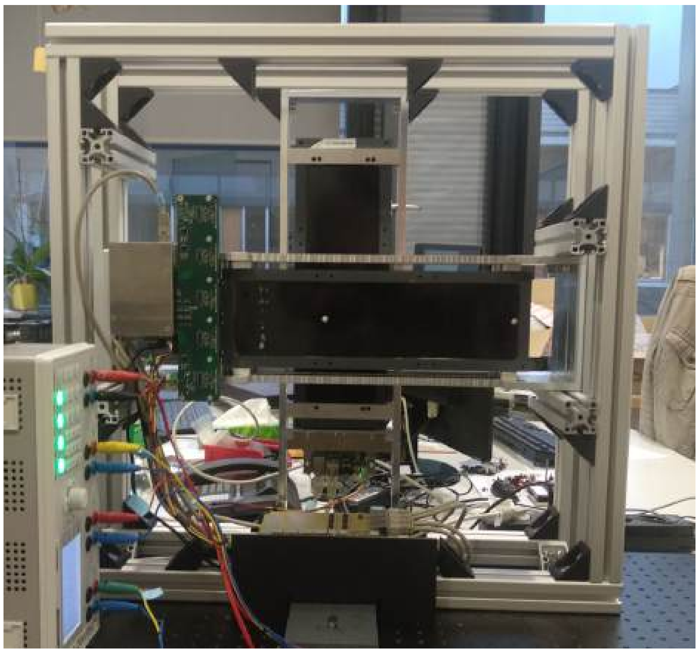
Beam Position Monitor

A bundle of random information



Author(s):	Michał Dziewiecki
Document revision:	1.0 (issue 1)
Hardware version:	1+2, mainly 2
Date:	2019-12-12

Revision history**:**

Document version	**Description**
1.0 (issue 1)	Primary issue. Don’t expect that you find answers for your questions! Don’t even expect that everything is true!

CONTENTS:

1. Introduction

1.1. General

This document is a description of the electronics hardware design and software for the Beam Position Monitor project. It covers basic information about the schematics, all firmware and PC software. It does not include mechanical documentation.

It focuses on version 2 of the DAQ system (FPGA-based). Version 1 (Microcontroller-based) is mentioned where it's needed to describe system interfacing.

1.2. Project directory description

A simplified directory structure of the project is shown below (many second-importance directories are omitted).

In this manual, short names (below in red) will be used. The majority of relevant files are placed in `[v2]`, while matlab post-processing scripts (common for both versions) can be found in `[v1]`.

```
+---application
+---beamplanner
+---DAQ2017                          [v1]
|   +---code
|   +---matlab_offline               [matlab]
|   +---matlab_online
|   +---pcb
|   +---reports
|   +---root
|   +---soft
|   +---step
+---documents
+---patent
+---pdf
+---timestamper                      [timestamper]
+---udpterm
\---v2.0                             [v2]
    +---doc                          [doc]
    +---fpga                         [fpga]
    |   +---output_files
    |   \---software                 [firmware]
    |       +---hit20_v3
    |       +---hit20_v3_bsp
    +---pads                         [pcb]
    |   +---assembly
    |   \---gerber
    +---pdf
    +---soft
    |   \---hit2017                  [pcsoft]
    \---root                         [root]
```

2. Hardware description

2.1. What it actually does...

Shortly speaking, it collects data from sensors and sends them to a PC.

Now more details:

The measurement ***setup*** may contain one or more ***boards***, each containing up to 2 (version 1) or 5 (version 2) ***sensors***. We want the boards to collect signal from sensors in a regular, synchronous manner (in form of ***frames***) and transmit them to a ***PC***.

Each of the 5 sensors (Hamamatsu S11865-64[5]) is an array of 64 photodiodes with an integration/readout circuit. This circuit allows simultaneous integration of photodiode currents for some period and sending it out as a series of voltage pulses over a single wire.

To perform this, it needs some constant-level signals:

– a clean power supply of 5V

– an extremly clean reference voltage of 4.5V (nom.)

– a photodiode bias voltage (essentially, the same signal as above)

Further, it needs two digital waveforms:

– a clock, which should be no more than 4 MHz

– an integration window signal (called RESET by Hamamatsu, don't ask why...)

Having these signals, it will produce the output signals:

– the Video signal – a series of analog pulses referred to the reference voltage (mentioned above) with negative polarity

– the Trig and EOS signals, which we don't use.

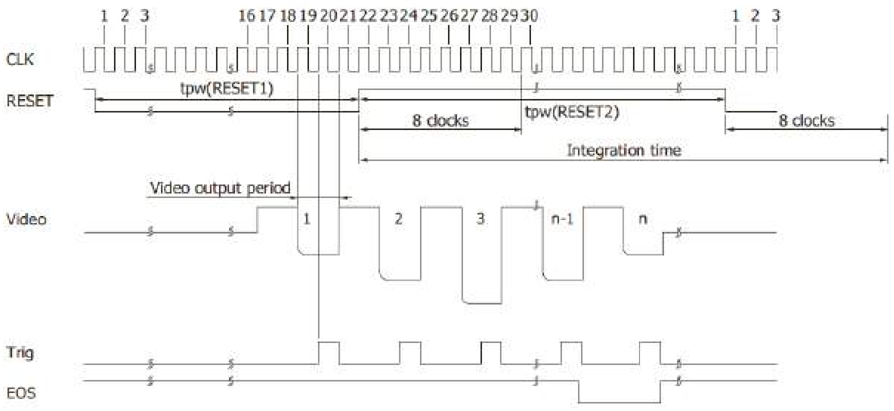


Fig. 2-1 Sensor's waveforms [5]

So, our task is now:

– Produce a valid clock signal

– Produce a valid Reset signal

– Collect Video pulses at right times and ADC them.

Again, an ADC (with an SPI interface) needs its own reference, clock and trigger, and a data line.

At the end, we should have 5·64 samples of sensor data somewhere in our memory. Now we can pack these data into a UDP packet and send it over Ethernet to a PC.

Simple? Not really. If we have multiple boards, we need to *synchronize* them. This is done by a synchronization link on two levels:

- ***frame-level synchronization***,
- ***fine synchronization***

The goal for the frame-level synchronization is to match corresponding frames from different boards. To do it, one of the boards (we call it ***master*** board, in contrast to ***slave*** boards) sends a number over the synchronization link after collecting each frame. This number automatically advances after each frame, giving a unique identifier. Now, it is received by all boards (including the master, over its internal loopback) and concatenated with the frame data.

Some technical details: the number is 9 bits wide and is sent over a RS-485 link with a baud rate of 1 Mbps, 1 stop bit, no parity.

This number, called the ***global frame counter***, is then used by the ***event builder*** (a part of the PC software) to put together data from different boards. There are also 16-bit-wide, unsynchronized ***local frame counters*** which might be helpful to monitor lost frames.

The task of the fine synchronization is to match integration window for all sensors and boards within some dozens of nanoseconds. This is done by another signal in the synchronization link, the frame clock signal. So, as for the frame-level synchronization, the trigger is produced by the master board and received by all of them. In this case, the signal has a form of a pulse (for version 1) or a square wave (for version 2). The duty cycle is actually unimportant; all devices trigger on the rising edge.

Unluckily, every board might have different delay between receiving the signal and triggering the measurement. On top of it, the synchronization link has its own delays. Therefore, each device has a programmable delay circuit to equalize all these delays. The equalization is a nasty, manual process, but fortunately it has to be done only once for each setup. See chapter 5.4: *Equalizing delays* for more details.

OK, two remarks about fine synchronization:

- All boards have their own oscillators, which remain unsynchronized. Therefore, the synchronization accuracy is limited by board's master clock frequency (which is 90 MHz for version 1 and 50 MHz for version 2). In practice, all boards can be synchronized within a ≈20 ns window.
- The sensor clock has a frequency of only 4 MHz (version 1) or 3.57 MHz (version 2). As the integration window must be synchronous with this clock, and we don't really want to do it (this would deteriorate our 20 ns to 250 ns), we do the opposite: we synchronize the sensor clock to our trigger (by extending it as needed)

And what if we want to synchronize our system to any external DAQ? There's yet another interface on our board: The SMA connector.

The name is historical, it was SMA for version 1; for version 2 it's an 8-bit parallel, digital input (in contrast to 1-bit digital input for version 1).

The idea is very simple: the data on the input is latched and transmitted with every frame. This way, we can collect any external, binary signal together with our data. This signal can be used for any purposes, including off-line synchronization. See chapter 3.2: *Timestamper* for more details.

Finally we have our data at the PC. They are received by a number of *receiver threads* (one per each board), buffered, matched together by the *event builder* and, last but not least, written to a disk.

2.2. ... and how

The project has several layers:

1. The hardware. This includes two physical PCBs (one custom, one from stock), some supporting mechanics (not described here), accompanying electronics (to provide synchronization, power, Ethernet etc.). One of the boards contains an FPGA, so:
2. The FPGA design. A project of the hardware layer of the FPGA. This project includes a software processor, so:
3. The firmware. This is the code for on-FPGA software processor.
4. On-line PC software, i.e. DAQ software.
5. Off-line PC software.

Between the lines are such things as communication protocols which connect various layers and elements of our system.

As mentioned above, there is also a Timestamper – a device (and related software) for producing pseudo-random waveform (timestamps) for synchronizing our system with another DAQ systems, like HIT’s Ethercat. From the point of view of our project, it’s an external device. Nevertheless, it will be described in this manual.

2.3. The Electronics

2.3.1. Overview

The electrical circuitry is divided into two PCBs, and therefore two main parts: the ***FPGA PCB*** and the ***frontend PCB***. They are both mounted on a common aluminum plate, which plays a role of a support, shielding, grounding and heat sink.

Additional mechanical components are used for supporting and shielding sensors and for mounting the whole entity to the detector.

The FPGA PCB is a standard Intel’s developer board for MAX10 devices: the *Max10 Development Kit.* See [1] for general description of the board and [2] for schematics diagram.

The frontend PCB includes all the circuitry to bridge sensors to the FPGA. This includes power regulators, sources of reference voltages, signal buffers, ADCs and interface logic.

It contains five almost identical blocks for five sensors, each including a signal buffer, an ADC and accompanying logic. They are divided into two groups (sensors 1-3 and 4-5) sharing common control signals.

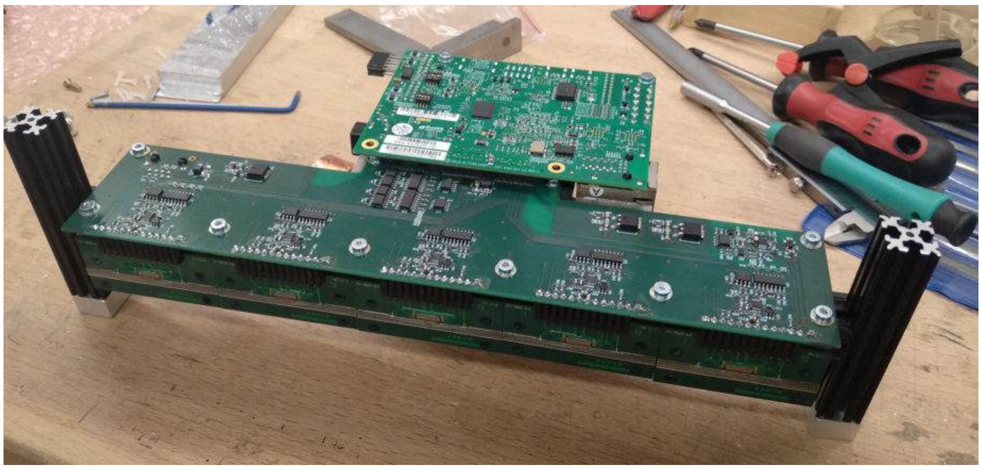


Fig. 2-2 The electronics (both boards connected together, without additional shielding on the FPGA board)

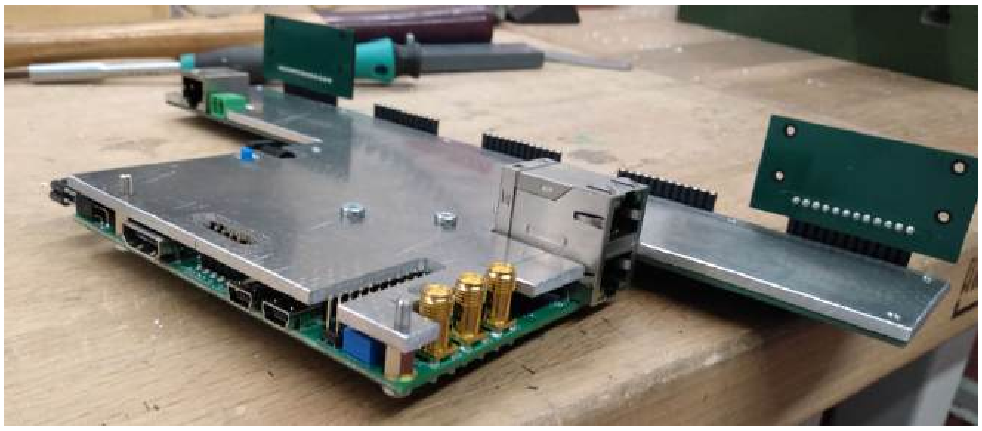


Fig. 2-3 Electronics with supporting plate

Power regulators (Fig. 2-11: +8V for analog circuits, +5V for sensors, +3.3V for logic and +2.5V for ADCs) as well as reference voltage sources (Fig. 2-10: 4.5V for sensors and 4.55V for ADCs) are common for all sensors.

It's important that the polarity of the sensor output signal is negative, i.e. the signal is close to reference for dark conditions and close to zero for maximum optical signal. So behaves the ADC output code, as well.

The difference between sensor reference and ADC reference creates a non-zero baseline (about 700 below maximum code) in the output signal. This behavior is intentional – zero baseline would lead to nonlinearities in presence of an additive noise.

The front-end board needs a single power supply of 12V/400mA. Depending on user's demands, it can be configured to draw its power from the FPGA board or from an external connector.

The two boards are connected via an HSMC connector. All digital connections between boards are LVDS to reduce noise.

2.3.2. Frontend Schematics

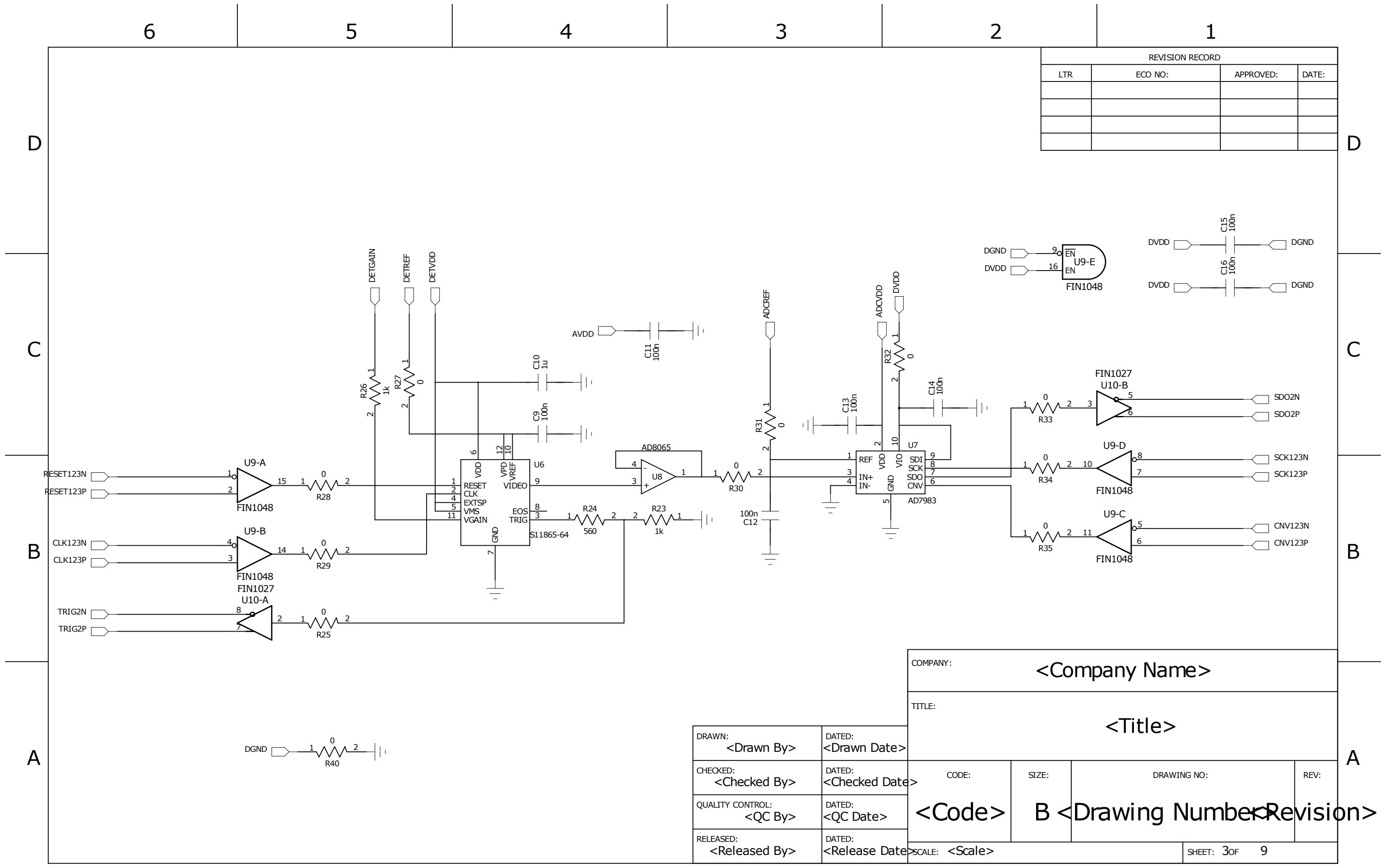


Fig. 2-4 Schematics diagram 1/9

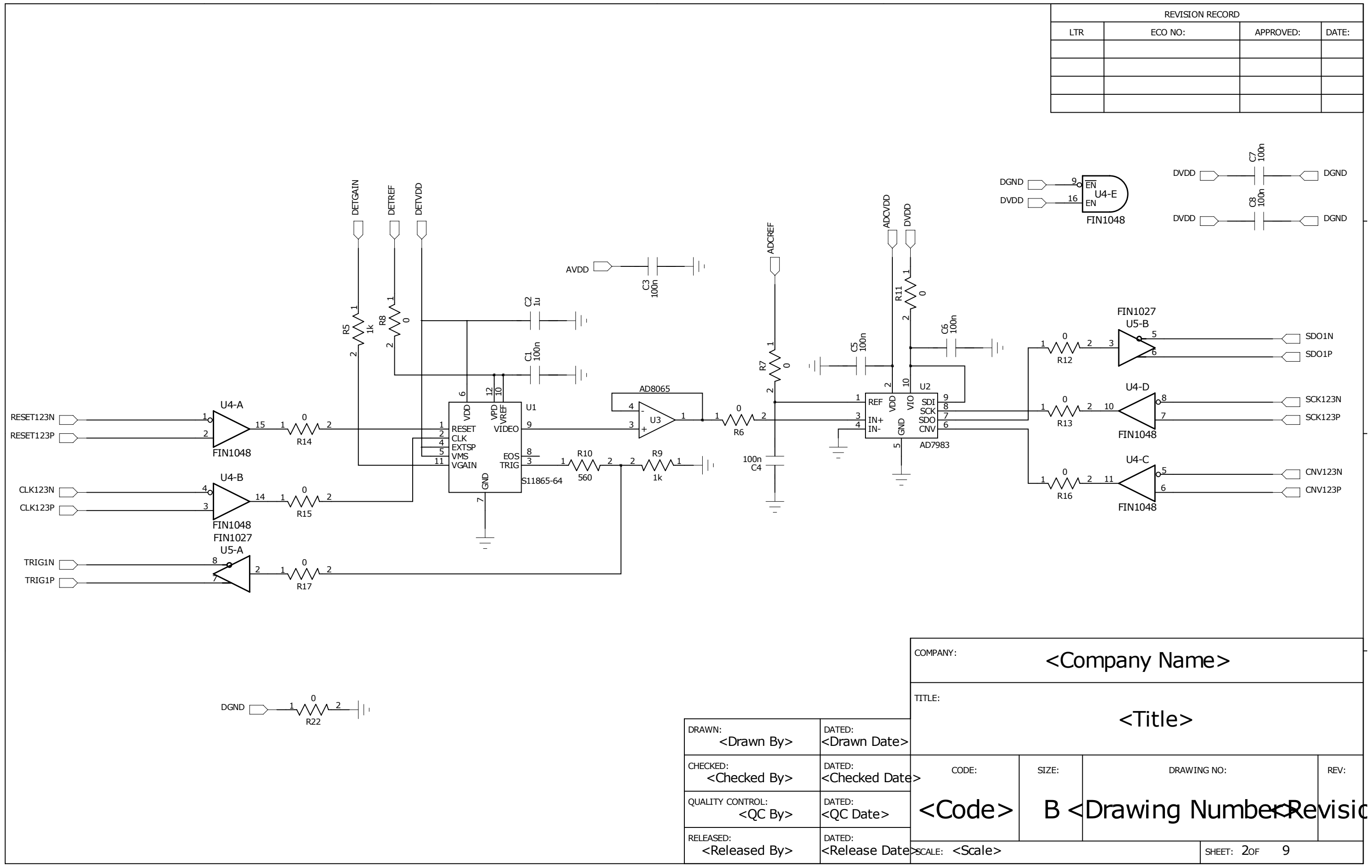


Fig. 2-5 Schematics diagram 2/9

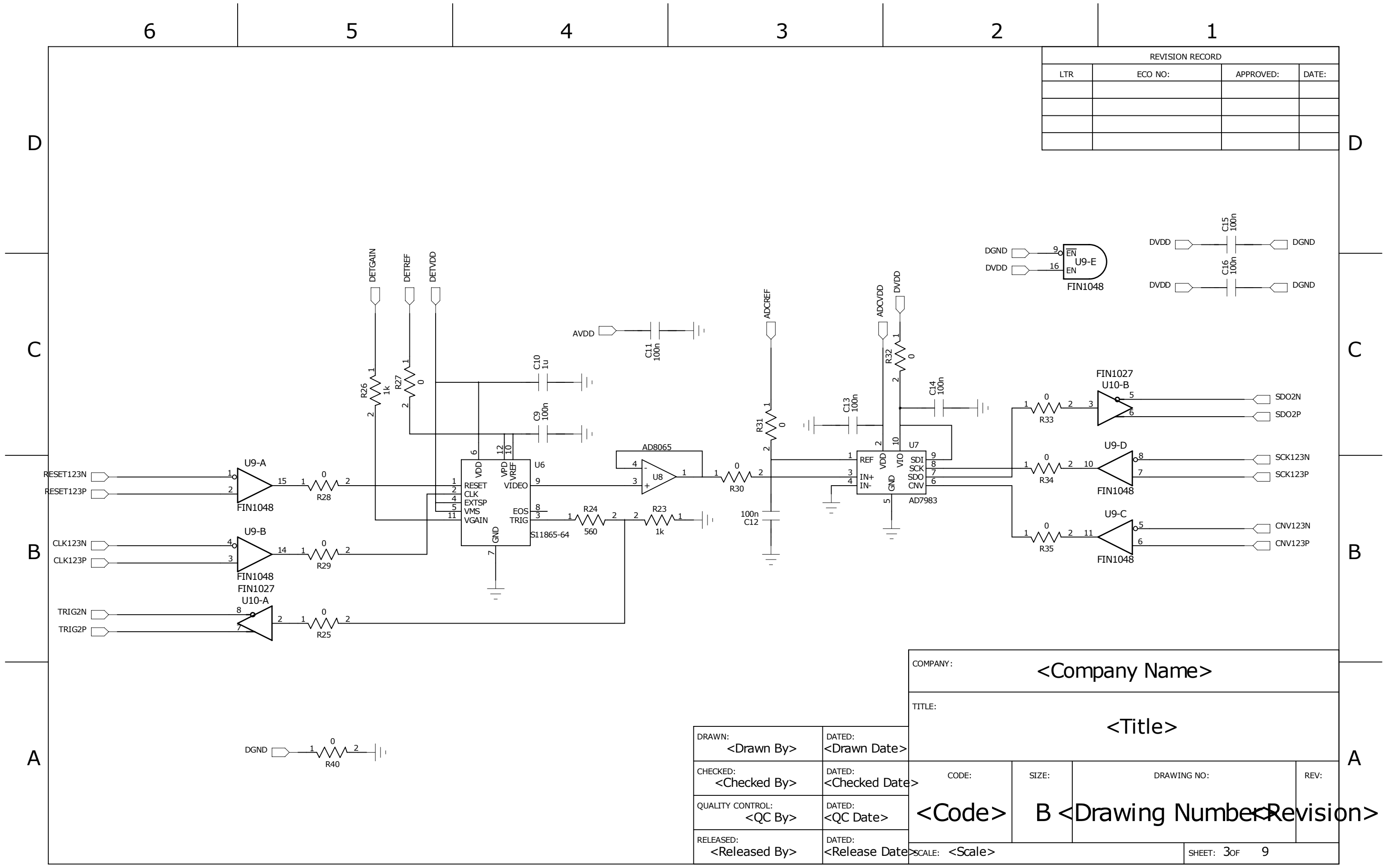


Fig. 2-6 Schematics diagram 3/9

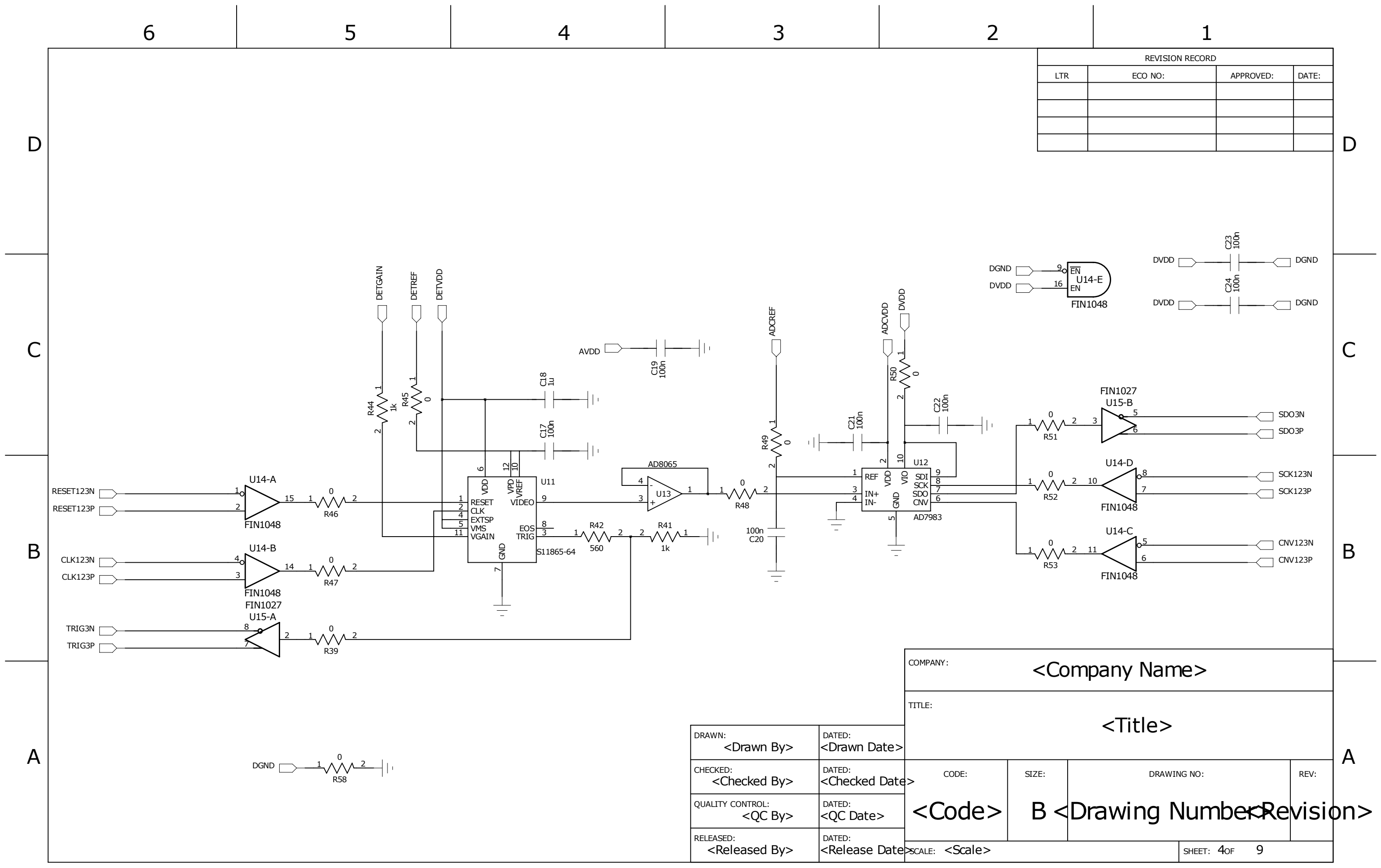


Fig. 2-7 Schematics diagram 4/9

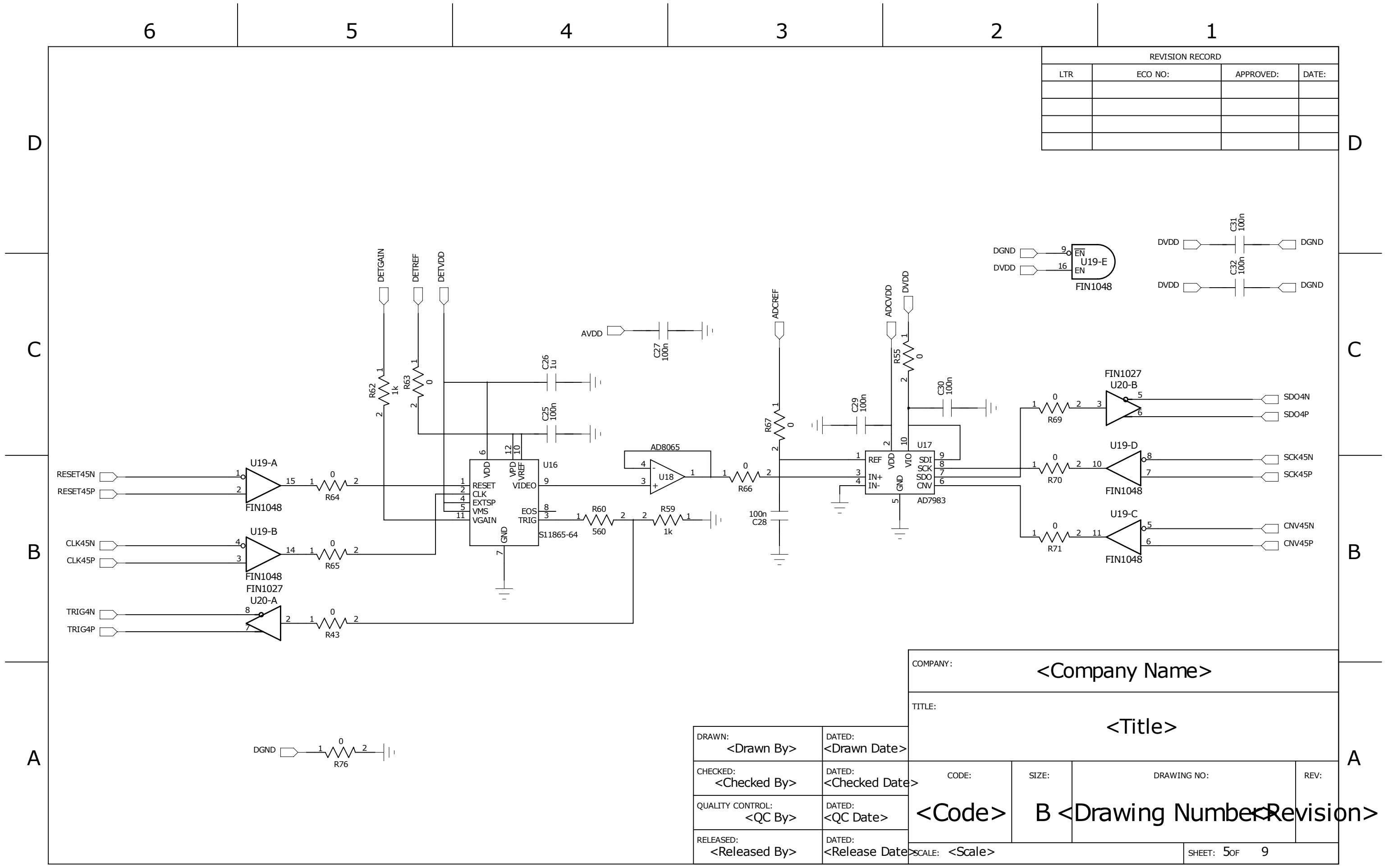


Fig. 2-8 Schematics diagram 5/9

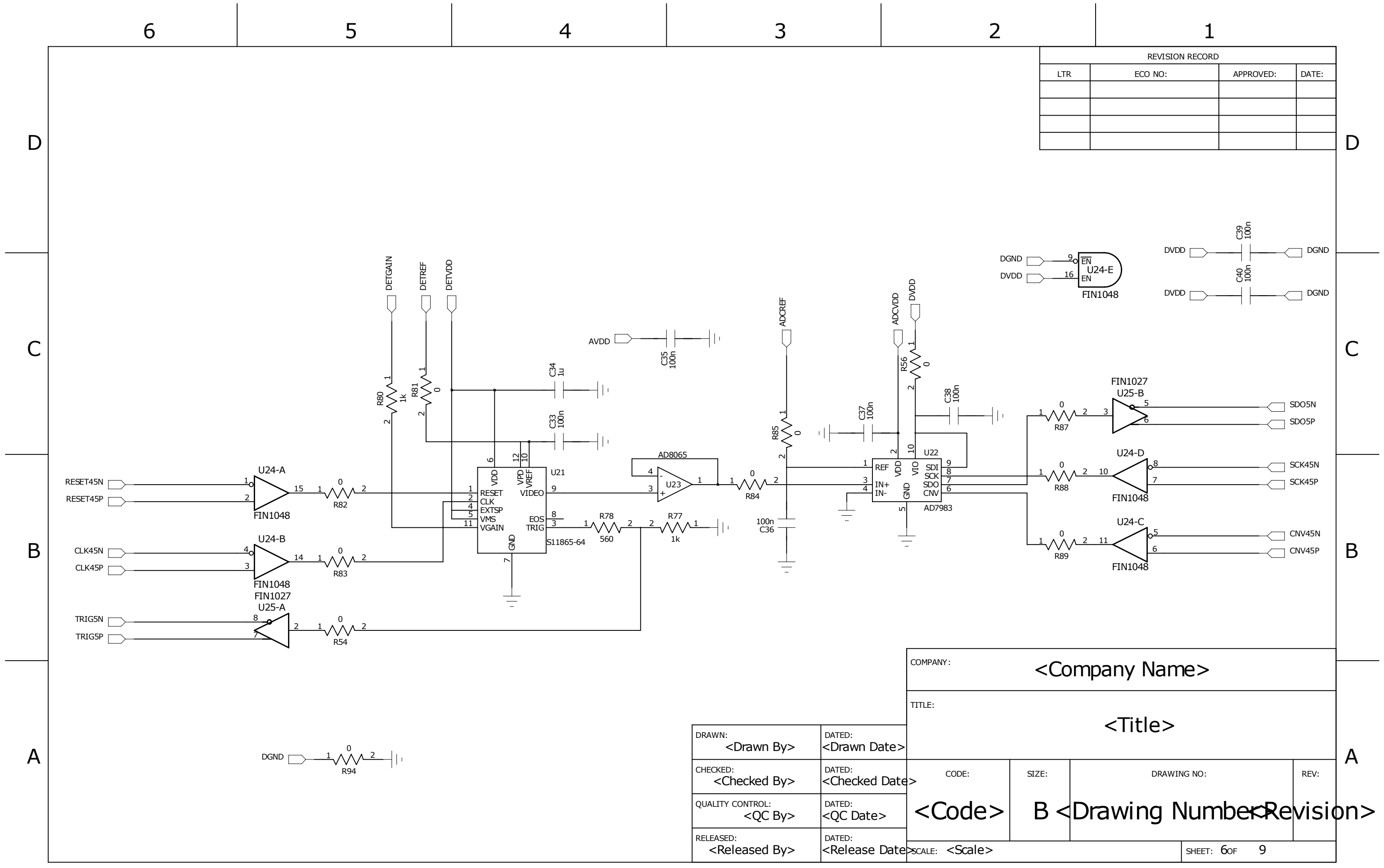


Fig. 2-9 Schematics diagram 6/9

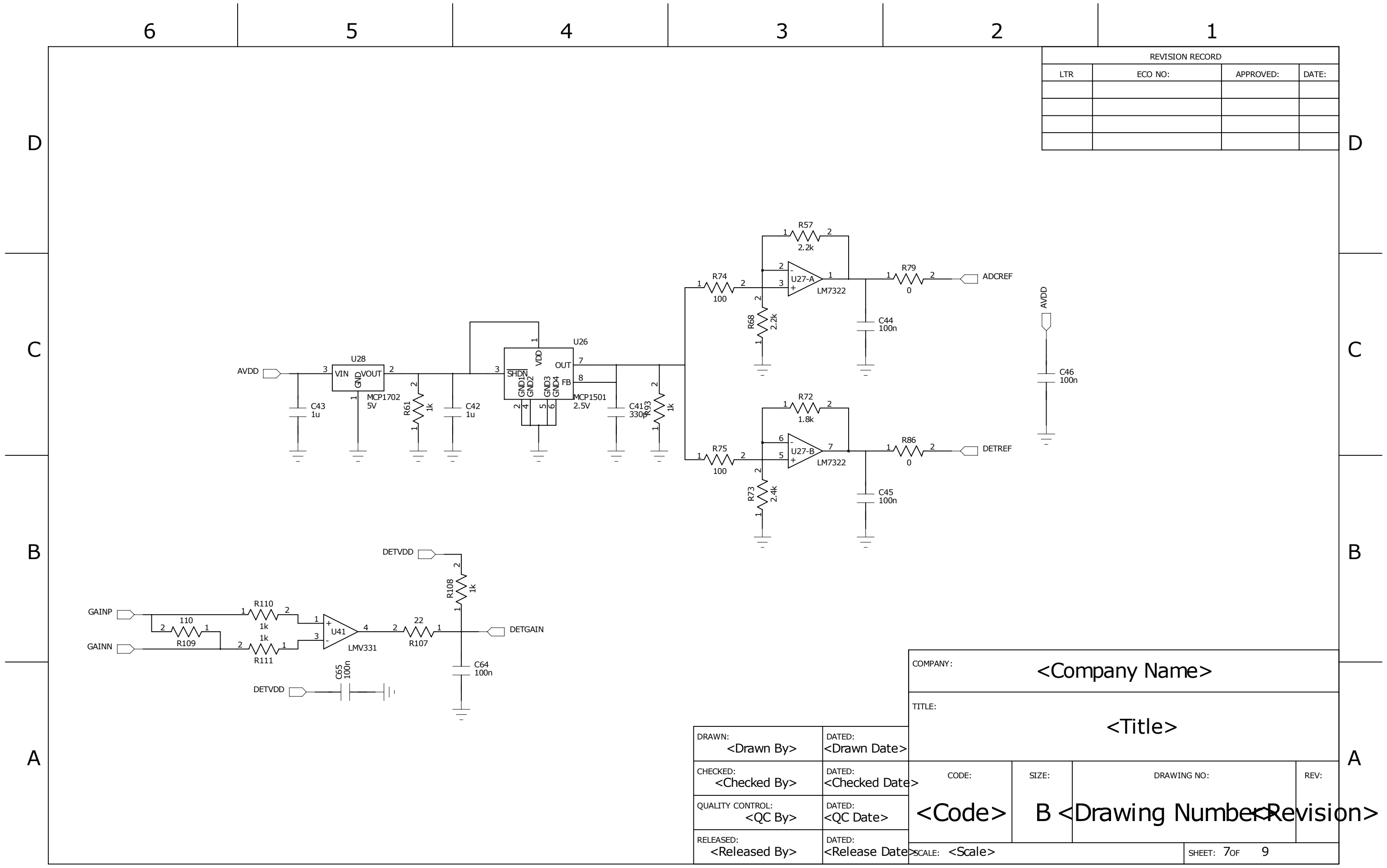


Fig. 2-10 Schematics diagram 7/9

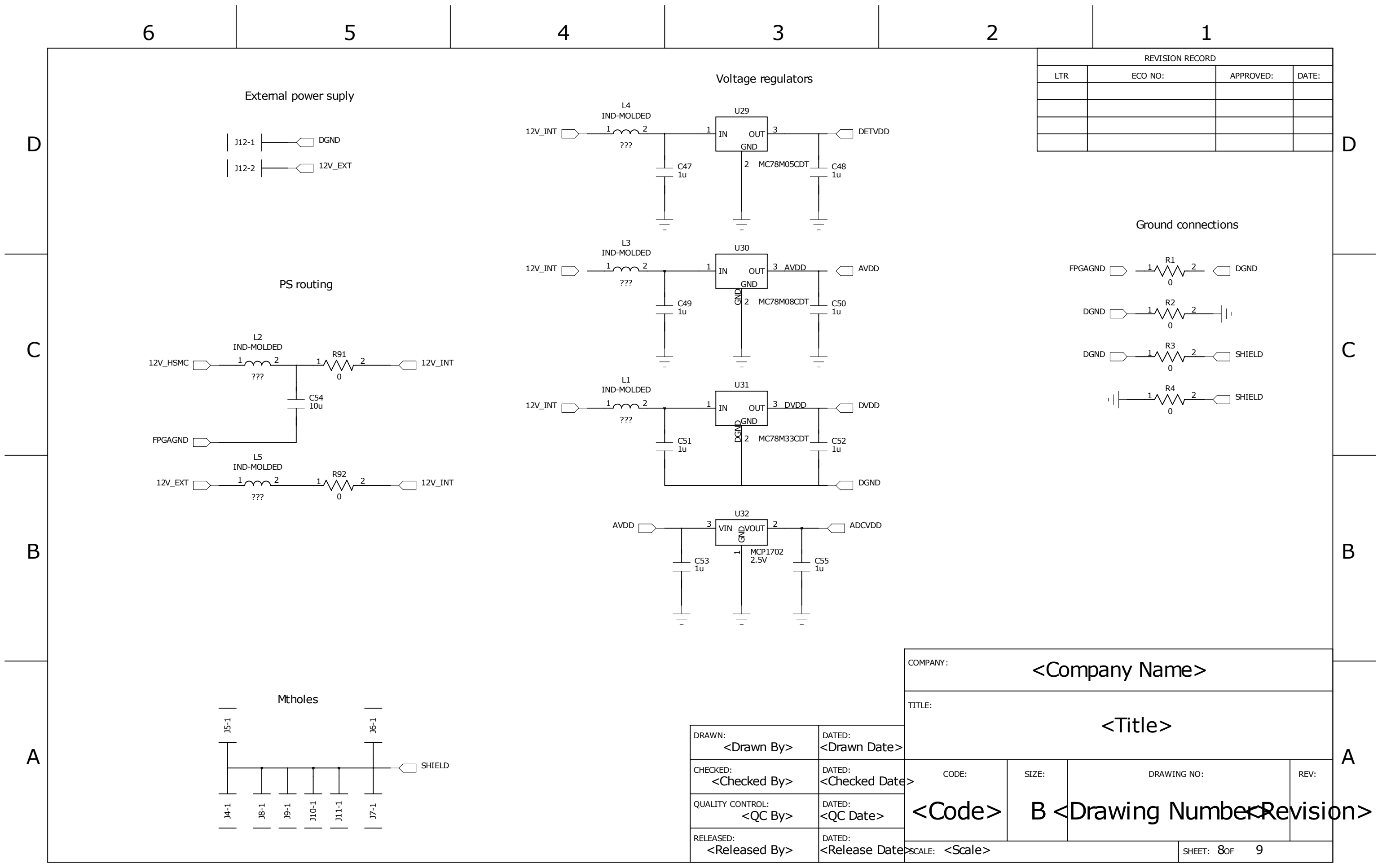

External power suply
J12-1 DGND
J12-2 12V_EXT
PS routing
12V_HSMC L2 IND-MOLDED ??? R91 0 12V_INT
C54 10u
FPGAGND
12V_EXT L5 IND-MOLDED ??? R92 0 12V_INT
Voltage regulators
12V_INT L4 IND-MOLDED ??? U29 MC78M05CDT DETVDD C47 1u C48 1u
12V_INT L3 IND-MOLDED ??? U30 MC78M08CDT AVDD C49 1u C50 1u
12V_INT L1 IND-MOLDED ??? U31 MC78M33CDT DVDD C51 1u C52 1u DGND
AVDD U32 MCP1702 2.5V ADCVDD C53 1u C55 1u
Ground connections
FPGAGND R1 0 DGND
DGND R2 0
DGND R3 0 SHIELD
R4 0 SHIELD
Mtholes
J5-1 J6-1 J4-1 J8-1 J9-1 J10-1 J11-1 J7-1 SHIELD
REVISION RECORD
LTR ECO NO: APPROVED: DATE:
COMPANY: <Company Name>
TITLE: <Title>
DRAWN: <Drawn By> DATED: <Drawn Date>
CHECKED: <Checked By> DATED: <Checked Date>
QUALITY CONTROL: <QC By> DATED: <QC Date>
RELEASED: <Released By> DATED: <Release Date>
CODE: <Code> SIZE: B DRAWING NO: <Drawing Number> REV: <Revision>
SCALE: <Scale> SHEET: 8 OF 9

Fig. 2-11 Schematics diagram 8/9

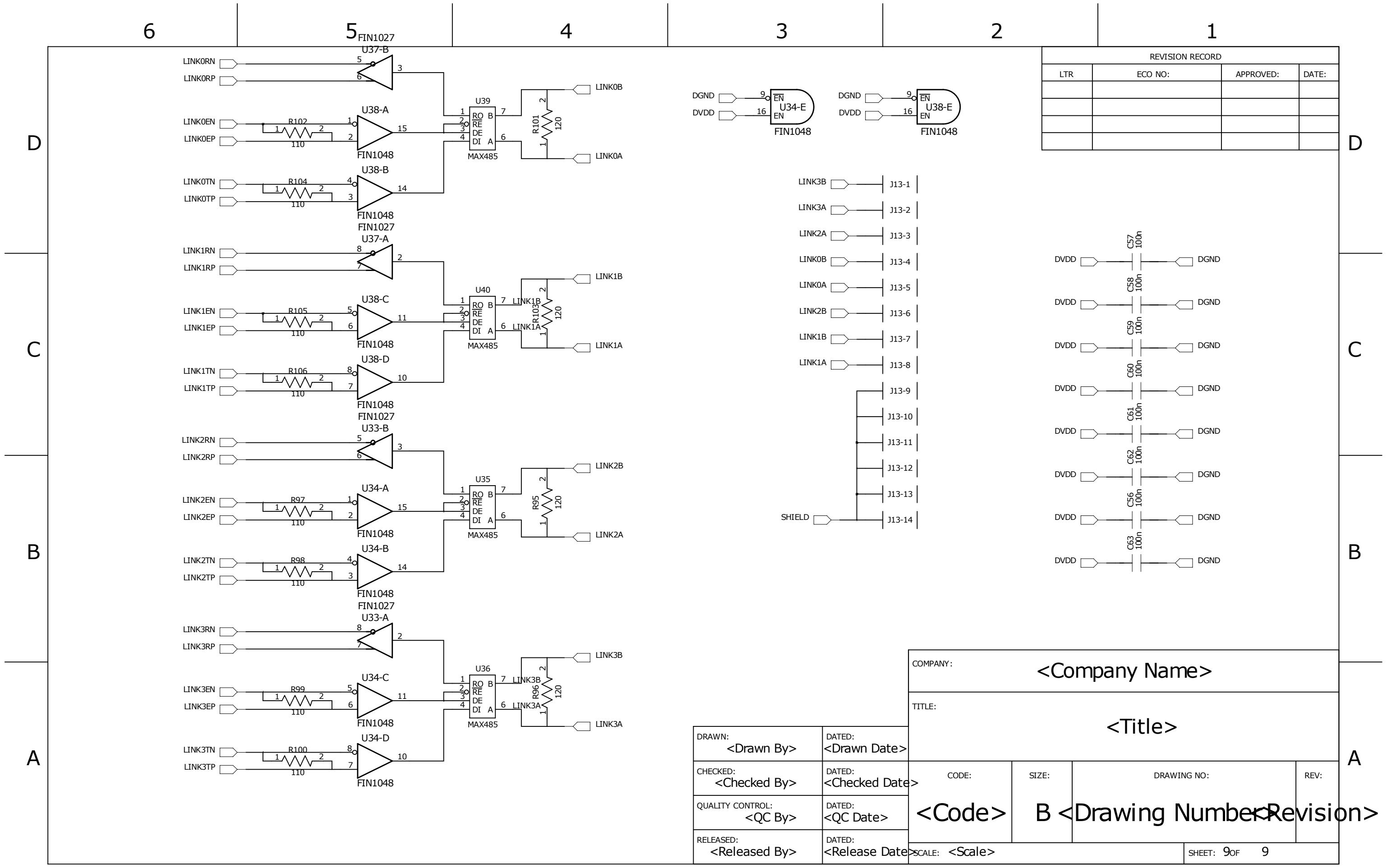


Fig. 2-12 Schematics diagram 9/9

2.3.3. Assembly drawings

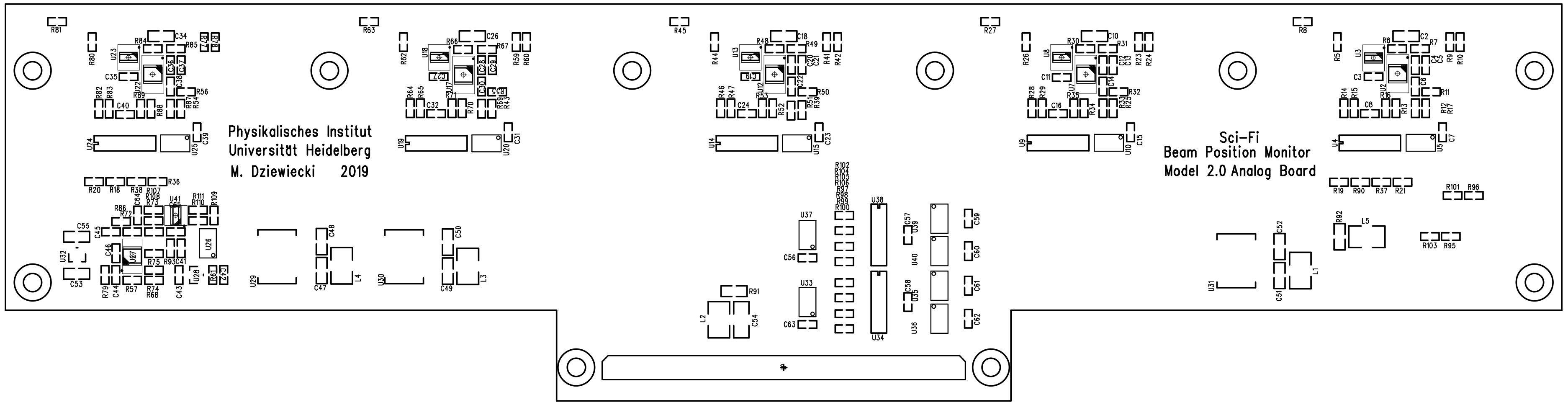


Fig. 2-13 Assembly drawing - top

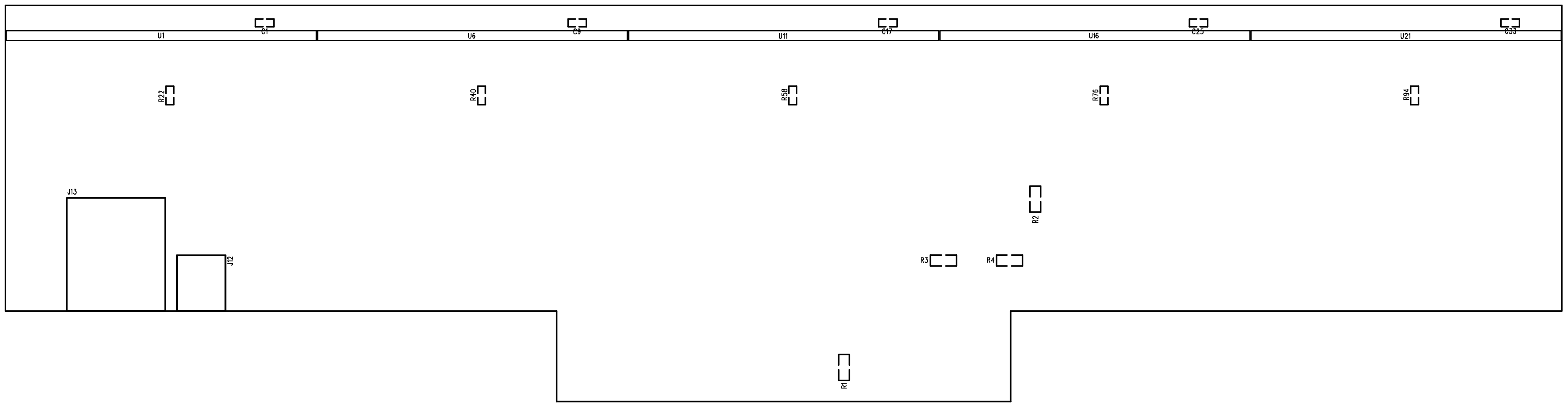


Fig. 2-14 Assembly drawing - bottom

2.3.4. Component list

Ref.	Part Name	Value
U2	AD7983	
U7	AD7983	
U12	AD7983	
U17	AD7983	
U22	AD7983	
U3	AD8065	
U8	AD8065	
U13	AD8065	
U18	AD8065	
U23	AD8065	
J1	ASP-122952-01	
C1	CAP0603	100n
C3	CAP0603	100n
C4	CAP0603	100n
C5	CAP0603	100n
C6	CAP0603	100n
C7	CAP0603	100n
C8	CAP0603	100n
C9	CAP0603	100n
C11	CAP0603	100n
C12	CAP0603	100n
C13	CAP0603	100n
C14	CAP0603	100n
C15	CAP0603	100n
C16	CAP0603	100n
C17	CAP0603	100n
C19	CAP0603	100n
C20	CAP0603	100n
C21	CAP0603	100n
C22	CAP0603	100n
C23	CAP0603	100n
C24	CAP0603	100n
C25	CAP0603	100n
C27	CAP0603	100n
C28	CAP0603	100n
C29	CAP0603	100n
C30	CAP0603	100n
C31	CAP0603	100n
C32	CAP0603	100n
C33	CAP0603	100n
C35	CAP0603	100n
C36	CAP0603	100n
C37	CAP0603	100n
C38	CAP0603	100n
C39	CAP0603	100n
C40	CAP0603	100n
C41	CAP0603	330p
C42	CAP0603	1u
C43	CAP0603	1u
C44	CAP0603	100n
C45	CAP0603	100n
C46	CAP0603	100n
C56	CAP0603	100n
C57	CAP0603	100n
C58	CAP0603	100n
C59	CAP0603	100n
C60	CAP0603	100n
C61	CAP0603	100n
C62	CAP0603	100n
C63	CAP0603	100n
C64	CAP0603	100n
C65	CAP0603	100n
C2	CAP0805	1u
C10	CAP0805	1u
C18	CAP0805	1u
C26	CAP0805	1u
C34	CAP0805	1u
C47	CAP0805	1u
C48	CAP0805	1u
C49	CAP0805	1u
C50	CAP0805	1u
C51	CAP0805	1u
C52	CAP0805	1u
C53	CAP0805	1u
C55	CAP0805	1u
C54	CAP1206	10u
U5	FIN1027	
U10	FIN1027	
U15	FIN1027	
U20	FIN1027	
U25	FIN1027	
U33	FIN1027	
U37	FIN1027	
U4	FIN1048	

Ref.	Part Name	Value
U9	FIN1048	
U14	FIN1048	
U19	FIN1048	
U24	FIN1048	
U34	FIN1048	
U38	FIN1048	
J12	HEADER02	
L1	IND-MOLDED	???
L2	IND-MOLDED	???
L3	IND-MOLDED	???
L4	IND-MOLDED	???
L5	IND-MOLDED	???
U27	LM7322	
U41	LMV331	
U35	MAX485	
U36	MAX485	
U39	MAX485	
U40	MAX485	
U29	MC78M05CDT	
U30	MC78M08CDT	
U31	MC78M33CDT	
U26	MCP1501	
U28	MCP1702	
U32	MCP1702	
R5	RES0603	1k
R6	RES0603	0
R7	RES0603	0
R8	RES0603	0
R9	RES0603	1k
R10	RES0603	560
R11	RES0603	0
R12	RES0603	0
R13	RES0603	0
R14	RES0603	0
R15	RES0603	0
R16	RES0603	0
R17	RES0603	0
R18	RES0603	110
R19	RES0603	110
R20	RES0603	110
R21	RES0603	110
R22	RES0603	0
R23	RES0603	1k
R24	RES0603	560
R25	RES0603	0

Ref.	Part Name	Value
R26	RES0603	1k
R27	RES0603	0
R28	RES0603	0
R29	RES0603	0
R30	RES0603	0
R31	RES0603	0
R32	RES0603	0
R33	RES0603	0
R34	RES0603	0
R35	RES0603	0
R36	RES0603	110
R37	RES0603	110
R38	RES0603	110
R39	RES0603	0
R40	RES0603	0
R41	RES0603	1k
R42	RES0603	560
R43	RES0603	0
R44	RES0603	1k
R45	RES0603	0
R46	RES0603	0
R47	RES0603	0
R48	RES0603	0
R49	RES0603	0
R50	RES0603	0
R51	RES0603	0
R52	RES0603	0
R53	RES0603	0
R54	RES0603	0
R55	RES0603	0
R56	RES0603	0
R57	RES0603	2.2k
R58	RES0603	0
R59	RES0603	1k
R60	RES0603	560
R61	RES0603	1k
R62	RES0603	1k
R63	RES0603	0
R64	RES0603	0
R65	RES0603	0
R66	RES0603	0
R67	RES0603	0
R68	RES0603	2.2k
R69	RES0603	0
R70	RES0603	0

Ref.	Part Name	Value
R71	RES0603	0
R72	RES0603	1.8k
R73	RES0603	2.4k
R74	RES0603	100
R75	RES0603	100
R76	RES0603	0
R77	RES0603	1k
R78	RES0603	560
R79	RES0603	0
R80	RES0603	1k
R81	RES0603	0
R82	RES0603	0
R83	RES0603	0
R84	RES0603	0
R85	RES0603	0
R86	RES0603	0
R87	RES0603	0
R88	RES0603	0
R89	RES0603	0
R90	RES0603	110
R93	RES0603	1k
R94	RES0603	0
R95	RES0603	120
R96	RES0603	120
R97	RES0603	110
R98	RES0603	110
R99	RES0603	110
R100	RES0603	110
R101	RES0603	120
R102	RES0603	110
R103	RES0603	120
R104	RES0603	110
R105	RES0603	110
R106	RES0603	110
R107	RES0603	22
R108	RES0603	1k
R109	RES0603	110
R110	RES0603	1k
R111	RES0603	1k
R1	RES0805	0
R2	RES0805	0
R3	RES0805	0
R4	RES0805	0
R91	RES0805	0
R92	RES0805	0
J13	RJ45_FRJAE-488	
U1	S11865-64	
U6	S11865-64	
U11	S11865-64	
U16	S11865-64	
U21	S11865-64	

Table 2-1 Components summary

2.3.5. Debug-time updates

The following table summarizes all changes to component values done during device debugging.

No	Component	Change	Explanation
1	R57	2.2k → 2.2k || 10k = 1.8k	Change ADC reference from 5.0V to 4.55V
2	C44	100n → 100n + 1u	Reduce noise
3	R73	2.4k → 2.4k || 36 k = 2.25k	Change detector reference from 4.375V to 4.5V
4	R74, R75	100 → 10k	Make LP filters which significantly reduce reference noise
5	U27A IN+, U27B IN+	Add 10uF to ground	
6	R109	100 → 10k	Reduce diff pair current (it's not LVDS)
7	C52	1u → 1u + 10u	Extinguish oscillations of a power regulator
8	C4, C12, C20, C28, C36	100n → 100n + 1u	Reduce noise
9	C1, C9, C17, C25, C33	100n → 100n + 1u	Reduce noise
10	C6, C14, C22, C30, C38	100n → 100n + 1u	Reduce noise

Table 2-2 Debug-time updates summary

2.3.6. Further remarks

- The power source of the FPGA board can be configured using R91 (0Ω):
  - Remove R91 to use external power supply over J12.
  - Install R91 to use HSMC power supply from the FPGA board.
- A number of further 0Ω resistors are intended for debug purposes: they can be replaced by EMI chokes or resistors to decrease noise level. Anyway, an extensive research has been already done on it and for these 0Ωs which are finally not replaced, there was no evident improvement in noise figure.
- By default, the LVDS receive lines are not terminated at the FPGA boards. It’s essential to install the termination resistors when cloning the setup.
- There are some LEDs on the FPGA board. They should be all optically isolated or completely removed to avoid parasitic light in the detector box.
- The FPGA and power regulators dissipate significant amount of heat. A total power dissipation of ca 10W is expected.
- The device is equipped with two Ethernet ports. However, currently only one (the bottom one) is configured. Do not connect anything to the other connector.

2.3.7. Connections and controls

2.3.7.1. Switches

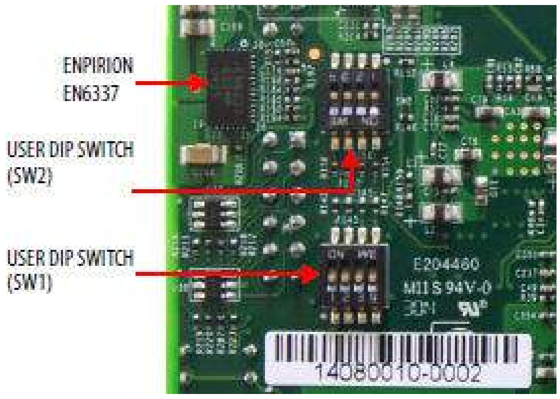


Fig. 2-15 Important DIP-switches

There are two important DIP-switches on the FPGA PCB back side (metal shield must be removed to access them). Their setting is summarized in the tables below.

Nr	Description	Default setting
SW1.1	IP setting:	OFF
SW1.2	Device IP is 10.0.7.16 + switch setting:	OFF
SW1.3	0 0 0 0 for 10.0.7.16	OFF
SW1.4	1 1 1 1 for 10.0.7.31	OFF
SW2.1	DHCP enable (should be always off)	OFF
SW2.2	CONFIG_SEL	**ON**
SW2.3	VTAP_BYPASS	OFF
SW2.4	HSMC_BYPASS	**ON**

Table 2-3 On-board DIP-switches

2.3.7.2. LEDs

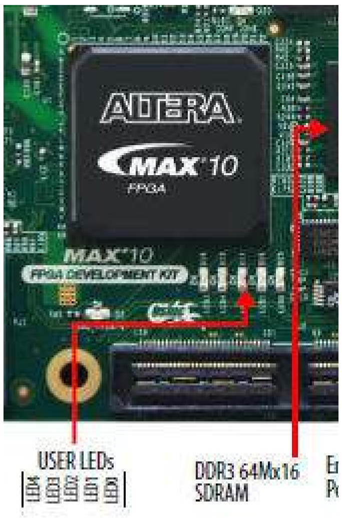


Fig. 2-16 LED placement

There are five debug LEDs on-board. During normal operation, they should be covered by a piece of sticky tape or painted black or removed. Anyway, they should be all turned off by software during run.

Remark: There are plenty of other LEDs on board and in the Ethernet socket. They should be all covered, painted black or removed (the LEDs or their series resistors). There is no way to switch off all of them by software.

LED nr	Description
0	Debug LED
1	Unused
2	Unused
3	Unused
4	Blinking if chip OK. Switched off for detector run.

Table 2-4 On-board LEDs

2.3.7.3. Sockets and connectors

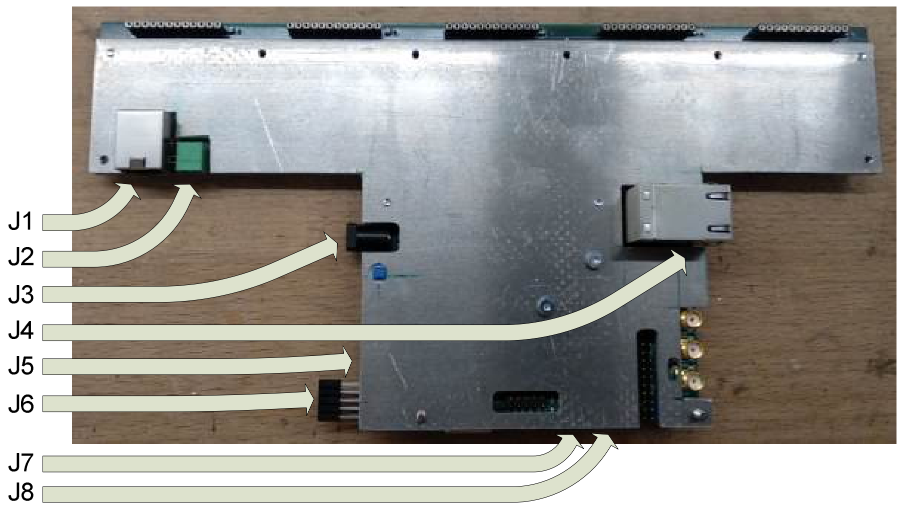


Fig. 2-17 Important connectors

All important connectors are marked in the figure above. Their function and pin-outs are described below.

J1: Synchronization connector: RJ-45

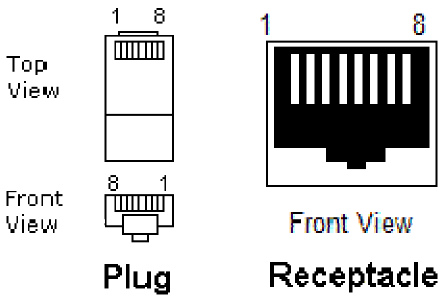


Fig. 2-18 RJ-45 pin numbering

Nr	Name	Function	Remarks
1	LINK3B	Frame clock	Output for master configuration, input for slave
2	LINK3A		
3	LINK2A	Synchronization serial link	Output for master configuration, input for slave
4	LINK0B	unused	
5	LINK0A		
6	LINK2B	Synchronization serial link	Output for master configuration, input for slave
7	LINK1B	unused	
8	LINK1A		

Table 2-5 Synchronization connector pinout

J2: Frontend power connector: 3.5 mm EDG connector

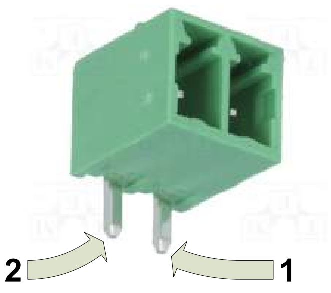


Fig. 2-19 Power connector pin numbering

Nr	Function
1	Ground
2	+12V

Table 2-6 Frontend power connector pinout

J3: FPGA power connector

Nr	Function
Sleeve	Ground
Pin	+12V

Table 2-7 FPGA power connector pinout

J4: Ethernet connector: RJ-45

Standard 1000Base-TX connector. Use the **bottom** socket (closer to the PCB)

J5: External synchronization input (PMODA): Intel PMOD connector (NDR)

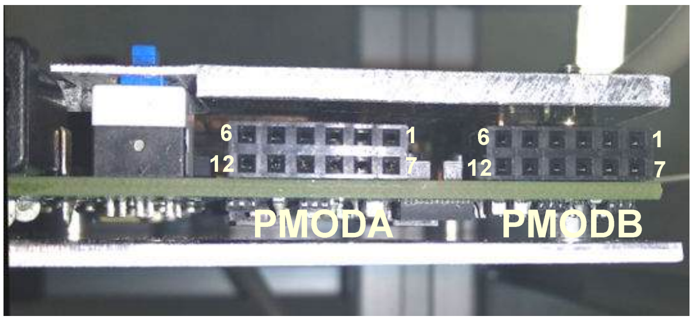


Fig. 2-20 Pmod connectors pin order

Description	Nr	Nr	Description
In bit 0 (LSB)	1	7	In bit 4
In bit 1	2	8	In bit 5
In bit 2	3	9	In bit 6
In bit 3	4	10	In bit 7 (MSB)
Ground	5	11	Ground
+3.3V	6	12	+3.3V

Table 2-8 External synchronization socket pinout

J6: Debug socket (PMODB): Intel PMOD connector (NDR)

This connector outputs some internal FPGA signals for debug purposes. It' not intended to be used in regular runs.

Description	Nr	Nr	Description
int_trig_debouncer_in	1	7	int_sensor_status [2]
int_sensor_in_trg	2	8	int_sensor_status [3]
int_sensor_status [0]	3	9	int_sensor_status [4]
int_sensor_status [1]	4	10	int_sensor_status [5]
Ground	5	11	Ground
+3.3V	6	12	+3.3V

Table 2-9 Debug socket pinout

J7: Debug USB serial port: mini-USB

Standard mini-USB connector.

J8: JTAG programmer connection: mini-USB

Standard mini-USB connector.

2.4. Details on synchronization

As already mentioned, the synchronization between multiple boards is performed on two levels:

- frame-level synchronization,
- fine synchronization

This is done by synchronization link (J1 connector on the frontend board) using two pairs of wires, each transmitting one signal in RS-485 standard. The two signals are the ***frame clock*** and the ***synchronization serial link*** (see chapter 2.3.7.3, *Sockets and connectors*). They may be transmitted either directly between a master and slave board (if there's only one slave board in the system) or through a synchronization board, as shown in Fig. 2-21.

The two signal are always produced by the Master board. They are transmitted via RS-485 converters (working as transmitters) outside the board. The same signals are routed to the sensor interface via debouncers (not shown).

On the slave side, the signals are received by two RS-485 converters (this time working as receivers) and routed to the sensor interface via debouncers.

The role of the synchro board is to populate synchronization signals to multiple boards.

The RS-485 chain introduces a substantial delay, which needs to be compensated for fine synchronization. Therefore, the frame trigger is delayed by a programmable time on each board, as shown in Fig. 2-22. Setting the delay time for each board is user's responsibility.

The synchronization serial link doesn't need any time compensation.

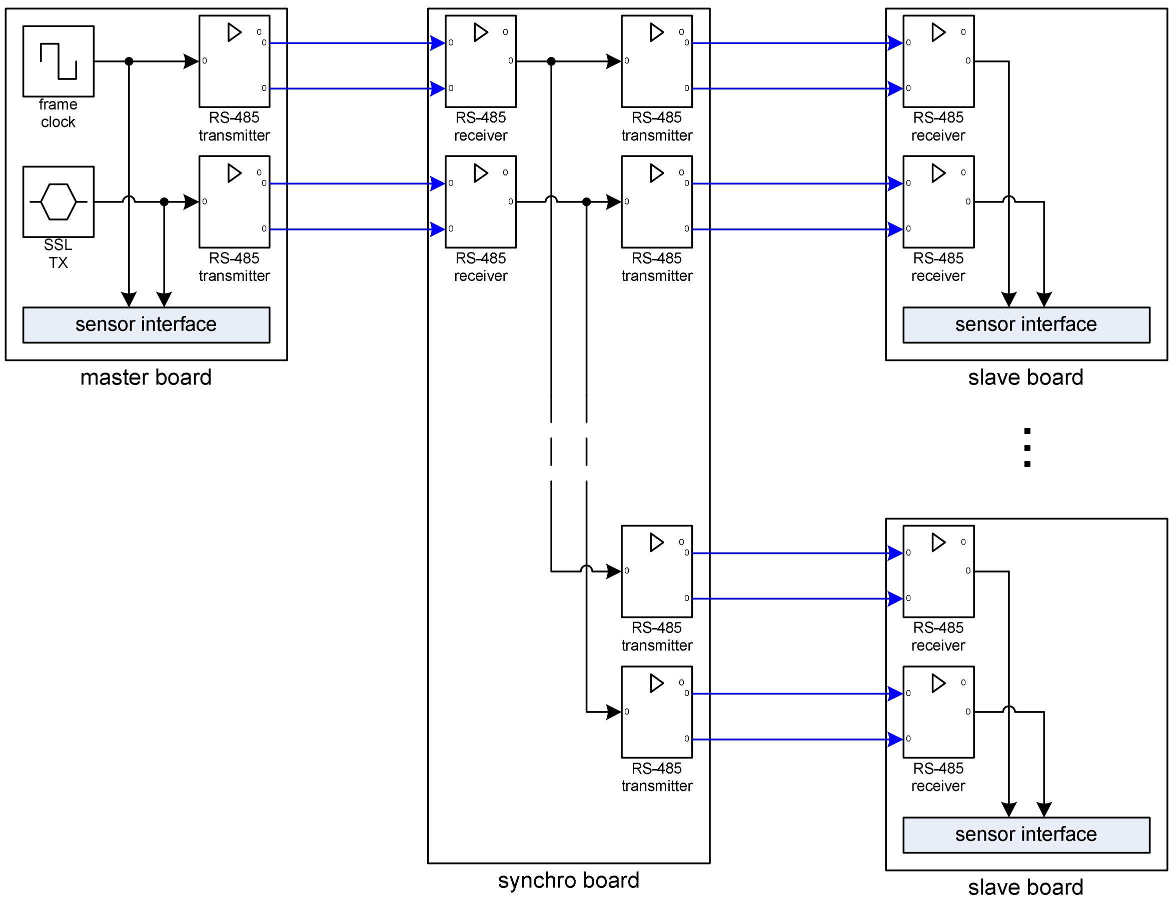


Fig. 2-21 Simplified diagram of the synchronization system. SSL TX is the transmitter for synchronization serial link. RS-485 pairs are shown in blue. Debouncers are omitted.

As shown in Fig. 2-23, the data over serial link is transmitted at the same time as the frame data gets transmitted over Ethernet. It means that new link data is not available (especially at slave side) when transmitting frame. Therefore, the transmitted link data is always used only for the *next* frame, both for master and slave, as shown in the figure.

As an implication, the first frame ever sent, it theoretically remains unsynchronized. However, as all frame counters are reset before starting a run, this first frame should be sent with a global frame counter of zero and will by correctly put together by the event builder.

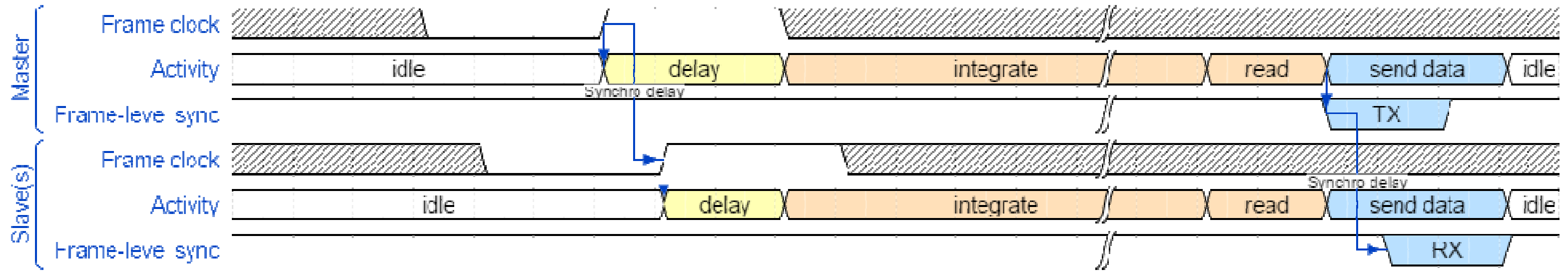


Fig. 2-22 Basic timing for master/slave synchronization

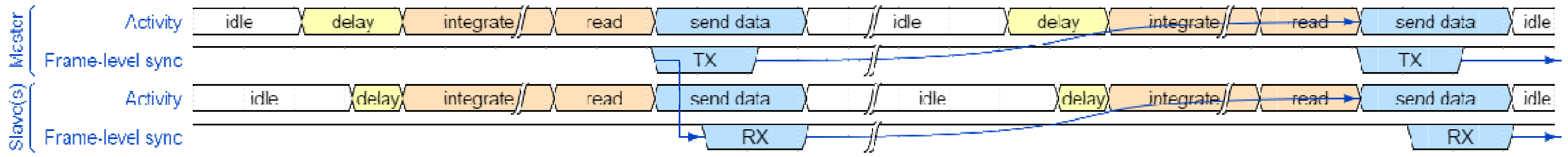


Fig. 2-23 Details on serial link timing

2.5. The FPGA

2.5.1. General

The FPGA code is derived from Intel's '*Simple socket server*' example project, however it's deeply modified. It's based on the QSYS architecture (warning, Intel changes this name every year, currently it's *System designer* or whatever), i.e. a framework for easy building various computer and computer-like systems.

All the custom code (and all generated code) is written in Verilog.

The FPGA code san be basically found in `[fpga]` directory. The main file is `[fpga]/m10_rgmii.v`.

2.5.2. Block system design

A simplified block diagram of the FPGA is shown in Fig. 2-24. Some signals (clocks, reset, debug) were omitted.

Most of the logic is incorporated into the QSYS component (described below). All other blocks are essentially an interface between the QSYS and external hardware. Most of them are circuits for generating and routing various clocks.

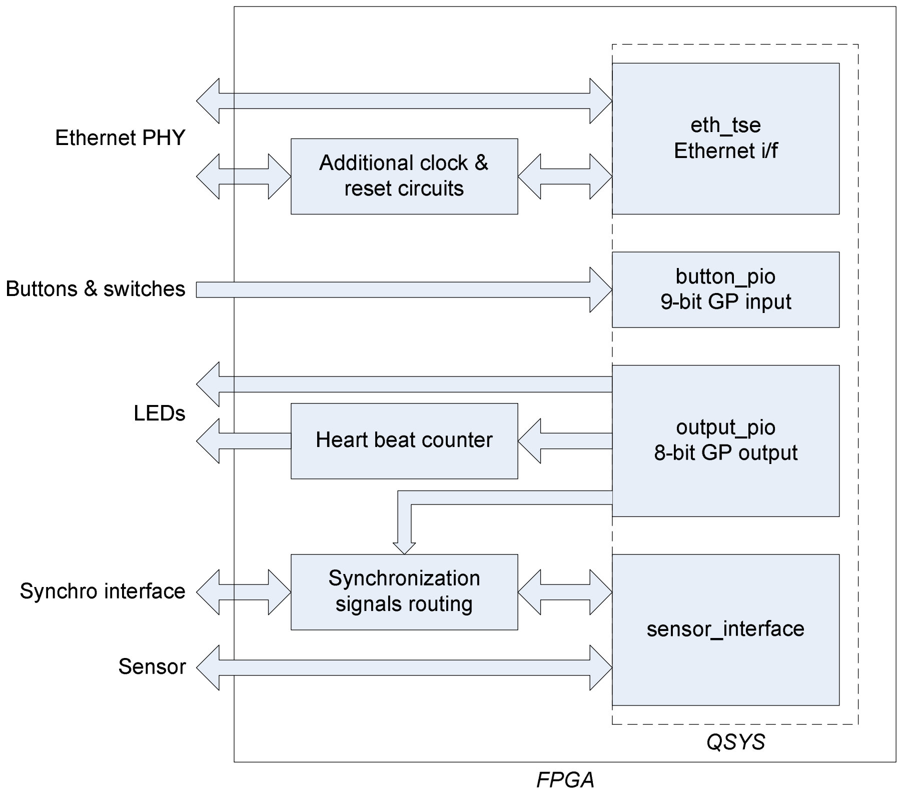


Fig. 2-24 Simplified block diagram of the FPGA

2.5.3. The QSYS

A simplified block diagram of the QSYS is shown in Fig. 2-25. Some second-importance components have been omitted.

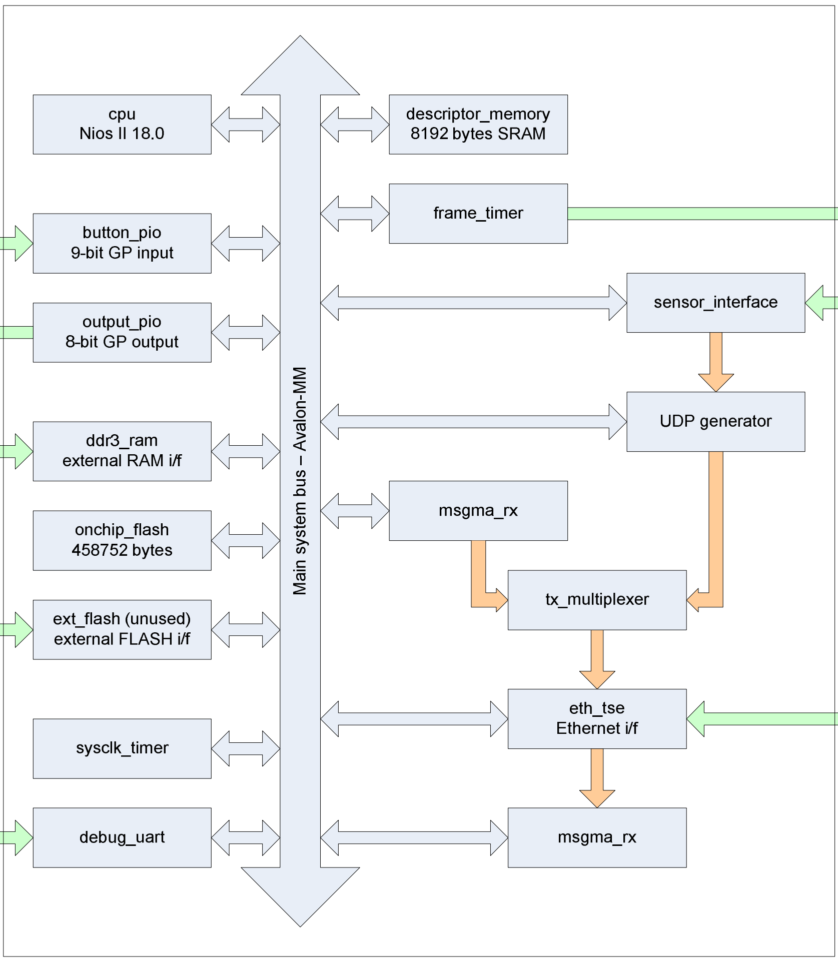


Fig. 2-25 Simplified block diagram of the QSYS

There are variety of memories in the system:

- An external DDR3 RAM. It serves as main program and data memory for the CPU.
- Onchip Flash memory. It’s the storage place for the program code. Upon system start, the contents of this memory gets loaded to the DDR3 RAM. This allows for faster program execution.
- External QSPI flash memory. It’s not used now, however it might be helpful in the future if the program code grows significantly or there’s a need to store a lot of non-volatile data (e.g. look-up tables for data processing)
- Descriptor-memory. It’s an on-chip static RAM used by DMA controllers for fetching commands. Use of a static memory speeds-up the DMA operation.

The most important part is to the right of the system bus. It shows the Ethernet interface (*eth_tse*) with its DMA controllers (*msgdma_tx* and *msgdma_rx*). The controllers use descriptor memory to fetch commands while the data itself is written to / read from the

DDR3 RAM. As the UDP generator must send data directly to Ethernet as well, there's the *tx_multiplexer* block which allows this.

The operation of the multiplexer is very simple: the first input which sends a packet start signal, passes through the multiplexer. If the other input sends its packet start before the first one finished, it's backpressed. This simple mechanism doesn't require any intervention of the CPU or any external control blocks.

The system includes a pair of general purpose I/O modules (*button_pio* and *output_pio*) for basic user interface (switches and LEDs) and controlling some out-of-QSYS components of the system.

Further, there are two timers: one (*sysclk_timer*) is used as clock for the operating system where another one (*frame_timer*) is used as frame clock.

Finally, the *debug_uart* module serves the on-board USB serial port used for debug purpoes and initial system configuration (see chapter 2.3.7.3: *Sockets and connectors*).

For the ones not familiar with QSYS: the term *system bus* is a bit misleading. It's really a mesh of connections between system components allowing multiple masters accesing various slaves at the same time. This makes the system faster, but another effect of this approach is that each master has its own address space.

Currently the system has three masters: the CPU and both DMA controllers. While the CPU has access to all peripherals, the DMA controllers can access only the DDR3 RAM and the descriptor memory.

The current address map is given in Table 2-10. But attention: any major change into the QSYS (like changing memory sizes, adding/removing slaves) will cause a need to re-build this map. Luckily, on the software side, all devices are called by specific defines, which are automatically updated after each update to the QSYS.

Device	Port	cpu		msgdma_rx			msgdma_tx		
		data_master	instruction_ master	mm_write	descriptor_ read_master	descriptor_ write_master	mm_read	descriptor_ read_master	descriptor_ write_master
button_pio	s1	0x18123580							
cpu	debug_mem_ slave	0x18122800	0x18122800						
ddr3_ram	avl	0x08000000	0x08000000	0x08000000			0x08000000		
debug_uart	s1	0x18123420							
descriptor_ memory	s1	0x18120000			0x18120000	0x18120000		0x18120000	0x18120000
enet_pll	pll_slave								
eth_tse	control_port	0x18123000							
ext_flash	avl_csr	0x18123540							
	avl_mem	0x14000000	0x14000000						
frame_timer	s1	0x18123400							
msgdma_rx	csr	0x18123500							
	prefetcher_csr	0x18123480							
msgdma_tx	csr	0x18123520							
	prefetcher_csr	0x181234a0							
onchip_flash	data	0x18080000	0x18080000						
	csr	0x18123590							
output_pio	s1	0x18123440							
sensor_ interface	csr	0x181234c0							
sys_clk_timer	s1	0x18123460							
sysid	control_slave	0x18123598							
udp_generator	csr	0x181234e0							

Table 2-10 Memory map of the QSYS

2.5.4. Custom code

Thanks to using QSYS, the amount of custom Verilog code has been minimized. Still, there are some blocks which were coded from scratch.

2.5.4.1. Sensor interface

The sensor interface was designed as a QSYS component and is responsible for interfacing sensor and ADC signals, as well as driving these components, collecting data and packing them into frames together with headers and synchronization data. The output from the interface has a form of an Avalon-ST link. The module is configured using a number of registers accessible via an Avalon-MM interface.

The sensor interface itself consists of a number of sub-blocks as shown in Fig. 2-26.

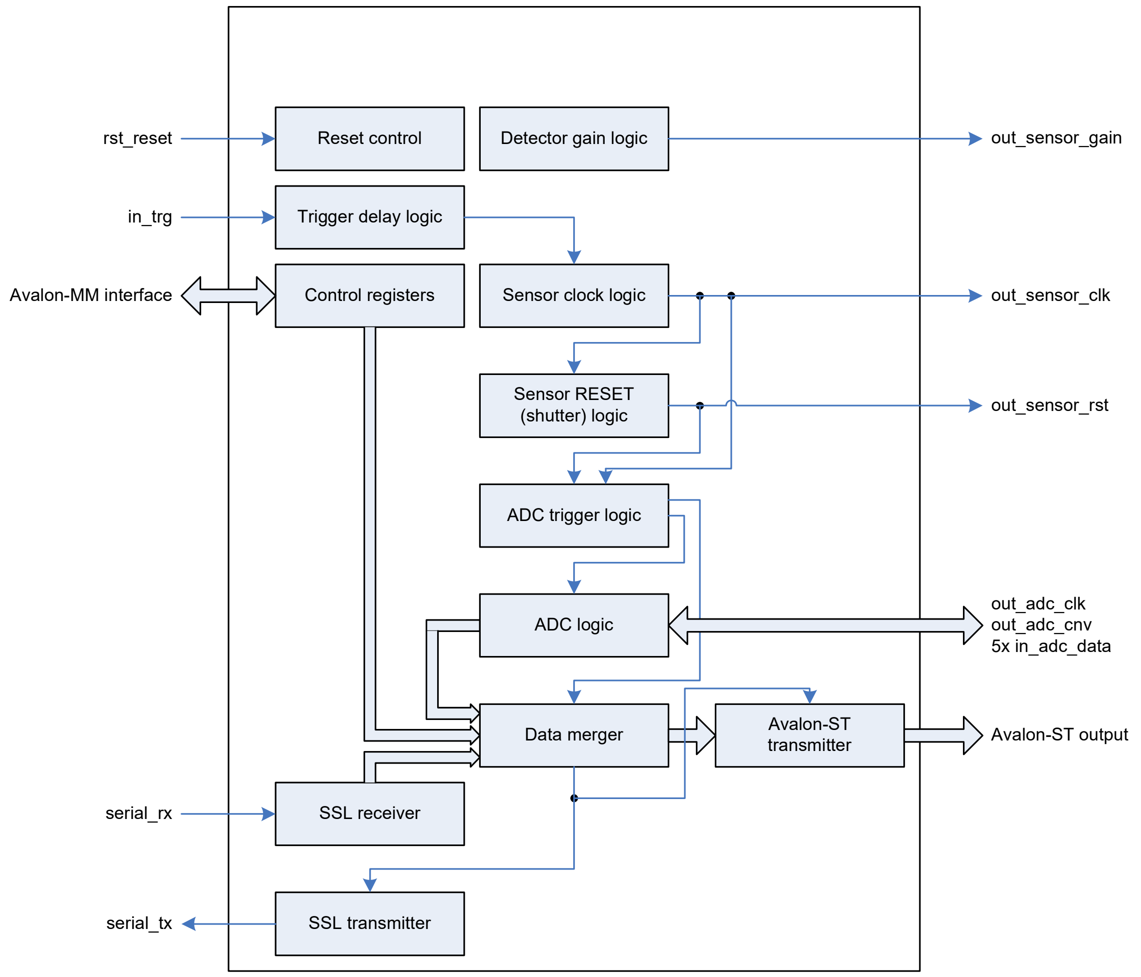


Fig. 2-26 Simplified block diagram of the sensor interface. Only most important data and trigger paths have been shown

Fig. 2-27 shows a *simplified, inaccurate and in general false* timing scheme of the sensor interface.

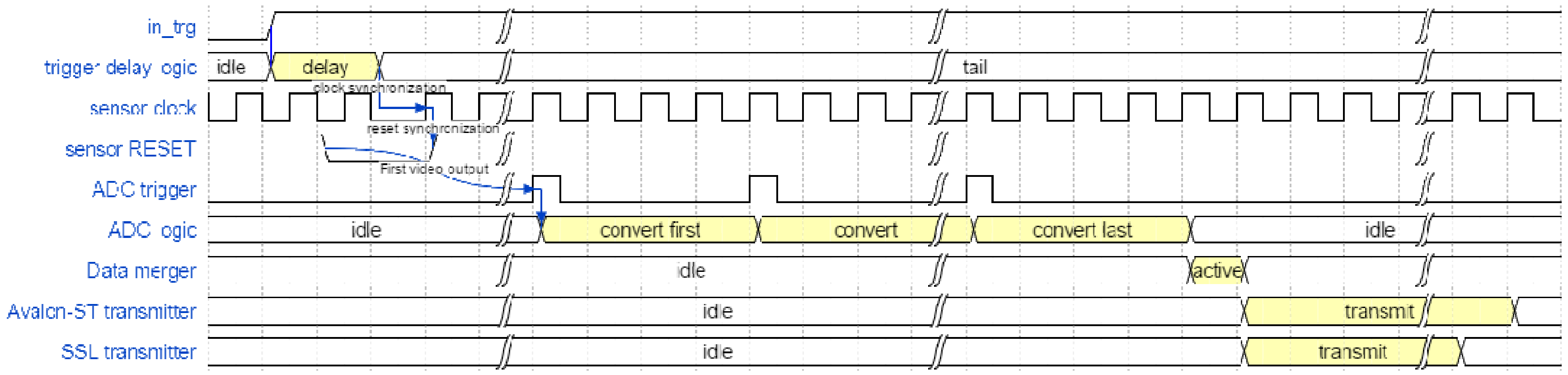


Fig. 2-27 A simplified timing scheme of the sensor interface

It's the case of long integration time (keeping RESET low as short as possible). For shorter integration times, the low-to-high RESET transition may appear within the acquisition phase (ADC convert). It's an allowed situation. It's also allowed that early activity connected to integration or conversion overlaps with data transmission for the previous cycle. The sensor data is double-buffered and buffer copy is realized during the data merger active state.

The following detailed information can be useful to understand the Verilog code. Almost the same is put as comments directly into the Verilog file.

A bank of **control registers** includes 4 32-bit R/W registers. They are used for setting up the module and checking its state. Register 0 is a bit different: when read, bits 15-8 (the status byte) are taken directly from device internals, not from register contents. The bus can be accessed 32-, 16- or 8-bit-wise.

The following tables summarize the register map of this block and details of the command and status registers.

Reg.	Bits	Name	Description
0	7-0	SENSOR_REG_COMMAND	Control byte
	15-8	SENSOR_REG_STATUS	Status byte
	23-16	SENSOR_REG_SENSORCLK	divider for producing sensor clock ('4' MHz) 6 of 8 bits used
	31-24	SESNOR_REG_ADCCNV	time of conversion pulse in ADC clocks should be > 500 ns
1	15-0	SENSOR_REG_DELAY	reset signal delay in master clock ticks 12 of 16 bits used
	31-16	SENSOR_REG_SHUTTER	integration time in sensor clocks 12 of 16 bits used
2	7-0	SENSOR_REG_SERSPEED	synchro baudrate divider set 50 for 1 Mbps
	15-8	SENSOR_REG_HEADER_ ANYDATA	user data transmitted with SMA state 8 bits SMA + 8 bits anydata
	31-16	SENSOR_REG_HEADER_ CMD	command field of the command header transmitted in packet must be 0x8000
4	15-0	SENSOR_REG RESERVED	Reserved

Table 2-11 Register map of the sensor interface

Bitmask	Name	Description
0x01	SENSOR_CSR_EN_BITMASK	enable operation
0x02	SENSOR_CSR_GAIN_BITMASK	gain selection
0x04	SENSOR_CSR_ADCK_BITMASK	ADC clock divider on/off
0x08	SENSOR_CSR_RESET_BITMASK	Reset all logic

Table 2-12 Command byte bitmasks for the sensor interface

Bitmask	Name	Description
0x01	SENSOR_STATUS_SEND	Sending data over Avalon-ST
0x02	SENSOR_STATUS_TRG_WAITING	The frame trigger has come and is being delayed now
0x04	SENSOR_STATUS_RESET_ACTIVE	The RESET (integration) signal for the sensor is active now
0x08	SENSOR_STATUS_ADC_ACTIVE	The ADC is converting data (signal high over all 64 channels) or just finished and waits for RESET high
0x10	SESNOR_STATUS_ADC_FINISHED	The ADC waits for RESET high
0x20	SESNOR_STATUS_TX_ACTIVE	Synchronization serial link is sending
0x40	SESNOR_STATUS_RX_ACTIVE	Synchronization serial link is receiving

Table 2-13 Status byte bitmasks for the sensor interface

The **reset control** allows putting the block in a known state. There are two types of reset:

– hardware reset resets everything

– software reset resets everything but registers (and their logic).

The **detector gain logic** is responsible for controlling the gain setting signal for the sensor.

The **trigger delay logic** allows for fine synchronization among multiple boards (see chapter 29). A 12-bit downcounter allows for delaying incoming trigger's rising edge by up to 4095 50MHz clock cycles, i.e. 81.9 us.

Note: A further delay of one sensor clock will be added in the sensor clock generation logic.

The **sensor clock logic** generates a clock of max. 4 MHz, which is needed by sensors. The actual clock speed is configurable by a register with a formula: $F_s = F_i/2/(\text{div}+1)$, where F_s is the sensor clock, F*i* is input clock (50 MHz master clock nom.) and div is a divider taken from the register. A 5-bit downcounter allows for frequencies from 0.39 MHz (div=63) to 12.5 MHz (div=1). The nominal setting is 6, giving 3.5714285 MHz.

Important: The sensor clock is synchronized to the rising edge of the (delayed) trigger input, so that there is a positive slope at exactly one sensor clock period after the trigger. This positive slope is then used to synchronize integration start signal (sensor reset or shutter). The synchronization is done so that one of clock states (0 or 1) gets extended, but never shortened.

Note: 4 MHz is also the top speed accepted by the ADC module.

The **sensor reset (shutter) logic** block generates a 'reset' (shutter) signal for the sensor. It defines the integration time and is programmable by means of a register in sensor clock units. The leading (positive) slope of this signal is synchronized with the first positive slope of the sensor clock after the trigger. This reset-clock alignment is required by sensor’s specification.

An internal 12-bit downcounter allows for a max. integration time of ca 1.15 ms. It can be further extended by slowing down the sensor clock. It's user's responsibility to check if the integration period fits into the trigger period. If this requirement is not fulfilled, some triggers will be skipped.

Important remark from sensor's specification: Rise of a RESET pulse must be set outside the video output period. This must by guaranteed by user by means of proper timing. First video comes 18 cycles after negative slope of sensor reset and is 2 cycles long. Then 2 cycles pause (here we can come with the positive reset slope), then next video and so on, every 4 cycles.

The minimum duration of reset low is 21 clocks, and 20 clocks for reset high. Our logic requires that new reset low comes only after all channels are read.

The **ADC trigger logic** produces triggering signals for ADC framework and cares about counting ADC samples. It doesn't rely on incoming Trig signals supplied by sensors, it just generates their ‘mirror’ internally by counting clocks.

The **ADC logic** gives an SPI interface for ADCs. It triggers conversion and reads out the data on each incoming ADC trigger signal from the previous module.

The ADC framework and SPI runs with full master clock frequency or with half of it, depending on register setting. (For our nominal throughput of 10 000 frames per second, only full clock frequency is applicable.)

The converter works in the 'CS Mode, 3-Wire Without Busy Indicator Serial Interface' [6].

The CNV signal length is defined by register and expressed in ADC clocks. It's user's responsibility to ensure that the CNV pulse is longer than 500 ns (required by ADC spec).

The task of the **data merger** is to put together all needed data (from ADC, serial synchronization link and registers) and create a packet from them. See the following table for the packet structure.

Part	Word	Octet	Field name	Value	Remarks
Command header	0	0	command_header.marker	0x5555	See chapter 2.6.4, Table 2-20 and Table 2-21
		1			
		2	command_header.command	Reg2[31:16]	
		3			
	1	4	command_header.length	323	
		5			
Synchronization frame		6	sync_frame.local_ctr	local_sync_ctr	Local synchronization counter
		7			
	2	8	global_sync_error	global_sync_error	Error of serial receiver (bit 25)
		9	sync_frame.global_ctr	global_sync_ctr	Global synchronization counter
		10	sync_frame.sma_state[15:8]	Reg2[15:8]	ANYDATA see Table 2-11
		11	sync_frame.sma_state[7:0]	PMODA	External input (SMA)
Sensor data	3-162	...	frame_buffer		ADC data Two octets per sample

Table 2-14 Output packet format for the sensor interface

The **Avalon-ST transmitter** is used to transmit collected sensor data together with sync frame. The data can be later packed into UDP by UDP generator and sent over Ethernet. (Or whatever user wants.) The transmitter has 8-bit symbol, 4 symbols per beat (see Avalon-ST specification to understend these odd names). It's backpressurizable and includes packet signals. The Empty signal is dummy (always zero), as the data is always aligned to 32-bit size.

The data packet produced by the sensor interface consists basically of three blocks:

- the command header to comply with general packet specification (see chapter 2.6.4)
- the synchronization frame, which contains all the information needed for synchronizing packet from different boards, as well as synchronizing with external systems
- sensor data containing 320 16-bit words for total of 320 sensor channels.

The **SSL transmitter** and **SSL receiver** are responsible for frame-level synchronization. The transmission is getting triggered by an internal trigger (same as for the Avalon-ST transmitter). The transmitter sends 9 LSBs of the local synchronization counter with programmed baudrate.

The default baudrate is 1 Mbps, which needs a setting of 50 for master clock of 50MHz.

The receiver puts freshly received data into a register to be used by data merger. It does not need any triggering.

2.5.4.2. UDP generator

The task for the UDP generator is collecting frames from the sensor interface and packing them into valid UDP/IP packets. It uses Avalon-ST links as data input and output. The configuration is done by means of an Avalon-MM interface and a set of registers.

The generated packet can be passed directly to the TSE MAC (Ethernet) IP core.

The following tables summarize the register map of this block.

Reg.	Bits	Name	Description
0	7-0	UDPGEN_REG_CSR	Control byte
	15-8		Status byte
	31-16	UDPGEN_REG_SIZE	Payload size in 32-bit words
1	31-0	UDPGEN_REG_SRCIP	Source IP, last octet first
2	31-0	UDPGEN_REG_DSTIP	Destination IP, last octet first
3	15-0	UDPGEN_REG_DSTPORT	Destination port
	31-16	UDPGEN_REG_SRCPORT	Source port
4	31-0	UDPGEN_REG_DSTMAC	Destination MAC address, last octet first
5	15-0		
6	31-0	UDPGEN_REG_RES1	Reserved
7	31-0	UDPGEN_REG_RES2	Reserved

Table 2-15 Register map of the UDP generator

Bitmask	Name	Description
0x01	transfer enable	Enable operation

Table 2-16 Control byte bitmasks for the UDP generator

Bitmask	Name	Description
0x07	state of the machine	State of the internal state machine 0: Idle 1-4: sending MAC header 5-9: sending IP header 10-11: sending UDP header 12: sending data 13: padding too short packet 14: dumping too long packet 15: finished transfer

Table 2-17 Status byte bitmasks for the UDP generator

The incoming and outgoing data must be packetized (packet signals are required for both Avalon-ST links). Any incoming packet will be converted to one outgoing packet.

The UDP generator doesn't buffer incoming data: it produces the UDP packet in real-time as the data is coming. This implies three important facts:

- The packet size must be known *a priori*. It's needed to fill various length fields in the packet header. It means also, that *exactly* the declared number of octets *must* be transmitted. Therefore, each packet longer than declared will be truncated. Any shorter packet will be padded with zeros. It's not a real problem for us, as we exactly know the size of ou packets.
- The incoming data streem must be *backpressed* for the time of preparing and sending packet header. This means, that the data source must support backpressure.
- The UDP checksum can not be used as it can be calculated only *after* transferring the payload, but it's needed *before*. Anyway, calculating the UDP checksum is not required by the standard, so the packet will be fully correct. Further, it's not really needed as there's another checksum added by Ethernet MAC in the MAC layer.

The structure of an outgoing UDP packet is shown below.

Part	Word	Octet	Field name	Value	Remarks
	0	0	Blank	0x0000	Used for header alignment for TSE MAC (Ethernet)
		1			
MAC		2	DSTMAC	Reg5[15:0]	
		3			
	1	4		Reg4[31:0]	
		5			
		6			
		7			
	2	8	SRCMAC	0	SRCMAC will be filled by TSE MAC IP core
		9			
		10			
		11			
	3	12			
		13			
		14	EtherType/length	0x0800	IP v. 4
		15			
IP v. 4	4	16	Version/header length	0x45	IP v.4, length: 5 words
		17	DSCP/ECN	0x0000	
		18	Total length	4*Reg0[31:16] + 28	Reg0[31:16] is the declared payload size in words
		19			
	5	20	Identification	ip_id	A unique identifier of a packet. Incremented after each packet.
		21			
		22	Flags/fragment offset	0x4000	Don't fragment flag set
		23			
	6	24	Time to live	0x80	TTL=128
		25	Protocol	0x11	UDP
		26	Header checksum	header_checksum	See description
		27			
	7	28	SRCIP	Reg1	
		29			
		30			
		31			
	8	32	DSTIP	Reg2	
		33			
		34			
		35			
UDP	9	36	SRCPORT	Reg3	
		37			
		38	DSTPORT		
		39			
	10	40	UDP length	4*Reg0[31:16] + 8	Reg0[31:16] is the declared payload size in words
		41			
		42	UDP checksum	0	UDP checksum is not used. This is allowed by the standard.
		43			
			DATA		The amount of data is always a multiple of 32 bits.

Table 2-18 Complete UDP packet structure

2.5.4.3. Debouncers

Two signal ***debouncers*** are used in the project: one for the incoming synchro trigger signal and another one for the receive data of synchro serial port. Their task is to remove glitches from the signal, which might arise due to external interference. It also helps dealing with long signal slopes. It's quite important as neither the trigger circuit nor the serial receiver are glitch-proof.

A simplified diagram of a debouncer is shown below.

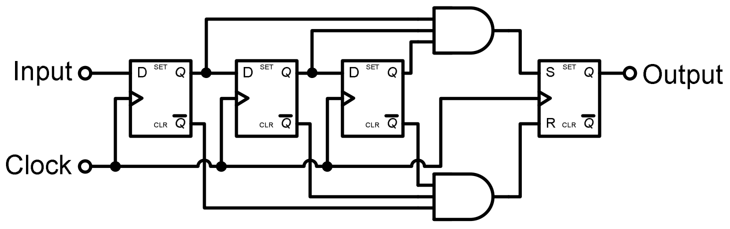


Fig. 2-28 Debouncer design (simplified)

Each debouncer incorporates a chain of D flip-flops (the number of these flip-flops is configurable; the drawing shows 3 of them) and an output circuit which changes its state only if all flip-flops show the same output state. This means, that only pulses longer than N clock periods, where N is the number of flip-flops, will pass through the circuit. A 50MHz master clock is used to drive these circuits.

It's also important to mention that a debouncer adds some delay to the signals. It's especially non-negligible for the trigger signal.

2.5.4.4. Test benches

For testing the VHDL components, a number of test benches have been prepared. hey are stored in the project folder and may be used if a change of any of the custom components has to be made.

Component	Test bench
debouncer.v	debouncer_testbench.v
sensor_interface.v	sensor_testbench.v
udp_generator.v	udp_testbench.v

Table 2-19 Summary of Verilog test benches

2.5.5. Additional information

As the project uses some high speed buses, proper time constraining is crucial for its operation. Especially, the Ethernet PHY connection is running a 125 MHz DDR interface (RGMII), which needs a very precise timing.

It has been found that original timing constraints for this interface are false and the device works only by means of luck. These constraints have been changed so that the system is functional after each re-compilation (so far), but there's no warranty that it will be always like this. If there are any problems with Ethernet connectivity for future FPGA compilations (especially with the RX path), try tuning these constraints. The most probable way is further increasing `Td_max` and `Td_min` parameters in `[fpga]/rgmii_sdc/rgmii_input.sdc`.

Below a code snippet from this file is shown, which is responsible for RGMII receive path timing.

```
#set Tco_max 0.350                 Changed M.D. 2019.10.07
#set Tco_min -0.350
set Tco_max 0.9
set Tco_min -1.3

#set Td_max 1.0 Changed M.D. 2019.10.07
#set Td_min 0.9
set Td_max 1.6
set Td_min 1.5

set longest_src_clk 0.0
set shortest_src_clk 0.0
set longest_dest_clk 1.0
set shortest_dest_clk 0.9
```

Another high-speed device, the DDR3 RAM, seems to work flawlessly. The on-board QSPI Flash has not been tested.

2.6. Internal processor firmware

2.6.1. General

The main task of the firmware is to control the on-chip hardware modules. It doesn't take part in data acquisition, as this is performed solely by hardware components.

The firmware uses a single TCP/IP connection for programming the device, while data transfer is done via UDP/IP.

The firmware project was derived from Intel's '*Simple socket server*' example and deeply modified. It's based on Micrium's *MicroCOS II* real-time kernel and uses *NicheStack* from Interniche to add network functionality. The original socket server code has been completely rewritten to improve it's stability, allow for multiple connections and clean up the totally odd Intel's code.

On top of it, a simple command interpreter has been built. In general, it's fully compatible with version 1 of the device, however some commands (mainly slow-control) are not implemented.

Important remark: The system uses **Booting method 2**[3]. It means, that the program code is stored in the internal Flash memory of the FPGA, but it's loaded to RAM (in this case, the external DDRAM) by a special bootloader program and executed from there. It allows for faster execution than directly from Flash. Anyway, working without external DDRAM is virtually impossible due to huge apetite of MicroCOS for data memory.

Remark for NIOSII-beginners: The firmware project is internally divided into two projects:

- the 'right' project is what is described here;
- the ***BSP project*** is an automatically generated set of libraries which is derived from the configuration of the QSYS. It must be re-generated after each QSYS change (otherwise the main project won't compile'). There are some files excluded from

automatic re-generation, as they were edited to render the project working and must stay in this state.

2.6.2. Socket server

The socket server is responsible for assigning sockets, handling incoming connections and socket data transmission. It's able to handle multiple listener sockets on different ports, however only one socket is currently used.

If a client connects to a port, a communication socket is being assigned for him. If another client wants to connect to the same port, the old client gets disconnected. This behavior looks very odd and nasty, but it gives a simple method of killing lost connections.

The socket server offers a simple programmer's interface consisting of three functions:

- `ethernet_listen()` for initializing listeners;
- `ethernet_read()` to read data from a socket
- `ethernet_write()` to write data to a socket.
- `ehernet_close()` to force connection closing.

Currently, `ethernet_listen()` can be called only once for each listener socket, so a socket can be configured only once and can not be *unlistened*.

All these functions are thread-safe and can be used from any user's thread. However, this thread must be created as Niche-compatible. It's described in the Niche Stack documentation [4], how to do it.

The socket server uses its own thread for handling incoming connections.

For details, look directly into the source code:

`[firmware]/hit20_v3/src/socket_server.c`

`[firmware]/hit20_v3/inc/socket_server.h`

Important – the base IP address of the device is defined in `socket_server.h`. Another important define there is `NR_CHANNELS`. It tells how many different port numbers can be used. If another TCP channel is needed in the future, this define must be adapted. See the snippet from `socket_server.h` below.

Please note that, even if the DHCP can be enabled by switch, it doesn't make any sense, as there is no reliable way of finding the DHCP-assigned address of the device (unless the DHCP server is configured to give certain 'static' IP addresses to given MAC). Also, the PC software has no support for dynamic addresses. Therefore, static addressing should be always used.

```
#define NR_CHANNELS 1               //number of listening sockets -
as in Wiznet

                                    //Each socket listens on its own
port and is able to open one "talking" connection at a time
                                                            //If
a new connection request comes, the old one gets preempted. This
allows us killing dead connections.


/*
 * If DHCP will not be used, select valid static _BASE_ IP addresses
here:
 *  The contents of DIPSW[3:0] will be added to the last byte of the
IP.
 *  DIPSW[4] is used to enable/disable DHCP.
 */
#define IPADDR0   10
#define IPADDR1   0
#define IPADDR2   7
#define IPADDR3   16

#define GWADDR0   10
#define GWADDR1   0
#define GWADDR2   7
#define GWADDR3   1

#define MSKADDR0  255
#define MSKADDR1  255
#define MSKADDR2  255
#define MSKADDR3  0
```

2.6.3. Control task

The control task is really where all the action takes place. So, it:

– reads data from Ethernet

– interprets it and does some actions

– sends back replies

All the rest of the code can be treated as a 'miraculous background' which is only responsible for handling read and write functions.

The main loop of the control task (which was derived from version 1) continuously attempts to receive packet headers, interpret them, receive data, if any, then take some actions and generate an answer. It uses HAL-layer code to interface to the hardware.

Connecting and disconnecting client(s) is done in background by the socket server itself. The control task doesn't need to control it.

For details, look directly into the source code:

`[firmware]/hit20_v3/src/control.c`

`[firmware]/hit20_v3/inc/control.h`

2.6.4. TCP control protocol

Table 2-21 summarizes all TCP commands, as listed in `[firmware]/hit20_v3/inc/dev_commands.h`

The general packet structure (being payload for a TCP packet) is shown in Table 2-20.

All fields are transmitted as **16-bit words with LSB first** due to historical grounds. As both the NIOS processor and x86 architecture are big endian, the data must be converted on both sides of the TCP link.

The offset and length values in the following tables are expressed in 16-bit units (one must multiply them by two to get byte values).

Nr	Offset	Length	Field name	Description
1	0	1	Marker	Must be 0x5555
2	1	1	Command	Command code
3	2	1	Length	Data length
4	3	= Length	Data	Packet data

Table 2-20 TCP packet structure

The packet format is the same for both transmission directions. The device always answers with the same command code as the request. There are no special error codes. In case of an error in the incoming packet, the device will not reply.

The UDP transmission channel uses the same internal packet structure (command code 0x8000 for sending data) as the TCP link.

Command	Description	Code	PC → device		device → PC	
			Len	Data	Len	Data
COMMAND_PING	Return the same	0x0001	0	[]	0	[]
COMMAND_DEBUG_LED_OFF	Turn off LED 0	0x0010	0	[]	0	[]
COMMAND_DEBUG_LED_ON	Turn on LED 0	0x0011	0	[]	0	[]
COMMAND_LEDS_DISABLE	Disable LED4 blinking. Other LEDs must be explicitly switched off.	0x0110	0	[]	0	[]
COMMAND_LEDS_ENABLE	Enable LED4 blinking.	0x0111	0	[]	0	[]
COMMAND_TRIGGER_DISABLE	Disable trigger generation in master mode	0x0210	0	[]	0	[]
COMMAND_TRIGGER_ENABLE	Enable trigger generation in master mode	0x0211	0	[]	0	[]
COMMAND_TRIGGER_SET_SLAVE	Set trigger to slave mode	0x0220	0	[]	0	[]
COMMAND_TRIGGER_SET_MASTER	Set trigger to master mode	0x0221	0	[]	0	[]
COMMAND_TRIGGER_SET_PERIOD	Set trigger period for master mode in master clock ticks	0x0230	1	[Period_ticks]	0	[]
COMMAND_TRIGGER_SET_TINT	Set integration time in sensor clock ticks	0x0240	1	[Tint_ticks]	0	[]
COMMAND_SET_GAIN	Set sensor gain (low/high	0x0250	1	[Gain]	0	[]
COMMAND_TRIGGER_SET_MASTER_DELAY	Set trigger delay time in master clock ticks for master mode	0x0260	1	[Tdelay_ticks]	0	[]
COMMAND_TRIGGER_SET_SLAVE_DELAY	Set trigger delay time in master clock ticks for slave mode	0x0270	1	[Tdelay_ticks]	0	[]
COMMAND_DAQ_DISABLE	Disable sending data	0x0310	0	[]	0	[]
COMMAND_DAQ_ENABLE	Enable sending data	0x0311	0	[]	0	[]
COMMAND_DAQ_RESET_COUNTERS	Reset synchronization counters	0x0321	0	[]	0	[]
COMMAND_DAQ_FLUSH_DATA	Send all remaining data over data socket (applies only to v.1)	0x0322	0	[]	0	[]

COMMAND_DAQ_CONFIG_PEER	Set connection settings (peer IP and port) for data transfer Warning: IP is sent as 4 shorts with MSB=0!	0x0331	5	[ip ip ip ip port]	0	[]
COMMAND_SLOWCTRL_SNAPSHOT	Slow control snapshot - read all channels of ADC (applies only to v.1)	0x0410	0	[]	10	[Readout of 5 ADC channels as 32-bit integers]
COMMAND_DATA_TRANSFER	Transfer data frame UDP packet!	0x8000	-	-	646	[Single data frame]

Table 2-21 TCP command summary

2.6.5. HAL for custom components

The hardware abstraction layer (HAL) functions for custom components have been collected in following files:

- `[firmware]/hit20_v3/inc/sensor.h` and `.../sensor.c` for sensor interface support
- `[firmware]/hit20_v3/inc/udpgen.h` and `.../udpgen.c` for UDP generator
- `[firmware]/hit20_v3/inc/utils.h` and `.../utils.c` for trigger routing, frame clock generation and LEDs.

The HAL consists mainly of simple wrapper functions to access hardware registers of specific blocks. `Utils` contains also some generic helper functions.

Look at the source code for details.

2.7. Brief description of v.1

2.7.1. Comparison to v.2

The v.1 is not being developed anymore, but it can still be used in various setups together with v.2. Most important differences between v.1 and v.2 are:

- v.1 supports only two sensors while v.2 supports five;
- v.1 is entirely based on a microcontroller (STM32F446) and its built-in peripherals while v.2 is based on FPGA;
- it uses 100Mbps Ethernet adapter (based on Wiznet chip) rather than 1 Gbps; it's enough for 2 sensors per board (it's *theoretically* enough even for five sensors).
- its main clock frequency is different. This implies differences in setting frame frequency, integration time and delays. Therefore the PC software has separate timing settings for v.1 and v.2. It's user's responsibility to set them properly;
- it supports packing output UDP packets into bunches. This allows for better bandwidth utilization (more big packets instead of many small ones give less overhead from packet headers). V.2 hasn't got this feature as a single data frame from 5 sensors is already close to maximum allowed by Ethernet. Anyway, the feature of packing is not supported by PC software, so it has never been used;
- power supplies are different.
- v.1 is equipped with an analog 'dosimetry' output reflecting the total current of all photodiodes. In v.2 this feature was cut out.

There are also some very important aspects where both versions are identical:

- they share the same standard of synchronization interface;
- they share the same TCP control protocol and the same UDP frame format (besides the packet size).

These features make it quite easy to mix both versions in a single measurement setup.

2.7.2. Connectors and controls

A general view of a v.1 board (with mounting plate) is shown below.

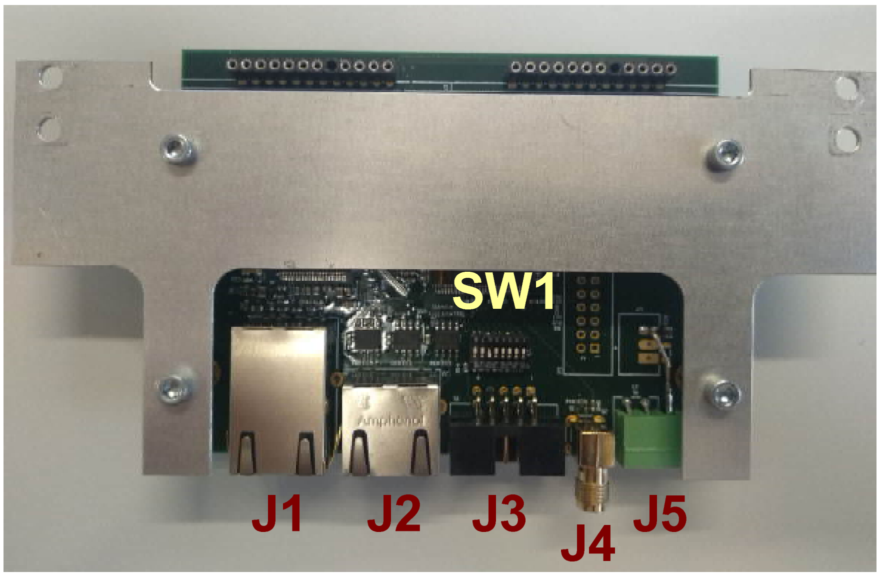


Table 2-22 Connectors and controls of v.1

J1: Ethernet connector: RJ-45

Standard 100Base-TX connector.

J2: Synchronization connector: RJ-45 – compatible to J1 in v.2 (see chapter 2.3.7.3: *Sockets and connectors*), but includes also analog dosimetry output.

Nr	Name	Function	Remarks
1	LINK3B	Frame clock	Output for master configuration, input for slave
2	LINK3A		
3	LINK2A	Synchronization serial link	Output for master configuration, input for slave
4	LINK0B	unused	
5	LINK0A		
6	LINK2B	Synchronization serial link	Output for master configuration, input for slave
7	LINK1B	Dosimetry output	Output of dosimetry signal – analog, differential
8	LINK1A		

Table 2-23 Synchronization connector pinout of v.1

J3: JTAG connector: IDC-10

This connector is used for programming and debugging the microcontroller.

J4: External synchronization connector: SMA

This is exactly what is called the 'SMA connector' throughout this document.

J5: Power connector: 3.5 mm 3-pin EDG connector

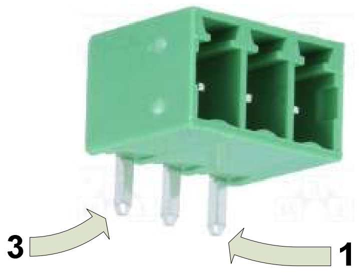


Fig. 2-29 V.1 power plug pin numbering

Nr	Function
1	+5V
2	GND
3	+9V

Table 2-24 Power connector pinout for v.1

SW1: 8-bit dip-switch which defines the last byte of board's IP address. The general IP pattern is the same as for v.2: 10.0.7.x.

2.8. External hardware components

Only a single board can be run without any external components (OK, still needs a power supply!). All more sophisticated configurations need at least two further components:

- the synchro board,
- an ethernet switch.

If synchronization to the HIT's Ethercat system is needed, then the timestamper is required.

2.8.1. Synchro board

The synchro(nization) board is responsible for routing synchronization signals among boards in a multi-board configuration (see chapter 2.4: *Details on synchronization* and Fig. 2-21).

Currently it's made as a prototype board with one master and three slave connections. The master socket is marked (see Fig. 2-30 below). It must be connected to a board configured as master (while the others must be configured as slaves, that's clear), otherwise the system won't work. Wrong configuration is inoperable, but not dangerous, i.e. burning RS-485 transmitters is not probable even if there's a bus conflict.

Additionally, the synchro board is used for power supply routing for v.1 boards. There are three power inputs, as shown in the table below:

Voltage	**Description**	**Power**		**Ground**	
		cable	**plug**	**cable**	**plug**
+3.3V	DAQ digital supply	red	red	blue	blue
+9V	DAQ analog supply	yellow	yellow	blue	blue
+5V	synchro board supply	yellow	green	blue	blue

Table 2-25 Power supply for the synchro oard

The output cables for DAQ bouards are equipped with three-way plugs, and they can be connected only in one (proper) way.

V.2 board(s) must be supplied independently of the synchro board. See chapter 2.3.7.3: *Sockets and connectors* and 4.1: *Hardware preparation* for details).

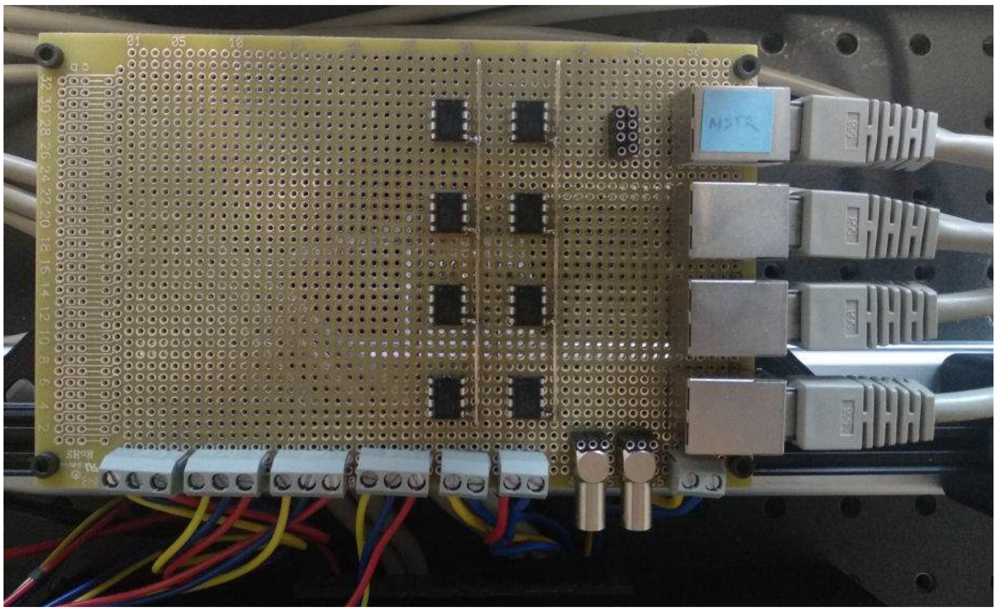


Fig. 2-30 The synchro board

The master connector has eight test pints (in form of a 2.54mm header) which allow direct observation of signals with an oscilloscope. The two Lemo connectors can be connected to any signal for easy routing signals to the scope.

2.8.2. Timestamper

The timestamper is a simple device to synchronize our system with an external one, e.g. the HIT's Ethercat. The idea is to put a pseudo-random digital signal to both readouts. Then, this signal can be used for off-line data synchronization.

So, this *pseudo-random* signal is really current system timestamp of the controlling computer. It's transmitted over a serial port four times a second with a baud rate of 250 bps.

Physically, the timestamper consists of a stock USB-to-serial converter and a small board with digital buffers. The power for the converter is drawn parasitically from serial port's control lines and fed through a 5V regulator. The board has two outputs with different logic levels. These levels are nominally 5V and 3.3V, however, due to limited power output of the USB-to-serial converter, it's less (at least for the converter we use).

In critical cases, the power regulator can be bypassed by a jumper, however this must be done with care, as the power voltage for the logic must not exceed 5.5V.

If a true serial port is available in the PC, it can be used without problems. It's even better than a USB interface as it's expected to offer higher current on control pins and true +/-10V RS-232 voltage standard (which is hard to find in case of USB ports).

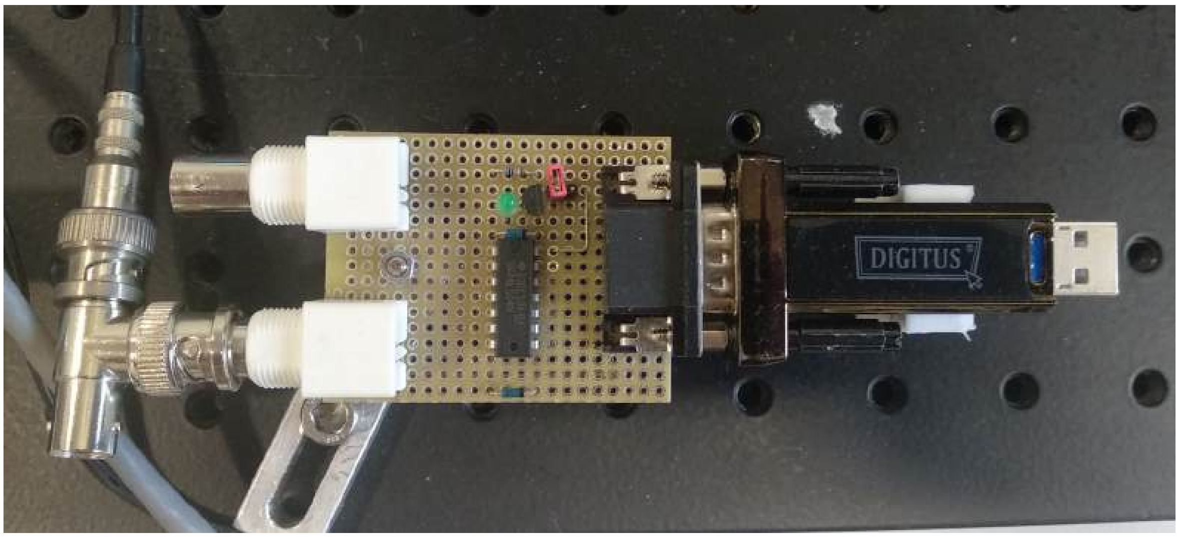


Fig. 2-31 General view of the timestamper

There's a special PC software which makes the timestamper work. See chapter 3.2: *Timestamper*.

2.8.3. Ethernet switch

The Ethernet switch is needed to connect multiple boards to a single adapter of the PC. In general, it should be a 1Gbps switch. Anyway, for limited number of channels, a 100Mbps switch should work as well. The expected data rate for a single sensor is 10 Mbps, so, *very theoretically*, up to 10 sensors can be served with a 100Mbps link.

Anyway, it's important that it's a switch and not a hub. Many collisions are expected when frame data are being sent (simultaneously from all the boards!). Since they are UDP packets and will never be retransmitted, it relies solely on the switch, that they are buffered, queued and flawlessly sent to the PC.

2.8.4. Power supplies

The v.2 electronics needs two PS channels to work, v.1 needs another two and the synchro board needs one. Typically, one 4-channel lab PS (Hameg HMP-4040) and one dedicated PS for FPGA is used. The Ethernet switch has its own power supply.

All PS channels are summarized below:

Nr	Voltage	Source	Destination	Typ. current
1	+5V	Lab PS (ch. 1)	Synchro board	270 mA
2	+3.3V	Lab PS (ch. 2)	v.1 digital	670 mA
3	+9V	Lab PS (ch. 3)	v.1 analog	290 mA
4	+12V	Lab PS (ch. 4)	v.2 frontend board	270 mA
5	+12V	dedicated PS	v.2 FPGA board	(no readout)

Table 2-26 Summary of power supply voltages

The typical currents are given for a setup containing three v.1 boards and one v.2 board.

Remarks concerning v.2 power supply:

- The system can be powered from single 12 power supply as well as with dual (see chapter 2.3.6: *Further remarks*)
- However, the power scheme has great influence on the noise figure. Single power supply has a way bigger noise due to influence of noisy FPGA on sensitive analog circuits.
- Best results are obtained when two physical power supplies are used (even not two channels of one PS!) Therefore, a lab PS can be used to supply the frontend board, while the FPGA board should be supplied from the attached PS.
- It's not known at this moment how the setup will behave when more than one v.2 boards are used. We have only one board.

3. PC software

The PC software can be divided into two groups: online and offline software. The online software are basically the DAQ and Timestamper programs. The offline software is everything used for data conversion, processing and analysis, as well as simulation. These are mainly various Matlab scripts.

3.1. DAQ software

3.1.1. General overview

The DAQ software allows for connecting to multiple DAQ boards, configuring them, reading frame data and storing them to hard disk. On top of it, it offers a real-time data monitor.

The software is also equipped with a number of diagnostic procedures which were used for debugging the v.1 electronics (these procedures have never been used with v.2, so it's not obvious, that they work with v.2).

The code is developed with QT 5 and compiled for Windows.

And **very important**: this software **is not** deeply maintained. Therefore is not idiot-proof and wrong settings (see chapter 4.2: *Setting up software*) may cause a crash. Also, some older test procedures may be useless for v.2 hardware or they may even crash.

3.1.2. Basic concepts

– The software allows connecting (theoretically) unlimited number of boards of both versions in any combination. They can be grouped in ***detector planes***. A plane is a purely virtual concept (doesn't need to be physically a single and entire plane) and it doesn't have any influence on how the data gets processed and stored. The only use of it is currently grouping data from various sensors into a single plot on the display.

– The data is stored in form of binary files. Prior to writing, data from all boards gets synchronized on frame level.

– Hardware configuration is stored using the QSettings mechanism. Ini files are used for this. In addition, for each data file, a copy of currently used ini file is created to keep track at which settings the data has been collected.

– The data is not deeply processed during acquisition. Basically, what comes from the hardware, gets written to the disk.

– Multi-threaded architecture increases overall throughput (by using multiple processor cores) and guarantees fast response of user interface.

3.1.3. Deeper into the code

The code can be found in `[pcsoft]`. Besides common QT structures (mainwindow etc.), there are some important classes to support communication with devices and data taking:

– `Device`: this is a wrapper class for a physical device. It allows connecting to hardware, configuring it and taking data. A helper class, `DeviceConfig`, is used to pass hardware configuration. Device objects are allocated dynamically upon system configuration.

– `DataReceiver`: it includes all the code needed for receiving data over UDP socket. Each data receiver uses its own thread which continuously receives and buffers data from a single board (either v.1 or v.2) The received frames are internally stored into cyclic buffers as objects of type `BufferData`. There is one data receiver per one device.

– `EventBuilder` is responsible for collecting data from cyclic buffers of each data receiver, merging them (with attention of frame-level synchronization) and writing to the disk. It uses a separate thread for its activity. In contrast to devices and data receivers, there is only one event builder in the system.

  An additional function of the event builder is filling amplitude histograms, which are used by some diagnostic procedures (used only for v.1).

– `DisplayServer` and `Display`: The display server is responsible for fetching some data frames (10 per second or whatever) and displaying them using multiple displays. There is only one display server, but the number of displays is equal to the number of defined detector planes (this is not equivalent to number of boards).

3.1.4. Output data format

The data is saved as a **stream of 16-bit words**. They are grouped into regular structures storing frames. Each frame contains a header and a set of data for every board. These sets of data contain sensor data as well as synchronization structures.

A structure of a single frame is shown below.

	Word offset	Symbol	Description
Header	0	N	Number of sensors
	1	C1	Board 1: number of channels
	(boards 2 to N-1)		
	N	CN	Board N: number of channels
Data	N+1	S1	Board 1: sync frame (8 words = 16 bytes)
	...		
	N+8		
	N+9	D1	Board 1: first sensor channel
	...		*(other sensor channels)*
	N+9+C1		Board 1: last sensor channel
	(boards 2 to N-1)		
	(X)+1	SN	Board N: sync frame
	...		
	(X)+8		
	(X)+9	DN	Board N: first sensor channel
	...		*(other sensor channels)*
	(X)+9+CN		Board N: last sensor channel

Table 3-1 Output file data format

The sync frame is described in the following table.

Word offset	Name	Description
0	local_ctr	Value of local frame counter (0-65535)
1	global_ctr	Value of global frame counter (0-511)
2	sma_state	State of the SMA/PMOD input
3	dummy	Nothing. This is to align following 32-bit values to 32-bit boundary.
4	device_nr	A unique number given to a device. It has minor to no meaning in data processing.
5		
6	data_ok	Diagnostic field. Frames with data_ok=0 should be skipped in offline processing.
7		

Table 3-2 Sync frame format

Important remark: The sensor signal has inverted polarity (baseline is close to maximum, i.e. 65535 or 0xFFFF while any signal decreases this value, see chapter 2.3.1: *(The Electronics) Overview*). It gets inverted by the software, so that intuitive 'non-Australian' values are written to the file.

3.2. Timestamper

The timestamper software is a simple program developed with QT5 for Windows. Its function and use are self-explaining. Below is a screenshot of its main and only window.

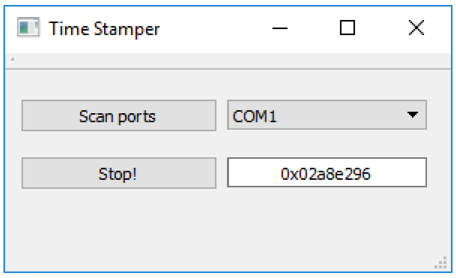


Fig. 3-1 Timestamper's main window

The program will send timestamps to selected serial port. These are just ordinary Unix timestamps: 32 bits, so 4 bytes are sent each time.

A quite important fact is that the exact time intervals between sent values are not guaranteed. Windows is not an RTOS. The good message is that has no meaning for synchronization purposes.

3.3. Offline software

The offline software is used for data post-procesing, including merging our detector's data with HIT's EtherCAT data. It's written mainly in Matlab (at least Michal's part). There is also a little c++ library to simplify loading data files into Root.

3.3.1. Matlab functions

The following paragraphs describe only most important functions. On top of them, there's a plenty of specific functions and scripts for processing data from various test sessions.

A remark on array sizes used here:

- Whereever **M** or **N** is used, it means <u>any number</u>. One M or N can be different from another M or N.
- Whereever **C** is used, it means number of channels in the data.
- Whereever **B** is used, it means number of boards.
- Whereever **F** is used, it means number of frames (samples).

3.3.1.1. load_data

```
function [data syncdata boardinfo] =
load_data
(filename, nr_boards, first_frame, nr_frames)
```

This is the basic function for reading binary files (`*.dat` and `*.da2`). It subsequently calls `load_data_v1()` or `load_data_v2()` functions depending on given file's extension.

Input arguments:

Name	Format	Default	Description
filename	1xN char	*required*	The name of the file to load with an extension (*.dat or *.da2)
nr_boards	1x1 double	0	Number of boards – important only for *.dat files. In other case optional and not used.
first_frame	1x1 double	0	First frame to read from file. Indexing starts from 0. Optional.
nr_frames	1x1 double	+inf	Biggest number of frames to read from file. +inf is legal. Optional.

Table 3-3 load_data: input arguments

Output arguments:

Name	Format	Description
data	CxF uint16	An array of all data as saved in the file. Dimensions: nr_channels x nr_samples.
syncdata	1x1 struct	A structure containing synchronization data. See table below.
boardinfo	1x1 struct	A structure containing information about number and version of boards. See table below.

Table 3-4 load_data: output arguments

Syncdata description:

Field name	Format	Description
local_ctr	BxF uint16	An array containing local frame counters for all boards and all samples. Dimensions: nr_boards x nr_samples.
global_ctr	BxF uint16	An array containing local frame counters for all boards and all samples. Dimensions: nr_boards x nr_samples.
sma_state	BxF uint16	An array containing the state of SMA/PMOD inputs for all boards and all samples. Dimensions: nr_boards x nr_samples.
device_nr	BxF uint16	An array containing device numbers for all boards and all samples (the device number gets added to each frame). Dimensions: nr_boards x nr_samples.
data_ok	BxF uint16	An array containing the state of the data_ok flag for all boards and all samples. Dimensions: nr_boards x nr_samples.

Table 3-5 load_data: syncdata structure description

Boardinfo description:

Field name	Format	Description
nr_boards	1x1 double	Number of boards in the system (B)
channel_counts	1xB uint16	A vector containing channel counts for all boards. Dimensions: 1 x nr_boards.

Table 3-6 load_data: boardinfo structure description

Remark: The loaded data may occupy a lot of memory, especially if it gets converted to double at some point. Therefore, user should consider loading files and processing data in smaller pieces (allowed by `first_frame` and `nr_frames` arguments). If only synchronization data is needed, the following function can be used.

3.3.1.2. load_sync_data

```
function [syncdata boardinfo] =
load_sync_data
(filename, nr_boards, first_frame, nr_frames)
```

Load only synchronization data from a given binary file. Allows loading synchronization data from big files with only moderate use of memory. It's helpful for synchronizing our data with HIT.

Input arguments:

See 3.3.1.1:

load_data, Table 3-3.

Output arguments:

See 3.3.1.1:

load_data, Table 3-4.

3.3.1.3. load_and_prepare_data

```
function [data_bl syncdata I_offspill I_onspill I_neutral boardinfo] =
load_and_prepare_data
(filename, nr_sensors, first_frame, nr_frames)
```

Load data, subtract baseline from each channel and find spills, offspills and transient states. This function uses `find_spills` function (see below), which can be used standalone.

Input arguments:

See 3.3.1.1:

load_data, Table 3-3.

Output arguments:

Name	Format	Description
data_bl	CxF double	An array of all data after substracting baseline for each channel. Dimensions: nr_channels x nr_samples.
syncdata	1x1 struct	A structure containing synchronization data. See Table 3-5.
I_offspill	1xN double	Indices of all frames qualified as off-spill.
I_onspill	1xN double	Indices of all frames qualified as on-spill. A quick method to see the spill. A quick method of viewing the spill structure is `hist(I_onspill,1000)` (1000, or any other big number, defines the resolution of our view).
I_neutral	1xN double	Indices of all frames qualified as transient state between on-spill and off-spill.

Table 3-7 load_and_prepare_data: output arguments

Remark: The algorithm used by this function needs that there is a beam-free 'warmup period' in the beginning, i.e. that the data begins with off-spill. This warmup time is must be at least 10000 frames long (1 second with nominal frame rate).

3.3.1.4. find_spills

```
function [I_offspill I_onspill I_neutral mask_offspill mask_onspill
mask_neutral] =
find_spills
(data, warmup_samples, drawplot)
```

Find spill structure in detector's data. The data must be baseline-aligned first.

Input arguments:

Name	Format	Default	Description
data	CxF double	*required*	An array of all data after substracting baseline for each channel. Dimensions: nr_channels x nr_samples.
warmup_samples	1x1 double	*required*	Number of samples for the warmup period. The warmup period must be off-spill.
drawplot	1x1 double	*required*	1 to plot data divided into on-spill, off-spills and transient states, 0 not to plot them.

Table 3-8 find_spills: input arguments

Output arguments:

Name	Format	Description
I_offspill	1xN double	Indices of all frames qualified as off-spill.
I_onspill	1xN double	Indices of all frames qualified as on-spill. A quick method to see the spill. A quick method of viewing the spill structure is `hist(I_onspill,1000)` (1000, or any other big number, defines the resolution of our view).
I_neutral	1xN double	Indices of all frames qualified as transient state between on-spill and off-spill.
mask_offspill	1xF double	A vector including value of 1 for each off-spill sample and 0 otherwise. Dimensions: 1 x nr_samples
mask_onspill	1xF double	A vector including value of 1 for each on-spill sample and 0 otherwise. Dimensions: 1 x nr_samples
mask_neutral	1xF double	A vector including value of 1 for each transient sample and 0 otherwise. Dimensions: 1 x nr_samples

Table 3-9 find_spills: output arguments

3.3.1.5. make_real_timestamp

```
function real_time =
make_real_timestamp
(sync_data, fs, threshold, fig_nr, subpl)
```

Calculate real timestamp for each frame based on data from timestamper. Timestamper bitrate is assumed to be 250 bps, 4 transmissions/s.

Input arguments:

Name	Format	Default	Description
sync_data	1xF double	*required*	A vector containing timestamper data, like `sync_data.sma_state` returned by `load_data()`. It can be a binary (boolean) or 'analog' signal.
fs	1x1 double	*required*	Sampling frequency (or frame rate) in Hz.
threshold	1x1 double	*required*	Threshold value for interpreting sync_data values. Set to 0.5 for boolean data.
fig_nr	1x1 double	*required*	Specify figure number to a plot report to. Set to 0 to disable plotting.
subpl	1x3 double or []	*required*	Specify position of the subplot in the figure as a 3-element vector (see Matlab's `subplot`) or empty array if subplots are not used.

Table 3-10 make_real_timestamp: input arguments

Output arguments:

Name	Format	Description
real_time	1xF double	Real timestamps for all frames. They are expressed in timestamper's units (milliseconds according to Unix timestamp), but with fractional resolution.

Table 3-11 make_real_timestamp: output arguments

The function currently uses linear interpolation to calculate timestamp values. It can be easily changed in the code. See help for Matlab's `interp1()` function.

3.3.1.6. export_all_timestamps

This **script** is to calculate and export all real timestamps (timestamper must be used!) for runs in given directory. It will ask user for the working directory, total number of boards and from which board the timestamper signal should be taken. The data will be exported as `*.csv` files.

3.3.1.7. recon

```
function [positions widths] =
recon
(data, recon_mode, recon_attributes, boardinfo)
```

This is a common, uniform interface for all reconstruction (i.e. beam finding) algorithms.

It divides the data sensor-by-sensor and then applies specified beam finding algorithm **for each sensor separately** by calling a specific function.

Input arguments:

Name	Format	Default	Description
data	CxF double	*required*	An array containing sensor data. Baseline must be subtracted first (see 3.3.1.3: *load_and_prepare_data*).
recon_mode	1xN char or 1x1 double	*required*	Reconstruction algorithm. Can be specified by name (char array) or identifier (double). See Table 3-14 for details.
recon_attributes	any	*[]*	A variable or structure containing algorithm-specific configuration.
boardinfo	1x1 struct	*[]*	Boardinfo structure as returned by `load_data()` (see 3.3.1.1: *load_data*). If not specified, it is assumed that all boards are v.1 (128 channels) and board count is derived from data size.

Table 3-12 recon: input arguments

Output arguments:

Name	Format	Description
positions	BxF double	Reconstructed beam positions (expressed in channels) for each board and each sample
widths	BxF double	Reconstructed beam widths (expressed in channels) for each board and each sample

Table 3-13 recon: output arguments

Remark regarding `boardinfo`: If there is a plane which contains more than one board, it is still possible to use `recon` as is, but a modified boardinfo structure must be provided, where all boards assigned to one plane are merged.

An example: Let's assume that we have four boards configured as two planes:

– Board 0, 128 channels, plane 0
– Board 1, 128 channels, plane 0
– Board 2, 128 channels, plane 1
– Board 2, 128 channels, plane 1

The boardinfo has the following structure:

```
boardinfo.nr_boards = 4
boardinfo.channel_counts = [128 128 128 128]
```

To merge boards in planes, use the following boardinfo

```
boardinfo.nr_boards = 2
boardinfo.channel_counts = [256 256]
```

Available reconstruction algorithms:

Name	Identifier	Description
gaussfit	1	Gaussian fitting
basic_moments	2	Basic center of gravity / variance
windowed_moments	3	Center of gravity / variance with iterative windowing
basic_phase	4	Phase information from first Fourier coefficient, no beam width calculation
fourier_phase	5	Phase information from more Fourier coefficients, no bram width calculation

Table 3-14 reconstruction algorithms

gaussfit: The beam profile is fitted by a gaussian function by a nonlinear least squares fit. To increase fit speed and reliability, the data gets divided into smaller blocks of frames. For each block, first the mean over all samples is calculated and fitted. Then, this fit is used as starting point for individual frame fitting. Block size can be configured using `recon_attributes`.

Field name	Format	Default	Description
block_size	1x1 double	1000	Default block size. Can be specified as +inf to treat all data as a single block.
blocks	Nx2 double	[]	Explicit boundaries for each reconstructed block. If this attribute is specified, block_size is disregarded. Gaps and block overlapping are allowed.

Table 3-15 gaussfit: recon_attributes

In general, small block sizes should be used if the beam position changes frequently, while large blocks are better if the beam position is rather stable.

If `blocks` attribute is used - output data will be aligned block-by-block – so if there are any gaps between blocks or they are overlapping, the indices of reconstructed points will not directly correspond to input data.

The gaussian fitting is used as a reference method. It can’t be used in reality due to huge computational effort, indeterminate calculation time and occasional inconvergence. In Matlab, it’s orders of magnitude slower than other methods.

basic_moments: Beam position is calculated as the center of gravity for all signals (i.e. mean/expected value if we treat the profile as a histogram). Beam width is derived from second central moment (variance). This method is very simple and fast, but it’s prone to common noise, especially f the beam is far from detector’s center.

This method hasn’t got any attributes to be configured.

windowed_moments: A more advanced algorithm, where basic moment calculation is iteratively repeated. After each iteration, the data gets windowed to cut off insignificant tails of the distribution. The procedure is configurable.\ Remark: the first iteration is always a non-windowed basic_moments. So, if M is, let’say, 5, then 6 iterations will be performed: one unwindowed and five windowed, according to configuration as described above.

Field name	Format	Default	Description
widths	1xM double	[3]	A vector of window widths (for each pass but the first which is basic_moments) referred to reconstructed beam width (i.e. 2.0 means the window would be twice as wide as the beam width found for the previous pass). The size of this vector defines the number of algorithm loops.
min_widths	1xM double	[20]	A vector of minimum window widths in detector's channels. Must be the same size as widths.
shapes	1xN char or Mx1 cell or 1xN double or MxN double	'gauss'	A string or a number or a vector (1,N), or a cell array, or a vector (M,1) or an array (M,N). **1xN char:** specify window shape for all passes as either 'rect' or 'gauss'. **Mx1 cell:** specify window shape for each iteration separately, e.g. {'gauss', 'rect', 'rect'} **1xN double:** specify any window shape. The size of this vector is not important; it will be resampled to give correct window width. The same shape will be applied for all iterations. **MxN double:** as above, but allows specifying different shape for each iteration.

Table 3-16 windowed_moments: recon_attributes

basic_phase: The beam position is derived from the phase of the first Fourier component of the profile. This method does not support width calculation (not yet). It is very fast (comparable to basic_moments) and largely insensitive to common noise. However, it's behavior with asymmetric beam shape must be studied.

There's nothing to configure here.

fourier_phase: The beam position is derived from the phase of first N Fourier components of the profile. This method does not support width calculation (not yet). It is only a bit slower than basic_phase and offers superb beam position reconstruction even for very bad SNR. It has been proven to be more stable than gaussian fitting. For good SNR, it gives comparable or slightly worse results than fitting. For bad SNR it's even better.

However, good reliability depends on proper setting of the `npoints` attribute (see below). In general, the wider the beam and *the smaller the detector plane (less channels)*, the lower value of this parameter should be used. Typical values range from 5 (v.1 board, wide beam) to 20 or more (v.2 board, narrow beam).

Still, its behavior with asymmetric beam shape must be studied. Also, beam width calculation must be implemented.

There is one configuration attribute:

Field name	Format	Default	Description
npoints	1x1 double	5	Number of points of Fourier transform taken into account. In general, the wider the beam, the less points make sense.

Table 3-17 fourier_phase: recon_attributes

other methods: Adding new methods to the recon framework is quite easy:

1. Find a nice name for the new method, In this example, it will be `my_method`.
2. Develop a function with a custom algorithm. Its declaration should have a form:

```
function [positions widths] =
recon_my_method
(data, attributes);
```

   Save the file as `[matlab]/recon_my_method.m`.

3. Edit `[matlab]/recon.m` and add the name of the method (in our case, '`my_method`', to the declaration around line 26. It will allow calling our method by identifier. If the method will be called only by name, this step is not needed. Below a code snippet from `recon.m` with `my_method` added:

```
    %Convert numeric recon_mode to string
recon_modes = {'gaussfit', 'basic_moments', 'windowed_moments',
'basic_phase', 'fourier_phase', 'my_method'};
if ~ischar(recon_mode)
    recon_mode = recon_modes(recon_mode);
end;
```

The programmer has full freedom with defining attributes for his method. However: the uniform `recon` interface requires that the `recon_attributes` parameter is optional. It is programmer's responsibility to provide a parameter control mechanism. Tip: see any of `recon_*.m` files to see how to deal with default parameters.

Note that any specific reconstruction function deals with a single board and it is assumed that there is only one beam, i.e. we search for a single peak.

3.3.1.8. load_hit_data

```
function out =
load_hit_data
(filename)
```

Load HIT (EtherCAT) text file.

Input arguments:

Name	Format	Default	Description
filename	1xN char	*required*	The name of the file to load with an extension (mostly probably *.csv)

Table 3-18 load_hit_data: input arguments

Output arguments:

Name	Format	Description
out	nested structure	A structure with multiple fields, each containing two sub-fields(see below). The field names are same as in HIT file (with dots '.' and minus signs '-' changed to underscore '_'.

Table 3-19 load_hit_data: output arguments

out structure sub-fields:

Name	Format	Description
time	1xF double	A vector containing a timestamp for each sample (it's HIT's timestamp, it has nothing to do with our timestamper)
values	1xF double	A vector ontaining the value of each sample.

Table 3-20 load_hit_data: out structure sub-fields

To clarify this odd format: if the HIT file contains data for `IC1` and `IC2`, each having 12345 samples, we can expect following structure at the output:

```
out.IC1.time: 1x12345 double
out.IC1.values: 1x12345 double
out.IC2.time: 1x12345 double
out.IC2.values: 1x12345 double
```

Important remark: Each channel in EtherCAT can have its own sampling frequency. Therefore, output vectors can be of different length for each channel. Use hit_resample() to make them uniform.

3.3.1.9. hit_resample

```
function out =
hit_resample
(in, fs)
```

Resample HIT data to common frequency and make the structure less complex.

Input arguments:

Name	Format	Default	Description
in	nested structure	*required*	The structure returned by `load_hit_data`. See Table 3-20 for details.
fs	1x1 double	*required*	Common sampling frequency for all channels after resampling.

Table 3-21 hit_resample: input arguments

Output arguments:

Name	Format	Description
out	1x1 structure	A structure with multiple fields. The field names are same as in the input structure and contain vectors with resampled data. There is an additional field called `rel_time` which contains common timestamps for all data after resampling.

Table 3-22 hit_resample: output arguments

To clarify this less odd format: if the HIT file contains data for `IC1` and `IC2`, and the input structure is like in the previous example, and we assume that we resample 1 second of data at 10 kHz, we can expect following structure at the output:

```
out.rel_time: 1x10000 double
out.IC1: 1x10000 double
out.IC2: 1x10000 double
```

3.3.1.10. process_all_hit

This **script** is to process all HIT data (`*.csv`) and convert it into `*.mat` files. It will ask user for the working directory. The data is resampled to 20 kHz and absolute timestamp from timestamper is added. The timestamper signal must be recorded as `Analog_IN1`.

If needed, the code can be edited to change resampling frequency or name of the timestampler channel.

3.3.1.11. export_hit_all

This **script** is to export all resampled HIT data with absolute timestamps to `*.csv` files, which can be further used e.g. by root scripts. The `*.mat` files must be generated first by using `process_hit_all`.

User will be asked for the directory name to process. The directory must contain a subdirectory called `with_timestamp/`. This is the place where converted files will be stored.

3.3.2. ROOT interface

First remark: It's not really a ROOT interface. It's a bundle of C++ objects which allows easy accessing the saved data from **any** C++ program. No single ROOT type is used here. But yes, it's compatible with *cint* without any steroids, which makes interfacing with interpreted ROOT scripts possible.

3.3.2.1. Data structure

The data structure (see Fig. 3-2) has several levels: The main `Hitdata` contains a number of `Fullframe`s, which contain a number of `Boardframe`s, which finally contain some data (exactly a single frame, single board) and a `Syncframe`.

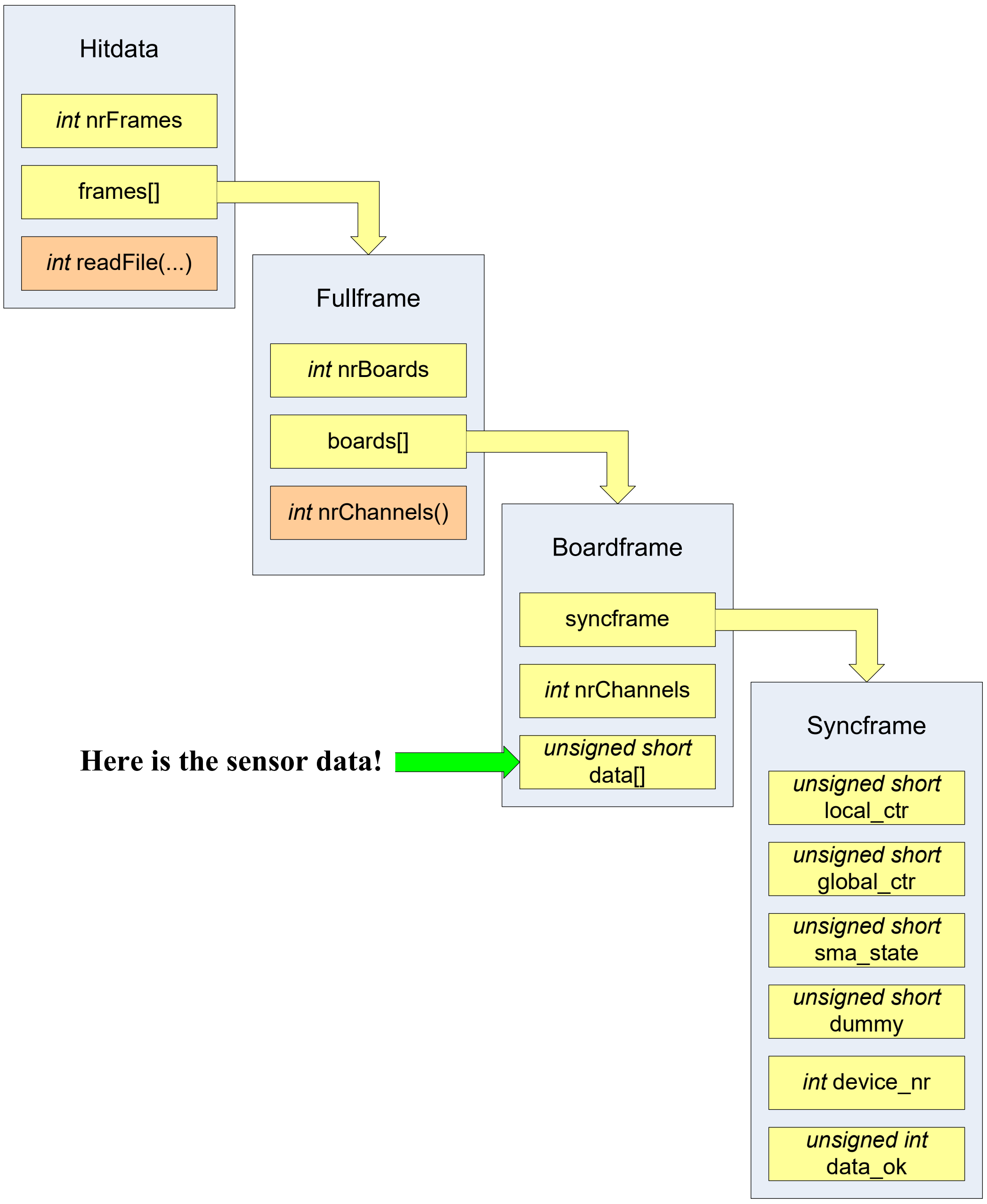


Fig. 3-2 Data strucure used by the ROOT interface

3.3.2.2. Reading data

To load some data, one might do simply:

```
Hitdata mydata;
mydata.readFile("my_file.da2")
```

However, there's one remark: while it's not a big deal to load even a big file with a **compiled** program, forget about it in the case of *cint*. Therefore, it's desirable to read process data in smaller bunches. This is allowed by using optional parameters. The full declaration of `Hitdata::readFile` is actually:

```
int readFile(char* filename,
             int first_frame = 0,
             int nr_frames = -1,
             int increment = 1)
```

The use of `first_frame` and `nr_frames` is obvious, where `nr_frames==-1` means 'read all frames till the end of the file'. The `increment` parameter allows reading only every *n*th frame, which might be helpful to see just a quick overview of the file.

The function returns the number of correctly read frames.

If talking about numbers, reading 10 000 frames with 11 sensors (704 channels) at once with *cint* (virtual machine, 2 GB RAM) is reasonable. For more than 40 000 frames, the execution time was significantly increasing due to problems with memory allocation (yes, it was only around 50 MB of data!)

3.3.2.3. Accessing data

Once we have loaded the data, we can access it. Here are some examples:

```
      //frame count
mydata.nrFrames
      //board count
mydata.frames[0].nrBoards
      //channel count in board 0
mydata.frames[0].boards[0].nrChannels
      //access channel 1 in board 2, frame 3
mydata.frames[3].boards[2].data[0]
      //access data_ok field
mydata.frames[3].boards[2].syncframe.data_ok
```

All data structures are safely copy-able (a deep copy of all arrays is performed), so one can 'extract' interesting parts and dump the rest. The following code (even if stupid) will work correctly:

```
       //We load all data
Hitdata* pmydata = new Hitdata;
pmydata->readFile("my_file.da2");
       //Let's say that we want to deal only with frame 3
       //Copy data
Fullframe myframe =  pmydata.frames[3];
       //Delete unneeded dataset
delete pmydata;
       //myframe is still accessible and functional.
```

A more advanced (and not that stupid) example:

```
       //We are gonna get data only for board 3, but all frames
       //First, load all data
Hitdata* pmydata = new Hitdata;
pmydata->readFile("my_file.da2");
       //Assign memory for our fine-cut data
Boardframe* pmyboarddata = new Boardframe[pmydata->nrFrames];
       //Copy data
for (int i = 0; i < pmydata->nrFrames; i++)
    pmyboarddata[i] = pmydata.frames[i].boards[3];
       //Delete unneeded dataset
delete pmydata;
       //pmyboarddata is still accessible and functional.
```

A 'simplified access' is possible thanks to overloading the `[]` operator. For Hitdata, `[]` is the same as `.frames[]`. The thing is more complicated for Fullframe, where the `[]` operator accesses directly sensor channels. The channel numbering is consecutive over boards, so first all channels of board 0, then all channels of board 1 and so on.

This allows to reduce our three-level data structure to two-dimensional `[frame][channel]` addressing, similar as for the 'old' v.1 ROOT interface and Matlab's `load_data` (see 3.3.1.1: *load_data*). Below an example:

```
       //First, load all data
Hitdata mydata;
mydata.readFile("my_file.da2");
       //Get a single value: frame 1, channel 2
unsigned short mysample = mydata[1][2];
       //That's all
```

Remember that what you see is an effect of operator overloading and not a true array – don't try to *memcpy* it. If you want to make a copy, you need to do it manually:

```
       //We assume that our output array is already allocated
for (int frame = 0; frame < mydata.nrFrames; frame++)
       for (int ch = 0; ch < mydata[frame].nrChannels(); ch++)
            output[frame][ch] = mydata[frame][ch];
       //Done.
```

In the example above, a convenience method `nrChannels()` of `Fullframe` is used, which calculates the total channel count for all boards.

4. User’s manual

This brief manual can be used as a checklist when running the system.

4.1. Hardware preparation

Besides the mechanical installation, check connections of all cables (see 2.3.7.3: *Sockets and connectors*):

- Power connections between lab PS, synchro board, v.1 boards and v.2 frontend board;
- Power supply connection for the v.2 FPGA board;
- Synchronization connections between DAQ boards and synchro board; check if correct board is connected to the Master socket (see 2.8.1: Synchro board)
- Timestamper connections (coax to DAQ and Ethercat, USB to the PC) – if timestamper is used;
- Ethernet: DAQ boards, switch, PC, connection to control room,
- Power supply of the Ethernet switch.
- Optional: USB serial port for debugging (see 6.1: *Debug port*)

Next, power on the system:

- Power up the PC;
- Ensure that the Ethernet switch is working;
- Switch on the FPGA board (on-board switch);
- Switch on lab PS and check voltages, then switch on its output;

Look at current consumption (see 2.8.4: *Power supplies*). Check if there’s no smoke and that the component’s don’t overheat (the FPGA board and metal plates around it, as well as all voltage regulators can be noticeably warmer). After this check, the black cover can be put on.

4.2. Setting up software

If any major changes to the setup have been introduced (changing number of boards, planes, new PC etc.), they have to be reflected in the configuration. All settings are collected in the DAQ software under `menu→Settings`.

Remark: Changing any settings needs the run be stopped. Some of them (host IP, device list) need the devices be disconnected (see chapter 4.3: *Data acquisition*). If these conditions are not fulfilled, the software will stop the run and disconnect devices prior to entering any configuration window. Or maybe I’m wrong, so it’s always better to do it manually.

4.2.1. Configuring host IP

The host IP configuration is needed to tell to all boards, where they should send frame data. The IP of this interface should be entered, which is used for communication with the measurement system. It must be compliant with `10.0.7.x` mask.

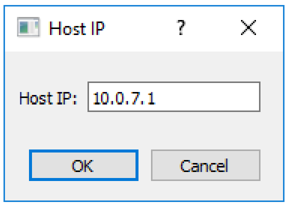
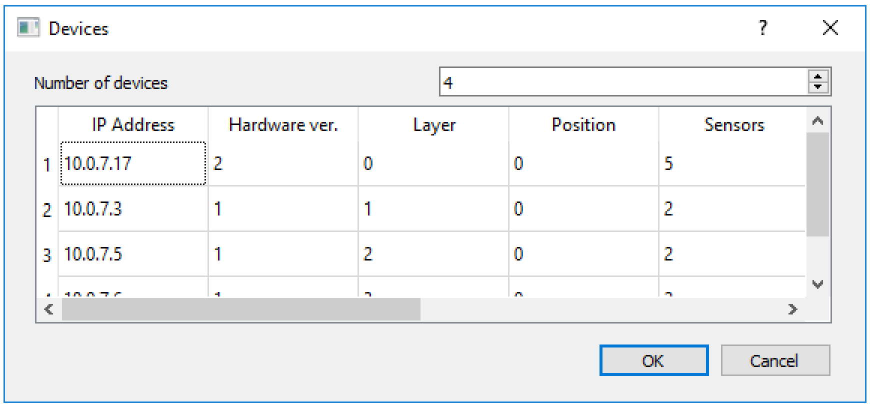


Fig. 4-1 Host IP configuration window

4.2.2. Configuring device list

Device list configuration can be found under `menu→Settings→Devices`. It's use is obvious and a screenshot is below.



Number of devices: 4

	IP Address	Hardware ver.	Layer	Position	Sensors
1	10.0.7.17	2	0	0	5
2	10.0.7.3	1	1	0	2
3	10.0.7.5	1	2	0	2
4	10.0.7.6	1	3	0	2

Fig. 4-2 Device list configuration window

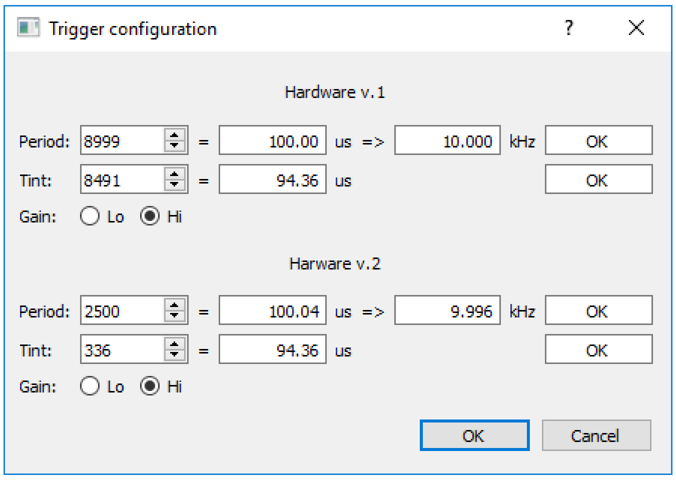
The following table summarizes all settings:

Nr.	Field	Description
1	Hardware ver.	Set 1 for v.1 devices and 2 for v.2. Other numbers are not supported.
2	Layer	Assign layer to each device. Layer numbers should be sonsecutive and start with 0. More than one device can be assigned to a single layer.
3	Position	Position of the device in its layer. These numbers should start with 0 and be consecutive for each layer.
4	Sensors	Number of sensors installed. The DAQ allows for installing less sensors on a board than its capacity. However, if removing some sensors, there should be always last ones (these which appear rightmost on the online display).
5	Master	Set 0 for slave board and 1 for master one. Exactly one board should have master setting. And no zero of them will you set as masters and not two, but exactly one. One is the number of boards which you set as master and the number of boards which you set as master is one.
6	Master dly	Delay setting when working in master mode. See chapter 2.4: *Details on synchronization*
7	Slave dly	Delay setting when working in slave mode. See chapter 2.4: *Details on synchronization*

Fig. 4-3 Device list configuration parameters

4.2.3. Setting up timing and sensor gain

These settings can be found under `menu→Settings→Trigger` config. The setting window is shown below.


Trigger configuration

Hardware v.1

Period: 8999 = 100.00 us => 10.000 kHz OK
Tint: 8491 = 94.36 us OK
Gain: Lo [✓ Hi]

Harware v.2

Period: 2500 = 100.04 us => 9.996 kHz OK
Tint: 336 = 94.36 us OK
Gain: Lo [✓ Hi]

Fig. 4-4 Trigger configuration window

There are two identical panes. The upper one is for configuring all v.1 devices while lower one is for v.2. The settings are given by user in clock units. For convenience, the software translates them to seconds and shows next to the entered setting. It also performs a basic check if the entered settings are correct. There basically three settings:

– `Period`: this is the period of the frame clock used by the master board. If a v.1 board is used as master, it's enough ot configure proper period for v.1 pane. The same story for v.2.

  The clock used is 90 MHz for v.1 and 50 MHz for v.2.

– `Tint`: this is the integration time (or, strictly speaking, the duration of positive RESET pulse) expressed in sensor's MCLK ticks. The MCLK frequency is 4 MHz for v.1 and 3.5714285 for v.2.

  Integration time is checked if it's shorter than frame period (minus a margin required by the sensor).

– `Gain`: It simply switches sensor gain between low and high. The difference in sensitivity is about a factor of 2. There's no defference in behaviour of v.1 and v.2 boards.

4.3. Data acquisition

To acquire data, proceed with following steps:

1. Connect devices (`menu→Device→Connect`)
2. Switch on online display if needed (`Show display` button; it will turn to `Hide display`)
3. Start run by pressing `Run` button. The button will turn into `Stop`.
4. Press `Start logging` button. A file dialog will open. Enter file name for saving data. The button will turn into `Stop logging`.
5. To finish, press `Stop logging`, `Stop`, `Hide display` and finally `menu→Device→Disonnect`.

In the status bar of the application, there's an information about current frame rate and buffer occupancy. During a run, the buffer occupancy for all boards oscillates around 12.5%. This level is maintained to allow fluent operation of the event builder. The information also contains a text: '`OK`' or '`Warning`', which is a quick indicator if buffer levels are considered 'safe'.

Important: Constant indication of 0%, 50% or 100% for any board means that there's a problem with synchronization.

Refer to chapter 3.1.4:

Output data format, Table 3-1 and Table 3-2 for file format specification. The files have `.da2` extension in contrast to `.dat` files of earlier software version which supported only v.1 electronics and had different file structure. Together with the dat file, another file with `.dat.ini` extension will be created. It's a copy of currently used ini file, i.e. the hardware configuration.

4.4. Data post-processing

The data may be conveniently post-processed by Michal's Matlab scripts or imported into ROOT. Look at chapter 3.3: *Offline software* for a quick reference of most important functions.

4.4.1. Post-processing in Matlab

Here a very brief description to see collected data in Matlab:

To load your data

```
[data_bl syncdata I_offspill I_onspill I_neutral boardinfo] = ...
load_and_prepare_data('your_filename'); ↵
```

If you don't want to load the whole file, but only a specific range of frames, use a longer form:

```
[data_bl syncdata I_offspill I_onspill I_neutral boardinfo] = ...
load_and_prepare_data('your_filename', [], first_frame, ...
nr_frames); ↵
```

Note that `data_bl` has already removed baseline.

To plot your data as a 3-D plot, simply type:

```
mesh data_bl(:,1:1000:end) ↵
view(0,90) ↵
```

The first command will make a mesh plot of every 1000th frame of your data; of course you can change this number to anything you like. We simply don't want to plot all frames, as it would kill your computer with e.g. few millions of them...

The second command is to change the viewport to see the plot directly from top (resulting a flat colorized surface). The plot can be rotated and zoomed by mouse.

Note that `load_and_prepare_data` returns not only the data itself, but also synchronization structures, an information on hardware (boardinfo) and further arrays used for frame classification: each frame is classified as either *on-spill*, *off-spill* or *neutral* (i.e. transient). In most cases, only the on-spill frames are interesting for reconstruction, so we can simply decrease the amount of data:

```
data_bl = data_bl(:,I_onspill); ↵
```

If `syncdata` is going to be used, the same operation should be performed for all `syncdata` arrays to keep them consistent with `data_bl`.

To calculate beam parameters (make reconstruction) for each frame, type:

```
[positions widths] = recon (data_bl, 'simple_moments', [], ...
boardinfo); ↵
plot(positions') ↵
```

Look at 3.3.1.7: *recon* for more details on reconstruction framework. The second used command is, of course, to plot your results...

If the data was collected with a timestamper, you might like to extract the timestamper data. To do this, type:

```
real_time = make_real_timestamp(sync_data.sma_state(my_board,:),...
10000, 0.5); ↵
```

where `my_board` ist the number of the board with connected timestamper. Please remember that numbering starts with 1 (not with 0) in Matlab.

Probably you want to generate real times for all your runs, and for both our DAQ and HIT's Ethercat, and finally store them as text files. To do this, proceed as follows:

```
export_all_timestamps ↵
<enter directory name when asked>
process_all_hit ↵
<enter directory name when asked>
```

Please look at the description of used scripts: 3.3.1.6: *export_all_timestamps* and 3.3.1.11: *export_hit_all*.

There are a plenty of further functions, scripts, etc. Look at 3.3.1: *Matlab functions* for more information. Look also at 6.2.3: *Problems with noise* for a description of the noise analysis script.

4.4.2. Post-processing in ROOT

Do it on your own:) What you get from Michal, is a library for loading data into an object-oriented interface. See 3.3.2:

ROOT interface for details.

5. Cloning the setup

This chapter describes how to build another copy of the DAQ system to work in a common trigger domain (i.e. to be connected to the same trigger board). Preparing a new trigger board is not described here.

5.1. Preparing hardware

5.1.1. The FPGA board

An *Intel MAX10 development kit* should be purchased. Following modifications to the PCB are required:

- Installing terminating resistors for LVDS receivers (110 Ohm):
  - Required: R252, R253, R257, R259, R261, R263, R265
  - Nice to have: R254, R255, R256, R258, R260, R262, R264
- Setting DIP switches according to Table 2-3, page 25.

The FPGA schould be programmed with proper programming file: `[fpga]/output_files/output_file.pof` (see chapter 1.2: *Project directory description*).

5.1.2. The frontend board

Four pieces of the frontend board's PCb have been produced, so there's no need to produce new ones in a close future. If needed, they can be produced from generated Gerber files (see chapter 1.2: *Project directory description*). There are no special requirements regarding plating, however the original PCBs were gold plated. There are no special requirements on the technology (smallest detail/gap: 6 mils) and virtually any 4-layer technology is acceptable.

The new PCB should be populated according to 2.3.3: *Assembly drawings* and 2.3.4: *Component list*. Manual soldering the HSMC connector is possible, but needs some experience. Reflow soldering is recommended.

5.2. First run

A fresh board needs configuring its serial number to generate a unique MAC number for the on-board Ethernet controller. To do it, a debug connection is required:

1. Connect the FPGA board's USB serial port to a Windows PC with a mini-USB cable
2. Power up the board and see the virtual serial port's number in the device manager. Power off the board.
3. Open a terminal software (e.g. *RealTerm*) and configure it as follows:

Setting	Value
COM nr.	as detected in Device Manager
Baud rate	115200 bps
Parity	none
Stop bits	1

Table 5-1 Debug comm port settings

4. Power up the board and immediately connect the terminal.
5. When asked for serial number, just type a random 9-digit number. The board configuration should go on.

5.3. Setting up the software

Proceed according to chapter 4.2: *Setting up software*.

5.4. Equalizing delays

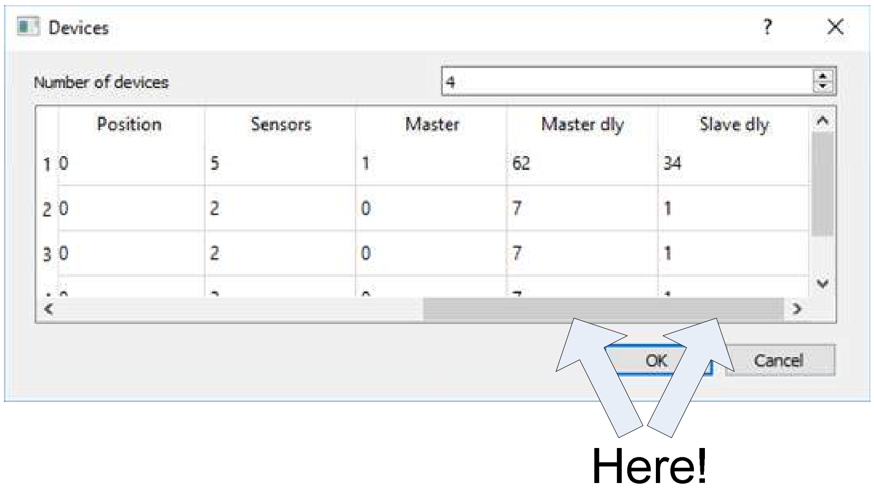
Equalizing delays (fine synchronization) is a quite kinky procedure and it requires an oscilloscope. In fact, it's not needed for new *boards*, but it's needed for any new configuration of our *measurement setup*, e.g. changing the master board or changing the length of synchro cables.

Proceed as follows:

1. Power off everything.
2. Safely connect a probe to RESET signal (i.e. pin 1) of one sensor in each board. You need four probes for four boards. Set up oscilloscope for positive trigger.
3. Power on everything.
4. Run acquisition.
5. Measure time differences between boards using a scope.
6. Stop acquisition and edit delays in board configuration pane in the DAQ software.

   V.1 uses a 90 MHz clock. Therefore, a single quantum of this setting is 11.1 ns. (or maybe 45 MHz and 22.2 ns, I'm not sure...) V.2 uses a 50 MHz clock, so the quantum is 20 ns. The master has obviously the shortest signal path, so master delay settings are expected to be bigger.

See the figure below for help.



Number of devices: 4

	Position	Sensors	Master	Master dly	Slave dly
1	0	5	1	62	34
2	0	2	0	7	1
3	0	2	0	7	1

Fig. 5-1 Delay configuration

7. Go to 4 and repeat the sequence until all sensors start integration simultaneously, that is within 50 ns or any accuracy you want.

6. Troubleshooting

6.1. Debug port

The FPGA board is equipped with a USB serial port. During operation, this port outputs some information regarding current state of the device.

To use the debug port, a terminal software (e.g. *RealTerm*) can be used. See Table 5-1 for port settings.

The operation of the device is fully independent of the debug port connection. The port can be connected and disconnected at any time.

6.2. Common problems

6.2.1. Connection-related problems

6.2.1.1. Board doesn't connect to the software

Symptoms:

The PC software reports socket error after attemting to connect to the board.

Explanation:

The software con not bind a TCP/IP connection with the board. This can be caused by wrong software configuration, wrong IP configuration of the host PC or the board, connectivity problems on Ethernet link, problems with the internal IP core responsible for Ethernet connectivity or finally problems with board's firmware.

Debugging:

- Check if all devices are powered on
- Check status LEDs on the Ethernet switch if all devices and the PC are connected. V.1 devices should report 100Mbps link, v.2 and the PC – 1Gbps.
- Restart software
- Check board IP in the configuration (`menu→Settings→Devices`)
- Compare it with board's jumper setting (see chapters 2.3.7.1: *Switches* and 2.7.2: *(Brief description of v.1) Connectors and controls*)
- Check if PC and board are located within the same subnet (subnet musk **must be** 255.255.255.0)
- Try pinging the device from the PC
- Powercycle the device(s)
- Restart software again
- Use debug port (see chapter 6.1: *Debug port* for details) to check Ethernet connection state.
- Try connecting to the device with a terminal (TCP port 4000)

  - Succesful connection: there’s indeed an error in the configuration of the PC software
- Use Ethernet spying software (e.g. *Wireshark*) to check if the device replies to any packet (including ARP).
  - No reply to any packets with active Ethernet link and correct IP settings: mostly probably a serious problem with interface timing on the FPGA board.

6.2.1.2. No incoming data when running

Symptoms:

After connecting the board and pressing *run* button, there’s no data (rate counter(s) show zero).

Explanation:

This problem can have two sources. First one is bad IP configuration. Another one is lack of trigger signal

Debugging:

- Check „own IP” in the software (`menu→Settings→Host IP`)
- Check if PC’s IP configuration for the used Ethernet adapter matches own IP in the software
- Set the problematic board as standalone master (`menu→Settings→Devices`)
  - Works now: the problem is the trigger connection. Check if all used boards are properly connected to the trigger board and which one is connected to the „Master” socket. It must be configured as Master in the software.
- If running with multiple boards, try using another board as master (needs re-connecting trigger connections and configuring the PC software in accordance to it)

6.2.2. Problems with signal integrity

6.2.2.1. No incoming data when running

See 6.2.1.2: *No incoming data when running*.

6.2.2.2. All channels show zero or maximum

Symptoms:

When running, the output from all channels is exactly zero or 65535.

Explanation:

It seems that there’s no signal from ADCs or the sensors are disconnected.

Debugging:

- Check power supplies for the FPGA board and frontend board (if used)
- Check the HSMC connector between the FPGA board and the frontend board.

- Check output voltages of on-board power regulators of the frontend board (see chapter 2.3.2: *Frontend Schematics*)
- Check the reference voltages (see chapter 2.3.2: *Frontend Schematics*)
- Check if sensors are properly attached. Aren't they shifted by one pin? New board: are the connectors mounted on the right side of the PCB?

6.2.2.3. One or more sensors show zero or maximum

Symptoms:

When running, the output from all channels from one or more sensors is exactly zero or 65535.

Explanation:

The most common reason for this problem is a broken or badly connected sensor. Another possible reason is malfunction of the readout circuitry.

Debugging:

- Check if the sensor is properly attached.
- Try swapping two sensors to check if the problem moves with the sensor.

6.2.2.4. Some channels show odd values

Symptoms:

When running, the output from some channels from one or more sensors exhibits following behavior:

- it suddenly jumps between different values
- it has outlying baseline
- it exhibits abnormal sensitivity (lower or higher)
- it outputs a constant value

Explanation:

Such kinds of artifacts happen for broken sensors. The most common reason for sensor damage is ESD.

Debugging:

- Try swapping two sensors to check if the problem moves with the sensor.
- Replace suspect sensor with a new one.

6.2.2.5. A whole sensor shows odd values

Symptoms:

When running, the output from all channels from one or more sensors exhibits following behavior:

- they suddenly jump between different values
- they have outlying baseline
- they exhibit abnormal sensitivity (lower or higher)

Explanation:

These artifacts may be caused by sensor malfunction or problems in the DAQ circuitry.

Debugging:

- Try swapping two sensors to check if the problem moves with the sensor.
- Replace suspect sensor with a new one.
- Visually inspect the PCB to find bad solders etc.
- Check the analog and digital path for the bad sensor (see chapter2.3.2: *Frontend Schematics*)

6.2.3. Problems with noise

General remark: There is an utility (written in Matlab) to perform quick noise analysis. The script name is `script_cov`. It's placed in the `[matlab]` directory.

It takes a file with collected (dark) data as an input and produces some noise statistics, including RMS (total, per-board and channel mean), spectrum and cross-covariance.

An example output for a correctly working setup is shown in Fig. 6-1 and Fig. 6-2. A single v2 board (first 5 sensors) and three v1 boards (following pairs of sensors) can be seen there.

Fig. 6-1 shows overall noise characteristics of the setup. The most interesting plots here are channel covariance plots (to the left). By definition, the diagonal of the matrix shows the noise power for each channel. What's outside the diagonal are covariances between channels. These values are a measure of *common noise* for sensors, boards and for whole detector.

The upper plot shows the whole covariance matrix. The lower plot shows the same data bat viewed 'from corner' (viewport 45,0); the 'pin' in the middle is a projection of the diagonal.

Under normal circumstances, the common noise of a sensor is higher than common noise of a board, and the common noise of a board is higher than common noise for whole system. The common noise of v2 board is significantly lower than v1. However, the channel noise is comparable for both versions.

The two plots to the right show the periodogram – i.e. the power spectrum of a noise in a single channel (upper plot) and for the whole detector (lower plot).

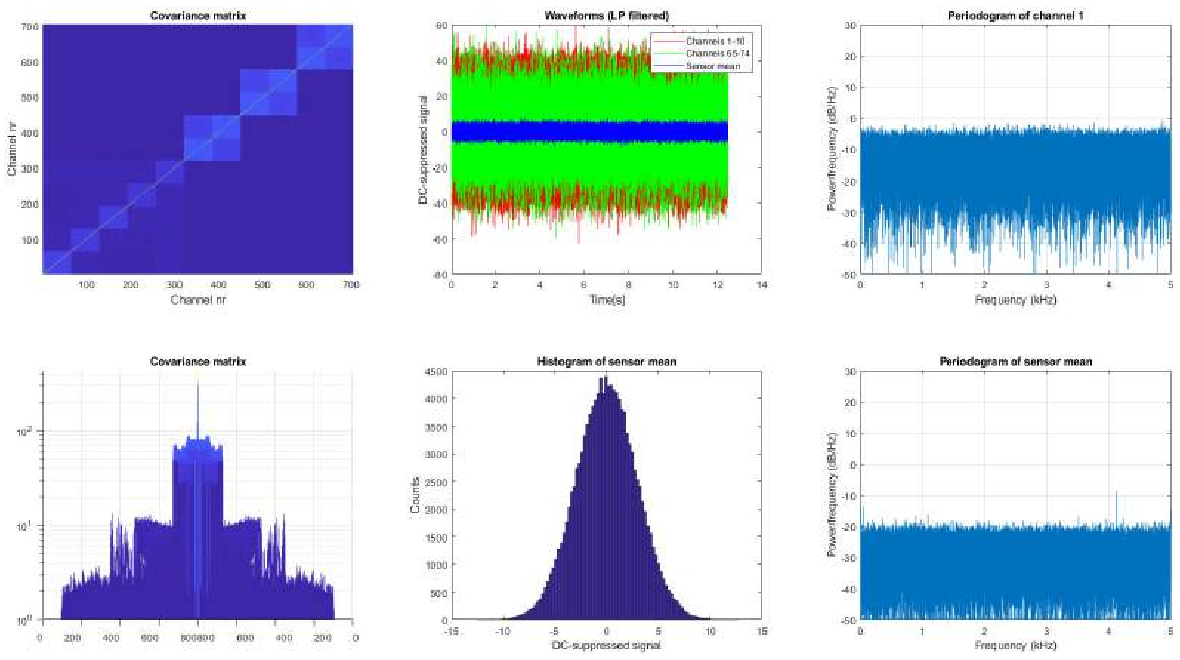


Fig. 6-1 Output of noise utility (1)

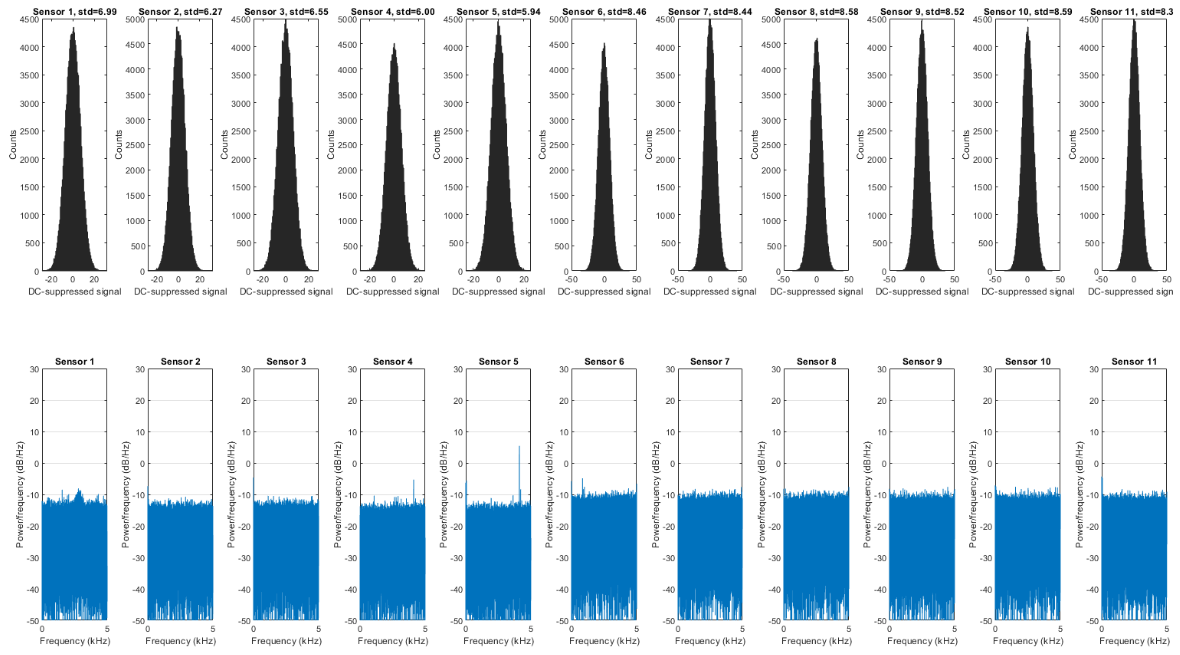


Fig. 6-2 Output of noise utility (2)

Fig. 6-2 shows histograms and power spectra of common noise for all sensors. On It can be seen in the example that the common noise for first 5 sensors (v2 board) is lower than for the following ones.

When collecting data for the script, two requirements should be fulfiled:

- at least 10 seconds of data (at 10 kHz) should be acquired
- there should be no optical signal – so completely dark.

6.2.3.1. Signal is "jumping"

Symptoms:

When running, the output from all channels „jumps" on the display

Explanation:

„Jumping" of the whole plot is typical for excessive common noise. The most common sources are parasitic light, heavy external interference, problems with power suply and, last but not least, with the reference

Debugging:

- Collect at least 10 seconds of data and run the Matlab utility. The desired noise RMS per sensor should be 6-9 ADC counts.
- Check if there's no active cell phone in the neighborhood.
- Check if there are any uncovered LEDs on the FPGA board.
- Switch off any light in the room and look for changes.
- Check power supply. If using power from FPGA board, try with an external PS.
- Try checking the noise on sensor and ADC references with a scope (due to limited sensitivity of a scope, this will detect only very serious problems!)

6.2.3.2. Signal is "jumping" for a single sensor

Symptoms:

When running, the output from all channels in one or more sensors „jumps" on the display

Explanation:

This behavior is quite typical for a broken sensor. Another problem could be connectivity issues in the reference and signal paths.

Debugging:

- Try swapping two sensors to check if the problem moves with the sensor.
- Replace suspect sensor with a new one.
- Visually inspect the PCB to find bad solders etc.
- Check the analog and digital path for the bad sensor (see chapter 2.3.2: *Frontend Schematics*)

6.2.3.3. Excessive channel noise

Symptoms:

Excessive channel noise is observed, but it’s uncorrelated between channels.

Explanation:

The most possible reason for such a behavior is a heavy source of external interference.

Debugging:

- Collect at least 10 seconds of data and run the Matlab utility. The desired noise RMS per sensor should be 6-9 ADC counts.
- Check if there’s no active cell phone in the neighborhood.
- Check for other possible sources of RF signals.

7. Known issues

7.1. Even/odd channel swapped

During 24.11.2019 test beam run at HIT, it turned out that even and odd channels are swapped in the output of the v.2 board. This has been corrected in hardware on 27.11. The Matlab read procedure (`load_data.m`) has become a patch which automatically swaps channels for files created earlier than the hardware correction. The files generated during test beam run have **not** been corrected, so one has to be aware of it if not using Matlab framework to read the data.

7.2. EVB loosing synchronization – dropping packets

During the same session, the event builder lost synchronization few times. Stopping the run and re-connecting was enough to bring it back to working state. Later it was checked that restarting run without reconnecting is enough,

The EVB algorithm was carefully checked. After trying to clone the problem on Michal's computer, it turned out that many data frames were lost in Ethernet transmission. What is important, these were long, consecutive bunches of frames (reaching few dozens and more frames from all boards).

Here is an explanation: If a bunch of lost frames is smaller or equal to 255, the event builder is able to re-synchronize properly. However, if the gap is bigger, a mis-synchronization (with a shift of a multiple of 512 frames) is possible.

The source of this problem is believed to be in the PC, as there are big differences between Michal's PC (many lost frames) and the lab PC (lost frames found only in one run over the whole session). The possible reason could be either the Ethernet adapter or the processing power of single core, or OS configuration, or whatever.

The DAQ software was tuned (changing task priorities, changing some constants regarding use of buffers) to minimize the number of lost packets, however the effect of this tuning is quite moderate. Still a fraction of ca 10^{-4}-10^{-3} frames gets lost (on Michal's PC), mostly in bunches of few dozens of frames.

As a workaround, the EVB algorithm has been upgraded. Now it checks if local frame counters are consistent with readouts of the global frame counter for each board (i.e. if `LFC%512==(GFC+1)%512`). If it's true, the local frame counters are used for synchronization instead of global one. As local counters have bigger capacity (16 bits instead of 9), they give a bigger safety margin, theoretically allowing up to 32767 frames be lost in one bunch.

However, if a discrepancy between local and global counters is found (this might happen e.g. if some boards loose some triggers), the algorithm falls back to comparing global counter readouts, with a safety margin of 255 frames.

Anyway, this solution only increases the safety margin of the event builder. **It does not solve the problem of loosing packets.**

7.3. Trigger configuration

In trigger configuration pane of the DAQ software, the integration time was checked against frame rate, but only regarding the same version of board. **It has been corrected.** The correction has no influence on data format or whatever else.

By the way, it was checked that the calculation of frame rate from timer setting is correct.

8. Index of terms

9. List of figures

10. List of tables

11. References

[1] Max10 Developer Kit manual – google it!

[2] Max10 Developer Kit schematics – google it!

[3] Max10 booting methods – google it!

[4] Niche stack documentation – google it!

[5] Hamamatsu S11865-64 spec – google it!

[6] AD7983 spec – google it!