FPGA Implementation of Bit Timing Logic of CAN Controller

PETER DZHELEKARSKI, M.SC, VOLKER ZERBE, DR., DIMITER ALEXIEV, ASSOC. PROFESSOR DR.
E-mail: pid@tu-sofia.bg
Technical University of Sofia, Bulgaria
Tel. +359 2 965 26 22

Abstract

Controller Area Network (CAN) protocol has two layers, Physical Layer and Data Link Layer (DLL). The upper sub-layer of Physical Layer, called Physical Signaling, and DLL are normally incorporated in CAN controllers.

This paper describes the implementation of Bit Timing Logic of CAN controller on Altera® Stratix™ FPGA board. The Bit Timing Logic corresponds to Physical Signaling sub-layer and is implemented as a schematic using Quartus® II Block Diagram Editor. The module is built up of 3 prescalers, PLL, synchronization- and receiving/ transmitting logic.

Keywords:
CAN, physical layer, physical signaling sub-layer, bit timing logic and CAN controller.

1. Introduction

Controller Area Network (CAN) is a serial communication bus originally developed for the automotive industry applications to replace the complex harness wiring by a two-wired bus. The specification allows signaling rates of up to 1 Mbps and features high immunity to electrical interference and ability to self-diagnose and repair data errors. These features have extended the range of applications to variety of industries including automotive, marine, medical, manufacture, military, aerospace, etc [1].

Distinctive characteristics of CAN interface [3]:

- multi-master and multicast capabilities;
- broadcasting of messages (called frames);
- very high error detection capability;
- sophisticated error handling and error confinement;
- simple and cost-effective implementation;
- world-wide accepted standard.

The main task of this project is to implement *Bit Timing Logic* (BTL) of CAN controller on Altera® FPGA Board, Stratix™ Edition. The module must conform to the CAN specification for all baud rates up to 1 Mbps.

2. CAN Overview

2.1 CAN Protocol

CAN is an ISO specified interface. The communication protocol *ISO 11898* describes how the information is transferred between different CAN nodes and conforms to the lowest two layers of the seven layer OSI/ISO model (Open Systems Interconnection). These layers are the *Data Link Layer* (DLL) and the *Physical Layer* (Fig. 1) [1], [5].

Although the application layer is not specified in ISO 11898, there are many industry specific CAN application protocols such as CANopen, SAE J 1939, DeviceNET, CAN Kingdom, etc.

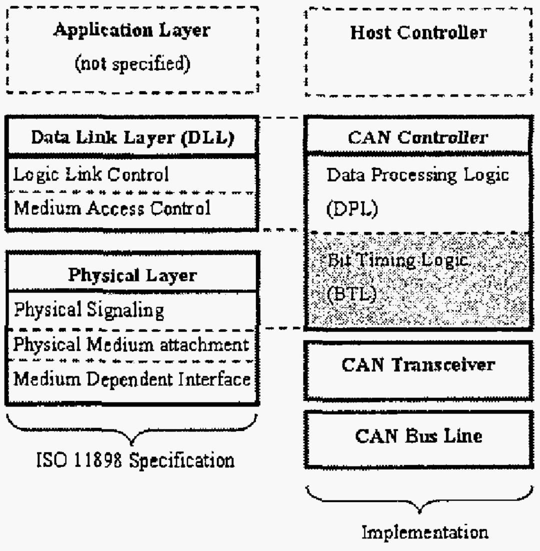

Fig. 1. Layered Structure of CAN Protocol.

The lowest two sub-layers of Physical Layer, named *Physical Medium Attachment* and *Medium Dependent Interface*, are normally implemented in *CAN transceivers* [8], [12], [13].

The upper sub-layer of Physical Layer, *Physical Signaling* and the DLL are implemented in *CAN* *controllers*. Each CAN controller can be divided into two parts: BTL and *Data Processing Logic* (DPL).

BTL represents the Physical Signaling sub-layer and is the subject of current project [2].

DPL represents the DLL and is a subject of another project.

2.2. *CAN Physical Signaling*

This sub-layer is responsible for the bit coding/decoding, bit timing and bus synchronization [3], [5].

The bit stream in a CAN message is coded according to NRZ (Non-Return to Zero) method. This means that during the total bit time the bit level is constant and is either *dominant* or *recessive*.

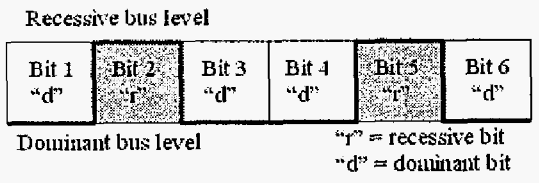

Fig. 2. CAN NRZ Bit Stream

The dominant state has a higher priority towards the recessive state. This means if two nodes attempt to set different states on the bus, the resulting state will be dominant. In most cases but not always recessive state is represented by logical "1" and dominant state represented by logical "0" on the bus.

The NRZ-coded signal provides no edges for resynchronization when transmitting a large number of bits of the same polarity. Therefore *bit-stuffing* is used to ensure the synchronization of all nodes.

The bit stuffing is specified in Data Link Layer and is not implemented in BTL.

The rule of CAN bit stuffing is to insert a bit of the opposite polarity after transmitting 5 bits of the same polarity. When receiving the bit stream *destuffing* is performed. The receiver checks the number of the bits of the same polarity and removes stuffed bits.

The CAN bit time is constructed of four non-overlapping time segments [2]. Each time segment consists of integer number of *time quanta* (TQ). The time quanta are generated by a programmable divide of an oscillator. The time quantum is the smallest discrete timing resolution used by a CAN node. Each bit must consist of 8 to 25 time quanta. Fig. 3 represents the structure of the bit time.

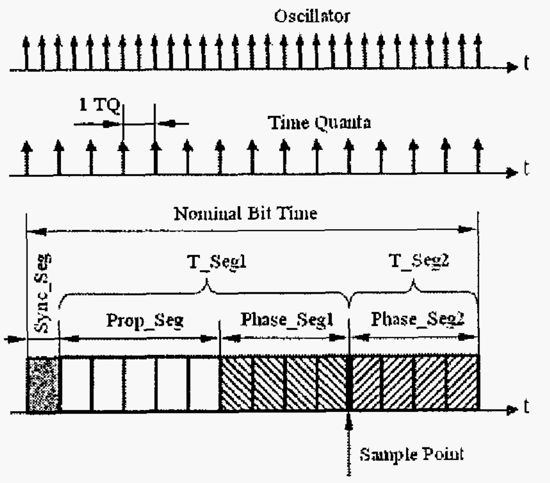

Fig. 3. CAN Bit Time

Description of CAN bit time segments:

- Synchronization Segment (*Sync_Seg*) is used to synchronize the various bus nodes. A bit state change is expected to occur within this segment by receiving nodes. The length of this segment is fixed to 1 TQ.
- Propagation Time Segment (*Prop_Seg*) is used to compensate for signal propagation delays across the network. The length of the segment is 1...8 TQ.
- Phase Buffer Segment 1 (*Phase_Seg1*) is used to compensate for edge phase errors. The segment may be lengthened during resynchronization. Its length is 1...8 TQ.
- Phase Buffer Segment 2 (*Phase_Seg2*) is used to compensate for edge phase errors. The segment may be shortened during resynchronization. Its length is 1...8 TQ and may not be greater than *Phase_Seg1*.

The length of the bit segments is programmable. For the ease of programming often the *Prop_Seg* and the *Phase_Seg1* are merged into one segment called *Timing Segment 1* (*T_Seg1*) and in this case the *Phase_Seg2* is renamed to *Timing Segment 2* (*T_Seg2*), that is shown in Fig. 3.

The *Sample Point* is the point of time at which the bus level is read and interpreted as the value of the respective bit. It is located between both phase buffer segments. During resynchronization the sample point position is moved with integer number of TQ.

The maximal allowed amount of time quanta by which the sample point position can be moved is specified and is called *Synchronization Jump Width* (SJW). The SJW is programmable in the range of 1...4 TQ.

After receiving the falling edge of the first SOF bit (start of frame) every CAN node must perform *hard synchronization*. This ensures that all nodes are synchronized at the beginning of the message (Fig. 4).

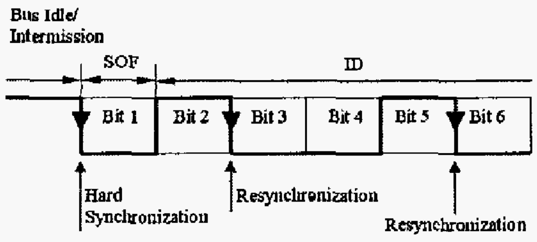

Fig. 4. CAN Bus Synchronization

In order to ensure correct sampling up to the last bit, the CAN nodes need to resynchronize throughout the entire frame. This is called *resynchronization* (soft synchronization) and is performed on each recessive to dominant edge (Fig. 4).

The hard synchronization is performed by restarting the internal bit-time generator in the CAN controller so that it becomes in phase with the received falling edge.

The resynchronization is performed by lengthening the *Phase_Seg1* of the current bit or by shortening the *Phase_Seg2* of the next bit with fixed amount of TQ.

During the bit time only one synchronization is allowed.

The resynchronization is done according to the value of phase error (e) of the received edge. The phase error values are integer (Fig. 5).

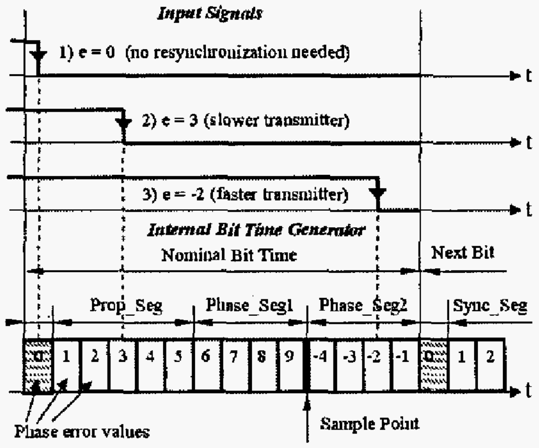

Fig. 5. Phase error of received edge

Resynchronization rules

• $e = 0$ if the received edge lies within the synchronization segment. No resynchronization is performed in this case.

• $e > 0$ if the received edge lies before the sampling point. The *Phase_Seg1* of the current bit is lengthened with the amount of e, if $e <$ SJW. Otherwise it is lengthened with the amount of SJW.

• $e < 0$ if the received edge lies after the sampling point. The *Phase_Seg2* of the next bit is shortened with the amount of $|e|$, if $|e| <$ SJW. Otherwise it is shortened with the amount of SJW.

3. FPGA Board Overview

The Stratix™ Board, used for the present project, is equipped with EP1S10F780C6 FPGA device [9]. It is a hardware platform for developing embedded systems with 10,570 logic elements and 920 kbits on-chip memory. The FPGA is look-up table based device with two dimensional row and column architecture. There are dedicated PLLs and clock networks. It allows programming of Nios® 32- or 16-bit RISC processor design [14], [15].

4. Implementation

4.1. Methods of Implementation

The following three methods of possible implementations of CAN controller have been found:

1) Completely hardware implementation using HDL. It produces a completely functional CAN controller with an efficient use of FPGA resources. This method is very complex due to the sophisticated CAN protocol. There are several intellectual property (IP) implementations created especially for Stratix FPGA using this method. An example is the CAN Module from IFI [6].

2) Completely software implementation. This is a software implementation in C/C++ language for the Nios 32-bit Reference Design which exists on the Stratix board. The Nios CPU has to handle both DLL and Physical Signaling sub-layer. Although the CPU is a RISC with 50 MHz clock frequency it can not perform all the necessary tasks required by the protocol for highest baud rates. Therefore this method is useful only for the slowest baud rates. It is a good illustrative example which can be used for educational purposes. Its advantage is that there is no need of hardware implementation.

3) Combined Hardware/Software Implementation. This method combines the advantages of the previous two methods. It is an convenient way of implementation of completely functional CAN controller. Its structural diagram is shown in Fig. 6. It consists of two parts: BTL – hardware block, created as a schematic, implemented comparatively easy because of its uncomplicated functions; DPL – microprocessor system containing Nios CPU with necessary peripherals, created by the SOPC Builder. The sophisticated requirements of DLL can be easily implemented in a programming language like C/C++.

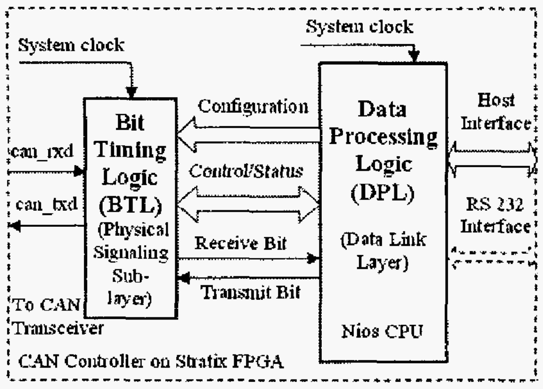

Fig. 6. Structural Diagram of CAN Controller

4.2. Bit Timing Logic Implementation

BTL is responsible for the bus synchronization and processing of bit time. This block changes its state on every time quantum. It defines the number of TQ per bit, the position of sample point and baud rate [2]. BTL is implemented as a schematic using Quartus® Block Diagram Editor [7]. Fig. 7 shows its structural diagram. This way of implementation has been chosen because of the uncomplicated functions of the block. The block diagram/schematic gives also a good visual representation of the logical functions.

BTL is directly connected to CAN transceiver with *can_rxd* and *can_txd* signals. It reads CAN bus state and sends bits to DPL or writes bits which are to be transmitted to the CAN bus (using *bit_rx* and *bit_tx* signals). BTL is configured by DPL in order to work with different baud rates and to construct CAN bit time segments (shown in Fig. 3).

DPL is connected to the host controller via host interface. The RS 232 interface can be used for testing and development purposes.

The full compilation of BTL module produces an entity, containing 139 total logical elements (1%) of available 10,570 logical elements on the Stratix EPS10F780C6 device [9], [10], [15].

In the following paragraphs each sub-module will be described in detail.

4.2.1. Baud-Rate Prescaler with PLL

This sub-module creates TQ clock (*clk_tq*) from the 50 MHz system clock (*clk*) on the Stratix board. The TQ frequency is configured by DPL with the *config_prescaler* signal. The *clk* is fed to a PLL which outputs high frequency signal *clk_pll* (160 Mhz). This signal is used by a configurable prescaler which produces the TQ. The frequency of TQ is according to the CiA DS-102 CAN baud-rate recommendations (Table 1, [4]). The *clk_pll* signal is used also for creating short pulses of asynchronous input signal *can_rxd*.

The output of the baud-rate prescaler is fed to the inputs of bit-time- and sample-point prescalers.

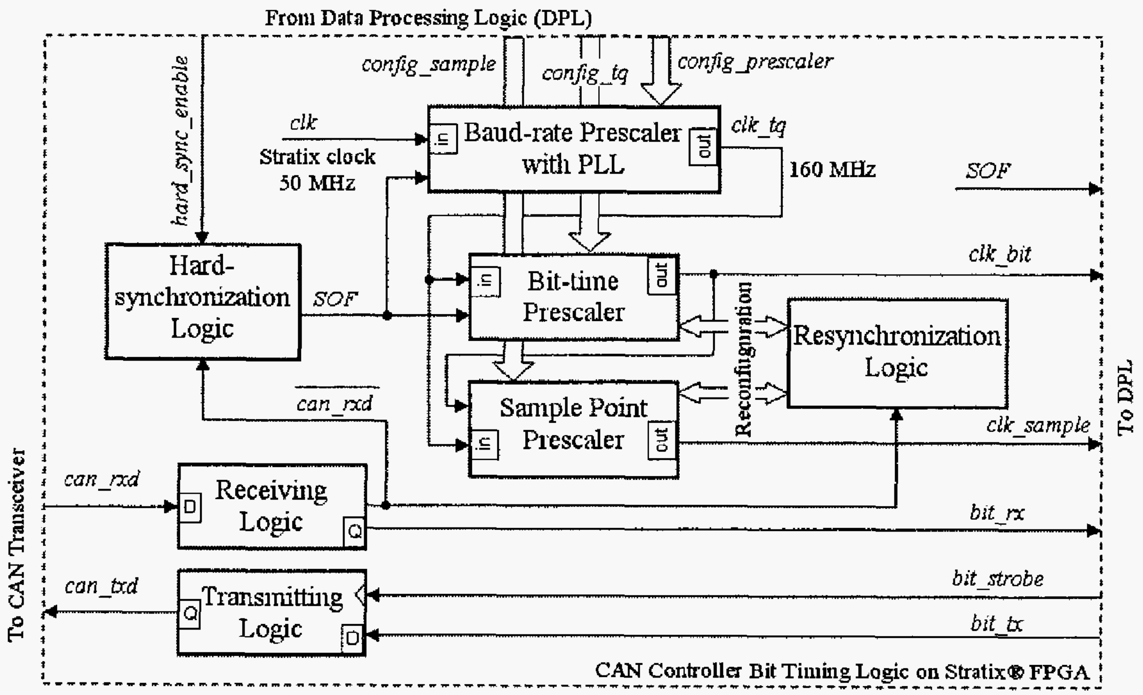

Fig. 7. Structural Diagram of Bit Timing Logic

TABLE 1. CiA DS-102 Baud-rate Recommendations

Bit Rate, kbps	Nominal bit time	TQ per bit	Sample point	TQ length	ck_tq, MHz
1000	1 μs	8	6 TQ	125 ns	8
800	1.25 μs	10	7 TQ	125 ns	8
500	2 μs	16	13 TQ	125 ns	8
250	4 μs	16	13 TQ	250 ns	4
125	8 μs	16	13 TQ	500 ns	2
62.5	16 μs	16	13 TQ	1 μs	1
50	20 μs	16	13 TQ	1.25 μs	0.8
20	50 μs	16	13 TQ	3.125 μs	0.32
10	100 μs	16	13 TQ	6.25 μs	0.16

The baud-rate prescaler is able to be restarted after hard synchronization by the *SOF* signal (start of frame) which is provided by the hard-synchronization logic (see 4.2.6) and is applied to the asynchronous load input of the counter.

4.2.2. Bit Time Prescaler

The task of the bit-time prescaler is to construct the bit time from configurable number of TQ (configured by DPL with *config_tq* signal). It must also be able to synchronize after hard-synchronization or resynchronization.

The core of this prescaler is a down counter which is clocked by TQ. The output signal *clk_bit* marks the beginning of bit time. The counter loads its configuration data on every cycle.

The hard synchronization is performed by the *SOF* signal which is applied to the asynchronous clear input of the counter. The resynchronization is performed with 2 bus multiplexers (for positive- and negative resynchronization respectively) which change the configuration data for the counter. The multiplexers are controlled by the resynchronization logic.

4.2.3. Sample Point Prescaler

The sample point prescaler marks the position of the sample point. It must be able to synchronize only after positive resynchronization ($e > 0$). The core of this prescaler is also a down counter, clocked by TQ and restarted from *clk_bit* at the beginning of each bit time.

This prescaler is configured by DPL with the *config_sample* signal. The positive resynchronization is performed with one bus multiplexer. The multiplexer is controlled by the resynchronization logic.

4.2.4. Receiving Logic

It contains one D flip-flop which reads directly the state of CAN bus (*can_rxd*) at moments, clocked by the *clk_sample* and transfers data directly to DPL with *bit_rx* signal. DPL is informed with *clk_sample* that a new bit has been received.

4.2.5. Transmitting Logic

It is built of two D flip-flops connected in series. The first flip-flip is clocked by the *bit_strobe* signal from DPL. The bit which is to be transmitted is sampled in the first flip-flop (*bit_tx* signal) and applied to the information input of the second flip-flop. The storing of bit must be performed during *Phase_Seg2* (between the sample point and the beginning of the next bit). At the beginning of next bit time the bit is sampled in the second flip-flop which outputs it to the CAN bus via *can_txd*. When DPL wants to stop the transmission it must sample lastly recessive bit ("1").

4.2.6. Hard-synchronization logic

Hard synchronization is enabled by DPL when a new frame is expected using the dedicated signal *hard_sync_enable*. After completing the hard synchronization it is automatically disabled by this logic.

When a falling edge on *can_rxd* is received (and hard synchronization is enabled) it is interpreted as the beginning of frame and hard synchronization is performed. A short pulse *SOF* is created at this moment and passed to other logic elements and also to DPL to inform it that the reception of a new message has just started. The three prescalers (baud-rate-, bit-time- and indirectly sample point prescaler) are restarted and the BTL becomes in phase with the receiving signal on the bus.

The moment of time in which the falling edge of *can_rxd* occurs is asynchronous with respect to the baud-rate prescaler and PLL. The utilized high frequency clock *clk_pll* (160 MHz, 6.25ns) decreases the phase error to approx. 15 ns (the latency time after receiving falling edge on *can_rxd* and restarting the counters – simulation value).

4.2.7. Resynchronization logic

The task of resynchronization logic is to modify (reconfigure) the prescaler configuration for the current or the next bit depending on the sign of phase error *e*. It is obvious from fig. 5 that two counters are necessary in order to determine the phase error (positive- and negative error counter).

A signal which automatically enables/disables the resynchronization is provided (*allowed_resync*). It is set and cleared observing special rules. During hard synchronization and after resynchronization the signal is cleared. At the earliest the *allowed_resync* must be set high after the sampling point of the next bit (or the bit after the next bit during negative resynchronization). As well, the *allowed_resync* must be cleared during the *Sync_Seg*. Observing of these rules makes the module insensible to interference spikes in receiving signal which occur when the resynchronization is disabled.

In order to mark the beginning of the resynchronization, a short pulse *SOR* (start of resynchronization) is produced (fig. 9, 10).

Positive Phase Error Resynchronization ($e > 0$)

The number of TQ in the current bit and the sample point position must be increased with one and the same amount of TQ (represented by *pos_SJW* signal). If $e \geq$ SJW then *pos_SJW* value is equal to SJW, otherwise it is equal to *e*. This logical statement is implemented using a comparator which controls one bus multiplexer.

The values which are stored in bit-time- and sample-point counters must be modified during the current bit before the sample point. Each new counter's value is produced using addition of the value present in the counter and the *pos_SJW* value. Two adders are utilized in implementing this operation. They output two signals containing the new modifying values for the counters.

At first, the new value is applied to the data inputs of each counter by switching the two bus multiplexers (in the two prescalers). After that, using the *SOR* signal, the new values are loaded asynchronously into the counters. At the end, according to the resynchronization rules, the *allowed_resync* signal is cleared and the multiplexers are switched back to their original state.

Negative Phase Error Resynchronization ($e < 0$)

The number of TQ in the next bit must be decreased with amount of TQ, represented by the *neg_SJW* signal. The *neg_SJW* value is determined by SJW and *e* in the same manner.

The determination of *neg_SJW* is performed using a comparator and one bus multiplexer. The configuration value for the bit-time counter must be modified during the current bit so that it will be loaded into the counter at the beginning of the next bit. The new counter's value is produced using subtraction of the *neg_SJW* value from the present value. One subtracter is utilized here. The result is applied to one of the bus multiplexers in the bit-time prescaler. This multiplexer is switched and hold in its new state until the sampling point of the next bit. This ensures that the synchronous load of the modified value will be done correctly as well as the multiplexer will return to its initial state.

5. Simulation Waveforms

All waveforms are from Quartus® II Waveform Editor and are result of timing simulation.

5.1. Waveforms from the Prescalers

Fig. 8 shows the signals: *clk_main* (a signal delivered from *clk_pll* with freq. 160 MHz); *clk_tq* with frequency 8 MHz and *clk_sample*. The baud rate is 1 Mbps with 8 TQ per bit and sample point position of 6 TQ.

5.2. Waveforms during Hard Synchronization

Fig. 9 shows the performing of hard synchronization. After receiving a falling edge on *can_rxd* the short pulse *SOF* is produced which restarts the counters. After that, further hard synchronization is disabled by the *allowed_hsync*. The baud rate is 1 Mbps with 8 TQ per bit.

5.3. Waveforms during Resynchronization

Fig. 10 shows the performing of positive resynchronization. The baud rate is 250 kbps with 16 TQ per bit with sample point position 11 TQ (SJW = 4). The *neg_num_tq* signal is the parallel output from bit-time prescaler which represents the negative error counter (*pos_num_tq* - positive). The *sample_counter* signal is the parallel output from the sample point prescaler. The *error_sign* signal is "1" when e < 0.

Fig. 11 shows the performing of negative resynchronization with the same configuration.

Conclusion

This paper describes the implementation of Bit Timing Logic of CAN controller on Altera® Stratix™ FPGA board. The operation of module is simulated and the results are in accordance with CAN specifications for all baud rates.

After completing the Data Processing Logic, which is a subject of further work, a fully-functional CAN controller can be created on the FPGA board.

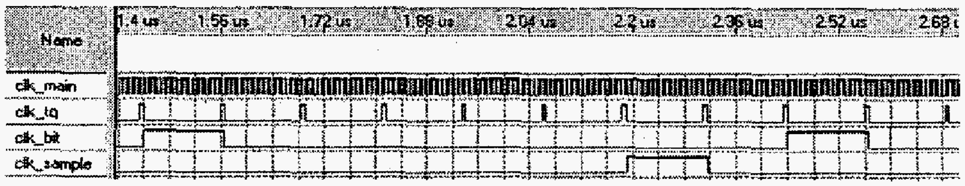

Fig. 8. Simulation Waveforms from the Prescalers

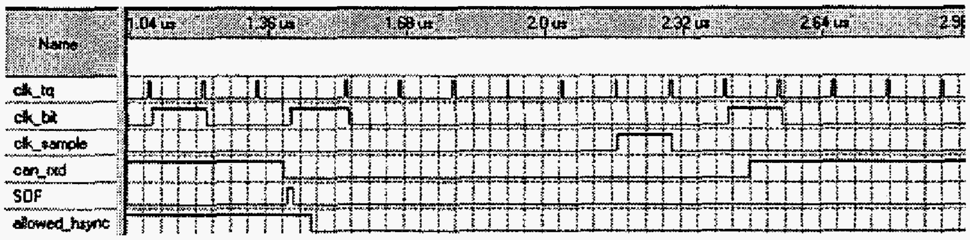

Fig. 9. Simulation Waveforms showing Hard Synchronization

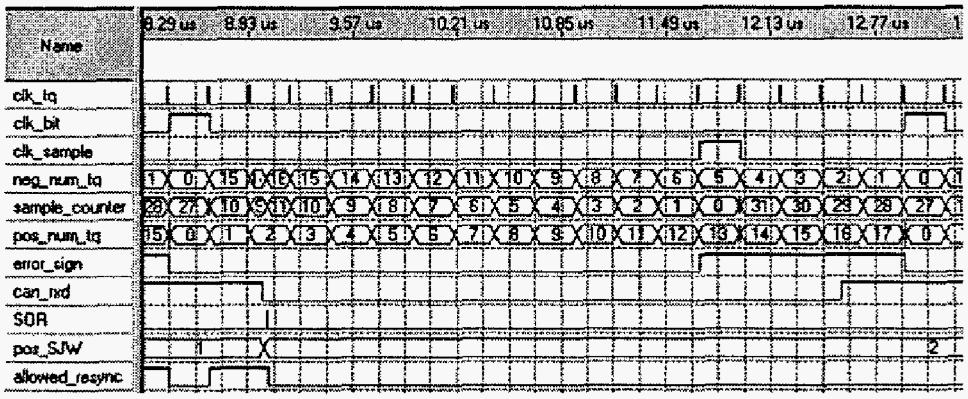

Fig. 10. Simulation Waveforms showing Positive Resynchronization

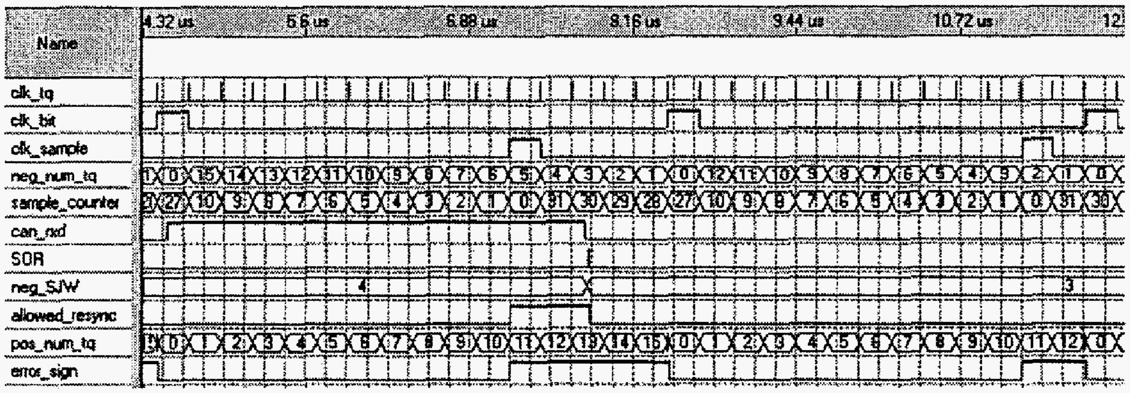

Fig. 11. Simulation Waveforms showing Negative Resynchronization.

References

Technical Articles and Books:

1. S. Corrigan. Introduction to the controller area network (CAN). Texas Instruments, Application Report SLOA101, 2002.
2. F. Hartwich and A. Bassemir. The configuration of the CAN bit timing. Robert Bosch GmbH, K8/EIS, 1997.
3. K. Etschberger. Controller-Area-Network. Grundlagen, Protokolle, Bausteine, Anwendungen. Carl Hansa Verlag, 2002.

Specifications and Datasheets:

4. CAN physical layer for industrial applications. CiA, 1994.
5. CAN specification version 2.0 Part A and B. Robert Bosch GmbH, Stuttgart, 1991.
6. IFI Avalon CAN module User Guide. I/F/I, 2003.
7. Introduction to Quartus® II. Altera Corporation, 2003.
8. MCP 2515. Stand-alone CAN controller with SPI™ interface. Microchip, DS21801B, 2003.
9. Nios development board reference manual, Stratix Edition. Altera Corporation, 2003.
10. Nios hardware development tutorial. Altera Corporation, 2003.
11. Nios PIO data sheet. Altera Corporation, 2003.
12. SJA 1000. Stand-alone CAN controller. Philips, Product Specification, 2000.
13. SN65HVD251 CAN transceiver. Texas Instruments, SLL545B, 2002.
14. SOPC Builder data sheet. Altera Corporation, 2003.
15. Stratix device handbook, Volumes 1, 2 and 3. Altera Corporation, 2003.