CAN Peripheral

Jesús Arias

1 Introduction

This is a CAN bus controller peripheral designed for 32-bits cores. It was initially designed for a RISC-V CPU with a different bus architecture (PicoRV, LaRVa, and similar cores) and, in order to simplify the interfacing to the QV core, only 32-bit reads and writes are allowed. This document presents the original peripheral for those cores.

The CAN bus is basically a quite complex serial port that, in addition to the serialization of transmitted and received data, has to deal with some other things like:

- A bidirectional, half-duplex, physical media with a wired-AND logic (logic #0 is dominant).
- Hard and soft bit time resynchronization. Bit time segments for sampling.
- Bit stuffing: An extra bit with opposite polarity is inserted in the data stream if the previous five bits have the same value.
- ID filtering.
- Bus arbitration.
- Receiver acknowledgment.
- CRC error detection.
- Error detection, signaling, counting, and error state management.
- Automatic retransmission of frames aborted due to errors or lost arbitration.

Of course, the proposed peripheral lacks many of these features because it was already complex enough without them. In spite of these limitations it is still capable of transmitting and receiving CAN frames and it could be made more specification compliant with some software help. This is a list of missing things:

- The received bits are sampled at the middle of the bit time. There are no time quanta nor bit time segments to configure. Yet, this can still work without problems with almost all the CAN buses around.
- Clock resynchronization is always hard. Any data transition in the receiver resets the clock divider and moves the sampling instant to half a bit time later.
- The receiver has no ID filters. All valid frames are received. Filtering has to be done by software.
- There are no error counters nor error state machines. The bus-off state isn't supported. All this functionality has to be handled by software.
- The transmitter attempts frame transmission a single time (one shot mode). It it fails the frame has to be transmitted again explicitly.

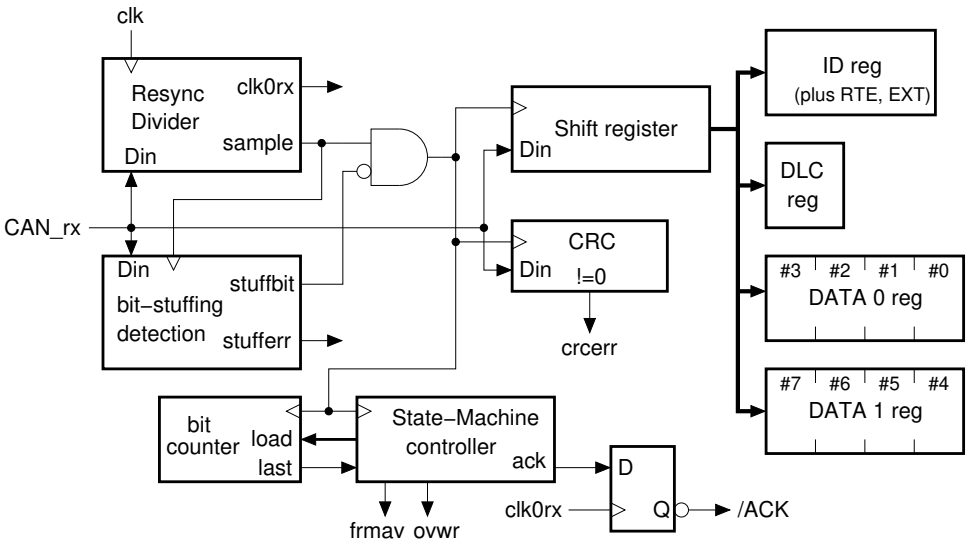
- The transceiver can't generate active-error frames (six or more consecutive zero bits). This controller is thus always operating in the error-passive state.
- There are no buffers. For the receiver this means that registers start to get overwritten as soon as a new frame arrives. In the transmitter the contents of the registers is lost after a transmission, even if the transmission was aborted, because these registers are in fact the output shift registers.

The CAN peripheral requires 641 logic cells in the ICE40HX FPGAs, this is about six times the complexity of the UART, and its maximum clock frequency is 82MHz. Internally it is divided into a separated receiver and transmitter, each of them with:

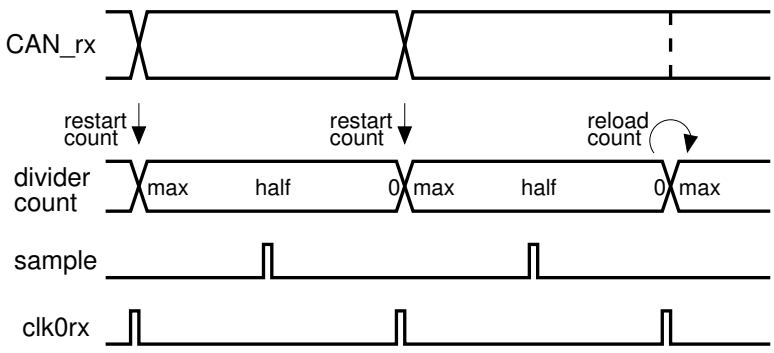
- A resynchronizable clock divider (10-bit downcounter).
- An state machine controller (8 states)
- A presetable bit counter (6-bit downcounter).
- A 5-bit shift register for the detection of stuffing bit conditions.
- Shift registers for ID, DLC, and DATA fields of the frame.
- A 15-bit CRC register.
- Flag management logic.

A description of the receiver and transmitter blocks follows.

1.1 CAN receiver



A simplified block diagram of the CAN receiver is shown in the above figure. This isn't an exact representation of the actual logic (the receiver used a single clock, not several), but I hope it still serves to clarify the workings of the circuit. The first block I want to mention is the resynchronizable clock divider.

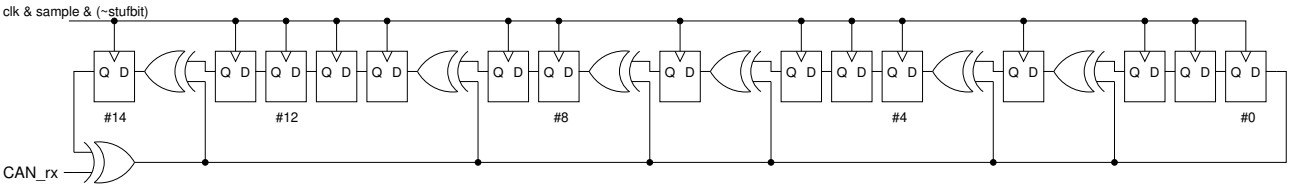


This circuit samples the CAN bus input using the fast system clock and compares the last two samples. If they are different the divider counter is loaded with its maximum value (from a register not shown in the figure) and starts downcounting. Two pulse outputs are generated, one when the counter reaches half of its maximum value (sample) and another when reaches zero (clk0rx). "sample" happens at the middle of the input bit time and it is used to drive the rest of the sequential logic, except for the generation of the ACK pulse that is timed by "clk0rx".

The next block is a 5-bit shift register that is used for stuffing bit detection. If its contents are all zeroes or all ones its "stufbit" output is asserted. This inhibits the shifting on the current bit into the shift register and the CRC checking blocks. The current bit is also checked and an error signal is generated if six zeroes or ones are detected.

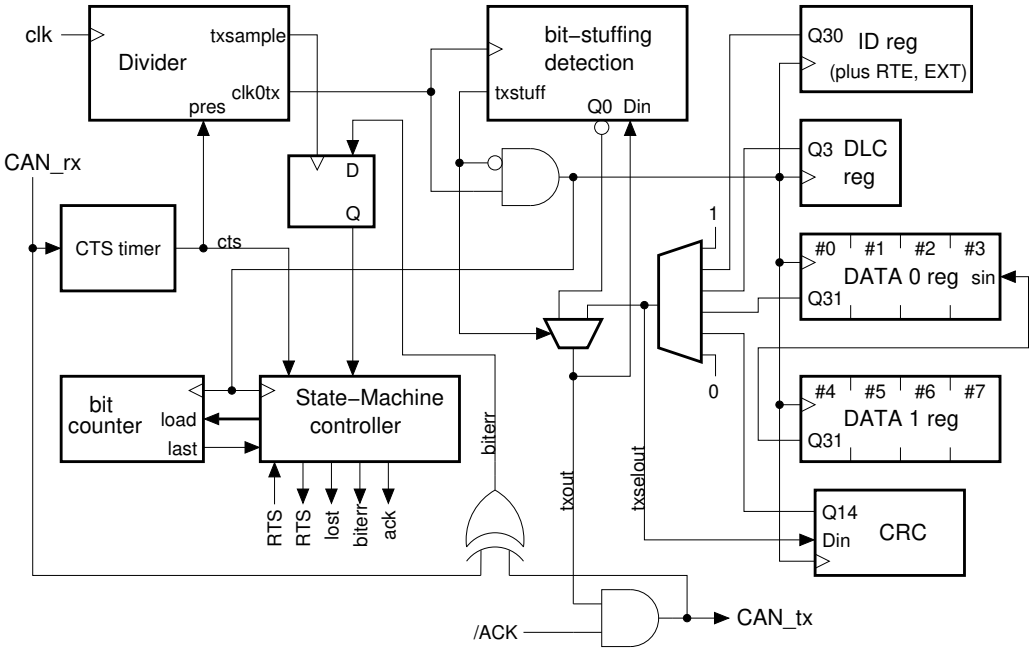
The shift register is a 21-bit, left shift register, that holds a temporary copy of the received data. Its partial contents are written into one of the destination registers at the proper time: 13 bits to the ID plus RTR/EXT flags register, followed by another 21 bits if an extended ID is received, 4 bits to the DLC register, and 8 bits to one of the 8 data registers. These last registers are grouped into two 32 bit registers: "DATA0" and "DATA1".

Finally, there is a finite-state machine controller that, along a bit counter (6-bit), handles all the signals and timing. There are 8 different states, many of then associated to a particular segment of the CAN frame being received. The controller stays in these states until the bit counter downcounts to one (the zero count is omitted to ease the counting of data bits: The counter stars with 8*DLC) and then new values are written into the state register and the bit counter. Error conditions are also included in the state machine logic (an error-active frame will move the state to an error state), and also the handling of the receiver flags FRMAV (Frame Available) and OVWR (overwrite). For instance, the FRMAV flag is set if the current state is the associated with the CRC field, the bit counter is one, and all the bits of the CRC register are zero (CRC check OK).



The CRC circuit follows the basic schematic of the above figure, also including the possibility of being reset at the reception of the Start Of Frame bit (SOF) and the OR of the 15 flip-flop outputs as the CRC error flag. All the non-stuffed bits of the frame are passed through the CRC circuit, including the CRC field itself. A frame without errors will result in a final CRC value of zero.

1.2 CAN transmitter



The block diagram for the CAN transmitter is presented above. Again, this diagram isn't an exact representation of the actual circuit, but it serves to highlight its main components. The first interesting thing about this transmitter is that it also requires the same data as the receiver, CAN_rx. This is due to the half-duplex characteristic of the CAN bus where transmitters can't start to send data as soon as they wish, but they have to wait until the bus isn't used by another node. In order to comply with this requirement the "Clear To Send" block is included. This block provides a CTS signal after 11 bit times with the bus in a recessive state, that account for the trailing ACK delimiter of the last frame, the 7 bits of the End Of Frame segment, and 3 additional bits for the inter-frame space. This CTS signal also synchronizes the clock divider in order to get the falling edge of the SOF bit exactly 11 bit times after the rising ACK edge. This is the only case when the clock divider is synchronized, it is just a free running counter afterwards.

A notable difference with the receiver is the shifting of bits. Here the shifting is done in the frame related registers themselves (ID, DLC, DATA0, DATA1, and also CRC) instead of using a separate shift register. The data registers are arranged in a mixed-endian configuration where the first data bit sent is the bit #7 of the byte #0 (bit 31 of DATA0). A multiplexer selects which shift register to route to the "txselout" signal depending on the state of the finite-state machine controller. This data can also be forced to be dominant during the SOF or recessive during the EOF, idle, or waiting states.

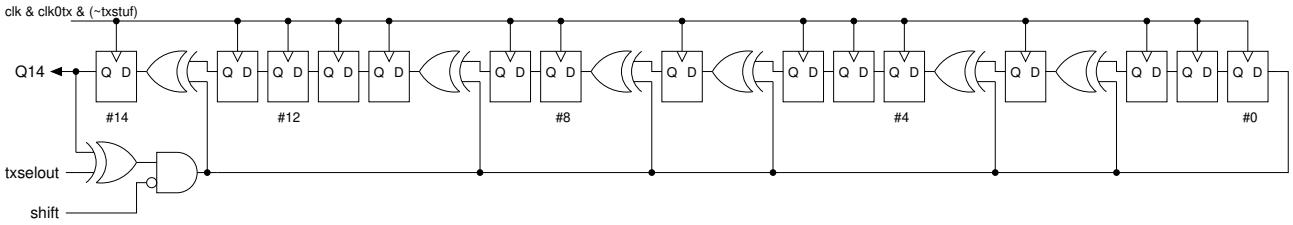
But this signal isn't yet the transmitter output, the stuffing logic can halt the shifting and insert an stuffing bit instead. This happens after 5 consecutive bits with the same logic value and on these cases the complement of the last bit is routed to the output, "txout". The bit stuffing is enabled only during the ID, DLC, DATA, and CRC segments of the frame. "txout" is finally ANDed with the /ACK output of the receiver, obtaining the "CAN_tx" output ready to drive the bus transceiver.

In the CAN bus any signal transmitted is also received and it supposed to have the same logic value as the data being transmitted, but these two values can be different if another node is transmitting a dominant bit while our bit is recessive. This can happen in three different cases:

- Another node is transmitting an ID field with higher priority. This isn't an error: it is a lost arbitration, but the transmitter has to abort immediately and to put its output into a recessive level.
- During the DLC, DATA, or CRC fields of the frame this can also happen if another node is sending an error-active frame (six or more dominant bits). This is an error situation and the transmitter has to abort immediately.

- During the ACK slot the receivers on the bus are going to send a dominant bit while the transmitter sends a recessive one. This is the normal ending of frames, not an error.

The CAN_tx output is compared to the CAN_rx input and the resulting 'biterr' signal is sampled at the middle of the bit time (txsample clock). The finite-state machine controller uses this information to abort the sending of frames or to signal the proper acknowledgment by the receivers. Three flags are provided to notify the losing during ID arbitration (LOST), the receiving of an error-active frame, (BITERR), or the correct acknowledgment, (ACK).



The CRC circuit is almost the same as that of the receiver, but with an small diference: It can be turned into a plain shift register by means of a single AND gate in series with the feedback line of the register (see above figure). This is done during the sending of the CRC contents, near the end of the frame.

1.3 System interface

The following signals have to be connected to the CAN controller module:

Name	Direction	active	Function	Comments
clk	input	rising edge	main clock	same as CPU clock
cs	input	high	Chip select	from address decoder
[1:0]rs	input	high	Register select	from CPU address
[3:0]bytesel	input	high	byte select	active for writes (one per byte)
[31:0]d	input	high	input data bus	from CPU data output
[31:0]q	output	high	output data bus	to CPU data input mux.
irqrx	output	high	RX IRQ	same as FRMAV flag
irqrxerr	output	high	RX Error IRQ	STUF or CRC flags active
irqtx	output	high	TX IRQ	RTS flag low
can_rx	input	dominant low	CAN bus RX	from bus transceiver
can_tx	output	dominant low	CAN bus TX	to bus transceiver

Notes:

- Registers are 32-bit wide. Therefore "rs[1:0]" is usually connected to CPU address "ca[3:2]".
- Frames received with errors don't activate "irqrx", but they activate "irqrxerr". These two lines could be ORed into a single "irq_rx" line.

1.4 Programmer's model

The whole CAN controller needs only the I/O space of four word registers selected via its RS inputs:

RS[1:0]	name	Write Register	Read Register
0	ID	ID to transmit	received ID
1	DLCF	DLC, RTS for transmit, BAUD	received DLC, flags
2	DATA0	first 4 bytes to Transmit	first 4 bytes received
3	DATA1	second 4 bytes to Transmit	second 4 bytes received

Notice that most registers are write-only or read-only, and, while they can share the same address they are in fact different registers. The content of these resisters is detailed next:

ID registers: ID of the frame to transmit if written, ID of the last frame received if read:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RO/WO	RO/WO		RO/WO												
EXT	RTR	-	ID MSBs (for ext IDs)												
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RO/WO					RO/WO										
ID MSBs (for ext. IDs)					ID LSBs (ext. or std. IDs)										

(RO: Read only. WO: Write only. R/W: read and write)

- Bit 31. EXT: Set to 1 for extended IDs with 29 bits. Set to 0 for standard IDs with 11 bits.
- Bit 30. RTR: Remote Request. Frames with this bit set lack a data payload even if their DLC fields is nonzero.
- Bits 28:0. ID field for extended-ID frames.
- Bits 10:0. ID field for standard-ID frames. Bits 11 to 28 are ignored or invalid for standard IDs.

The reading of the ID register also has the effect of clearing the receiver flags (bits 4 to 7) in the DLCF register.

DLCF register: DLC field for frames to transmit if written, or DLC of the last received frame if read, along with receiver flags (bits 4 to 7), transmitter flags (bits 8 to 11), and baud divider:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R/W	R/W	R/W				R/W									
IET	IEE	IER	-	-	-	BAUD divider									
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				RO	RO	RO	R/W	RO	RO	RO	RO	RO/WO			
-	-	-	-	ACK	BIT	LOST	RTS	OVWR	FRMAV	CRC	STUF	DLC			

(RO: Read only. WO: Write only. R/W: read and write)

- Bits 3:0. DLC: Data Length Code or number of data bytes of the frame. Values 9 to 15 are invalid. RTR frames have no data but usually have nonzero DLC fields.
- Bit 4. STUF. A value of 1 in this flag means the reception of more than 5 consecutive bits with the same logic value and the abortion of the frame. This could happen if a device on the bus sends an error-active frame (6 or more dominant bits) interrupting and overwriting another frame. This flag is cleared after reading the ID register or at the start bit of a new frame.
- Bit 5. CRC. A value of 1 in this flag means the reception of a frame with a wrong CRC. This flag is cleared after reading the ID register.
- Bit 6. FRMAV. Frame Available. This flags is set after the reception of a valid frame and it is cleared after the reading of the ID register. This bits is set on the last CRC bit of the frame and we must wait no more than the equivalent time of 28 bits before reading the received data or registers could get overwritten. Notice that frames with Stuffing errors or CRC errors don't activate FRMAV.

- Bit 7. OVWR. Overwrite. This flag is set if a frame is received while FRMAV is still active. It is cleared reading the ID register. An overwrite error exist as soon as the ID of the new frame is received if FRMAV is active.
- Bit 8. RTS. Request To Send. This read/write bit must be written with one to start the transmission of the frame previously loaded into the ID, DLC, and DATAx registers. This flag is cleared automatically after the transmission of the frame (or its abortion)
- Bit 9. LOST. Arbitration was lost to a device with higher priority during the ID/RTR phase of the frame. This means that some of our recessive bits (ones) were read as dominant in the bus, resulting in the abortion of the transmission.
- Bit 10. BIT error. A recessive bit was read as dominant in the bus during the DLC, DATA, or CRC phase of the frame, resulting in the abortion of the transmission. This could happen if an error-active frame is transmitted over the bus a the same time than our data.
- Bit 11. ACK. At least one receiver on the bus acknowledged our frame if set.
- Bits 25:16. BAUD divider. The bit time is obtained multiplying the core clock period by (BAUD+1), or conversely, the data rate is: $bps = f_{CLK}/(BAUD+1)$.
- Bit 29: IER. Interrupt Enable on RX. Enables the interrupt when FRMAV is set.
- Bit 30: IEE. Interrupt Enable on Errors. Enables the interrupt when STUFF or CRC are set.
- Bit 31: IET. Interrupt Enable on TX. Enables the interrupt when RTS is zero.

Some of these flags can be used to trigger interrupts if desired. FRMAV can request an interrupt when a valid frame is received, while STUF and CRC can request interrupts on receiver errors. In the case of the transmitter the interrupt can be requested when RTS becomes zero, meaning a new frame can be transmitted (or the same frame if LOST or BIT were active)

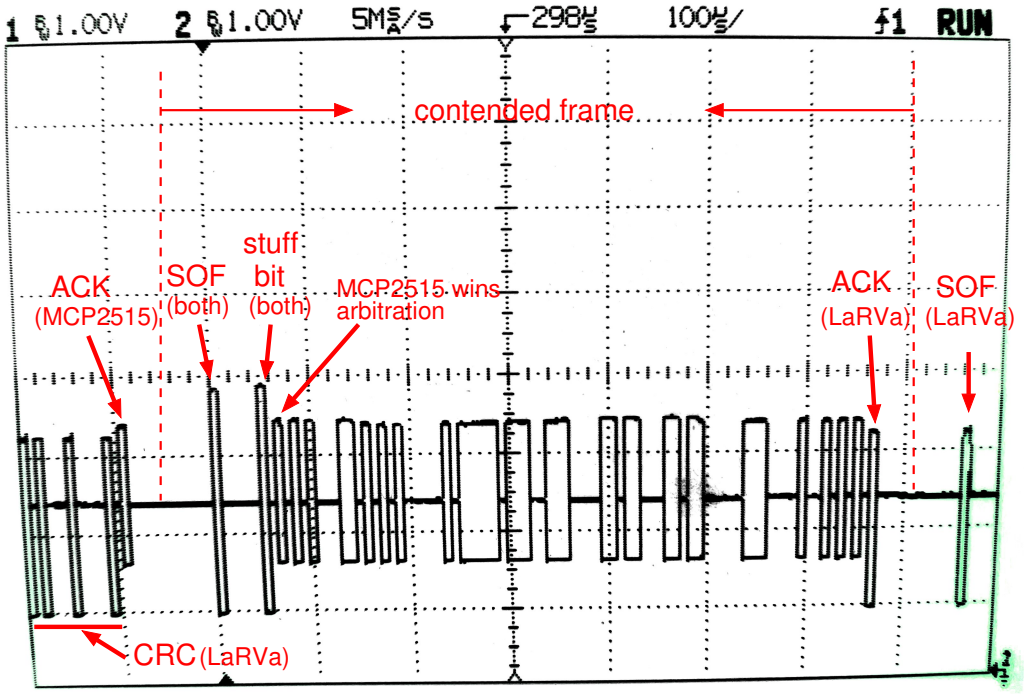
1.5 CAN bus arbitration

As a last feature the receiver is muted while transmitting. This avoids being receiving our own frames, and more importantly, acknowledging them. But this has to be done with care due to arbitration. The receiver is only muted after the transmitter wins its arbitration, that is after the ID/RTR field of its frame. In a won transmission the receiver will thus detect an stuffing error and will abort its frame, but no error flag will be set.

Bus arbitration was a constant worry. It is easy to say “If several nodes start transmitting simultaneously the one with the lowest ID wins the arbitration...”, but there must be some way to synchronize the nodes to start transmitting simultaneously to begging with. My idea here was to generate a “Clear To Send” signal that becomes active after 11 recessive bits are detected on the bus, and a transmitter clock that starts in phase with this CTS signal.

A pending transmission will start sending its Start Of Frame bit out just after CTS gets active, and, if other nodes were awaiting transmission they will also start at this precise moment, so, arbitration is going to take place correctly.

This behavior was hard to debug, but at the end I was able to capture one of such contentions in a scope screen:



The LaRVa node was using a discrete transistor transceiver that generates lower voltages in the CANL line than the integrated MCP2551 of the other node (a MCP2515 controller), so, looking at the pulse amplitudes we can differentiate who is driving the dominant bits of the frames. In fact, in the capture shown in the snapshot, the MCP2515 node generates the ACK bit of the last frame before the 11-bit gap between frames. Then, the Start Of Frame bit, and also the following stuffing bit, both dominant, have a bigger amplitude than usual because both nodes are driving the bus in perfect synchrony. This last until the MCP2515 node sends a dominant bit while the LaRVa bit is recessive and the former wins the arbitration (extended ID 0x1FAA55F8 wins over 0x1FFF1234). The rest of the frame shows a smaller amplitude in the dominant bits because the LaRVa transmitter is now disabled, until the ACK bit, that is generated by the LaRVa receiver.

In the LaRVa system the end of a transmission triggers an interrupt and in it's service routine the LOST flag is detected. This forces the sending of the same frame again until it is successfully transmitted (error flags clear and ACK set)

The schematic of the 3.3 volt, discrete transistor, CAN transceiver and split terminator follows: