Western Norway
University of
Applied Sciences

Readout Unit DCS CAN Bus

High Level Protocol (HLP)

Simon Voigt Nesbø

February 22, 2019

CAN High Level Protocol

- Based on protocol used in STAR TOF:
  - J. Schambach et al, “CANbus protocol and applications for STAR TOF Control”, Journal of Physics: Conference Series, Volume 331, Part 2: Online Computing
- Standard CAN frames (11-bit ID)
- 8 MSBs of ID used for Node ID (each RU is a node)
- 3 LSBs of ID used HLP commands
  - WRITE, WRITE_RESPONSE, READ, READ_RESPONSE
- Acceptance filtering used in CAN Controller to filter messages based on Node ID only
- DCS initiates WRITE and READ commands, RU responds with WRITE_RESPONSE, READ_RESPONSE

	Command type			
Data byte	**WRITE (b″010″)**	**WRITE_RESPONSE (b″011″)**	**READ (b″100″)**	**READ_RESPONSE (b″101″)**
0	ADDR MSB	ADDR MSB	ADDR MSB	ADDR MSB
1	ADDR LSB	ADDR LSB	ADDR LSB	ADDR LSB
2	DATA MSB	DATA MSB	N/A	DATA MSB
3	DATA LSB	DATA LSB	N/A	DATA LSB

RU responds with data that was written in WRITE_RESPONSE

Firmware

CAN High Level Protocol Module – Branch/Repo

Repository: **RUv1_Test** on gitlab

https://gitlab.cern.ch/alice-its-wp10-firmware/RUv1_Test

Branch: **dcs_canbus** (will be merged to master eventually)

Code is located in **modules/dcs_canbus**

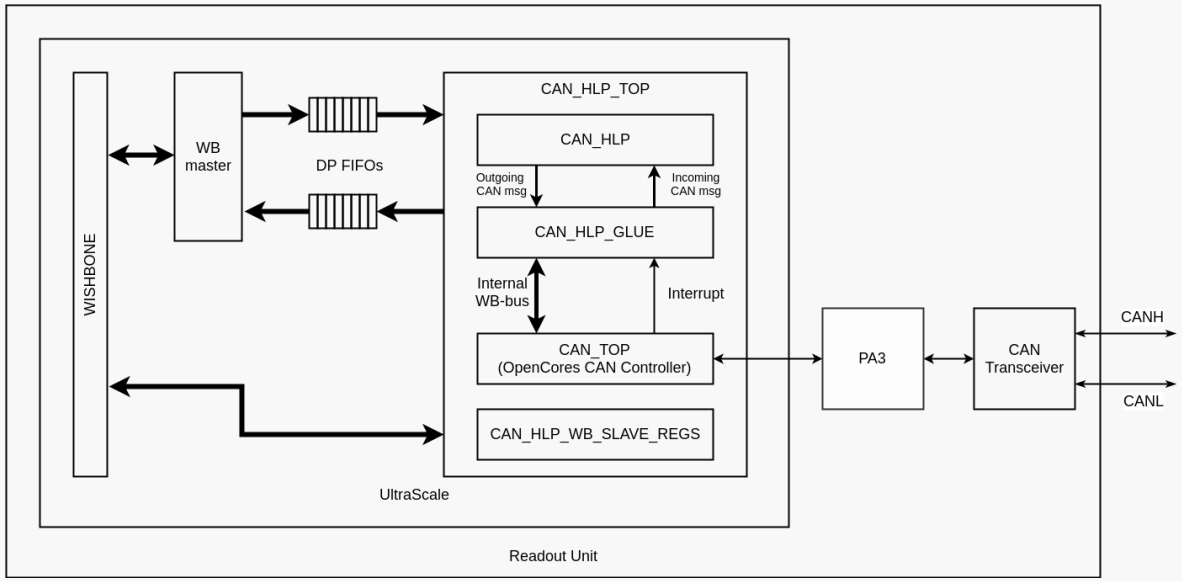
The CAN HLP module is a master on the Xilinx UltraScale's wishbone bus, and is able to read from and write to internal registers in the US firmware, using the custom CAN HLP protocol.

The CAN HLP module uses an instance of the OpenCores CAN Controller for the CAN bus communication, and includes the custom HLP protocol and FIFOs for communicating with the Wishbone bus in the Xilinx US firmware.

CAN High Level Protocol Firmware

Structure:

- CAN_HLP_TOP
  - CAN_HLP
  - CAN_HLP_GLUE
  - CAN_TOP
  - CAN_HLP_WB_SLAVE_REGS



CAN_HLP:

- WB master interface via DP FIFOs
- Reuse modules used for USB/GBT
- Interprets HLP format in CAN packages

CAN_HLP_GLUE:

- Glue logic between OpenCores CAN controller and CAN_HLP module
- Configures CAN controller on reset (bit rate etc.)
- Talks to CAN controller via dedicated WB bus

CAN High Level Protocol – Integration with RUv1_Test firmware

In **fabric_top.vhd:**

- The **can_hlp_top** module is created by the **can_hlp_top_tmr_wrapper** instance
  - **can_hlp_top_tmr_wrapper** wraps the **can_hlp_top** module to allow connection to triplicated WB signals/array
  - **can_hlp_top_tmr_wrapper** connects to an instance of **wishbone_master_tripleout** dedicated for CAN bus. The **wishbone_master_tripleout** module enables wishbone communication with **can_hlp_top** using FIFOs
  - **can_hlp_top_tmr_wrapper** does not actually triplicate the logic, just triplicates bus signals

- DIPSWITCH(7:0) is used for CAN HLP node ID
- PA3_IO_o(9) used for CAN_TX (was previously used for PA3 Wb2Fifo)
- PA3_IO_i(10) used for CAN_RX

CAN High Level Protocol – PA3 firmware

The CAN controller on the RUv1 board is not connected directly to the Xilinx US FPGA, it is connected to Microsemi PA3 FPGA

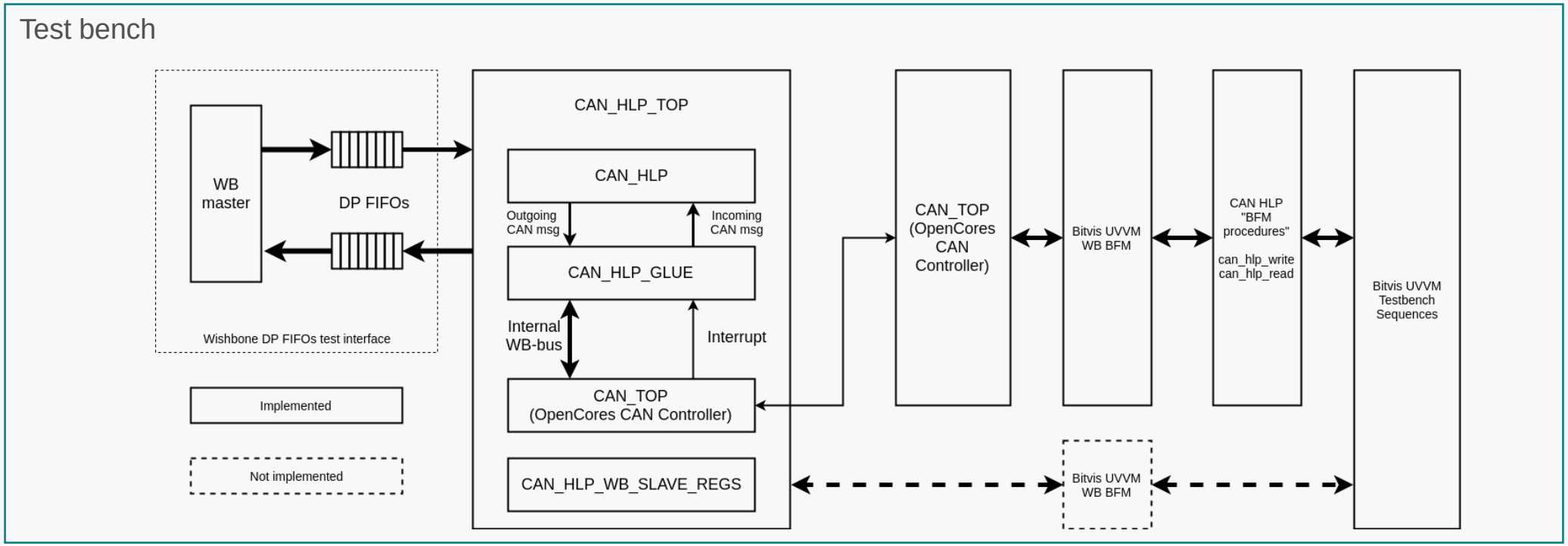
- CAN signals from/to US need to pass through PA3
  - Requires a firmware in PA3 where signals to/from US are connected to signals to/from CAN controller
- ~~This is implemented in **CAN_PASSTHROUGH** branch of PA3 firmware~~
  - ~~https://gitlab.cern.ch/alice-its-wp10-firmware/RUv1_auxFPGA/tree/CAN_PASSTHROUGH~~
  - ~~This will eventually be merged into **master** branch of PA3 firmware~~
- Passthrough of CAN signals is included in latest release of PA3 firmware (v0206).

CAN High Level Protocol Test bench

- Testbench using Bitvis UVVM library.
- Uses an instance of OpenCores CAN controller as «CAN BFM».

can_hlp_write/read() procedures to read/write to CAN_HLP with HLP protocol.

- Implemented using Bitvis WB BFM and OpenCores CAN controller



CAN High Level Protocol Test bench

The CAN HLP module testbench has not been integrated into the main testbench for the Xilinx at the time this is written, but there is a standalone testbench for the module.

- To run CAN HLP test bench:
- “**make environment**” from RUv1_Test top level directory
- “**do 07-compile_and_run_can_hlp.do**” from modules/dcs_canbus/sim directory in **vsim**

- Testbench code is in **can_hlp_uvvm_tb.vhd**, found in modules/dcs_canbus/source/bench/can_hlp

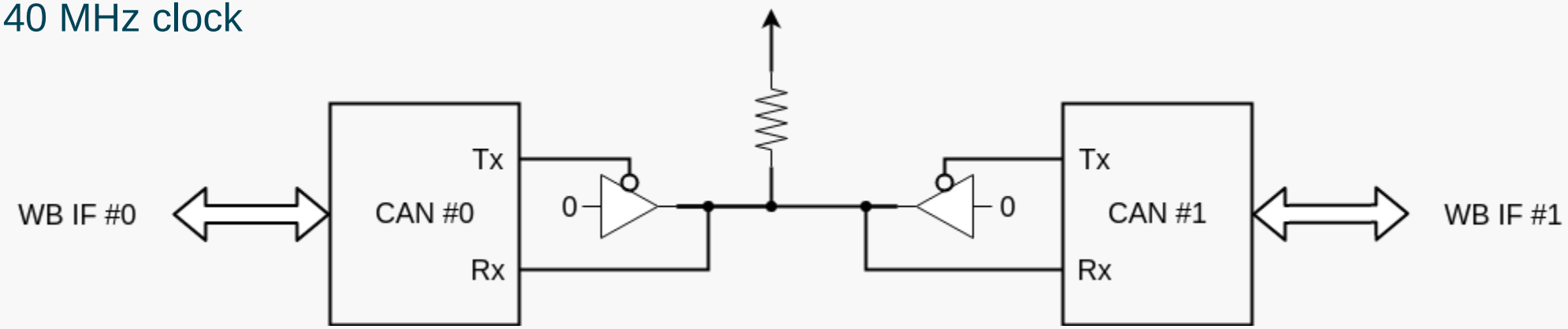
OpenCores CAN Controller

OpenCores CAN Protocol Controller

- https://opencores.org/project,can
- Written in Verilog
- Supports basic CAN and extended CAN
  - 11-bit ID and 29-bit ID
  - ID mask and filtering
- Up to 1Mbps operation
- Transmit/receive/error interrupts
- Wishbone interface (8-bit address and data)
- Register map compatible with Philips SJA1000 CAN Controller IC
  - https://www.nxp.com/docs/en/data-sheet/SJA1000.pdf
- Size: 12k gates (930 flip-flops)
- Sources located in modules/dcs_canbus/source/rtl/can_controller

OpenCores CAN Protocol Controller Test bench

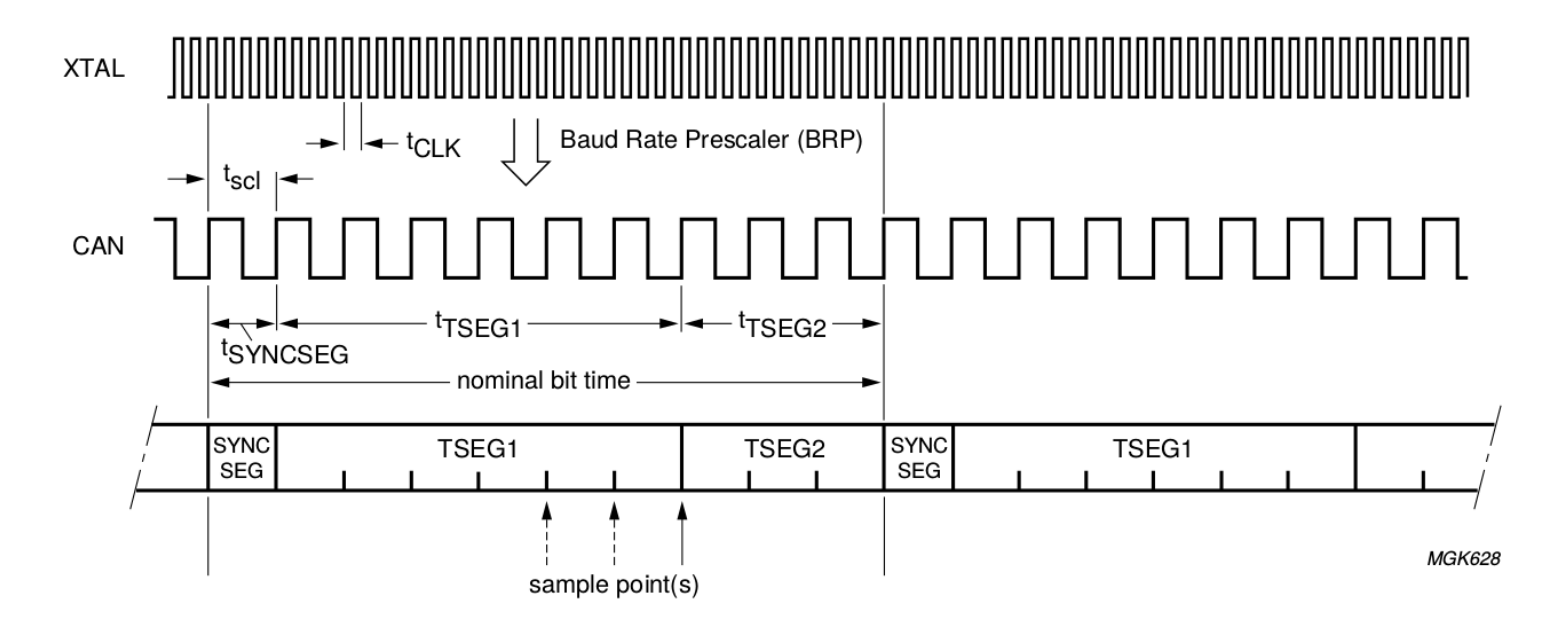
- Simple Bitvis UVVM style test bench (in VHDL)
- Testbench code located in modules/dcs_canbus/source/bench/can_controller
- Testbench scripts located in modules/dcs_canbus/sim
- Testbench consists of two instances of the CAN controller connected together
- Two separate WB interfaces to write to each controller
- 40 MHz clock



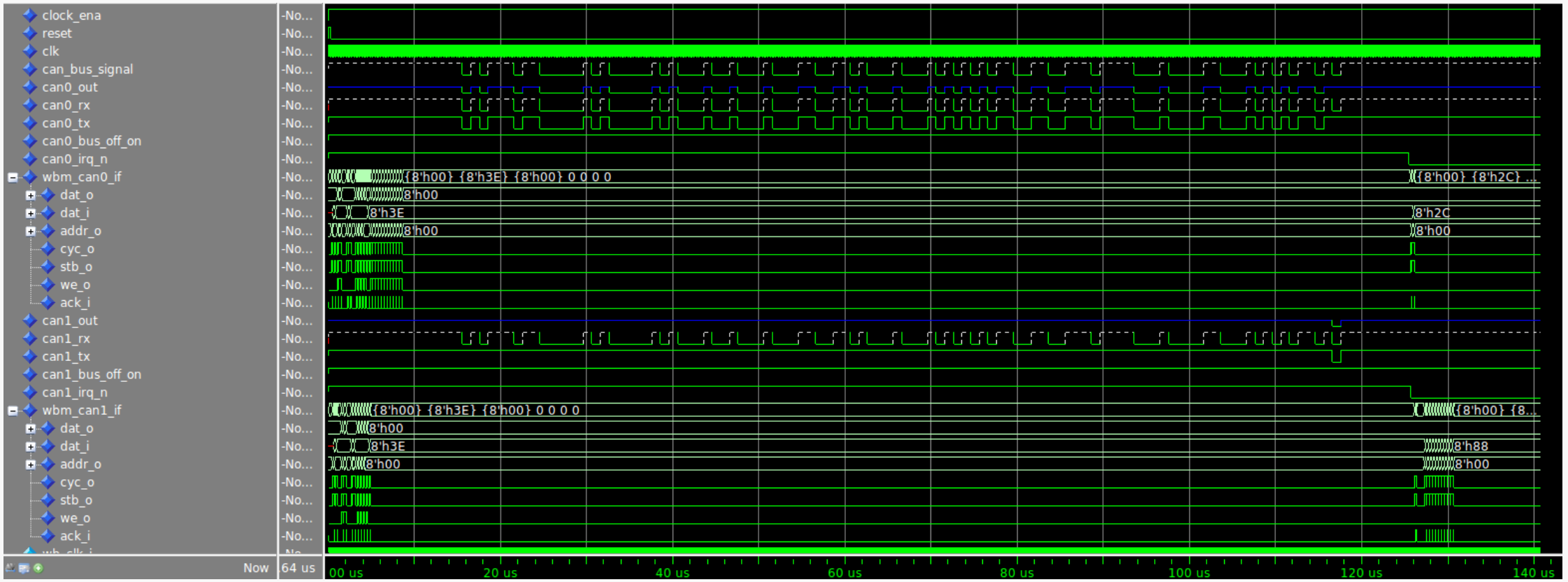
- To run simulation: run "**07-compile_and_run_can_ctrl.do**" from modules/dcs_canbus/sim directory in **vsim**

OpenCores CAN Protocol Controller Test bench

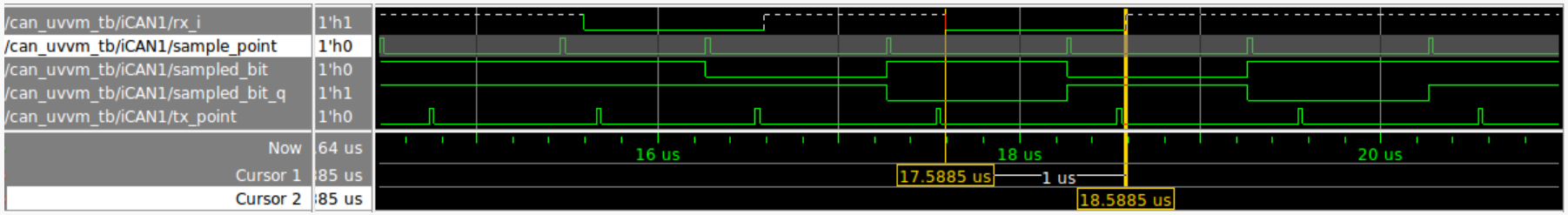
- Bit Timing Register (BTR0) configured for 4x baud clock prescale
- t_{SEG1} set to 7 baud clocks, t_{SEG2} set to 3 baud clocks, in BTR1
- $t_{SYNCSEG}$ is always 1 baud clock
- Gives us a bit rate of 40 MHz / (4 * (1+7+3)) = 1Mbps



OpenCores CAN Protocol Controller Test bench



OpenCores CAN Protocol Controller Test bench



OpenCores CAN Protocol Controller Test bench

Testbench log

Set up TX buffer on CAN0 with message for CAN1 (ID 0xBB)

Wait for CAN1 to receive message

Verify message contents

```
# UVVM: ID_LOG_HDR                     4940.0 ns  TB seq.          Start a transaction from CAN0 to CAN1
# UVVM: ------------------------------------------------------------------------------------------------------------------------------------------------------
# UVVM: ID_BFM                         5115.0 ns  TB seq.          wb_write(A:x"0A", x"BB") completed. Set TXID1 to xBB
# UVVM: ID_BFM                         5465.0 ns  TB seq.          wb_write(A:x"0B", x"08") completed. Set TXID2 to x08, 8 bytes data
# UVVM: ID_BFM                         5815.0 ns  TB seq.          wb_write(A:x"0C", x"11") completed. Set data1 to x11
# UVVM: ID_BFM                         6165.0 ns  TB seq.          wb_write(A:x"0D", x"22") completed. Set data2 to x22
# UVVM: ID_BFM                         6515.0 ns  TB seq.          wb_write(A:x"0E", x"33") completed. Set data3 to x33
# UVVM: ID_BFM                         6865.0 ns  TB seq.          wb_write(A:x"0F", x"44") completed. Set data4 to x44
# UVVM: ID_BFM                         7215.0 ns  TB seq.          wb_write(A:x"10", x"55") completed. Set data5 to x55
# UVVM: ID_BFM                         7565.0 ns  TB seq.          wb_write(A:x"11", x"66") completed. Set data6 to x66
# UVVM: ID_BFM                         7915.0 ns  TB seq.          wb_write(A:x"12", x"77") completed. Set data7 to x77
# UVVM: ID_BFM                         8265.0 ns  TB seq.          wb_write(A:x"13", x"88") completed. Set data8 to x88
# UVVM: ID_BFM                         8615.0 ns  TB seq.          wb_write(A:x"01", x"01") completed. Request transmission on CAN0
# UVVM:
# UVVM:
# UVVM: ID_LOG_HDR                     8615.0 ns  TB seq.          Wait for CAN1 to receive message
# UVVM: ------------------------------------------------------------------------------------------------------------------------------------------------------
# UVVM:
# UVVM:
# UVVM: ID_LOG_HDR                   125613.5 ns  TB seq.          Got interrupt from CAN1.
# UVVM: ------------------------------------------------------------------------------------------------------------------------------------------------------
# UVVM: ID_BFM                       125790.0 ns  TB seq.          wb_check(A:x"00", x"XX")=> OK, received data = x"3E". Check that CAN0 transmit interrupt was set
# UVVM: ID_BFM                       126140.0 ns  TB seq.          wb_check(A:x"02", x"XX")=> OK, received data = x"2C". Check that CAN0 transmit complete status bit is set
# UVVM: ID_BFM                       126315.0 ns  TB seq.          wb_check(A:x"00", x"XX")=> OK, received data = x"3E". Check that CAN1 receive interrupt was set
# UVVM:
# UVVM:
# UVVM: ID_LOG_HDR                   127315.0 ns  TB seq.          Verify message received by CAN1
# UVVM: ------------------------------------------------------------------------------------------------------------------------------------------------------
# UVVM: ID_BFM                       127490.0 ns  TB seq.          wb_check(A:x"14", x"BB")=> OK, received data = x"BB". Verify received RXID1
# UVVM: ID_BFM                       127840.0 ns  TB seq.          wb_check(A:x"15", x"08")=> OK, received data = x"8". Verify received RXID2, 8 bytes data
# UVVM: ID_BFM                       128190.0 ns  TB seq.          wb_check(A:x"16", x"11")=> OK, received data = x"11". Verify received data byte 1
# UVVM: ID_BFM                       128540.0 ns  TB seq.          wb_check(A:x"17", x"22")=> OK, received data = x"22". Verify received data byte 2
# UVVM: ID_BFM                       128890.0 ns  TB seq.          wb_check(A:x"18", x"33")=> OK, received data = x"33". Verify received data byte 3
# UVVM: ID_BFM                       129240.0 ns  TB seq.          wb_check(A:x"19", x"44")=> OK, received data = x"44". Verify received data byte 4
# UVVM: ID_BFM                       129590.0 ns  TB seq.          wb_check(A:x"1A", x"55")=> OK, received data = x"55". Verify received data byte 5
# UVVM: ID_BFM                       129940.0 ns  TB seq.          wb_check(A:x"1B", x"66")=> OK, received data = x"66". Verify received data byte 6
# UVVM: ID_BFM                       130290.0 ns  TB seq.          wb_check(A:x"1C", x"77")=> OK, received data = x"77". Verify received data byte 7
# UVVM: ID_BFM                       130640.0 ns  TB seq.          wb_check(A:x"1D", x"88")=> OK, received data = x"88". Verify received data byte 8
```

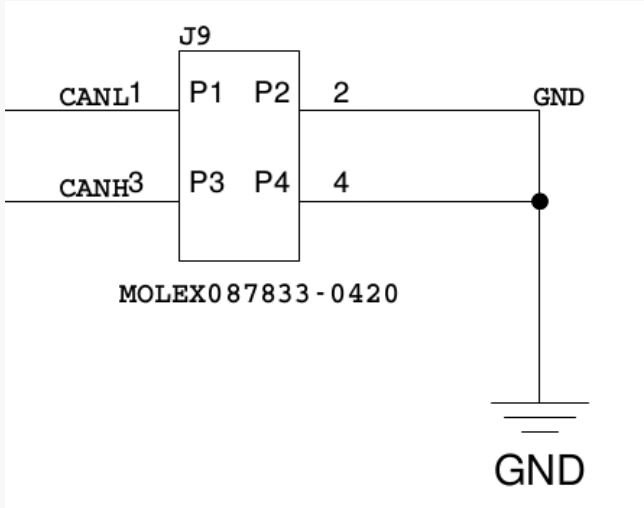
CAN HLP Hardware Setup

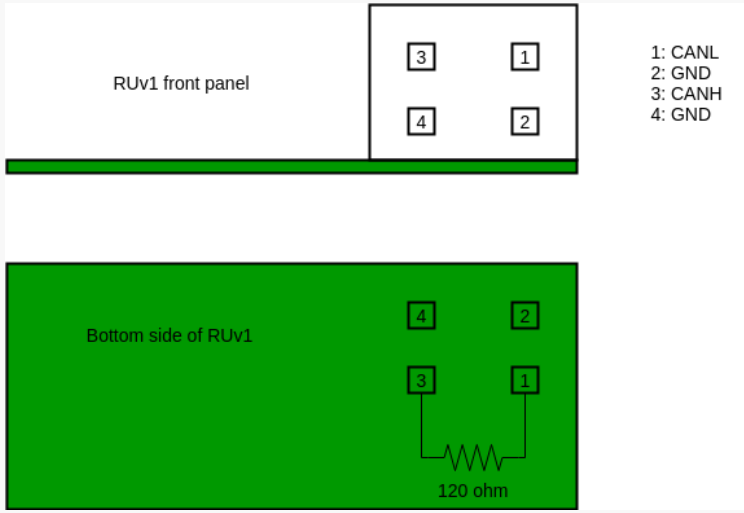
CAN High Level Protocol – Hardware Setup

- Connector J9 on RUv1 is used for CAN bus
- Located on bottom right side of front panel

- CAN bus requires 120 ohm termination at the end of the bus
  - It should not be terminated at each node

- Termination resistor can be soldered to the last RU on the bus
  - There is enough space to add a 1/4W through hole resistor to pins 1 and 3 on the backside of the RU, as indicated on the picture





CAN High Level Protocol – Hardware Setup

CAN bus connector on RUv1 board (MOLEX 087833-0420):

- https://www.digikey.ch/product-detail/en/molex-connector-corporation/87833-0420/WM18858-ND/718131?cur=CHF&lang=en

Housing and precrimped leads to mate with RUv1 CAN bus connector:

- https://www.digikey.ch/product-detail/en/molex-llc/0503948051-12-B6/0503948051-12-B6-ND/6047435
- https://www.digikey.ch/product-detail/en/molex-llc/0511100450/WM18030-ND/267654

Actual crimp contacts:

- https://www.digikey.ch/product-detail/en/molex-llc/0503948051/WM1128TR-ND/467800

CAN High Level Protocol – Hardware Setup

CAN HLP node ID:

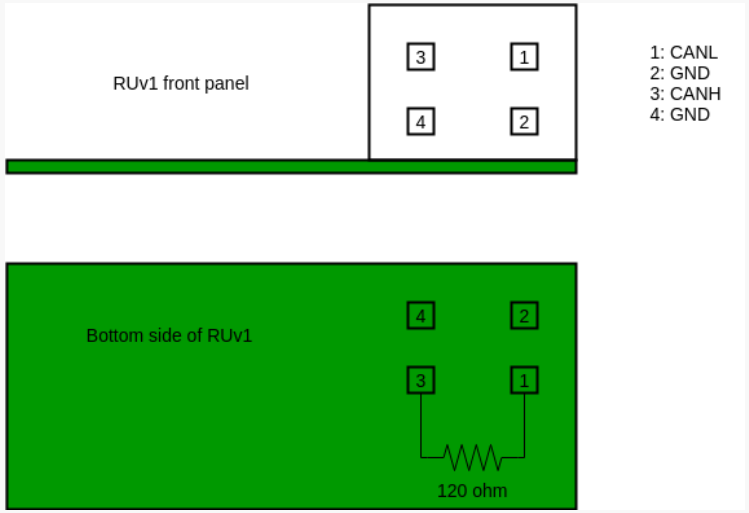
- The node ID is configured with DIPSWITCH[7:0] in the firmware
- At the time of this writing, the DIPSWITCH assignments are mirrored in the firmware compared to the hardware
  - This means that currently switch 10 to 3 on the board is used for ID, where switch 10 is the LSB of the ID.

- The DIPSWITCH is at the left side of the RUv1 board, when seen from the front panel
  - It is marked S8

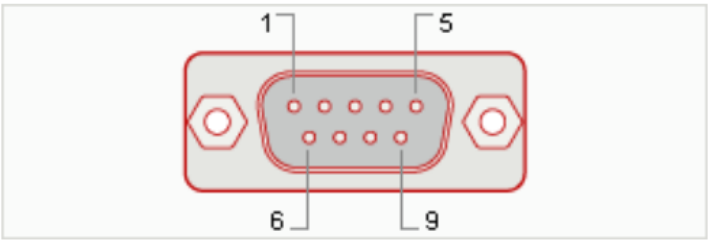
CAN High Level Protocol – PEAK CAN-USB Hardware Setup

CAN HLP has been tested with PEAK CAN to USB adapter

- PEAK CAN to USB adapter needs internal solder links to enable termination
  - Refer to user manual for instructions



Pin assignment D-Sub



Pin	Pin assignment
1	Not connected / optional +5V
2	CAN-L
3	GND
4	Not connected
5	Not connected
6	GND
7	CAN-H
8	Not connected
9	Not connected / optional +5V

CAN HLP Software

Python scripts

Software located in modules/dcs_canbus/software/

- can_hlp.py
  - Contains **CanHlp** class
    - Has a readHLP() and writeHLP() function for reading/writing registers using HLP protocol
    - Requires python-can package and python 3
- can_hlp_test.py
  - Tests HLP by reading/writing some registers
- can_hlp_test_loop.py
  - Reads/writes registers in a loop until interrupted with CTRL+C.

PEAK CAN bus Adapter Setup

- Set up SocketCAN interface for CAN/USB adapter:
  - modules/dcs_canbus/software/can_hlp/setup_socketcan.sh

- Must be run with sudo
- Modprobes kernel modules for canbus
- Sets up can0 SocketCAN interface for PEAK adapter

AnaGate CAN Controller Setup

Connect the AnaGate controller and set up ethernet on the computer so that they are on the same network. The AnaGate controller is configured with the IP address 192.168.1.254 by default, so the following settings for ethernet can be used on the computer:

- IP: 192.168.1.2
- Mask: 255.255.255.0
- Gateway: 0.0.0.0

Scripts are located in modules/dcs_canbus/software/can_hlp/SocketCANGateway/

Run the script to set up the virtual CAN bus interface:

- SocketCANGateway/InitVCAN.sh

Next run the bridge (located under x86_64_Release or x86_Release):

- SocketCANGateway vcan0 --baudrate=250000 --termination=1 --highspeed=1 --canport=0

That will configure the AnaGate controller to use the vcan0 interface, run at 250kbit, use internal termination, highspeed CAN, and use the first CAN port on the AnaGate (CAN A port on AnaGate Quattro).

Running test script

After a CAN controller has been configured to use SocketCAN, the can_hlp_test.py and can_hlp_test_loop.py scripts can be run to test the interface.

Running can_hlp_test.py will:

- Read out git hash (address 0x0100 and 0x0101 for git hash LSB/MSB)
- Read out counter registers in CAN HLP module
- Test writing to test register in CAN HLP module

Note:

- The scripts needs to be edited to use the SocketCAN interface configured for the CAN controller (typically can0 for PEAK adapter, and vcan0 for AnaGate controller)
- The scripts send commands to node id 1 by default. The RU needs to have the DIP switches configured for ID 1, or the scripts needs to be modified reflect the position of the DIP switches on the RU.

can_hlp_test.py output

candump log:

```
  IF   ID    SIZE    DATA
can0   035    [4]   12 05 00 00
can0   034    [2]   12 06
can0   035    [4]   12 06 00 08
can0   034    [2]   01 00
can0   035    [4]   01 00 E7 CF
can0   034    [2]   01 01
can0   035    [4]   01 01 0A 52
can0   032    [4]   12 06 00 08
can0   033    [4]   12 06 00 08
can0   034    [2]   12 00
can0   035    [4]   12 00 06 5E
can0   034    [2]   12 01
can0   035    [4]   12 01 05 B6
can0   034    [2]   12 02
can0   035    [4]   12 02 05 6A
can0   034    [2]   12 03
can0   035    [4]   12 03 00 4E
can0   034    [2]   12 04
can0   035    [4]   12 04 00 00
can0   034    [2]   12 05
can0   035    [4]   12 05 00 00
can0   034    [2]   12 06
can0   035    [4]   12 06 00 08
```

can_hlp.py log:

```
simon@simon-ThinkPad-T450s:~/Code/Python/CAN_HLP$ python3 can_hlp.py
Read response received.
Read response received.
githash: a52e7cf
Write response received.
Read response received.
Read response received.
Read response received.
Read response received.
Read response received.
Read response received.
Read response received.
Read response received.
can_reg_rx_msg_count: 1640
can_reg_tx_msg_count: 1472
can_reg_read_count: 1395
can_reg_write_count: 79
can_reg_status_alert_count: 0
can_reg_unknown_count: 0
can_reg_test_count: 8
exiting SAFELY
```

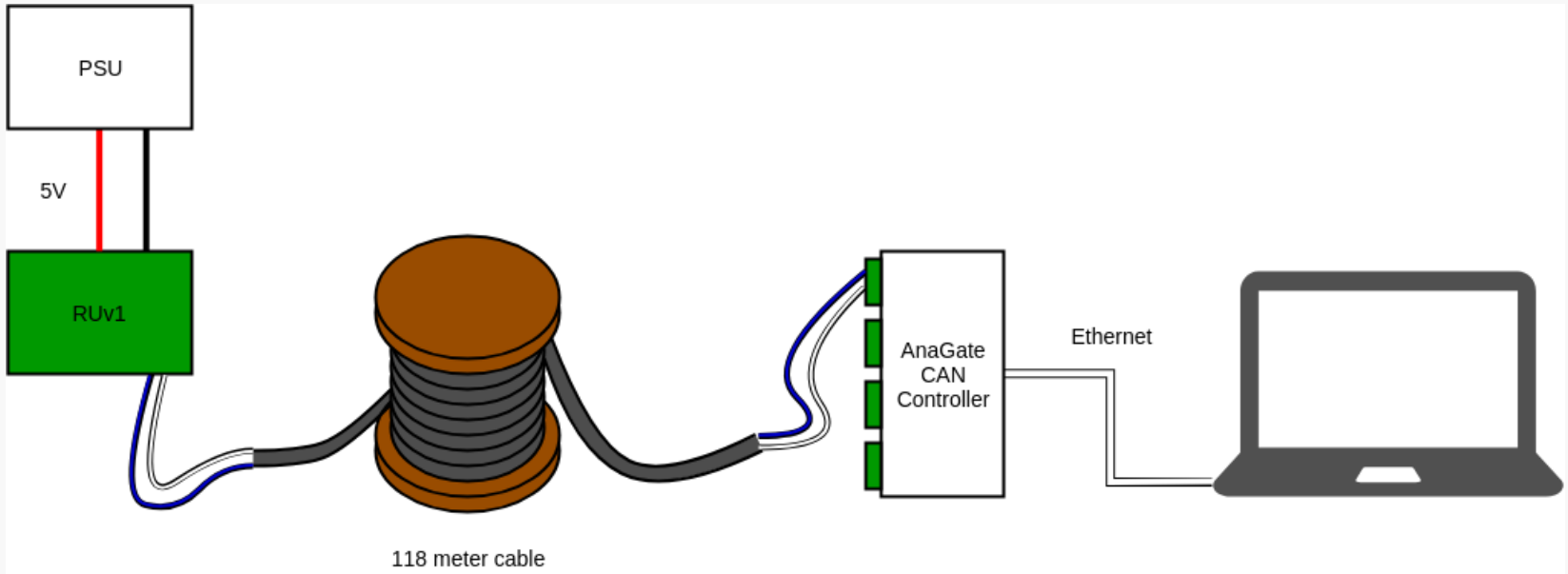
commit a52e7cf9801c9df01dd737a2faf599e0f3855790
Author: Simon Voigt Nesbo <svn@hib.no>
Date: Mon May 28 19:42:42 2018 +0200

[dcs_canbus] SVN bugfixes to CAN BFM, and random message generation in CAN BFM tb.

Testing with 118 meter cable

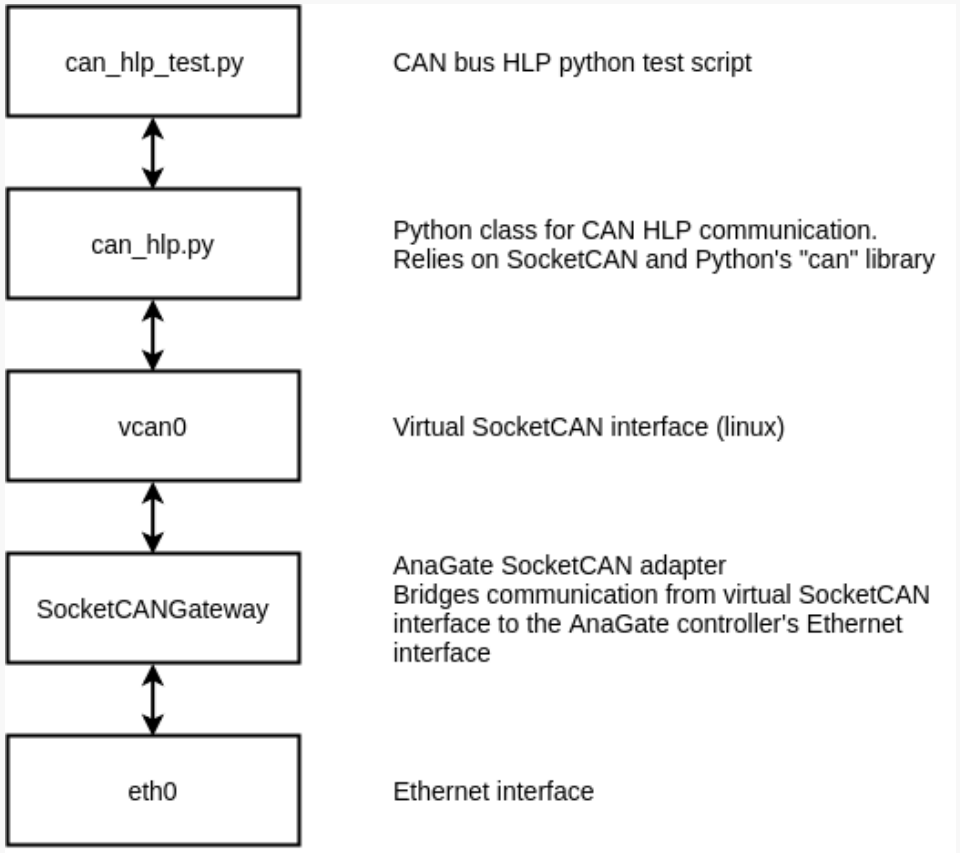
Test setup - hardware

- The test setup looks roughly as shown in the figure. Test scripts running on the laptop communicate with the AnaGate controller via Ethernet, which is connected to the RUv1 with the cable.
- The cable has two pair of wires, a white/blue and a white/orange pair. The blue/white pair (pair number 1) in the cable was used. This pair should have 120 ohm impedance, and is intended to be used for the CAN bus data transmission.



Test setup - software

› From a software point of view, the test setup is roughly as indicated in the figure.



A test in the script was comprised of:
Read out the githash registers from the Xilinx, and verify that the values read were as expected (2 CAN HLP read transactions)
Write a random value to a test register (1 CAN HLP write transaction)
Read back the value from the test register and verify that it was correctly written and read back (1 CAN HLP read transaction)

Each CAN HLP transaction consists of 2 CAN bus frames, the command from the PC, and the response from the RUv1.

Test results

- 3 different bit rates were tested for CAN bus:
  - 1 Mbit
  - 500 kbit
  - 250 kbit

1 Mbit and 500 kbit

- The 1 Mbit and 500 kbit did not work reliably. The first transaction succeeded in some cases, but the data returned was not as expected.

250 kbit

- The 250 kbit bit rate appeared to work reliably. A test loop script was run for around 18 hours without errors.

Test results 250 kbit

The test loop script ran for about 18 hours in total at 250 kbit, without any errors.

- 5775279 tests
- 17325835 CAN HLP read transactions
- 5775278 CAN HLP write transactions

- 82.89 tests per second on average
- 331.57 CAN HLP transactions per second on average (3.02 ms per CAN HLP transaction on average)

Test results – eye diagrams

- Some eye diagrams were measured for the CAN bus using Agilent DSO9254A oscilloscope with Agilent InfiniiMax differential probe.
- Initially this was performed at each of the 3 bit rates, with the AnaGate transmitting and the RUv1 powered off.
- The CAN controller on the RUv1 does presumably have high-impedance inputs, and the CAN bus lines were terminated with 120 ohm resistance at each end (internal in the AnaGate controller, external resistor on RUv1). So from an electrical point of view it should not make a difference on the signal whether the RUv1 is powered or not.
- Measurements were also performed at 250 kbit with the RUv1 responding, but this did not yield very good eye diagrams since the CAN voltage levels from the RUv1 were not exactly the same as from the AnaGate controller, and way to trigger on either the RUv1 or the AnaGate could not be found.

Test results – eye diagrams

See next couple of slides for eye diagrams etc.

- Eye diagrams actually don’t look that bad at 500 kbit and 1 Mbit
  - Perhaps the firmware could run at those baud rates?
  - Changing bit timing settings in firmware may help?

- Signal level from RUv1 lower than from AnaGate
  - Around 1.75 Vpp out from RUv1
  - Around 2.75 Vpp out from AnaGate

Test results – eye diagrams @ 1 Mbit – RUv1 not powered

Measured at RUv1 side

Test results – eye diagrams @ 500 kbit – RUv1 not powered

Measured at RUv1 side

Test results – eye diagrams @ 250 kbit – RUv1 not powered

Measured at RUv1 side

Test results – eye diagrams @ 250 kbit – RUv1 powered

Measured at RUv1 side

- Could not find a way to trigger only on the signal from either of the RUv1 or AnaGate was, so eye diagram looks messy
- Notice different signal levels. Output from RUv1 is a bit lower than from AnaGate

Test results – eye diagrams @ 250 kbit – RUv1 powered

Measured at AnaGate side

- Could not find a way to trigger only on the signal from either of the RUv1 or AnaGate was, so eye diagram looks messy
- Notice different signal levels. Output from RUv1 is a bit lower than from AnaGate
- Signal from RUv1 attenuated quite a bit when it reaches AnaGate (compare to previous slide)

Test results – eye diagrams

Eye diagrams measured with Agilent DSO9254A oscilloscope with Agilent InfiniiMax differential probe.