HurriCANe

VHDL CAN Controller core

Version: *Alpha 4.0 (not for 5.x)*

Date: *March 2000*

Luca Stagnaro
Spacecraft Control and Data Systems Division
Automation and Informatics Department
European Space Agency
ESTEC
KEPLERLAAN 1 - 2201 AZ NOORDWIJK - THE NETHERLANDS
TEL. (31) 71 5656565 - FAX (31) 71 5656040

Chapter
1

Forewords

This document is the user manual of the CAN Controller VHDL core.
The manual describes the internal details of the core to help the users to customize the design for their needs, or to improve the current design.

The design of this VHDL code started as a prototype activity with the intention to understand the internal of a CAN controller and not to develop the full controller. It was then gradually taken to the present level in a low level priority activity. The HurriCANe has now gone through 4 major versions with a major improvement in the release 3. The latest release, of which you are reading the manual, includes the CAN B features not available in the previous releases. NOTE: This document does not apply to HurriCANe versions 5.x.
The CAN community uses a testbench provided by Bosch to verify the design, but as a free software approach this is not expected. At present the design is tested in a lab against commercial controller, with the intention of debugging the core. If anybody has instead already the Bosch testbench is welcome to adapt it and let the community know the results.

Limit and agreement

The CAN controller is a property of the European Space Agency. It is distributed under the ESA licensing conditions, see: http://www.estec.esa.int/microelectronics/core/corepage.html

The CAN controller comes with no guarantee about the correctness and compatibility of the design to the CAN protocol.

No responsibility is taken, direct or indirect, by the author or by ESA about for any consequences derived from the use of the present core.

The code can not be distributed to third parties unless explicit written agreement of ESA..

Deviation from the CAN standard

The VHDL core in his ALPHA rel. 4.0 version implements the CAN standard B with the exception of the generation of the overload frame that is not supported. An overload frame can not be generated and is recognized as error if it is detected by the controller. The main reason for the absence of overload frame handling is that the effort are concentrating in the debug of the core with respect to the standard, with the idea that is shall be a trivial task the further improvement.

Improvements with respect to the version 3.02

The core has been modified only in the receiver and transmitter part to allow the transmission/reception of frames with extended identifiers. Two bugs has been removed since the version 3 that was originally published in the receiver side.
Since version 3 all the internal machine works on the same clock and the output CAN bit is generated on a sequent clock. The receiver is still the only protocol checker inside and is *passive* when the transmitter is on (no error frame can be generated) and active when is receiving. The stuffer/destuffer is one for the all controller as for the crc builder.
The core is now 1935 line of VHDL code and take 48% of an ACTEL SX32 in its configuration B mode.
In the file CANPckgs.vhd you find now the configuration parameters to set the core to mode CAN Standard A or B. Here below the extract of the file were the parameters are.

```
----------------------------------------------------------------------------
  --                                                                      --
  --                     CONFIGURATION PARAMETERS                         --
  --                                                                      --
  --------------------------------------------------------------------------
  -- INSTRUCTION
  -- You must comment out the declaration of the standard you do not want to
  -- use, and uncomment the other. This is limited, as you see, to
  --     CANStd, ARB_FIRST, ARB_LAST, DATA_FIRST, FrameBytesCounter
  --
  -------------------------------------------------------------------------

  --------------------------------------------------------------------------
  --                                                                      --
  --                    Standard A: 11 bits identifier                    --
  --                                                                      --
  --------------------------------------------------------------------------
  constant CANStd      : CANStdType := A;

  constant ARB_FIRST   : natural := 0;
  constant ARB_LAST    : natural := 10;
  constant DATA_FIRST  : natural := 18;

  subtype FrameBytesCounter is integer range 0 to 15;

  ---------------------------------------------------------------------------
  --                                                                       --
  --                    Standard B: 11 and 28 bits identifier              --
  --                                                                       --
  ---------------------------------------------------------------------------
--   constant CANStd       : CANStdType := B;

--   constant ARB_FIRST    : natural := 0;
--   constant ARB_LAST     : natural := 10;
--   constant ARBEXT_FIRST : natural := 13;
--   constant ARBEXT_LAST  : natural := 30;
     constant DATA_FIRST   : natural := 38;
```

You must comment out the standard that is not implemented, and live the other part uncommented. Each part defines the CANStd, DATA_FIRST, ARB_FIRST, ARB_LAST, ARBEXT_FIRST, ARBEXT_LAST and FrameBytesCounter for each of the modes.

You probably want to use the mode A only when you want to save some gates in your design, because in mode B you are capable of receiving CAN frames with both 11 and 29 bits.

Core Interfaces

The following table list in the input/output port of the CAN core

Core declaration

```
entity can_core is
  port(
    bit_stream : in std_logic;
    remote_frm : in std_logic;
    reset      : in std_logic;
    clock      : in std_logic;
    tx_msg     : in CANMsg;
    start      : in std_logic;

    sample       : out std_logic;
    bus_off      : out std_logic;
    tx_bit       : out std_logic;
    rx_completed : out std_logic;
    tx_completed : out std_logic;
    rx_msg       : out CANMsg;
    err_passive  : out std_logic;
    rx_err_cnt   : out std_logic_vector(0 to 7);
    tx_err_cnt   : out std_logic_vector(0 to 7);
    err_bit      : out std_logic;
    ack_bit      : out std_logic;
    enable       : out std_logic);
end can_core;
```

NAME	Dir.	Meaning
REMOTE_FRM	In	Remote Frame response It tells the transmitter in the core that the data frame is to be transmitted only as response to a remote frame
BIT_STREAM	In	Bit Stream This is where the bit read from the bus are input in the can core.
RESET	In	General Reset It is active high, as all the reset in the core, and is used to reset the core.
TX_MSG	In	Data Frame Bits This is the input message that the transmitter put on the bus according to the CAN protocol.
START	In	Start the message transmission

		It requests the transmitter inside the core to initiate the transmission of the message contained in the input "msg".
CLOCK	In	Input clock This is the clock that is used to internally to operate the core. It is normally 16 time faster than the actual CAN bus frequency.
BUS_OFF	Out	CAN is in bus off This signal is to notify externally that the error counters have reached the maximum and that the controllore shall go in bus from the bus.
TX_BIT	Out	Transmitter out bit This is the bit stream generated inside the CAN controller
RX_COMPLETED	Out	The reception of one CAN message is completed This signal that the receiver in the CAN core has succesfully received one message. This signal last 1 CAN clock cycle, therefore it must be latched outside.
TX_COMPLETED	Out	The transmission of the message is completed This signal notify outside that the requested message has been succesfully transmitted and that the transmitter is waiting to complete the transmission cycle. The controller of the core shall now lower the start signal to allow the transmitter to return to the idle state.
RX_MSG	Out	Copy of the received message This is the copy of the received message
RX_MSG_ID	Out	ID of the received message This is only the arbitration part of the message received
SAMPLE	Out	Debug signal asserted when a new CAN bit is issued
ERROR_PASSIVE	Out	Asserted when the core goes in error passive mode
RX_ERR_CNT	Out	Receiver Error Counter For debug porpoise or to check the quality of the line.
TX_ERR_CNT	Out	Transmitter Error Counter For debug porpoise or to check the quality of the line.
ACK_BIT	Out	Signal that generates the acknowledge bit
ERR_BIT	Out	Signal that generates the error bit
ENABLE	Out	CAN Controller internal clock

Transmission of a Data Frame

The transmission of a data frame starts with the assert of the start signal. The transmitter inside the CAN core does not start immediately the transmission but can be put into wait by an on-going reception of a frame. For this reason the start signal shall be kept asserted until a confirmation of the actual transmission of the bit contained in the msg bus is given. When the tx_completed signal is asserted the start signal shall be then de-asserted to allow the transmitter machine to get ready for the next message to transmitt. The tx_completed signal is not de-asserted until the start signal is to be sure the core user is not missing this information.

Transmission of a Remote Frame

The transmission of a data frame and remote frame differs for only two details: first the remote_frm signal shall be asserted before the start one and second the actual transmission message does not start immediately even if the bus is idling. In this case the transmitter expect first the receiver part to acquire a message with the same arbitration and dlc before sending its message. If before this happens the core user has de-asserted the start signal the operation is cancelled. Therefore it is important to **always keep the start signal asserted until the tx_completed signal is asserted.**

Reception of frames

The core user shall wait for the rx_completed message to be asserted before reading the value contained in the rx_msg bus. Be aware that the rx_completed signal is asserted only for 1 CAN clock cycle (16 times the core clock) and therefore it must be either sampled quick enough or latched to not missing this information.
Rx_msg_id is changed every time a complete arbitration is acquired by the receiver regardless the correct reception of the message itself. It is used to trigger in time the message filtering part in a complete controller. There is not signal telling when this value changes.
Rx_msg instead is the received data and is copied into this bus after a complete and successful reception of the message. The receiver implements a double-buffer structure and therefore before the rx_msg bus is overwritten it requires the complete reception of a new frame.

Controller Structure

The controller is divided into the following units

- Synchronizer
  It synchronizes the controller with the incoming bit_stream and generates the internal clock and warm resets.

- CRC_CALC
  It is the common crc calculator for both receiver and transmitter. It always uses the receiving stream to calculate the crc and when needed by the transmitter it activated by the crc_first and crc_shiftout signals.

- CAN_RX
  This unit received and interprets the message. It has no direct possibility to write on the bus if not indirectly via the “do_error_frame” signal to the transmitter. It communicate the frame stage to the transmitter, checking all the errors with the exception of the bit_error that can only be checked by the transmitter.

- CAN_TX
  It is the CAN message transmitter. The message to transmit come directly from the decoder. It contains an state machine to keeps control over the part of the message being sent and a possible errors, a shift-register a CRC calculator and a bit stuffer.

- Error Frame Generator

It generates error frame up request of the CAN_RX or CAN_TX and also checks that the overcomplicated error specification is followed.

- StuffHandler
  It advise the other state machine when the incoming stream contains (or should contains) a stuffing bit. This information is used to insert stuffing bit in transmission or to delete stuffing bit in reception. The actual stuffing is done in the synchronizer for the transmitter part.

- Error_counter
  This simple counts the number of error and set the states of error_passive and bus off as stated by the standard.