# CAN FD IP Core – future milestones

Ondrej Ille

25.11.2017

## Document purpose

This document is supposed to propose the future work on CAN FD IP Core before the publishing of the IP Core. The actual (November 2017) released version of the IP Core is v2.0 with last code modifications from January of 2017. No heavy modifications have been done since, only minor code refactoring and adding GPLv2 headers to VHDL code since then. This document will be rewritten into GitLab of FEE CTU after the publishment of the Core. All the references to „actual implementation“ or „actual version“ reffer to version 2.0

## Version 2.1 milestones

The main purpose of this release is to optimize the performance and resource usage of the IP Core and prepare the Core for implementation of DMA access into main CPU memory.

### RX Buffer optimization

#### Description

This modification involves changing architecture of the RX FIFO buffer. The RX FIFO buffer has actually parallel data interface from the IP Core (whole CAN frame is available in parallel). Once the frame is properly received (EOF field), the signal „rec\_valid“ starts loading the frame into the RX buffer. The frame is stored in following up to 20 clock cycles, 32-bit word per clock cycle. The other part of the RX Buffer has only one 32-bit word at a time available. This word corresponds to address of the „read\_pointer“ value. „read\_pointer“ is incremented by each read on Avalon Interface. Read of the frame is performed by repetitive read from the same address.

The aim of this modification is to create the same interface between RX Buffer and registers (Avalon) as between RX Buffer and CAN Core. This involves modifying the register map of the IP Core. Offset on the Avalon bus will be added to the read pointer and it will create direct address to the RAM memory of RX Buffer. Avalon Adress must be added combinationally to the read pointer value, to be able to get the data on the output of the RAM in the next clock cycle. The multiplexor must be created to drive the RAM address based on the Avalon address range. Actual implementation moves to the next word in the memory (increments read pointer) by performing the Avalon read. Additional bit must be added to register map. Writing logic 1 into this bit will increment the value of the pointer to point to the first word of next CAN frame. Thus user would be responsible for erasing the frame from RX buffer after reading it!

An optional inference of the RX Buffer via VHDL generic must be added during this step. This option will allow to either inferr or not inferr the RX buffer. In the case of the buffer presence the Avalon address will create the RAM address based on chosen register. In the absence of the RX buffer the Avalon address would create the offset in the parallel interface on the ouput of CAN Core. In case of Buffer absence the user is responsible for reading the frame soon enough before the Core erases it at the SOF of next frame!

#### Subtasks

1. RTL implementation
   1. Change the register map implementation and create the combinational multiplexor
   2. Add a bit for manual moving into next frame in RX buffer
   3. Create the addressing logic into RX buffer by adding the Avalon address to the „read\_pointer“
   4. Add the generic option into the CAN Core for inferrence of the RX buffer
   5. Add adressing logic for the case of RX buffer absence
   6. Implement the changes on CAN top level entity. Modify all instances of the CAN top level entity to use new generic for RX buffer inferrence.
2. Testing
   1. Modify the RX buffer unit test to be compatible with changes in the RTL implementation. This involves change in the CAN Test library function for reading the RX frame.
   2. Make sure that RX buffer unit test runs sucessfully.
   3. Make sure sanity test runs succesfully.
3. Driver changes
   1. Modify the CAN FD driver. Function for reading CAN frame must be modified.
4. Real word verification
   1. Synthesize the IP Core with the change. Verify timing performance is not significantly affected by the usage of new addressing logic. Verify that RAM is still inferred for the buffer, and the buffer is NOT synthesized out of FF!
5. Documentation
   1. Update the Lyx documentation with the latest change. This involves changing the register map in the documentation and changing the describtion of the RX buffer circuit. Also new RX buffer usage generic must be added into Core parameters.

### 8 and 16 bit access extension

#### Description

Actual implementation of the IP Core supports only full 32 bit accesses. The origin of the implementation is in the test platform where byte enable support was not needed. On the road to the full compatibility with the Avalon spec, byte enable signal must be added. Inactive bits of this signal will mask out the write data and not write the bytes which are forbidden for writing by byte-enable signal. Adding byte enable signal will add support for accessing the registers from uint8\_t and uint16\_t types in C. All side effects (like clearing interrupt vector by performing read) must be also masked out by byte enable signal.

#### Subtasks

1. RTL implementation
   1. Modify the Memory registers to support byte enable signal.
   2. Modify top-level entity to include the byte enable signal in the Avalon interface.
2. Testing
   1. Add support for byte enable into all tests which are instantiating the IP core. Update the test library (access functions). Run all the tests and make sure they are passing.
   2. Write overall memory registers test which would include the byte enable signal and verify that all side effects are properly masked out by the byte enable signal. There is not memory registers unit test existing at this time.
3. Driver change
   1. Optimize the driver for usage of native 8,16 bit accesses.
4. Documentation
   1. Modify the documentation of the IP Core to state there is also 8,16 bit access support.

### Sync/async reset

#### Description

Actual implementation of the IP Core implements all VHDL processes with asynchronous reset. Since the future intention is to use the IP Core in Xilinx based systém on AXI interface (instead of Avalon as now), Xilinx reccomendations must be kept for reset. Xilinx Coding guidlines reccomend to use the synchronous reset instead of asynchronous reset. The aim of this modification is to change the resets into synchronous. This change provides immunity against metastability possibly caused by violating the Reset recovery time by asynchronous reset.

#### Subtasks

1. RTL implementation
   1. Added reset synchroniser into the Core to synchronise the external reset, or re-implement all the processes to use either sync/async reset based on generic, or use sync reset all the time. First option is more suitable
2. Verification
   1. Reset is always assigned in test Framework for multiple clock cycles. This change does not necessarily need to be verified in RTL verification.
3. Synthesis
   1. Perform synthesis with the new reset. The actual synthesis of the Core has reset synchronisation implemented outside of the core in the top level entity. Thus no change should be seen in synthesis results.

### TX buffer reorganisation

#### Description

Re-implement the structure of the TX buffer. The actual implementation stores the TX frame in 3 stages of pipelines. 1-in registers, 2- in TX buffer(1 or 2) and in the CAN Core. The aim of this task is to remove the first part of the pipeline and create addressing logic directly into the TX buffer from Avalon address. Selection of the TX Buffer would be done by control bit in the memory registers. In the actual implementation the whole frame is moved into the TX buffer in one clock cycle. Since the user will directly access the buffer after this change, additional logic is needed to satisfy that half-written frame will not be loaded into CAN Core by TX Arbitrator. A „lock-bit“ can be added for this purpose into the memory registers, or txt\_empty\_reg can be used for this purpose and made visible in user registers.

#### Subtasks

1. RTL implementation
   1. Modify the memory registers module to support the new command bits (like lock bit, TX buffer (1 or2) where to write the frame etc…).
   2. Modify the top level implementation to support new interface into the register module (address and data, instead of the actual 640 parallel bits and store command)
   3. Modify the TX buffer implementation to support addressing in the buffer by the new interface. Modify the logic that signals to the TX arbitrator which frame has valid content to include the „lock-bit“.
2. Verification
   1. Rewrite the CAN test lib to suppor new way of loading the data directly into TX buffer.
   2. Update TXT buffer unit test. Make sure the test is passing and include support for lock bit. No half-written frame can be signalled to the CAN Core as valid!
   3. Make sure all tests (feature,sanity) are passing with the new implementation
3. Driver
   1. Modify the driver to support new method of TXT buffer access.
4. Synthesis
   1. Synthesize the buffer with the new version of the TXT buffer access and verify the drop in register usage by removing the first TX buffer pipeline!
5. Documentation
   1. Update register map in the documentation
   2. Update the TX buffer description.

### Retransmitt frame dropping

#### Description

The actual implementation rettransmitts the frame once the Core lost arbitration or error occured. The repetition of the re-transmission can be limited by the user. However if the retransmitt limit option is disabled the frame will re-transmitt forever and might possibly block this core for longer time. The aim of this task is to detect if higher priority frame is present in either of TX Frames before re-transmitting the frame. If yes, the actual frame should be dropped, and the higher priority frame should be loaded to CAN Core for transmission. The behaviour must be configurable from user-registers. Thus user can choose whether the frame will be dropped or re-transmitted in presence of higher priority frame in the buffer. The implementation involves comparison between the actual frame identifier in the CAN Core and the Identifiers in the TX buffers. This feature will be implemented in the TX Arbitrator circuit or Protocol Control FSM.

#### Subtasks

1. RTL implementation
   1. Add the support for the frame dropping into the CAN Core, TX arbitrator and user-registers. Implement the changes in CAN top level entity to have the support for new feature. The default behaviour is kept as in actual implementation.
   2. Add Status bit concering occurence of such a frame drop, or even counter of this occurences.
2. Verification
   1. Modify TX arbitator unit test to pass in the same manner as now, with the new feature
   2. Add support for this feature, either in TX arbitrator unit test or as feature in the feature tests. A clear case where higher priority frame will be stored after lower priority frame, bus value will be forced to invalid value, causing bit error, and verification that the higher priority frame was transmitted instead of lower priority frame. Verify the Status bit or occurence counter.
   3. Verify that all tests are passing with the new feature. Since the feature is disabled by default, the behaviour should bet he same as in actual version.
   4. Sanity test support for this feature is difficult, since sanity test is checking the reception of each frame which was stored for transmission. Such a frame drop would cause failure of sanity test. An option would be adding the read of dropp-counter (or status bit) and re-inserting the dropped frame from transmission). The order of the reception does not matter in sanity test, since it performs the look-up on each received frame in each memory!!
3. Driver
   1. Modify the driver to support the frame dropping feature.
4. Documentation
   1. Modify the documentation to be compliant with the new feature. Modify the register map and describe this feature.
5. Synthesis
   1. This feature should not affect synthesis in a significant manner.

### TX buffer into SRAM

#### Description

The actual implementation of the TXT buffer is using Flip-flops. In order to use SRAM memory on FPGA, the load of the data into the CAN core must be serialized, since FPGAs does not offer 640-bit width memories. This task involves creation of the FSM for serial loading the TX frame into CAN Core. This task must be implemented after the serialization of access from user registers and after adding the frame droppig feature and must be compliant with both of these features. As first, identifier must be loaded to the CAN Core, beacuse loading will také up to 20 clock cycles. If the identifier is not already stored in the CAN Core during the first bit of the Identifier transmission, Invalid value will be transmitted. This situation must be avoided.

#### Subtasks

1. Subtasks within this task still must be defined

### Bug-fix of the switching between data rates

#### Description

The actual implementation of the Prescaler does compenasation for the bit time duration during bit-rate switching. This compensation depends always on the data bit time with the assumption that the nominal time quanta is set to value higher than 4. This behaviour is not correct and should be changed to support all combinations of Nominal Time quanta and Data Time Quanta duration. This behaviour was not observed in the reference hardware testing since value of 8 was used for nominal bit time prescaler and thus the bit counter truly was not affected.

#### Subtasks

1. RTL Implementation
   1. Implement change in the Prescaler and perform the compensation of the bit time counter based on both time quanta values
2. Verification
   1. Make sure that sanity tests are passing with the new implementation
3. Synthesis
   1. Synthesize the IP Core with this change, verify the resource usage.
   2. Check the functionality of the bug-fix withing the CAN FD Tester from Kvaser! With old implementation and Nominal time quanta set to 1 any frame with bit-rate switch should cause error frames due to mismatch in the duration of bits where switching occurs. To trully get the error minimize the SJW, not to allow the resynchronisation compensate for this bug.

## Version 2.2 milestones

The main purpose of this release is to implement the DMA access for the CAN FD IP Core on new, Avalon master interface.

### Defining the DMA functionality

#### Description

Decide on particular features of DMA interface. The DMA interface will be present on new CAN Core interface – Avalon Master, which will by itself able to initiate transaction on Avalon SoC bus. This interface will be routed in the Avalon SoC in such a way that it will be able to reach the main systém memory (over interconnect component). The final step of this task should be clear register map and clear approaches on DMA behaviour from both SW and HW sides!

#### Subtasks

1. The DMA should be supported on RX Data (write transaction on Avalon master) as well as on TX Data (Read transaction on Avalon master).
2. The DMA support should be optional, allowing for synthesis of the Core with or without of the DMA support.
3. On RX Data the DMA will start the transaction once the data are available in the RX buffer or on the output of CAN Core in the case of RX buffer absence.
4. On TX Data the DMA will start the read transaction if any of the TX buffers is emptied due to propagation of the CAN frame into CAN Core for transmission or if it gets the command on Avalon Slave interface (from user).
5. The DMA will provide support for firing interrupt once the data are in the main memory. How are we going to satisfy this???
6. Even if synthesized the DMA engine can be turned off!
7. DMA engine can be possibly in conflict with Avalon Slave interface, since it can access the buffers by itself. What should be the behaviour in case of simultaneous access by user and by DMA engine? Is there a priority defined? Or should the transaction from user on Avalon Slave interface be aborted (returning something like memory access error in the Kernel)
8. What should bet he user interface to DMA. A proposal for using the ring buffer in SW is the most probable one. See following points
9. In RX direction: A pointer to physical address will be provided by the user and DMA engine will store the frame into this address and increment its own address pointer. It will notify the user by interrupt about this action. The user will read the latest frame from the ring buffer and increment its own pointer (in SW). This approach can play nicely with Linux DMA API and functions like: dma\_map\_single, where this interrupt based approach is used…
10. In TX direction: A pointer to physical address will be provided by the user. The user will store frame to this address, notify the DMA engine by a write into command register and increment its own pointer. The DMA engine will read the frame and store it for transmission. Another version would be that DMA engine would be polling periodically on the status of the SW ring buffer and once the SW pointer in the ring buffer would be incremented, it would automatically take the data from the main memory.
11. Decide on the support of the transaction types. Are we going to use single accesses? Are we going to use bursts? In case of burst a SoC where all the components do support the bursts must be used! If bursts will be used the CAN FD Tester PCIe SoC must be extended with burst functionality. It is assumed that the DMA support will be first developed on this SoC… How is it going to be with read bursts. Read performance of any kind of transfer is terrible unless it is a pipelined version of the Avalon interface (AXI has this feature in its architecture, since Read address and read data are separated channels)…

### Implement the DMA functionality in HW

#### Description

The RTL implementation of the DMA engine.

#### Subtasks

1. Implement the DMA module in the IP Core.
2. Modify the register map to be compliant with the DMA spec. from previous point.
3. Add the Avalon Master interface into the CAN Core.
4. Add support for interrupt generation in the interrupt manager on the finish of the RX Data DMA transfer.
5. Modify the documentation according to specification.

### Implementing the testbench for DMA

#### Description

The implementation of the testbench for DMA engine.

#### Subtasks

1. Implement the DMA test for the DMA engine. Is it possible to implement this test as part of feature tests? This feature test will contain the Ring buffers as the driver in SocketCAN would. The testbench would operate on both interfaces (Master and Slave) of the core and generate CAN frames into the TX ring buffer. On the other side it would read the data from the other ring buffer based on interrupts from the IP Core.
2. A variable latency should be included in this testbench to account for the latency of the real SoC where lot of data can be flowing…
3. Insert the test into the CAN Test Framework and automatize it. Together with the previous task, debug the DMA engine.

### Implement the Driver support for DMA

#### Description

A simple proof of concept of the DMA functionality in SW.

#### Subtasks

1. Embedd the Avalon master into SoC which on Avalon Slave interface can access the main systém memory (like the CAN FD PCIe tester beiing developer).
2. Since the Windows driver for CAN FD PCIe card will be used, the initial implementation can be done in this driver. The driver will be extended with the implementation of the DMA and it will create simple ring buffers for the DMA operation.

## Summary:

Only after all this stuff will be done, there will be possible integration of the CAN IP Core with the DMA into the SocketCAN Framework.