CLAN – A technology-independent synthesizable CAN controller

Arnaldo S. R. Oliveira, Nelson L. Arqueiro, Pedro N. Fonseca

University of Aveiro / IEETA – Portugal

The CLAN intellectual property core is a CAN 2.0B controller developed at the Electronics and Telecommunications Department of the University of Aveiro, for research and educational purposes and in particular with the aim of providing the adequate hardware support to implement and validate higher layer protocols such as TTCAN or FTT-CAN. It was modeled at RTL level using the VHDL hardware description language, synthesized, implemented and tested on Xilinx FPGAs. However, the model is technology independent and can be synthesized for different implementation technologies from FPGAs to ASICs. The CLAN IP core fully implements the CAN 2.0B specification and it includes also a synchronous parallel microprocessor interface, interrupt generation logic and some advanced features, such as message filtering, single shot transmission and extended error logs and statistics. The data bus width can be 8, 16 or 32 bits wide. For applications where microprocessor interface is not needed or a different interface is required, the core internal module that implements the protocol can be used separately. The CLAN controller with microprocessor interface logic occupies about 30% of a Xilinx Spartan-IIE XC2S300E FPGA, corresponding to 100,000 equivalent logic gates, approximately. It was tested with other commercial controllers within a bus operating at 1Mbit/seg.

Introduction

Defined in the late 80's, CAN (Controller Area Network) [1] found wide-spread acceptance in embedded distributed control systems, from automotive to industrial applications. In spite of its popularity, the application of CAN in safety-critical systems is, nevertheless, impaired by the event-triggered characteristics of the original definition. In CAN, a node can send a message at any time, provided there is silence on the bus (CSMA); the MAC mechanisms will handle the resulting collisions. As a consequence, a node sending a message has no guarantee in what concerns the delivery time of that message; depending on the message priority, it may loose contention for several consecutive times, thus postponing the effective sending of the message.

For critical applications, time-triggered systems are preferred, due to their scalability, composability and dependability properties [2]. The last few years saw the outcome of some proposals to improve the time characteristics of CAN (e.g., TTCAN [3], FTT-CAN [4]). These take advantage of the fact that Bosch's and ISO specifications define only layers 2 and (partially) 1 of the ISO OSI model. With major or minor changes on the original definition, these new proposals impose some determinism in the message exchange behavior, namely by allowing a node to send its message at well defined instants in time. This is achieved by properly defining mechanisms in the layers above the original definition.

TTCAN (Time-Triggered Communication on CAN) started in ICC'98, the International CAN Conference, where an expert group, including CiA (CAN in Automation), chip providers, users and academia, joined the ISO TC22/SC3/WG1/TF6. The result was ISO 11898-4, part 4 of the ISO 11898 standard, that specifies time triggered communication on CAN [5].

FTT-CAN has been proposed at the University of Aveiro as a mean to merge flexibility and timeliness in CAN systems. The aim is to achieve a communication paradigm that allows systems to be both timely, delivering the messages under the specified time constraints, and flexible, by not requiring the message set to be statically defined during system operation.

Motivation and Objectives

Both proposals for time triggered operation of CAN are built on top of the existing protocol with little or no modifications (the aim of FTT-CAN is also to provide timely behavior with standard CAN controllers). The development of a new communication protocol requires its validation.

Although simulation and formal validation play an important role here, they are not, on their own, sufficient. The last step in validation is always field tests, and these have to be performed with hardware devices. These tests should also involve the verification that the adopted solution is better than the alternatives. Ideally, we should have a flexible communication controller that can be programmed to follow some specification and that can be modified. Another issue to test is the robustness of the new protocols, mainly in what concerns fault tolerance. To do this, faults have to be introduced in the system in a controlled and predictable way. Again, we meet the need for a controller that we can modify to our desire. In fact, these requirements cannot be easily fulfilled with the CAN products commercially available [6], thus a CAN controller was developed based on the *CANSim* simulator [7]. The initial requirements were the following:

- Complete CAN 2.0B implementation;
- Internal status fully visible to effectively support the implementation of higher layer protocols;
- Enhanced logging capabilities (individual error counters) and extended statistics (message counters);
- Message filtering capabilities;
- Flexible interface - parallel/serial, (a)synchronous, (de)multiplexed buses.

Architecture

The developed CAN controller fully implements the CAN 2.0B specification. The developed IP block was split in two modules, to separate the logic that implements the protocol from the interface:

- The *CLAN Core* module which implements the CAN 2.0B protocol;
- The *CLAN Controller* module which provides a synchronous parallel interface with demultiplexed buses.

CLAN Core Module

The *CLAN Core* module contains all the circuits required to implement the MAC and the LLC layers of the CAN 2.0B specification. It can be used separately directly connecting to sensors/actuators in a CAN node without a microprocessor or it CAN be used as a building block to create a controller with a customized interface.

Interface Ports

The external interface of the *CLAN Core* module is shown on Figure 1. The ports are divided into the following functional groups: *Synchronization and Initialization* (Table 1), *Timing Configuration* (Table 2), *Mode Setup* (Table 3), *General Status and Statistics* (Table 4), *Transmission and Reception Data* (Table 5), *Transmission and Reception Configuration* (Table 6), *Error and Fault Confinement* (Table 7), *Message Filtering Setup* (Table 8) and *Bus Interface* (Table 9). A short description of each port is given into the tables below.

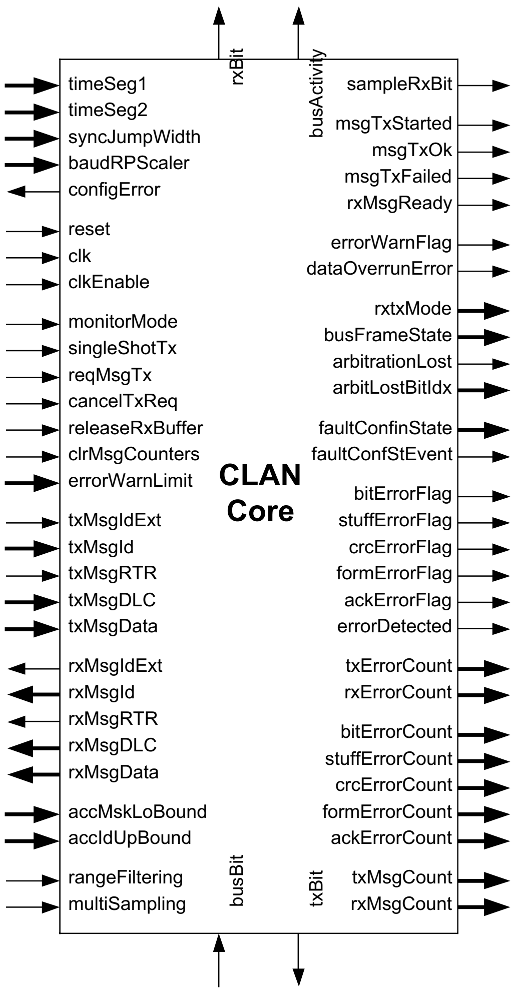


Figure 1 - *CLAN Core* module interface.

The *sampleRxBit* port (Table 1) provides a clock synchronous with the *Sample Point*. It is useful for TTCAN implementation.

Name	Type	Description
reset	*In*	Asynchronous reset input
clk	*In*	Main synchronization signal
clkEnable	*In*	Enable input for the "clk" signal
sampleRxBit	*Out*	Sample point synchronous clock

Table 1 - Synchronization and Initialization ports.

Name	Type	Description
timeSeg1	*In*	"Length – 1" of Time Segment 1 measured in time quanta
timeSeg2	*In*	"Length – 1" of Time Segment 2 measured in time quanta
syncJumpWidth	*In*	Synchronization Jump Width value
baudRPScaler	*In*	Baud Rate Pre-Scaler value

Table 2 - Timing Configuration ports.

Name	Type	Description
singleShotTx	*In*	When active disables automatic message retransmission in case of error
multiSample	*In*	Sampling mode for improved noise imunity
monitorMode	*In*	When active sets the output driver permanently to "recessive" level

Table 3 - Mode Setup ports.

Name	Type	Description
rxtxMode	*Out*	Current RX/TX mode (None, Rx, Tx, Arbitration)
busFrameState	*Out*	Current state of the frame present on the bus
arbitrationLost	*Out*	Activated during one CAN bit time in case of arbitration lost
arbitLostBitIdx	*Out*	When "arbitrationLost" = 1 this output indicates the bit where arbitration was lost
txMsgCount	*Out*	Number of successfully transmitted messages
rxMsgCount	*Out*	Number of successfully received messages
clrMsgCounters	*In*	When activated clears the "txMsgCount" and "rxMsgCount" counters
busActivityFlag	*Out*	Indicates the presence of bus activity

Table 4 - General Status and Statistics ports.

Name	Type	Description
txMsgIdExt	*In*	Tx message extended identifier flag
txMsgId	*In*	Tx message identifier
txMsgRTR	*In*	Tx message RTR flag
txMsgDLC	*In*	Tx message DLC value
txMsgData	*In*	Tx message Data bytes
rxMsgIdExt	*Out*	Rx message extended identifier flag
rxMsgId	*Out*	Rx message identifier
rxMsgRTR	*Out*	Rx message RTR flag
rxMsgDLC	*Out*	Rx message DLC value
rxMsgData	*Out*	Rx message Data bytes

Table 5 - Transmission and Reception Data ports.

Name	Type	Description
reqMsgTx	*In*	When activated, requests the transmission of the message applied to the txMsg(IdExt / Id / RTR / DLC / Data) ports
cancelTxReq	*In*	When activated, cancels the previous transmission request, if still pending
msgTxStarted	*Out*	Activated during one CAN bit time at the start of a message transmission
msgTxOk	*Out*	Activated during one CAN bit time at the end of a successful message transmission
msgTxFailed	*Out*	Activated when a message transmission fails
rxMsgReady	*Out*	Activated during one CAN bit time at the end of a successful message reception
releaseRxBuffer	*In*	When activated, releases the Rx buffer, allowing the core to write a new received message on the buffer accessible trough the rxMsg(IdExt / Id / RTR / DLC / Data) ports
dataOverrunError	*Out*	Activated when a new message was received before an external release of the Rx buffer containing the previous received message. The newly message received is discarded

Table 6 - Transmission and Reception Configuration ports.

Name	Type	Description
configError	*Out*	Activated when the timing parameters are invalid
faultConfinState	*Out*	Current fault confinement state ("Error Active", "Error Passive", "Bus Off")
faultConfStEvent	*Out*	Activated during one CAN bit time after a change on the fault confinement state
bitErrorFlag	*Out*	Active during one CAN bit time in case of a bit error
stuffErrorFlag	*Out*	Active during one CAN bit time in case of a stuff error
crcErrorFlag	*Out*	Active during one CAN bit time in case of a CRC error
formErrorFlag	*Out*	Active during one CAN bit time in case of a form error
ackErrorFlag	*Out*	Active during one CAN bit time in case of a acknowledge error
errorDetected	*Out*	Active during one CAN bit time in case of a bit, stuff, CRC, form or acknowledge error
txErrorCount	*Out*	Tx Error Count as defined on the CAN specification
rxErrorCount	*Out*	Rx Error Count as defined on the CAN specification
errorWarnLimit	*In*	Threshold value used to flag a disturbed bus
errorWarnFlag	*Out*	Activated when one of the error counters is greater than the "errorWarnLimit" value
bitErrorCount	*Out*	Number of bit errors occurred
stuffErrorCount	*Out*	Number of stuff errors occurred
crcErrorCount	*Out*	Number of CRC errors occurred
formErrorCount	*Out*	Number of form errors occurred
ackErrorCount	*Out*	Number of acknowledge errors occurred

Table 7 - Error and Fault Confinement ports.

Name	Type	Description
rangeFiltering	*In*	When activated, filtering is performed based on the lower and upper bound of message identifiers specified by the next two ports; when deactivated, filtering is based on identifier patterns
accMskLoBound	*In*	Specifies the identifier lower bound or don't care bits of the identifier used in message filtering
accIdUpBound	*In*	Specifies the identifier upper bound or significant bits of the identifier used in message filtering

Table 8 - Message Filtering Setup ports.

Name	Type	Description
busBit	*In*	Current bus level detected by the input transceiver
rxBit	*Out*	Bus level at the previous sample point
txBit	*Out*	Current level applied to the output transceiver

Table 9 - Bus Interface ports.

Internal Structure

The internal structure of the *CLAN Core* module is shown on Figure 2. A short description of each block is given into the following subsections.

Bit Stuffing Unit

The *Bit Stuffing Unit* is used to:

- insert stuff bits on the transmitted bit stream;
- check/remove stuff bits from the received bit stream.

It is shared by the transmission and reception parts of the core, because unless an error has occurred or the transmitter looses arbitration, within the stuffed fields the transmitted and the received bits should match.

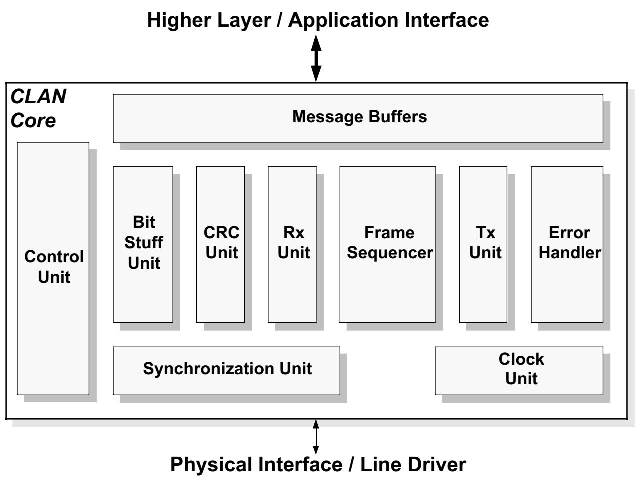


Figure 2 - *CLAN Core* internal block diagram.

CRC Unit

The *CRC Unit* calculates and checks the CRC sequence included in the frame. Similarly to the *Bit Stuffing Unit*, it is shared among the transmission and reception parts of the controller. In transmit mode, it calculates the CRC sequence during the *Start of Frame*, *Arbitration*, *Control* and *Data* fields. During the *CRC Sequence* field, the calculated sequence is shifted into the bus. In reception mode it compares the received sequence with the locally computed sequence in order to detect errors on the received bit stream.

Reception Unit

The *Reception Unit* latches the bus bit at the *Sample Point* and acknowledges a frame during the *Acknowledge Slot* field.

Transmission Unit

The *Transmission Unit* determines the bit to be transmitted by the node and sets it at the beginning of the bit time. The sources for the transmitted bit are the following:

- A message bit from the *ID*, *RTR*, *DLC* or *DATA* fields;
- A stuff bit;
- A CRC bit;
- A fixed polarity bit - recessive/dominant;
- An acknowledge bit generated by the *Reception Unit*;
- An error frame bit produced by the *Error Handler*.

Frame Sequencer

The *Frame Sequencer* plays a central role within the controller, performing the following tasks:

- Arbitration;
- Accepting requests to transmit messages;
- Detecting a start of frame in the bus;
- Sequencing fields in *Data* and *Remote Transmission Request* frames;
- Signaling the successful transmission of a message and the end of a message reception;
- Responding to overload frames.

Figure 3 shows a partial behavioral specification of the *Frame Sequencer*. It consists of two parallel state machines: the *Rx/Tx Mode State Machine* and the *Frame State Machine*. The former defines the operating mode of the controller. The last establishes the sequence of fields for all frame types except error frames, which are generated directly by the *Error Handler*.

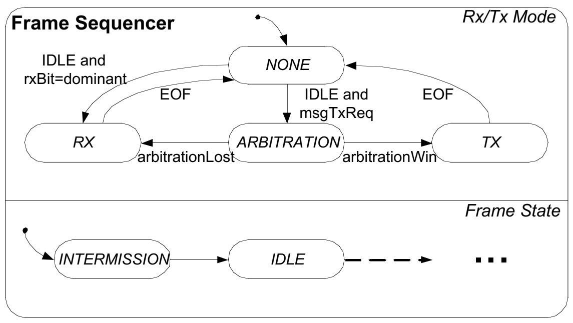


Figure 3 - Partial behavioral specification of the *Frame Sequencer* module.

Error Handler

The *Error Handler* performs all activities related to fault confinement, error detection, counting and signaling. Internally it implements the mechanisms to detect the different error types and the error counters specified on the standard. When an error frame has to be sent, the transmission is performed through the *Transmission Unit* and the *Frame Sequencer* is reinitialized.

Message Buffers

As the name implies, the *Message Buffers* are used to store messages. Two buffers are accessible from the outside of the module: one for transmission and the other for reception. However, internally the *Transmission* and *Reception* units contain shift registers that act as temporary buffers.

Clock Unit

The *Clock Unit* generates all the clocks required to control and synchronize the activities of the other core components. Behavioral modeling of the CAN controller has shown that two clock signals are required for such purposes [7]:

- a synchronization clock with frequency f_{SYNC}

$$f_{SYNC} = \frac{1}{T_Q}$$

where T_Q is the *Time Quantum* period.

- a control clock with frequency f_{CTRL}

$$f_{CTRL} = 2 \times f_{SYNC}$$

It means that for a given *Time Quantum* value, an input clock with only twice the frequency is needed.

Control Unit

The *Control Unit* generates all signals that control the other units, mainly enable and reset signals. Figure 4 shows a simplified view of the core internal control sequence within a CAN bit time. All the units are triggered at the rising edge of the *Control Clock* and during the *Time Segment 2*, i.e. after the *Sample Point*. This imposes some restrictions on the duration of the *Time Segment 2*, namely its minimum duration must be 2 *Time Quanta*. This constraint is required to decrease the number of internally generated clock signals and to limit the frequency of the clock applied to the core for a given transmission rate.

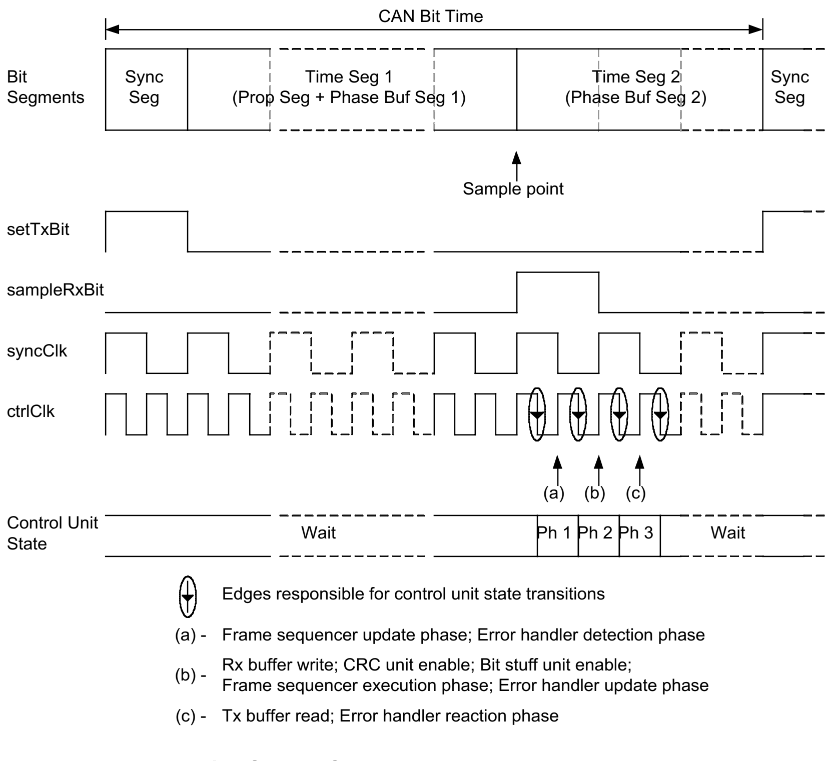

CAN Bit Time
Bit Segments
Sync Seg
Time Seg 1 (Prop Seg + Phase Buf Seg 1)
Time Seg 2 (Phase Buf Seg 2)
Sync Seg
Sample point
setTxBit
sampleRxBit
syncClk
ctrlClk
(a) (b) (c)
Control Unit State
Wait
Ph 1
Ph 2
Ph 3
Wait
Edges responsible for control unit state transitions
(a) - Frame sequencer update phase; Error handler detection phase
(b) - Rx buffer write; CRC unit enable; Bit stuff unit enable; Frame sequencer execution phase; Error handler update phase
(c) - Tx buffer read; Error handler reaction phase

Figure 4 - *CLAN Core* internal control sequence.

Synchronization Unit

The *Synchronization Unit* generates the *sampleRxBit* and the *setTxBit* clocks used to latch the reception and transmission signals at the correct time instants, based on bus transitions and on the timing parameters of the node. The period of the CAN bit is given by the following expression:

$$T_{BIT} = T_{CLK} \cdot 2 \cdot (baudRPScaler + 1) \cdot (timeSeg1 + timeSeg2 + 3)$$

where T_{CLK} is the period of the external clock applied to the core. The period T_{CTRL} of the control clock is:

$$T_{CTRL} = T_{CLK} \cdot (baudRPScaler + 1)$$

The values of the timing parameters must respect the following relation:

$$timeSeg1 > timeSeg2 > syncJumpWidth$$

otherwise the *configError* output will be active and the core will remain in the reset state.

CLAN Microprocessor Interface Module

Based on the *CLAN Core* module, different interfaces can be created. The first interface built was a synchronous parallel interface with a data bus of 8, 16 or 32 bits.

Interface Ports

The external interface of the *CLAN Controller* module is shown on Figure 5. A short description of each port is given on Table 10.

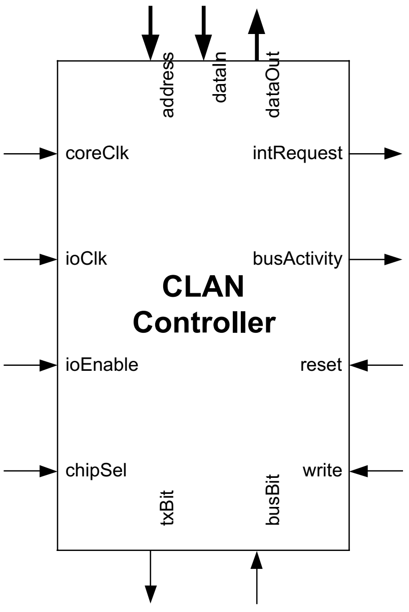


Figure 5 - *CLAN Controller* module interface.

Name	Type	Description
reset	*In*	Asynchronous reset input
coreClk	*In*	Core internal synchronization signal
ioClk	*In*	Interface synchronization signal
chipSel	*In*	Global interface enable signal
ioEnable	*In*	Enable signals for individual bytes of a multi-byte data bus interface
write	*In*	Write enable signal
address	*In*	Address bus
dataIn	*In*	Data input bus
dataOut	*Out*	Data output bus
intRequest	*Out*	Interrupt request for microcontroller
busActivity	*Out*	Flag that indicates activity on the bus
busBit	*In*	Port for connection to the reception transceiver (line-driver)
txBit	*Out*	Port for connection to the transmission transceiver (line-driver)

Table 10 - *CLAN Controller* ports.

Figure 6 shows examples of read and write cycles. The microprocessor changes the signals at the falling edge of the input/output clock. The *CLAN Controller* validates the signals at the rising edge of the same clock.

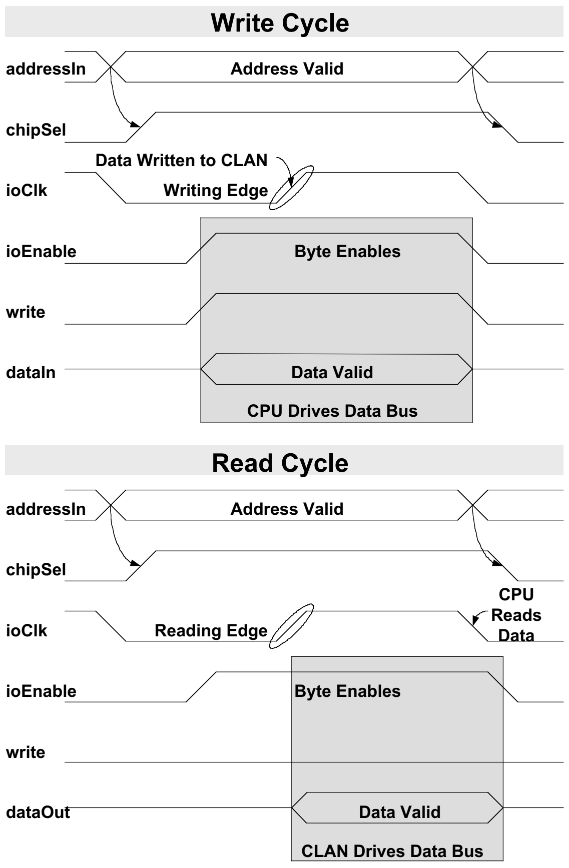


Figure 6 - *CLAN Controller* write and read cycles.

Configuration Registers

The configuration registers map into a space of 128 addresses all the input and output ports of the *CLAN Core* module. All registers are at a fixed offset location independently on the bus width (Table 11).

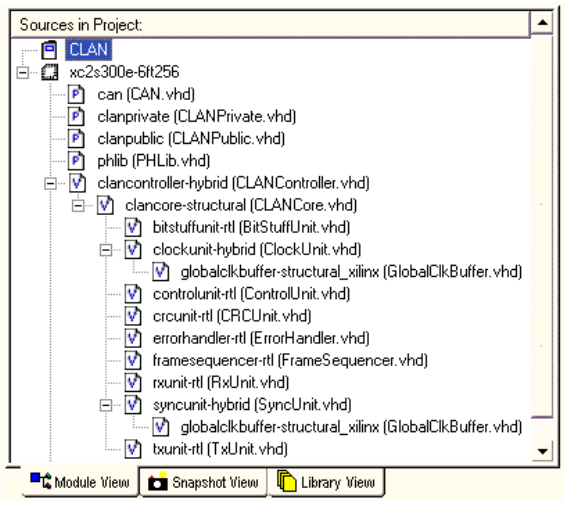
Offset (Hex)	Register Name	Access Type
00h	**Command**	*W*
00h	**Status 0**	*R*
04h	**Status 1**	*R*
08h	**Control**	*RW*
0Ch	**Rx/Tx Status**	*R*
10h	**Arbitration Lost Capture**	*R*
14h	**Error Status**	*R*
18h	**Bus Timing**	*RW*
1Ch	**Interrupt Enable**	*RW*
20h	**Interrupt Identification**	*R*
24h	**Rx Error Count**	*R*
28h	**Tx Error Count**	*R*
2Ch	**Error Warning Limit**	*RW*
30h	**Bit Error Count**	*R*
34h	**Stuff Error Count**	*R*
38h	**CRC Error Count**	*R*
3Ch	**Form Error Count**	*R*
40h	**Acknowledge Error Count**	*R*
50h	**Acceptance Mask/Lower Bound**	*RW*
54h	**Acceptance Identifier/Upper Bound**	*RW*
58h	**Rx Message Count**	*R*
5Ch	**Tx Message Count**	*R*
60h	**Rx Message Control**	*R*
64h	**Rx Message Identifier**	*R*
68h	**Rx Message Data 03**	*R*
6Ch	**Rx Message Data 47**	*R*
70h	**Tx Message Control**	*RW*
74h	**Tx Message Identifier**	*RW*
78h	**Tx Message Data 03**	*RW*
7Ch	**Tx Message Data 47**	*RW*

Table 11 - Register names and offsets.

Modeling and Simulation

The CLAN IP block was modeled with the VHDL hardware description language because VHDL provides the adequate abstractions to model the CAN controller building blocks, such as multiplexers, registers, state machines, etc. The model created contains about 3700 lines of code and it is completely independent of the implementation technology. Figure 7 shows the complete project hierarchy. To use the CLAN IP block as a black box in a project three components must be included:

- The file containing the synthesized netlist;
- The file *CAN.VHD* containing a package with generic CAN definitions;
- The file *CLANPublic.vhd* containing a package with CLAN specific definitions.



```
Sources in Project:
  CLAN
  xc2s300e-6ft256
    can (CAN.vhd)
    clanprivate (CLANPrivate.vhd)
    clanpublic (CLANPublic.vhd)
    phlib (PHLib.vhd)
    clancontroller-hybrid (CLANController.vhd)
      clancore-structural (CLANCore.vhd)
        bitstuffunit-rtl (BitStuffUnit.vhd)
        clockunit-hybrid (ClockUnit.vhd)
          globalclkbuffer-structural_xilinx (GlobalClkBuffer.vhd)
        controlunit-rtl (ControlUnit.vhd)
        crcunit-rtl (CRCUnit.vhd)
        errorhandler-rtl (ErrorHandler.vhd)
        framesequencer-rtl (FrameSequencer.vhd)
        rxunit-rtl (RxUnit.vhd)
        syncunit-hybrid (SyncUnit.vhd)
          globalclkbuffer-structural_xilinx (GlobalClkBuffer.vhd)
        txunit-rtl (TxUnit.vhd)
```

Figure 7 - CLAN project hierarchy.

Synthesis, Implementation and Test

The CLAN IP block was synthesized and implemented on a Xilinx XC2S300 Spartan-IIE low cost FPGA. The synthesis report is shown on Figure 8. It occupies about 30% of the available slices (logic cells) corresponding to 100,000 logic gates. The core internal logic can operate up to 42MHz.

The *CLAN Core* was tested within a bus with other commercial CAN controllers operating at 1Mbit/seg. The test setup is depicted on Figure 9. The main purpose of this setup is to perform a simple functional validation of the controller that must retransmit all received messages. The *CLAN Controller* module was also integrated on the ARPA System-on-Chip with a MIPS32 processor optimized for real-time systems [8].

```
Final Synthesis Report
=============================================
Device utilization summary:
Selected Device : 2s300eft256-6
Nº of Slices:           931 out of 3072 ( 30%)
Nº of Slice Flip-Flops: 863 out of 6144 ( 14%)
Nº of 4 input LUTs:    1513 out of 6144 ( 24%)
Nº of TBUFs:             32 out of 3072 (  1%)
Nº of GCLKs:              2 out of    4 ( 50%)
Timing Summary:
Speed Grade: -6
Min. period: 23.3ns (Max. Frequency: 42.8MHz)
Min. input arrival time before clock:  11.4ns
Max. output required time after clock: 10.2ns
Maximum combinational path delay:       3.8ns
```

Figure 8 - CLAN synthesis report.

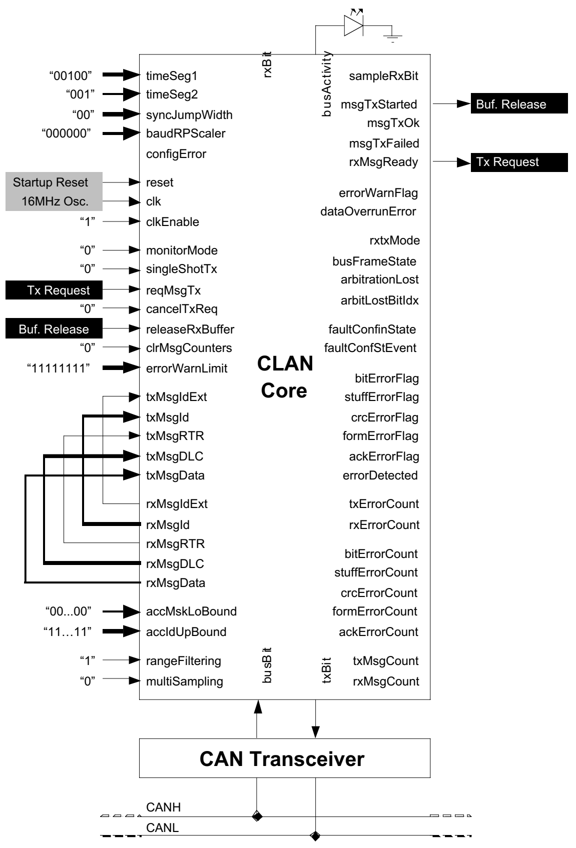


Figure 9 - *CLAN Core* loopback test setup.

Conclusion

A full CAN 2.0B controller with synchronous parallel microprocessor interface was presented in this paper. The IP core was developed for educational and research purposes. It interoperates correctly with other commercial controllers. However, it is important to refer that it was not validated with the CAN conformance tests. The web page of the CLAN project with detailed and updated information can be found at the following address: http://www.ieeta.pt/~arnaldo/projects/CLAN

References

[1] Bosch; "CAN specification version 2.0"; 1991; http://www.can.bosch.com.

[2] H. Kopetz; "Real-Time Systems: Design Principles for Distributed Embedded Applications"; *Kluwer International Series in Engineering and Computer Science, 1st ed.*; Kluwer Academic Publishers; 1997.

[3] T. Fuhrer, B. Muller, W. Dieterle, F. Hartwich, R. Hugel, M. Walther; "Time triggered communication on CAN (Time Triggered CAN - TTCAN)"; *proceedings of ICC 2000 - 7th International CAN Conference*; CiA - CAN in Automation; 2000.

[4] L. Almeida, P. Pedreiras, J. A. Fonseca; "The FTT-CAN protocol: Why and how"; *IEEE Transactions on Industrial Electronics*; vol. 49-6; December 2002 pp. 1189-1201.

[5] ISO / TC 22 / SC 3 / WG 1; "Road Vehicles - Controller Area Network (CAN) Part 4: Time Triggered Communication"; *Technical Report ISO/WD 11898-4*; ISO; 2000.

[6] CiA - CAN in Automation; "CAN Products"; http://www.can-cia.org.

[7] A. Oliveira, P. Fonseca, V. Sklyarov, A. Ferrari; "An Object-Oriented Framework for CAN Protocol Modeling and Simulation"; *proceedings of FET'03 – The 5th IFAC International Conference on Fieldbus Systems and their Applications*, Aveiro, Portugal, July 2003, pp. 243-248.

[8] A. S. R. Oliveira, V. A. Sklyarov, A. B. Ferrari, "The ARPA Project – Creating an Open Source Real-Time System on Chip", *Electrónica e Telecomunicações*, vol. 4, n.º 3, September 2004, pp.389-392.

Author Information

Arnaldo S. R. Oliveira
University of Aveiro / DET
Campus Universitário de Santiago
+351 234 370 355
+351 234 381 128
arnaldo@det.ua.pt
http://www.ieeta.pt/~arnaldo

Nelson L. Arqueiro
University of Aveiro / DET
Campus Universitário de Santiago
+351 234 370 355
+351 234 381 128
nelsonlafm@mail.pt
http://www.ieeta.pt/

Pedro N. Fonseca
University of Aveiro / DET
Campus Universitário de Santiago
+351 234 370 355
+351 234 381 128
pf@det.ua.pt
http://www.ieeta.pt/~pf