LaRVa Peripheral Collection

Jesús Arias

These are some peripherals designed to be included along the LaRVa core to provide it with interfaces to the usual serial data buses around and with other commonly used functions. There were designed as the simplest possible circuit block in most of the cases while also retaining the required functionality for their practical use.

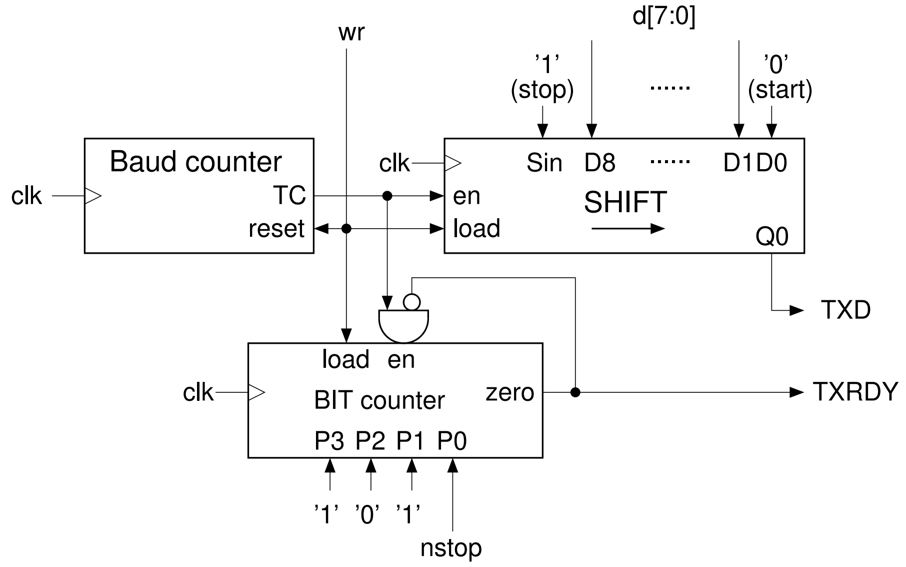
1 UARTs

Minimal

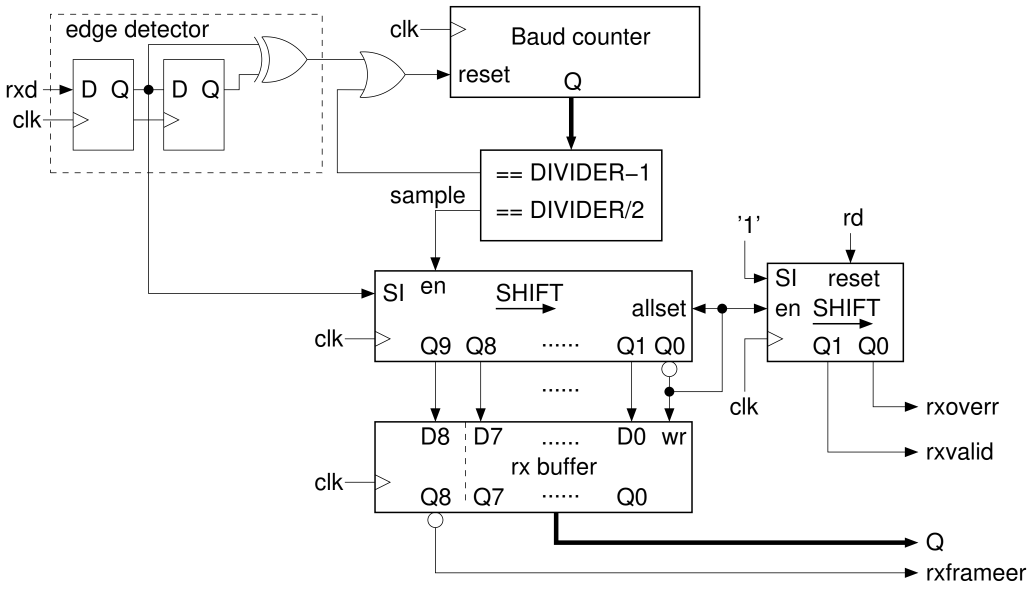
Two different UART modules were designed: a minimal one and another with more features. The simpler one has the following characteristics:

- 8 data bits, No Parity. 1 or 2 Stop bits.
- Fixed baud rate with no 1/16 or 1/8 prescaler. Baud rates as fast as $f_{CLK}/6$ are possible. Baud rate is defined with a clock divider parameter, 'DIVIDER', during synthesis.
- RX clock resynchronization on every data transition.
- No buffer register for TX. One byte buffer for RX.

A simplified diagram of the transmitter is shown next. It includes a clock divider, a 9-bit shift register, and a bit counter. The clock divider (baud counter) gets reset when a data is written into the shift register or when it reaches its maximal count, and provides an strobe pulse for data shifting and bit counting. The bit counter is loaded on writes with ten or eleven, depending on the number of Stop bits, and downcounts until it reaches zero. When in zero state the tx_ready flag is asserted. This is all the logic needed for a functional UART transmitter.



The diagram of the receiver is shown next. It also includes a baud counter that get reset when it reaches its maximal count (DIVIDER-1) or when an edge is detected in the incoming data stream. Also, a sampling strobe is asserted at the middle of the bit time, enabling the shifting of the incoming bits. The shift register has 10 bits and when the start bit of the data arrives at Q0 all its bits are preset as ones while the rest of the register is copied to the output buffer, including the received Stop bit. A value of zero in that bit means a framing error. Another two flags are also present: an 'rxvalid' flag is asserted every time a data is stored in the buffer, and a 'rxoverr' flag can also be set if 'rxvalid' is already active when data is stored. These two flags are reset by means of an external read strobe, 'rd'.



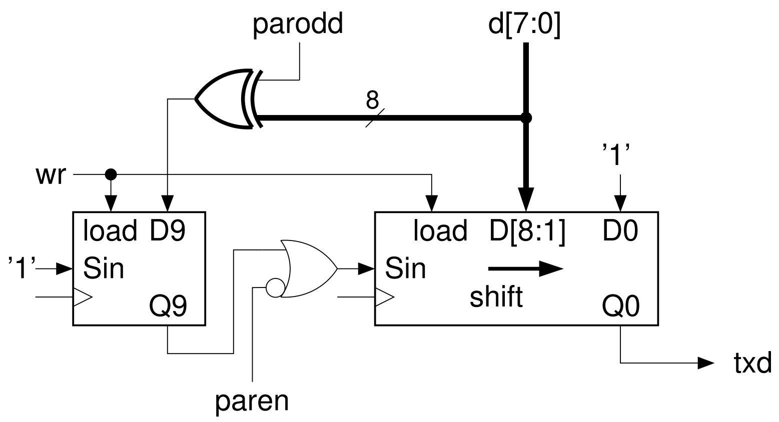
This minimal UART required 76 logic cells in an ICE40HX FPGA for a DIVIDER value of 217. Its maximum clock frequency was nothing less than 303 MHz.

Enhanced

An enhanced UART version was also designed. The differences with respect the minimal one are:

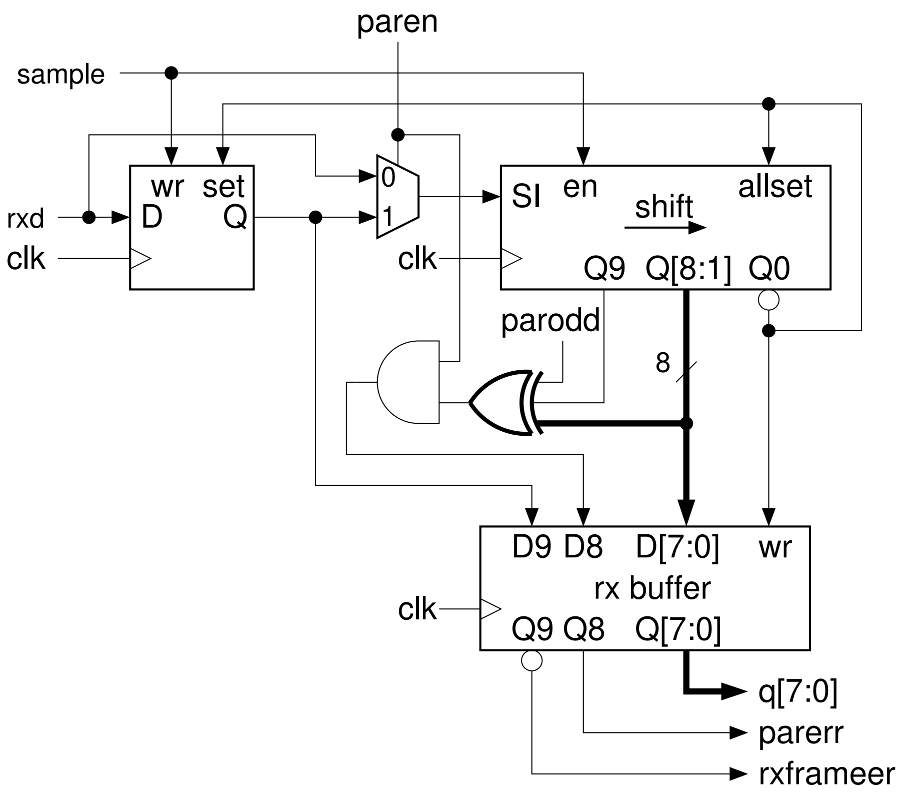
- Baud rate selectable at any time thanks to a 'bauddiv' input (DIVBITS wide).
- Parity is now supported. Two control inputs, 'paren' for parity enable, and 'parodd' for even / odd parity selection, are now included. Also, a new flag 'parerr' will signal the reception of a data frame with wrong parity if parity is enabled.

Of course, some changes were required in both the transmitter and the receiver. First, the maximal count for baud counters is now coming from an external input instead of being a constant, and next, parity management resulted in tweaks in the bit shifting, and in the transmitter case also in the bit counting that now can be 10, 11, or 12 bits depending on the number of stop bits and the enabling of parity. The transmitter shift register now has another bit more attached to its most significant bit in the following way:



As we can see, the new most significant bit of the shift register gets loaded with the parity of the data, but if parity is disabled this bit is ignored and an '1' (stop bit) is shifted into the lower bits of the register instead.

The receiver shift register also has a bit more, totaling 11 bits. But that bit can be bypassed if parity isn't enabled. And, of course, it also includes a parity checker circuit:

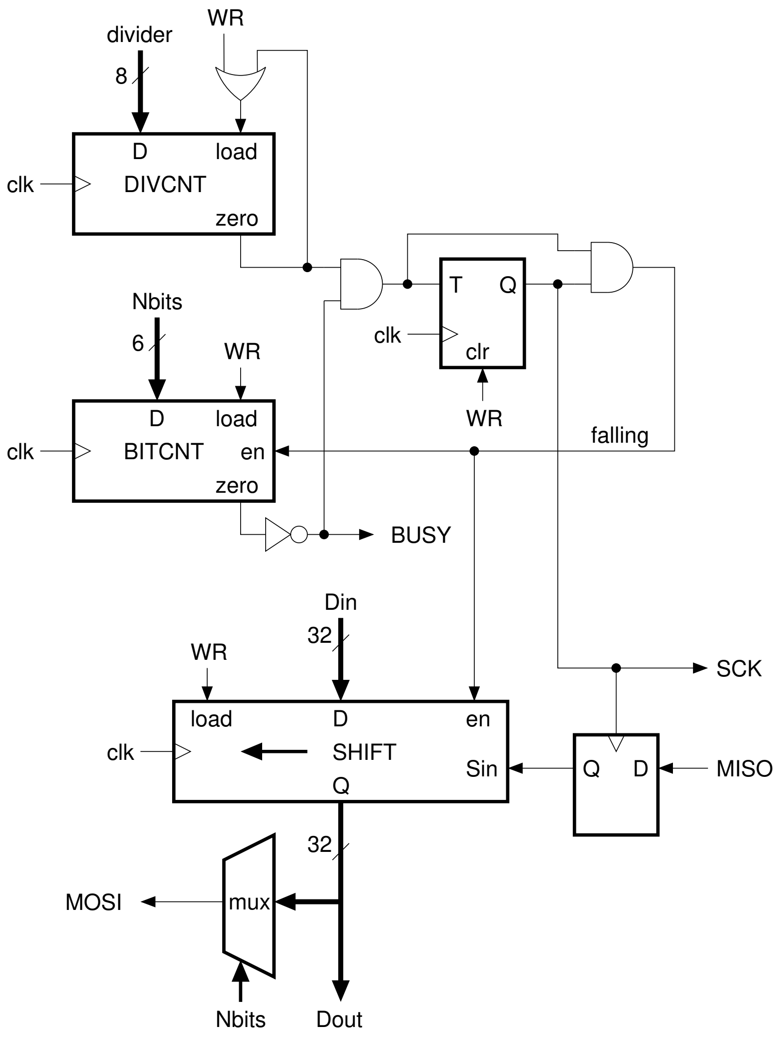


In these diagrams the XOR gate with thick lines is in fact the parity generator or checker. (A parity generator is no more than a cascade of XOR gates)

The enhanced UART requires 99 logic cells if baud rate counters are 8-bit wide, or 109 logic cells for 12-bit counters, and its maximum clock frequency is 183 MHz. Of course, this UART could be further improved to include other features found in commercial parts, like FIFOs, 9-bit data instead of parity, automatic Break generation...

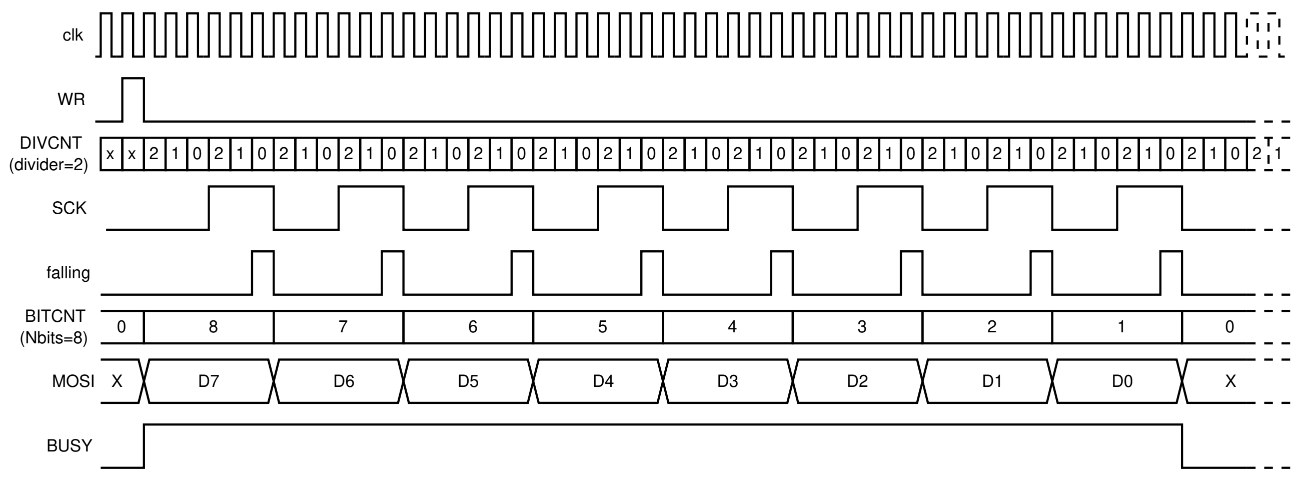
2 SPI controller

The SPI controller is always a master that can transfer data with a variable number of bits between 8 and 32. This feature was included to ease the interface to some devices with weird data lengths (mainly ADCs like the HX711). The block diagram of its inner logic is shown next:



In this schematic the values "divider" and "Nbits" come from a control register not shown, and also the "BUSY" flag can be read from an status register. This is a 99% synchronous design with almost all clock signals connected to the same clock as the CPU (the exception is the flip-flop for the sampling of the MISO input). The "WR" input is pulsed high when a data is written to the SPI data register, that is in fact the same register used for the shifting of bits.

There are two counters that count down and activate a "zero" signal when their value is zero. One is used for the generation of the SPI clock, while the other counts the number of bits pending to transfer. Both counters get loaded when the SPI data register is written, and the DIVCNT counter also gets loaded every time its count reaches zero, thus generating a periodic pulse that toggles the SCK signal. The resulting waveforms are shown in the following graph, were "divider" was 2 and "Nbits" was 8.



As we can see, SCK toggles every 3 clock cycles and the SPI frequency is thus 1/6 of the clock. The formula for the SCK frequency is then: $f_{SCK} = f_{CLK}/[2*(divider+1)]$ The "falling" pulses are used to notify the shifting of bits and the decrementing of the bit counter. The incoming data is also shifted into the same register, but it has to be sampled on the rising edges of SCK, so, and additional flip-flop is used for the sampling of the MISO input. The MOSI output is selected from the bits of the shift register depending on "Nbits" by means of a multiplexer.

Also, not show in previous figures, is the clearing of all the unused higher bits of the shift register, that required the placement of AND gates controlled by "Nbits" between the flip-flops of the shift register. This last feature was added to avoid nonzero values in the MSBs of the received data when the data length was less than 32 bits.

The SPI controller uses 227 logic cells and has a maximum clock frequency of 115MHz when synthesized in an ICE40HX FPGA.

3 I2C controller

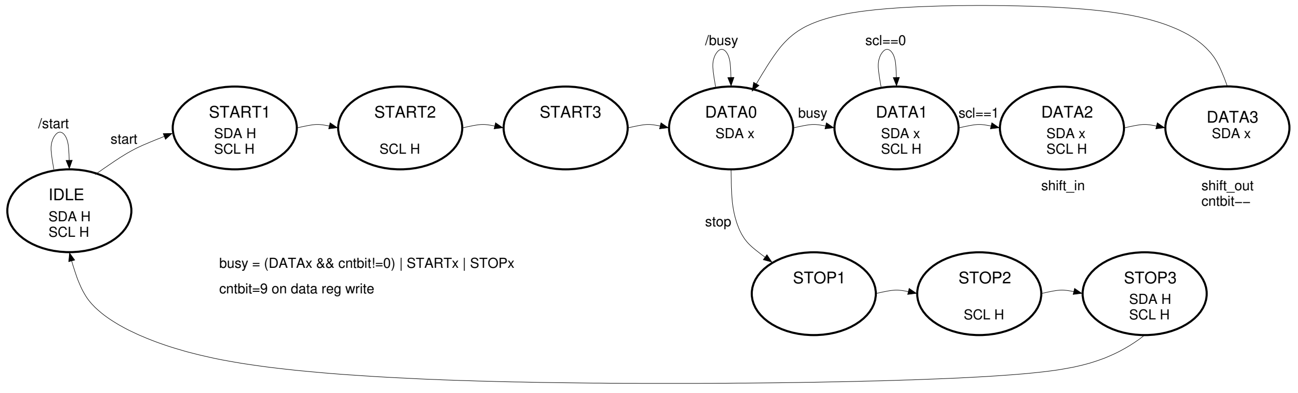
This peripheral acts as a master I2C controller and is byte oriented. It basically includes:

- A programmable clock divider that defines the time quantum of the controller (a single bit takes 4 quanta)
- A 9-bit shift register (8 bit for data and one extra bit for ACK) that is used for both reads and writes. An additional flip-flop is included for the sampling of SDA at its proper time.
- A bit counter that gets loaded with a value of 9 when data is written and gets decremented with each bit transmitted until it reaches zero. A busy flag is set when this counter is nonzero.
- A finite state machine controller (FSM).

The SCL and SDA signals of the I2C bus are bidirectional and therefore they were split into input and output signals in the controller. These 4 signals are then merged at the FPGA pins and their open-drain characteristic is also obtained thanks to the tri-state drivers of the I/O pins. In the following state diagram outputs are stated in uppercase (SCL, SDA) and inputs in lowercase (scl, sda). During the data phase of the frame "SDA" comes from the higher bit of the shift register and "sda" ends stored into the lower bit of the same register. "SDA" is forced high or low by the FSM during non-data states. On the other hand "SCL" is always driven by the FSM.

The I2C controller also provides storage for two control bits used to signal the start and stop of I2C frames, and includes a busy flag. A write only register, I2CDCTL, includes the data plus the ACK bits to be transmitted along with the START & STOP bits, while a read only register, I2CDSTA, contains the data plus ACK bits received and the busy flag.

The state diagram of the FSM is shown next (only outputs with high levels are stated):



The controller remains in the IDLE state until its I2CDCTL register is written with the START bit in one. This causes the transition of the FSM to the DATA0 state and the generation of an I2C START sequence (falling edge in SDA with SCL high). Then, the FSM stays in the DATA0 state until the I2CDCTL register is written again with a 9-bit data value or an STOP bit in one. A data write causes nine DATA0 to DATA3 transitions with the generation of 9 SCL pulses and the shifting of 9 bits, the last one being the ACK bit. For address or write bytes the ACK bit written to the I2CDCTL register has to be one, while for read bytes the data has to be 0xFF and the ACK bit 0 (except for the last byte of the frame). The busy flag is set to zero when the 9-bit shifting is completed. If the I2CDCTL register is then written with the STOP bit in one the FSM moves to the STOP states, generating an I2C STOP condition (rising edge in SDA with SCL high), and reaching the IDLE state.

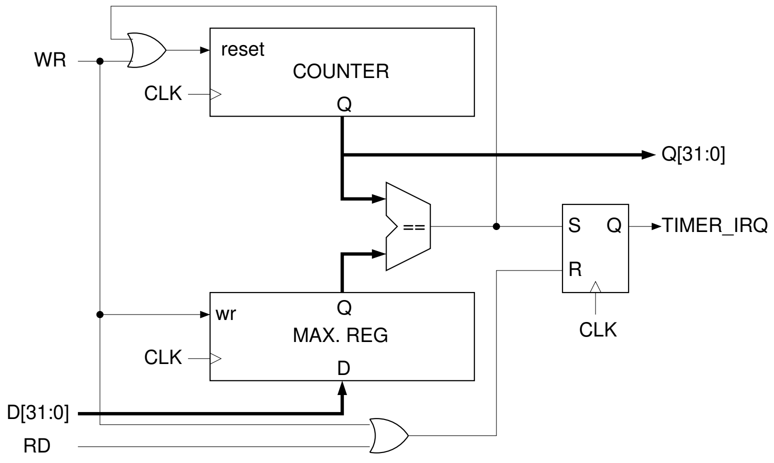
An usage example is shown next, where the "i2cframe" function uses the controller for the sending of a complete I2C frame, either read o write depending on the lower bit of "addr":

```
/**************************************************************************
      address   |      WRITE        |      READ
    -----------|-------------------|---------------
    0xE0000040 | I2C data/control  |  I2C data/status
    0xE0000044 | I2C divider       |

    I2C Data/Control: bit 10  bit 9  bit 8  bits 7-0
                        STOP   START   ACK     DATA
        STOP:  Send Stop sequence
        START: Send Start sequence
        ACK:   ACK bit. Must be 1 on writes and 0 on reads except last one
        DATA:  Data to write (Must be 0xFF on reads)
          - ACK and DATA are ignored if START or STOP are one
          - Do not set START and STOP simultaneously
          - Repeated START is not supported
          - Writing to this register sets the BUSY flag until the start,
            stop, or data, is sent

    I2C Data/Status:          bit 9  bit 8  bits 7-0
                               BUSY   ACK     DATA
        BUSY:  Controller busy if 1.
        ACK:   Received ACK bit (for writes)
        DATA:  Received data (for reads)

    I2C Divider: bits 6-0
        SCL frequency = Fcclk /(4*(DIVIDER+1))

**************************************************************************/

uint32_t i2cframe(uint32_t addr, uint8_t *buffer, uint32_t len)
{
    uint32_t i,j,ret=0;
    I2CDCTL = 0x200; // start
    while (I2CDSTA&0x200);
    I2CDCTL = 0x100+addr; // slave address, ACK High
    while ((i=I2CDSTA)&0x200);
    if ((i&0x100)==0) {
        if (addr&1) {   // Read frame
            for (j=len;j;j--) {
                I2CDCTL = (j==1)? 0x1FF : 0x0FF; //NACK or ACK reply
                while ((i=I2CDSTA)&0x200);
                *buffer++=i;
            }
        }else{  // Write frame
            for (j=len;j;j--) {
                I2CDCTL = 0x100+(*buffer++);
                while ((i=I2CDSTA)&0x200);
                if (i&0x100) {ret=-1; break;}
            }
        }
    } else ret=-1;
    I2CDCTL = 0x400; // stop
    while (I2CDSTA&0x200);
    return ret;
}
```

I hope this code is simple enough to understand. When I think about all the burden of other hardware I2C controllers, like NXP's, I can't get the Unicode 0x1F4A9 out of my mind. And yes, these controllers can also be used as I2C slaves, but they are just so complex that, at the end, people resort to bitbanging instead.

The I2C controller requires only 72 logic cells in an ICE40HX FPGA and has a maximum clock frequency of 175MHz.

4 Timer



The timer includes two registers: a counter that gets incremented with each clock pulse, and a maximum value register. When the counter matches the maximum value it gets reset to zero on the next clock pulse. This event can also request an interrupt to the core.

Any data written to the timer is stored in its maximum value register while the counter is also reset to zero. Reads returns the current value of the counter. Reads or writes to the timer clear the interrupt flag.

The timer uses 97 logic cells and runs with a maximum clock frequency of 94MHz.

5 CRC coprocessor

This peripheral is a hardware accelerator for the computation of CRCs, not an I/O device. It computes any kind of CRC one bit at a time, so, it takes 8 clock cycles for a byte. Its features also include:

- CRC polynomials up to 32 bits
- Normal or "reflected" CRCs
- Any initial value allowed

And, for the programmer's model it has the following registers:

Address offset	Write	Read
+0	CRC	CRC
+4	POLY	STAT
+8	DATA	CRC _reflected
+12	DATA_reflected	

The register CRC holds the current value of the CRC and it has to be written with its initial value before doing any CRC calculation. For some communication standards its initial value is zero, but there are cases where the initial value is nonzero. This register is 32 bit wide, and for CRC polynomials with less than 32 bits its contents have to be MSB aligned. As an example lets consider the initialization for the CRC16 of the X25 standard:

```
CRC32=0xFFFF0000; // Initial value (MSB justified)
```

In this case only 16 bits are going to be used and the lower 16 bits should be zero.

The POLY register contains the polynomial to use for computations and it also has to be MSB aligned. Following the same example, the CRC polynomial for X25 is $x^{16}+x^{12}+x^{5}+x^{0}$, and that means the POLY value is 0x1021 (bits #12, #5, and #0 as ones), but it has to be MSB aligned:

```
CRCPOLY=0x1021<<16; // Polynomial (MSB justified)
```

After CRC and POLY are initialized, any write to DATA, or DATA_reflected, registers will start the CRC processing. But these registers can be written as 8, 16, or 32-bit values, and therefore the number of clock cycles used depends on the width of the data. When the CRC is busy the bit #0 of the STAT register is zero, meaning we have to wait for the result. Writes to the DATA_reflected register results in the order of the bits being reversed (LSB being sent first). This is what happens in the X25 example:

```
CRCREFL32=0x125555ff;   // Little-endian bytes: 0xff, 0x55, 0x55, 0x12
while ((CRCSTAT&1)==0);
CRCREFL8=0x55;          // Single byte: 0x55
while ((CRCSTAT&1)==0);
```

In this example a 32-bit, little endian, data, is sent first to the DATA_reflected register. Then we wait until the data is processed, and next, a single byte more is also sent to the same register but using an 8-bit data width. The data width for these registers is defined using the following code (listing for DATA register only):

```
#define CRCDATA32 (*(volatile uint32_t*)0xE00000A8)
#define CRCDATA16 (*(volatile uint16_t*)0xE00000A8)
#define CRCDATA8  (*(volatile  uint8_t*)0xE00000A8)
```

Here, any value assigned to the CRCDATA32 symbol will generate an “store_word” instruction, while assignments to CRCDATA16 or CRCDATA8 will result in “store_halfword” or “store_byte” instructions respectively.

At the end the CRC result can be read from both the CRC and CRC_reflected registers. In the last case the result is LSB justified. This is the case for the X25 standard, where the resulting CRC must also have its bits inverted:

```
_printf("crc=0x%04x\n",CRCREFL16 ^ 0xFFFF);
```

Here the CRC_reflected register is read as a 16-bit halfword, and then its value is inverted.

The CRC coprocessor has the following interface:

```
module CRC (
    input  clk,      // System clock
    input  cs,       // Chip Select
    input  [1:0]rs,  // register select (address)
    input  [3:0]wrl, // Write Lanes (one per byte)
    input  [31:0]d,  // input data
    output [31:0]q   // output data
);
```

The synthesis of this module used 213 logic cells and had a maximum clock frequency of 183MHz in an ICE40HX FPGA.

6 CAN bus controller

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.

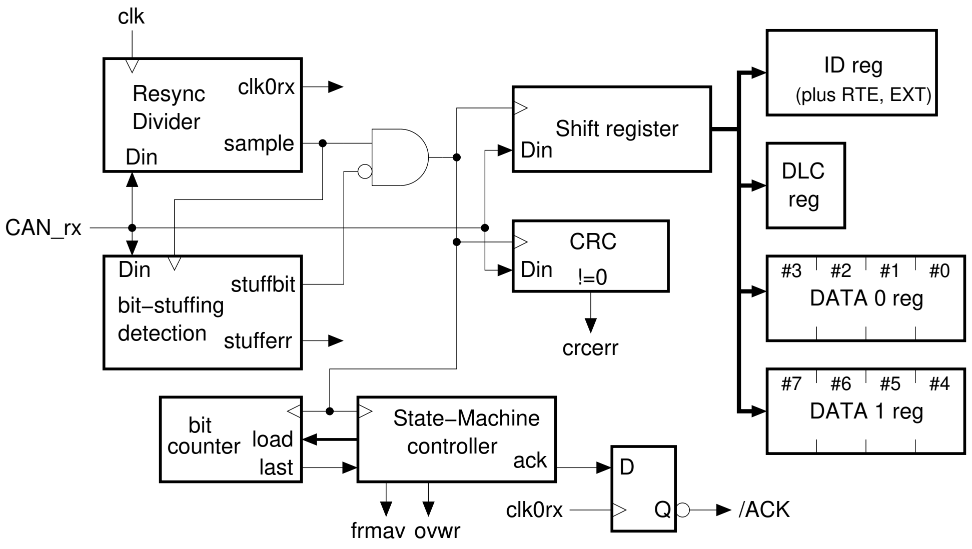
The CAN peripheral requires 641 logic cells in the ICE40HX FPGAs, this is about six times the complexity of the UART, and its maximum clock frequency is 82MHz. Internally it is divided into a separated receiver and transmitter, each of them with:

- A resynchronizable clock divider (10-bit downcounter).
- An state machine controller (8 states)

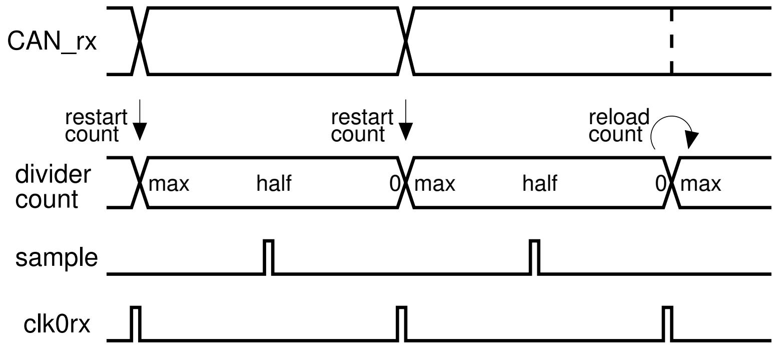
- A presetable bit counter (6-bit downcounter).
- A 5-bit shift register for the detection of stuffing bit conditions.
- Shift registers for ID, DLC, and DATA fields of the frame.
- A 15-bit CRC register.
- Flag management logic.

A description of the receiver and transmitter blocks follows.

6.1 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.

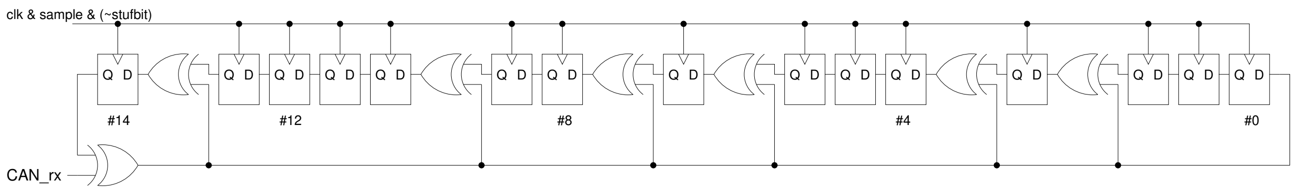


This circuit samples the CAN bus input using the fast system clock and compares the last two samples. If they are different the divider counter is loaded with its maximum value (from a register not shown in the figure) and starts downcounting. Two pulse outputs are generated, one when the counter reaches 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.

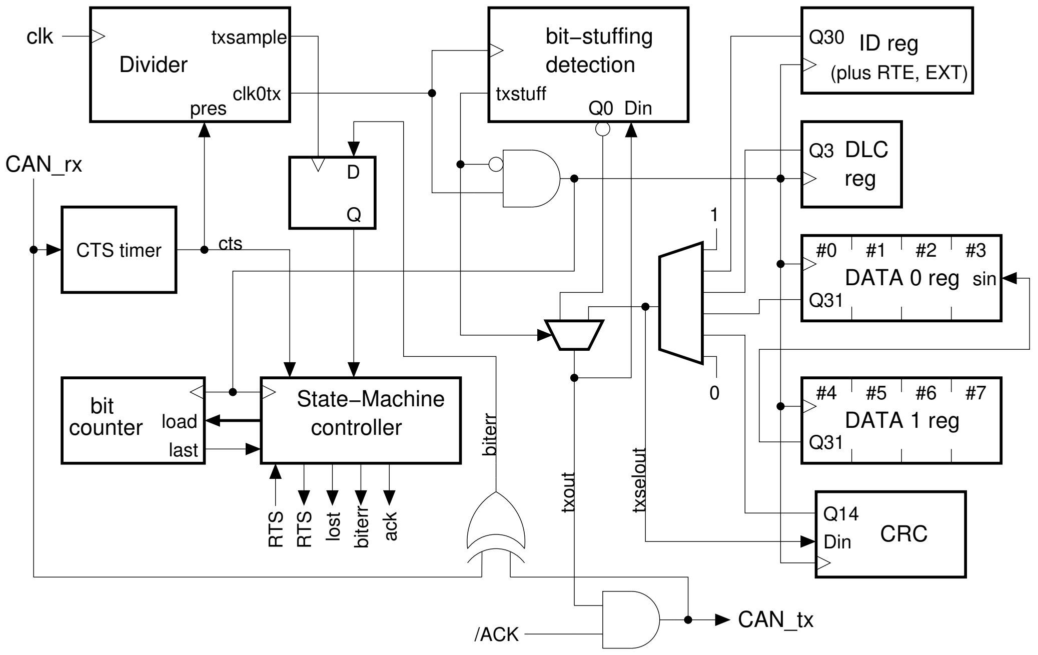
The shift register is a 21-bit, left shift register, that holds a temporary copy of the received data. Its partial contents are written into one of the destination registers at the proper time: 13 bits to the ID plus RTR/EXT flags register, followed by another 21 bits if an extended ID is received, 4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

6.2 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.

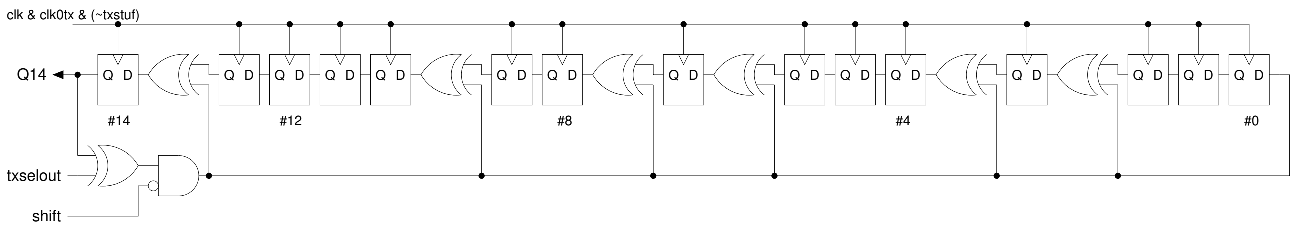
In the CAN bus any signal transmitted is also received and it supposed to have the same logic value as the data being transmitted, but these two values can be different if another node is transmitting a dominant bit while our bit is recessive. This can happen in three different cases:

- 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.

6.3 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.

6.4 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.

Data registers, DATA0 and DATA1, are 32-bit wide but they can also be addressed as bytes or halfwords, and for that cases they are little-endian. This means the first data byte of a CAN frame is that located in the lower 8 bits of the DATA0 register, and so on. The ID registers are accessible only as 32-bit words, and the DLCF register is better read and written as 16-bit words or bytes. 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
						WO									
-	-	-	-	-	-	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, write-only. The bit time is obtained multiplying the core clock period by (BAUD+1), or conversely, the data rate is: $bps = f_{CLK}/(BAUD+1)$. This field has to be addressed as a word or halfword, never as a byte (a halfword is a very convenient size).

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)

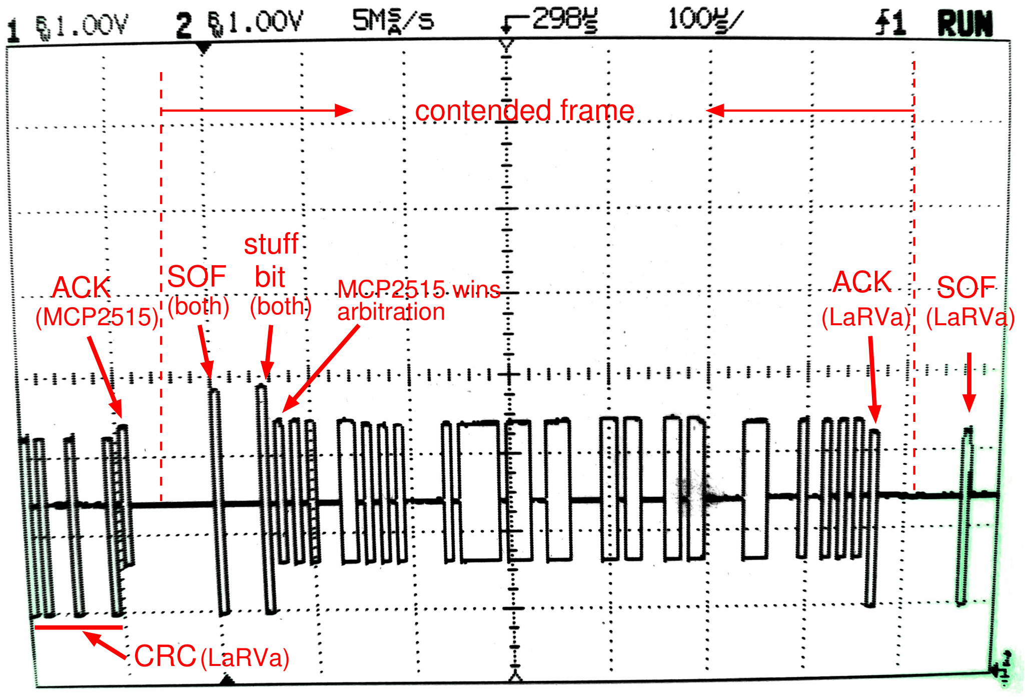
6.5 CAN bus arbitration

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.

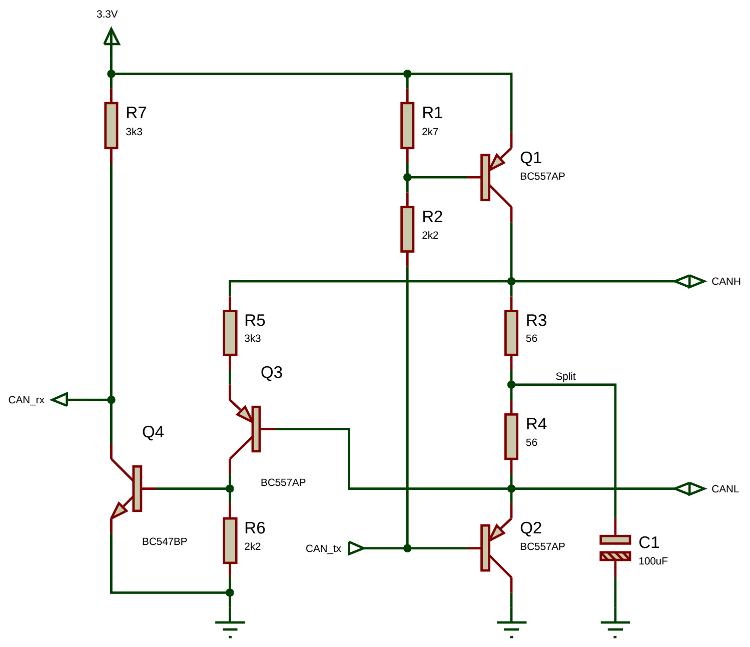
This behavior was hard to debug, but at the end I was able to capture one of such contentions in a scope screen:



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:


3.3V
R7
3k3
R1
2k7
Q1
BC557AP
R2
2k2
CANH
R5
3k3
R3
56
Q3
Split
CAN_rx
Q4
R4
56
BC557AP
CANL
R6
2k2
Q2
BC557AP
BC547BP
CAN_tx
C1
100uF