F	E	D	C	B	A	9	8
ident(12)	ident(11)	ident(10)	ident(9)	ident(8)	ident(7)	ident(6)	ident(5)
7	6	5	4	3	2	1	0
ident(4)	ident(3)	ident(2	ident(1)	ident(0)	reserved	reserved	reserved

- Contains the 13 least significant bits of the identifier of the frame to be sent (extended).
- Not used in case of base identifier.
- The last three bits (reserved) are not needed, can be set to 0.

5.11 Transmission Data Registers

transmit data1 + data2 register

F	E	D	C	B	A	9	8
data1(7)	data1(6)	data1(5)	data1(4)	data1(3)	data1(2)	data1(1)	data1(0)
7	6	5	4	3	2	1	0
data2(7)	data2(6)	data2(5)	data2(4)	data2(3)	data2(2)	data2(1)	data2(0)

- Contains the transmit user data bytes 1 and 2.

transmit data3 + data4 register

F	E	D	C	B	A	9	8
data3(7)	data3(6)	data3(5)	data3(4)	data3(3)	data3(2)	data3(1)	data3(0)
7	6	5	4	3	2	1	0
data4(7)	data4(6)	data4(5)	data4(4)	data4(3)	data4(2)	data4(1)	data4(0)

- Contains the transmit user data bytes 3 and 4.

transmit data5 + data6 register

F	E	D	C	B	A	9	8
data5(7)	data5(6)	data5(5)	data5(4)	data5(3)	data5(2)	data5(1)	data5(0)
7	6	5	4	3	2	1	0
data6(7)	data6(6)	data6(5)	data6(4)	data6(3)	data6(2)	data6(1)	data6(0)

- Contains the transmit user data bytes 5 and 6.

transmit data7 + data8 register

F	E	D	C	B	A	9	8
data7(7)	data7(6)	data7(5)	data7(4)	data7(3)	data7(2)	data7(1)	data7(0)
7	6	5	4	3	2	1	0
data87)	data8(6)	data8(5)	data8(4)	data8(3)	data8(2)	data8(1)	data8(0)

- Contains the transmit user data bytes 7 and 8.

5.12 Receive Control Register

F	E	D	C	B	A	9	8
ovflindc	recindc	reserved	reserved	reserved	reserved	reserved	ienable
7	6	5	4	3	2	1	0
reserved	reserved	remote	extended	DLC3	DLC2	DLC1	DLC0

- **Ovflindc**: This field informs the user that the receive data registers have been overwritten before the latest message has been read out by the user.
- **recindc**: Indicates that since the last reset of this bit a can message has been received.
- **ienable**: This field was supposed to enable an interrupt if a can message was received (not supported).
- **remote, extended**: Specifies the frame type.
- **DLC3-DLC0**: Contains the received data length code information. This determines how many data receive registers must be read out.

5.13 Receive Identifier Register 1: Bit 28 – 13 register

F ident(28)	E ident(27)	D ident(26)	C ident(25)	B ident(24)	A ident(23)	9 ident(22)	8 ident(21)
7 ident(20)	6 ident(19)	5 ident(18)	4 ident(17)	3 ident(16)	2 ident(15)	1 ident(14)	0 ident(13)

- Is used to store the base identifier or the upper half of the extended identifier.

5.14 Receive Identifier Register 2: Bit 12 – 0 register

F ident(12)	E ident(11)	D ident(10)	C ident(9)	B ident(8)	A ident(7)	9 ident(6)	8 ident(5)
7 ident(4)	6 ident(3)	5 ident(2	4 ident(1)	3 ident(0)	2 reserved	1 reserved	0 reserved

- Is used to store the lower half of the extended identifier.
- Not used in case of base identifier.
- The last three bits (reserved) are not relevant.

5.15 Receive Data Registers:

receive data1 + data2 register

F data1(7)	E data1(6)	D data1(5)	C data1(4)	B data1(3)	A data1(2)	9 data1(1)	8 data1(0)
7 data2(7)	6 data2(6)	5 data2(5)	4 data2(4)	3 data2(3)	2 data2(2)	1 data2(1)	0 data2(0)

- Contains the received data for the bytes 1 and 2.

receive data3 + data4 register

F data3(7)	E data3(6)	D data3(5)	C data3(4)	B data3(3)	A data3(2)	9 data3(1)	8 data3(0)
7 data4(7)	6 data4(6)	5 data4(5)	4 data4(4)	3 data4(3)	2 data4(2)	1 data4(1)	0 data4(0)

- Contains the received data for the bytes 3 and 4.

receive data5 + data6 register

F data5(7)	E data5(6)	D data5(5)	C data5(4)	B data5(3)	A data5(2)	9 data5(1)	8 data5(0)
7 data6(7)	6 data6(6)	5 data6(5)	4 data6(4)	3 data6(3)	2 data6(2)	1 data6(1)	0 data6(0)

- Contains the received data for the bytes 5 and 6.

receive data7 + data8 register

F	E	D	C	B	A	9	8
data7(7)	data7(6)	data7(5)	data7(4)	data7(3)	data7(2)	data7(1)	data7(0)
7	6	5	4	3	2	1	0
data87)	data8(6)	data8(5)	data8(4)	data8(3)	data8(2)	data8(1)	data8(0)

- Contains the received data for the bytes 7 and 8.

6. User Guide Documentation

The CANakari CAN-Controller is controlled by writing to its 16-bit width control- and data-registers. To access the different registers the controller is provided with the aforementioned Avalon interface and additional output ports on the user side . The ports also contain different interrupt and debug ports. An port overview is given below:

	Port	Breite	In/Out	Beschreibung
Avalon Interface	clock	1	IN	Taktsignal
	reset	1	IN	Reset (active low)
	address	5	IN	Adressierung der Register
	readdata	16	OUT	Daten-Bus für Read-Cycles
	writedata	16	IN	Daten-Bus für Write-Cycles
	cs	1	IN	Dient dazu, den Beginn eines Lese- oder Schreibvorgangs zu signalisieren.
	read_n	1	IN	Read request Signal
	write_n	1	IN	Write request Signal
CAN	rx	1	IN	Receive
	tx	1	OUT	Transmit
Interrupt Ports	irq	1	OUT	Interrupt
	irqstatus	1	OUT	Interrupt status
	irqsuctra	1	OUT	Interrupt succesful Trransmission
	irqsucrec	1	OUT	Interrupt succesful receive
Debug Ports	statedeb	8	OUT	State Debug
	Prescale_EN_debug	1	OUT	Prescale Enable Debug
	bitst	7	OUT	bitst Debug Signal

Figure 49: Pinout of the CANakari CAN Controller

6.1 Reset and Configuration Flow

Before the CANakari CAN-Controller can be used, a hard reset is required. After the reset happened an additional soft reset thorugh the Avalon Interface afterwards is recommended. Then the necessary registers for the CAN communication regarding the prescaler, bit-timing, interrupts and the acception mask can be configured. The full configuration flow which is currently used in the can conformance test and is suggested to be best practice, looks like this:

- Hard Reset:
  - Force a low active reset on the reset port of the CANakari.
  - Keep the reset signal for a time of about 4 Clk cycles (400ns for a 10Mhz clock).
  - After the reset, wait for a time of about 2 Clk cycle (100ns for a 10Mhz clock).
- Soft Reset:
  - reset bit in the general register needs to be set.
  - Afterwards wait for 4 clock cycles including the intermission time.

- Configure: Prescale Register:
  - Configure the prescaler by setting values for high and low.

- Configure: General Register:
  - Set the timing segments tseg1, tseg2 and the sjw (bit-timing).

- Configure: IRQ-Register (Interrupt):
  - Set the enable flags for the different interrupts.
  - Enable the transmit function of the physical layer (!),

otherwise the controller will not be able to send transmissions or ACK.

- Configure: Acception Mask Registers:
  - Set the acception mask to exclude specific identifiers, but it is

recommended to set both registers 0 if the feature is not needed.

- Wait until CANakari reaches Idle State:
  - Use the debug port *statedeb* to check the fsm state of the MAC.
  - Wait until the fsm state reaches *bus_idle_sample* (09_{hex} or 00001001_{binary})

6.2 Prescaler and Bit-Timing

To adapt the controller to different bit rates on the CAN bus, a Prescaler is implemented hereby the input clock of typically 10MHz gets divided down. To realize unsymmetrical divider ratios, the time where the output clock is is logically zero can be set independently from the time where the clock is logically one. The divider is implemented to respond only to positive clock edges. For this reason the smallest divider ratio is 1:2, the largest output frequency is consequently 5 MHz. The divider reads the values from the newly added prescale register, which is has received the last free address 0x1E. To ensure a clean switching, the counters in the the counters in the prescaler are reset as soon as this register has changed. The configuration of the prescaler is done through the prescale register, which includes the two 4-Bit prescaler configuration parts high and low. These parts must be seen separately, but together they form the resulting prescaler value. The following formula shows how the prescaler value is calculated:

$$f_q = f_{main} \cdot \frac{1}{(Prsc_{high} + 1) + (Prsc_{low} + 1)}$$

The high and low values refer to the high- and low-level parts of the system clock. So normally you choose equal values for high and low. For example, if you want to divide the system frequency by a prescale value of 10, you would set both parts to a value of 4. This is the value that has be written to the register. The new prescaled clock is only used for CAN communication, the rest of the internal modules of the CANakari still use the faster non prescaled clock. The prescaler value can only be set to values between 2 and 32.

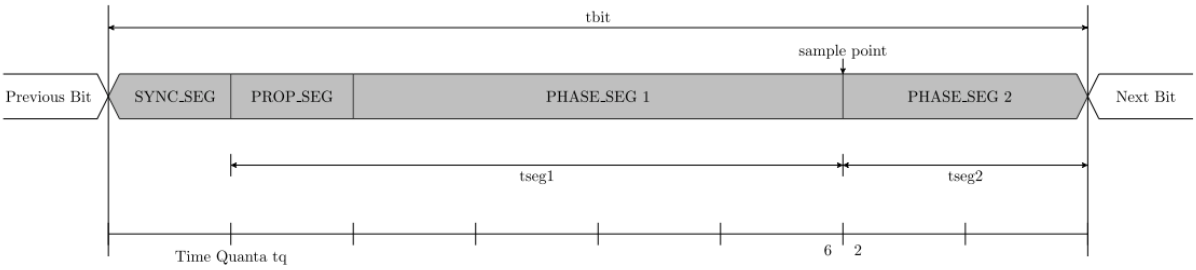


Figure 50: Bit-Timing Structure of the Canakari

When it comes to bit-timing, the CANakari CAN-Controller sticks to the same functionality as all the other CAN-Controllers (Figure 50). The bit-timing is configured inside the general register through the corresponding parts for the synchronization jump width (sjw), timing segment 1 (tseg1) and timing segment 2 (tseg2). Timing segment 1 contains the propagation segment and the phase segment 1. The different variables can take the following values:

- sjw [0 to 7]
- tseg1 [1 to 8] written as [0 to 7]
- tseg2 [1 to 8] written as [0 to 7]

But in case of the timing segments (tseg) the written values are always written as tseg-1 values. This is related to the 3-bit size of the fields in the register and the intention to save memory space. For example, the range of the 3-bit fields go from 000 to 111 in binary or 0 to 7 in decimal, but the range that is allowed for tseg values only goes from 1 to 8 in decimal and to store an 8 in binary you need 4 bit. So as the 0 is already excluded on purpose from the tseg range, the interpretation of the written values was simply shifted to exclude 0 and fit 8 inside the range of 3 bit. For example, if you write the following values to the register of the CANakari they will be interpreted as +1 values:

Register Value (written by user)	Real Value (CANakari interpretation)
000_{binary}	$1_{decimal}$
101_{binary}	$6_{decimal}$
111_{binray}	$8_{decimal}$

6.3 Send and Receive CAN Messages

Once the CANakari CAN-Controller is operational messages can be sent by using the transmit and receive registers. To start a transmission, the following registers in the presented order can be used to store the related data:

- Transmit Identifier Registers
  - Set the base- or extended identifier in the transmit identifier registers.
- Transmit Data Registers
  - Store the data bytes you want to send in the four transmit data registers,

  data will be used according to the DLC given in the message control register.
- Transmit Message Control Register
  - Set the enable flag *treq* to activate transmission.
  - Set the remote filed to 1 if a remote frame is sent.
  - Set the extended field to 1 if an extended identifier is used.
  - Set the DLC of the can message to specify length of data fields (bytes sent).

After the transmission is sent and delivered (ACK) the CANakari CAN-Controller will inform about the successful transmission through the interrupt port irqsustra.

Note: Interrupts that appear on the interrupt ports are directly related to the interrupt status bits in the IRQ-Register. Every time an input is received, an interrupt must reset the related status bits to 0. Otherwise the interrupt remains set. But caution to not accidentally reset the TX enable or IRQ Enable Bits.

In case of a received transmission the CANakari CAN-Controller signalises it through the interrupt port irqsucrec. If this happens you might want to read out the identifier and the data related to the received can message. To do so you can use the following registers in the presented order:

- Receive Control Register
  - Check the extended field to specify the type of identifier.
  - Check the remote field to specify if it is a remote frame.
  - Check the DLC field to specify the data length of the message.

- Receive Identifier Registers
  - If base identifier only load receive identifier register 1 (28-13),

  this register fully covers the 11-bit base identifier.
  - If extended identifier load both receive identifier registers.

- Receive Data Registers
  - If CAN message is a remote frame no data register needs to be read.
  - If not, read all data registers that contains data depending on DLC.

6.4 Interpretation of Identifier Registers

You may have already realized that an extended 29-bit identifier won't fit in a 16-bit register and therefore an extended identifier is represented by two registers. In the following figures can be seen how the base identifier and an extended identifier are stored inside the identifier registers. This representation suits both, the transmit and the receive registers.

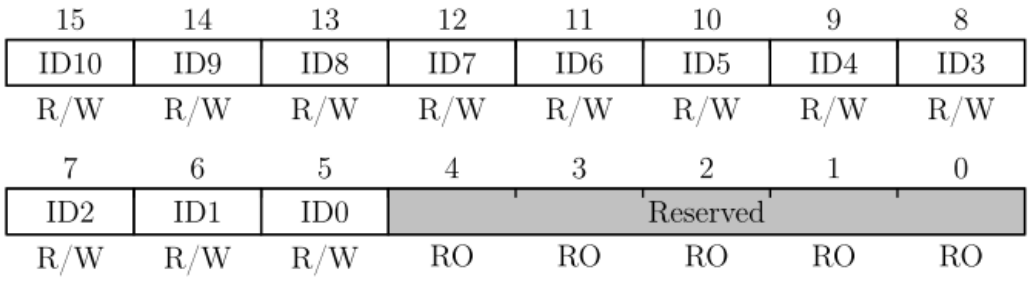


Figure 51: Storage of a base identifier in transmit or receive identifier register 1 (28 -13)

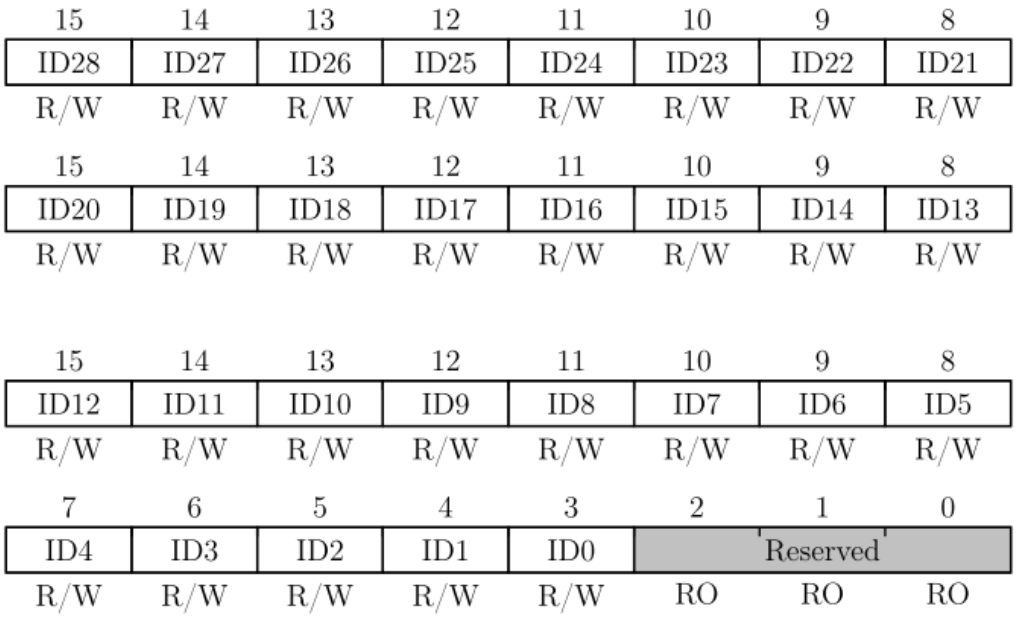


Figure 52: Storage of an extended identifier in both transmit or receive identifier registers

List of References

D.D Gajski, Specification and Design of Embedded Systems, Prentice Hall, 1994

D.D Gajski: SpecCharts: A VHDL Front-End for Embedded Systems, Technical Report,1993 [Gajski93]

D.D Gajski: Translating System Specification to VHDL, Technical Report, 1993

BOSCH: CAN Specification 2.0,1991

MOTROLA: Bosch Controller Area Network (CAN) Version 2.0 Protocol Standard, 1997 [motorolaCAN]

Wolfhard Lawrenz: CAN Controller Area Network, Hüthig, 1997

Allen Dewey: Analysis and design of digital Systems with VHDL

K.C. Chang: Digital Design and modeling with VHDL and synthesis, IEEE Computer Society

A. Mäder: VHDL Kurzbeschreibung, Universität Hamburg

Tobias Krawutschke: Test und Synthese eines CAN-Controllers, Fachhoschule Köln, 2001

Tim Hetzler: Implementation eines VHDL-CAN Controllers in ein Embbeded System, Fachhoschule Köln, 2004

Aaron Beer: Canakari Verification, Fachhoschule Dortmund, 2019/2020

Phillipp Ledüc: User Guide Canakari, Fachhoschule Dortmund, 2020

Michael A. Karagounis: Design eines CAN Controllers mit VHDL und SpecCharts, Fachhoschule Köln, 2000

Alexander Walsemann: Entwicklung einer Testbench in VHDL zur Verifikation von CAN-Controllern, Fachhoschule Dortmund, 2018