Logical Design of a CAN Controller

Eng. Boriana Ivanova Petrova, Ass. Prof., PhD Marin Hristov Hristov - Technical University of Sofia
Eng. Georgy Atanasov Gegov, Eng. Stanislav Nikolaev Markov – Innovative MicroSystems – Sofia,

1. Abstract.

CAN protocol comprises a complete set of rules for realising the transfer layer of a serial bus network. ROBERT BOSCH GmbH standardised it as a network protocol in 1985. At first intended to facilitate the "in-car" data communications, today CAN finds its application in a wide variety of network systems where low cost, high speed and high reliability are important.

2. Introduction.

The Controller Area Network (CAN) protocol offers a comprehensive solution to managing communication between multiple CPUs. This protocol uses the Data Link Layer and the Physical Layer in the ISO - OSI model. CAN is a multi-master bus with open, linear structure with one logic bus line and equal nodes. The bus nodes do not have a specific address. Instead, the address information is contained in the identifiers of the transmitted messages, indicating the message content and the priority of the message. CAN is a CSMA/CD-A, or Carrier Sense Multiple Access by Collision Detection using Arbitration protocol. Through a multi-master architecture, messages of length 8 bytes or less are sent on a serial bus. Error detection mechanisms, such as a 15-bit Cyclical Redundancy Check (CRC), provide a high level of data integrity. Erroneous messages are automatically retransmitted. Temporary errors are recovered. Permanent errors are followed by automatic switch-off of defective nodes. There is guaranteed system-wide data consistency. The CAN protocol uses Non-Return-to-Zero or NRZ bit coding. For synchronisation purposes, Bit Stuffing is used.

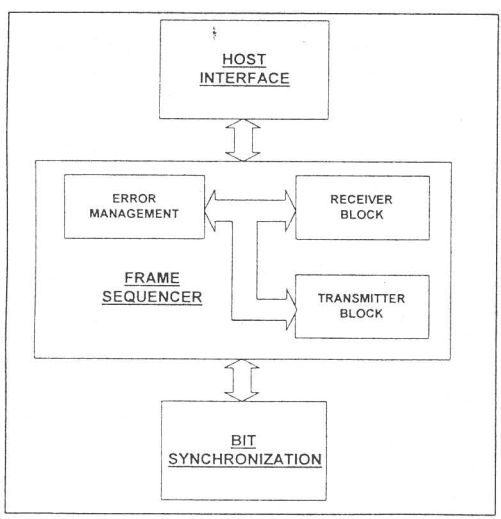
This article reveals one way of designing such CAN module, using the top-down methodology for creation of a synthesizable HDL description of the protocol functionality. A description like that might be used as a foundation of a silicon implementation of the synthesized circuit. Being independent of the final silicon technology, the design becomes an intellectual property (IP). It might be chosen by different chip manufacturers and included in ASICs with more complex functionality.

3. Block Diagram of the CAN controller.

According to the CAN specification two main blocks can be distinguished to realise the protocol. The first one takes care for the message send over the bus as a serial bit-stream. It is responsible for the message framing, bus arbitration, acknowledgement, error detection and signalling. The second block achieves the synchronisation of the different nodes over the bus between each other. This block doesn't care for the message itself or for the value of the logical line, but keeps track of the edges that appear between the successive bits with opposite values. The core realising the CAN protocol must be interfaced to another control unit (e.g. CPU) that implements the application layer of the CAN network. We propose a separate block for the host interface. This can be represented as a top-level VHDL providing easy customisation according to the peculiarities of the host system.

The corresponding diagram is shown at figure 1 and the three blocks are respectively Frame Sequencer, Bit Synchronization and Host Interface.

Figure 1. Block diagram.



We further decompose the Frame Sequencer into three sub-blocks. In this way we discriminate the receiver which permanently "listens" to the bus, from the transmitter which is activated either by a host request, or by detection of an error by the receiver. We find it convenient to separate the error management unit, because it monitors the

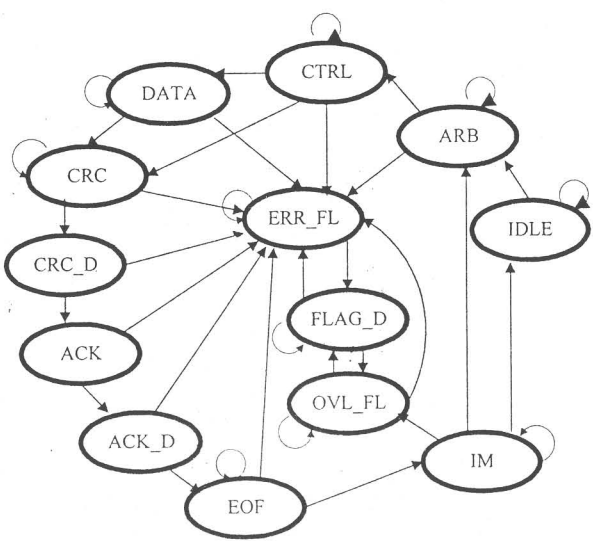
current states of both the receiver and the transmitter. This sub-block determines the error and bus status of the CAN controller.

4. Design of the Frame Sequencer.

Modeling the behavior of the Frame sequencer can be based on a finite state machine. The states correspond to the different frame fields as defined by the protocol. We use a separate counter to determine the number of the current bit inside a particular frame field. Transitions from one state to another depend on the current frame field, the value of the counter and the bus value read by the receiver. Figure 2 illustrates the state machine that we build for the receiver.

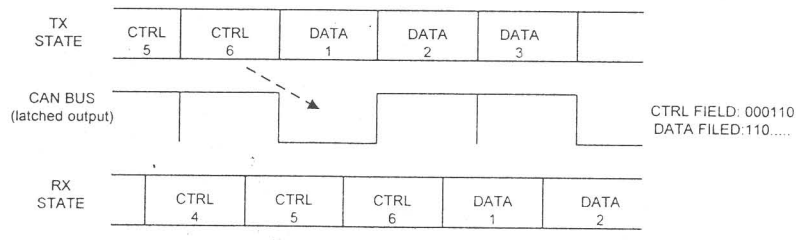
While the receiver is working permanently, the transmitter might be idle (e.g. reception of a message). Yet there are certain interactions between these two blocks during the whole message frame. Depending on the bus events (monitored by the receiver), the transmitter can become active and start transmission (e.g. Error Flag). We faced some problems when calculating the value of the next bit to be transmitted. We consider that only the state machine of the receiver and the bit-counter are not sufficient for that task. This is due to the following reasons. The moment at which the transmitter places the new bit on the bus is ahead of the moment at which the receiver samples the same bit. The time for calculation the next bit is called information processing time and is equal to one time quantum (as defined by the CAN specification).

Figure 2 State diagram of the receiver.



It starts with the sampling point of the current bit. The state transitions and the bit counter are synchronized with this sampling point. The transmitter issues the new bit value before the end of the current bit (next sample point) for the receiver. This means that the transmitter is actually one bit ahead of the receiver i.e. its counter equals the counter of the receiver plus one. We consider the case when the receiver is at the end of a frame field. Then the transmitter should be already at the next field at the same time. As it is shown at figure 3, there are certain moments at which the transmitter and the receiver are in different states. The counter of the transmitter is two bits ahead of that one for the receiver. This is because after extracting the next bit value from the transmit buffer, the transmitter latches it one clock later.

Figure 3. Transmission of the message - CTRL and the beginning of the DATA field.



So we introduce another state machine and a separate counter to describe the conditions of the transmitter. These are both synchronized with the transmission point of the current bit. The Bit Synchronization block determines the position of that point and the sample point. The value to be transmitted as the next bit is calculated according to the states of both the receiver's and the transmitter's states and counters and the current bus value. The state machine of the transmitter consists of almost the same states except that EOF combines the following frame fields: CRCD, ACK, ACKD, EOF, IM.

5. Bit synchronization.

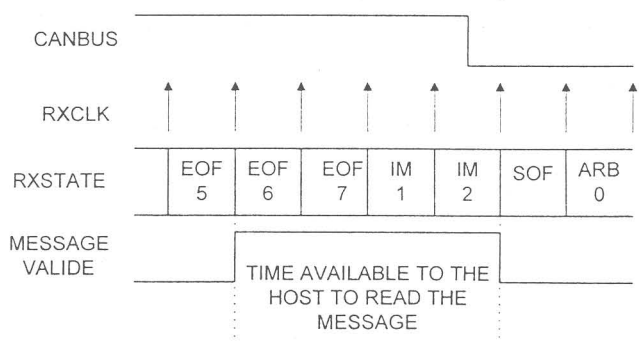
Although the CAN bus doesn't provide any clock signal, the CAN protocol handles the message transfer synchronously and all bus nodes derive the clock information from the incoming bit-stream. Once they are synchronized at the beginning of each message with the first falling edge of a frame, which belongs to the Start of Frame bit. To ensure correct sampling up to the last bit the CAN nodes need to re-synchronize throughout the entire frame. This is done on each recessive-to-dominant edge. According to the protocol, one CAN bit time is specified as four non-overlapping time segments. Each segment is constructed from an integer multiple of the Time Quantum. The Time

Quantum or TQ is the smallest discrete timing resolution used by a CAN node. A programmable divider of the CAN node's oscillator frequency generates its length. There is a minimum of 8 and a maximum of 25 Time Quanta per bit. The bit time, and therefore the bit rate, is selected by programming the width of the Time Quantum and the number of Time Quanta in the various segments. We find it is more practical to concatenate the first three time segments in one, hence to have two time segments only. A CAN node issues the new bit value at the beginning of the first segment and samples the bus at its end. In order to compensate for the oscillator frequency and phase differences between different nodes, the sample point must be moved inside the bit. This is done by changing the duration of the segments.

6. Relationship between the CAN module and the local host controller.

The last problem we examine concerns the message buffer for the receiver and the event of occurrence of an overload condition at the node. If only one message buffer is available, an overload frame should be generated automatically by the receiving node after the message validation. This is due to the following reasons. The received message is considered to be valid at the 6th bit of the End-of-Frame field. The host can get a sign for this event no earlier than the last bit of the End-of-Frame field. It must read the received message before the start of the transmission of another message over the CAN bus. But the time available for this task is equal to 2 bit lengths for the worst case as shown at the diagram below.

Figure 4. Reception of two consecutive messages with one message buffer.



In most cases this time is not enough to the commonly used microcontrollers to complete the message-read. If we don't want to loose the last message because of a new one, we must delay the transmission of the following telegram by sending an overload

frame. This should be done immediately after the EOF field as explicitly stated in the CAN protocol. As a result the bus capacity will become lower.

In order to avoid this problem, we utilised double receive buffer. Thus the host may read one message while the reception of another is in progress. The second receive buffer is not visible for to the host. The second message received is automatically transferred to the visible data buffer (if the host already releases it) upon its validation. The CAN controller sends overload frame if the second message becomes valid and the first one is not read yet.

7. Conclusion.

We included the propositions discussed above in our design of a reusable VHDL CAN core. We created a specific checklist in order to test and verify the correctness of the model. This checklist incorporates the necessary test scenarios for provoking the various events described in the CAN protocol specification. We built a VHDL test bench based on this checklist and verified the logical description. Next we synthesized the project using different ASIC libraries (ISCDIG, MIETEC and ALTERA) and performed successful simulations.

8. Reference:

"CAN Specification, Version 2.0", ROBERT BOSCH GmbH, 1991