ISOTDAQ 2019

Lab Book

ROYAL HOLLOWAY UNIVERSITY OF LONDON

International School of
Trigger & Data Acquisition
10th Anniversary Edition
3- 12 April 2019

LHCb ATLAS EXPERIMENT CMS ALICE DELL CERN

Tutors and their Labs

1. **VMEbus programming** – Markus Joos
2. **NIM** – Francesca Pastore, Andrea Negri, Vincenzo Izzo
3. **NIM & Scintillator** – Kostas Kordas, Roberto Ferrari
4. **Muon DAQ** – Enrico Pasqualucci, Cristovao Beirao
5. **FPGA Basics** – Dominique Gigi, Petr Zejdl
6. **MicroTCA** – Hannes Sakulin, Marc Dobson
7. **LabView DAQ** – Gary Boorman, Adriaan Rijllart
8. **ADC Basics for TDAQ** – Manoel Barros Marin
9. **Network Programming** – Fabrice Le Goff, Adam Abed Abud
10. **Microcontrollers** – Mauricio Feo, Barthélémy von Haller
11. **Storage Systems** – Paolo Durante, Tommaso Colombo
12. **Control of DAQ Systems** – Wojciech Brylinski, Danilo Cicalese
13. **SoC FPGA** – Johannes Martin Wuthrich
14. **GPU** – Giuseppe Lamanna

The main page of the 2019 ISOTDAQ School is:
https://indico.cern.ch/event/739424

For up-to-date information on times and places, please see:
https://indico.cern.ch/event/739424/timetable/

Local Organising Committee

William Panduro Vazquez (RHUL)
Francesca Pastore (RHUL)
Pedro Teixeira-Dias (RHUL)
Veronique Boisvert (RHUL)
Simon George (RHUL)
Ian Murray (RHUL)

CERN Organising Committee

Paolo Durante
Markus Joos
Hannes Sakulin
Barthelemy von Haller

Laboratory Usage Note

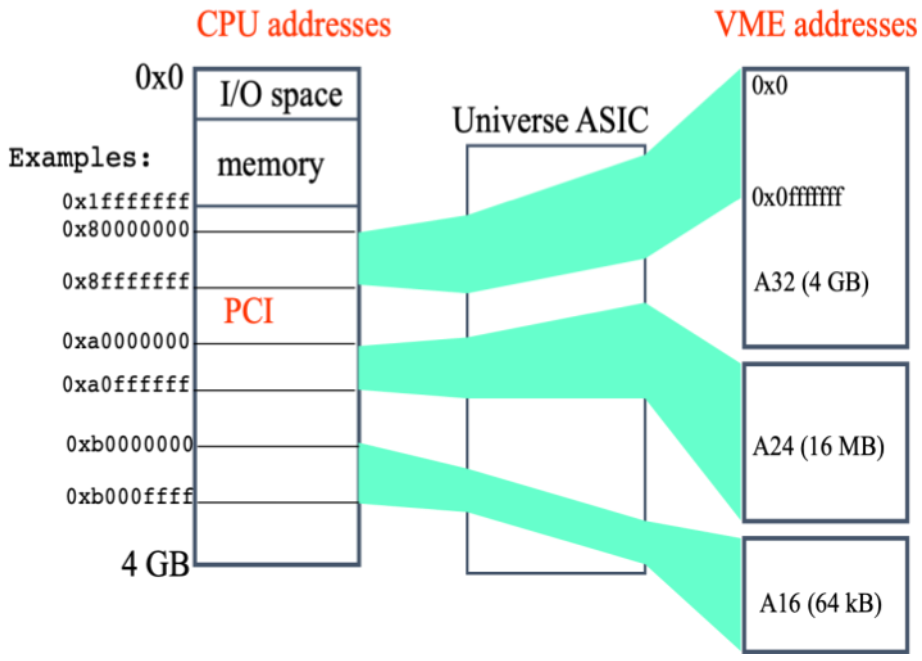
Maintaining a safe working environment in the laboratory is paramount. You should at all times act in a safe and responsible manner. In particular, note that you must not eat, drink, or act unprofessionally in the Laboratory. Certain experiments require you to use additional safety equipment (e.g. tongs for radioactive sources, goggles, or gloves). You must follow such requirements. Any safety concerns should be raised with the department safety officer **Andy Alway** or the laboratory supervisor **Ian Murray**.

1. *Lab 1: VMEbus programming*

Introduction

For the moment forget what you (hopefully) have learnt about the VMEbus protocol and the details of the H/W. For this exercise you have to look at a VMEbus slave as if it was a piece of memory in your PC. The purpose of this exercise is to demonstrate that in some respects there is little difference between internal and external memory; as far as the programming is concerned. The exercise also shows the differences between the two types of memory.

What is important to understand is that the VMEbus memory has to be mapped into the (virtual) address space of a user process before it can be accessed. This ties 3 busses together: CPU, PCI and VMEbus as shown in the picture below.



The first part of the exercise is to figure out how to create the appropriate mappings for the type of VMEbus access that you have to do. Then you actually transfer the data. This is done in single cycle mode which means that the CPU controls the data transfer.

In the second part of the exercise you will perform block transfers (DMA). This requires a different programming technique since it is not the CPU that moves the data but an external

device (a DMA controller). Such DMA controllers are not VMEbus specific. You find them everywhere (e.g. in Network interfaces, disk controllers, USB devices, etc.)

Before you start you should be able to answer these questions:

1) What does the acronym A24D32 mean?
2) What is endianness and how do you deal with it?
3) What are the advantages of block transfers?

1. On the VMEbus single board computer log on with the DAQ school account (daqschool / g0ldenhorn).
2. Run “source setup” and then change directory to exercise1/groupX
3. Open the file solution.cpp with an editor of your choice (vi, nedit).
4. Add the missing code to “solution.cpp” to execute the VMEbus cycles listed below:
    1. Write 0x12345678 to address 0x08000000 in A32 / D32 mode. Use the "safe" cycles
    2. Read the data back from address 0x08000000 and compare it
    3. Write 0x87654321 to address 0x08000004 in A32 / D32 mode. Use the "fast" cycles
    4. Read the data back from address 0x08000004 and compare it
    5. Write a block of 1 KB to address 0x08001000 in A32 / D32 / BLT mode. You have to prepare the data in a cmem_rcc buffer.
    6. Read the data back from 0x08001000 in A32 / D64 / MBLT mode and compare it
5. Run “make” to compile the application
6. Run “solution” and catch the VMEbus transfers with the VMEtro VBT325 analyser

Good practice:

- Check all error codes
- Do not forget to undo all initialization steps (return memory, close libraries) before you exit from an application

2. *Lab 2: A simple Trigger Exercise*

Introduction

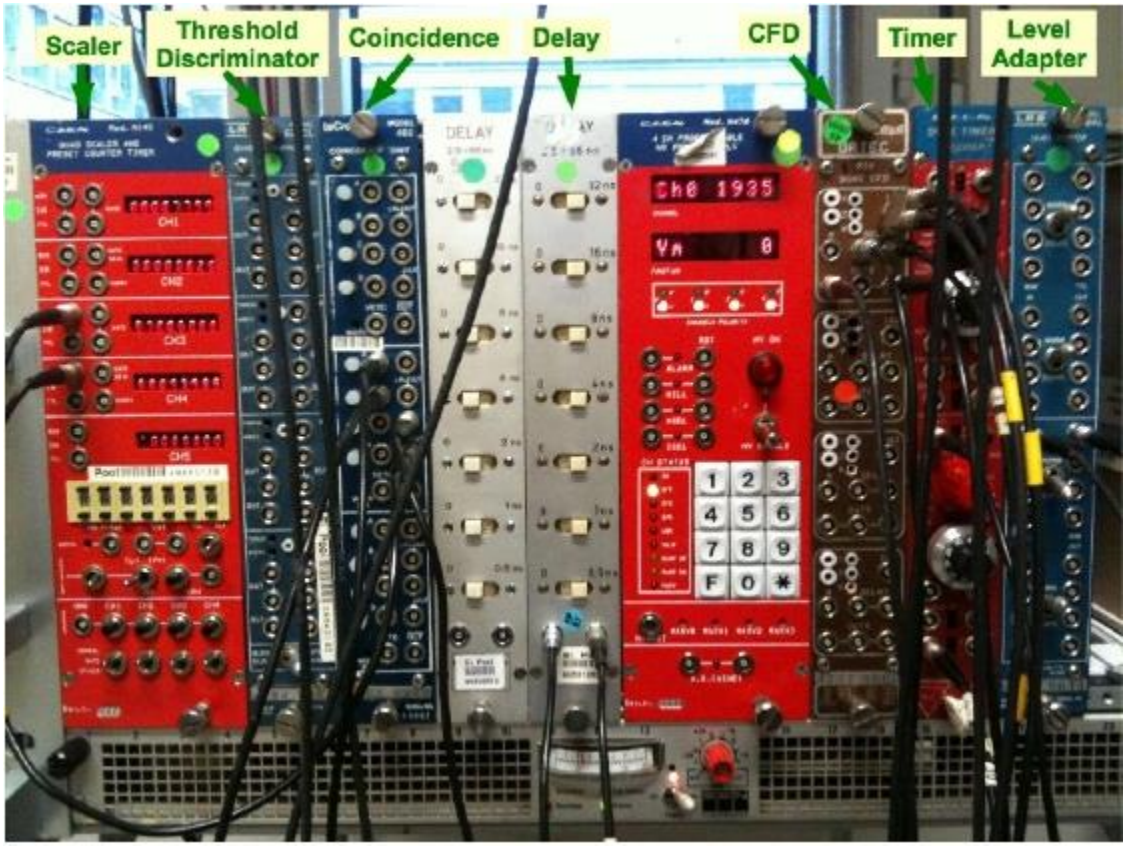


Figure 1: NIM modules.

This is a basic exercise based on the trigger lecture. It introduces all the elements and concepts needed in exercise 3 and 4. The available NIM modules are showed in Fig.1. The exercise is composed of 4 parts. At each step, look at the corresponding schema and follow the instructions.

A trigger is given by the transition of a signal from the logical 0 to 1. Before setting up any trigger system, you must have decided the levels corresponding to these logical levels and all the components of the system need to be coherently configured.

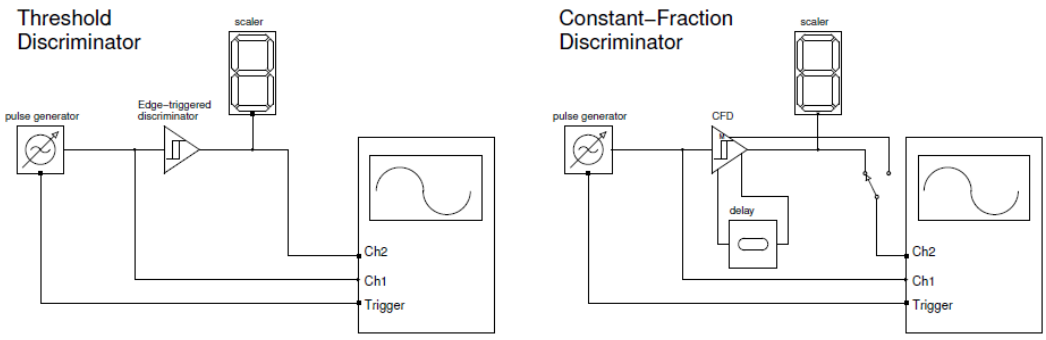


Figure 2: Scheme of threshold and constant fraction discriminators.

Part 1a: Threshold Discriminator

The Signal Generator is pre-configured to provide a triangular pulse with a period of 300 µs. Look at the signal (Channel Output) with the oscilloscope (CH1), using the Trigger Output of the generator as oscilloscope trigger (EXT). The Trigger Output is TTL signal.

How do you expect it?

Why do we use it?

Now try to characterize the signal:

Leading edge time:	
Trailing edge time:	
Width:	

Using the LEMO cables, try to implement the schema shown in the left part of Fig. 2, i.e.:

• Split the generator output signal: connect the two parts to the input of the **Threshold Discriminator** and to the oscilloscope.

• Connect one output signal of the discriminator to the scaler module and a second output to the oscilloscope (CH2).

We have set-up a simple trigger system: you have a digital answer based on the amplitude of a signal. Reproduce the oscilloscope display shown in Fig. 3 and observe the amplitude of both the signal and its corresponding trigger.

Can you modify their amplitude?

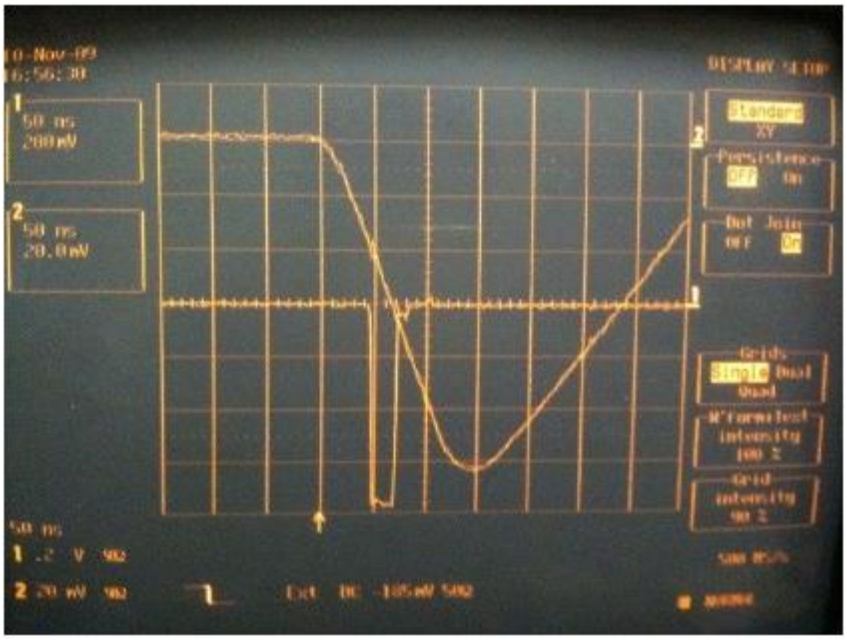


Figure 3: Input signal and threshold discriminator output.

The threshold set on the discriminator can be measured with a Voltmeter (x10 output) and changed with a screwdriver. Change the threshold value: observe the behaviour of the discriminated signal on the scope and its rate on the scaler.

Can you relate them to the threshold values?

In real experiments, how is the best threshold value found?

Part 1b: Threshold Discriminator, the jitter

Using the above set-up, set the discriminator threshold to 60 mV and change the amplitude of the input signal.

Which is the effect on the discriminated signal?

How does it affect a timing measurement?

Measure the discriminated signal delay with respect to the reference as a function of the amplitude of the input signal (-100, -150, -200, -250 mV) and fill up Table 1 with your numbers.

Input signal amplitude (mV)	Threshold D (ns)	CFD (ns)
100		
150		
200		
250		

Table 1: Measured delays on the discriminated signal with respect to reference.

Part 2: Constant Fraction Discriminator (CFD)

Using the above set-up, set the discriminator threshold to 60 mV and change the amplitude of the input signal.

Now use the Constant Fraction Discriminator to make a trigger from the generator signal and implement the layout shown in the right diagram of Fig. 2.

Using the Voltmeter and the screwdriver, set these CFD parameters:

threshold (T):	60 mV	Measure with Voltmeter (x10 output)
walk (Z):	2 mV	Measure with Voltmeter (x10 output)
delay (D):	80 ns	Set with delay module + 2x10ns cables

Connect the CFD monitor output (M) to the scope CH2 and reproduce Fig. 4.

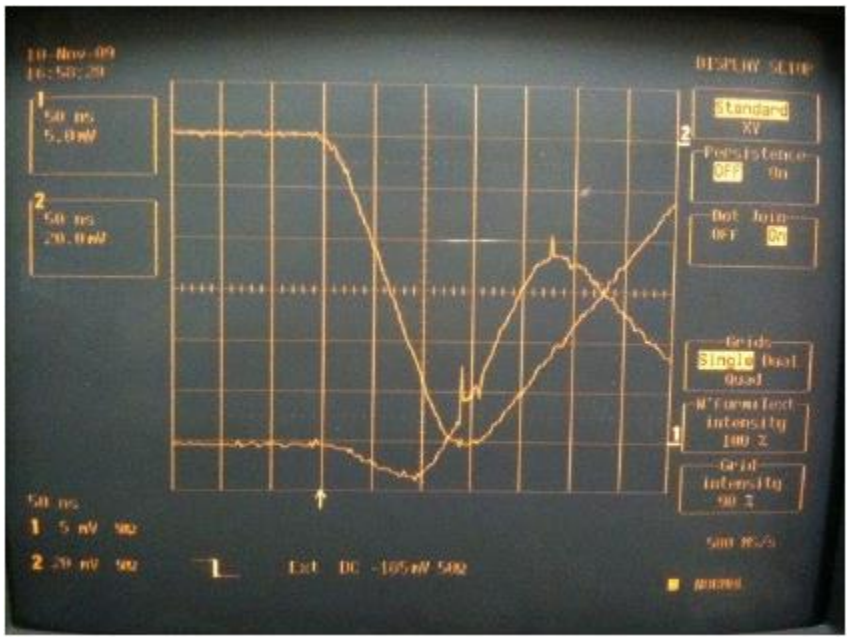


Figure 4: Input signal and CFD monitor output.

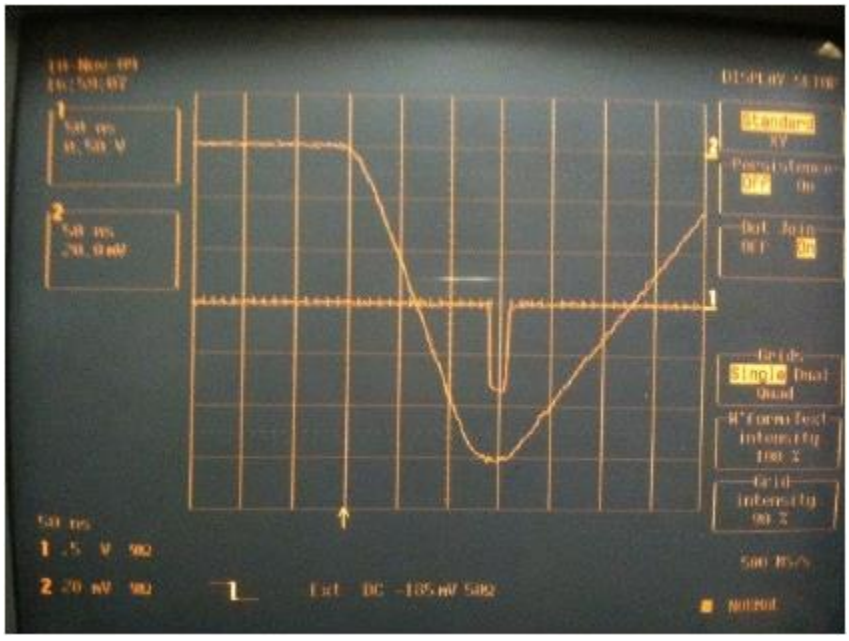


Figure 5: Input signal and CFD output.

Can you recognize the CFD technique?

Which is the effect of varying the value of the delay D?

Connect now the CFD output to the scope (CH2) and change the amplitude of the input signal.

What happens to the output of the discriminator?

Measure the discriminated signal delay with respect to the reference as a function of the amplitude of the input signal (-100, -150, -200, -250 mV). Fill up Table 1 with your numbers. Compare the results with the previous measurements.

Can you see the advantage?

Can you make the CFD behave like a normal threshold discriminator?

Which configuration parameter has to be modified?

Part 3: Making a timing coincidence

We try now to simulate the coincidence of two different trigger signals, in a simplified way. For that, use an additional output of the signal generator, which is configured to generate a triangular pulse similar to the first one. Use two threshold discriminator units to discriminate both signals, as described in Fig.6.

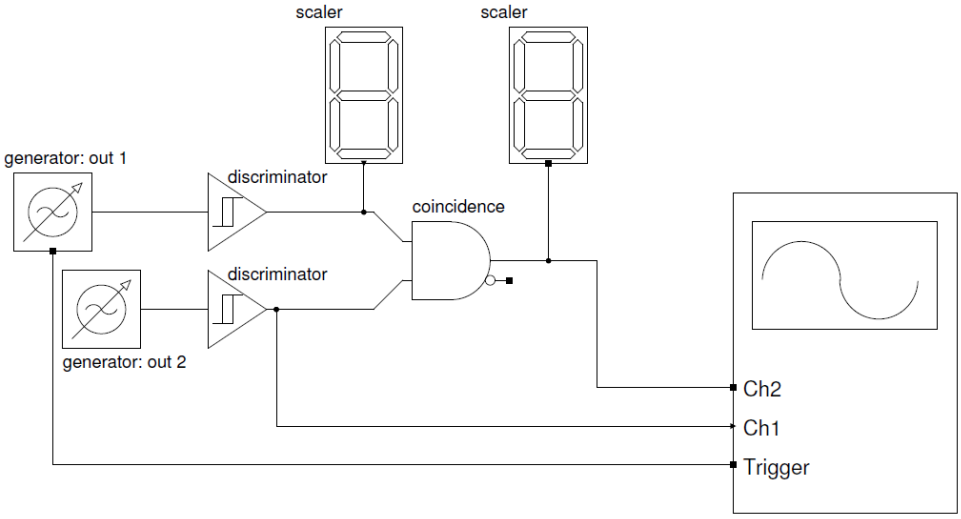


Figure 6: Coincidence layout.

We have now two independent trigger signals with similar characteristics. Look at them in the scope.

Which parameters are important when making a coincidence?

Use one unit of the **Coincidence Module**, which is able to generate the logical AND of its input signals. The module has two outputs: OUT and LIN-OUT.

Can you guess the timing behaviour of the AND output?

When are you expecting the AND output to rise?

The Scaler Module is a simple and useful tool in a trigger system: it allows you to simply count the triggers and verify if your system is behaving correctly. Then use the scaler to measure the counting rate of your coincidence and try to answer these questions:

Can you count any trigger? How can you recover the coincidence rate?

After your adjustments, which is the width of the coincidence signal?

Can you explain the different behaviour of the OUT and the LIN-OUT signals?

Which is better to use in a real trigger system?

How can you save the trigger efficiency if one of the signals have large jitter?

Which is the drawback?

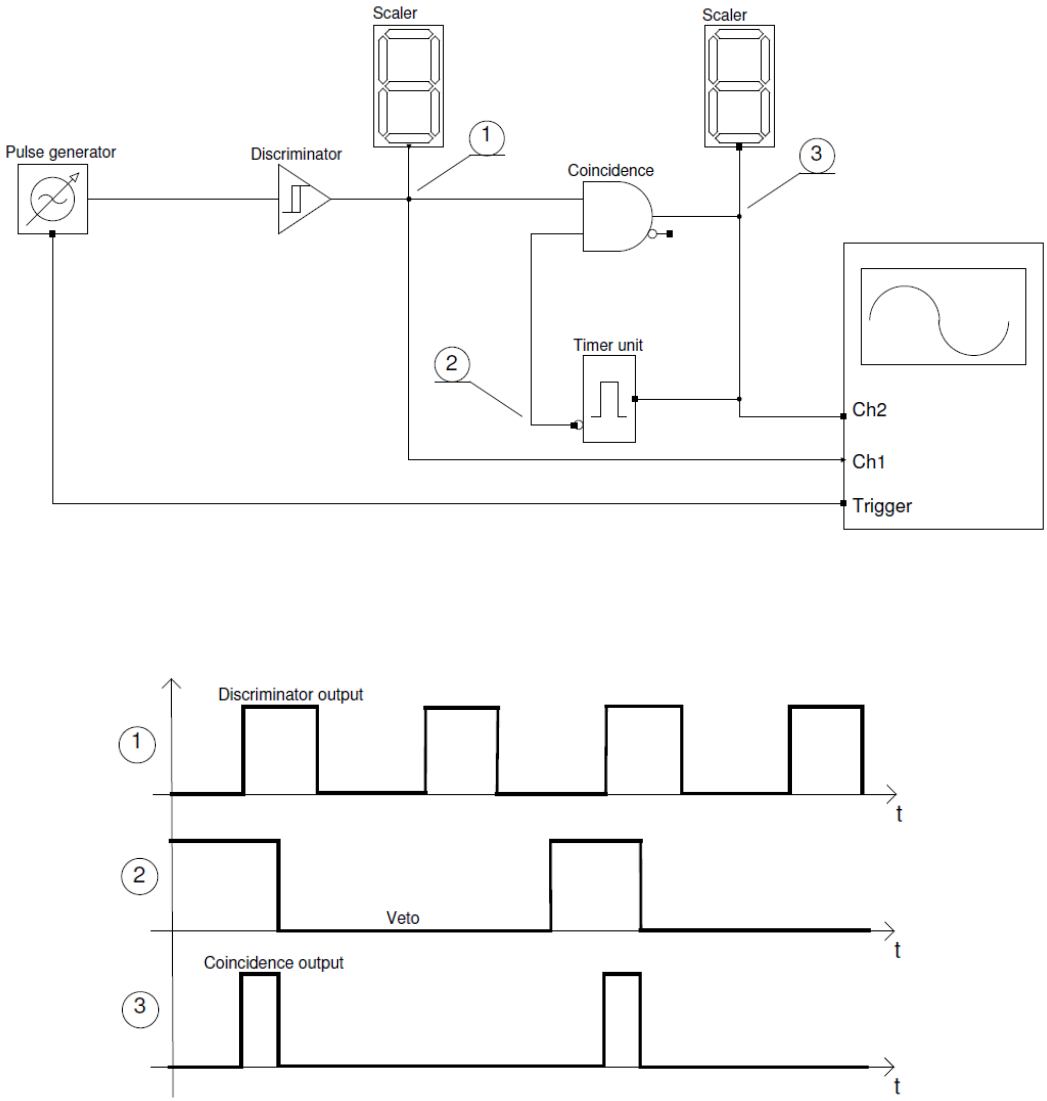


Figure 7: Top: busy logic schema with readout processing time simulated via a dual timer module. Bottom: time diagram of signals at the discriminator output (1), after the veto (2) and at the coincidence output (3).

Part 4: Trigger veto and dead-time

A busy logic can be implemented using the coincidence module and a Dual-Timer Module which simulates a readout system with a fixed processing time (readout dead-time).
Configure one stage of a dual timer module to generate signals with 10 ms width. Then implement the busy logic in a second stage of the coincidence unit as shown in Fig. 7:

- one input of the coincidence unit is the trigger signal;
- to simulate the start of the readout, and so the trigger ACCEPT signal sent to the readout system, use the output of the busy coincidence to drive the timer module (START);
- use the output of the timer as the VETO of the busy coincidence: this is the BUSY signal sent back to the trigger system;
- connect the trigger signals before and after the busy logic to the scaler and check the correct logic.

You can easily make a rate measurement configuring the Scaler to work with a time gate of 1s with a GT+CLR configuration. Compare the trigger ACCEPT rate and the readout rate (after the BUSY) on the scalers.

How do they relate with the timer module setting?

Can you reproduce the numbers using the LIN-OUT of the coincidence unit?

Alternatively you can make an AND between the trigger and the output of the timer (with inverted logic) and not as a veto.

Where is the difference in the logic? Which risk are we taking?

Can you explain the behaviours observed disabling either one or the other input of the coincidence unit?

Appendix: the Constant Fraction Discriminator

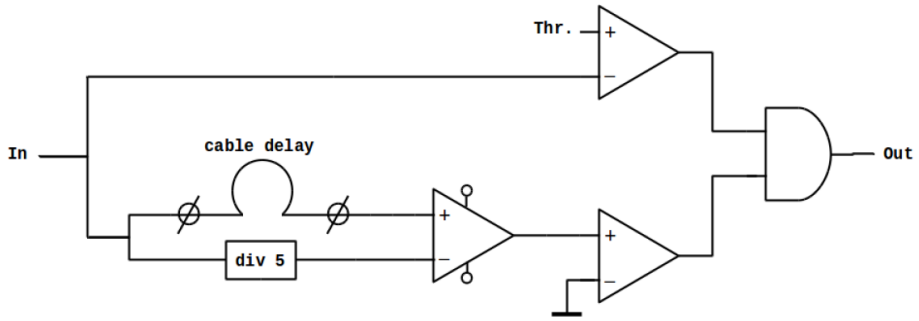


Figure 8: CFD function diagram.

The CFD functional diagram is showed in fig. 8. The input signal is treated in two different discrimination branches, whose results are then merged by the final AND gate. The top branch is a standard threshold discriminator, where the input signal is compared against a (configurable) threshold Thr.
The bottom branch implements instead the constant fraction technique. Technically, the input signal is split: one copy is delayed, while the other is attenuated by a factor 5. The two copies are then subtracted and the final result is compared with a threshold of (close to) zero. In fact, the zero-crossing time of the resulting signal in nearly independent from the input signal leading edge steepness (i.e. the source of time jitter in a standard threshold discriminator).

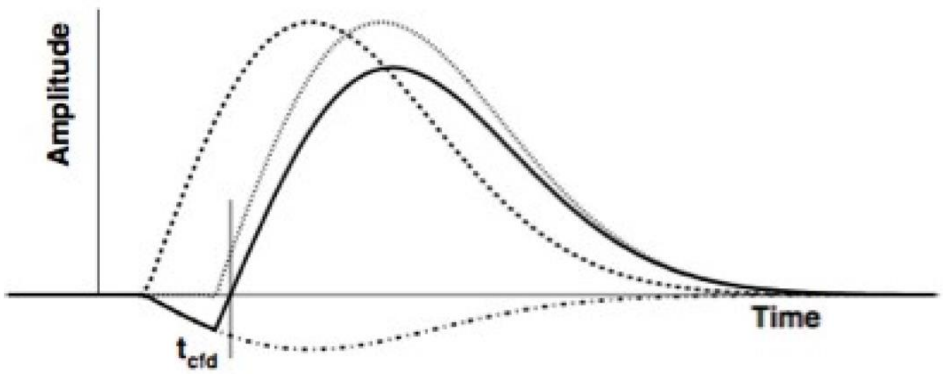


Figure 9: CFD function diagram.

Fig. 9 shows in detail the signals in the bottom branch of the CFD. The input pulse (dashed curve) is delayed (dotted) and added to an attenuated inverted pulse (dash-dot) yielding a bipolar pulse (solid curve). The output of the bottom branch fires when the bipolar pulse changes polarity which is indicated by time t_{cfd}. From a practical point of view, a small threshold, as close as possible, is actually used in the final comparator of

the bottom branch. This is needed to avoid fake signals possibly caused by the noise. Such a small threshold is normally called walk (Z).

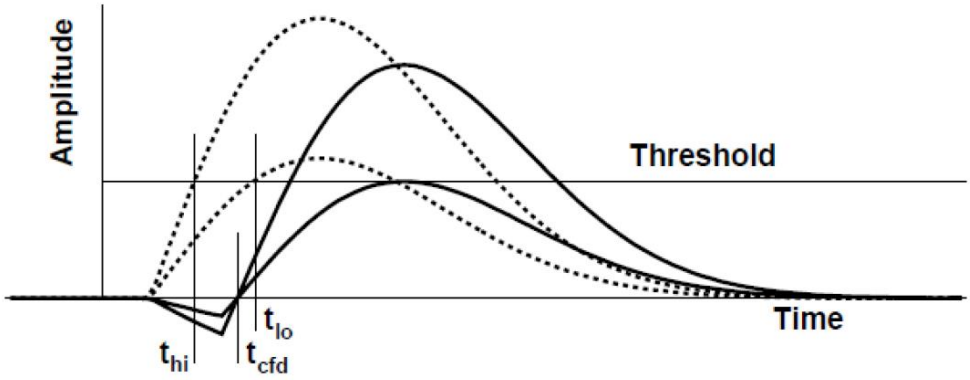


Figure 10: CFD function diagram.

In order to complete the CFD description, the merging of the top and bottom branch signals has to be considered, with the help of fig. 10. In the top branch, the threshold discriminator fires at time t_{hi}, that depends on pulse leading edge characteristics. The bottom branch instead fires at a time t_{cfd}, as discussed above, which is almost constant. Due to the delay introduce in the bottom branch, normally $t_{cfd} > t_{hi}$. Therefore, the overall CFD, defined as the signal generated by the final AND gate, will fires at t_{cfd}, achieving both our requirements:

- only select signal above a given amplitude *Thr*;
- provide an output trigger whose timing is independent from input signal amplitude.

As can be seen in the above figure, the CFD operating principle is not retained for all the possible combinations of configured delay, threshold and input signal amplitude. As the top branch timing depends on the signal amplitude, a small enough signal can make it fire at a time $t_{lo} > t_{cfd}$. In this case the CFD will behave like a normal threshold discriminator, as the output AND gate will be driven by t_{lo}.

3. *Lab 3: Detector and Trigger*

Scintillators, trigger logic, input to readout modules (ADC & TDC)

Introduction

This exercise consists in building the trigger logic and the input signals to the VMEbus readout modules for a detector (exercise #4) using the experience with NIM electronics acquired in exercise #2. The detector comprises two scintillation counters detecting cosmic rays (muons). A schematic diagram of a scintillation counter is shown in
Figure 1. When a charged particle traverses the scintillator, it excites the atoms of the scintillator material and causes light (photons) to be emitted.
Through a light guide the photons are transmitted directly or indirectly via multiple reflections to the surface of a photomultiplier (PM), the photocathode, where the photons are converted to electrons. The PM multiplies the electrons resulting in a current signal that is used as an input to an electronics system. The PM is shielded by an iron and mu metal tube against magnetic fields (of the Earth). The scintillator and light guide are wrapped in black tape to avoid interference with external light. The scintillation counter setup is shown in Figure 2.
The NIM modules used to build the trigger and the input to the readout system and provide the high voltage is shown in
Figure 3.

Outline:

The aim of the exercise is to get an understanding of the detector and trigger logic used in Exercise 4. The signals from two scintillation counters are analyzed using an oscilloscope and transformed into logic NIM signals that allow to build a trigger based on a coincidence between the signals. The coincidence rate i.e. the rate of cosmic muons is counted using a scaler and the charge content of the scintillator signals is measured on the oscilloscope. In addition the inputs to the readout modules (QDC and TDC) are set up.
A schematic diagram of the full trigger and readout electronics is shown in Figure 4.

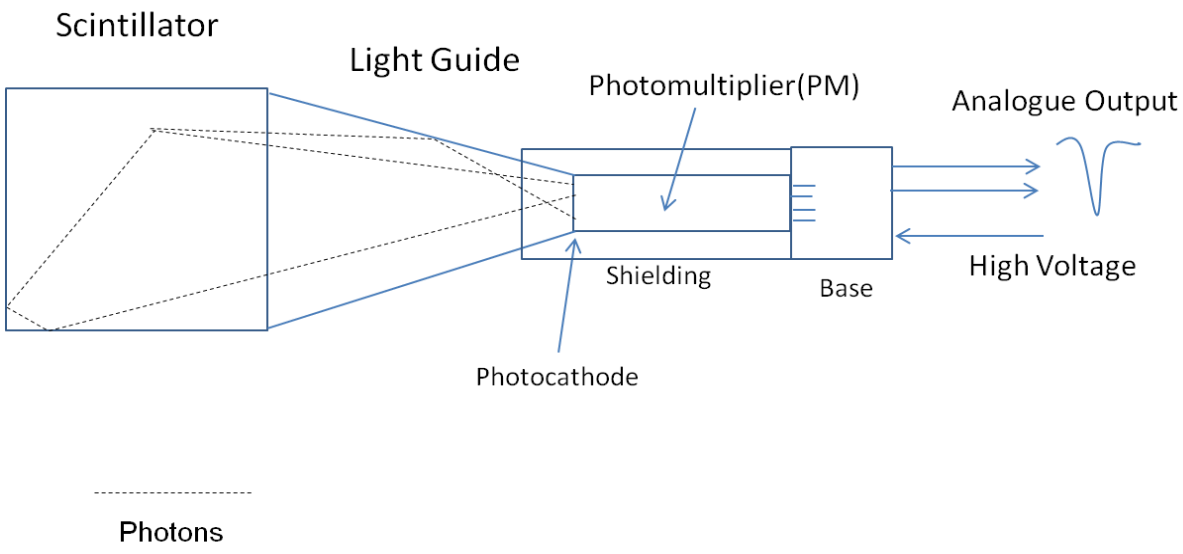


Figure 1. Schematic diagram of a scintillation counter.

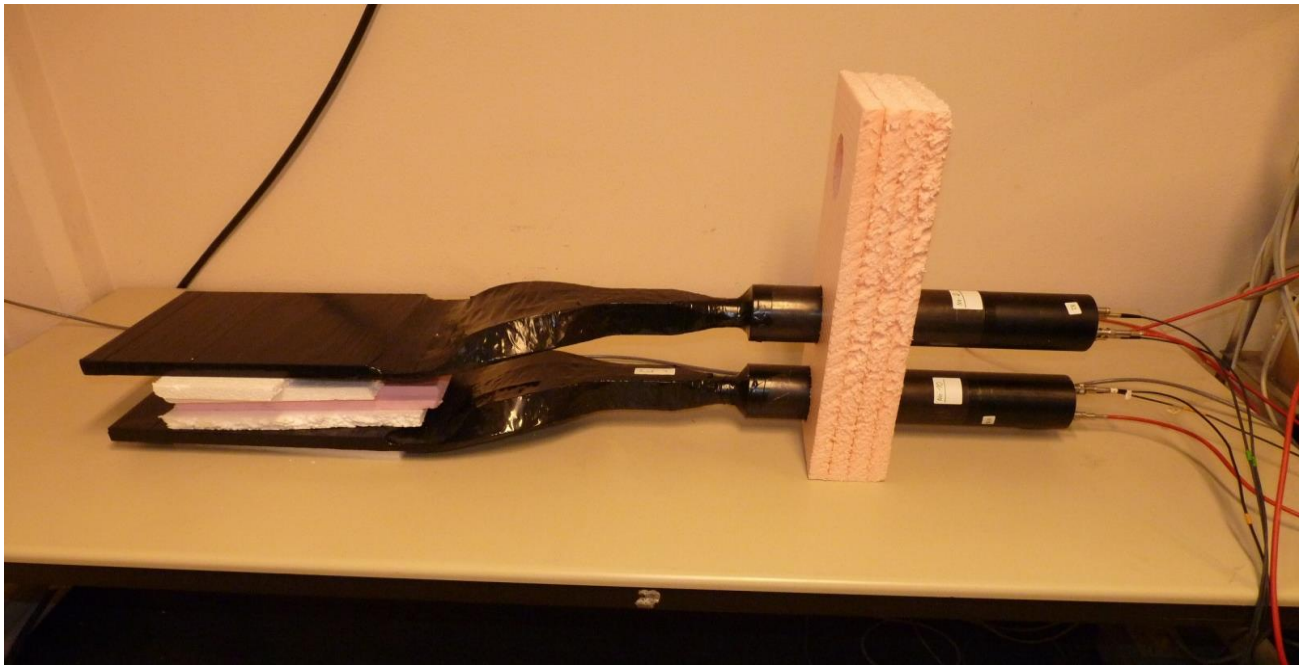


Figure 2. Scintillation counter setup

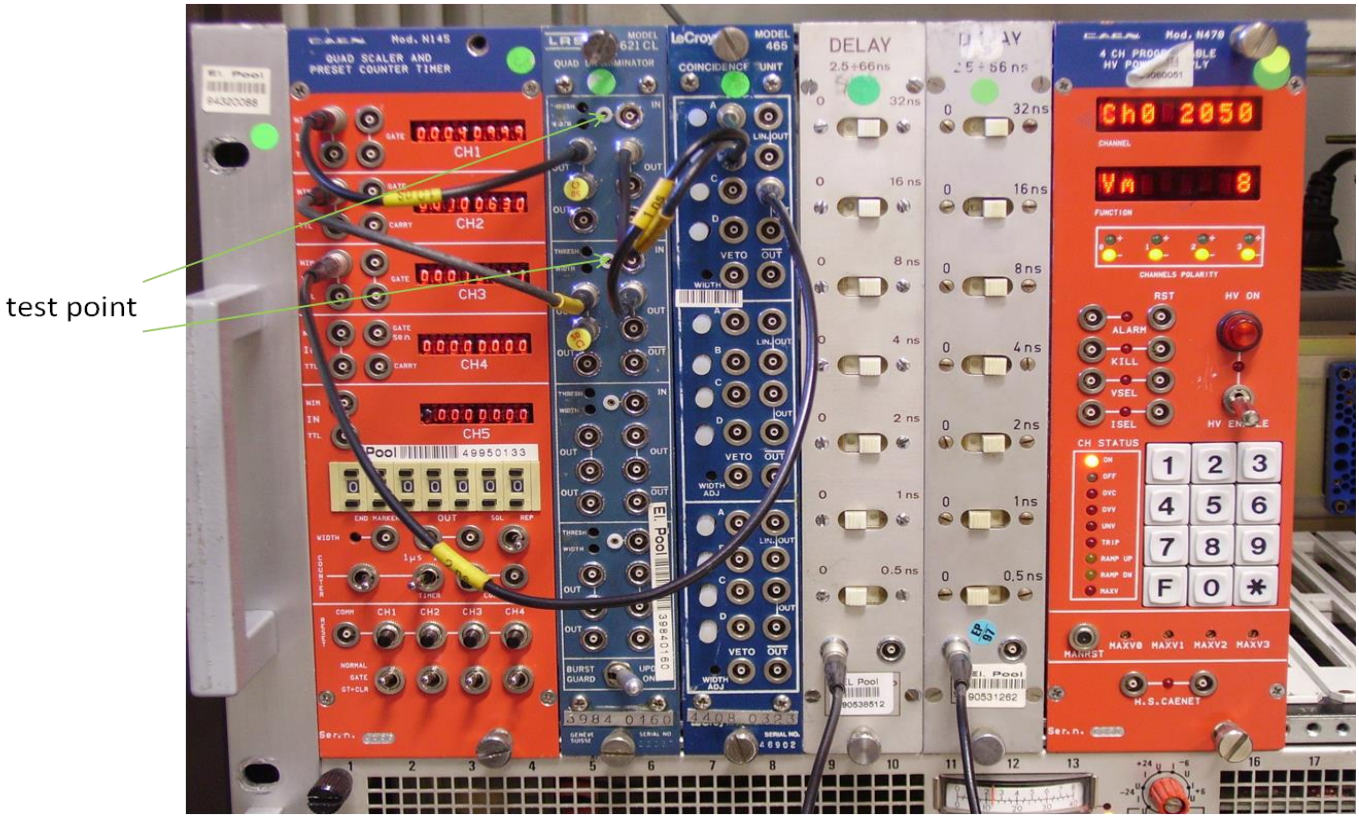


Figure 3. NIM trigger electronics. From left to right: scaler (counter), discriminator, coincidence unit, delay modules and high voltage power supply.

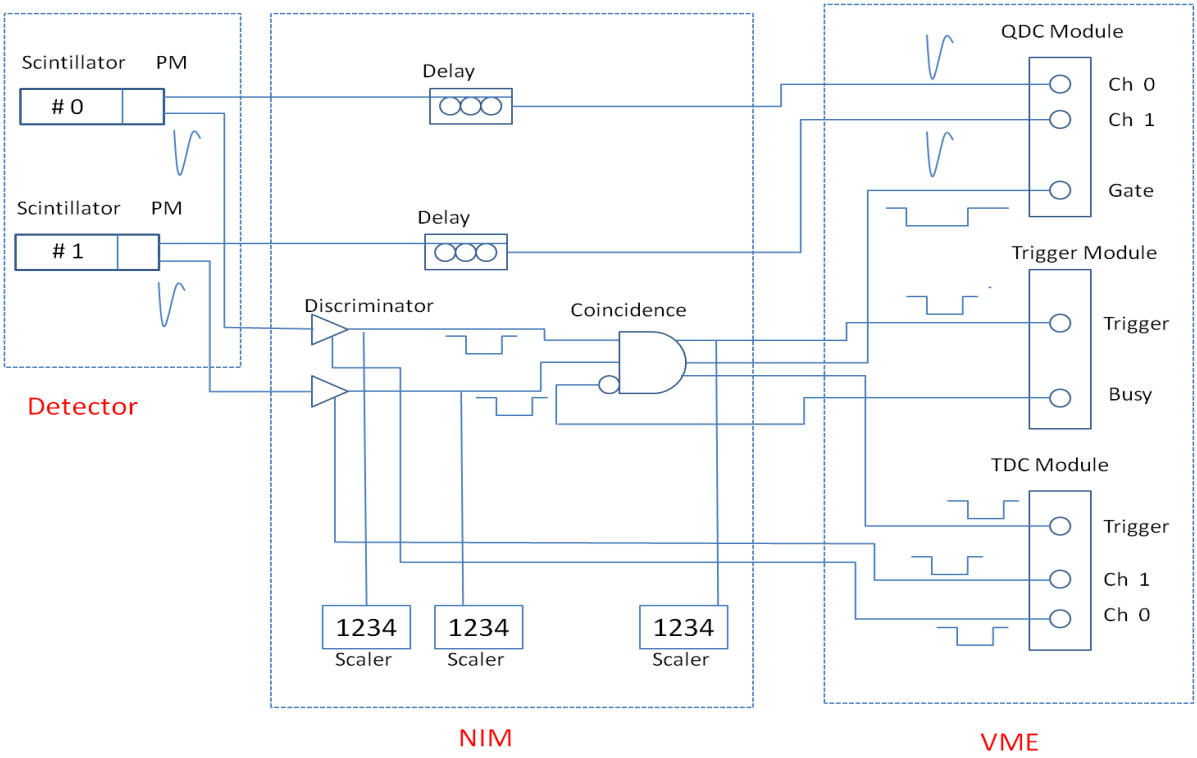


Figure 4. Diagram of the electronics for the detector, trigger and readout of the scintillator counter setup.

Work plan:

Note: whenever there are two parallel outputs from a (NIM) module one needs to make sure that they are both cabled, i.e. either terminated with 50 Ohm or connected to another unit. This ensures that the pulses have the correct NIM voltage levels: 0 and -0.8 Volts.

1. Install the scintillation counters close to each other with maximum overlap between the scintillator areas.
2. Check that the scintillator photomultiplier bases are connected to the N470 NIM high voltage supply.
3. Switch ON the NIM crate.
4. Connect an output from scintillator 0 (the upper one) to an oscilloscope (10ns LEMO), terminate the other output with 50 Ohm.
5. Set the nominal high voltage on scintillator 0 using channel 0 of the N470 HV supply. The voltage is marked on the label glued onto the base. Refer to 0 at the end of this exercise for a short guide to using the N470 HV supply.
6. Look at the signal on the oscilloscope (volts/div ~ 50 mV, time/div ~ 20ns). What is the maximum voltage of the signal?
7. Connect the cable to the input of the first channel of the discriminator.

Connect an output to the oscilloscope (0.5 Volts, 50 ns) and adjust the pulse width to around 100 ns using a small screwdriver (terminate the other output with 50 Ohm), see

8. Figure 3.
9. Connect the output to the first channel of the NIM scaler (N415) using a short LEMO cable (1ns).

Set the discriminator threshold to 50 mV: adjust the voltage on the test point using a DC voltmeter

and a small screwdriver, see

10. Figure 3. The voltage is 10 times the threshold value i.e. the voltage should be around 0.5 Volts. This step may require teamwork.
11. What is the scaler rate?
12. Vary the threshold around 50 mV and check the variations in scaler rate.
13. Repeat points 4 to 11 above for scintillator #1 (the lower one), connecting this scintillator in addition to the one already connected.
14. **Given the scaler rates measured above, what is the probability of random (unphysical) coincidences between pulses from the two scintillators?**
15. Connect an output from each of the two discriminator channels to the oscilloscope and check that they have a timing overlap i.e. are coincident.
16. Connect the cables from the discriminators to the first inputs of the coincidence unit (LeCroy 465) using short LEMO cables (1ns).
17. Connect an output from the coincidence unit to a scaler input. What is the rate? Given that the rate of cosmic muons is about 100 per second per square meter, does the rate make sense?
18. Connect an output of the coincidence unit to channel 1 of the oscilloscope.
19. Connect the (other) analogue output from scintillator 0 to a delay unit (LEMO 10ns) and the output of the delay unit to channel 2 of the oscilloscope.
20. Using channel 1 as a trigger, observe the analogue signal on channel 2. Channel 2 will then show the scintillator signals for the cosmic muons. Assuming that the signal is triangular, what is the charge of the signal? See Figure 5. **Note down the charge. You will need it again in exercise 4**
21. Adjust the delay unit such that the analogue signal falls within the NIM pulse from the coincidence unit: inputs to the charge to digital converter (QDC) in Exercise 4 are now ready (analogue signal and gate).
22. Repeat point 21 for scintillator 1.
23. Connect a cable from the first discriminator to channel 2 of the oscilloscope and check the timing with respect to the output from the coincidence (channel 1). The signal from the discriminator should precede the coincidence. Similarly for the second discriminator. The inputs to the time to digital converter (TDC) in Exercise 4 are now prepared (trigger and timing signals).
24. The signals from the discriminators are sometimes about twice as long as expected. What could the reason be?

Appendix 1: Short User's Guide to the CAEN N470 High Voltage Supply

This is a short list of the most common operations for the N470 High Voltage Supply used in Exercises 3 and 4. The manual can be found at
http://www.caen.it/nuclear/product.php?mod=N470#

To select a channel: F0*(channel number)* e.g. F0*0*
To set the High Voltage on the selected channel: F1*(type value)* e.g. F1*2000*
To read the voltage on the selected channel: F6*
To read the current on the selected channel: F7*
To turn the selected channel ON: F10*

Notes:
The maximum voltage on the channels has been set to around 2300 Volts (on the potentiometers). These can be checked via F13*. The current limits have been set to 2mA (via F2*).

Appendix 2 : Charge of scintillation counter current pulse

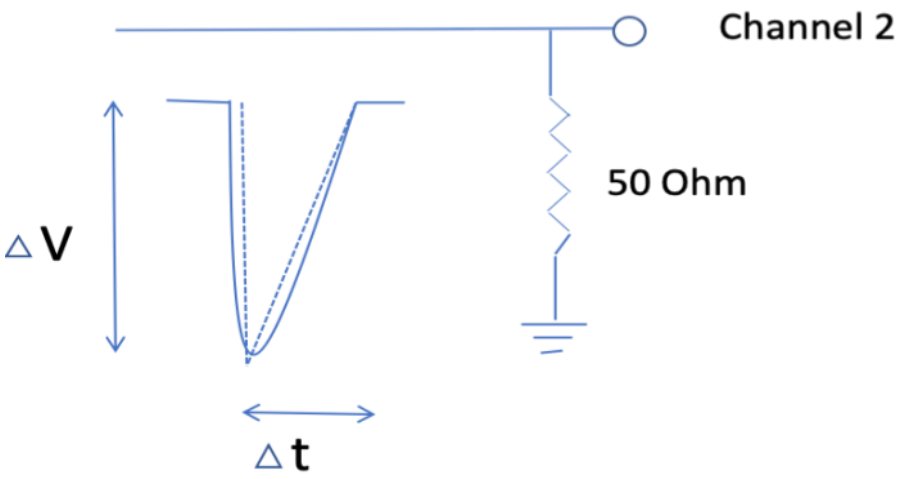


Figure 5. Input to the oscilloscope from a scintillation

4. *Lab 4: A small physics experiment detector, trigger and data acquisition*

Introduction

This exercise comprises all the components of a typical experiment in high energy physics: beam, detector, trigger and data acquisition. The "beam" is provided by cosmic rays (muons) and the detector consists of a pair of scintillation counters, see Figure 2 in Exercise #3. The trigger logic, built in NIM electronics, forms a coincidence between the signals from the scintillation counters which indicates that a muon has traversed the detector, see
Figure 3 in exercise #3. A data acquisition system based on VMEbus is used to record the pulse heights from the scintillation counters and measure the time of flight of the muon. The VMEbus crate is shown in Figure 6 and the VMEbus modules shortly described in 0, 0 and 0. The overall run control and monitoring is provided via software running on a (Linux) single board computer (SBC).

Outline

This exercise is a continuation of exercise # 3. First, standalone programs are executed to give an understanding of the QDC and TDC VMEbus modules. A full DAQ system is then run on a multi-processor configuration, with the readout, run control, GUI and infrastructure on a VMEbus SBC. Event rates and dumps are examined. An event monitoring program produces histograms of the QDC and TDC channel data which allow to compute the charges of the input signals to the QDC and the speed of the cosmic muons.

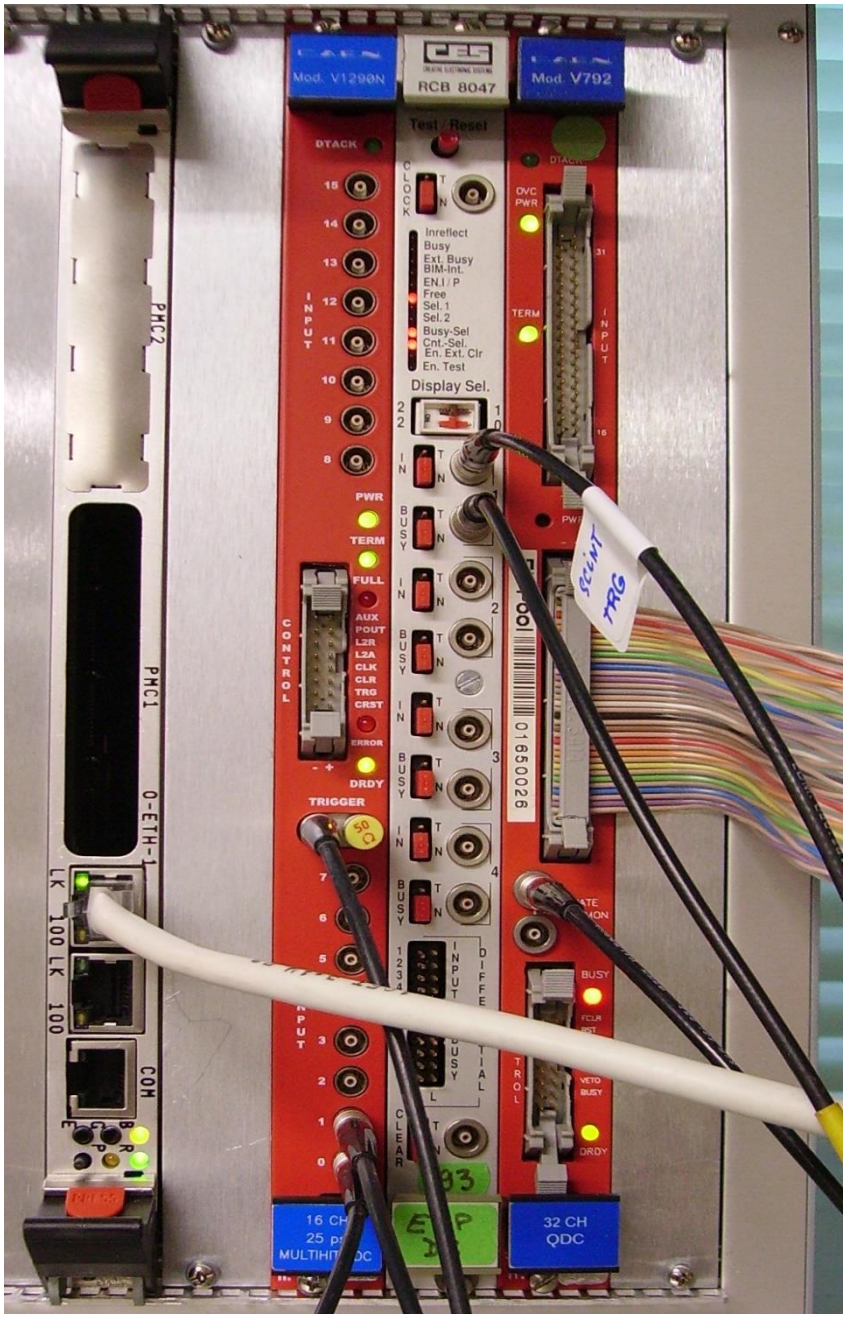


Figure 6. VMEbus data acquisition system: SBC (Single Board Computer), TDC (Time to Digital Converter, Trigger Module (CORBO), QDC (Charge to Digital Converter)

Work plan

- Verify that the detector is working i.e. the scaler counts for scintillator 0, scintillator 1 and the coincidence are counting such that the TDC and QDC receive signals (note for the tutor: if the coincidences are not counting, remove the CORBO busy from the trigger coincidence by pushing the button).
- Login to the SBC as user daqschool, password g0ldenhorn
- Start a Terminal window from the toolbar
- Go to TDAQ directory: **cd ~/TDAQ** and run the command **source ./setup_RCDTDAQ.sh** to define the environment
- Run the program **v1290scope** which is a low-level test and debug program for the CAEN V1290 TDC
  1. Run command**: v1290scope** (Use defaults for the command parameters).
  2. VMEbus base address = 0x4000000
  3. Dump the registers (option 2). Is data ready? (bit DREADY in the status register). What are the values of the match window width and the window offset? See 0.
  4. Configure the TDC (option 3)
  5. Read an event (option 5). How many words are read? (Check in the global trailer). What are the values of the TDC measurements (in ns). Do they make sense? See 0. Exit from the program by choosing menu entry 0.

- Run the program **v792scope** which is a low-level test and debug program for the CAEN V792 QDC
  1. **v792scope**
  2. VMEbus base address = 0x0
  3. dump the registers (option 2). Is data ready? Check also the LED on the module.
  4. read an event (option 5). How many words are read? Which channels have data and which are pedestal (empty) values?
- We now run the full DAQ system
  1. Start the DAQ system: **./setup_RCDTDAQ.sh start**. This script will read the configuration database and start a number of processes on the server: run control, GUI and a number of infrastructure SW components. This is a somewhat long procedure and should result in a message 'OK!'.
  2. Now start a GUI display: **./start_Igui.sh**. The "folders" in the infrastructure panel should be green! You may need help from the tutor here ...
  3. We now go through the run states in order to start a run. But first please obtain a 'Control' access by selecting the 'Control' radio button in the top menu 'Access Control'. The initialize button should become active. Now, click on INITIALIZE and then wait for

the RCDApp (in RCDSegment) to reach the INITIAL state. The readout application is now loaded on the VMEbus processor.

4. Click the CONFIG button followed by OK on the "Remember to ..." dialog box. This configures the VMEbus modules, the CORBO, QDC and TDC.
5. If you don't see the DFPanel tab close to the top of the GUI, click LOAD Panels and load the first panel: DFPanel should now appear in the bar above the Run Control panel.
6. Click START in the control panel (on the left)
7. Data taking should now start. Click on the DFPanel and the L1 button to display the event rate. Is it what you would expect after exercise # 3? Check also the LEDs on the VMEbus modules (the event rate is computed by the Information Service (IS) which periodically sends a command to the Readout Application to obtain the rate which is then retrieved by the GUI).

- Event Monitoring

  This part demonstrates event monitoring. An event monitoring program obtains a sample of events from the readout application and analyses them, in this example by producing histograms of the values from the QDC channels as well as the time difference between the two TDC values. The histograms can then be viewed via the GUI. The code for the monitoring program can be found in ~/RCDTDAQ/RCDMonitor/

  1. Open another terminal window.
  2. cd ~/TDAQ
  3. source ./setup_RCDTDAQ.sh to define the environment.
  4. Run the event monitoring task: ./event_dump.sh -e -1

(-1 means to run forever. I you want only one event, please change it to 1.). Once you have seen the raw data output of the even_dump you can terminate this application with ctrl+c.

5. The first nine words of the data constitute an Event (ROD) header. The following words are the data from the QDC and the TDC. Do you recognize the data?
6. On the terminal start the monitoring program by executing monitor. This program monitors data, like the event_dump program, publishing measurements to the histogramming service.
7. In the GUI click on the OH button (Online Histogram). Click on Histogram Repository, part_RCDTDAQ, RCDMonitor. Double click on the histograms to view them.

Alternatively, using a new terminal execute **source ./setup_RCDTDAQ.sh** followed by **./start_ohp.sh**. This is the online histogramming presenter. In the panel Histograms (on the left) select SCMonitor to view TDC histograms or RCDMonitor to view also the QDC histograms that we are producing.

8. Record the mean values of the QDC histograms and the mean value of the time difference histogram. The time histogram is not centered around zero. Why?

9. The charge that you find in the histogram is not the charge delivered from the PMT to the QDC. What is the reason for that and how can we measure the proper charge?

10. The monitoring of the statistics can be reset by stopping and starting the monitoring program (Ctrl+C to terminate). This restarts the monitoring program described in point 6.

11. Display the histograms of the QDC channels. Record the pedestal values.

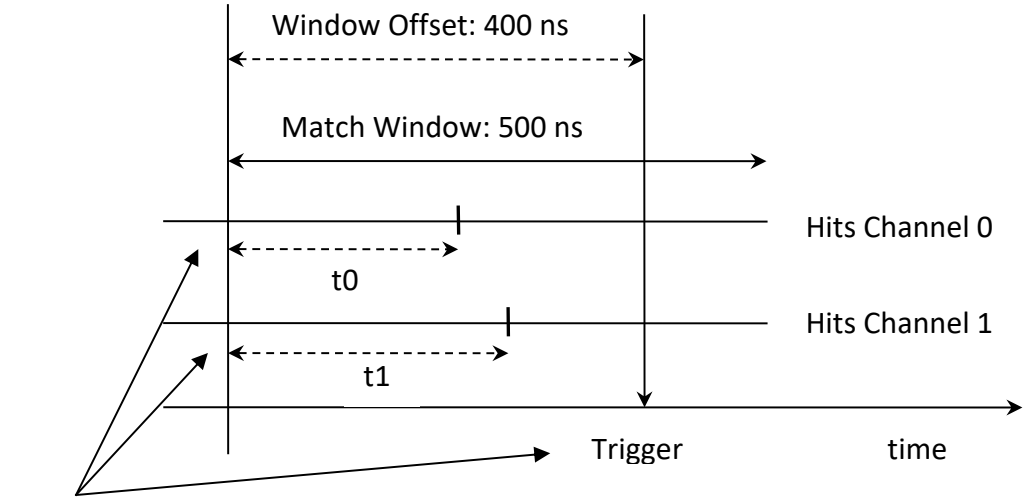
12. Using the formula shown in 0, compute the mean charges of the signals from the scintillators. Do they agree with the results obtained in exercise #3?

- We now want to measure the time of flight of the muons between the two scintillators.
  1. In the histogram for the data from the TDC we already get a Δt. This value, however, is not the time of flight of the muon. Why? How can we modify the set-up in such a way that we can correct the Δt for errors and measure the actual time of flight?
  2. Restart the monitor program from the IGUI per point 10 above. Record the new mean value of the Δt histogram.
  3. What is the difference with respect to the value measured before? Compute the speed of the cosmic muons.

Appendix 1: TDC CAEN V1290 VMEbus module

The TDC is operated in *trigger matching* mode. This means that the TDC measures the time of arrival of the hits on a channel within a *match window*. The TDC receives a trigger and the channel signals as shown in the diagram of the complete setup, Figure 4 of exercise #3 and seen in the picture of the VMEbus crate, Figure 6. A trigger match window is then defined by a window offset with respect to the trigger and a match window size as shown in the figure below. The hits occurring on channel 0 and channel 1 within the match window are recorded by the TDC and the values in units of 25ps stored in the memory of the module.

The module is shown in the photo of the VMEbus crate and the manual for the module can be found at https://www.caen.it/products/v1290n-2esst/ (registration required)



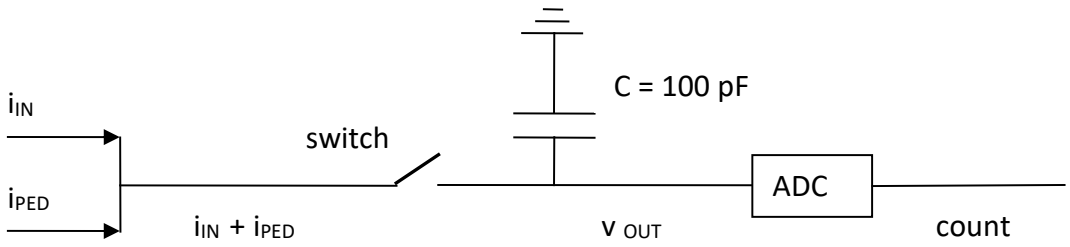
Input signals to the TDC

Appendix 2: QDC CAEN V792 VMEbus module

This page explains briefly how to calculate the charge of the input signal to the QDC from the data readout from the module over VMEbus. The module is shown in Figure 6.

The manual for the module can be found at https://www.caen.it/products/v792/ (registration required)

The circuitry of a channel is shown schematically, below.



The switch is closed as long as the gate input signal is present. The input current is the sum of i_{IN}, the current input to the module via the front panel (from the scintillator), and i_{PED}, a bias (or pedestal) current which is generated internally. The bias current allows to handle input signals with small positive voltage components. When the switch is closed during the time of the gate signal, the input current charges the capacitor C. When the switch is opened again, the voltage across C, v_{OUT}, is converted by an ADC and stored in the memory of the module. The ADC has the property that **one count = 1 mV**.

We now have for the charge of the capacitor:

$Q = C * v_{OUT} = 100$ (pF) * count (mV) = 0.1 * count (pC)

To compute the charge in the signal input to the channel, corresponding to i_{IN}, we have to correct for the pedestal value:

$Q_{IN} = 0.1 * (count - count_{PED})$ (pC)

count = channel data with input signal present

$count_{PED}$ = channel data with input signal removed ($i_{IN} = 0$)

Appendix 3: CES RCB 8047 CORBO VMEbus trigger module

When a NIM signal is sent to a channel on the CORBO, a bit is set in a status register and an interrupt on VMEbus is generated, optionally.

The DAQ process on the VMEbus processor can then execute the code to readout the data from the QDC and TDC modules. In addition, the CORBO generates a busy signal which allows to block further triggers until the readout code is terminated.

The CORBO module is shown in Figure 6.

5. *Lab 5: FPGA programming*

(Ver2014_v01)

INTRODUCTION:

In a lot of digital designs (DAQ, Trigger, …) the FPGAs are used. The aim of this exercise is to show you a way to logic design in a FPGA. You will learn all the steps from the idea to the test of the design.
In this exercise you will:

-discover how we can do parallel applications

-program a FPGA from the design up to the implementation and the test

The boards used are ALTERA development kit (Figure 1) based on a small FPGA (CYCLONE) with multiple additional interface components like audio CODEC, switches, button, seven-segments display, LEDs, ….
and a home-made board (named detector in the following pages) connected to the development kit with a flat cable (figure 2)

The initial design is loaded into the board.
You will follow the example to understand the design flow. Four exercises are proposed to modify the original design functionality.

Figure 1: development kit

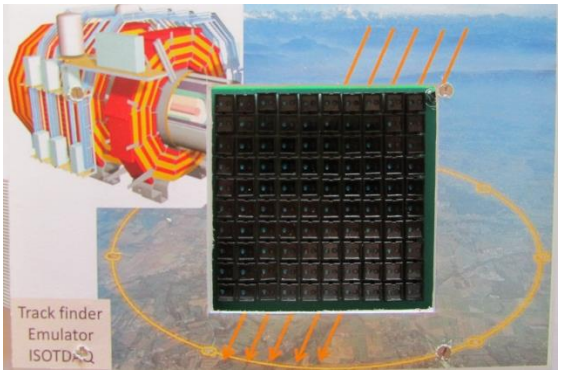


Figure 2: detector

QUICK START:

1) Programs used are: QUARTUS (FPGA tool), ModelSim (simulator), LabView
2) Ask the tutor if you have question(s) or problem(s)

EXERCISE (example)

When you switch on the kit, the initial design is loaded into the FPGA.
On the LabView window, you can see the progression of the marker on the detector.
At the same time, you can see on the two 7-segments LED (the right ones on ALTERA kit) the column and the line number over which the maker is positioned.

DESIGN ENTRY

The design file is named “CII_Starter_Default.bdf” (for all exercises you should work with the same design file).

The design is divided in three parts:

a) A green rectangle which is used to transmit the information to the computer via the RS232 connection to display the trace on LabView.
b) A blue rectangle in which the design generates the clock and the logic to control the detector (see Appendix A for detailed functionality).
c) A red rectangle, which contains the logic to detect the trace. You will change the logic in this rectangle in the following exercises.

The idea of all exercises is to detect a trace. As soon as the trace is detected one 7-segment LED blinks (the third for the right side).
Click on key0 (Altera kit) to stop the blinking. Now generate another trace.
Spend some time to understand how this design works.
Do you understand it?

COMPILATION

This design is the entry of your logic, it should be compiled now; go to QUARTUS *Processing->Start Compilation*.
The design is compiled for the chosen component (Cyclone II).
The compiler executes multiple tasks:

- ✓ logic optimization
- ✓ generates a binary file used to program the FPGA (memory array),
- ✓ extracts the timing between each logic elements used for the timing analyses
- ✓ generate an output VHDL file used for the simulation

SIMULATION

When the compilation is finished, you can check the design with a simulator. To do this you will use ModelSim.
Check in the "Project" TAB if there is a file marked with a bleu "?", if YES, compile it (right-clic on it, Compile-> compile selected)
In the "Transcript" tab, type 'source sim.tcl', ENTER. The simulator opens the waveform, loads the signals, and starts the simulation.
At the end, stimuli and results are displayed in the wave window.
This simulation emulates a trace starting from the top left and finishing at bottom right describing a straight line on the detector.
(The tutor will give you some explanations on the results and the signals shown in the waveform)
Remember where the signal OK goes to "TRUE".

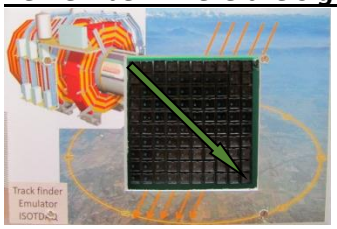


Figure 3: straight line

When you finished with the simulator type 'quit –sim ' ENTER in the "Transcript" tab.

PROGRAM THE KIT

To download the design on the board, (QUARTUS program) go to on *Tools->Programmer* (Check that the Hardware is USB-Blaster, if not ask the tutor).
One file is shown in the window: it is your design. Click on Start .The programmer takes few seconds. At the end, a message appears to inform you that the programming is completed (or not successful: in this case usually the board is switched OFF, or the cable is not well connected).

TEST

Draw a straight line from top left to bottom right to see if the design works well!
Now, you are ready to do the other exercises by yourself.
Good Luck!

EXERCISE I

The exercise above uses the graphic to describe the design. In this exercise, we want to do the same with a text design entry (VHDL).

In the QUARTUS design entry (file "CII_Starter_Default.bdf"), delete the line between inst_graph and JKFF inst_result and connect the output 'result' of "track1"box to the JKFF inst_result with a line.
-Compile the design
-Simulate the design

Go to ModelSim:

- ✓ Compile the file marked with a ? in the "Project" tab (select the file to be compiled – Menu Compile-> Compile selected)
- ✓ Type "quit –sim" in the "Transcript" tab.
- ✓ Type "source sim.tcl " in the "Transcript" tab.

Find out the difference with the previous result (check where the signal OK goes to "TRUE").

Can you explain the difference? Can you modify the file "track1.vhd" to have the same result as in the previous exercise?

-Download the design
-Test the design

EXERCISE II

In this exercise we want to detect a curved trace.

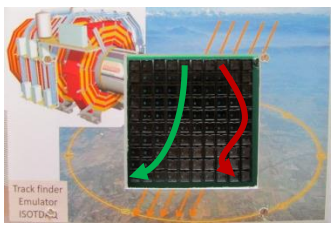


Figure 4

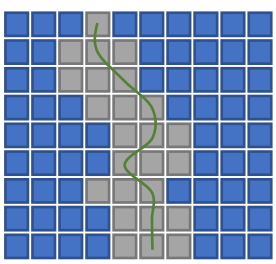


Figure 5: example of trace expected.

In the QUARTUS design entry (file "CII_Starter_Default.bdf"), delete the line between output 'result' of "track1" box to the JKFF inst_result, and connect the output of the "trck_fnd01" box to JKFF inst_result.
The "trck_fnd01" box logic detects only a straight trace. Compile the design and do a simulation:
-Compile the design (QUARTUS)
-Simulate the design

Go to ModelSim, compile the file marked with a ? in the "Project" tab (click on the file to compile – Menu Compile-> Compile selected)
To simulate:

- ✓ Type "quit –sim" ENTER in "Transcript" tab to exist any running simulation.
- ✓ Type "source sim2.tcl" ENTER in "Transcript" tab to start the simulator.

A signal OK becomes true if the logic detects the expected trace (here a straight trace).
In this exercise, you will examine the implementation of the design in the FPGA and see how we can change the results (max. frequency …)

1. In QUARTUS open TimeQuest (Tools -> TimeQuest timing Analyser)
   -double click on Report Fmax Summary ("Tasks" window)
   You can see the maximum frequency of each clocks implemented in the design (Note the max frequency that "scan_clk" can reach)
2. Go back to QUARTUS,
   Open the partition window (Assignments -> Design partitions window)
   Right-click on the partition named "trck_fnd01:instzigzag" (Locate-> Locate in Chip Planner)

Now, you will specify the place where your logic will be implemented:

There is a blue rectangle in the Chip planner (named "trck_fnd01:instzigzag").
Place it where you want (not at the place where the logic is actually implemented) to implement the logic at the next compilation.
Compile the design (Quartus), and execute the TimeQuest (see point 1). Normally the maximum frequency will change.

This give you an idea of the importance of the place of you logic or how to reserve a place if you work in a team (each person will have a reserved place to implement his logic).
NB: For your information, for each clock of the design, the frequency to reach **has to be specified** in a **constraint file.**

EXERCISE III

The exercise consists to modify the "trck_fnd01" box logic to detect any curve trace as in figure 4.
The trace should start at any pixel in the first line and goes to next line going to a pixel adjacent to the pixel of the first line and so forth (figure 5).
To help you, you have to change code in the "mask_build" entity (beginning of the "trck_fnd01.vhd").
-Compile the design
-Simulate the design

Go to ModelSim, compile the file marked with a ? in the "Project" tab (click on the file to compile – Menu Compile-> Compile selected)
To simulate:

- ✓ type “quit –sim’ ENTER in “Transcript” tab to exist any running simulation.
- ✓ type ‘source sim2.tcl’ ENTER in “Transcript” tab to simulate in this exercise.

A signal OK becomes true if the logic detects the expected trace.

-Download the design

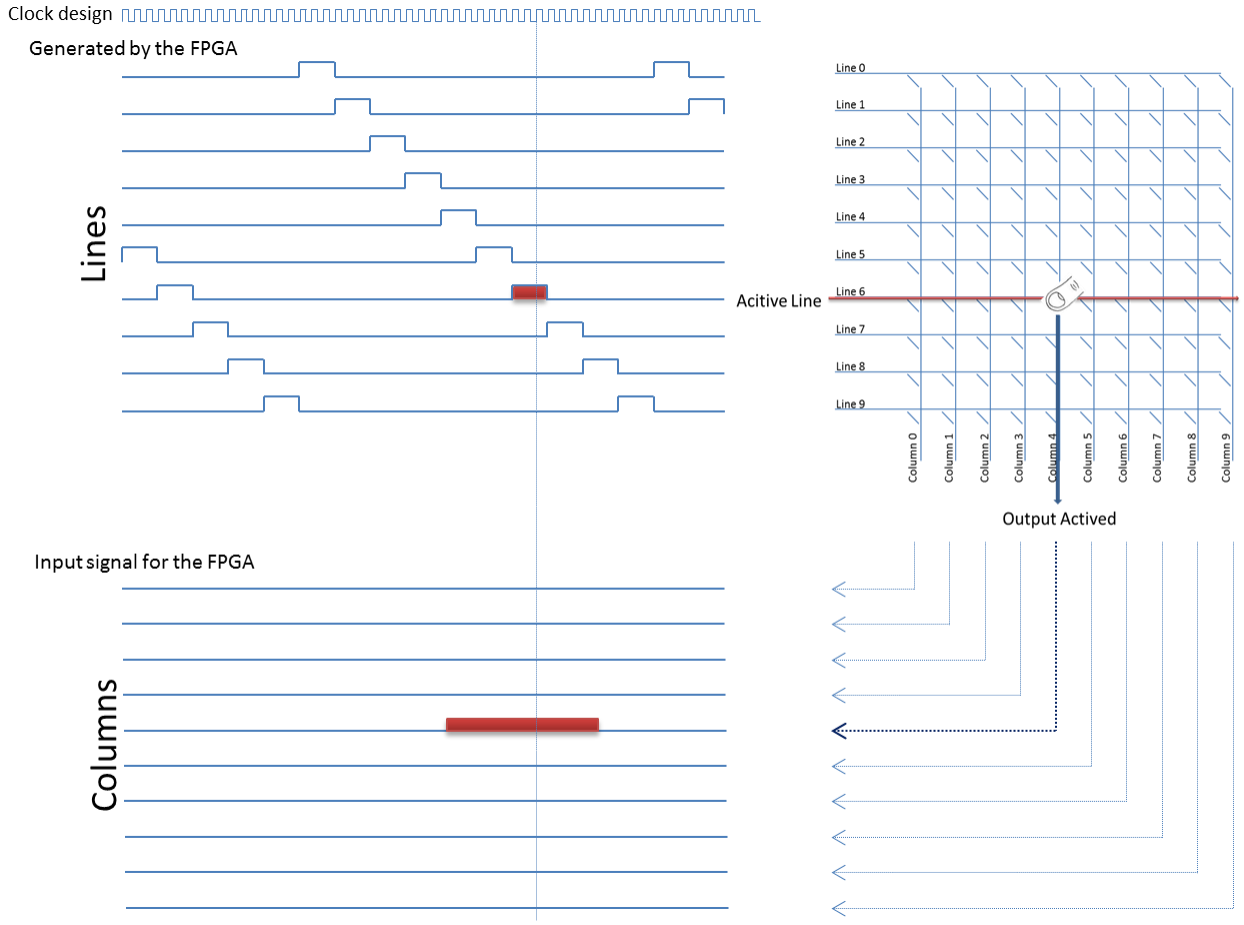
-Test the design

EXERCISE IV

If you have time, you can modify the previous file to detect only the curve trace on right or left (not in zigzag like the red trace in figure 4).

APPENDIX A

The detector is a matrix of 10 lines and 10 columns (100 pixels). Only one line is activated at a time.



When a line is activated the result of each column indicates if the marker is over a pixel. Each line is activated one after the other (0, 1, 2… 8, 9, 0, 1, …) . Each line is activated during 4 clocks cycles. The detection logic checks the result (if pixel is masked by the marker) only during the third clock cycle (signal “check” in the design).

6. *Lab 6: Micro TCA*

Overview

In this exercise you will …
- explore the Micro-TCA technology
- learn about the PCI (express) bus
- write data acquisition software to sample music and display the wave forms

Introduction

In this exercise you will work with modular electronics based on the rather new Micro-TCA standard. TCA stands for Telecommunications Computing Architecture, an architecture that is used in Telco industry to provide high-bandwidth, high-availability solutions. Both Micro-TCA and its bigger brother Advanced TCA (ATCA) will be used in the upgrades of the LHC experiments. Like VME or Compact PCI, Micro-TCA defines a rack—called *shelf* in TCA speak—and boards, called Advanced Mezzanine Cards or *AMCs*. Typical shelves have space for 12 AMCs but we will work with a slightly smaller version. A *Micro-TCA Carrier Hub (MCH)* performs management functions, such as monitoring temperatures and regulating fan speed to provide the necessary cooling. A Micro-TCA shelf may contain a second MCH for redundancy (but we will work with only one). The *backplane* contains high-speed serial links that are suitable for transferring data at rates of 10 Gb/s or more using various protocols. Typically the backplanes have a single or dual-star layout with all high speed-links going from each AMC to the MCH slot(s). The MCH then contains a switch for the desired protocol—in our case PCI Express (PCIe). Other backplane layouts exist with high-bandwidth links between neighboring AMC slots.

The test setup

We are using a small ELMA Micro-TCA shelf containing:
- a built-in power module and a built-in fan
- a backplane with star and mesh connections
- an MCH by Samway (IP 137.138.63.22)
- an AMC containing a Processor running Linux (IP 137.138.63.15)
- an I/O AMC (AMC-ADIO24) providing digital and analog IO
- optionally, an AMC that can generate a programmable load on the crate

Figure 1. *The exercise setup*

We will be working directly on the processor AMC. Keyboard, mouse and screen are directly connected to this card which runs a standard Scientific Linux CERN (SLC) distribution. We will use the network port of this card to communicate with management port of the MCH.

Get to know the setup

Log into the processor AMC:

User: student (ask your tutor for the pwd)

Explore the system using the Webserver

- open Firefox
- Connect to the Samway MCH webserver at IP 137.138.63.22

You can use the webserver to browse the different Field Replaceable Units (FRUs) in the system. You can get information about the units, their voltages and temperatures as well as the valid operating ranges for all these quantities.

Turn on all load groups of the load AMC as shown in the next section. See how the MCH reacts.

Connect to the MCH by telnet

You can see details of the processes in the MCH by connecting to it with telnet (*telnet 137.138.63.22*). Login: user

Try it. (use command *help* to display help).

Note that the backspace may not work. In this case you can use ctrl+h.

To see the fan-speed and fan-levels:

sensor cu 1
cu

As an alternative way to looking at the webserver, you can check the operating ranges and current readings of all sensors via the telnet connection:

sensor amc <slot>

IPMI

The webserver communicates with the MCH through IPMI (Intelligent Platform Management Interface) commands. The MCH either answers to the IPMI commands itself or it forwards the request to a FRU using a dedicated I2C (Inter-Integrated Circuit, often pronounced I-squared-C) link.

You can also directly use the IPMI protocol to talk to a card in the sytem. For example, we can program the load of the load board (produced at CERN) through IPMI. For this we use the program *ipmitool* with the following syntax:

ipmitool -I lan -H <ip_address> -U admin -P ADMIN -T 0x82 -t <AMC_address>
-b 7 -B 0 raw 44 7 0 0 <group_number> <action> 0 15

Where:
<ip_address> is your MCH address
<AMC_address> is the target AMC address (Slot1 = 0x72, Slot2 = 0x74, Slot 3= 0x76 ....)
<group_number> is the LED/Load group number from 4 to 11
<action> if 0xff group ON, if 0x00 group OFF

- Try switching on all the load-groups of the AMC
- See the reaction on the temperature of the AMC by repeatedly running the sensor command (see above)
- When a non-critical threshold is reached the MCH should increase the fan-speed of the crate
- Check this by running the cu command
- Now you should turn off the load groups quickly to avoid that the system overheats (the fan is not powerful enough in this crate)

Explore the Backplane

The telnet prompt has commands that allow to display the links of all FRUs.

- *bpppc* # backplane connectivity
- *pcie* # PCIe status

Try them. The backplane connectivity information needs some decoding. There is some information given at the top of the output. Some further information may be found in the Samway MCH user manual (ask your tutor for a printed copy or find it in /ISOTDAQ/doc/).

AMCs usually have 21 ports, MCHs up to 84 ports. An MCH connects to multiple connectors and is composed of a number of printed circuit boards stacked on top of each other. The boards are called the *tongues* of an MCH. Figures 2 and 3 show the block-diagrammes of the two tongues of the Samway MCH. Figure 4 shows another block-diagram of a MCH with more functionality compared to the one used in the test setup.

Ports are grouped into fabrics. MCHs usually provide *switches* for a certain fabric.

In our test setup, the MCH contains

- a Gigabit Ethernet Switch on Fabric A going to ports 0 and 1 of each AMC.
- a PCI-express Gen3 Switch on fabrics D-G supporting up to 4 lanes, going to ports 4-7 of each AMC.

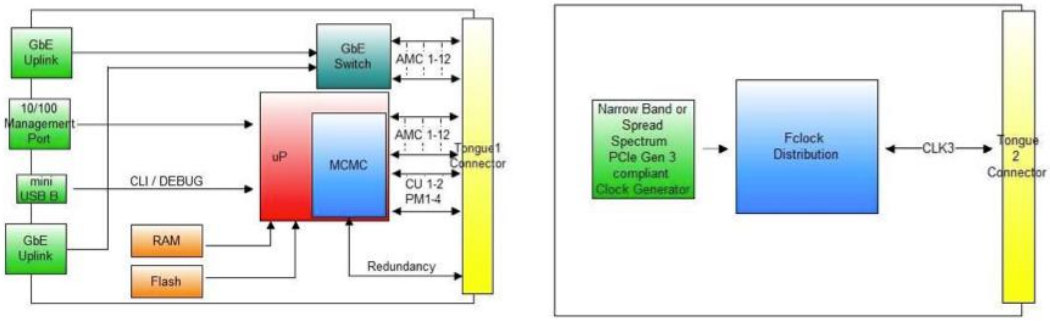


Figure 2. *Block diagram of the Samway MCH.*

Figure 4 shows the backplane of our test shelf, figure 5 shows a typical backplane of a larger shelf with 12 AMCs and redundant MCHs.

Take a look at the backplane with the bpppc command, which AMC has which connectivity? You can also have a look with the pcie.

For some MCH dedicated tools are available to illustrate the backplane connectivity, figure 6 shows one such example, the NATView Backplane viewer.

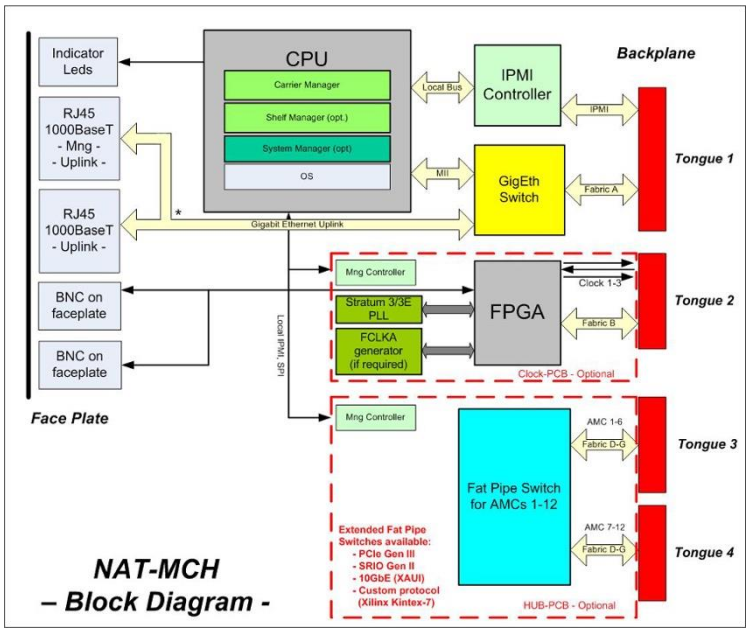


Figure 3. *Block Diagram of the NAT MCH.*

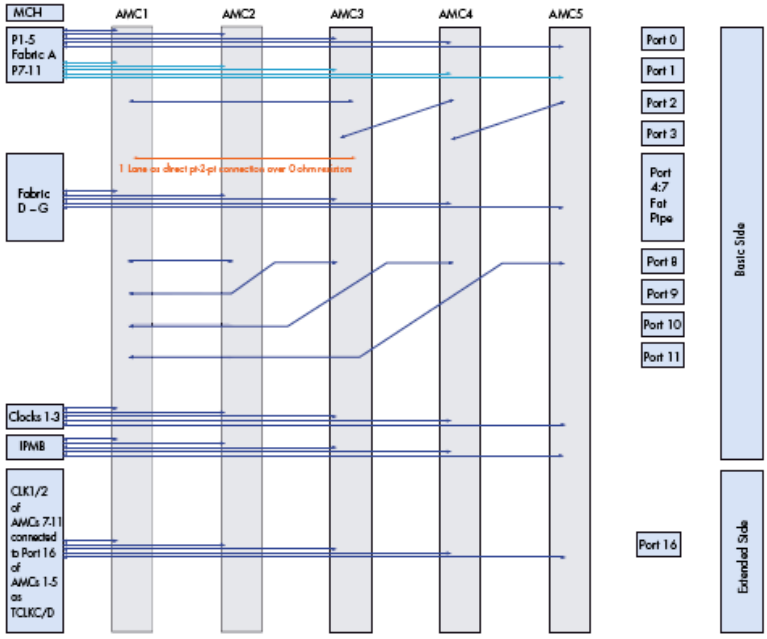


Figure 4. *Backplane of the ELMA blue eco shelf used in the exercise.*

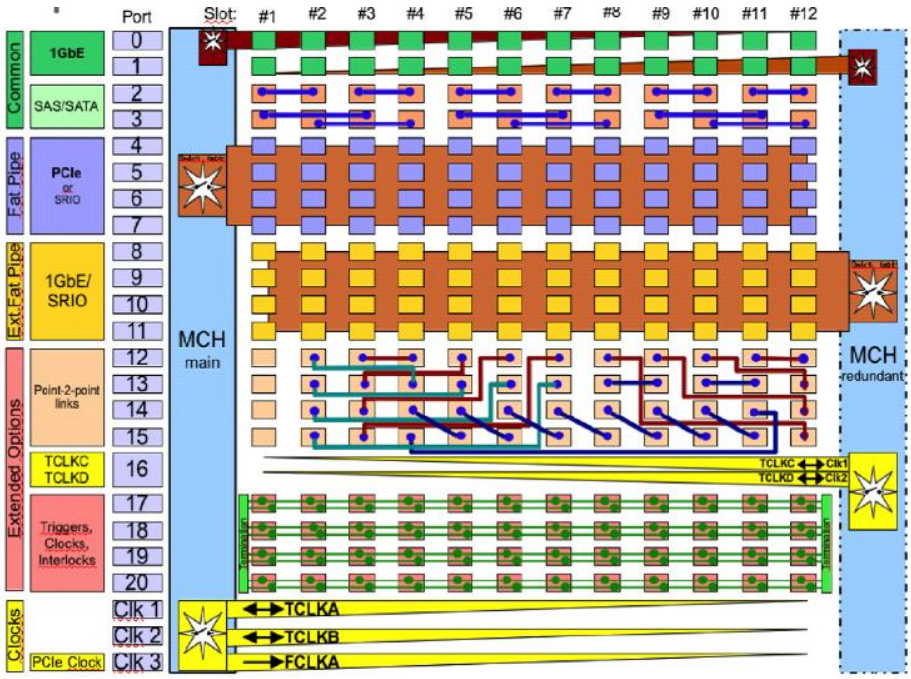


Figure 5. *Backplane of a typical larger Micro-TCA crate with 12 AMCs. (not used in the exercise)*

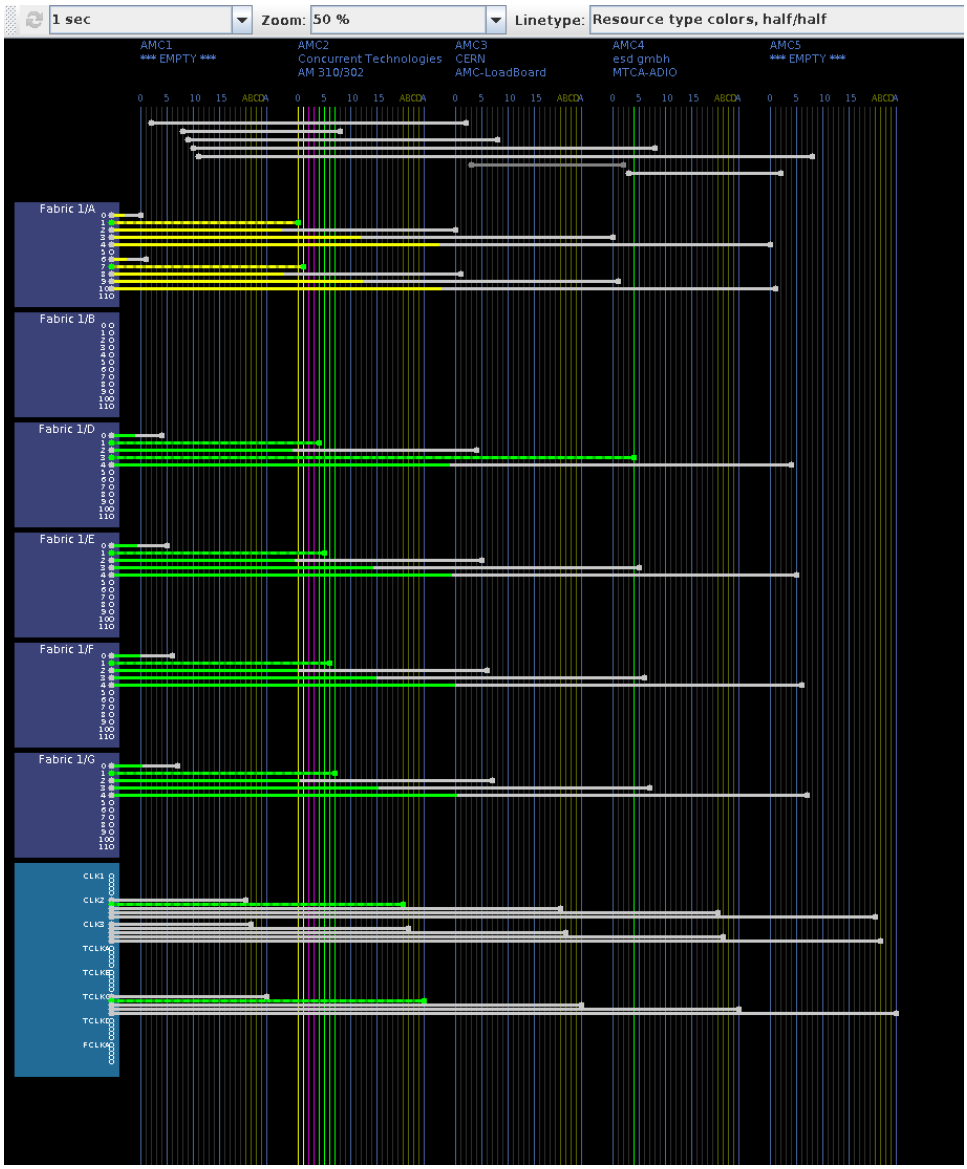


Figure 6. *NAT Backplane viewer*

PCI Express

The MCH in our test system provides a PCIe switch. Cards in the shelf may use it to communicate with each other. In our system, the Processor AMC communicates with the IO AMC via PCIe. Unlike its predecessor, PCI, PCI express is a serial link. Up to 32 serial links may be combined to form a link. Table 1 shows the speeds in Giga-transfers (GT) per second per lane. Physically, PCIe links are point-to-point, as opposed to a bus topology used in PCI. Data are transferred in packets (like in Ethernet) with data integrity checks, re-transmissions and flow control. PCIe switches are used to connect multiple devices to a single controller (called *root complex* in PCIe). Despite all these differences in the lower link layers, PCI-express is software compatible to PCI.

Table 1. Speed of PCI-express

	per lane	**per lane**
Gen-1	2.5 GT/s	250 MB/s
Gen-2	5 GT/s	500 MB/s
Gen-3	8 GT/s	985 MB/s
Gen-4	16 GT/s	1970 MB/s

To discover the bus structure and devices, you can use the *lspci* tool.

For example, try: *lspci -tv* to display the bus in a tree structure
or try options –*v* and -*vv* to display detailed information about the devices.

Try to locate the IO AMC. Find its bus address and its base address.

Hot Plugging (demo by your tutor)

In a Micro-TCA system, AMCs may be hot-plugged (i.e. exchanged without switching the shelf off). We will try hot-plugging the IO AMC card. Since the IO AMC is a PCI device connected to the Processor AMC, we have to let the Processor AMC know.

Together with the tutor, try these steps (you need to be *root* on the machine):

- show the PCI devices with *lspci*
- Pull gently on the black lever
- wait till the blue light is on
- pull the card out by its lever
- repeat *lspci* (did anything change ?)
- push the card back in
- wait till the blue light is on
- push the black lever in
-
- *echo 1 > /sys/bus/pci/devices/[address]/remove*
- (*address* is the address of this device: *PCI bridge: PLX Technology, Inc. PEX 8111 PCI Express-to-PCI Bridge*)
- repeat *lspci* (did anything change ?)
- *echo 1 > /sys/bus/pci/rescan*
- repeat *lspci* (did anything change ?)

Build your own digital scope

Now let's get our hands dirty and do some programming. You will use the Analog-to-Digital converter on the IO/AMC to repeatedly sample an analog input channel and to display the waveform of the sampled signal. The A/D converter continuously samples its input at a programmable frequency. The acquired data may either be polled or transferred to host memory by Direst Memory Access (DMA).

- First have a look at the provided example program *adio_scope.cpp (in /home/student/amc_adio/src).* See how the program maps the address space of the IO AMC into the Processor AMC's memory space and how it then addresses the IO AMC's registers by simple read and write operations to a data structure. Run the program (from */home/student/amc_adio/bin*) and play with it. You can recompile it by running *make* in */home/student/amc_adio.*

- The scope should sample Analog Input 0 at 44.1 kHz. Have a look at the documentation (Hardware Manual) of the IO AMC and find out how to set up the ADC to sample at this frequency.

    - First set up the timestamp counter to provide timestamps in microseconds (register DIVMODE)
    - Then find out how fast you can poll a register of the IO AMC. Is polling fast enough to do a scope working at 44.1 kHz?
    - Now set up the IO AMC to sample Analog input 1 at 44.1 kHz You will need to set up registers FGENAB and ADCMODE.

- After setting up the card, your program should acquire a few hundred samples from the ADC.

- Your program should produce a file with lines of two columns, containing a timestamp in microseconds and the voltage in volts (separated by a space). You can start from the routine *acquire_shot()* which already provides parts of the solution.

- A ROOT program to plot you file is available. So you don't need to worry about producing the graphics. The program will let you choose the file name to plot.

  root –l
  root [0] .x /home/student/gui_cint.cpp

- As an input signal, you can connect the provided head-phone jack to your smart phone and play some music (you'll have to turn up the volume). Or you download a function generator onto your phone – for example RADONSOFT Signal Generator. (No smart phone in your group? Ask you tutor.)

References (.pdf files available in /ISOTDAQ/doc)

- Short intro to uTCA. *MicroTCA_ShortOverview.pdf*
- http://en.wikipedia.org/wiki/PCI_Express - *PCIExpressWikipedia.pdf*
- More info about PCIe: http://xillybus.com/tutorials/pci-express-tlp-pcie-primer-tutorial-guide-1 *PCIExpress_Xillybus[1-3].pdf*
- Info about the shelf: *ELMA_BlueEco_Shelf.pdf*
- MCH doc: *Samway_MCH_Usermanual_Rev_1.6*
- Manual for the IO AMC: *HardwareManual_IO_AMC.pdf*

7. *Lab 7: System Development using LabVIEW*

Introduction

LabVIEW is systems engineering software for applications that require test, measurement and control, with rapid access to hardware and data insights. It uses a graphical approach to visualize every aspect of the application, including hardware configuration, measurement data and analysis. LabVIEW excels at acquiring data from electronic measurement systems, such as that shown schematically in Fig. 1.

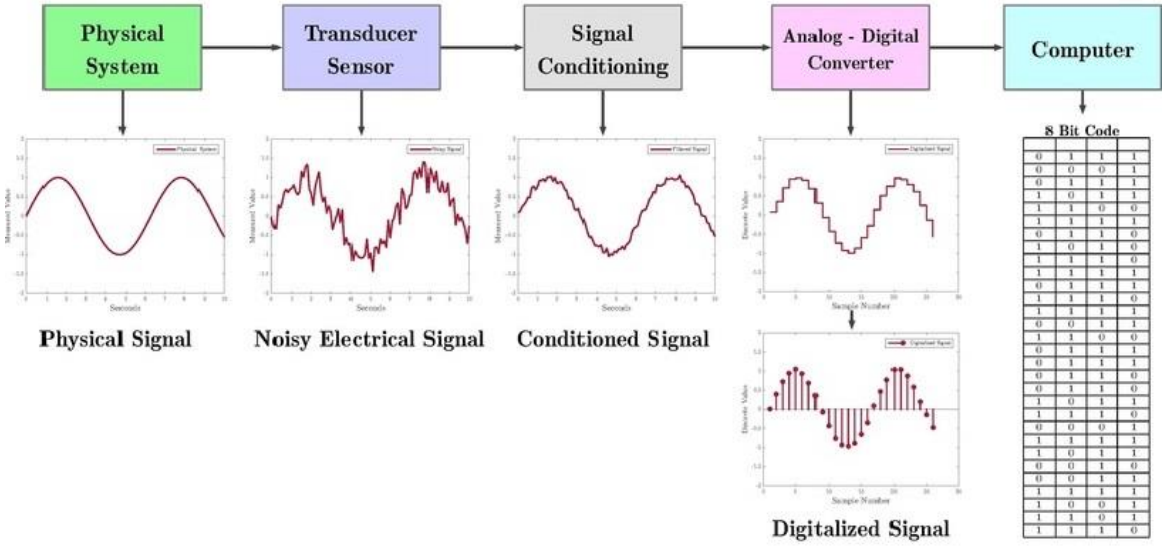


Figure 1: Data acquisition system (from Wikipedia)

The sensor, or transducer, converts a physical quantity (temperature, pressure, accelerate etc) into an electrical signal. Conditioning of this signal, usually with analogue electronics such as amplifiers or filters, is sometimes necessary. The resulting signal is converted into the digital domain using an analogue-to-digital converter (ADC) and a computer system can then process and store these digital data. The numbers and the physical types of the measured signals can all be different, as can the processing algorithms. The computer can also generate digital data, such as an output of a control system, which can drive actuators (mover systems, valves etc) or be converted into an electrical signal via a digital-to-analogue converter (DAC). If all types of signals could be processed then any measuring problem could be solved. Modular instrumentation allows a system to be built that can process many different types of signal.

National Instruments (NI), the producer of LabVIEW, has been developing measurement equipment since the 1980s. They make modular, programmable hardware for a wide range of applications, from acquiring simple analogue signals at a few Hz, to complex FPGA-based high-speed control systems capable of running real-time software for high reliability.

This laboratory session makes use of both LabVIEW and a NI Compact DAQ chassis (cDAQ-9178), into which are installed several types of input/output module, both analogue and digital, summarized in Table 1.

Table 1: Function of Data Acquisition modules

Device	*Number of channels*	*Application*
NI-9211	4	Thermocouple readout 24 bit
NI-9474	8	High-speed digital output
NI-9263	8	General-purpose analogue +/- 10 V output
NI-9205	16/32	General-purpose +/- 10 V input (16 differential, 32 single-ended)

The cDAQ chassis has the following (programmable) internal electronics system: four 32-bit general-purpose counters, FIFO for analog/digital inputs (with 127 depth per slot), FIFO for digital outputs (2047 samples), clock generators (base clocks of 20 MHz, 10 MHz and 100 kHz with divisors: 1 to 16), digital trigger circuit. Using LabVIEW, the chassis can be programmed to control the modules (a typical set-up is shown in Fig. 2), and a computer can then visualize the acquired data and perform analysis on it.

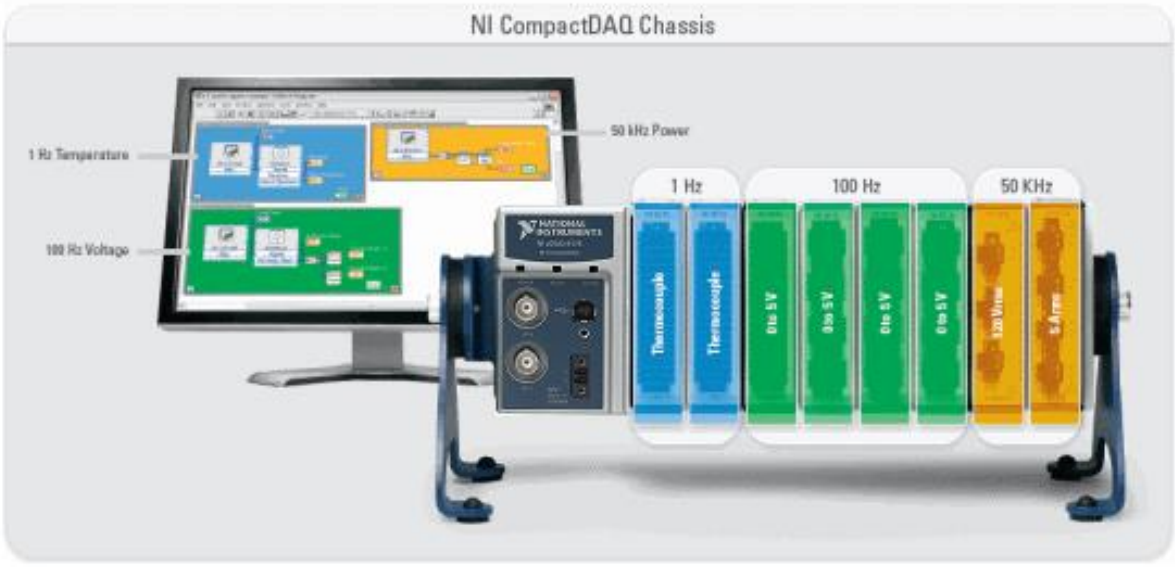


Figure 2: Typical CompactDAQ setup controlled with LabVIEW

Answer the following questions:

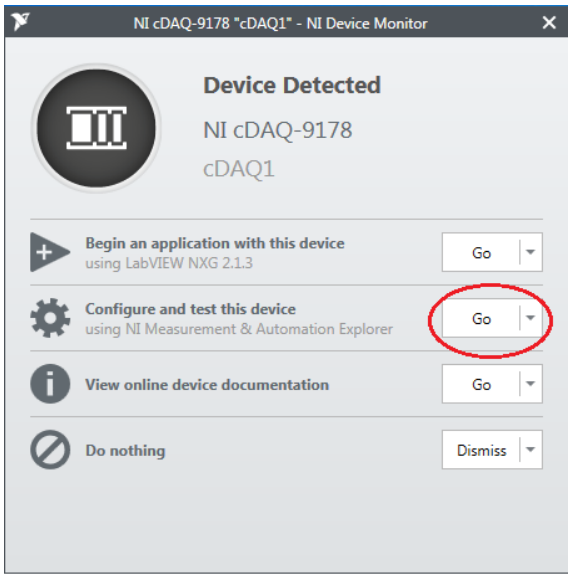
- If this chassis contains only some counters and FIFOs why does it cost approx 1000 €?
- What is the advantage of this modularity when developing a measurement system?
- What would be the problem if we would like to use this system as a control system or as a real-time signal processing application? How would you solve this problem?
- For which application would you choose a real-time configuration? Why?

Exercise 1: Make a basic measurement with CompactDAQ

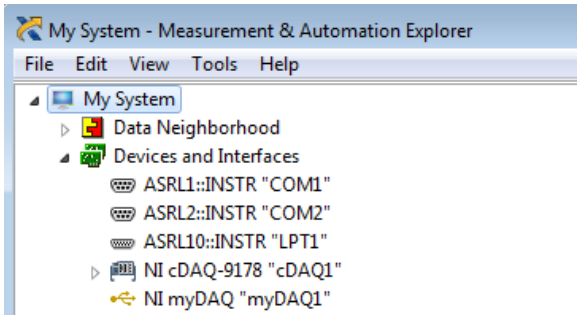
This exercise uses LabVIEW and NI CompactDAQ to quickly set up a system to acquire and display temperature data.

Set up the Hardware

1. Ensure the NI CompactDAQ chassis (cDAQ-9178) is powered on.
2. Connect the chassis to the PC using the USB cable.
3. The NI-DAQmx driver installed on the PC will automatically detect the cDAQ chassis and bring up the following window.



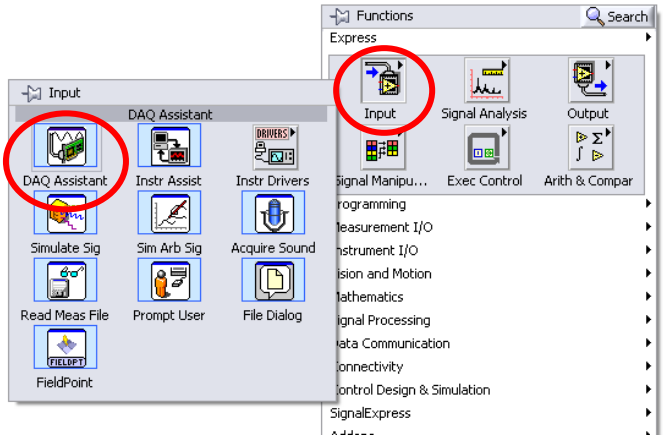
4. Select *Configure and test this device using NI Measurement & Automation Explorer*. NI Measurement & Automation Explorer (MAX) is a configuration utility for all National Instruments hardware. It can take several seconds to start, depending on how many devices are connected to the PC.
5. Within MAX, the *Devices and Interfaces* section under *My System* shows all the National Instruments devices installed and configured on the PC. By default, the NI CompactDAQ chassis NI cDAQ-9178 shows up with the name 'cDAQ1'.



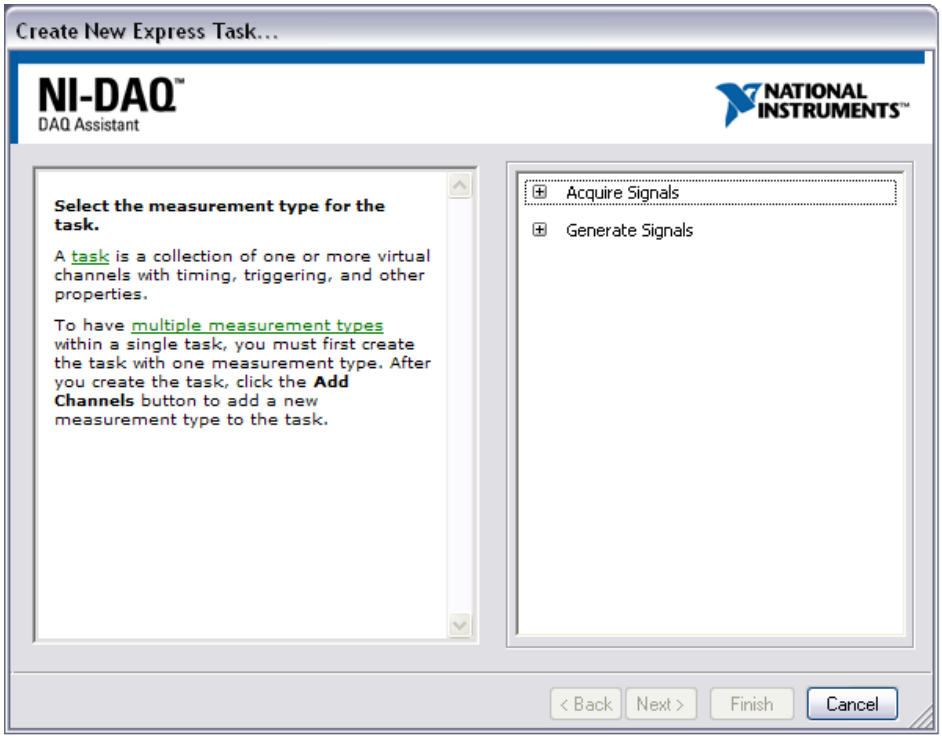
6. Click the triangle to the left of the cDAQ entry to show the modules contained in the chassis.
7. Right-click on NI cDAQ-9178 and select *Self-Test*.
8. The cDAQ passes the self-test, meaning it has initialized correctly and can communicate with the modules, and is ready to be used in a LabVIEW application.

Create a LabVIEW Application

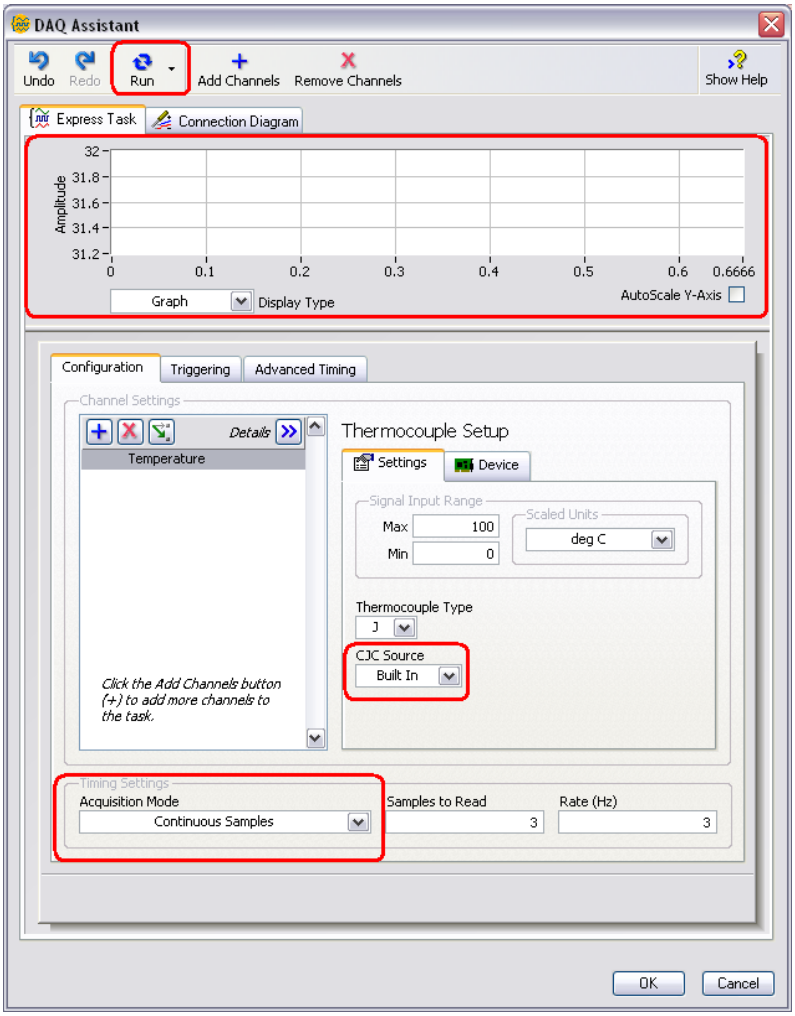
9. Create a new VI from the Project Explorer. Right click on the Exercises folder and select **New» VI**. Once opened, Save the VI in the Exercise folder under the name '1-Basic Measurement.vi'.
10. *(Optional)* Press <Ctrl +T> to tile front panel and block diagram windows.
11. Pull up the Functions Palette by right-clicking on the white space on the LabVIEW block diagram window.
12. Move your mouse over the **Express» Input** palette, and click the DAQ Assistant Express VI. Left-click on the empty space to place it on the block diagram.



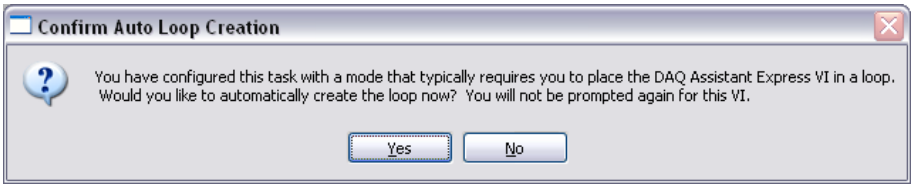
13. The Create New Express Task… window then appears:



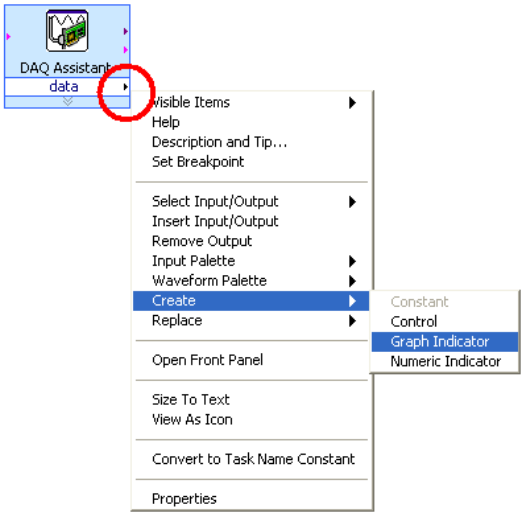
14. To configure a temperature measurement application with a thermocouple, click on **Acquire Signals» Analog Input» Temperature» Thermocouple**. Click the + sign next to the cDAQ1Mod1 (NI 9211), highlight channel ai0, and click Finish. This adds a physical channel to your measurement task.
15. Change the CJC Source to Built In and Acquisition Mode to Continuous Samples. Click the Run button. You will see the temperature readings from the thermocouple in test panel window.



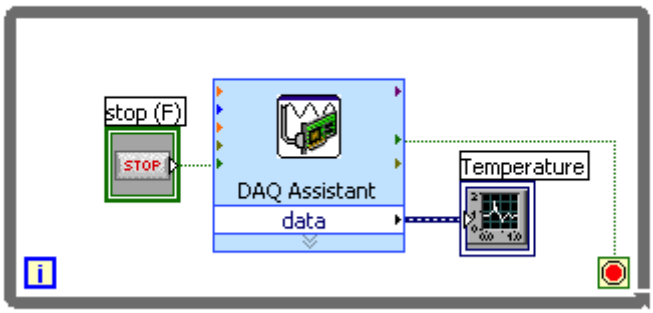
16. Click Stop and then click OK to close the Express block configuration window to return to the LabVIEW block diagram.
17. LabVIEW automatically creates the code for this measurement task. Click Yes to automatically create a While Loop.



18. Right-click the data terminal output on the right side of the DAQ Assistant Express VI and select **Create» Graph** Indicator. Rename "Waveform Graph" to Temperature.



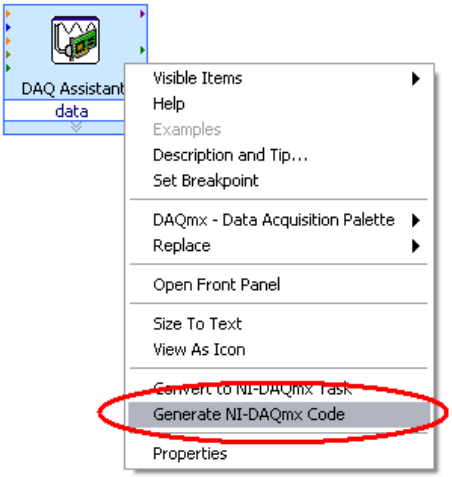
19. Notice that a graph indicator is placed on the front panel.
20. Your block diagram should now look like the figure below. The while loop automatically adds a stop button to your front panel that allows you to stop the execution of the loop.



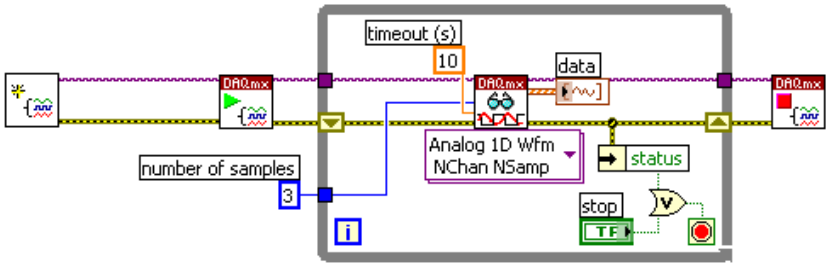
Additional Steps

Express VIs make creating basic applications very easy. Their configuration dialogs allow you to set parameter and customize inputs and outputs based on your application requirements. However, to optimize your DAQ application's performance and allow for greater control you should use standard DAQmx driver VIs. Right Click on block diagram Functions» Measurement I/O Palette» NI-DAQmx.

21. Before you generate DAQmx code you need to remove all the code that was automatically created by the Express VI. Right click on the while loop and select "Remove While Loop." Then click on the Stop button control, and press the Delete key to remove the Stop button. Repeat actions for Temperature Graph as well as any additional wires that may remain. You can press <Control + B> to remove all unconnected wires from a block diagram.
22. Convert Express VI code to standard VIs. While not all Express VIs can be automatically converted to standard VIs, the DAQ Assistant can. This will allow for greater application control and customization. Right-click on the DAQ Assistant Express VI you created in this exercise and select "Generate NI-DAQmx Code."

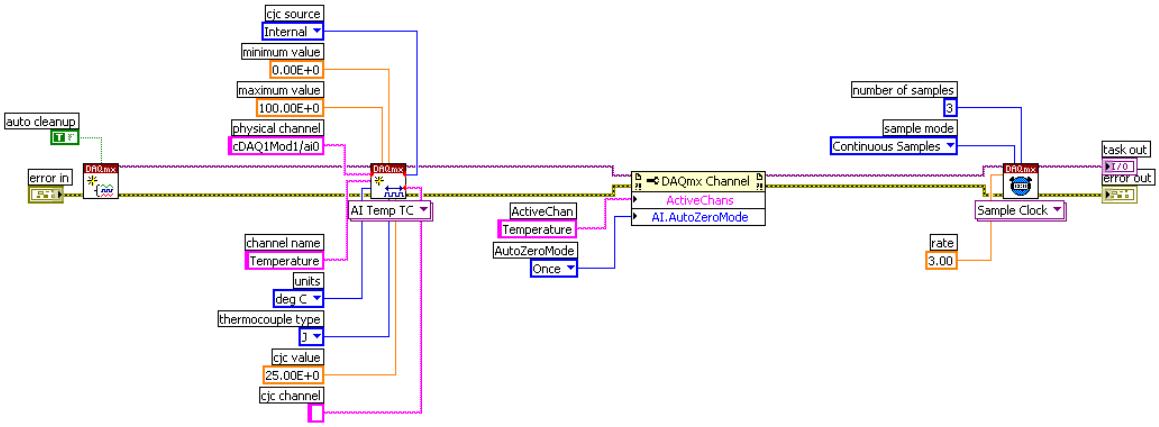


Your block diagram should now appear something like this:



The Express VI has been replaced by two VIs. We'll examine their functionality in the following steps.

23. Open Context Help by clicking on the Context Help icon on the upper right corner of the block diagram. Hover your cursor over each VI and examine their descriptions and wiring diagram.
24. DAQmx Read.vi reads data based on the parameters it receives from the currently untitled VI on the far left.
25. Double-click on the untitled VI and open that VI's block diagram (code shown below).



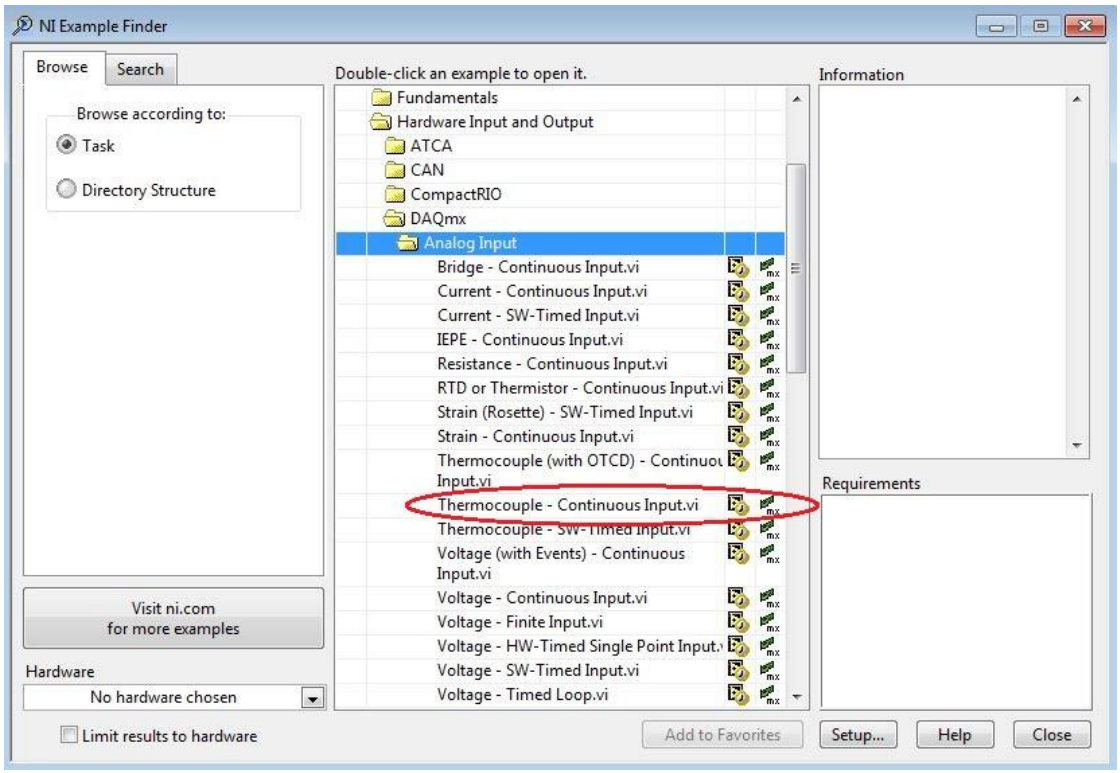
All the parameters that are wired as inputs to the different DAQmx setup VIs reflect the setting you originally configured in the DAQ Assistant Express VI.

Note: By moving these parameter and setup VIs onto the block diagram, you can now programmatically change their values without having to stop your application and open the Express VI configuration dialog, saving development time and possibly optimizing performance by eliminating unnecessary settings depending on your application.

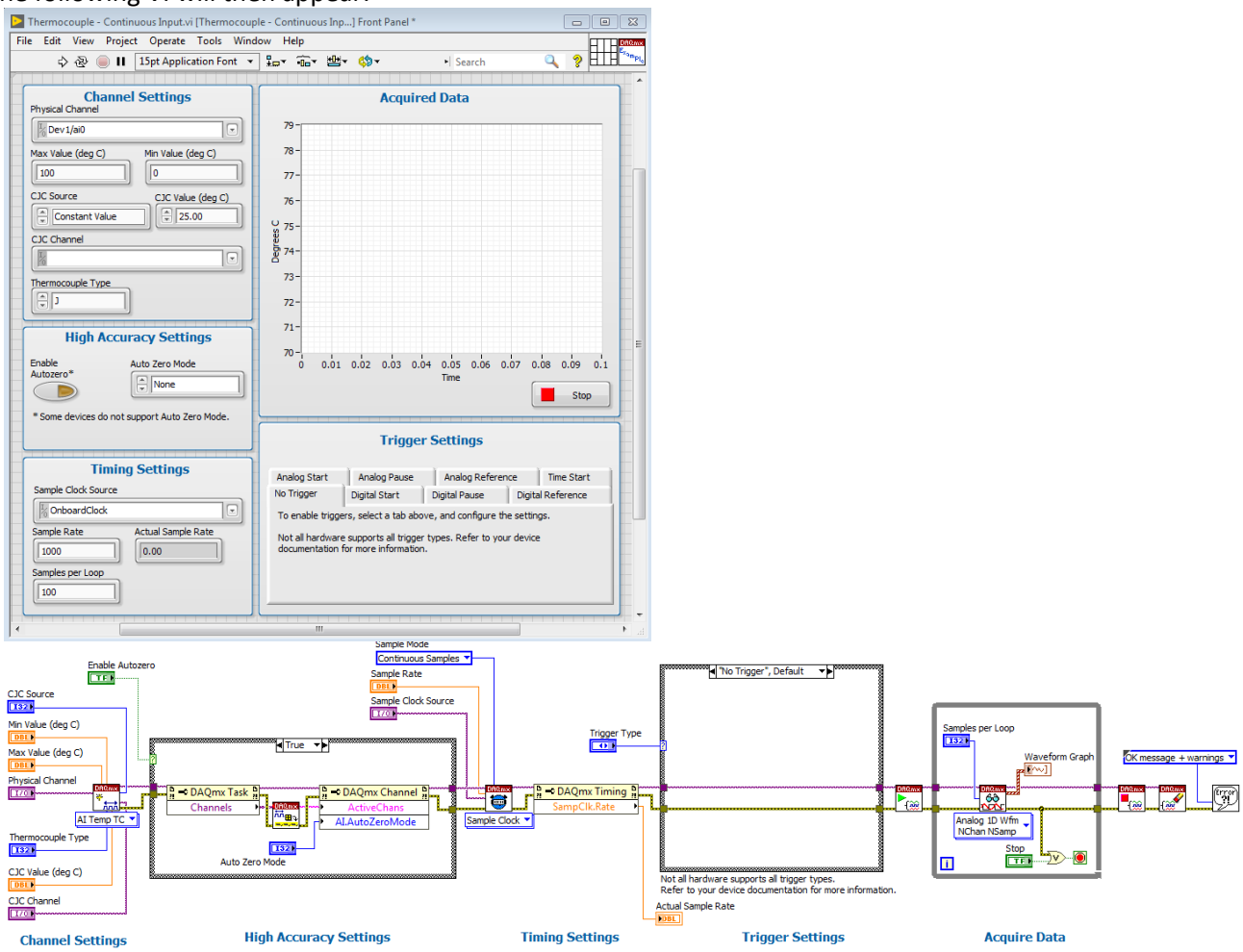
Using the LabVIEW Example Finder

The LabVIEW Example Finder provides hundreds of example application to use as reference or as the starting point for your application.

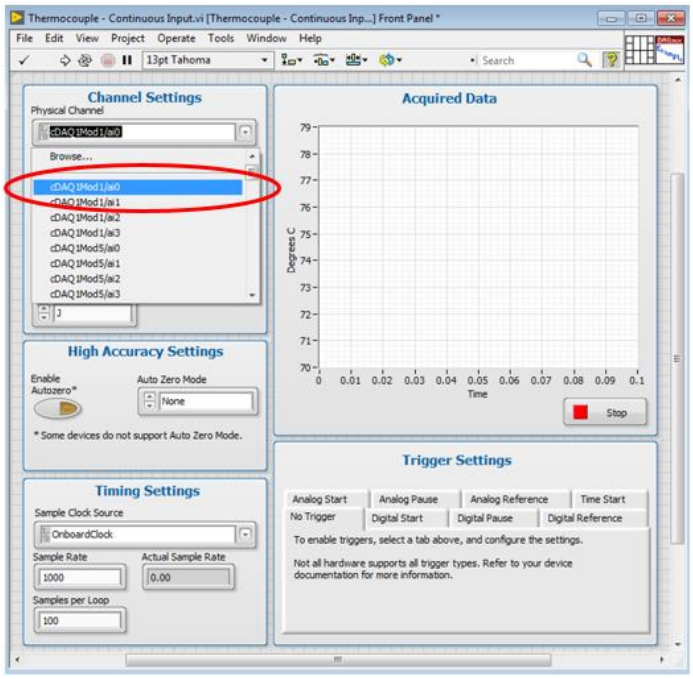
26. Open the LabVIEW Example Finder to find DAQ examples that use DAQmx standard VIs. Go to **Help» Find Examples…** to launch the LabVIEW Example Finder.
27. Browse to the DAQmx Analog Measurements folder from the Browse tab at **Hardware Input and Output» DAQmx» Analog Input** and select *Thermocouple – Continuous Input.vi*.



28. The following VI will then appear:

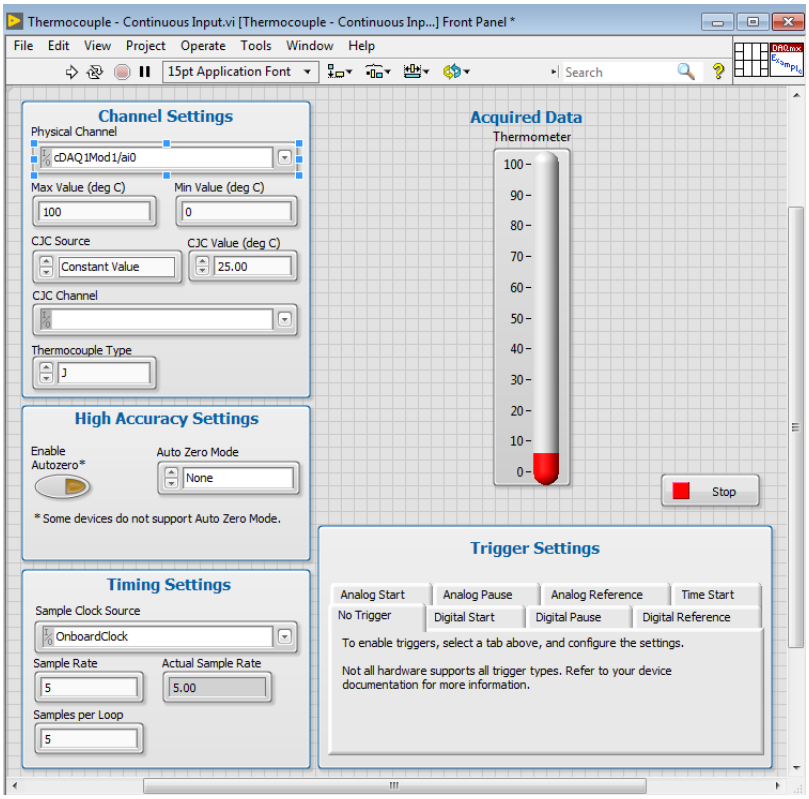


29. Set the Physical Channel to match the CompactDAQ chassis channel and run the application. Expand the physical channel control from the Front Panel and select cDAQ1Mod1/ai0.

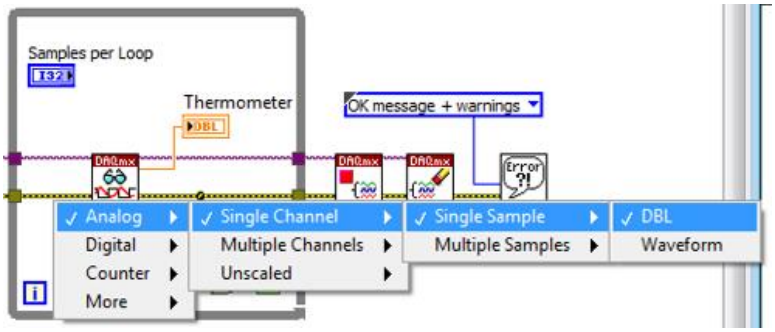


Press the Run button and then hold and release the thermocouple on the CompactDAQ chassis and observe the value change on the front panel.

30. Open the block diagram and examine the code. This VI only uses standard VIs instead of Express VIs, which allows much more customization of inputs and run-time configuration. *Thermocouple – Continuous Input.*vi has a while loop to allow for continuous execution, but the default Timing settings are not useful in this instance.
31. In the *Timing Settings* panel change *Sample Rate* to 5 (samples per second), and *Samples per Loop* to 5, and then run the VI.
32. The Y-scale of the graph is currently in *AutoScale* mode. Right-click the Y-scale and deselect *AutoScale Y*. Set the lower Y-scale value to 20, and the upper value to 35, and then run the VI.
33. The graph can be replaced with a thermometer, but this can only display a scalar (single) value, rather a set of values as does the graph. Delete the graph from the front panel, and add a *Thermometer* from the *Numeric* palette. Expand the thermometer to a suitable size, and change the upper and lower values to 35 and 20.



34. Now edit the block diagram. Move the *Thermometer* terminal into the 'Acquire Data' while loop. Change the instance of *DAQmx Read* (a polymorphic VI) to **Analog» Single Channel» Single Sample» DBL**. The wire connecting *Samples per Loop* to *DAQmx Read* will now be invalid, shown as a dotted line. Click on the wire to highlight it and then delete it. Wire the output of *DAQmx Read* to the Thermometer terminal.



35. Run the VI and change the temperature of the thermocouple.
36. Save the customized example VI to the Project. Go to **File» Save As…**, select **Copy» Substitute Copy for Original** and name the VI "Thermocouple Customized Example.vi." Save this VI. This allows for further development without overwriting the LabVIEW example.

End of Exercise 1

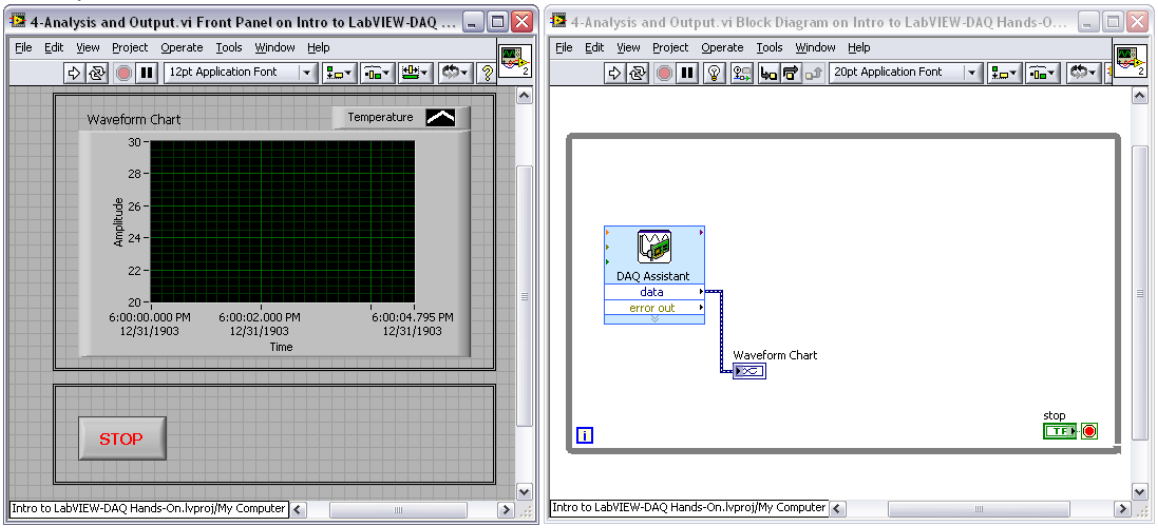
Exercise 2: Add Analysis and Digital Output to the DAQ Application

Set up Hardware

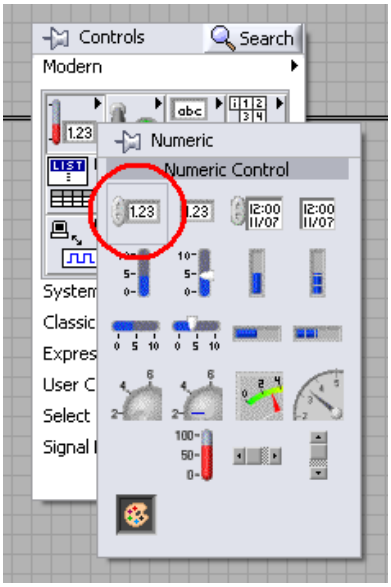
1. Confirm that the CompactDAQ chassis is powered on and connected to the PC via the USB cable. If not, or if it is not behaving as expected, repeat steps #1-8 from Exercise 1.

LabVIEW Application – Compare signal to user-defined alarm

2. Exercise 2 is functionally the same as the end result of Exercise 1. You can open Exercise 1 to synchronize with the illustrations in this section. Open 1-Analysis and Output.vi from the Exercises folder in the Project explorer. The VI will appear like the image below, with additional space on the block diagram to add functionality:

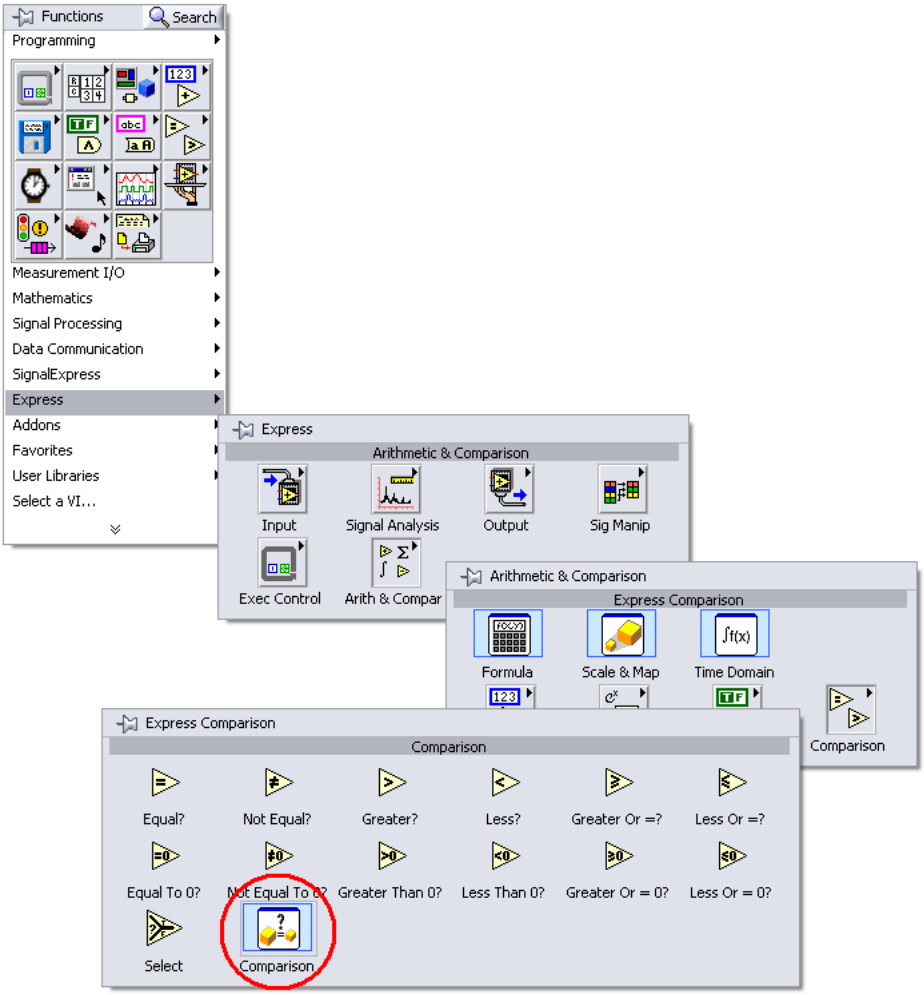


3. Create an alarm that signals if acquired temperature goes above a user-defined level. On the front panel, right-click to open the Controls palette Programming» Numeric and place a numeric control on the front panel.

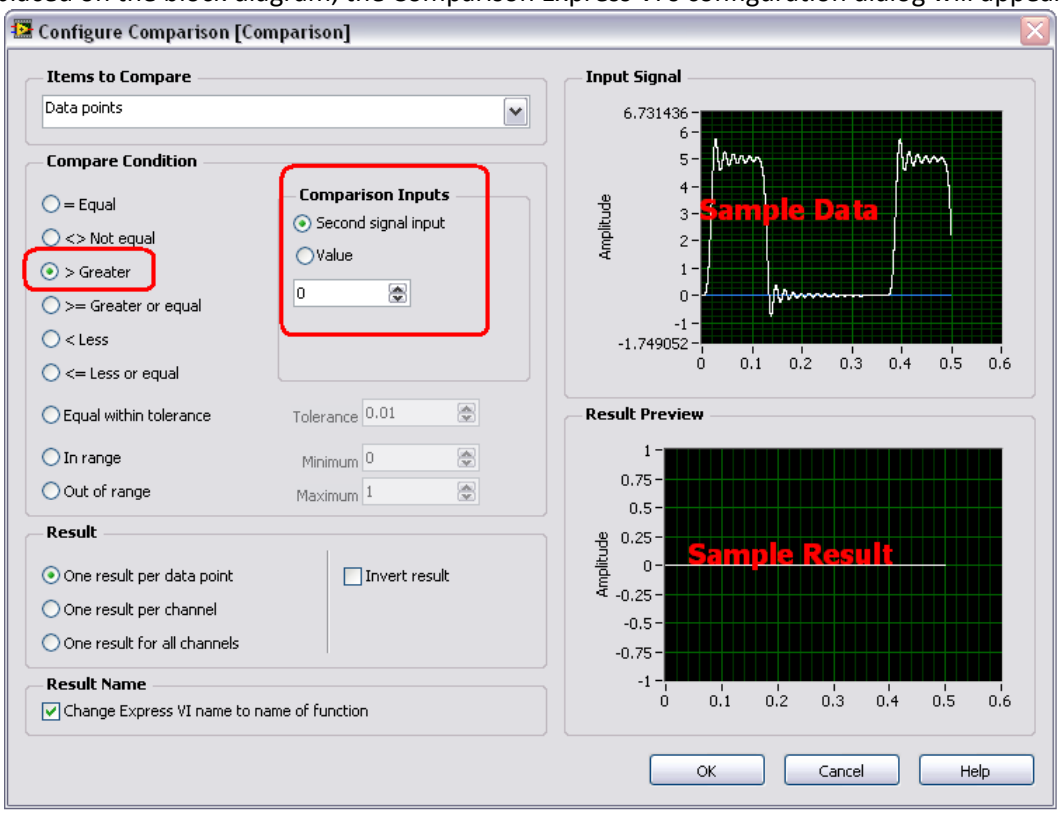


4. Change the numeric control's name to "Alarm Level." Double-click on the control's label and replace the generic text with "Alarm Level"

6. Use the Comparison Express VI to compare the acquired temperature signal with the Alarm Level control. Switch to the block diagram, right-click on an empty space and open the Functions palette. Place the Comparison Express VI on the block diagram from Functions» Express» Arithmetic & Comparison» Comparison.

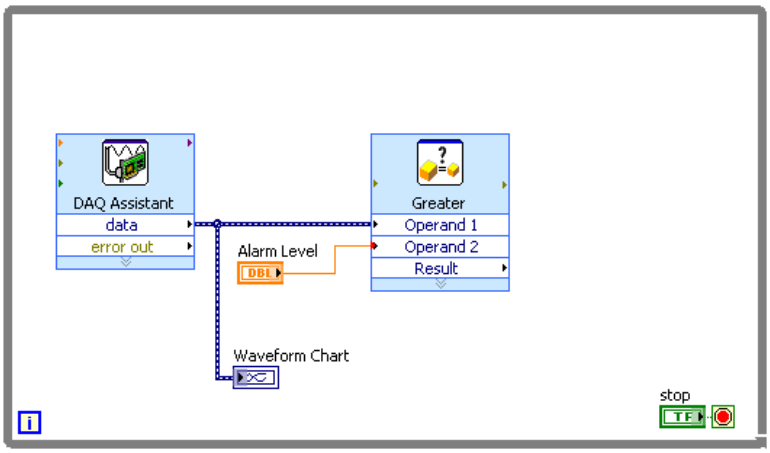


7. Once placed on the block diagram, the Comparison Express VI's configuration dialog will appear.

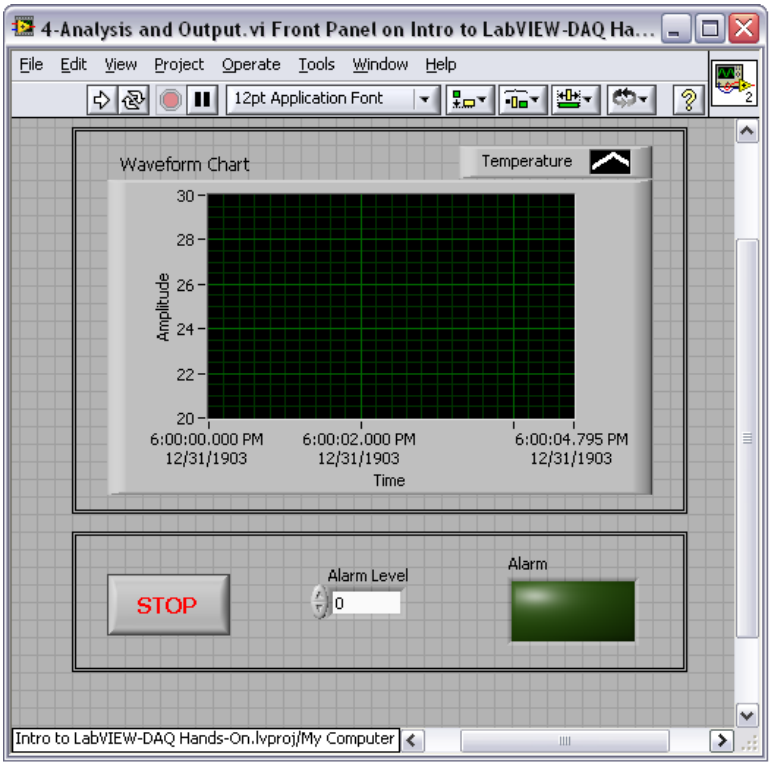


Select "> Greater" in the Compare Condition section and "Second signal input" from the Comparison Inputs section then click OK.

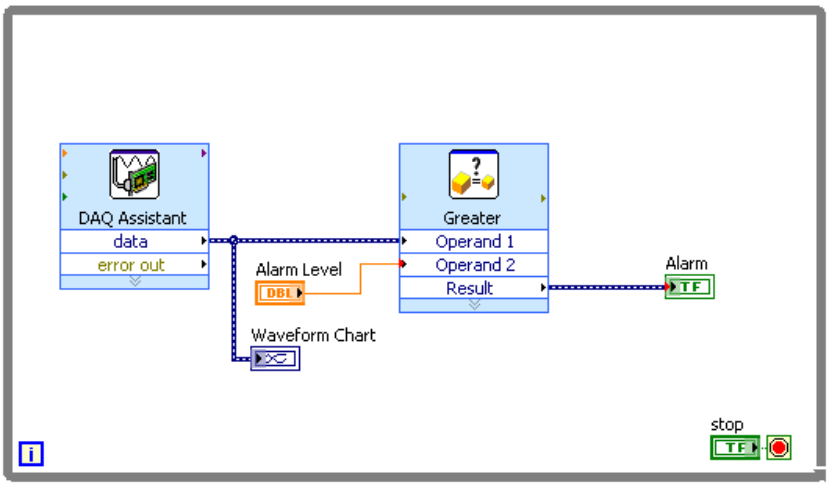
8. Connect the acquired temperature data and Alarm Level inputs to the Comparison Express VI. Hover over the output of the DAQ Assistant until the spool icon appears on your cursor, then left-click and drag you mouse to the Operand 1 input on the Comparison Express VI. Perform the same hover, drag and connect to wire the Alarm Level control and the Operand 2 input on the Comparison Express VI. Your block diagram should now look like this:



9. Display the result of the Comparison Express VI on the front panel. On the front panel, right click, open the Controls palette and add a Square LED indicator. The square LED is found at **Controls» Modern» Boolean**. Resize the Square LED so that it is easier to see and rename it "Alarm." Your front panel should look similar to this:



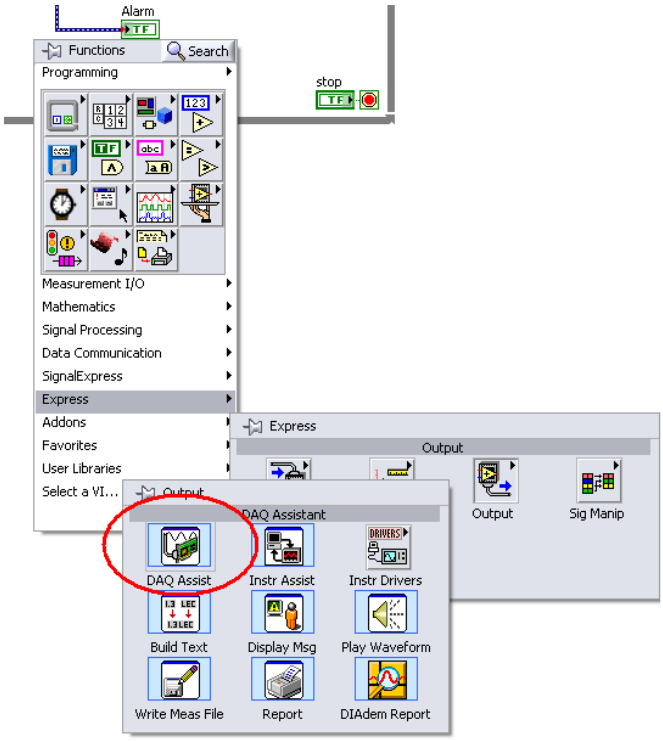
On the block diagram, wire the output of the Comparison Express VI to the input of the Alarm indicator's terminal.



10. Run the application. Press the Run button and then change the Alarm Level control to some level above the current acquired temperature signal. Hold the thermocouple until the temperature exceeds the Alarm Level value. The Alarm LED turns on when the acquired temperature signal goes above the level set on the front panel.

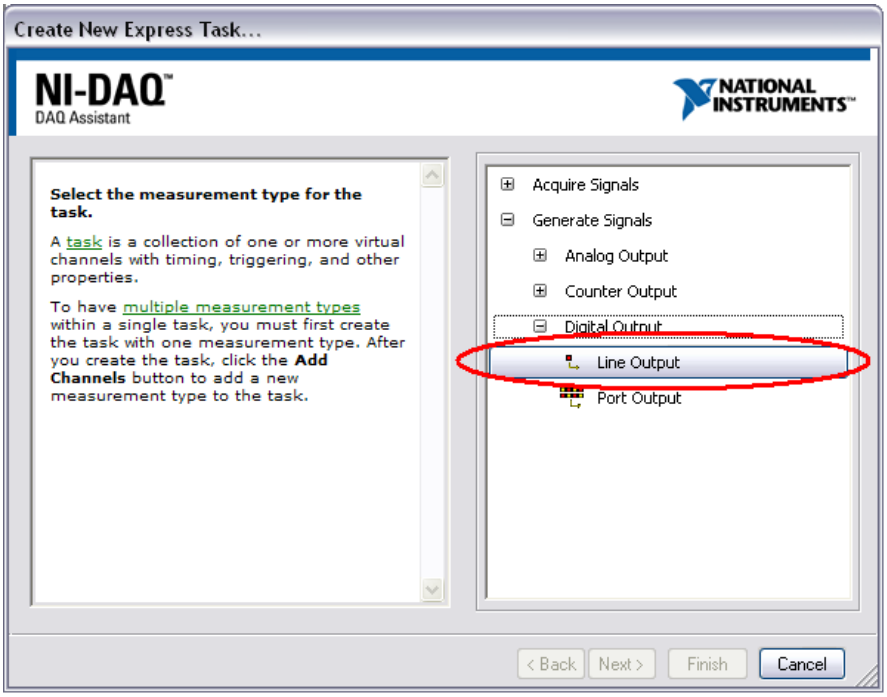
Output Alarm to CompactDAQ Chassis

11. Use another DAQ Assistant Express VI to output Alarm's status to the CompactDAQ's 9474 module. Open the Functions palette on the block diagram and find the DAQ Assistant Express VI at Functions» Express» Output.

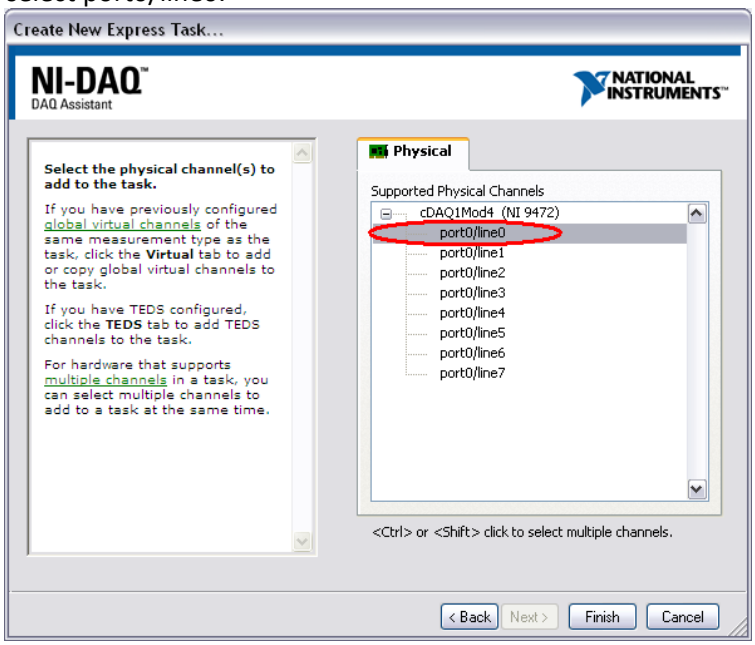


<picture of palette w/ DA circled>

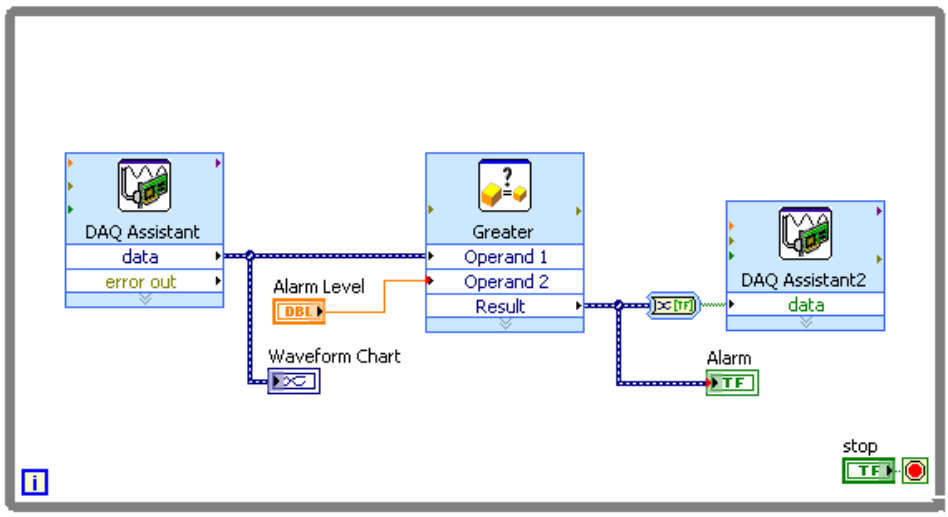
12. Select Generate Signals» Digital Output» Line Output from the Create New Express Task… window.



13. Select the physical channel you want to use as output. Expand the + sign next to cDAQ1Mod4 in the following window and select port0/line0.



14. Press OK in the DAQ Assistant window that appears, since all of its settings are correct for the application.
15. Create an additional wire that connects the Comparison Express VI's Result output to the **data** input on the new DAQ Assistant Express VI. A Convert from Dynamic Data function appears automatically. LabVIEW will always try to coerce unlike data types when two nodes are wired together. In this case, the output of the Compare Express VI is a Dynamic Data type, and the input of the DAQ Assistant is Boolean. LabVIEW placed the Convert from Dynamic Data node in between the two nodes so they could be connected. You can double-click the Convert from Dynamic Data to view its configuration. Your block diagram should now look like this:

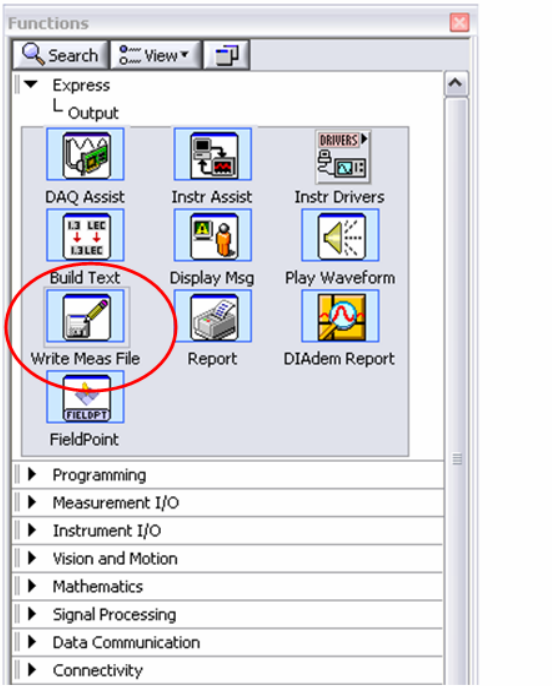


16. Run the VI. Press the Run button. Notice that the LED bank on the CompactDAQ 9474 module turns on and off to match Alarm's value on the front panel.
17. Save and close the VI.

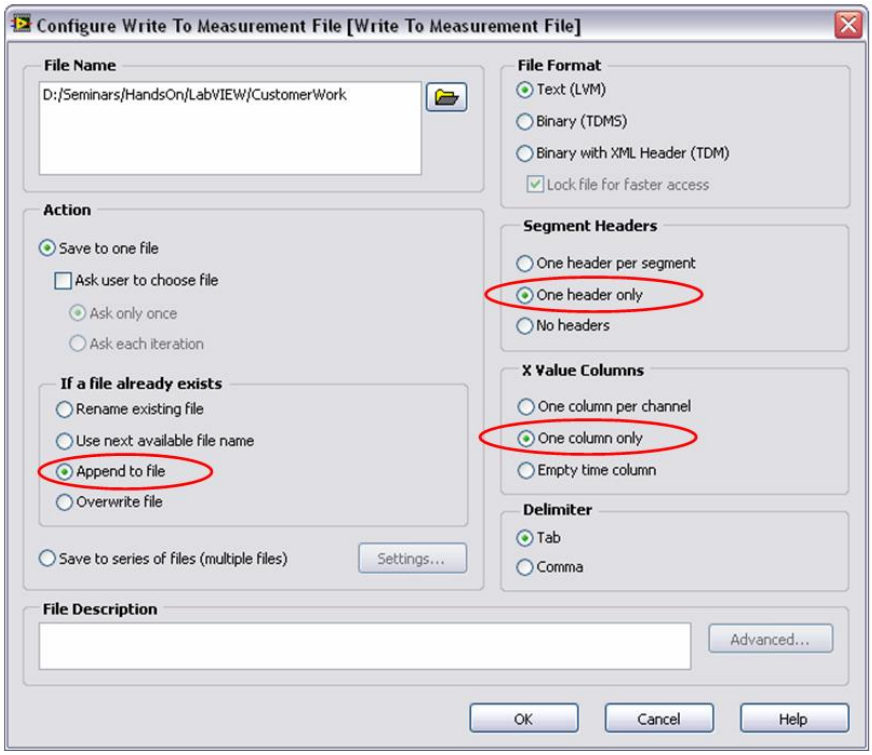
End of Exercise 2

Exercise 3: Writing Data to File with LabVIEW

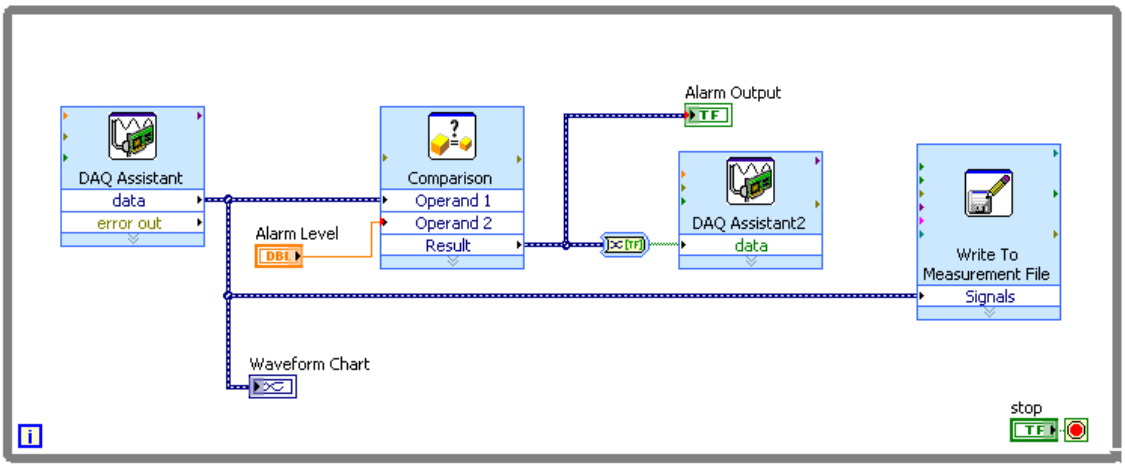
1. In the Exercise folder in the Project Explorer, open 2-Analysis and Output.vi. We will use the final program from the last exercise as the beginning of this exercise.
2. Right-click on the block diagram and select **Functions» Express» Output» Write to Measurement File** and place it inside the While Loop on the block diagram.



3. A configuration window will appear. Configure the window as shown below and click OK.



4. Wire the output of the DAQ Assistant Express VI to the input of the Write to Measurement File Express VI.
5. Your block diagram should now resemble the following figure.



6. Save the VI by using the **File» Save As…** menu, select the **Copy» Open Additional Copy** and name it 3-Write to File.vi.
7. Run the VI momentarily and press STOP to stop the VI.
8. Your file will be created in the folder specified.
9. Open the file using Microsoft Office Excel or Notepad. Review the header and temperature data saved in the file.
10. Close the data file and the LabVIEW VI.

End of Exercise

Exercise 4: Generate, Acquire, Analyze and Display

This is a challenge exercise and step-by-step instructions are not provided, but rather the end goal is given. It is up to you to figure out how to come up with the program to achieve the given task.

Different VIs can be made for the steps below, but the VIs can be run concurrently.

1. Generate a sine waveform using the analog output module, and check the output signal with an oscilloscope (if available).
2. Acquire the sine waveform using the analog input module, and display on a chart.
3. Perform analysis of the waveform to calculate the frequency of the acquired sine-wave. Use the Buneman Frequency Estimator VI, with appropriate scaling, to measure and display the frequency.

8. *Lab 8: ADC basics for TDAQ*

Manoel Barros Marin (manoel.barros.marin@cern.ch, tutor)

Table of Contents

Concepts of this lab

Figure 1 below shows the generic signal flow of a Data AcQusition (DAQ) system to perform Analog to Digital Conversions (ADC).

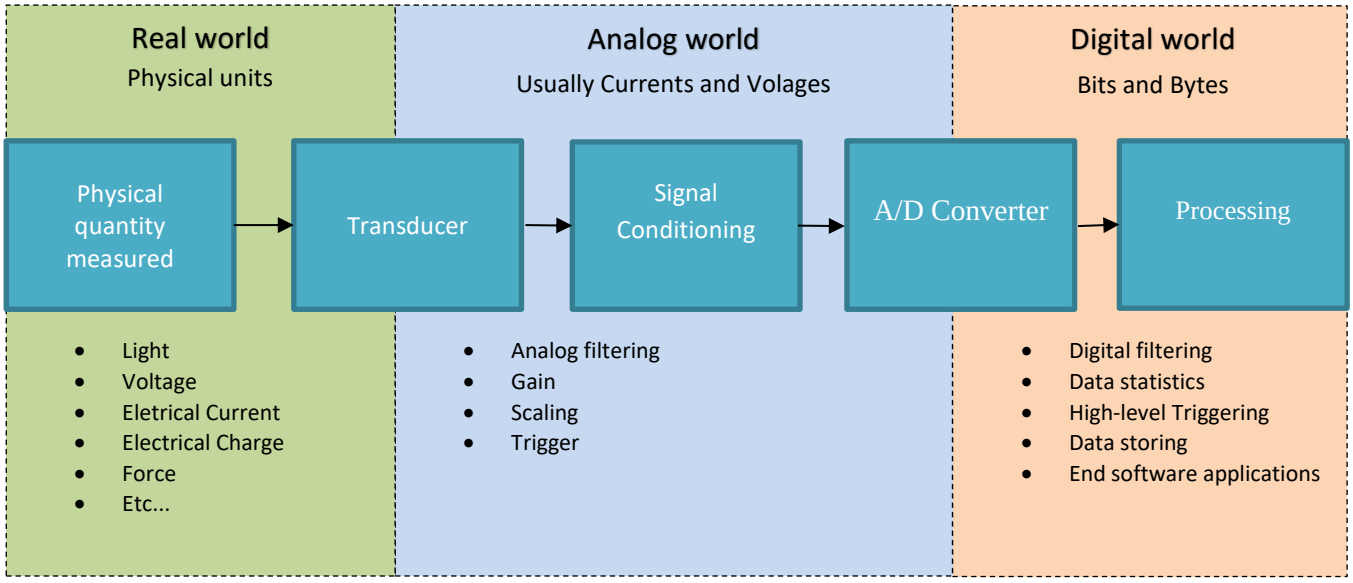


Figure 1- Basic ADC DAQ chain

The scope of this laboratory experience is to understand and experiment with the following components of a Data AcQuisition system:

1. Triggers:

   1.1. External triggers: accelerator like type of triggers

   1.2. Triggering on the signal: astroparticle like type of triggers

2. Analog to Digital Converter (ADC): we will try to understand fundamental parameters that come into play when measuring with ADC; as for example its resolution, speed, bandwidth, acquisition window, etc.

Lab setup

Figure 2 below depicts the setup for Lab8.

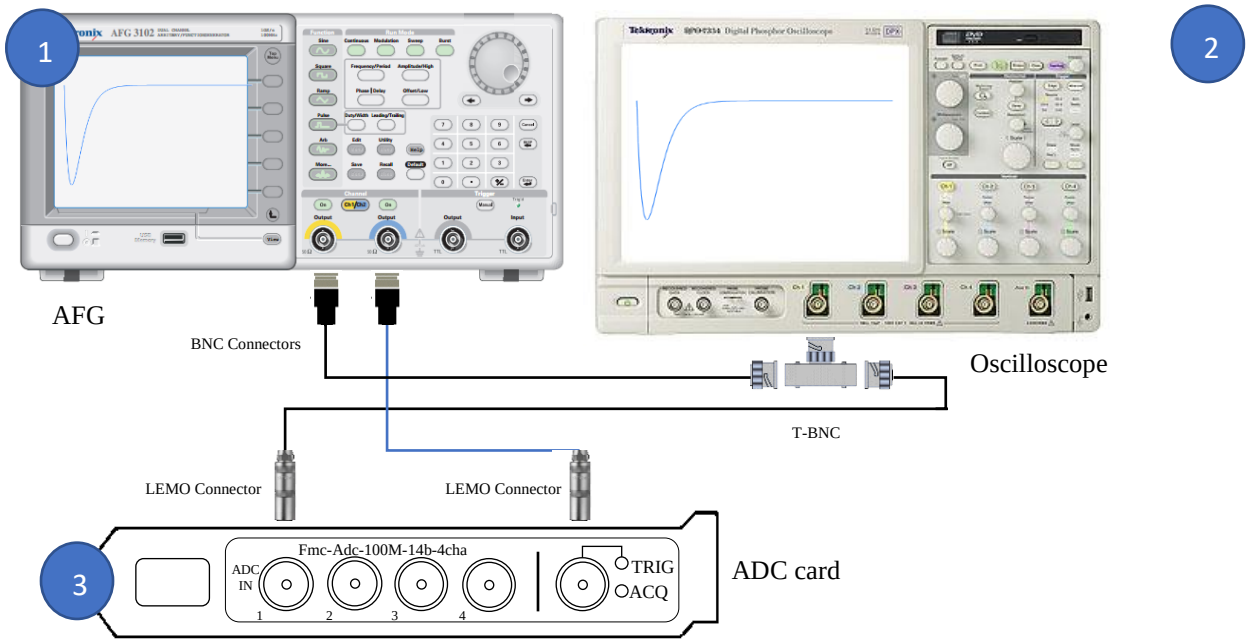


Figure 2 - Equipment setup

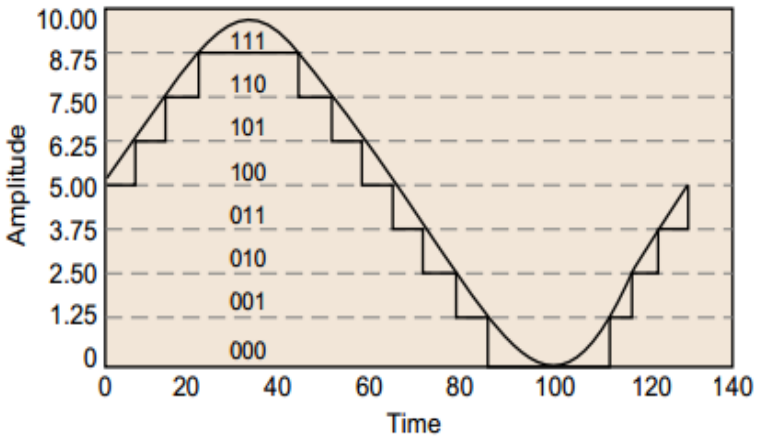
1. Tektronix AFG3252: Arbitrary Function Generator, is the **input** of our system
2. Tektronix DPO70404C: Oscilloscope, it the **monitor** of our system (to cross-check that the signals fed to the system are truly what intended)
3. SPEC+FMC ADC card plus host PC: this is the **core ADC DAQ**
   3.1. FPGA Mezzanine Card (FMC) ADC: http://www.ohwr.org/projects/fmc-adc-100m14b4cha
   3.2. Simple PCI Express Carrier (SPEC) card: http://www.ohwr.org/projects/spec/wiki
   3.3. Linux based host PC

Introduction to Analog to Digital Conversion (ADC)

An Analog to Digital Converter (also known as “digitizer”) is a device which converts the level of an analogue signal into an integer number which is closest to the real value of the signal in terms of ratio. There is a limit for the number of choices of this integer which is determined by the number of bits used for the digitization.

Example: We have a voltage signal whose range is [0 - 10] Volt, and we have a digitizer which has just 3 bits. We can have 8 different integer numbers with 3 bits. Each of these numbers will correspond to a voltage. See table below.

Voltage	Digitizer bits	Integer
0.00 - 1.25	000	0
1.25 – 2.50	001	1
2.50 - 3.75	010	2
3.75 – 5.00	011	3
5.00 - 6.25	100	4
6.25 – 7.50	101	5
7.50 – 8.75	110	6
8.75 – 10.00	111	7



This kind of conversion is performed at regular intervals (**samples**), and the repetition (frequency, Hz) is called **sampling frequency** (normally expressed in samples per seconds). Thus, digitization is not only performed in the signal domain, but also in the time domain.

Also note that to improve the precision of the measurement one should:

- Increase the number of bits, so each corresponding range is small with respect to the signal to be sampled
- Sampling should be done at the proper frequency (see Nyquist frequency); and even more importantly, a precise sampling clock should be used, such that the deviation between consecutive samples is kept as constant as possible.

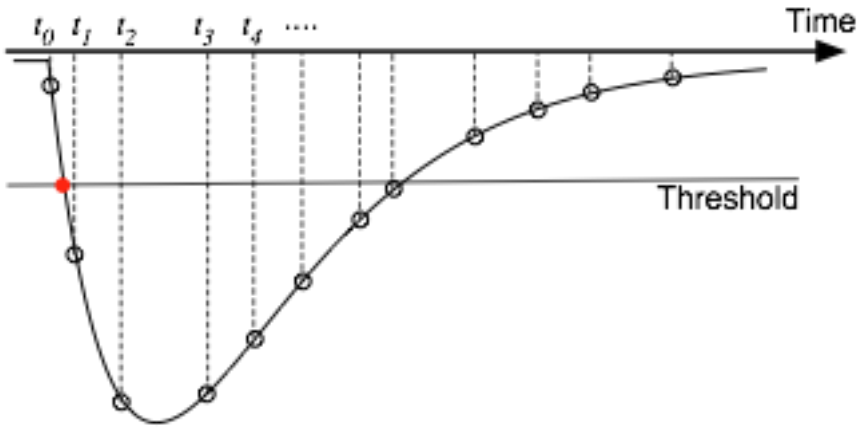
If you are interested in more details about ADC parameters, please check: http://www.analog.com/en/analog-to-digital-converters/products/index.html

If are interested in understanding more in detail the importance of clock stability (jitter), please check: http://anlage.umd.edu/Microwave%20Measurements%20for%20Personal%20Web%20Site/Tek%20Intro%20to%20Jitter%2061W_18897_1.pdf

The ADC card we will be using has 14 bit voltage resolution and a typical conversion time of 10 ns (100 MS/s). In this exercise, we will operate the ADC in order to understand and push its limits.

The Measurement

Qualify in the best possible way a set of signals mimicking real experimental equipment. The tutors will provide some pre-cooked ones (sine waves, positive pulses, scintillator like pulses) but you can try to think of your real world input.



In general terms, a digitization process is characterized by the following:

- Signal range: how much the signal can vary to be correctly interpreted by the device (voltage)
- Sampling frequency: the rate at which it can convert an analogue value to a discrete signal (bits)
- Latency: How long does the device takes to finalize the acquisition process in the chain (time)
- Noises: physical quantities responsible for the signal deterioration (e.g. added noise, intrinsic noise, quantization errors, etc.)

The best measurement is achieved by understanding and controlling those parameters. **The designers (so YOU) have to decide how to setup your TDAQ:** choose a trigger type, define the acquisition windows, find the signal in the window, maximize the scaling for increasing the accuracy of the measurement, etc.

Architecture of a Linux Device Driver for the PCIe Card

Before running the DAQ system, an **overview on the communication between the software and the hardware layers and an introduction to the role of device drivers**, is fundamental to understand. *Figure 3* shows a simplified diagram of the Layers of a Linux Operating System (OS).

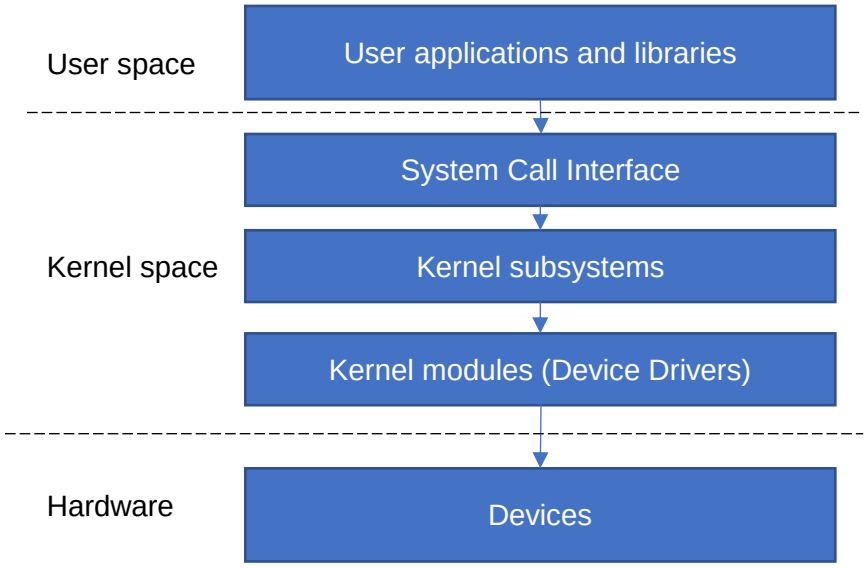


Figure 3- Layers of a Linux OS

The basic control is on the hardware peripheral itself. The lowest level software for this system resides in the kernel of the OS as a “device driver.” There are certain control/status registers on the ADC card. These registers can be accessed just like a regular memory access in a C program.

As seen earlier in this document, in our context, the ADC DAQ hardware used is composed of two electronic boards: the SPEC carrier board and the FMC ADC module. The first board is attached directly to a PCIe slot connector on the PC, and is the host for the ADC module. The SPEC card acts as a bridge for the electrical signals of the ADC FMC to be interfaced and converted to PCIe. For the scope of this lab, the “bridging” is done by some “black box” electronics. Thus from a software perspective, one kernel module is instantiated to use the SPEC card, another one for the FMC ADC module.

For simple sensors and devices vendors provide information about their operation and use; which can be seen as a map of internal registers and/or procedures required to perform operations or change their current states (more about in Exercise 12 "DAQ Online Software”). For the sake of simplicity and reusability of software code, software engineers created the concept of frameworks. Among others, this concept was used at CERN for creating a flexible interface for the development of input and output drivers. The target is for very-high-bandwidth devices, with high-definition time stamps, and a uniform metadata representation. Such framework is named the ZIO (with Z standing for “The Ultimate I/O” Framework).

In short, the way ZIO provides to talk to our hardware is through two different channels, one for control and one for data. *Figure 4* shows the two data flows.

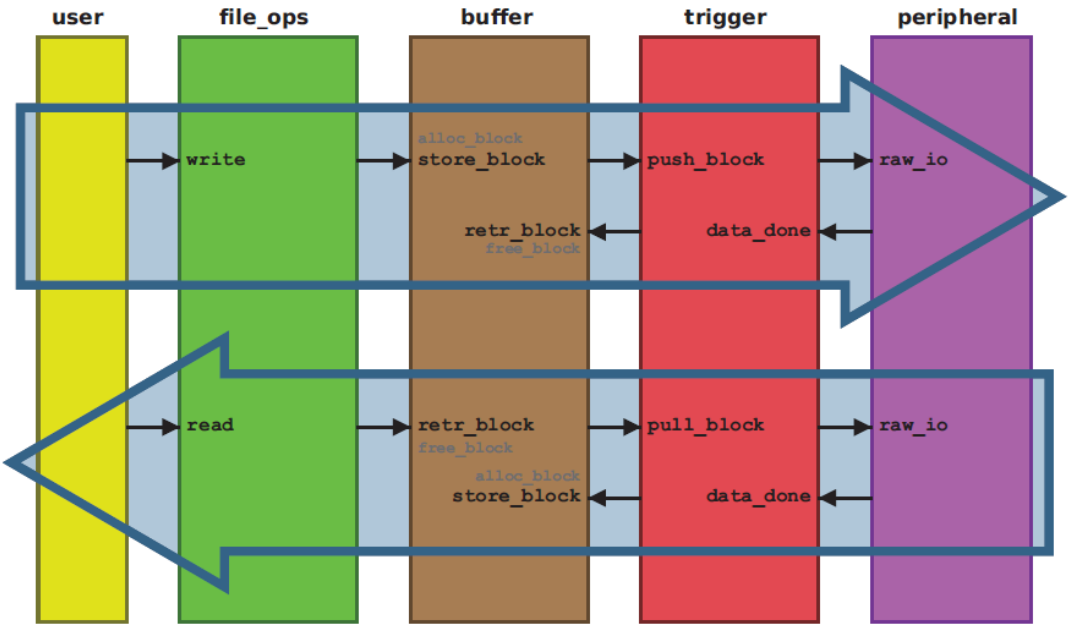


Figure 4 - ZIO data pipeline

When designing a piece of low-level software to communicate to an acquisition device, it is important to choose the way data streams should be passed back and forth the hardware device. To illustrate this they can be qualified in three types: **Polling mode, Interrupt based, and DMA (Direct Memory Access) transfers**.

- **Polling mode:** the CPU checks the state of the device's registers every time it can, this has the advantage of providing a fast and low latency communication and data transfer between them, but at the cost of high CPU load.
- **Interrupt based**: this type of transfers eases the CPU load time as it only have to care about the hardware when it tells the CPU to do so; it has the advantage of a lower impact on CPU loads and fast transfer, but with variable latency.
- **DMA transfers**: relies on a shared memory area the CPU provides to the DMA controller to work on the transfer with the hardware device, this frees completely the CPU from controlling the hardware transfer representing a true concurrent system. Highest of all transfer rates allied to an acceptable latency.

Operating the setup

1) Construct/confirm the experimental setup according to the sketch you find at the beginning of this document. **Check that both outputs of the AFG are switched OFF.**

   The signal generated by the Channel-1 (**source input**) of the AFG unit is supplied to the oscilloscope, and to the Channel-1 input of the ADC card. Use a T-BNC to split the signal at the AFG output. Also check if the Channel-2 output (**trigger input**) of the AFG is connected to the Trigger input of the ADC card. Again you can monitor the trigger on the scope by spitting with a T-BNC on the AFG output.

2) Using the AFG, set Channel-1 to generate a sine waveform with a frequency of 1 MHz and amplitude of 1 Vpp. Completed this operation, set Channel-2 to generate a pulse waveform with a frequency of 1 KHz and amplitude of 4 V. **Don't turn them ON yet, and check if they have not an amplitude above 5 Vpp**.

3) Login to the PC and reset the lab8 directories, so all the work/changes done by the previous group are removed and a fresh copy of the files are installed

4) In order to load the drivers, open the `~/adc` directory and execute the script for loading the drivers:

```
$ cd ~/adc
$ ./load_drivers.sh
```

   You need the root password in order to execute the command above. Ask your tutor for it.
   Spying the content of our script shows the correct order for loading the different kernel modules and set the user permission to talk with the ADC board. It is also possible to check the current kernel modules loaded by issuing the command `lsmod`.

5) Now it is time to test our ADC, turn ON only Channel-1 of the AFG and check if the signal is correctly displayed by the oscilloscope. Run the acquisition program which will subsequently show an acquisition plot:

```
$ cd Trigger_ext
$ ./fald-acq -a 1000 -b 0 -n 1 -l 1 -g 1 -r 10 -e -X 0100
```

   Where acq-program has the following parameters:

```
--before|-b <num> number of pre samples
 --after|-a <num> n. of post samples (default: 16)
--nshots|-n <num> number of trigger shots
 --delay|-d <num> delay sample after trigger
--under-sample|-u|-D <num> pick 1 sample every <num>
--external|-e use external trigger
--threshold|-t <num> internal trigger threshold
--channel|-c <num> channel used as trigger (1..4)
--range|-r <num> channel input range: 100(100mv) 1(1v) 10(10v)
--negative-edge internal trigger is falling edge
 --loop|-l <num> number of loop before exiting
--graph|-g <chnum> plot the desired channel
--X11|-X Gnuplot will use X connection
```

Did the program made the acquisition?

6) Turn ON Channel-2 of the AFG. Does it run now?

7) Now that we know that the ADC is working properly we can go to some real time data analysis. Bearing this in mind, there is a small program called `V_t_continous.C` which uses the ROOT framework functionalities and runs on top of `fald-acq`. To run `V_t_continous.C` issue the command:

```
$ root V_t_continous.C
```

8) Try now to modify Channel-1 frequency in steps of Hz while checking if the acquired waveform remains true to it. Check also if the measured amplitude is the same as set on the AFG or, say, twice its value. **Make sure you are changing the frequency NOT the amplitude.**

9) One issue you may get while changing the frequency is the 'non-stopping' graph, meaning it is continuously sweeping horizontally. What causes this?

   Since we are using Channel-2 of the AFG as an external trigger, its triggering frequency dictates how, or at which point in time, the acquisition of Channel-1 signal starts. In practice: if the frequency of our sine wave is not a harmonic of Channel-2 pulse, i.e. not an integer multiple, the ADC doesn't capture the signal at the same phase.

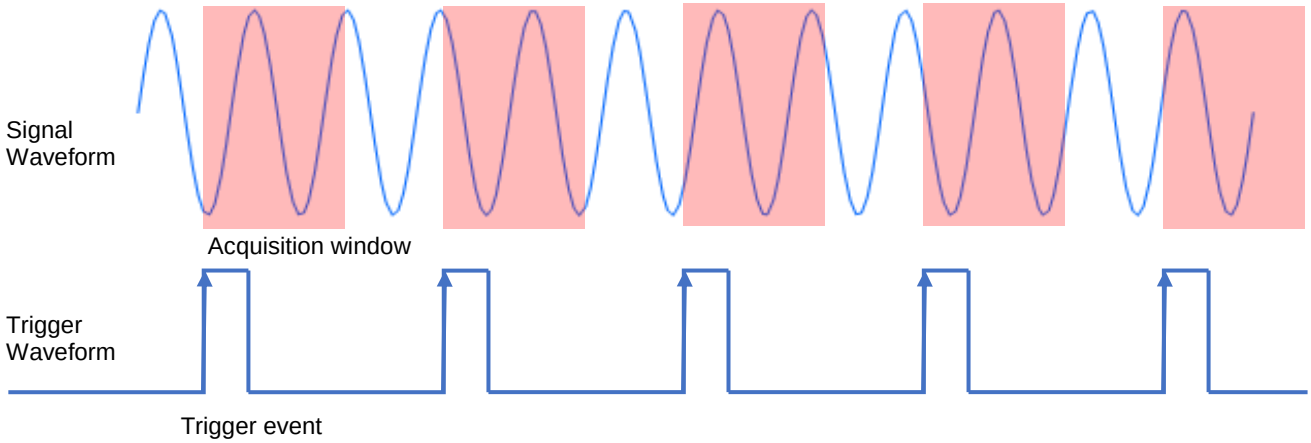


Figure 5- Signal frequency non multiple of Trigger frequency

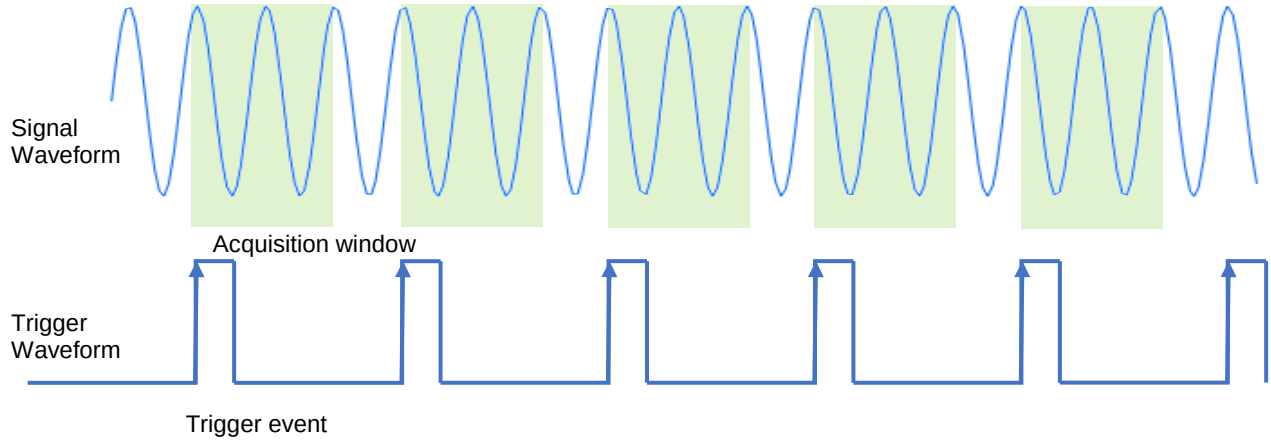


Figure 6 - Signal frequency multiple of Trigger frequency

10) Going further, let's push the frequency of our signal to the limits of the ADC capabilities. Recalling that our ADC specifications say that the default sampling rate is 100 Million Samples/s try to increase the signal frequency in steps of MHz. *Please note: it is recommended to decrease the number of samples in our acquisition window*; otherwise it would become hard to analyze the signal in a cloud of points.

    For this, open the `V_t_continous.C` file with your favourite editor and change the number of post samples (-a parameter) to 100 or less, it is located on the line which calls `fald-acq`.

    Run `root V_t_continous.C` again.

    Can you see the relation between the acquisition window and the signal speed?

11) You can see that the closer you get to 100 MHz the worst the acquisition signal looks like. Can you define the maximum AFG signal frequency where our sine wave keeps its shape in a single acquisition shot (i.e. you can still see a sine wave with the same frequency as the original signal)?

    This value is called the **Nyquist frequency**. The **Nyquist theorem** states that: *the minimum sampling rate of an acquisition device must be at least two times the maximum frequency of the original signal, otherwise information would be lost in the process*. See *Figure 7* below: the original signal is in blue, the sampling points are pointed by the black arrows and the acquired data is represented in orange. When this criteria is not respected an effect called aliasing appears.

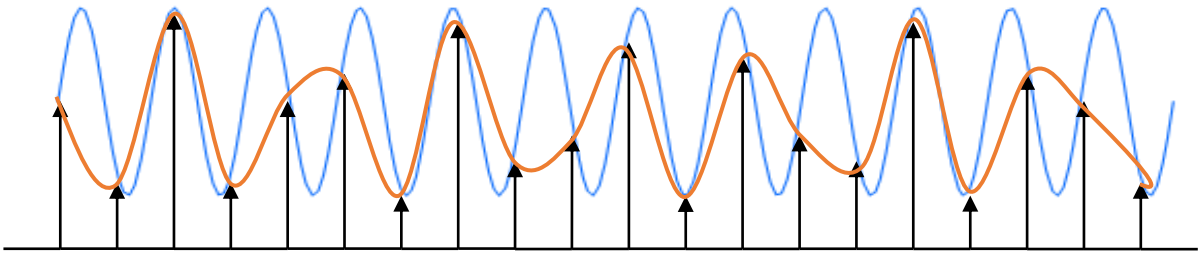


Figure 7 - Aliasing effect

12) The maximum Nyquist frequency is computed on a signal ideally composed of a single sinusoid. In fact any real function of a complex signal is mathematically described as the sum of a series of trigonometric functions in the **frequency domain** (called *signal spectrum*) instead of time. **Fourier series and transforms** base their analyses on this concept. The spectral representation of a sine wave is rather simple: it is only a single line centred around the frequency it coverts from the time domain. The spectral content is more complex for different waveforms signals such as square, saw-tooth and other more complex signals.

    In order to check how our acquisition systems behaves with complex signals go back to the frequency of 1 MHz but change the type from sine-wave to square signal. Increase the frequency to values below Nyquist criteria.

13) Ask the tutor to present a real time Fast Fourier Transform (FFT) and **ask further questions!**

Acknowledges

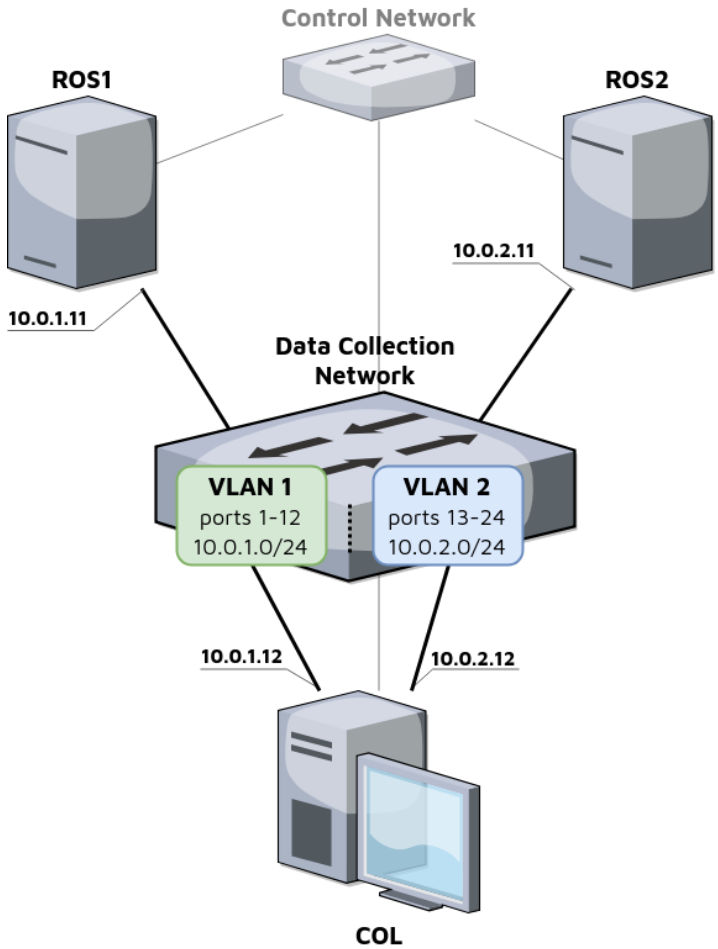
Andrea Borga (**andrea.borga@nikhef.nl**, initiator and tutor of this lab in ISOTDAQ2016)
Cairo Caplan (**cairo.caplan@cern.ch**, initiatior and lab assistance in ISOTDAQ2016)
Diogenes Gimenez (**diogenes.gimenez@usp.br**, initiator and lab assistance in ISOTDAQ2016)

9. *Lab 9: Networking for Data Acquisition Systems*

I. Introduction

Through the use of a simplified network setup this lab aims at getting you familiar with the following notions involved in networking:

- configuration: MAC and IP addresses, switch management, VLAN, routing
- monitoring: SNMP, RRD, traffic analysis
- testing: peformance benchmarking, TCP/IP
- optimization: QoS, DSCP



The lab setup consists of:

- 2 headless desktop computers mimicking readout system (ROS) servers that read data out of the detector, process them, and transfer them: they will be the data sources.
- 1 desktop computer mimicking a collector (COL) server receiving and formatting data from different ROSs: it will be the data destination.
- 1 enterprise-grade network switch mimicking a data collection network.

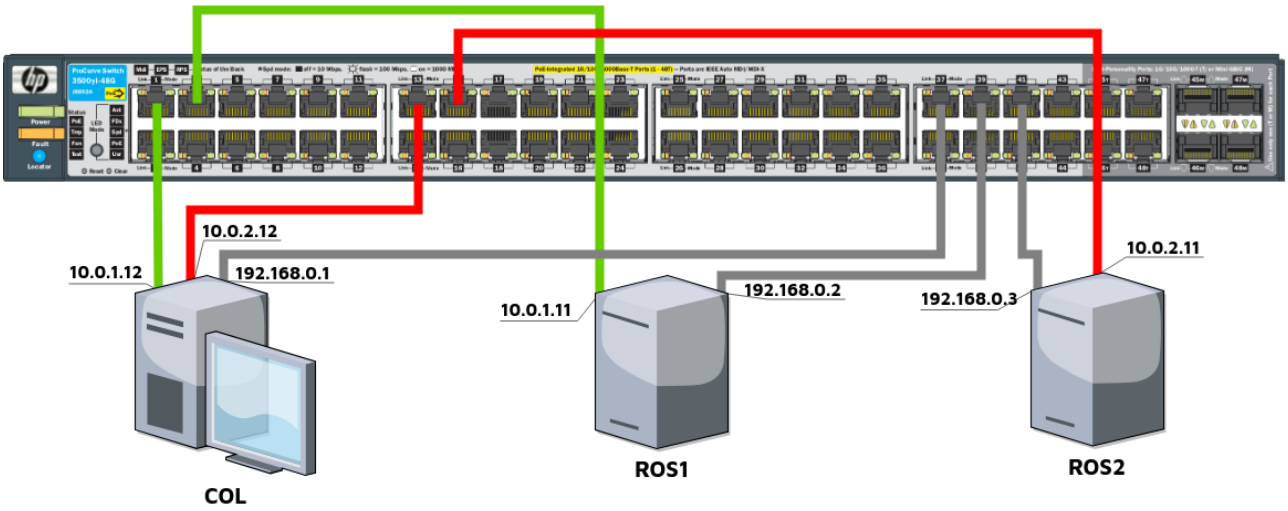
- 1 control network used to connect to the different computers, collect monitoring data and allowing us to configure and use the data collection network without interfering with it.

In DAQ systems, it is a common pattern to isolate the control network used for configuration and monitoring from the data collection network used to transfer data from the detector. These two types of network support different types of communication: low latency is usually expected from a control network whereas the data collection network has to provide high bandwidth and efficiency. Nevertheless, more and more, recent technologies allow the different network types to use the same physical devices and to be isolated from each other at a logical level only.

II. Exercises

1. Switch Configuration

1. The switch has just been powered on and has default factory configuration. Network cabling is as follows:



2. By default the switch has no IP address and there two ways to configure it: setup a DHCP server that will provide it with an IP address automatically, or connect via a serial cable. We use the latter.

Use "screen" to connect to the serial console of the switch. Hit [enter] two times to activate the connection: you are in the "manager" mode; these terminals usually have 4 modes: operator (read access), manager, global configuration, context configuration. The switch terminal support auto-completion. Display and analyze its default configuration:

```
[student@col]$ sudo screen /dev/ttyS0
HP-3500yl-48G# show running-config
```

3. Configure an IP address on the switch via the "configure" shell.

```
HP-3500yl-48G# config
HP-3500yl-48G(config)# vlan 1
HP-3500yl-48G(vlan-1)# ip address 192.168.0.10/24
```

4. Check the configuration.
5. Now the switch can be managed remotely via telnet or SSH. Try to ping and then telnet the switch from the COL computer. The switch also runs a web server and you can try to connect to its address with a web browser.

```
[student@col] $ ping 192.168.0.10
[student@col] $ telnet 192.168.0.10
```

6. Configure two additional VLANs on the switch to match the setup described above. The **untagged** commands assign the given port range to the current VLAN; "untagged" refers to an optional Ethernet header field which can be used to discriminate different types of traffic coming through the same port. In our case,

packets are not tagged.

```
HP-3500yl-48G(config)# config
HP-3500yl-48G(config)# vlan 2
HP-3500yl-48G(vlan-2)# untagged 1-12
HP-3500yl-48G(config)# exit
HP-3500yl-48G(config)# vlan 3
HP-3500yl-48G(vlan-2)# untagged 13-24
```

7. Using the "ping" command, check that COL and ROS1 can communicate together *via the data network*. Do the same for COL and ROS2.
8. Move the ROS2 cable to a port in vlan 2. Test the communication between COL and ROS2 again. Can you explain why it does not work anymore? At the end, move the ROS2 cable back to the original port.

2. Network Monitoring, Routing and Traffic Analysis

Before we further test connectivity in our network, let's setup a simple monitoring system to have a real-time view on traffic. We can simply do it with the SNMP features provided by the switch.

1. Request some simple information from the switch via SNMP using "snmpget": switch description and input traffic for port 1.

```
[student@col] $ snmpget -v 2c -c public 192.168.0.10 sysDescr.0
[student@col] $ snmpget -v 2c -c public 192.168.0.10 ifInOctets.1
```

2. Note that SNMP uses two representations for object identifier (OID): numerical and human-readable. You can use "snmptranslate" to translate one into the other:

```
[student@col ]$ snmptranslate -On IF-MIB::ifInOctets.1
.1.3.6.1.2.1.2.2.1.10.1

[student@col ~]$ snmptranslate .1.3.6.1.2.1.2.2.1.16.3
IF-MIB::ifOutOctets.3
```

3. The directory "swmon" contains basic scripts that uses SNMP and RRD tools to fetch monitoring information. "monitor.sh" periodically requests traffic information for the 4 connected ports, writes them to the database, and generates the associated plots
4. In a *dedicated* terminal run "monitor.sh" script. Then open "index.html" in a web browser. Have a look at the plots to understand them.

```
[student@col swmon]$ ./monitor.sh &
(... periodically prints out what it's doing...)
[student@col swmon]$ firefox index.html &
```

5. Generate traffic from COL using "ping" and match the generated traffic with the plots. Notice that "input" and "output" on the plots refer to the point of view of the switch. You can use the "-s" option to increase the size of the ping packets and have more readable plots.

```
[student@col swmon]$ ping -s 1000 10.0.1.11
[student@col swmon]$ ping -s 1000 10.0.2.11
```

6. Now try to ping 10.0.2.11 (ROS2) *from ROS1*. Notice that the symptom is different from the situation at the end of exercise 1. Try to explain why it does not work interpreting the error message from ping. Can you imagine a solution to make it work? Let's implement it.

   Print routing table of ROS1:

```
[student@ros1]$ ip route
```

   No route matches the destination network. Add an explicit route to reach ROS2 subnet using COL as a gateway:

[student@ros1]$ **sudo ip route add 10.0.2.0/24 via 10.0.1.12**

Try to ping again. Why do you think it still does not work? At this stage you would like to observe the network traffic to understand the situation. We can do this with a network analyzer (run on COL):

[student@col]$ **sudo wireshark**

Start a capture on interfaces with IP 10.0.1.12 and 10.0.2.12 (you can find their names with "ip addr"). Try to ping again and look at the packets captured by Wireshark. Do you understand the situation now?

Configure COL to act as a gateway:

[student@col]$ **sudo sh -c "echo 1 > /proc/sys/net/ipv4/ip_forward"**

Ping again and with the help of Wireshark guess what is the problem now.

Finally, add a route to reach ROS1 subnet from ROS2 using COL as a gateway:

[student@ros2]$ **sudo ip route add 10.0.1.0/24 via 10.0.2.12**

The ping between ROS1 and ROS2 should now work. Hopefully you've understood a bit more about subnets and routing.

Disable packet forwarding:

[student@col]$ **sudo sh -c "echo 0 > /proc/sys/net/ipv4/ip_forward"**

3. Performance Testing and Tuning

1. On the COL computer, start an iperf3 server, binding it to one of its IP addresses:

   [student@col]$ **iperf3 -s -B 10.0.1.12**

2. On the ROS1 node, start an iperf3 client which connects to the previous server. This will generate a TCP network flow between the two hosts running close to link speed (1Gb/s). Analyze the command output: why is the reported bandwidth lower than the link speed?

   [student@ros1]$ **iperf3 -c 10.0.1.12**

   You can use Wireshark to analyze the TCP connection: Statistics > TCP stream graph > Throughput Graph

3. On the ROS1 node, start an iperf3 client but this time using UDP. UDP having no flow control, we need to tell iperf3 the amount of data we want to generate (**-b 0** stands for "as much as possible"). Analyze the command output: what differences can you observe in comparison to TCP?

   [student@ros1]$ **iperf3 -c 10.0.1.12 -u -b 0**

4. On the ROS 1 node, start an iperf3 client using TCP traffic but lowering the maximum segment size (MSS). Analyze the command output: what differences can you observe in comparison to default TCP?

   [student@ros1]$ **iperf3 -c 10.0.1.12 -M 750**

5. On the ROS1 node, start an iperf3 client using TCP traffic but lowering (one of) the TCP window's max size. Analyze the command output: what differences can you observe in comparison to default TCP?

   [student@ros1]$ **iperf3 -c 10.0.1.12 -w 16k**

   Use Wireshark to show the throughput graph and compare it to the previous situation.

4. Quality of Service

1. Change the network setup so that all three computers are part of the 10.0.1.0/24 network associated with VLAN2. For this, you first need to connect to the ROS2 node using the management network and modify the IP address of the data collection interface.

   [student@col]$ **ssh ros2**

[student@ros2]$ **ip addr**

Identify the name of the interface (10.0.2.11 address)

[student@ros2]$ **sudo ip addr del 10.0.2.11/24 dev INTERFACENAME**

[student@ros2]$ **sudo ip addr add 10.0.1.13/24 dev INTERFACENAME**

Then the port on the switch to which the modified interface is connected needs to be reassigned to vlan 1.

[student@col]$ **telnet 192.168.0.10**

HP-3500yl-48G# **config**

HP-3500yl-48G(config)# **vlan 2**

HP-3500yl-48G(vlan-1)# **untagged** PORT

HP-3500yl-48G(vlan-1)# **sh ru** # check configuration

HP-3500yl-48G(vlan-1)# **exit** # 4 times

Check that ROS1 can be reached from ROS2.

[student@ros2]$ **ping 10.0.1.11**

2. Start two iperf3 servers on COL in two different terminals, each with a different TCP port.

   [student@col]$ **iperf3 -s -B 10.0.1.12 -p 5001**

   [student@col]$ **iperf3 -s -B 10.0.1.12 -p 5002**

3. Start clients to generate traffic from ROS1 and ROS2. Observe and try to explain what is happening to reported data rates.

   [student@ros1]$ **iperf3 -c 10.0.1.12 -t 300 -p 5001**

   [student@ros2]$ **iperf3 -c 10.0.1.12 -t 10 -p 5002**

4. You will now use a QoS control mechanism to better control the sharing of common resources. The packets will be marked by clients with a specific DSCP value classifying them in different "services". And the switch will be configured to process packets according to the value of this field.

   Configure the switch to map DSCP values to different priority queues.

   HP-3500yl-48G# **config**

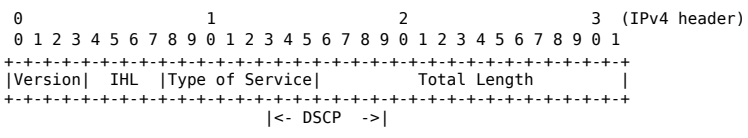
   HP-3500yl-48G(config)# **qos dscp-map 000000 priority 1**

   HP-3500yl-48G(config)# **qos dscp-map 000001 priority 7**

   HP-3500yl-48G(config)# **qos type-of-service diff-services**

5. Start the clients again setting the DSCP field to desired values.

   (This version of iperf3 does not know DSCP but knows ToS which is a deprecated mechanism that uses the same IP header field but interpreted differently; DSCP occupies the 6 most significant bits of the ToS field: $DSCP_2(000001) = ToS_2(00000100)$)



   [student@ros1]$ **iperf3 -c 10.0.1.12 -t 300 -p 5001 -S 0**

   [student@ros2]$ **iperf3 -c 10.0.1.12 -t 300 -p 5002 -S 4** # DSCP(000001)

6. Now starting another pair of client and server, you can observe how the bandwidth is shared among the three concurrent connections.

   [student@col]$ **iperf3 -s -B 10.0.1.12 -p 5003**

   [student@ros2]$ **iperf3 -c 10.0.1.12 -p 5003 -S 0** (or **4**)

5. Bonus Exercise

In the first exercise the switch was configured with a static IP address through the serial connection. Another way of doing this necessary first step is to have a DHCP server configured to provide the switch with an IP. The switch is indeed configured by default to request an IP via DHCP. Let's configure and start a DHCP server on COL:

```
[student@col]$ sudo vim /etc/dhcp/dhcpd.conf
subnet 10.0.1.0 netmask 255.25.255.0 {
        range 10.0.1.100 10.0.1.200;
}
host switch0 { # we want the switch to always have the same ip address
        option host-name "TheAwesomeSwitch";
        hardware ethernet [get the MAC address of the switch from the its back];
        fixed-address 10.0.1.99;
}
[student@col]$ sudo systemctl start dhcpd
[student@col]$ telnet 192.168.0.10
HP-3500yl-48G# config
HP-3500yl-48G(config)# vlan 2
HP-3500yl-48G(vlan-2)# ip address dhcp-bootp
HP-3500yl-48G(vlan-2)# show ip
```

III. Bonus Questions

- What happens if you have a static entry in the ARP cache and the NIC for that target computer is changed?
- How could you find the physical address of the Ethernet card installed on your computer?
- What is the purpose of the TTL field in the IP frame?
- You are the network administrator of a Class C network. Your network consists of 100 computers. Your ISP assigns the address 137.138.111.0/24 to your network. Your network requires 10 subnets with at least 10 hosts per subnet. Which subnet mask should you configure to meet this requirement?
- What is the dotted decimal notation of subnet masks for the following IP addresses?
    - 192.168.10.1/23
    - 5.5.5.5/16
    - 203.40.21.58/27
    - 9.2.3.1/9
- What is the prefix notation of the following subnet masks?
    - 255.255.0.0
    - 255.248.0.0
    - 255.255.255.255
- Using Wireshark to start a new capture. Ping another host using packet size=2900. Stop the capture and view the captured frames. What do you notice?

IV. Useful definitions and glossary

MAC address: (Media Access Control) unique identifier associated with a physical network interface. Also named hardware address or Ethernet address in the case of an Ethernet device.

IP address: (Internet Protocol) numerical identifier associated with a device connected to an IP network.

Most of the time the MAC address is provided by the device manufacturer and never changes, and the IP address is the logical identifier associated to this device by the network manager according to the purpose and location of the device.

Switch: network device that interconnects devices using frame-based switching at the data-link layer (layer 2). For Ethernet, the frame header contains the destination MAC address which allows the switch to determine the physical port to send frames to.

Router: network device that interconnects LANs using network-layer (layer 3) mechanisms. IP makes use of IP addresses and routing tables to implement such mechanisms.

LAN: (Local Area Network) limited-area computer network. It usually consists of devices interconnected via a network switch. Any device belonging to a specific LAN can communicate with any other within this LAN without the need for routing mechanism. A LAN is equivalent to a broadcast domain: broadcast messages reach every device in the LAN.

VLAN: (Virtual Local Area Network) broadcast domain logically created at the data-link layer from a larger physical broadcast domain. For Ethernet, VLANs are implemented with a specific Tag field in the frame header (802.1Q).

SNMP: (Simple Network Management Protocol) protocol to monitor and control network devices. SNMP defines a structured organization of network-related information and the ways to request it from a device. Typically network switches and routers implement SNMP to enable access to monitoring information (traffic metrics, errors, etc.).

RRD tool: (Round-Robin Database Tool) time series database implementing a circular buffer strategy to enforce constant footprint. Widely used to store monitoring information, especially for networks.

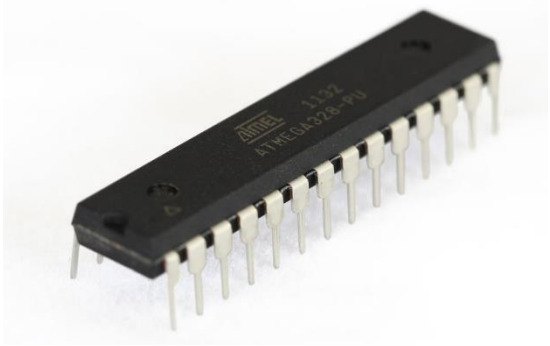
QoS: (Quality of Service) the whole set of mechanisms used to monitor and control the performance of networks. Metrics usually include throughput, latency, packet loss, etc.

DSCP: (Differential Service Code Point, or DiffServ) feature of the IP protocol to classify and manage network traffic. DSCP uses 6 bits in the 8-bit DS field of the IP header to indicate the class of a packet, and network devices may use this information to handle different classes of packets with different policies. Examples: low latency, bandwidth constraint.

DHCP: Dynamic Host Configuration Protocol. Protocol that dynamically allocates unique IP addresses and other network parameters (e.g. hostname, default gateway) to hosts on demand.

10. *Lab 10: Microcontrollers Exercise*

Maurício Féo Rivello (m.feo@cern.ch)



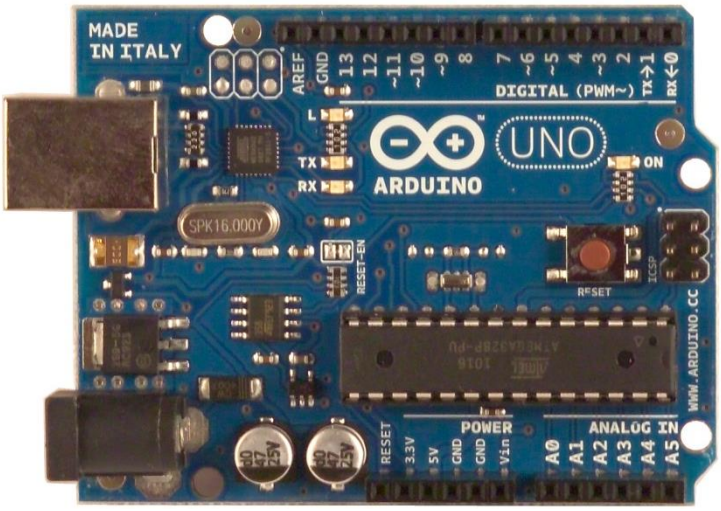
Introduction:

Microcontrollers are small computers with all its components (CPU, memories, peripherals) integrated on the same chip. Apart from their capability of processing data, they are low power, usually inexpensive devices that easily interfaces with sensors and actuators, making them perfect to use in embedded systems.

On this lab we are going to learn the basics about microcontrollers, how to use and program it, as well as common applications and explore the most relevant peripherals through the hands-on exercises and a challenge.

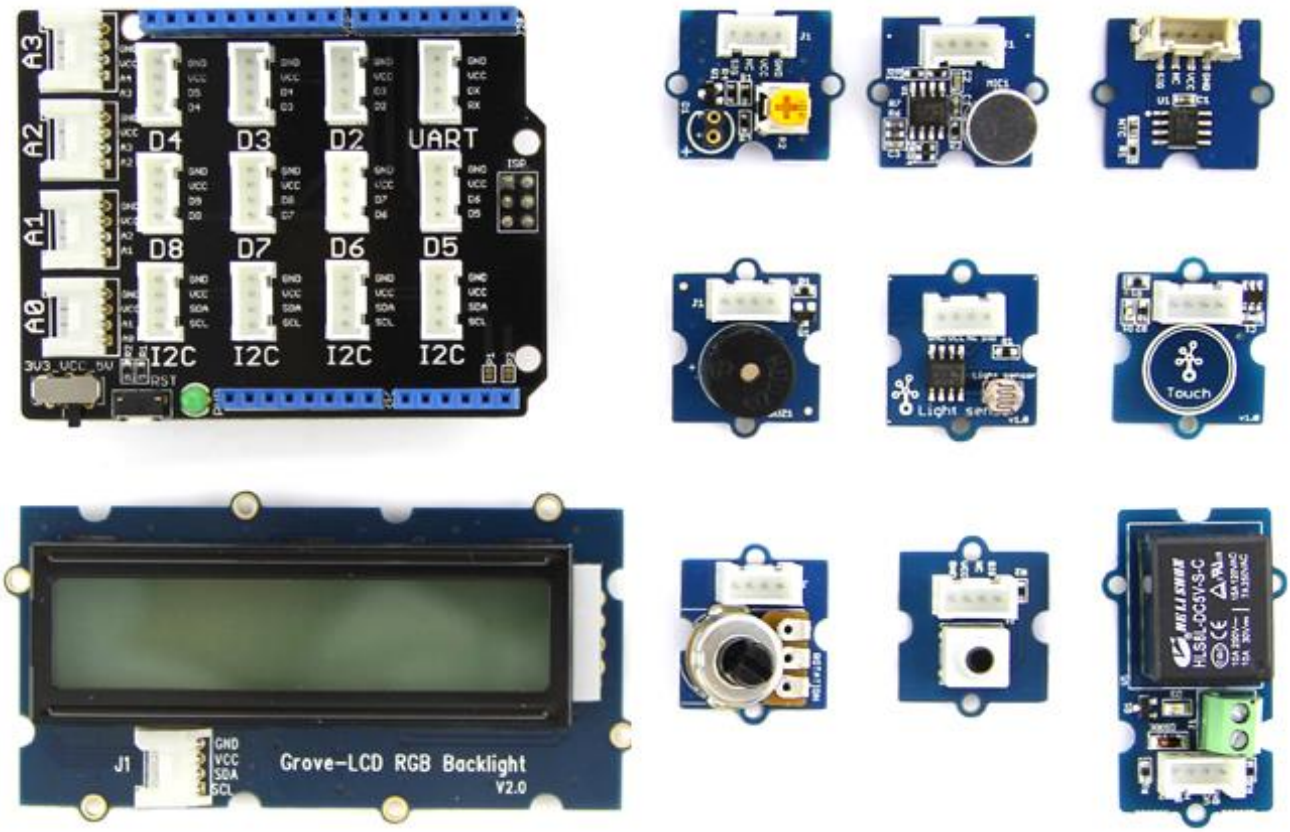
Arduino:

Arduino is an open-source electronic prototyping platform based on easy-to-use software and hardware. In short, it is the most popular microcontroller development board, with a lot of "shields" (extension boards that you pile up on top of the Arduino) to add functionalities and libraries to use the most common sensors and devices used with microcontrollers.



Grove Starter Kit:

Together with the Arduino, we are going to use the Grove Starter Kit, which provides a few gadgets (sensors, actuators and a LCD display) together with a shield and cables that provides a nice plug-and-play interface to ease and speed up the process of wiring the gadgets together.



Setting up the Arduino.

In case you want to set up the Arduino on your own laptop, the Arduino official webpage provides a very simple quick start guide. Basically you just have to download the software, plug the board on the USB port and install it. For further details, visit: http://arduino.cc/en/Guide/HomePage

Before you start:

The Arduino software and drivers will be already configured on the lab PC. Plug the Arduino board on the USB port of the PC and start the Arduino IDE (desktop shortcut). Select the Board you are using (Uno) under the IDE Menu “Tools”. Then select the Port under the same menu. The Port number depends on which USB the Arduino is connected to and it is listed after you plug it in.

Hands On Exercise

We are going to do a few exercises to learn the basic functions of Arduino and then, using the hardware provided, we are going to implement a simple project so solve a challenge with what we have learned so far. As you do the exercises, save your code because you might reuse it in the project ;-)

Visit the site rivello.me/isotdaq for a summary of the functions that you will mostly use during the exercises. For detailed information about the main functions from the Arduino libraries, please refer to: www.arduino.cc/en/Reference

HANDS ON

Exercise 1: Blinking a LED.

Blinking a LED is the microcontroller equivalent of printing "Hello World". The basic structure of an Arduino program is very simple: It must contain a **setup()** and a **loop()** function. Open the Arduino software, start a new program and write the following structure:

```
void setup() {
  ...
}

void loop() {
  ...
}
```

The **setup()** function is executed once every time the Arduino is powered on. As the name suggests, it should be used for setting up your application, like initializing classes and variables, declaring pin modes, etc.

The **loop()** function keeps being executed in loop ad eternum. This is where the main logic of your program should be.

To get started, let's build a program to blink a LED. There is already a LED attached to pin 13 on every Arduino. You should first of all declare the mode (output or input) of the pin to be used with the following function:

```
pinMode( [pin_number] , [OUTPUT/INPUT] );
```

In our case, the pin_number is 13 and the mode is OUTPUT. So to blink the LED we can write a HIGH and then a LOW signal to the pin 13, adding a delay between each command.

```
digitalWrite( [pin_number], [HIGH/LOW] );
delay( [milisseconds] );
```

Now try to make yourself a program to blink the LED on pin 13 once every second.

Exercise 2: Reading the state of a Push-Button.

The same way we can declare a pin as output and write a state (HIGH or LOW) to it, we can also declare one as INPUT and read its state with the following function:

```
boolean_variable = digitalRead( pinNumber );
```

It returns TRUE or FALSE (HIGH or LOW). Let's use it to read the status of the Push-Button from the Grove kit, which can be connected to any of the digital pins of the Arduino. In the Grove shield, these are the connectors marked with the letter D.
Use the code from exercise 1 and the LED to identify the pressing of the button.

Exercise 3: Serial communication.

In this exercise, we're going to use the Arduino Serial library to send and receive characters from the PC using the Serial Monitor tool of the Arduino IDE. First thing to do in your code is to initialize the Serial with the following command:

```
Serial.begin( 9600 );  // (9600 is the baudrate).
```

As it only needs to be executed once, it should be on the setup() function. Now there are 3 more important functions to learn. The available() returns whether there is a character available to be read:

```
boolean_variable = Serial.available();
```

The read() reads into a variable a single character from the serial buffer, and the println() works like in C, printing on the serial port a string. It also converts numbers into characters. The Serial library is very handy and there are more functions. For reference visit: http://arduino.cc/en/Reference/Serial

```
byte inByte = Serial.read(); // reads into inByte a character from the buffer.
Serial.println("Hello World");  // Writes a string to the serial port.
```

Now let's write a code that does the following:

1) Toggles the LED whenever the button is PRESSED DOWN. (Not when released)
2) Prints to the Serial port for how long the LED has been ON whenever it is turned OFF.

You can make use of one of the time functions, like micros(), which returns the amount of microsseconds since the microcontroller started:

```
unsigned long var = micros();
```

Once you succeed, what about controlling the LED from the Serial Monitor as well? Use the described functions to try to control the LED from bytes sent to the Arduino from the Serial Monitor.

Exercise 4: Reading an analog input.

The Arduino UNO has 6 analog inputs. Reading one of them is as easy as reading a digital pin. It returns an integer value ranging from 0 to 1023. The scale varies from 0V to a reference voltage, which is by default 5V (but can be changed). In short: 0 -> 0V; 512 -> 2.5V; 1023 -> 5V.
To read an analog pin, use the following function:

```
int integer_variable = analogRead( pin_number );
```

The Grove Starter Kit has four analog sensors: a sound sensor, a light sensor, a temperature sensor and a rotary switch. Plug any of them into one of the connectors labeled with the letter 'A' and try the following code (don't remove the code from the previous exercises as it will be used again):

```
int analog_value;
void setup() {
  Serial.begin(9600);
}
void loop() {
  analog_value = analogRead( plug_number );
  Serial.println( analog_value );
  delay(200);
}
```

Open the Serial monitor and see the results and how they change when you interact with the sensor used. Try the Serial Plotter as well.

Exercise 5: Interrupts.

Note what happens when you try checking the time between presses of the button from exercise 3 while running the code from exercise 4. Did you note that the code on the loop cannot identify properly when the button is pressed when the processor is "stuck" on the delay(200) function?
It happens because you are reading the button by polling it's state at every loop. For critical applications where you need a precise timed response, interrupts should be used instead.
I/O interrupts are implemented in a very simple way by Arduino. You can set up an interruption in a pin using the following function (Arduino UNO only supports interrupts on the pins 2 and 3):

```
attachInterrupt( digitalPinToInterrupt( pin ), function_To_Be_Called,
RISING/FALLING/CHANGE );
```

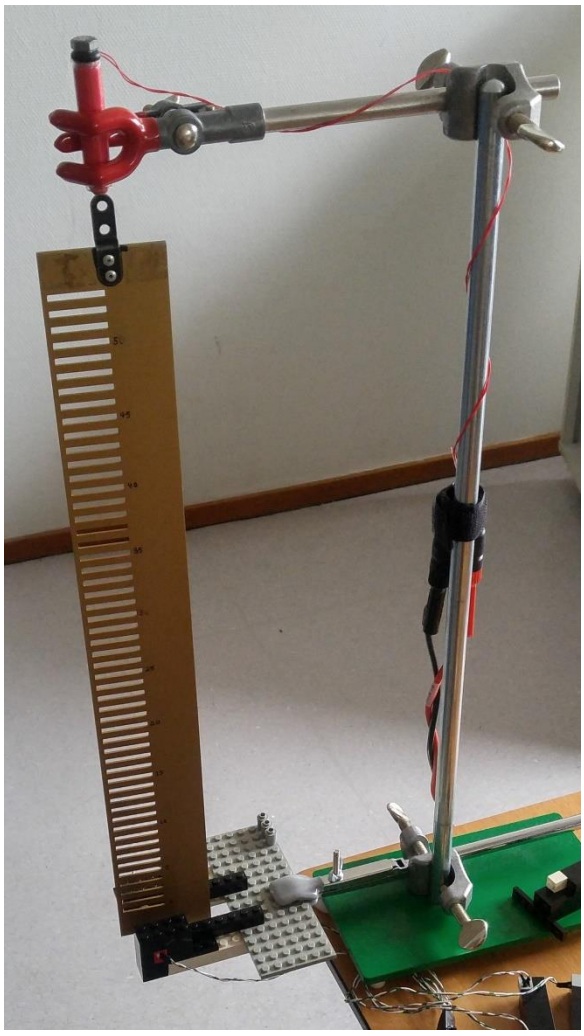
Now whenever the value of the pin rises or falls or changes (depending on your choice) the function passed will be called. The name of the function is arbitrary and you should define it in your code:

```
void function_To_Be_Called(){
// code to be executed on the interrupt
}
```

Reimplement the printing of the time in between presses of the button but now using interrupts instead of polling.

Challenge: Measuring the acceleration of gravity.

In this challenge we are going to use all what we learned above to measure the acceleration of gravity using a drilled bar that falls through an infrared beam. The space in between the first edge of each hole is 7.2mm on average. The precise measurements can be found on rivello.me/isotdaq
We have an infrared LED and an infrared phototransistor that works like a digital input for the Arduino (just like the push button). For a smooth drop of the bar, we can use an electromagnet controlled from the relay board of the Grove Starter Kit.
Given all that we learned and the apparatus available, how could one calculate the acceleration of gravity? For any help setting up the equipment, ask the tutor. And good luck! ☺



11. *Lab 11: Storage Exercise*

Overview

The aim of this lab is to provide an overview about how to configure and evaluate a storage setup.

Objectives

- partition storage units
- setup raid systems
- performance measurements
- evaluate different raid strategies
- evaluate different storage technologies

Tools

dd:

The linux dd tool allows you to make copies of files at a block level. Its basic syntax is :
dd if=/path/to/input/file of=/path/to/output/file bs=X count=Y
Where X is the block size of the individual transfers, and Y the amount of blocks you want to copy. We recommend 32M as X value.
The **seek** option allows you to skip a certain amount of blocks at the start of the output.
The **skip** option allows you to skip a certain amount of blocks at the start of the input.
The **oflag** is used to set particular flags that are used on the output stream. In our case the 'direct' option is useful. It forces the operating system to not use the write behind cache on the stream.

fdisk and sfdisk

Be very very careful. These tools can very easily wipe out the entire operating system if used on the wrong disk. Make sure you are only working on /dev/isotdaq/XXX

The fdisk tool is used to manipulate the partition table of a disk. The tool has an interactive shell and the important commands for the following exercises are:
n: create a new partition
d: erase a partition
w: write the new partition table to disk
p: show the current partition layout
h: help

The sfdisk tool allows you to dump the partition table of a disk into a file using the -d option, and then to apply the same schema to another disk using the redirect operator (<).
For example: sfdisk -d /dev/isotdaq/disk1 > file //to dump the partition table into file

sfdisk /dev/isotdaq/disk2 < file //to read a partition table from a file

mdadm

The mdadm tool is used to manipulate the Linux software raid devices.
Its main running modes are 'Create' and 'Manage'. In order to create a new raid, the 'Create' mode

is obviously to be used. You need to provide it with information about the raid level you want to create, and on which device it will reside.

Create a raid array:
mdadm --create=md<x> --level=<x> --raid-devices=<N> <device1> <device2> … <deviceN>

Stop a raid array:
mdadm --misc --stop /dev/md/md<x>

fio

Fio is an advanced tool for characterising IO devices. It can be used to simulate different IO loads and profiles and evaluate disk performances. In our case we will use it to measure iops at a fixed block size. The following syntax will be enough for all of the exercises:

fio --rw=<opt1> --bs=<opt2> --runtime=<opt3> --filename=<opt4> --direct=1 --ioengine=libaio --name=isotdaq

opt1: randread or randwrite
opt2: 4096
opt3: 60
opt4: /dev/isotdaq/disk<X> or /dev/md/md<Y>

Exercises

Exercise 1: Determine the raw throughput of a single disk

For this exercise use **dd** to measure the throughput of one of the hard disks in the machine for read and write performance.
For write performance use /dev/zero as input file and /dev/isotdaq/disk<N> as output.
For read performance use /dev/isotdaq/disk<N> as input file and /dev/null as output.

Use the seek and skip options of dd to measure the performance at the end of the disk too.
Use the count option to write/read only 1GB of data.
Use oflag=direct during writing.
Hint: If you use an I/O block size of 32M the end of the disk should be around 7000

Questions:

- What is the read/write throughput of the disk in MB/s?
- The oflag=direct option circumvents the operating system cache for the disk. Why is this important for this measurement?
- Which disk corresponds to which physical disk inside the enclosure?
- Optional: Usually, for disk based storage, the write throughput is the same as the read throughput. Do you have any idea why it is different in this case?

Exercise 2: Determine the IOPS of a single disk

For this exercise use the **fio** tool to measure the random read and write Input Outputs Per Second (IOPS) of a single disk.
Good values for the parameters are a run time of 60s and IO size of 4096. For reading use --rw=randread. For writing use --rw=randwrite.

Questions:

- What are the values for random reading and random writing for these disks?
- Why are these important values?
- Why are the reading and writing values different?
- Using the result from Ex.1: Calculate the IOPS of the throughput measurement. Why are the values for random IO so much smaller?

Exercise 3: Partitioning the disks

For the purpose of this exercise, we will create 4 partitions on each disk. They will later be used to host different raid types.

Create 4 partitions on /dev/isotdaq/disk1 using fdisk. The partitions should be of type 'primary', and 2 Gb each.

After this you can either use fdisk to create the same partitions on the other 3 disks or use sfdisk to dump the layout of the first disk and import it to the other three disks.

Questions:

- Make sure that /dev/isotdaq/disk<0-3>part<1-4> exist

Exercise 4: Creating the raid arrays

You will now create 4 different kinds of raid sets to measure their different properties. Use mdadm to create the following raids:
Raid0 on disk1part1, disk2part1, disk3part1, disk4part1
Raid1 on disk1part2, disk2part2
Raid5 on disk1part3, disk2part3, disk3part3, disk4part3
Raid6 on disk1part4, disk2part4, disk3part4, disk4part4

Hint: To create a raid set of a particular kind use
mdadm --create md<x> --level=<x> --raid-devices=<N> <device1> <device2> … <deviceN>

Raid levels are 0, 1, 5 and 6. For easier recognition you can use the same number <x> for the raid level and the device name.

You can create and initialize multiple arrays in parallel.

Questions:

- Use 'cat /proc/mdstat' to follow the initialisation of the raid arrays.
- Why do raid1, raid5 and raid6 need initialisation and raid0 does not?
- Explain the different sizes of the finished raid sets.

Exercise 5: Performance Measurements

In this exercise you are going to explore the different performance values of the different raid types. Use dd and fio like in exercise 1 and 2 to determine the throughput and IOPS of the four raid sets you created earlier. For the throughput you can skip the measurement for the end of the device (Why?).
Remember to use /dev/md/md<x> for your measurements and not /dev/isotdaq/...
Questions:

- What values did you expect for the different raid sets?
- Is what you got coherent with what you expected?

- Which raid would you use for a data acquisition system?
- What would you use for a normal file system/database?

Exercise 6: Failures

Remove disk1 and check if you can still access all your raid sets. Repeat the performance measurements for the raid sets that still work.

- Explain why the performance of all raid sets has deteriorated.
- Would you still choose the same raid type for your data as in exercise 5?
- Why does raid 6 exist?

12. *Lab 12: DAQ Online Software*

1. Outline

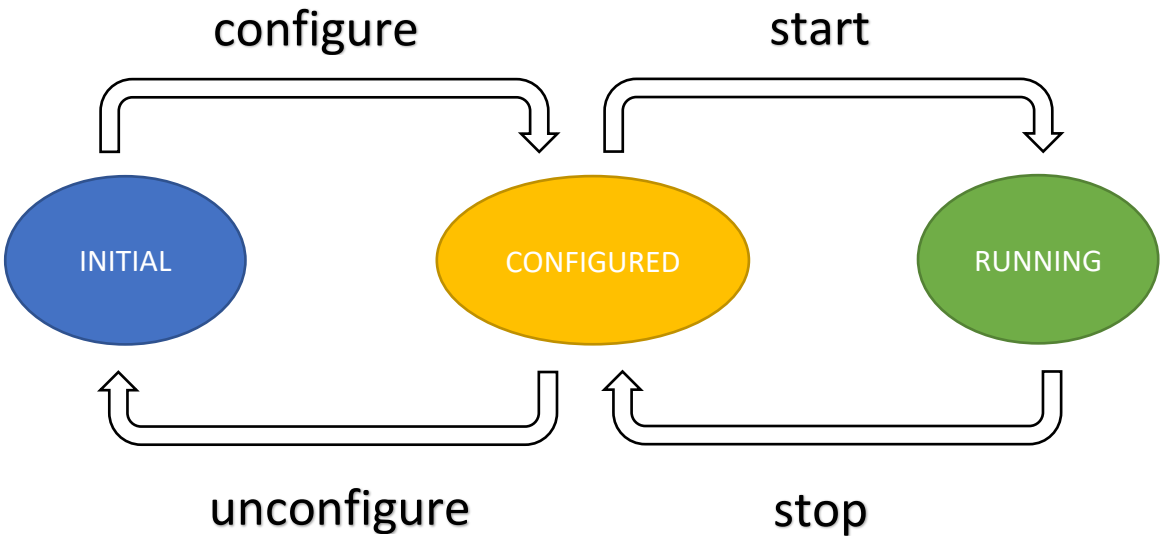
The aim of this exercise is to develop a control system for a DAQ system, which acquires health-monitoring data (CPU, Memory usage, etc.) from a set of hosts. The system is composed of different independent sensors that simulate our data acquisition applications. The control system steers the behavior of these applications, individually and, in the final configuration, through a central controller.

Students will develop the system relying on the control and configuration capabilities provided by a simplified version of a real DAQ system.

Students will learn about the most common situations in controlling DAQ applications in a distributed environment and how they can be addressed. They will also learn about the main capabilities provided by the online software framework in a DAQ system.

Sensors

The Sensors are the data acquisition applications of our simple DAQ system. The main goal of a Sensor is to read and publish machine health metrics (CPU, Memory usage, etc.). Sensors can differ in what metrics they read and how they are implemented.
Since we want a uniform way to control potentially different sensors, we define a simple Finite State Machine (FSM):



All the applications in the system must behave according to this FSM!
The FSM defines two main concepts:

1. Transition commands
   `configure, unconfigure, start, stop`

2. States
   `INITIAL, CONFIGURED, RUNNING`

Sensors are publishing the following data:

1. *`hostname`*: Indicates the name of the machine the sensor is running on.
   Published in every state.

2. *`runtime`*: Increasing counter measuring the time (in seconds) since last start.
   Published when the application is in the RUNNING state, this value is updated every second.

3. *`type`*: indicates the type of the published information (e.g. “CPUUsage” or “MemoryUsage”).
   Published when the application is in the CONFIGURED or RUNNING states.

4. *`value`*: application-provided data (e.g. idle CPU percentage or free memory).
   Published when the application is in the RUNNING state.

All applications publish their FSM state (i.e. INITIAL, CONFIGURED, RUNNING). **Additionally, they provide the application status that can be OK or ERROR - if the application raises an Exception.**

2. DAQ Python framework

For this exercise, a simplified version of the DAQ system was developed in the Python programming language. This system consists of a master server and applications publishing information to it. The applications need to implement a common interface to expose a REST API to the master server.

A web interface running on the master server is also provided, showing an overview of all applications, information published and allows sending commands to each application. The master server also exposes a REST API, allowing each of the application to retrieve data from and send commands to other applications.

2.1 HTTP REST API

All communication between the applications and master server is done over HTTP. Both, the master server and the application run a web server and expose GET/POST endpoints that can be called. For applications, the communication is hidden behind the *DAQInterface* and is not exposed to users. They only need to implement this interface.

When an application is started the following HTTP requests happen in the background:

1. The application subscribes itself to the master server calling http://master.server/subscribe and submits the parameters *hostname*, *port* and *name*. Then the master server knows how to make requests to this application.

2. Before the website http://master.server/ is visited a request is sent to all subscribed applications on http://hostname:port/hostname, http://hostname:port/value, http://hostname:port/state, …
   The HTTP response of each of this calls is a string and is displayed on the website.

3. A state transition on the application can be triggered by calling
   http://hostname:port/<configure>|<start>|<stop>|<unconfigure>

A controller contains names of all sub-applications and can make calls to the master server specifying the sub-application and state transition like
http://master.server:port/name/<configure>|<start>|<stop>|<unconfigure>
and the master server will execute a state transition on the application with this name.

All this HTTP requests are hidden behind method calls on Python classes implementing the *DAQInterface*.

hello_monitor.py application

The *hello_monitor.py* application is the simplest possible monitoring application. It always publishes the value "Hello World" of type "Greeting". Before running the application, the master server has to be started:

```
$ cd lib
$ python master_server.py
```

Then, all applications can be run as a regular python application, e.g.:

```
$ python hello_monitor.py <name>
# Application started with ID: 25818
# Visit http://localhost:36500 for an overview of all applications.
```

To uniquely identify each application a *<name>* needs to be assigned to the application (e.g. `python hello_monitor.py hello_monitor`).

The state and data published can be viewed by visit the mentioned URL (Figure 1).

Applications

Name	Hostname	Status	State	Runtime	Type	Value
memory	lxplus093.cern.ch	OK	RUNNING stop	4 s	MemoryUsage	25%

Figure 7: Web view

It is possible to trigger state changes from the web interface.

The *HelloMonitor* class inside *hello_world.py* implements the *DAQInterface*. Having all applications implement the same interface unifies the way we control them.

The DAQInterface expects you to provide the following methods:

1. *__init__():* Is called on startup and allows for parsing of passed arguments.

2. configure(), unconfigure(), start(), stop(): This code is executed if a state change is triggered on the application. If it does not raise an Exception the state change is executed.

3. value(): Returns the data to be published.

Exercise 12.0: Change the value returned by the hello_monitor.py application

Change the code in *hello_monitor.py* to return "Hello ISOTDAQ!".

Note: If there are no syntax errors the application will be automatically restarted after saving the file inside the editor.

Exercise 12.1: Fix the CPU sensor according to the sensor rules

Compare the behavior of the CPU sensor with the Memory sensor, and make sure that it complies with the finite state machine rules defined in the Outline.

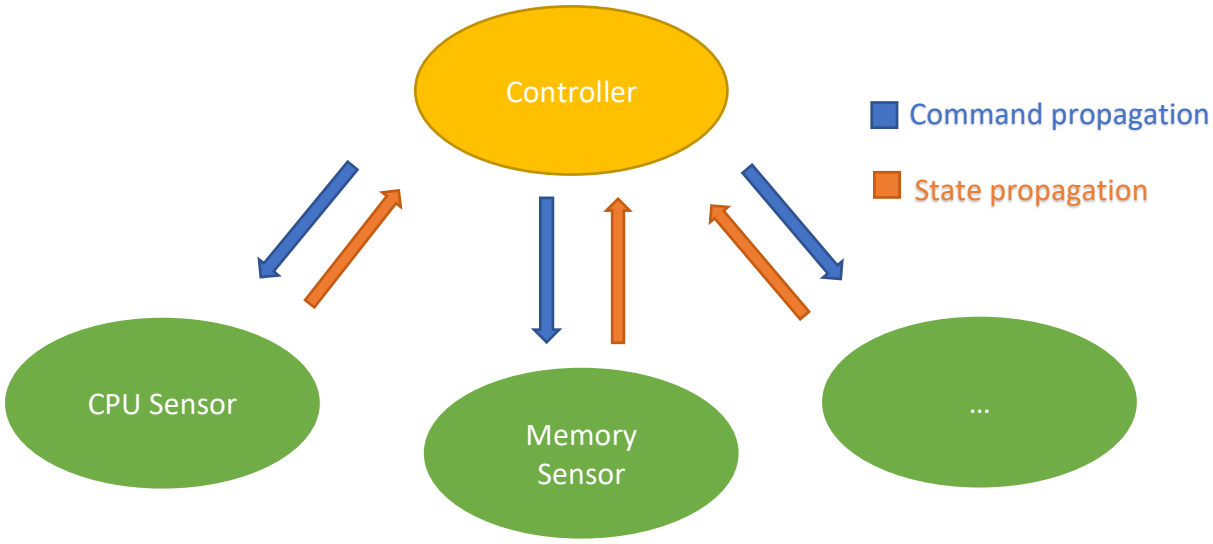
Start both sensors in different terminals and send commands to them to spot the different behavior in the web interface.

```
$ python cpu_monitor.py <name>
$ python memory_monitor.py <name>
```

Find the bugs and happy fixing!

3. Controller

The second part of this exercise focuses on the role of the Controller application.
The distributed monitoring system has to manage and gather information from a set of sensors running on different machines. All these sensors have to be properly configured and running at the same time to provide meaningful data, and this introduces the need for an entity to manage the control flow. This is the role of the Controller, an application that receives FSM commands and forwards them to a set of children applications. The Controller application must also check the proper execution of the FSM transitions, deal with common problems, etc.



The Controller implements the same interface as every other application and is able to receive commands.

Exercise 12.2: Start a Controller application

Start the CPU monitor:
```
$ python cpu_monitor.py cpu_monitor
```

Start the memory monitor:
```
$ python memory_monitor.py memory_monitor
```

Start the controller by passing the names of the applications you want to attach to:
```
$ python controller.py <name> cpu_monitor memory_monitor
```

Now use the web interface to send commands to the controller and observe the propagation of commands and state changes.

Exercise 12.3: Improve controller to handle a faulty application

The faulty sensor is an application simulating a failure in the processing of a command. This can happen for many different reasons in the real world. Students have to improve the Controller code in order to handle this type of failure.

Start the faulty application:
```
$ python faulty_memory_monitor.py faulty_memory_monitor
```

Attach it to a controller:
```
$ python controller.py <name> faulty_memory_monitor
```

When attempting a *configure* state change the *faulty_memory_monitor* may sometimes return "ERROR".

Improve the controller to manage the faulty application. Think about multiple ways to handle errors.

Exercise 12.4: Enhance the Controller's flexibility

The controller implements a strict FSM logic. If one controlled application changes state, the controller goes into the error state and will not clear the error even if all children applications go back into the same, consistent state.

Change the controller logic to clear the error state if all child applications are in a coherent state, corresponding to the state of the controller.

Exercise 12.5: Add a root Controller and send commands to all sensors

A controller can be attached to another one, they implement the same interface as applications.

Run a root controller controlling all other controllers. Both groups work together on this exercise.

13. *Lab 13: System on Chip (SoC) FPGA*

(Version: 1.1)

Tutor:
Johannes Wuethrich (CERN) (johannes.wuethrich@cern.ch)

Lab Developers:
Manoel Barros-Marin (CERN) (manoel.barros.marin@cern.ch),
Elena-Sorina Lupu (Caltech) (eslupu@caltech.edu)

Contents

13.1. *INTRODUCTION*

The aim of the **SoC FPGA laboratory** at the **International School Of Trigger Data AcQuisition (ISOTDAQ)** is to provide students a brief overview of the different stages in the SoC FPGA design workflow and the knowledge to determine when a SoC FPGA is the most appropriate core for their project. After the completion of the lab, the students should be able to understand the interaction between the two main building blocks of a SoC FPGA (FPGA fabric and Hard Processor (HCPU)) and assess the challenges of implementing such a system.

1.1 *SoC FPGA*

Processors (CPU) and **Field Programmable Gate Arrays (FPGAs)** are the hardworking cores of most Trigger DAQ systems. Integrating the high-level management functionality of processors and the stringent, real-time operations, extreme data processing, or interface functions of an FPGA into a single device forms a more powerful embedded computing platform. **System on Chip (SoC) FPGA** devices integrate both processor and FPGA architectures into a single chip. Consequently, they provide higher integration, lower power, smaller board size and higher bandwidth communication between the processor and FPGA. They also include a rich set of peripherals, on-chip memory, an FPGA-style logic array and high speed transceivers. As a result of the previously mentioned qualities, coupled with a wide range in terms of cost and performance, the SoC FPGA devices are becoming more and more popular among digital electronics and software designers.

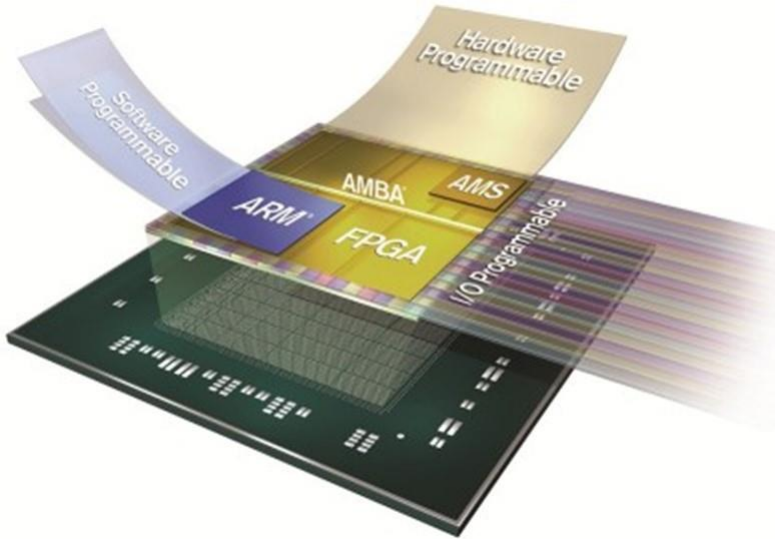


Figure 1: *Model of SoC FPGA*

1.2 *SoC FPGA workflow*

A typical SoC FPGA workflow is illustrated in Figure 2.

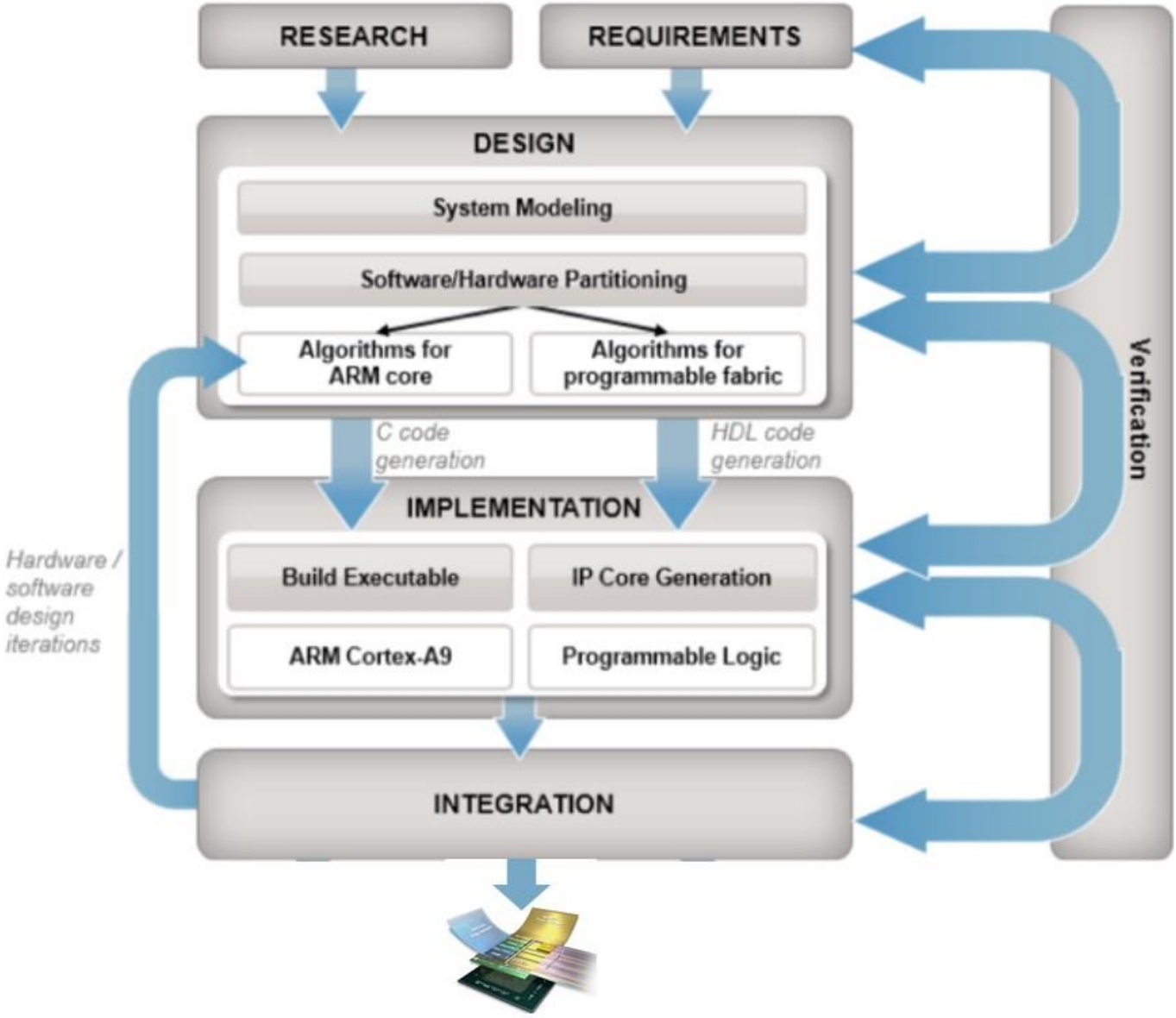


Figure 2: *SoC FPGA workflow*

2 LAB SETUP

2.1 Specifications

With the help of your team, you are going to implement an **emulator of a typical TDAQ system in High-Energy Physics (HEP) experiments**. For simplicity reasons, this TDAQ system may be divided in two groups. On one side, the **front-end (FE)** electronics, placed "close to the experiment". On the other side, the **back-end (BE)** electronics, placed "close to the control room". In this case, the communication between the FE and the BE is performed through a bidirectional serial link over copper cable. In a typical HEP experiment, the analogue signal from the sensor is filtered and shaped by the analogue FE electronics. This conditioned signal is digitized by an Analog to Digital Converter (ADC). Once in the digital domain, the raw data from the ADC is evaluated by the digital FE electronics (please note that this evaluation may be done in the analogue domain instead). When an event of interest occurs (e.g. particle crosses the sensor), the value of the raw data surpasses a preset threshold, triggering its transmission to the BE through the serial link. In the BE, the serial data is deserialized, processed, stored in memory and sent to the farm of computers though Ethernet by a CPU. Once in the farm of computers, this data may be post-processed, analysed and plotted by the users. In this lab, a devkit featuring an ambient light sensor (ALS-GEV) plays the role of the HEP experiment sensor and its related analogue and digital FE electronics. The communication between the FE and the BE is performed through a bidirectional Inter-Integrated Circuit (I2C) link over copper cable. A SoC FPGA devkit (MYIR Z-turn) is used as BE electronics. This SoC FPGA devkit communicates through Ethernet with a laptop running Python scripts over Linux, which emulates the farm of computers and the user's computer. The block diagram of the emulated HEP experiment is depicted in Figure 3.

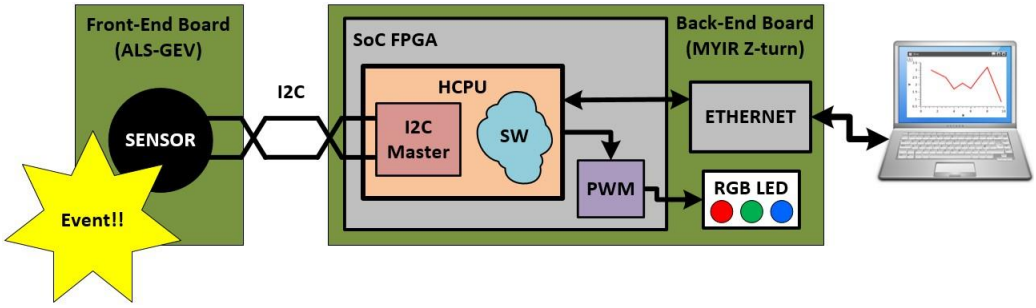


Figure 3: *HEP experiment emulation block diagram*

2.2 Hardware

2.2.1 ALS-GEVB

The **Front-End electronics** of our emulated HEP experiment is based on the **Ambient Light Sensor (ALS) Shield Evaluation Board (ALS-GEVB)**, ilustrated in Figure 4). This board is the devkit of the NOA1305, an ambient light sensor (ALS) designed for handheld applications. The NOA1305 integrates a 16-bit ADC, a 2-wire I2C digital interface, internal clock oscillator and a power down mode. The built in dynamic dark current compensation and precision calibration capability coupled with InfraRed (IR) and 50-60 Hz flicker rejection enables highly accurate measurements from very low light levels to full sunlight. The device can support simple count equals lux readings in interrupt-driven or polling modes. The NOA1305 employs proprietary CMOS image sensing technology from ON Semiconductor to provide large signal to noise ratio (SNR) and wide dynamic range (DR) over the entire operating temperature range.

The optical filter used with this chip provides a light response similar to that of the human eye.

Figure 4: *Front-End board (ALS-GEVB)*

2.2.2 MYIR Z-turn Board

The **Back-End electronics** of our emulated HEP experiment is based on the **MYIR Z-turn Board**, which is a low-cost and high-performance System On Chip FPGA devkit. This board is based on the Xilinx Zynq-7000 family, featuring integrated dual-core ARM Cortex-A9 processor with Xilinx 7-series Field Programmable Gate Array (FPGA) logic.

The MYIR Z-turn Board takes full features of the Zynq-7010 (or Zynq-7010) SoC FPGA, it has 1GB DDR3 SDRAM and 16MB QSPI Flash on board and a set of rich peripherals including USB-to-UART, Mini USB OTG, 10/100/1000Mbps Ethernet, CAN, HDMI, TF, JTAG, Buzzer, G-sensor and Temperature sensor. On the rear of the board, there are two 1.27mm pitch 80-pin SMT female connectors to allow the availability of 96 / 106 user I/O and configurable as up to 39 LVDS pairs I/O.

The Z-turn Board is capable of running Linux operating system. MYIR has provided Linux 3.15.0 SDK, the kernel and many drivers are in source code. An image of the MYIR Z-turn Board, highlighting its main components is illustrated in Figure 5).

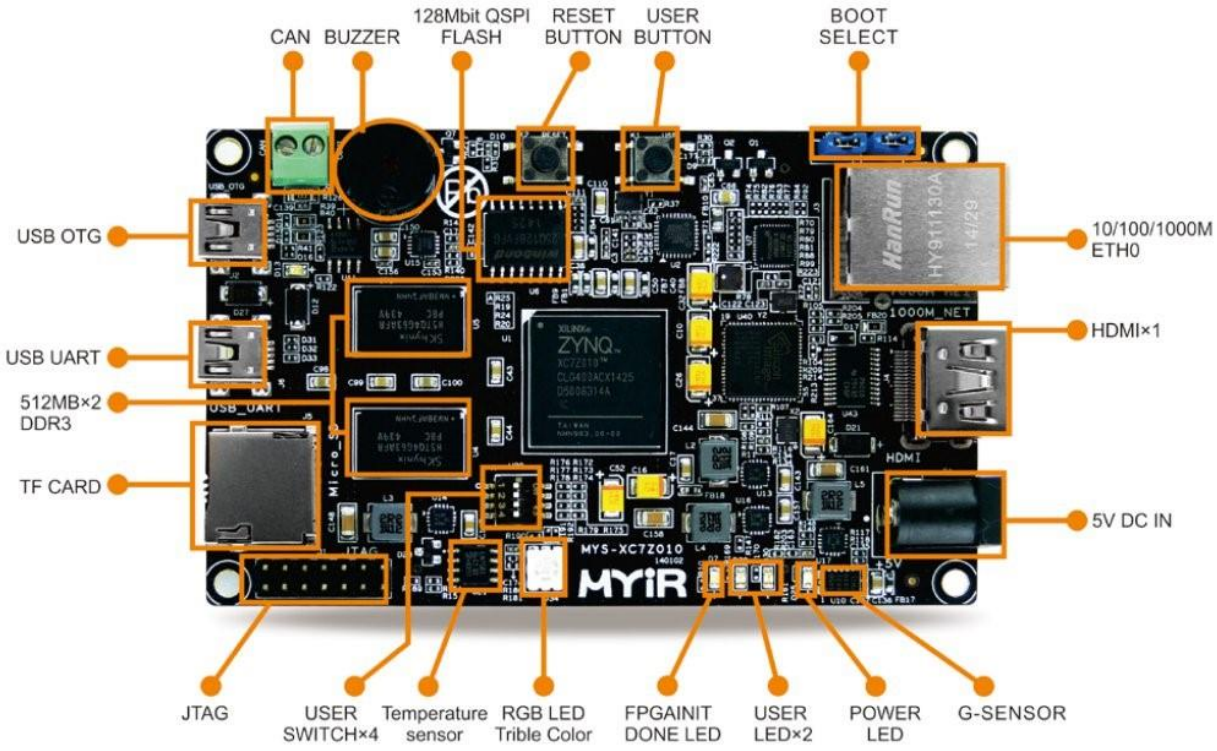


Figure 5: *Z-Turn board capabilities*

2.2.3 Laptop

The **farm of computers and user's laptop** of our emulated HEP experiment is based on a **laptop running Linux** (Ubuntu in this case). Here the different Electronic Design Automation (EDA) tools for implementing the SoC FPGA project and software scripts for data analysis and plotting are executed. An example of laptop running Linux is illustrated in Figure 6.

Figure 6: *Laptop running Linux*

2.2.4 Miscellaneous

The rest of the hardware components required for this lab are the following:

- **Custom cable** (ALS-GEVB power & ALS-GEVB/MYIR Z-turn I2C communication) (see Figure 7)

Figure 7: *Custom cable*

- **Mini-USB to USB cable** (MYIR Z-turn power & MYIR Z-turn/Laptop UART communication) (see Figure 8)

Figure 8: *Mini-USB to USB cable*

- **RJ45 CAT5e cable** (MYIR Z-turn/Laptop Ethernet communication) (see Figure 9)

Figure 9: *RJ45 CAT5e cable (Ethernet cable)*

- **Xilinx Platform Cable USB II** (MYIR Z-turn Zynq JTAG programming) (see Figure 10)

Figure 10: *Xilinx Platform Cable USB II*

2.3 GateWare

A typical SoC FPGA GateWare (GW) is composed by two main parts. On one hand, the SoC, with the HCPU and its peripherals (e.g. I2C, timers). On the other hand, FPGA fabric and its hard blocks (e.g. BRAMs, Multi-Gigabit Transceivers (MGT)).
For this lab, you have to implement the GW for MYIR Z-turn. As previously mentioned, the MYIR Z-turn features a Xilinx Zynq SoC FPGA (please note that there are other vendors in the market such as Altera, Microsemi, etc.). For that reason, the Xilinx EDA tool for FPGA and SoC FPGA (Vivado) is used throughout this lab.

2.3.1 SoC

The typical approach for implementing the SoC part is through **schematics and wizards**. This facilitates the configuration of such a complex, but well defined, architecture. An example of SoC FPGA schematic and wizard are illustrated in Figure 11.

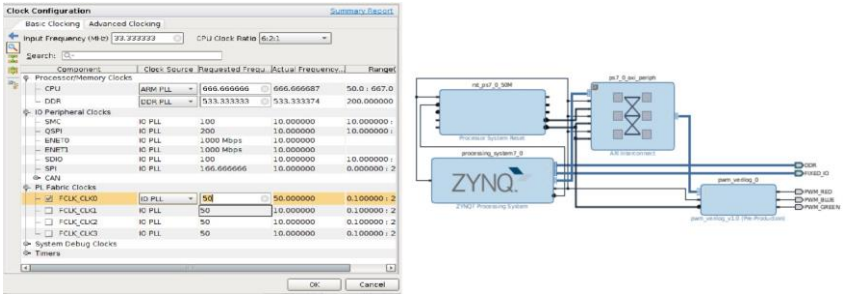


Figure 11: *SoC FPGA wizard (left) & schematic (right) example*

The SoC part of the GW for our emulated HEP experiment requires a HCPU with an I2C master for communicating with the ALS-GEVB and a UART for communicating with the Laptops (an Ethernet interface for communicating with the laptop will be added at the end of the lab).

2.3.2 FPGA fabric

The modules on the FPGA fabric are usually implemented through **Hardware Description Language (HDL)**, such as SystemVerilog or VHDL. This allows to exploit the versatility of the FPGA fabric. In some specific cases, it may bee interesting to implement these modules using other techniques (e.g. schematics, high-level synthesis), but HDL is the most common by far. An example of HDL code (SystemVerilog in this case) is illustrated in Figure 12.

```
9  module up_counter    (
10 out     ,  // Output of the counter
11 enable  ,  // enable for counter
12 clk     ,  // clock Input
13 reset      // reset Input
14 );
15 //---------Define Ports-------------
16 //---------Output Ports-------------
17 output [7:0] out;
18 //-----------Input Ports-------------
19 input enable, clk, reset;
```

Figure 12: *HDL code example*

The FPGA fabric side of the GW for our emulated HEP experiment will feature a **Pulse Width Modulation (PWM) block** for controlling an on-board **RGB LED**. For the details of the PWM module, please refer to Appendix A.

2.3.3 SoC/FPGA fabric interface

The most common approach for interfacing the modules of the FPGA fabric with the SoC part is to generate a **schematic wrapper** for the HDL module and connect it to the HCPU as any other SoC peripheral (this is the approach used in this lab). Please note that the other way around would be possible too, using a HDL wrapper for the SoC part and add it to the rest of the HDL code.

2.4 Software

The HCPU of the SoC FPGA requires software to execute for performing the tasks assigned to it. There are two main approaches when implementing software for a SoC. This software can be either **stand-alone** or executed over an **Embedded Operating System (EOS)**, such as embedded Linux. It is important to mention that either approach you chose for your project, it is necessary to export the SoC FPGA GW to the Software Development Kit (SDK) in order to generate the required software drivers. As previously mentioned, the SoC FPGA of our emulated HEP experiment is base on Xilinx Zynq. For that reason, a dedicated SDK from this vendor (Xilinx SDK), based on Eclipse, is used for throughout the lab.

2.4.1 Stand-alone

In the stand-alone approach, user software scripts are directly executed on the HCPU, just having the software drivers between the hardware and the software scripts. This approach is very useful

when dealing with **single-threaded** and **real time** systems because simplifies the implementation and facilitates time determinism. The different tiers of the stand-alone software approach are illustrated in Figure 13.

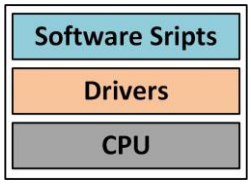


Figure 13: *Different tiers of the stand-alone software approach*

Before start writing the operational software, it is very important to verify the integrity of the the different hardware components. The stand-alone software approach comes in very handy for that. As first step in the software development you will write a code in python for reading the data from the FE board. When an event of interest is detected (the value o the raw data crosses the predefined threshold), the software will change the colour of an on-board RGB LED.

2.4.2 Embedded Operating System

The use of an embedded operating system (EOS) is required (or highly recommended) when implementing systems **running several software processes in parallel**. In these cases, the integrated arbitration capabilities of the EOS will handle the execution of these processes without requiring interaction from the developer, although at the cost of a more complex software implementation. Moreover, the Real-Time Operating Systems (RTOS), a specific type of EOS, are also capable of executing the processes with a deterministic latency. The different tiers of the EOS approach are illustrated in Figure 14.

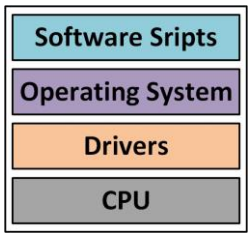


Figure 14: *Different tiers of the EOS approach*

Once the hardware components of our emulated HEP experiment have been tested, it is time to implement the operational software. In this case, we need a way to read the data from the Front-End while board transferring data through Ethernet to the computers farm (just one laptop in this case ;b). For that reason you need to add an operating system and dedicated software scripts to accomplish this tasks. Besides it is also necessary the software scripts for analysing and plotting the data on the user's laptop. Most likely you are thinking how are you going to implement all that in such a limited amount of time... do not worry. You will get some help from the tutor.

3 LAB EXERCISES

3.1 Hardware assembly

The hardware connection of the SoC FPGA lab is illustrated in Figure 15.

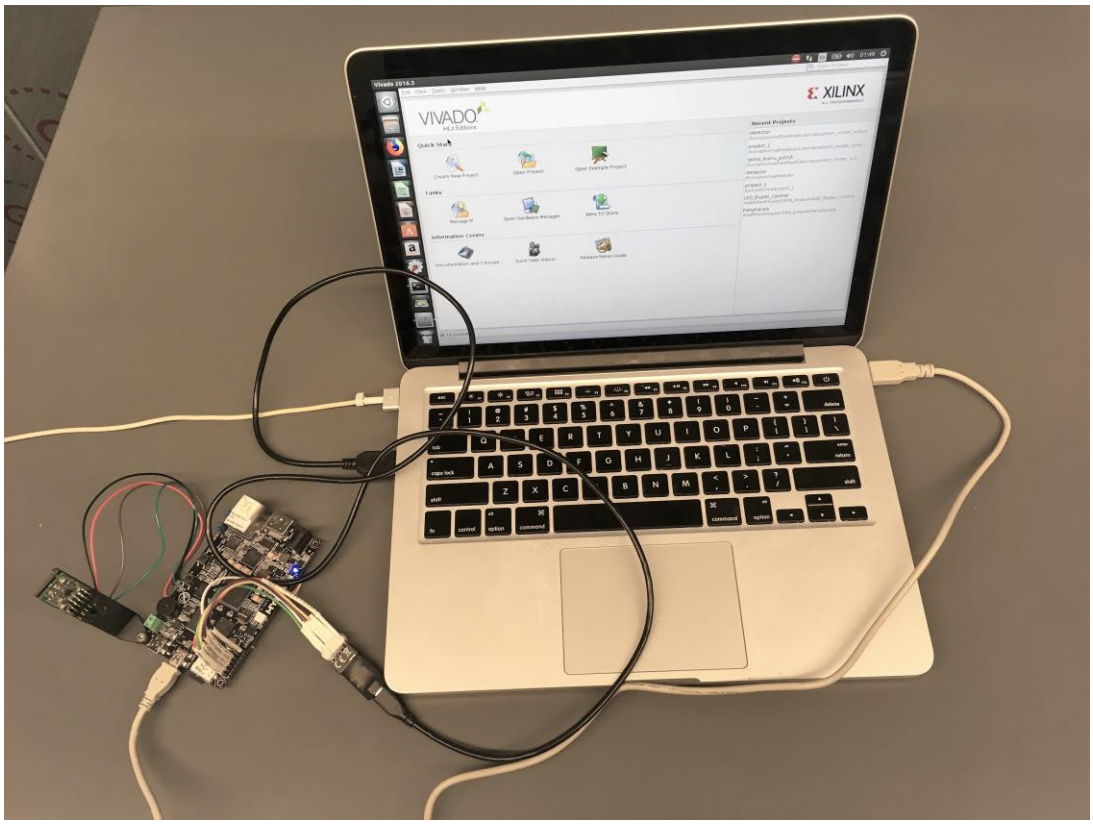


Figure 15: *SoC FPGA lab hardware*

- Ensure that your board is powered on by connecting the Mini-USB to USB cable from computer to the USB_UART mini-USB socket. This connectivity will also allow us transmit and receive data through serial communication (See Figure 16).

Figure 16: *USB_UART Mini-USB cable*

- Connect the programmer on the JTAG port. Make sure the connector matches the pinout from Figure 17.

3.2 GateWare development

The GateWare for this lab is developed using **Vivado**, the vendor specific EDA software from Xilinx.

3.2.1 Project setup

1. Open an Ubuntu terminal(CTRL-T) and type the following to launch Vivado Design Suite.

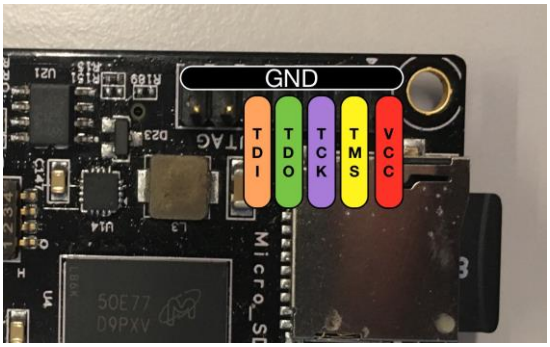


Figure 17: *JTAG pinout*

```
vivado&
```

2. Create a new project and call it **isotdaq_soc_fpga_lab**.

   Click **Next** and tick **RTL Project**. Next, you will be asked to add sources to your design. For the moment, we will keep the project empty. However, we need to specify **VERILOG** for both *Target Language* and *Simulator Language*.
   Press **Next**, and you will be asked about Existing IP and Constrains files. Keep them both empty.

3. Choose the hardware platform where we will run the applications. The development board is entitled MYS-7Z010-C-S Z-turn (see Figure 18). Press **Next** and then **Finish**.

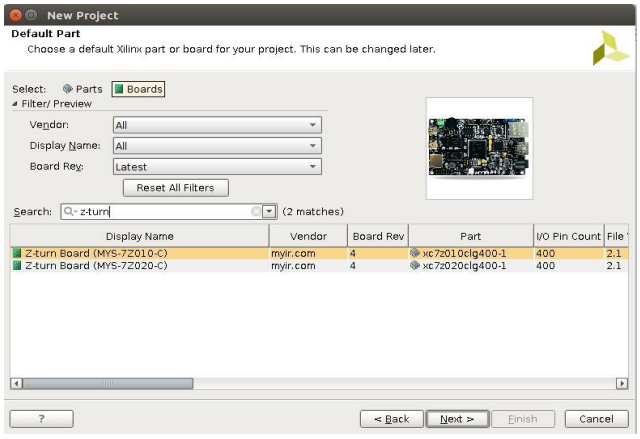


Figure 18: *Hardware Selection*

The starting page of the project should be like in Figure 19.

3.2.2 Block design for SOC

1. Create the Block Design

   On the left of the window, in the **Flow Navigator**, choose **Create Block Design** and call it **detector**.

2. Define SoC with the wizard

   Press **Add IP** and choose **Zynq7 Processing System**. Click **Run Block Automation** to auto-configure it.

   Double click on it to open the wizard.

3. Discuss with your tutor the structure of the System on Chip. We will leave most of the settings as default, apart from several.

   • Click on PS-PL Configuration -> HP Slave AXI Interface and De-select S AXI HPO INTERFACE (see Figure 20 (left))

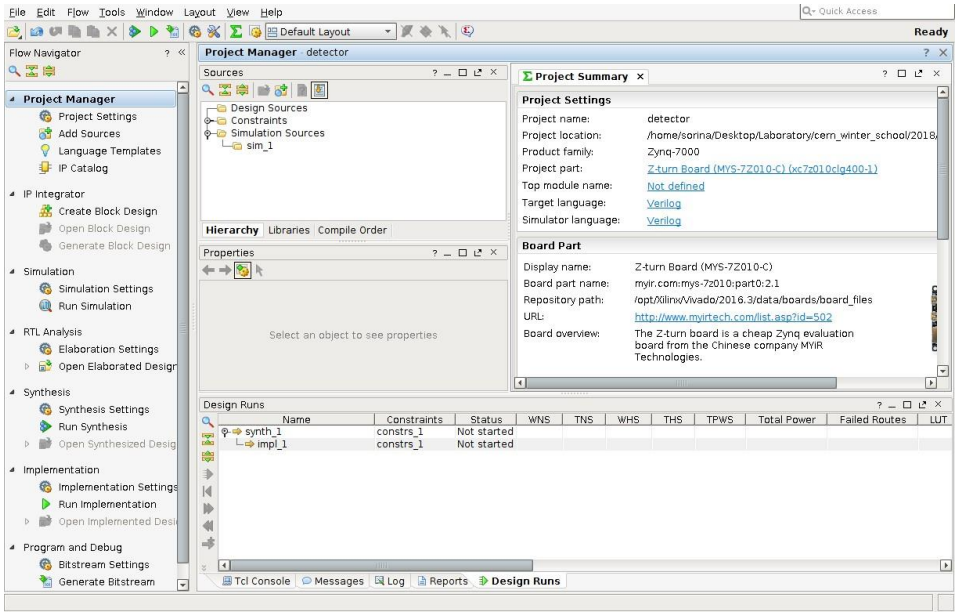


Figure 19: *Initial Project Structure*

• Click on Peripherals I/O Pins and leave ticked only the peripherals which we are using (see Figure 2 (centre)):

– UART1 - for communication with the laptop

– I2C1 - for interface with the Light Sensor

• Clock Configuration. Select only FCLK_CLK0 and change its value to 50 MHz. (see Figure 20 (right))

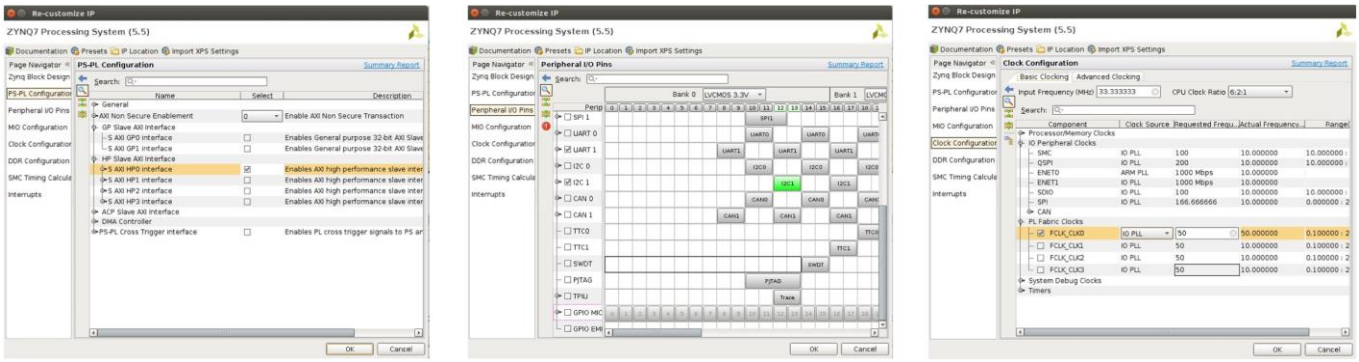


Figure 20: *Zynq Wizard (left) PS-PL Configuration (centre) Peripherals Timing (right)*

3.2.3 Block design for FPGA Fabric

1. Add the PWM IP module to the project

   Please see the details of the PWM module in Appendix A
   The PWM IP was already implemented for you, but you need to add it to the library. Click on the **IP settings** icon . Choose **Repository Manager** from the upper tabs. Click on + symbol and add the content found at the following path relative to your main folder:
   /ip/pwm_verilog_1.0
   Now the PWM folder is added to your project so you can choose it from the IP Library. Press **Add IP** and type "pwm".

2. Let's dig into how the PWM Peripheral was designed

   Right click on the **pwm_verilog_v1 IP** and select **Edit in IP Packager**. Another project (see Figure 21) that contains the files written by the tutors for this IP will open. In the Source part, you will see an hierarchical design in Verilog. The top level is called *pwm_verilog_v1_0.v* and contains an instantiation of the *pwm_verilog_v1_0_S00_AXI.vhd* where the logic is implemented. If you open *pwm_verilog_v1_0_S00_AXI.v*, you
   will see the implementation of PWM code outlined by the following comments:

   -- User to add parameters/ports/logic here
   ...
   -- User parameters/ports/logic ends

Discuss the code with the tutor.

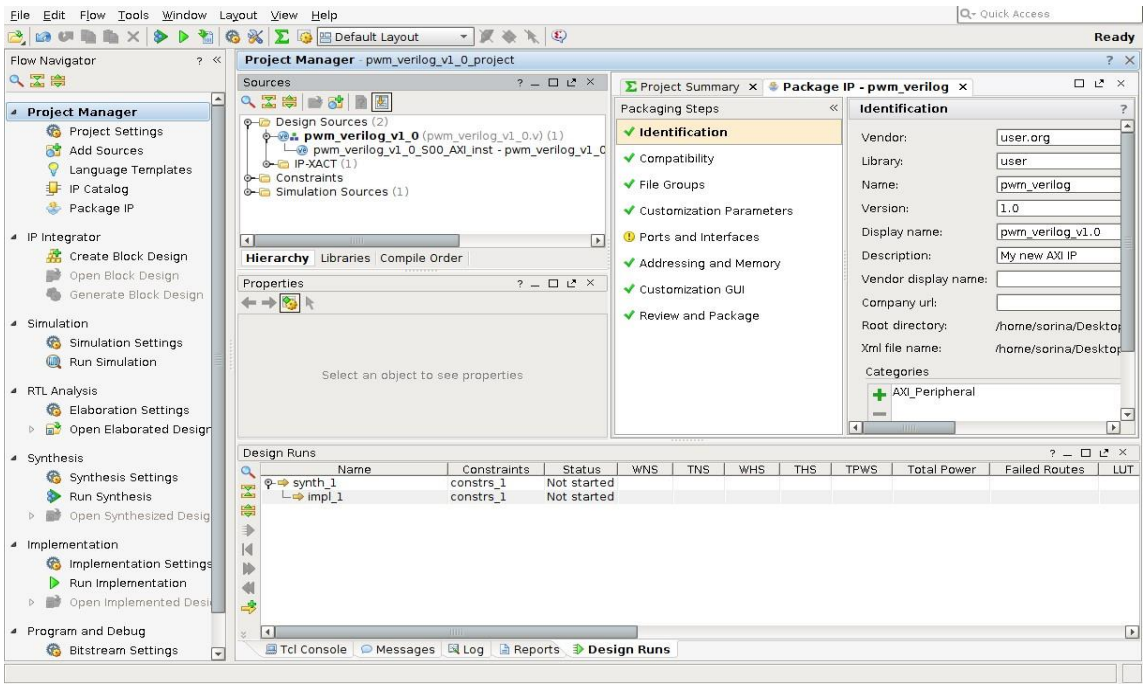


Figure 21: *PWM IP Project Structure*

3. Connect the PWM controller HDL module to the SoC

We need to create PWM outputs, in order to connect them to our SoC. Click on each PWM output and press CTRL-K or **Create Port** from the Menu (see Figure 22). Do so for all the 3 ports.

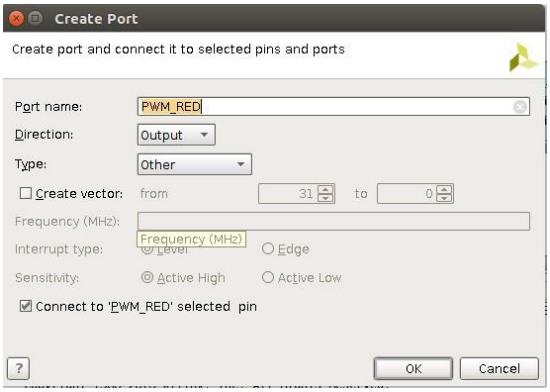


Figure 22: *Port Creation for each PWM output*

Right now, both systems should look like in Figure 23.
Last step is to choose **Run Connection Automation**, let the settings as default, press OK and let the Designer Assistance to do this work on your behalf. The window should now look similar to Figure 24. However, the Design Assistant is placing the blocks in a non-very-organised way. For a better view, we advise you to click on

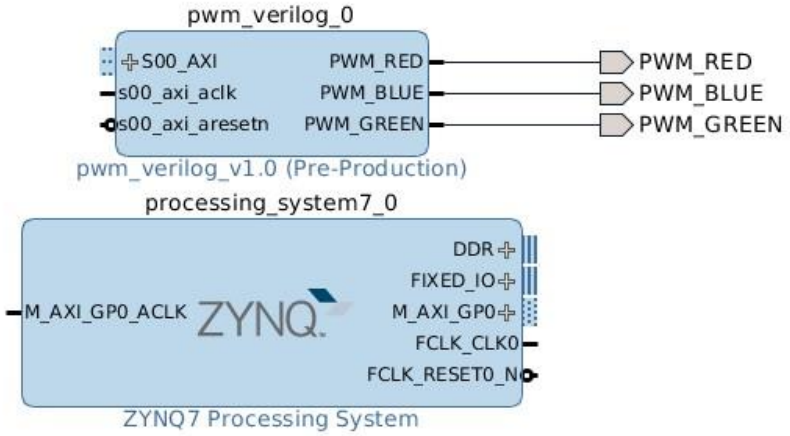


Figure 23: *Both IPs*

the Regenerate Design button in the right Menu. Looks better, right?

Discuss the block diagram with your tutor.

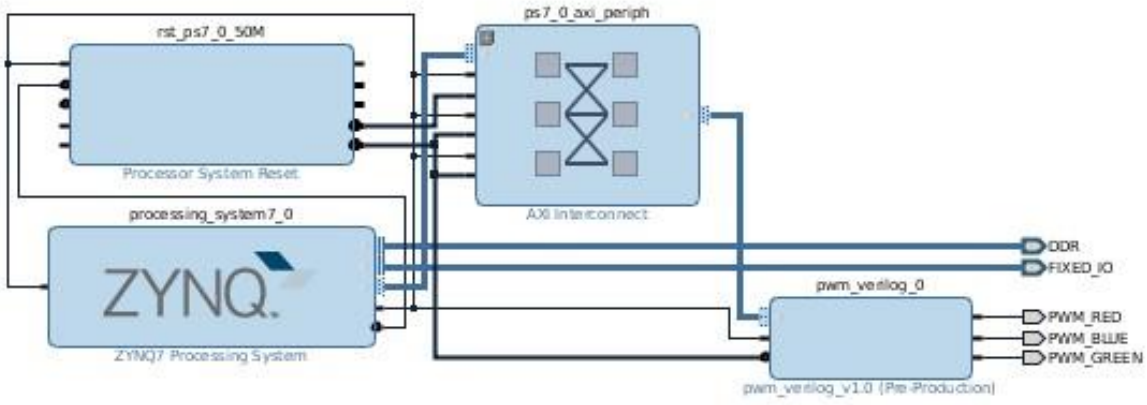


Figure 24: *Block Diagram of our System*

4. Create Wrapper

From the block design, we need to create the Verilog code. Therefore, in the Project Sources - Design Sources, you will right click on the **detector.bd** and select *Create HDL Wrapper*. Let the default option and press OK. Now the VHDL Wrapper should have been created. Verify that the signals PWM_RED, PWM_BLUE and PWM_GREEN appear instantiated (see Figure 25).

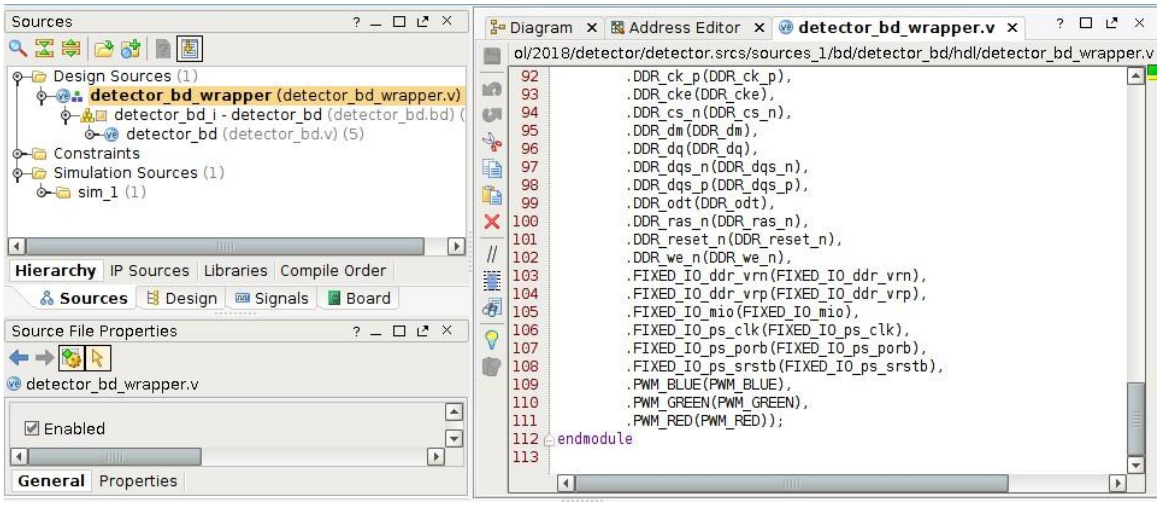
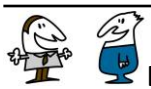


Figure 25: *Wrapper with the PWM outputs instantiated*

3.2.4 Synthesis, implementation & static timing analysis



Discuss with your tutor the differences between Synthesis, Implementation and Bitstream generation for an FPGA.

1. Create Constraint File

   The Constraint File is used in the implementation phase. After the synthesis will be run, and the available top-level nets are "known", they will be matched to the "real" pins.
   Right click on the Constraints directory and click **Add Source** (Add or Create Constraints). The constraints file are present in */constraints/* directory. Open it and check it together with the tutor

2. Generate the bitstream

   The last step in FPGA design is the generation of the bitstream. To generate the bitstream, look into the Flow Manager on the left of the screen and press **Generate Bitstream (Figure 27)**. You will need to wait a bit longer until the bitstream is created. Check for errors in the process and solve them.

3. Static timing analysis

   Please do not forget to check the Static Timing Analysis report and verify that all constraints have been met (see Figure 26)

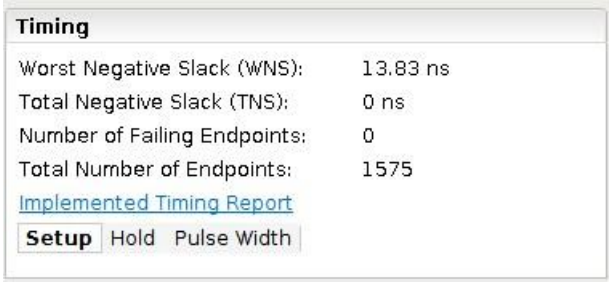


Figure 26: *Static Timing Analysis report*

3.2.5 Export the hardware

Now, that the bitstream was successfully created, we need to export the hardware such as we can use it together with the ARM dual-core microcontroller.

To export the hardware, press **File** > **Edit** > **Export** > **Export Hardware**. Tick *Include bitstream* like in Figure 2 (left).

Figure 27: *Export Hardware Window (left) Generate Bitstream in Flow Manager (right)*

3.3 Software development

The Software for this lab is developed using **Xilinx SDK**, the vendor specific EDA software from Xilinx.

3.3.1 Stand-alone

1. Create the BSP

   In Vivado, press **File** > **Launch SDK**. Now, the hardware shall be automatically imported in Xilinx SDK and you can create a software application.
   From now on, we will create the link between the FPGA and the dual-core ARM microcontroller. In our case, the
   ARM microcontroller is used to read the detector sensor and send signals to the FPGA fabric that further controls an RGB LED.

2. Create a New Application Project

   Select **File -> New -> Application Project**. Select Empty Application and call it **detector_controller**.

3. Add the Cpp file

   Some files were already written to make your life easier, such as the driver to read the light detector through I2C.
   In the **Project Explorer**, go to folder *detector_controller/src* and add the source files available in **sw** folder.

4. Modify the Cpp file

   Open the **main.c**. Read the instructions and solve the simple state machine.

```
/*
 * i2c_getDataDetector(IIC_DEVICE_ID) returns the value
 * read by the sensor as an integer
 *
 * Xil_Out32(COLOR, brightness) sets the colour of the LED
 * from 0 to MAX_BRIGHTNESS
 *
 * xil_printf("%d", value) prints an integer using UART*/

/* Write your code here
 * Implement a state machine that turns on the RED LED
 * when the value exceeded certain threshold
 *
 * Otherwise, make the BLUE LED's brightness follow the
   sensor * readout (e.g. brightness = x*sensor_value
 *
 * Print the sensor_value
 */
```

5. Execute the code on the SoC FPGA

   Ask your tutor for help in running the application and programming both the FPGA and the micro-controller. 6. Verify the results

3.3.2 Embedded Operating System

The implementation of an EOS is a complex task. For that reason, SoC FPGA vendors provide pre-implemented EOS (typically Embedded Linux) which just need to be loaded on the SoC FPGA. In this lab, you are going to use the Embedded Ubuntu Linux provided by MYIR for their devkit.

1. Insert the SD card with the file system on the slot of the Z-turn
2. Find Jumper 1 and Jumper 2 (JP1 and JP2) and make sure they are in the following order for loading the embedded Linux from the SD card:

   JP1
   OFF
   JP2
   ON

3. Execute the program

To run the web server, you need to type in the emulator the following:

```
bokeh serve --host 192.168.2.2:5006 read_event.py
```

Then, in your host computer, open a web browser and type the IP, as seen in Figure 28.

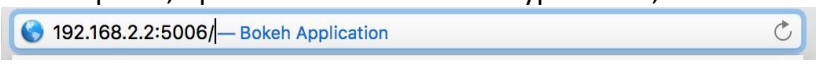


Figure 28: Host address

4. Check the webserver and verify the results

Then, the plot will be displayed interactively (see Figure 29). Have fun!!
Now you can play with the setup and discuss about it with the tutor.

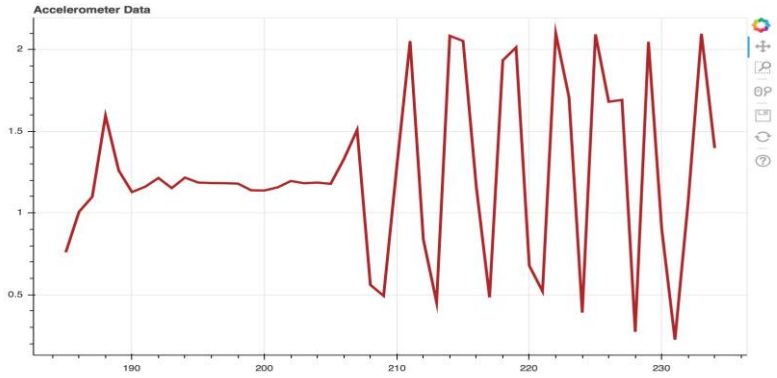


Figure 29: *Interactive plot*

Appendices

A) PULSE WIDTH MODULATION (PWM) SLAVE PERIPHERAL FOR LED CONTROL

The FPGA runs at a clock frequency (for this application, we will choose 50 MHz). We can create a PWM waveform from this clock frequency, by counting a number of clock cycles. There are two important parameters that characterise the PWM waveform: **the duty cycle** and the **PWM period**. The PWM period is set by "waiting/counting" a number of clock cycles, while the duty cycle tells us, in one period, how many clock cycles the PWM signal is low. In Figure 30, you can see a basic PWM waveform.

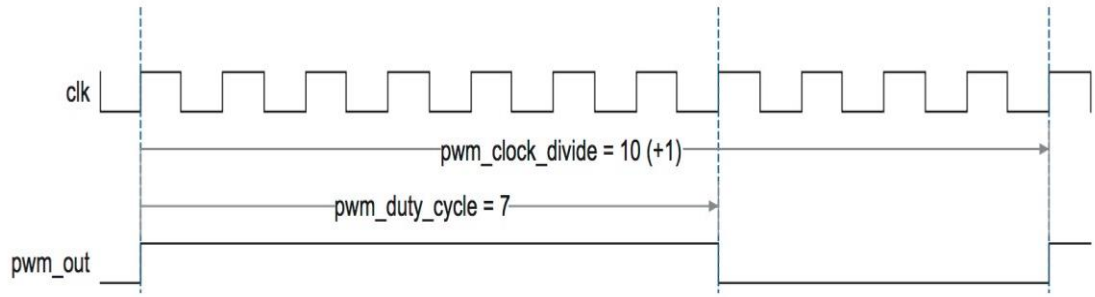


Figure 30: *PWM waveform (Source: Application Note 333, Altera)*

Now let's see how we are going to use these two simple parameters to make an RGB LED change its colour.

RGB LED

In the case of RGB LED, we need 3 PWMs signals (one per each colour), such that the brightness of each of the three LEDs can be controlled independently.

How do we choose the PWM period? The driving frequency of the PWM should be fast enough to avoid the flicker effect. A normal human being sees this effect until up to 100 to 150 Hz, so a higher frequency should be better to avoid this effect. For this exercise, we will use a frequency of 1.5 kHz. Let's make some calculations.

The main clock frequency of the FPGA is 50 MHz. We want to use a 1.5 kHz frequency. How much do we need to count?

$$counter = \frac{50000000}{15000} = 30000 \tag{1}$$

How do we choose the PWM duty cycle? The brightness of the LEDs is controlled by the duty cycle. This is up to you to do experiments. You can choose basically any duty cycle in the range of 0% to 100%.

B) ACKNOWLEDGES

- Markus Joos (CERN) & other organisers of ISOTDAQ
- Ton Damen (NIKHEF)
- Peter Jansweijer (NIKHEF)
- The other members of the ISOTDAQ SoC FPGA Lab Team:
  - Elena-Sorina Lupu (EPFL/Caltech)
  - Patryk Oleniuk (Hyperloop One)
  - Andrea Borga (NIKHEF)
  - Manoel Barros Marin (CERN)

14. *Lab 14: Introduction to GPU programming*

V.1.0 - Gianluca Lamanna (gianluca.lamanna@cern.ch)

Overview

This lab aims at introducing the main concepts of GPU programming. After simple exercises to become familiar with the video card and software environment, we will study the influence of the GPU (Graphics Processing Units) architecture on code optimization. The students are invited to use the pieces of code provided in this lab book as a source of inspiration and to write is own code.

Introduction

Historically most of the code, written for standard processors, runs sequentially. To speed-up the software execution most of the developers relied on the continuous increase in hardware system performances with small or null code optimizations. In the last years, the processors' speed is gone towards a kind of saturation due to integration limits and power constraints. Parallel processors, such GPUs, have shown to overcome the limits of single-core processors, thanks to a different architecture. The GPUs are designed to have many-threads running in parallel with the purpose to increase the computing throughput for parallelizable problems. In contrast, the CPUs (Central Processing Units) are designed with a multi-core architecture, in which each core is optimized to maximize the execution speed of sequential programs. Nowadays commercial GPUs are designed to have an order of 10 TFlops (Floating point operations per second) computing throughput while first class multi-core CPUs rarely exceed 1 TFlops.

Fig.1

One could ask why there is such a large difference in performance between many-threads GPUs and general-purpose multi-core CPUs. As shown in figure 1 the main reason is due to the very different structure between the two types of processors. In CPU most of the “transistors” are devoted to control logic and large cache to reduce instructions and data access latencies, while in GPU most of the space is used for real computing. Another important reason is the difference in memory bandwidth since several applications are limited by the rate at which the data are delivered to the processor for computing: in the GPU the memory bandwidth is at least x5 higher than in CPU. This is due to the fact that in standard CPU the memory management is conditioned by legacy operating systems, applications and I/O, while in GPU only data movement from memory to computing units occurs.

From a software point of view, to exploit the power for these GPUs the application is expected to be written with a large number of parallel threads. The hardware takes advantage of the large number of threads to find work to do when some of them are waiting for long-latency memory accesses or arithmetic operations (throughput oriented). On the other hand, the CPUs are designed to minimize the execution latency of a single thread (latency-oriented). For programs that need one or very few threads, CPUs with lower operation latencies can achieve much higher performance than GPUs. When a program has a large number of threads, GPUs with higher execution throughput can achieve

much higher performance than CPUs. It should be clear now that GPUs are designed as parallel, throughput-oriented computing engines and they will not perform well on some tasks on which CPUs are designed to perform well.
A heterogeneous computing model, in which both serial and parallel processors are used at the same time, is able to speed-up several computational problems. This kind of philosophy is used, for instance in the CUDA model on NVIDIA video cards. Until 2006, graphics chips were very difficult to use because programmers had to use the equivalent of graphics API (Application Programming Interface) functions to access the processing units. Starting from 2007 CUDA has been introduced to allow developers to direct access to computing resources, in an environment integrated with standard programming languages.
The CUDA model is not the only way to exploit GPU power for computing. Nowadays several other possibilities are present on the market. OpenCL is a low-level API, equivalent to CUDA but suitable for several types processors (including FPGAs) and GPU vendors. Directives-based programming model (such as OpenACC) and GPU libraries (such as Thrust) allow to invoke GPU computing during serial code execution, to parallelize specific task, in a user transparent way. For educational reasons, in this lab, we will concentrate on CUDA, as it allows to better understand the GPU programming philosophy and the interplay with the hardware architecture.

Exercise 1: Discover the GPU performances

The CUDA developer SDK provides examples with source code and utilities to get started writing software with CUDA. In the CUDA samples directory, you can find several programs ranging from basic to advanced level. In particular, in the directory "`1_Utilities`" two simple applications are ready to be compiled (using `make` for the moment): `deviceQuery` and `bandwidthTest`. Both of them provide information about the system, type, and characteristics of GPU (if correctly recognized).
Try to run them and write down the features that seem most important to you.

Exercise 2: Parallel "Hello World!"

The code to program the GPU is substantially subdivided into two parts: **HOST** and **DEVICE**. The HOST part is executed in the PC while the DEVICE part (called KERNEL) runs on GPU. In the HOST part, together with the serial part of the program, a GPU related part must be present, such as data preparation and instructions to copy data on device (the GPU). Let's try to write a "Hello World!" program (see *Snippet 1*)

```
#include <stdio.h>
__global__ void mykernel(void) {
  printf("Hello World from GPU! (block: %d thread: %d)\n"   ,blockIdx.x,threadIdx.x);
}
int main(void) {
  mykernel <<<1,5>>>();
  cudaDeviceSynchronize();
  printf("Hello World from Host!\n");
  return 0;
}
```

Snippet 1

In this simple program, there are two key points of GPU programming: *kernel definition* and *kernel launch*. The kernel is essentially a C function defined through a qualifier. The qualifier (`__global__`) tells to the compiler that the function is callable from host to be executed on the device. The syntax to launch the kernel in CUDA is: `mykernel<<<DimGrid, DimBlock>>>(arg)`, where `mykernel` is the name of the kernel defined above with, eventually, the arguments of the function between the brackets (), while the variables `DimGrid`

and `DimBlock` define how to use the GPU resources at run time. We will discuss this point later, for the moment we just note the kernels have access to two built-in variables (`threadIdx`, `blockIdx`) that allow threads to distinguish among themselves. The CUDA SDK toolkit provides tools for compilation and debugging. The C compiler is "`nvcc`", very similar to cc/gcc in several aspects. You can compile the "Hello world" code just with:

```
nvcc file_name.cu –o HelloWorld –arch=compute_50 –code=sm_50
```

Try to compile and run the code: Which is the output? Try to play a little bit with the parameters in the kernel launch (1 and 5 in this example), what happens?

Exercise 3: Vector Add and GPU optimization

In order to illustrate the structure of parallel computing, we start with a somewhat more complex example. Suppose you have two very big vectors (let's say with 1048576 elements) to add up. The task is quite simple since each element in the vector result is just the sum of the corresponding elements in primary vectors:

c[i]=a[i]+b[i]

but must be repeated a huge number of times. This problem is particularly suitable for parallelization: each element in the sum is independent of the other. As a starting point let's try to write a serial version of the code as suggested in snippet 2

```
#include <stdio.h>
#define N 1048576
void RandomVector(int *a, int nn){
  for (int i=0;i<nn;i++) {
    a[i]=rand()%100+1;
      }
 }
//serial sum
void VecAddSerial(int *a, int *b, int *c){
  for (int i=0;i<N;i++){
    c[i] = a[i]+b[i];
  }
}
int main(void) {
  int *h_a, *h_b, *h_c;
  int size = N*sizeof(int);
  float time;
  cudaEvent_t start,stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  //Alloc in Host (and filling)
  h_a = (int *)malloc(size);
  h_b = (int *)malloc(size);
  h_c = (int *)malloc(size);
  RandomVector(h_a,N);
  RandomVector(h_b,N);
```

Snippet 2.a

```
//start time
  cudaEventRecord(start);
  //Launch Serial Sum on CPU
  VecAddSerial(h_a,h_b,h_c);
 //stop time
  cudaEventRecord(stop);
  cudaEventSynchronize(stop);
  cudaEventElapsedTime(&time, start, stop);
 //Print Result
 //  for(int i=0;i<N;i++){
  //    printf ("%d) h_a:%d h_b:%d h_c:%d\n",
  //    i,h_a[i],h_b[i],h_c[i]);
  // }
  //print time
   printf("Time: %3.5f ms\n",time);
 //Cleanup
  free(h_a);
  free(h_b);
  free(h_c);
  return(0);
}
```

Snippet 2.b

In this case, the vector sum is done in a standard C function (`VecAddSerial`). Some CUDA function is added to measure the execution time with the same mechanism we will use in the GPU version. For this reason, this code must be compiled with `nvcc`, even if it will run only on the host. Let's try to modify this code to exploit the GPU computing power for the vector sum. Before starting we discuss a moment as the parallelism works in the GPU. After the kernel launch, the CUDA runtime

system generates a *grid of threads* organized in a two-level hierarchy. Each grid is organized into an array of *thread blocks*, which will be referred to as *blocks*. All blocks of a grid are of the same size; each block can contain up to 1,024 threads.

Go back to exercise 1 to discover the maximum dimension of grid and blocks in the GPU we are using.

```
<skip>
//kernel
__global__ void VecAddGpu(int *a, int *b, int *c){
  c[blockIdx.x] = a[blockIdx.x]+b[blockIdx.x];
}
<skip>
//Alloc in Device
  cudaMalloc((void **)&d_a, size);
  cudaMalloc((void **)&d_b, size);
  cudaMalloc((void **)&d_c, size);
//Copy input vectors form host to device
  cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);
<skip>
//Launch Kernel  on GPU
  VecAddGpu<<<N,1>>>(d_a,d_b,d_c);
  cudaDeviceSynchronize();
<skip>

//Copy back the results
  cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);
<skip>
//Cleanup
  free(h_a);
  free(h_b);
  free(h_c);
  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);
```

Snippet 3

The total number of threads in each thread block is specified by the host code when a kernel is launched. Each thread is identified by the thread number within the block (`threadIdx.x` we saw in exercise 2) and the block number within the grid (`blockIdx.x`). Another important aspect in preparing the GPU modification of the serial code is that the user must provide the data to the GPU. This means that, before the kernel launch, the data must be explicitly copied in the GPU. When the computation on GPU ends, the results must be copied back from device to host by the user. This is done using the PCI express bus, slower with respect to the processor memory direct connection. One can imagine that the movement of data from host to device (and back) is one of the main contributions to the total latency in heterogeneous application. This overhead in time execution can be eventually "masked" with the gain in the computational part. Now, try to modify the serial code by using the suggestions in snippet 3. The `cudaMalloc` function is used to allocate memory on the device, while data from the host to the device are copied by using the function `cudaMemcpy`. After the kernel launch, another `cudaMemcpy` will copy back the results.

Try to compile and run.

The result is not what we expected. Although we are using a GPU with 1 TFlops and our problem is very parallelizable (and simple), the execution time is worse than the serial code version!

Take a minute to have a close look at the code.

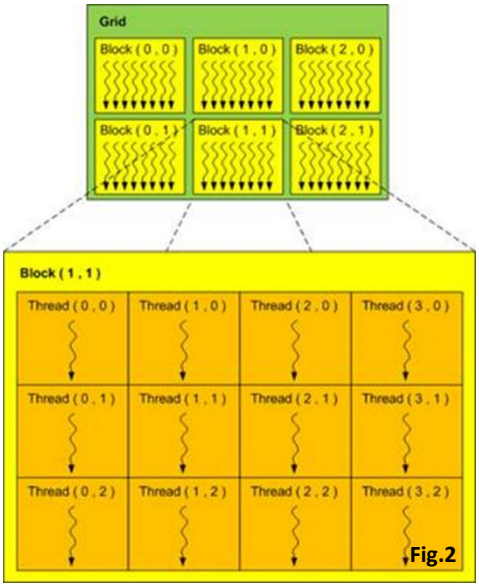


Fig.2

The reason for this failure is that we are using the GPU in a very inefficient way. In fig. 2 the hierarchy of the threads described above is shown. Roughly speaking each block is executed in a multi-processor (this is not completely true), that, in hardware, is a group of single CUDA cores. The GTX750, we are using, has 512 computing cores grouped in just 4 multi-processors. In the kernel launch in the snippet above we define the structure of the parallelism with `VecAddGpu<<<N,1>>>`, this means that we asked for 1048576 blocks with one thread each. Since the number of multi-processors is limited, only 4 threads are executed concurrently in the GPU. In each block, one thread is working while 127 threads are in idle, with a lot of blocks waiting for their turn to be executed.
Then let's try to change the point of view and rewrite the code (Snippet 4) to fully exploit the threads.

```
<skip>
//Launch Kernel  on GPU
  VecAddGpu<<<1,N>>>(d_a,d_b,d_c);
  cudaDeviceSynchronize();
```

Snippet 4

In this case, we are using one single block and a huge number of threads.

Let's try to compile and run.

Now the execution time is very very small. Try to compute the speed-up factor with respect to the serial code. As a rule of thumb a well-written code can give a factor of 100, but no more than that. But here it is even bigger. This is quite suspicious! **Try to print out the result of the kernel execution (after the copy on the host) to figure out what is happening.** Probably you can notice that there is something wrong. Another possibility to spot the error is to use the error managing system of CUDA. Try to add:

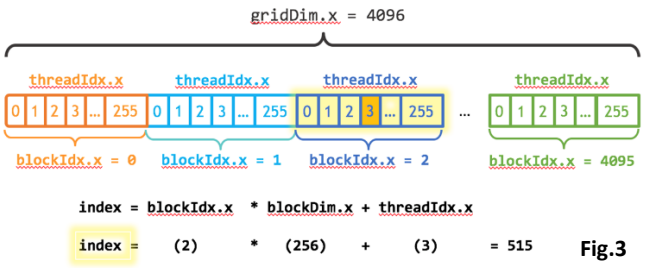


Fig.3

```
cudaError_t KernelError=cudaGetLastError();
printf("Error %s\n",cudaGetErrorString(KernelError));
```

just after the kernel call.

What message do you get? Another possibility is to use CUDA debugger. It is very similar to the c gdb debugger with additional support to investigate the behavior of single threads. Try with:

`cuda-gdb <executable>` (remind to compile with the flag –g).

As suggested by debugging tools we are running the GPU out of his resources. Try to have a look (exercise 1) at the maximum number of threads per block allowed for the board we are using. The winning strategy is to use both blocks and threads to exploit the maximum number of resources available. In doing that we need to change two things (Snippet 5): first in the kernel call we have

```
<skip>
#define THREADS_PER_BLOCK 128

<skip>
//kernel
__global__ void VecAddGpu(int *a, int *b, int *c){
  int index = threadIdx.x + blockIdx.x*blockDim.x;
  c[index] = a[index]+b[index];
}
<skip>
//Launch Kernel  on GPU
VecAddGpu<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a,d_b,d_c);
cudaDeviceSynchronize();
```

Snippet 5

explicitly configured the GPU architecture to have the correct number of blocks in the grid and the correct number of threads for each block, second we have to change the kernel in order to be sure that each of the N threads running operates on different vector elements (fig.3 is just a generic example on how to define the index in the kernel).

Try to compile and run the code. How much is the execution time now?

Exercise 4: Memory management

In the previous exercise, we learned how to organize the GPU resources to use a huge number of threads, concurrently. Although we saw how to optimize threads and blocks to get better performances, this is not the end of history. The CUDA kernels that we have learned so far will likely achieve only a tiny fraction of the potential speed of the underlying hardware. The poor performance is due to the fact that in GPU programming it is responsibility of the user to organize data in the proper way inside the memory. Let's try to understand better this point with an example. In our previous example for each operation (vector elements sum), there are 2 accesses to the memory to extract the sum addenda. Thus the ratio of floating point calculation to memory access is 1:2. We refer to this ratio as *compute-to-global-memory-access ratio*, defined as the number of floating-point calculations performed for each access to the memory. For our GTX 750 the memory bandwidth is (exercise 1) 80 GB/s, this means that with 4 bytes for each single-precision floating point operation one can expect to load 80/(2*4) =10 Giga single-precision operands per second. Assuming a compute-to-global-memory-access of 0.5 the maximum number of operations per

seconds is 10 GFlops that is quite far from the nominal computing power of 1 TFlops of our GPU. The programs in which execution speed is limited by the memory access throughput are called *memory bound* programs. CUDA provides a number of additional resources and methods for accessing memory that can remove the majority of traffic to the memory. We will briefly discuss the use of CUDA memories and how to increase the timing performances, through a simple example. In GPU there are several types of memories that can be accessed from different levels of the parallelism hierarchy. In fig.4 you can see the structure of the two main memories: the global and the shared memory. The global memory is accessible by each multi-processor and each thread running in single cores of a multi-processor. It is physically implemented in DDR and is connected through the PCI express bus to receive data coming from the host: when you perform a standard cudaMemcpy operation you are copying data in this memory, that is, usually, quite big (2GB in GTX 750). On the other hand, the scope of the shared memory is limited to a single block. Only the threads running in that block can use the same shared memory, that is implemented on the GPU chip. The quantity of shared memory per block is very limited (49kB on GTX 750) but the speed is very high (at a level of 1 TB/s). This structure of memories is exploited to increase the peak performance in the GPU in several problems.

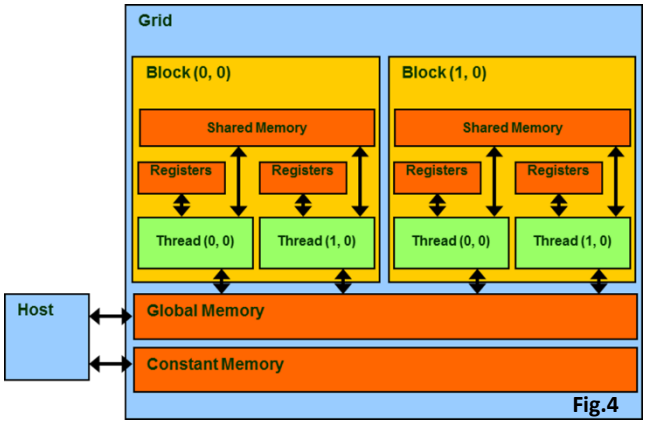


Fig.4

Let's try to understand better how to exploit the shared memory with a new example: the matrix multiplication. Take a look at the snippet presented on the next page (Snippet 6). The structure is similar to the one analyzed above, even if the code is organized in a different way. **Try to understand what the code is intended to do by yourself.** In this version of the code data are copied in the global memory from host and then each thread read the data directly from the global memory. A simple improvement in memory managing is achieved by observing that the global memory access is coalesced. This means that when all threads of a warp (a group of 32 threads in the same block) execute a load instruction, if all accessed locations fall into the same burst section, only one DRAM request will be made and the access is fully coalesced, as described in fig.5.

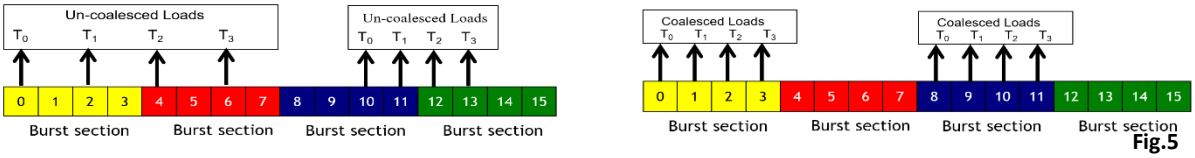


Fig.5

<MatrixMultiplication_global.h>

```
#include <stdio.h>
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row *
M.width + col)
typedef struct {
int width;
int height;
float* elements;
} Matrix;
// Thread block size
#define BLOCK_SIZE 16
__global__ void MatMulKernel(const Matrix,
const Matrix, Matrix);
```

<MatrixMultiplication_global.cu>

```
#include "MatrixMultiplication_global.h"
// Matrix multiplication - Host code
void MatMul(const Matrix A, const Matrix
B, Matrix C) {
  // Load A and B to device memory
Matrix d_A;
d_A.width = A.width;
d_A.height = A.height;
size_t size = A.width * A.height *
sizeof(float);
cudaError_t err =
cudaMalloc(&d_A.elements, size);
 printf("CUDA malloc A:
%s\n",cudaGetErrorString(err));
 err = cudaMemcpy(d_A.elements,
A.elements, size, cudaMemcpyHostToDevice);
printf("Copy A to device:
%s\n",cudaGetErrorString(err));
Matrix d_B;
d_B.width = B.width;
d_B.height = B.height;
size = B.width * B.height * sizeof(float);
err = cudaMalloc(&d_B.elements, size);
printf("CUDA malloc B:
%s\n",cudaGetErrorString(err));
err = cudaMemcpy(d_B.elements, B.elements,
size, cudaMemcpyHostToDevice);
printf("Copy B to device:
%s\n",cudaGetErrorString(err));
  // Allocate C in device memory
Matrix d_C;
d_C.width = C.width;
d_C.height = C.height;
size = C.width * C.height * sizeof(float);
err = cudaMalloc(&d_C.elements, size);
printf("CUDA malloc C:
%s\n",cudaGetErrorString(err));
  // Invoke kernel
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid((B.width + dimBlock.x - 1) /
dimBlock.x,(A.height + dimBlock.y - 1) /
dimBlock.y);
MatMulKernel<<<dimGrid, dimBlock>>>(d_A,
d_B, d_C);
err = cudaThreadSynchronize();
printf("Run kernel: %s\n",
cudaGetErrorString(err));
```

Snippet 6.a

```
// Read C from device memory
err = cudaMemcpy(C.elements, d_C.elements,
size,
cudaMemcpyDeviceToHost);
printf("Copy C off of device:
%s\n",cudaGetErrorString(err));
// Free device memory
cudaFree(d_A.elements);
cudaFree(d_B.elements);
cudaFree(d_C.elements);
}

// Matrix multiplication kernel called by
MatMul()
__global__ void MatMulKernel(Matrix A,
Matrix B, Matrix C) {
 // Each thread computes one element of C
 // by accumulating results into Cvalue
 float Cvalue = 0.0;
int row = blockIdx.y * blockDim.y +
threadIdx.y;
int col = blockIdx.x * blockDim.x +
threadIdx.x;
if(row > A.height || col > B.width) return;
 for (int e = 0; e < A.width; ++e)
Cvalue += (A.elements[row * A.width + e]) *
(B.elements[e * B.width + col]);
C.elements[row * C.width + col] = Cvalue;
}
// Usage: multNoShare a1 a2 b2
int main(int argc, char* argv[]){
Matrix A, B, C;
int a1, a2, b1, b2;
 // Read some values from the commandline
a1 = atoi(argv[1]); /* Height of A */
a2 = atoi(argv[2]); /* Width  of A */
b1 = a2;           /* Height of B */
 b2 = atoi(argv[3]); /* Width  of B */
A.height = a1;
A.width = a2;
A.elements = (float*)malloc(A.width *
A.height * sizeof(float));
B.height = b1;
B.width = b2;
B.elements = (float*)malloc(B.width *
B.height * sizeof(float));
C.height = A.height;
C.width = B.width;
C.elements = (float*)malloc(C.width *
C.height * sizeof(float));
for(int i = 0; i < A.height; i++)
     for(int j = 0; j < A.width; j++)
           A.elements[i*A.width + j] =
(float)(random() % 2);
 for(int i = 0; i < B.height; i++)
      for(int j = 0; j < B.width; j++)
           B.elements[i*B.width + j] =
(float)(random() % 3);
MatMul(A, B, C);

} //end
```

Snippet 6.b

To exploit the coalescence, the memory reading should be organized in a proper way. The next level of improvement is to exploit the shared memory. The general idea is to subdivide the matrix into sub-matrices and to assign each sub-matrix to a thread block. The data are copied once from the global memory to the shared memory of the block and then re-used according to the needs of the matrix multiplication algorithm. The shared memory is defined with the keyword `__shared__` followed by type and dimensions. Also the shared memory, similarly to the global memory, is organized in banks (16 in older GPU, 32 in newer GPU) that can be accessed simultaneously to achieve high memory access. In your working directory, you should have a file `MatrixMultiplication_shared.cu` file, where the implementation with shared memory is done.

Try to take a look to understand how it works and modify the code to try different solutions and to measure the timing (adding the "events" statement properly).

Exercise 5: Fitting rings with GPU

In this last exercise, you can use the code in the directory "`rings`" in your working directory. This is a simple example of possible use of GPU for pattern recognition. The so-called *Crowford algorithm* is used to search for center and radius of a limited number of hits on a circle in, for instance, a Cherenkov Ring counter. You are invited **to try to optimize the code** in order to increase the speed of the total process. A good way to understand what is happening is to use the *profiler* provided in the CUDA toolkit. You can invoke it just with `nvprof` and the name of the executable for the textual version and `nvvp` and the name of the executables for the GUI version (fig. 6).

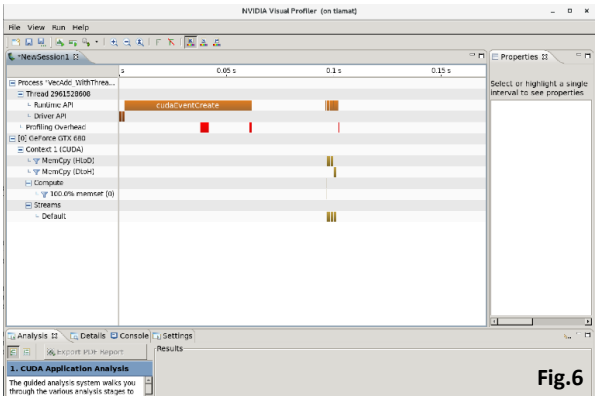


Fig.6

GLOSSARY

GPGPU (General Purpose computing on Graphics Processing Units): is the idea to use GPU for standard computing usually performed on CPU, by exploiting the intrinsic parallelism of graphics processors.

Threads: A thread is a simplified view of how a processor executes a sequential program in modern computers. A thread consists of the code of the program and the values of its variables and data structures. A CUDA program initiates parallel execution by launching kernel functions, which causes the underlying runtime mechanisms to create many threads that process different parts of the data in parallel.

Blocks: Groups of threads.

CUDA core: the smallest arithmetic computing unit in a GPU (a.k.a. single core).

Multi-processors: a group of single cores.

Built-in Variables: Many programming languages have built-in variables. These variables have a special meaning and purpose. The values of these variables are often pre-initialized by the runtime system and can be referenced in the program.

Memory Space: Memory space is a simplified view of how a processor accesses its memory space is usually associated with each running application. The data to be processed by an application and instructions executed for the application are stored in locations in its memory space. Each location typically can accommodate a byte and has an address. Variables that require multiple bytes – 4 bytes for float and 8 bytes for double are stored in consecutive byte locations. The processor gives the starting address (address of the starting byte location) and the number of bytes needed when accessing a data value from the memory space.

SIMD (Single Instruction Multiple Data): It is a computing model in a parallel architecture. It described processing units where the same instructions are executed simultaneously on a different set of data.

SIMT (Single Instruction Multiple Threads): The computing item is the thread that is implemented as a sequence of SIMD operation