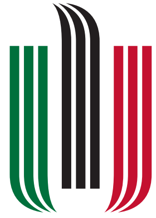


AGH University of Science and Technology
Faculty of Physics and Applied Computer Science

Master thesis

Adrian Matoga

major: **Applied Computer Science**

specialisation: **Computer Techniques in Science and Technology**

Development of data acquisition system for luminosity detector at International Linear Collider

Supervisor: **Dr hab. inż. Marek Idzik**

Kraków, September 2011

Aware of criminal liability for making untrue statements I decleare that the following thesis was written personally by myself and that I did not use any sources but the ones mentioned in the dissertation itself.

Kraków, 20 September 2011

The subject of the master thesis and the internship by Adrian Matoga, student of 5th year major in applied computer science, specialisation in Computer Techniques in Science and Technology

The subject of the master thesis: **Development of data acquisition system for luminosity detector at International Linear Collider**

Supervisor:	Dr hab. inż. Marek Idzik
Reviewer:	Dr inż. Bartosz Mindur
Place of the internship:	WFiIS AGH, Kraków

Programme of the master thesis and the internship

1. First discussion with the supervisor on realization of the thesis.
2. Collecting and studying the references relevant to the thesis topic.
3. The internship:
   - Installing Linux on a Xilinx evaluation board using existing designs.
   - Modifying the installation procedure to suit a different board.
   - Preparation of the internship report.
4. Collecting requirements for the data acquisition firmware.
5. Designing and implementing the firmware.
6. Creating modules for integration with existing desktop data acquisition software package.
7. Testing the developed firmware and measuring its performance.
8. Porting parts of the firmware to Linux.
9. Final analysis of the results obtained, conclusions—discussion with and final approval by the thesis supervisor.
10. Typesetting the thesis.

Dean's office delivery deadline: 30 September 2011

. . . .
(Head of the Department's signature)

. . . .
(Supervisor's signature)

dr inż. Bartosz Mindur
Wydział Fizyki i Informatyki Stosowanej AGH
Katedra Oddziaływań i Detekcji Cząstek

Merytoryczna ocena pracy przez recenzenta:

Tematyka pracy obejmowała szerokie zagadnienie dotyczące systemu zbierania danych, budowanego na potrzeby detektora świetlności powstającego w ramach kolaboracji FCAL. Podstawowym celem pracy było zaprojektowanie i wykonanie oprogramowania sterującego dedykowaną elektroniką odczytową. W ramach projektu miały powstać moduły służące do konfigurowania parametrów pracy systemu, monitorowania jego zachowania oraz zbierania odczytywanych danych. Całość projektu miała zostać wpasowana w odpowiednie ramy tak aby powstałe oprogramowanie było kompatybilne z systemem EUDAQ. Dodatkowo autor w ramach pobocznego celu tej pracy podjął się próby weryfikacji koncepcji wykorzystania systemu operacyjnego Linux do wyżej już wspomnianego systemu zbierania danych.

Na podstawie manuskryptu jasno i wyraźnie widać olbrzymią pracę wykonaną przez autora podczas realizacji tegoż zadania. Powstałe oprogramowanie niewątpliwe jest bardzo dobrze zaprojektowanie niemniej jednak odczuwa się pewien niedosyt jeśli chodzi o opis procesu testowania oprogramowania razem z elektroniką odczytową. Dodatkowo, pomimo iż autor nie zajmował się analizą danych zebranych podczas testów na wiązce i nie było to podstawowym celem jego pracy, to jednak ze względu właśnie na fakt wykorzystania tegoż oprogramowania podczas wspomnianych testów można by oczekiwać, że trochę więcej wyników zostanie jednak w pracy umieszczonych. Nie można jednak nie wspomnieć, iż praca jest zredagowana starannie, tylko z drobnymi i nielicznymi niedociągnięciami oraz dodatkowa należy na duży plus zaliczyć to, iż została napisana w języku angielskim. Dzięki temu powstały manuskrypt może z łatwością służyć jako dokumentacja projektu wykorzystywana przez członków międzynarodowej kolaboracji.
Końcowa ocena pracy przez recenzenta: 5.0

Data: 27.9.2011r   Podpis: ....

Skala ocen: 5.0 – bardzo dobra, 4.5 – plus dobra, 4.0 – dobra, 3.5 – plus dostateczna, 3.0 – dostateczna, 2.0 – niedostateczna

dr hab. inż. Marek Idzik
Wydział Fizyki i Informatyki Stosowanej AGH
Katedra Oddziaływań i Detekcji Cząstek

Merytoryczna ocena pracy przez opiekuna:

Tematem pracy magisterskiej studenta Adriana Matogi jest rozwój systemu akwizycji danych dla detektora świetlności przy przyszłym akceleratorze liniowym International Linear Collider (ILC). Wykonana praca stanowi istotną część działalności badawczo-naukowej Katedry Oddziaływań i Detekcji Cząstek, prowadzonej w celu budowy detektora świetlnosci dla przyszłego akceleratora liniowego ILC.

Zadaniem magistranta był projekt, realizacja, a następnie eksperymentalna weryfikacja systemu akwizycji danych dla prototypowego modułu detektora świetlności. W szczególności miał on, dla istniejącej platformy sprzętowej zawierającej układ programowalny FPGA Xilinx Spartan-3E oraz mikrokontroler Atmel ATxmega, stworzyć oprogramowanie, tak niskiego jak i wysokiego poziomu, umożliwiające efektywną akwizycję danych z prototypowego modułu detektora. Oprogramowanie to miało też być zintegrowane z pakietem EUDAQ, rozwiniętym w ramach projektu europejskiego EUDET, a służącym do jednoczesnej akwizycji danych z różnych detektorów, podczas eksperymentów fizyki cząstek. Dodatkowo, myśląc o przyszłym rozwoju systemu akwizycji, magistrant miał też za zadanie uruchomienie systemu operacyjnego Linux na bardziej wydajnej platformie, a mianowicie na procesorze PowerPC, dostępnym w układzie FPGA Xilinx Virtex-5 FXT.

Wszystkie wymienione cele zostały w pełni zrealizowane. Ze względu na terminarz międzynarodowych zobowiązań Katedry Oddziaływań i Detekcji Cząstek, weryfikacja stworzonego przez magistranta systemu akwizycji danych została przyspieszona, zmuszając magistranta do ekstremalnego wysiłku w ostatniej fazie prac, i odbyła się podczas przeprowadzonych w lipcu 2011 roku testów prototypowego modułu detektora świetlności na wiązce elektronowej w Hamburgu. Podczas tych testów magistrant miał możliwość wprowadzenia ostatnich poprawek w swoim oprogramowaniu. Wysiłek włożony przez magistranta w opracowanie systemu akwizycji danych został zrekompensowany możliwością aktywnego uczestniczenia w eksperymentalnych testach na wiązce prowadzonych przez międzynarodową Współpracę FCAL.

Jak już wspomniano, praca wykonana przez magistranta jest częścią działalności naukowo-badawczej Katedry Oddziaływań i Detekcji Cząstek, a wyrazem tego jest wygłoszenie przez magistranta referatu naukowego podczas spotkania Współpracy FCAL, jak również przygotowywana aktualnie publikacja naukowa, w której jest on jednym z głównych autorów. Na uwagę zasługuje fakt napisania pracy w języku angielskim, przez co jej wyniki będą dostępne dla zainteresowanej międzynarodowej społeczności.

Podsumowując, stwierdzam że wykonana praca zdecydowanie wykracza poza wymagania stawiane pracom magisterskim i zasługuje na najwyższą ocenę.
Końcowa ocena pracy przez opiekuna: 5.0

Data: 25.9.2011r. Podpis: ....

Skala ocen: 5.0 – bardzo dobra, 4.5 – plus dobra, 4.0 – dobra, 3.5 – plus dostateczna, 3.0 – dostateczna, 2.0 – niedostateczna

Acknowledgements

First of all, I would like to thank my supervisor dr hab. inż. Marek Idzik for introducing me to the interesting work of the FCAL Collaboration, as well as for encouragement, patience and instant help.

I am especially grateful to mgr inż. Szymon Kulis for continuous help with hardware, valuable suggestions for the software design and detailed bug reports, as well as for explaining, in simple words, the physics behind the principles of the developed system and obtained results.

I would also like to thank Jonathan Aguilar and Itamar Levy for modifying RootMonitor to be useful with the system, and all members of the FCAL Collaboration for the knowledge and ideas they shared with me.

Last but not least, I want to thank my girlfriend Justyna Kosmala and my friend Michał Szwaczko for proofreading drafts of this thesis, as well as the reviewer dr inż. Bartosz Mindur for valuable remarks on its final draft. Naturally, all the mistakes you will find here are mine.

Contents

Introduction **13**

1 International Linear Collider **15**

1.1 Challenges of modern Particle Physics . . . 15
1.2 The accelerator . . . 17
1.3 Detectors . . . 19
1.4 Data Acquisition . . . 20

2 Forward luminosity calorimeters in ILC **23**

2.1 LumiCal . . . 23
2.1.1 Detector characteristics . . . 23
2.1.2 Development . . . 23
2.2 Test beam setup . . . 24
2.2.1 Physical structure . . . 24
2.2.2 Data acquisition scheme . . . 25
2.2.3 Prototype boards . . . 27

3 The data acquisition firmware **29**

3.1 Requirements . . . 29
3.2 Top level architecture . . . 30
3.3 Implementation conventions . . . 31
3.3.1 Language and toolchain . . . 31
3.3.2 Memory issues . . . 32
3.3.3 Error handling . . . 33
3.3.4 Object-oriented style . . . 33
3.4 Build system and directory structure . . . 35
3.4.1 Requirements . . . 35
3.4.2 Definitions . . . 35
3.4.3 Makefiles . . . 36
3.4.4 Modules and ports . . . 37
3.4.5 Configurations . . . 38

3.4.6 Build options 39
3.5 Unit testing 40
3.5.1 Motivation 40
3.5.2 Implementation 40
3.5.3 Code coverage 42
3.6 Multitasking 43
3.6.1 Kernel abstraction 43
3.7 Input/Output 45
3.7.1 I/O streams 45
3.7.2 Debug console 46
3.7.3 Serial port driver 46
3.8 Drivers for readout board subsystems 47
3.8.1 Data concentrator 47
3.8.2 Front-end and ADC ASICs 47
3.9 Command line interface 48
3.9.1 Implementation overview 49
3.9.2 Syntax 49
3.9.3 Parsing and executing commands 50
3.9.4 Link-time composed arrays 50
3.9.5 Command completion 52
3.9.6 Command history 52
3.10 Data acquisition 53
3.10.1 Pipelined buffers 53
3.10.2 Data format 54
3.10.3 Text protocol 55
3.10.4 Binary protocol 55
3.10.5 Class design 56
3.11 Hardware monitor and power pulsing measurements 57
3.11.1 Motivation 57
3.11.2 Implementation 58
3.12 Results 59
3.12.1 DAQ performance 59
3.12.2 Power pulsing 60

4 Integrating with EUDAQ **63**
4.1 EUDAQ and its architecture 63
4.2 The Producer 64
4.2.1 Overview 64
4.2.2 The `FCALProducer` main loop 67

4.2.3 Configuring the Producer and the readout board 67
4.2.4 Reading and transmitting event data 68
4.3 The Data Converter Plugin 70
4.4 Results 70

5 Setting up Linux Operating System 73
5.1 Motivation 73
5.2 Installation process outline 73
5.3 Avnet Virtex-5 FXT Evaluation Kit 75
5.3.1 Configuration methods 75
5.3.2 Previous Linux installations 76
5.4 Software used in the installation 76
5.5 Hardware configuration 77
5.5.1 Creating embedded system design 77
5.5.2 Exporting Board Support Package 79
5.6 The U-Boot bootloader 79
5.6.1 Creating board definition 79
5.6.2 Building and installing the U-Boot image 80
5.7 Building and booting Linux 82
5.7.1 Configuring and building the kernel 82
5.7.2 Root file system in ramdisk 83
5.7.3 Compiling the device tree 83
5.7.4 Downloading and booting 84
5.8 Building the root file system 85
5.8.1 Configuration 86
5.8.2 Modifying the root file system 86
5.8.3 Building the image 87
5.9 Flash memory organization 88
5.9.1 Determining the bitstream location 88
5.9.2 Bootloader 88
5.9.3 Linux kernel, device tree and storage 88
5.9.4 Passing the partition table to the kernel 88
5.10 Programming the images 89
5.11 Porting the DAQ firmware to Linux 90
5.11.1 Data acquisition 90
5.11.2 Command interface 91
5.12 Summary 91

Summary and future work 93

A Source code (selection) 97

A.1 Firmware 97

A.1.1 Build system 97

A.1.2 Data acquisition 100

A.2 FCAL Producer for EUDAQ 108

B DVD-ROM contents 117

Bibliography 119

List of Listings 125

List of Figures 127

List of Tables 129

Introduction

Paricle physics studies basic constituents of matter and radiation. The most important achievement of particle physicists was the formulation of the theory called the *Standard Model* in the second half of the 20th century. This theory identifies a set of fundamental particles, of which all the matter in the universe is composed. These particles are interacting through the three fundamental forces: strong, weak, and electromagnetic forces, using mediating gauge bosons: the gluons, W^-, W^+ and Z bosons, and the photons. The Standard Model also predicts the existence of yet another boson, the Higgs boson, which is believed to give other particles their masses as they interact with it [1].

The experimental proofs of the Standard Model have been obtained with the use of particle colliders—a type of particle accelerators, in which two beams are directed against each other so that the particles collide and the products of such collisions can be identified and their properties measured. This is exactly what is being done now in the Large Hadron Collider (LHC) in CERN, where protons are accelerated to very high energies and collided. Among the products of these collisions, the Higgs boson is expected to be found. However, even if found, which is not obvious from the latest LHC results, precise analyses of its properties will require a different, simpler type of collisions. For that reason, construction of an electron-positron collider is planned in the near future. Two concepts of such collider are being studied, and the decision to build one of them will be made based on the energy scale in which new physics phenomena will be found in the LHC. Both of these concepts, the International Linear Collider (ILC) [2] and the Compact Linear Collider [3], are under active research and development studies which will determine the structure of the future collider and its detection systems.

Experiments in particle colliders are the source of huge amounts of data which have to be properly stored and made available for analyses done by thousands of scientists around the world. Therefore the infrastructure for the data acquisition and storage is also a major challenge and must be carefully designed to meet the requirements imposed by the large number of readout channels and different types of data produced by various detectors. Although the research towards the computing infrastructure for the future detectors is currently in relatively early stage, anticipating the continuation of the rapid development in the data storage and networking technology market, the current prototypes of the detectors also need data acquisition systems to collect data needed to evaluate their performance.

In this thesis, the development of one of such systems is described. It is a data acquisition system for prototypes of LumiCal and BeamCal, the calorimeters of the very forward region of the future linear collider. The system is composed of the firmware controlling the operation of a dedicated readout board, and the desktop application to store the data and convert them to the representation suitable for further analyses. Although the system was meant to operate in a very particular environment, its design allows for adopting it in other applications, as well as makes it easy to be extended for improved prototypes built using different hardware.

The thesis is organized into five chapters followed by a brief summary.

In the first chapter, the ILC is introduced, including its role in the expected progress in our understanding of the basic structure of matter, as well as the physical structure of the accelerator and planned architecture of the detector system and the data acquisition infrastructure.

The second chapter outlines the structure of the luminosity calorimeters in the very forward region of the accelerator, briefly summarizes current R&D status of the detectors and provides an outline of the environment in which the data acquisition system was meant to run.

In the third chapter, the readout board firmware is presented. First, the analysis of requirements is reported. The design and implementation of the firmware and its modules are then described, followed by a brief discussion of the preliminary results that were obtained.

The fourth chapter covers the integration of the system into EUDAQ—a desktop data acquisition software package used in multiple physics experiments. An outline of the package itself is given, and the process of extending it to work with the developed prototype board is reported. The chapter ends with examples of basic data analyses.

The fifth chapter deals with the installation of the Linux operating system on an embedded board with Virtex-5 FPGA. Outline of the board features is given first, followed by a discussion of several steps of preparing, building and installing software, and evaluation of the performance of the installed system. Initial attempts to port the firmware presented in the third chapter to the new system are also reported.

The thesis concludes with a brief discussion of the achieved results, highlighting the work done by the author, and giving some outlook for future work.

Chapter 1

International Linear Collider

1.1 Challenges of modern Particle Physics

Theoretical and experimental work in particle physics concentrates around discovering the universal principles which can be observed at scales of distance and time usually being far beyond our natural perception. Large effort is directed into unifying the results of experiments in particle accelerators with astrophysical observations of particles occuring naturally, aiming at construction of a simple and consistent general theory explaining the basic laws governing our world [1].

The greatest success of this effort is known as the Standard Model, a theory developed in the early 1970s, which explains the structure of matter as made entirely of twelve basic subatomic particles, interacting through three fundamental forces. The Standard Model fitted well into experimental results that had been collected until that time and found more evidences later as well [4].

According to the Standard Model, summarized in Figure 1.1, the matter is constructed of quarks and leptons, while the interactions between them are based on the exchange of force carrier bosons. There are six quarks, further divided into three pairs, called "generations". The first generation is formed of the "up quark" and the "down quark", the "strange quark" and the "charm quark" are in the second generation, while the "top quark" and the "bottom quark" constitute the third. Similar arrangement has been applied to leptons—the "electron" and the "electron-neutrino", the "muon" and the "muon-neutrino", and the "tau" and the "tau-neutrino". Particles in the first generation are lighter and they are known to form all stable matter in the universe, heavier particles being unstable and likely to decay [4].

The three fundamental forces are associated with particles called "gauge bosons": the "gluon" (appearing in eight different types or "colors") carries the strong force, the "photon" carries the electromagnetic force, and the "W^+ boson", the "W^- boson" and the "Z boson" are associated with the weak force.

The Standard Model has shown to be incomplete, though. The way to fit the gravity into it hasn't been found yet [4], and there are many other questions to which the Standard Model

	Fermions (matter)			Bosons (forces)
Quarks	2.4 MeV, 2/3, 1/2, u, up	1.27 GeV, 2/3, 1/2, c, charm	171.2 GeV, 2/3, 1/2, t, top	0, 0, 1, γ, photon
Quarks	4.8 MeV, -1/3, 1/2, d, down	104 MeV, -1/3, 1/2, s, strange	4.2 GeV, -1/3, 1/2, b, bottom	0, 0, 1, g, gluon
Leptons	<2.2 eV, 0, 1/2, ν_e, electron neutrino	<0.17 MeV, 0, 1/2, ν_μ, muon neutrino	<15.5 MeV, 0, 1/2, ν_τ, tau neutrino	91.2 GeV, 0, 1, Z^0, weak force
Leptons	0.511 MeV, -1, 1/2, e, electron	105.7 MeV, -1, 1/2, μ, muon	1.777 GeV, -1, 1/2, τ, tau	80.4 GeV, ±1, 1, $W^{\pm}$, weak force

Legend: mass, charge, spin, name

Figure 1.1: Fundamental particles of the Standard Model [5].

doesn't give answers.

The current research is mainly focused on the phenomena anticipated at Terascale energies[1]. Several hypotheses extending the Standard Model to that range need experimental confirmation. According to [1], they include, but are not limited to:

Higgs field The hypothetical Higgs field gives other particles mass as they interact with this field. Currently built accelerators are expected to find Higgs boson (or bosons) and measure their properties. The Higgs field also breaks the electroweak force into electromagnetic and weak forces at the Terascale. Predicted force unification at TeV and higher energy scales is shown in Figure 1.2.

SUSY Supersymmetry is a theory which predicts that each of currently known Standard Model particles has a corresponding “sparticle” that differs from it by half a unit spin. Currently, the most well-motivated variant of this theory (the “broken symmetry” theory), assumes that the supersymmetric particles are heavier than their respective counterparts, and the lightest of them may be found at TeV-scale.

Extra dimensions Some particles associated with extra spatial dimensions may be found, if they masses are less than a few TeV.

The Higgs boson (or bosons) and some lighter particles of supersymmetry, such as squarks or gluinos, are likely to be found in the ATLAS and CMS experiments at the Large Hadron Collider (LHC) if they exist [1][7][8]. As of August 2011, the most recent results from the experimental search of the Higgs boson in the CMS and ATLAS experiments excluded its existence in wide ranges below 600 GeV with high level of certainty [9][10].

However, since the particles collided at the LHC (protons) are complex particles composed of three quarks bound with strong interaction, their collisions give large number of diverse products,

[1]That is, energies measured in teraelectronvolts (TeV)

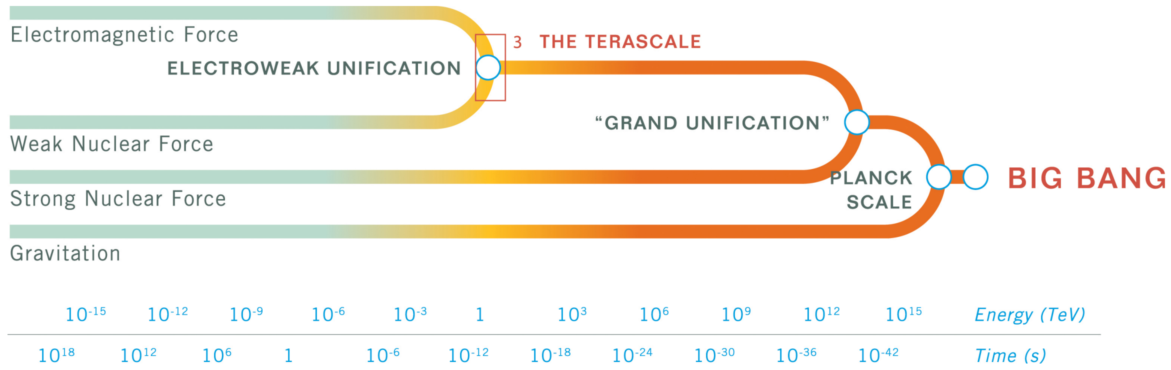


Figure 1.2: Energy scale for the unification of fundamental interactions. Source: [6].

which cause the background noise make the data analysis difficult [7].

Colliding fundamental particles of the Standard Model, such as electrons and positrons is expected to give less side products, allowing more precise measurements. Currently, two concepts of new teraelectronvolt electron-positron collider are being researched, and the decision, which one will be installed, depends on the energy scale in which interesting results will show up in the LHC. Should it be less than 1 TeV, the International Linear Collider [2] is more likely to be built, otherwise the new collider will be CLIC (Compact Linear Collider) [3].

The results from the LHC and the new linear collider will complement, the first allowing for measurements in higher energy scale, and the second, for very high precision. Since the basic principles of CLIC and ILC are similar, and research on the ILC has been carried out for slightly longer time, only ILC will be described here.

1.2 The accelerator

Colliders of the highest energy at the time of this writing are proton synchrotrons. The Tevatron in Fermilab, USA, accelerates protons and antiprotons to 980 GeV per beam. The LHC in the European Organization for Nuclear Research (CERN) is capable of colliding protons at 7 TeV (3.5 TeV per beam) and approaches twice that energy in the near future.

Such scale of energy is unreachable for the current electron synchrotrons. The energy losses due to synchrotron radiation are inversely proportional to the fourth power of mass of particles. Electrons are particles lighter than protons by three orders of magnitude, and the loss rate for electrons is over 10^{13} times the loss rate for protons of the same energy in the accelerator of the same diameter [11]. Consequently, the design of the new teraelectronvolt collider for electrons and positrons assumes accelerating them in a straight line.

The projected parameters of the ILC, required by the anticipated physics experiments are as follows:

- maximum energy of collisions up to 500 GeV, with possible upgrade to 1 TeV,
- ability to perform scans in small steps starting from 200 GeV,

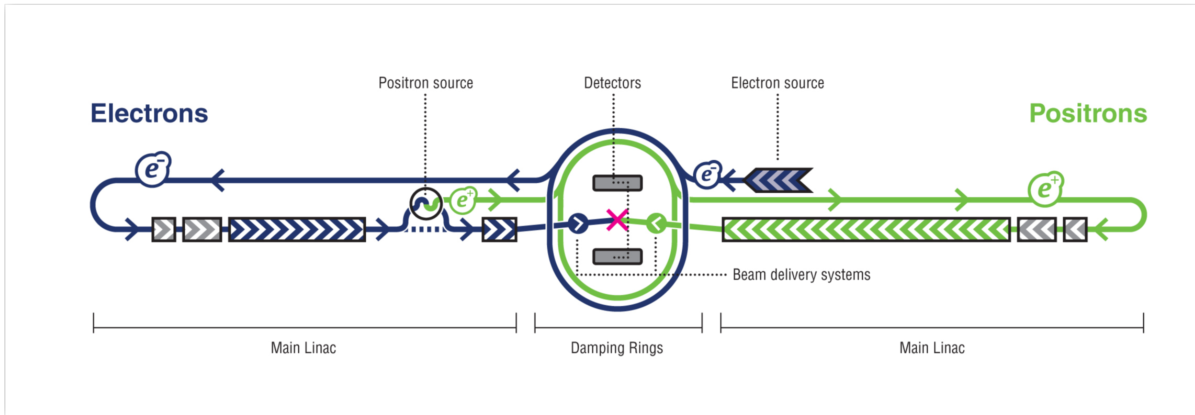


Figure 1.3: A schematic layout of the International Linear Collider. Source: [13].

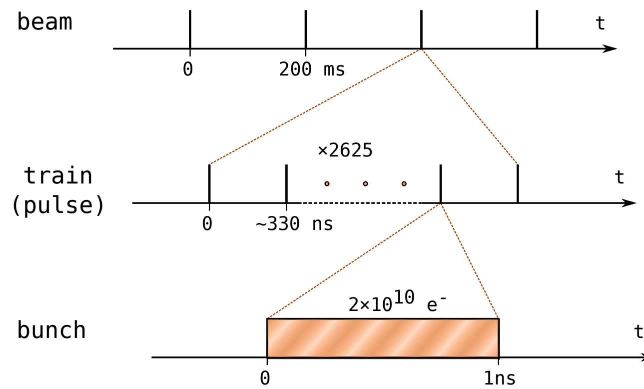


Figure 1.4: ILC beam structure and timing [12].

- peak luminosity of 2×10^{34} cm^{-1}s^{-1}, giving the total luminosity of 500 fm^{-1} over the first four years of operation,
- polarization with a degree larger than ±80% for electron beam and ±50% for positron beam,
- both beam polarization and energy should be stable and measurable with the precision of about 0.1% [1][12].

A schematic layout of the proposed architecture for the International Linear Collider, fulfilling the above requirements, is shown in Figure 1.3. In this design, the accelerator is based on 1.3 GHz superconducting radio-frequency accelerating cavities operating with electric field gradient of 31.5 MV/m. The structure and timing of beam pulses is shown in Figure 1.4. Electrons are emitted in *bunches* of approx. 2×10^{10} particles within 1 ns, repeated 2625 times every 330 ns, to constitute a *bunch train* lasting for about 1 ms. The trains are repeated every 200 ms [12].

The beams are initially accelerated to 5 GeV in low energy damping rings, being 6.7 km in circumference. They are then directed to 11-km-long main linacs to gain the target energy of 250 GeV, and then to 2.2-km-long beam delivery systems, strongly focusing the beams before they finally reach the interaction point. The whole site will be 31 km long, and will be extended when upgrading to 1 TeV [12].

1.3 Detectors

The interaction point will be surrounded by a set of detectors, aiming at extracting as much precise tracking and calorimetric data as possible from the events, allowing reconstruction of energy, momentum and tracks of every product of the collisions. As the requirements on precision and performance are significantly beyond what has been developed for the LHC, the design of the detectors is under intensive R&D studies of international groups of scientists, from which four concepts of detector block emerged. Planned installation will include realization of two of these concepts. They will work in "push-pull" scheme, sharing a single interaction point, i.e. one detector will operate while the other will be parked aside in the same hall [14].

In 2008, development in two of these concepts, Large Detector Concept (LDC) and Global Large Detector (GLD), was unified under joint cooperation known as the International Large Detector (ILD) [15].

There are several detector systems in the ILD, performing complementary measurements. Overview of the location of ILD subdetectors around the interaction point is given in Figure 1.5. Their characteristics, according to [16], are as follows:

Vertex Detector (VTX) Placed at the closest distance from the IP, at a radius of several centimeters, it consists of several layers of precise silicon pixel sensors, allowing accurate measurement of positions of charged particles.

Time Projection Chamber (TPC) Filled with gas and placed under high voltage across its length, causes charged particles passing its length to ionize the gas molecules which then drift towards the detector borders. Position of ionization is then determined based on the time of this drift.

Electromagnetic Calorimeter (ECAL) Measures energy of photons and charged particles passing through it. It uses tungsten absorber layers to develop a shower of secondary particles out of the particles. The shape of the shower indicates the type of the original particle.

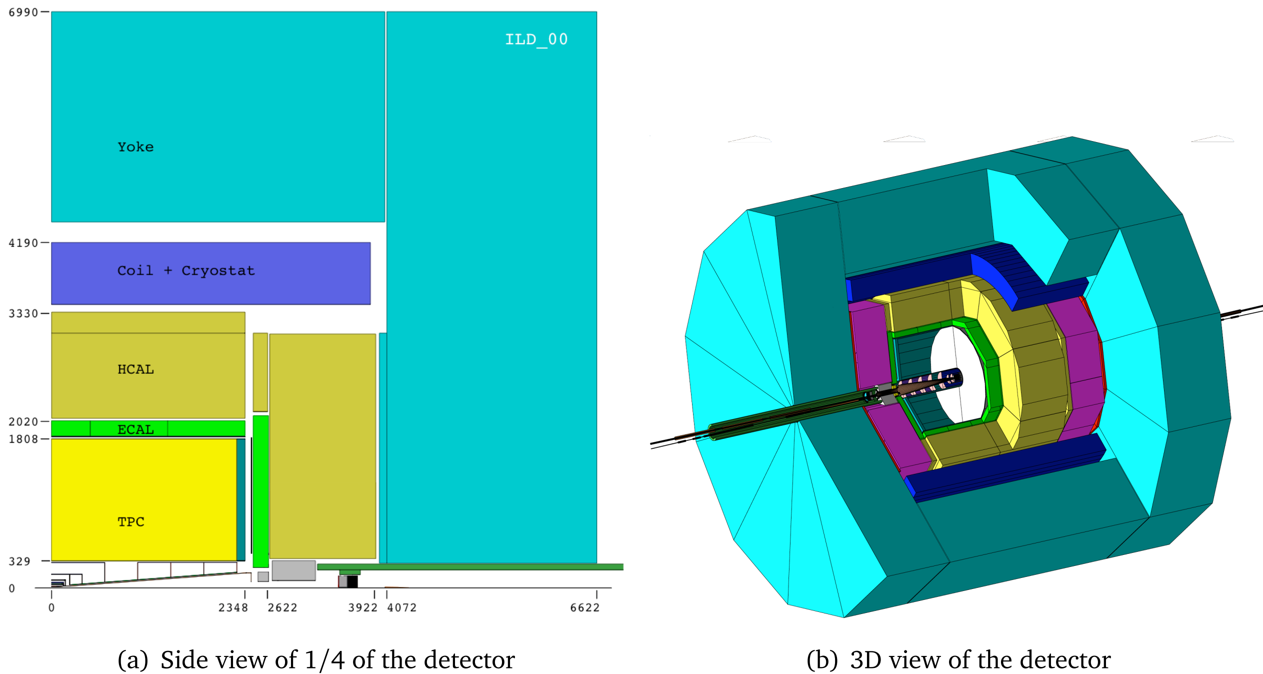
Hadronic Calorimeter (HCAL) Based on a similar principle to ECAL, uses steel absorber to develop showers out of neutral and charged hadrons in order to measure energy deposited by these hadrons.

Forward Calorimeters (FCAL) Cylindrical electromagnetic calorimeters placed around the beam pipe at very small angles, close to the interaction point. They include LumiCal, for precise luminosity measurements, and BeamCal, giving fast estimate of the luminosity.

Muon Detector Placed at the outer radius of ILD, because muons are not significantly affected by the calorimeters.

The above systems are closed inside the superconducting solenoid coil producing magnetic field of several T, required to bend tracks of charged particles inside the detector [16].

Identifying particles and measuring their properties will be done using the Particle Flow Approach (PFA), in which results from trackers and calorimeters are combined and used as input



(a) Side view of 1/4 of the detector

(b) 3D view of the detector

Figure 1.5: Layout of ILD model used in simulations. Source: [17].

for automated clustering algorithms [18]. Figure 1.6 gives the basic idea about how different particles can be identified.

1.4 Data Acquisition

The data acquisition (DAQ) mechanism for ILD is still only roughly defined, staying in a conceptual phase. There are two reasons to keep the binding decisions in this area postponed to as late as possible. First, the ILD DAQ architecture will be highly dependent on the final design of each particular subdetector system. Second, fast development in high performance computing and networking infrastructure will likely cause that the systems chosen in a few years will likely be charaterized by higher throughput and smaller power consumption at the same or lower cost. The following brief summary of DAQ and computing concepts is mainly based on the description in [17].

The required precision and non-continuous mode of operation make the ILC quite different from the accelerators built so far, also in terms of requirements imposed on the planned DAQ architecture. The main source of complexity will be the total number of readout channels, estimated as nearly 10^9 (i.e. 10 times the number of channels in LHC), producing ~ 340 MB of raw data per bunch train. On the other hand, the overall throughput and CPU power required will be significantly smaller than in LHC, because of the pulse operation mode of ILC beams, giving reasonable amount of time to compress and transmit the data.

Proposed general layout of the DAQ system includes the following steps in data acquisition and processing:

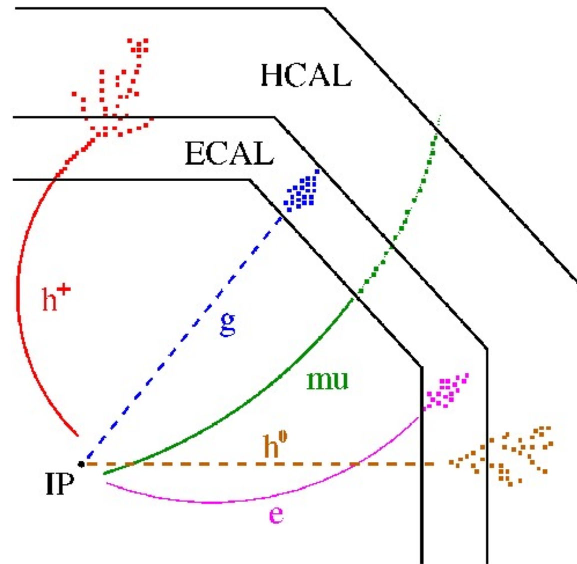


Figure 1.6: Traces left in tracker and calorimeters by different particles: charged hadron (h^+), photon (g), muon (mu), neutral hadron (h^0) and electron (e). Source: [18].

Detector Front Ends Continuously shaping and digitizing signals from the sensor, and capable of storing data from the whole bunch train, and initially reducing the bandwidth by multiplexing channels and simple compression techniques, such as zero suppresion.

Interface Readout Units With larger, multi bunch train buffers and flexible input interfaces, will provide buffering and unified output for the next layer.

Common Event Building Network and Data Processing Nodes Performing event building, processing and classification (further reduction of data volume by selection of "bunches of interest"). The peak throughput required for the network is estimated at 20 GB/s.

Data Storage and Analysis Infrastructure Capable of storing and processing data volumes of tens of petabytes per year.

A similar, but less demanding side infrastructure for registering metadata such as beam parameters and beam conditions will also be developed.

Software presented in next chapters of this thesis contributes to the research on the data acquisition system for ILD.

Chapter 2

Forward luminosity calorimeters in ILC

This chapter provides background for the data acquisition firmware requirements. The planned detector structure is described, followed by the description of the current prototype and its test environment.

2.1 LumiCal

2.1.1 Detector characteristics

LumiCal is one of the calorimeters in the very forward region in the ILD concept. It will give precise luminosity measurements based on Bhabha scattering. Hardware design of LumiCal is mainly consigned to the Department of Particle Interactions and Detection Techniques at AGH-UST and the Institute of Nuclear Physics of Polish Academy of Sciences.

LumiCal will be positioned around the beam pipe, over 2 meters from the interaction point. Its position among other components of the very forward region is shown in Figure 2.1. It will consist of 30 cylindrical layers of tungsten interleaved with silicon sensors, covering the range of polar angles from 32 mrad to 74 mrad. Each layer will be segmented radially and azimuthally into pads of different sizes (Figure 2.2). The readout electronics will be placed at the outer radius of LumiCal, between tungsten plates. Total number of readout channels is estimated to be about 200,000 [17].

2.1.2 Development

Current development towards the luminosity detectors, carried out by the FCAL Collaboration [19], includes simultaneous work on the following aspects of the detectors:

- mechanical arrangement of sensor and absorber planes and readout electronics,
- silicon sensor for LumiCal and GaAs sensor for BeamCal, including their geometries, electrical charateristics and connection to readout electronics,

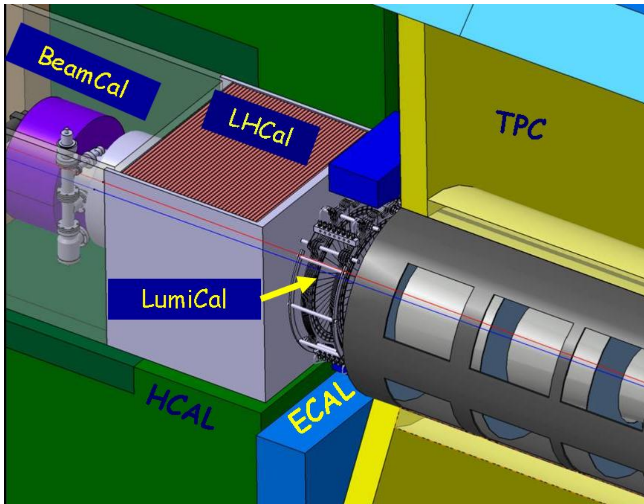


Figure 2.1: Position of LumiCal within the very forward region. Source: [17].

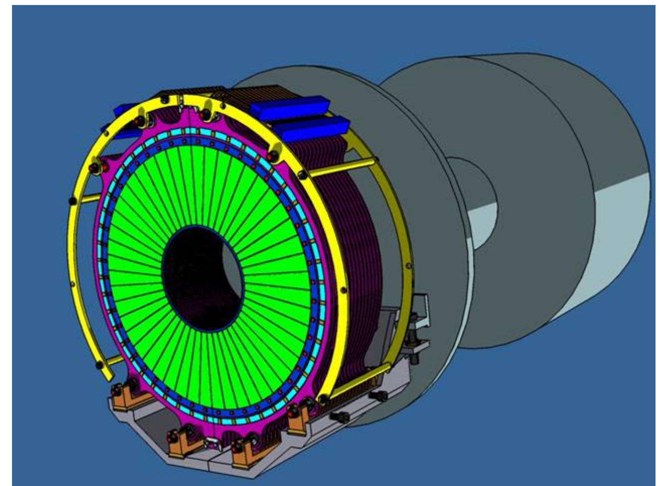


Figure 2.2: Overview of the structure of LumiCal. The sensor sectors are marked green. Source: [17].

- front-end electronics, converting charge deposited in sensor pads into voltage pulses,
- Analog-to-Digital converters (ADCs),
- firmware for FPGA (Field-Programmable Gate Array) and/or microcontroller for controlling the process and data transmission.

In terms of readout electronics, the aim is to reduce power consumption (taking advantage of the pulse operation mode of ILC), noise and area per readout channel, as well as make both front-end and ADC fast enough to precisely measure pulses from electromagnetic showers coming from distinct bunches [20][21].

Apart from laboratory tests with cosmic radiation and radioactive isotopes, prototypes are tested on test beams, such as the 6-GeV electron test beam facility in DESY, Hamburg, Germany [22]. Data obtained on test beam runs are used to evaluate performance of hardware and algorithms used for energy deposit reconstruction.

2.2 Test beam setup

2.2.1 Physical structure

Tests performed in July 2011 were carried out using the setup presented in Figure 2.3. The setup includes two detectors installed on a common optical bench: the LumiCal prototype and the MVD Telescope. The LumiCal prototype consists of two boards connected to each other: the sensor board, on which only a silicon pad sensor is installed, and the readout board, containing all circuits required to process the signals from the sensor and transmit the data to PC. The MVD Telescope consists of three detector planes of dense silicon strips arranged perpendicular to each other and is used for precise measurement of beam position [23]. To synchronize both readout

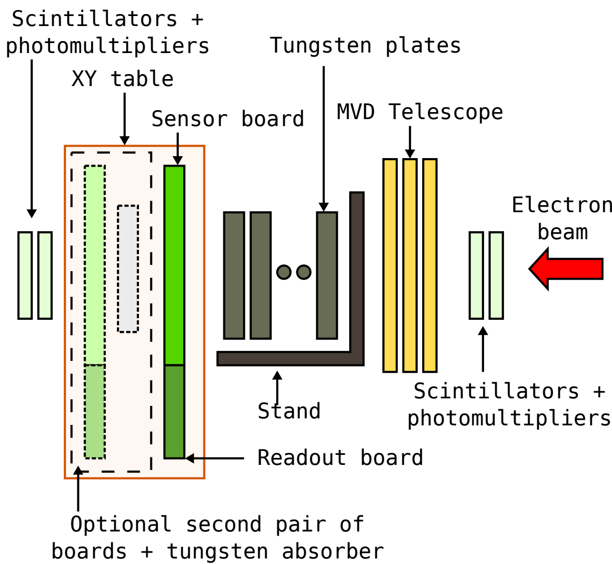


Figure 2.3: Schematic diagram of the test beam setup.

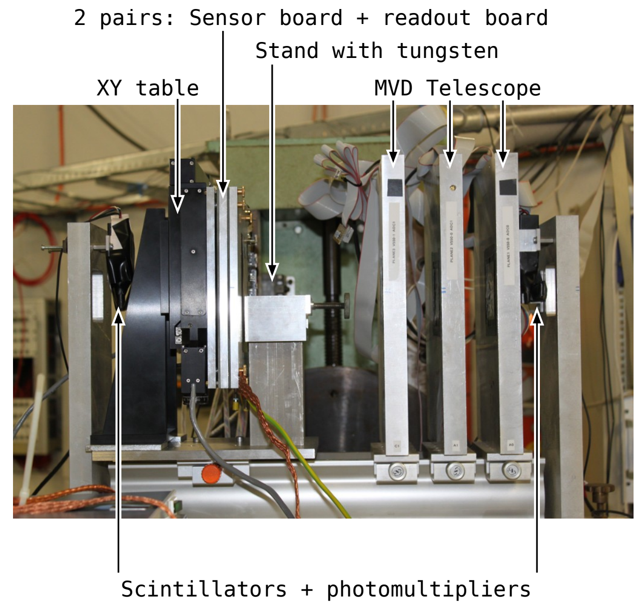


Figure 2.4: Photograph of the test beam setup (O. Novgorodova).

systems with the beam and with each other, a triggering circuit is used, which consists of two pairs of scintillators followed by photomultipliers, located at both sides (i.e. entry and exit) of the setup. They are connected to inputs of the Trigger Logic Unit (TLU), which generates trigger pulses for the detectors once it registers a coincidence of pulses at its inputs [24].

The LumiCal prototype boards are placed on an XY table controlled by dedicated software, which allows for directing the beam at precisely defined regions of the sensor. The table may accomodate one or two detector planes, with a single tungsten plate between them in the latter case. In front of the sensor an additional stand is situated, on which several tungsten plates may be placed in order to study electromagnetic shower development.

Photograph of the setup installed during tests in July is presented in Figure 2.4. Similar setup is planned for tests of LumiCal and BeamCal prototypes in November 2011.

2.2.2 Data acquisition scheme

General architecture of the developed data acquisition system is presented in Figure 2.5. The design of LumiCal prototype boards follows the typical architecture of DAQ systems—the analog signal from the sensor is amplified and shaped, and then digitized. Digital data are processed and transmitted to the PC computer, on which dedicated software controls the process, presents histograms of collected data on-line and stores the data for further processing.

The timing diagram of readout is shown in Figure 2.6. ADCs work continuously. Samples are stored in real time in a circular buffer of up to 128 samples per channel. When the TLU receives simultaneous pulses from all photomultipliers, it is likely that there is something interesting to measure (front-end response in the diagram). If no detector is busy processing the previous event

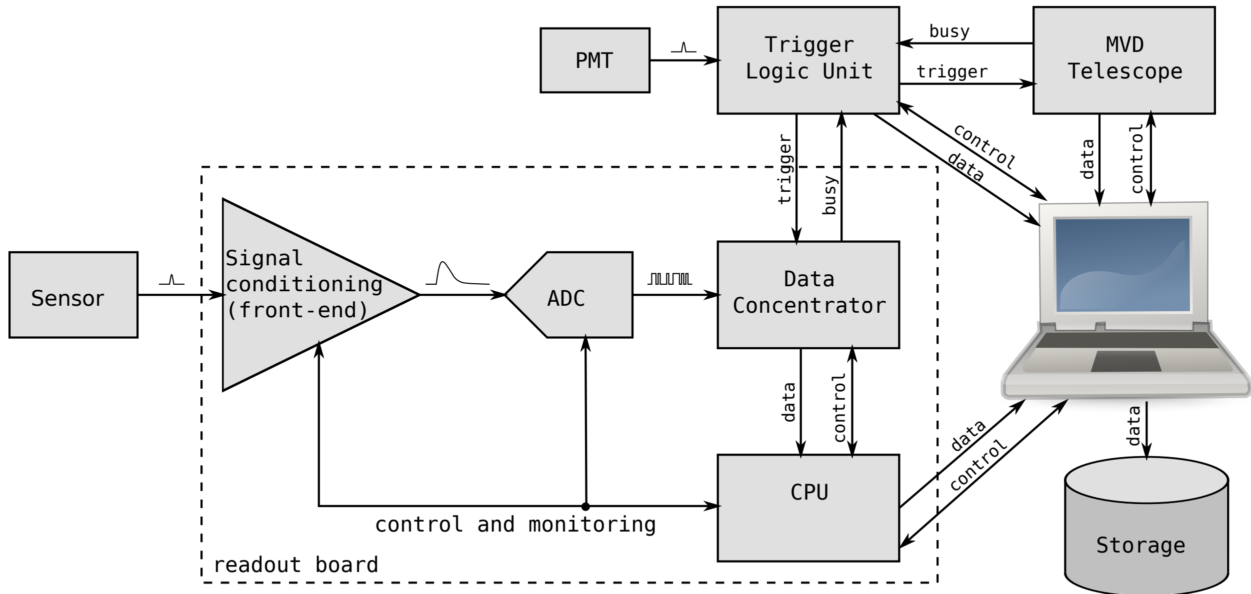


Figure 2.5: Signal and data flow in the FCAL DAQ system.

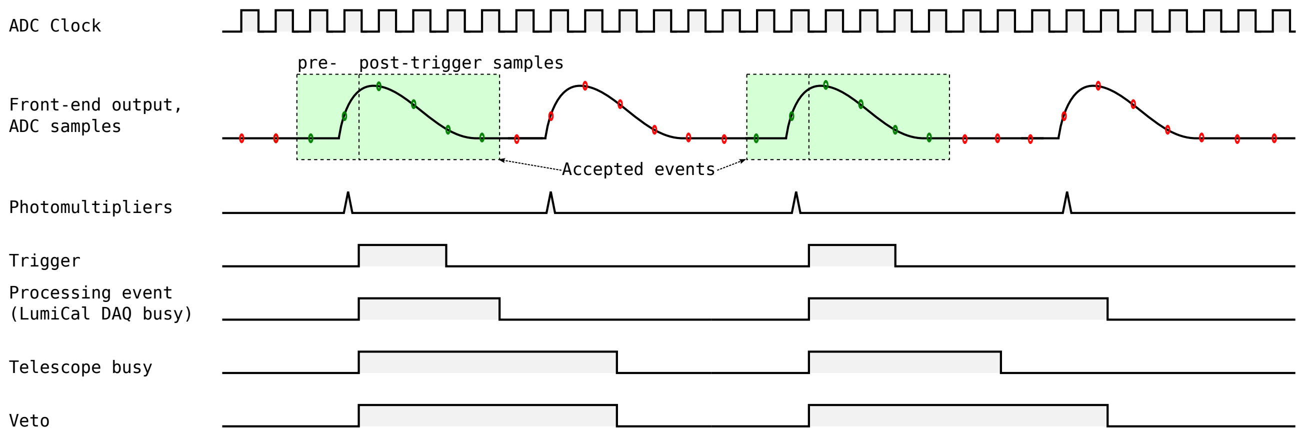


Figure 2.6: Data acquisition timing diagrams, showing relationships between signals for a single readout board and the whole setup.

(the `veto` signal, being a logical sum of `busy` signals from all detectors, is low), TLU generates a trigger pulse for all detectors.

In response, the data concentrator drives its `busy` signal high and continues reading samples until a predefined number of consecutive samples (*post-trigger* samples) is read, and then generates interrupt for the microcontroller. The microcontroller which builds a data packet out of samples stored before the trigger (*pre-trigger* samples) and post-trigger samples, transmits the packet and clears the `busy` flag, informing the rest of the system that it is able to accept new events. The telescope DAQ behaves similarly.

Using the triggers allows for greatly reducing the amount of data to be transmitted, stored and analyzed, by suppressing the data measured during idle periods of the beam at very early stage. The `veto` mechanism ensures that data collected from all detectors refer to the same set of events.

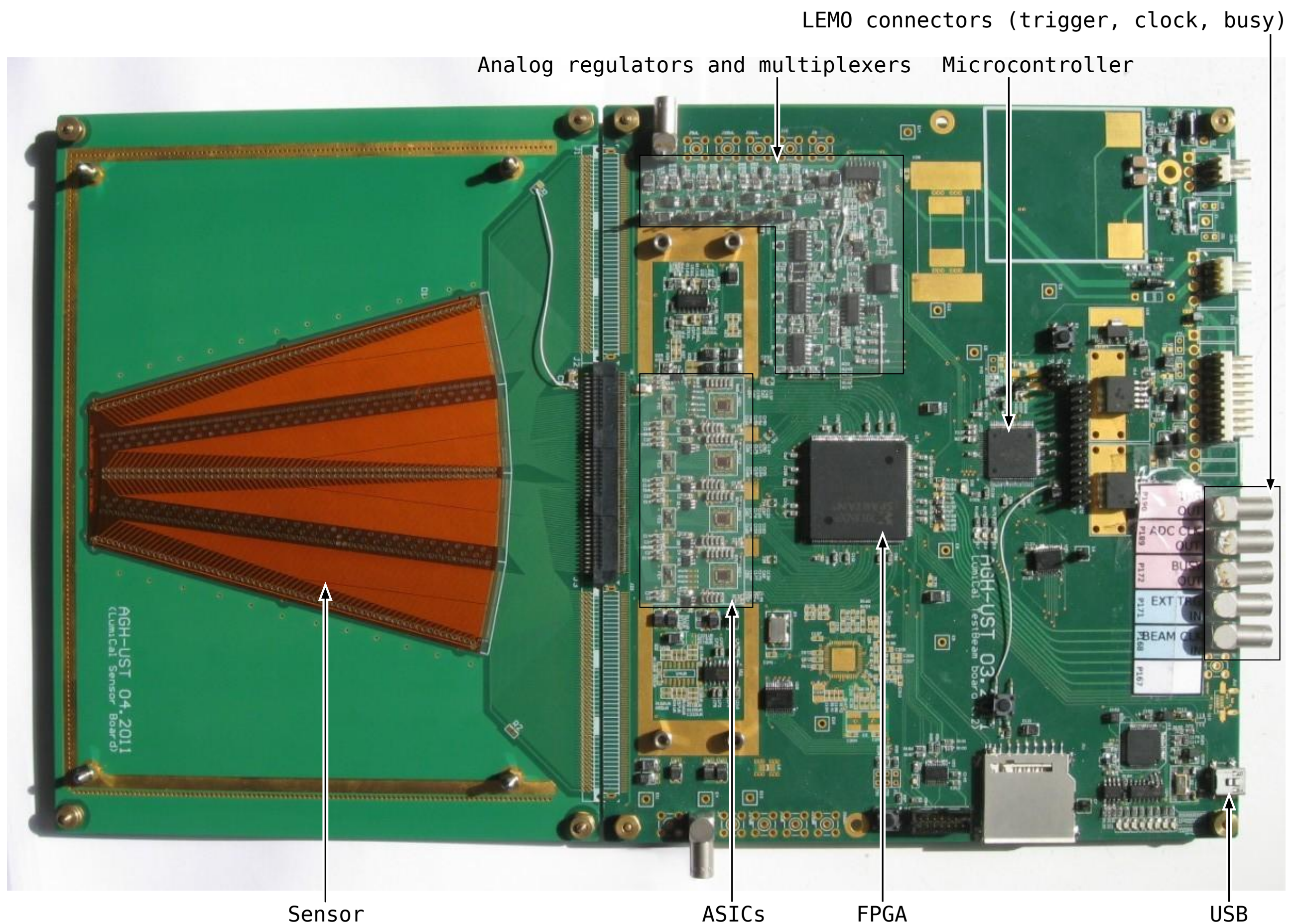


Figure 2.7: Photograph of the sensor board and the readout board connected to each other.

2.2.3 Prototype boards

Prototype boards were developed as a close approximation of a section of LumiCal. Photograph of both boards connected to each other is shown in Figure 2.7.

The sensor board includes a dedicated silicon sensor manufactured by Hamamatsu Photonics and connectors to attach the readout board. The sensor is segmented into 4 sectors, 64 pads each, giving the total number of 256 pads [25]. Although all sensor pads are connected to the readout board connector, only 32 of them are handled by the readout board.

The readout board comprises the following blocks:

Sensor board connectors Different sensor boards may be attached using standard 2×45-way gold pin connectors.

Front-end and ADC ASICs Four pairs of dedicated 8-channel front-end [20] and ADC [21] integrated circuits are installed. The ADCs work with resolution of 10 bits and sample rate of 20 MSps, giving the total raw data rate of 6.4 Gbps.

Data concentrator Implemented in Xilinx Spartan-3E FPGA [26], is responsible for providing clock signals for ADCs, copying the data from ADC to the internal circular buffer and handling the trigger logic.

Microcontroller Atmel ATxmega128A1 8-bit RISC microcontroller operating at 24 MHz. Its features include: 128 kB of flash memory for program code, 8 kB of Static RAM (SRAM) for data, 2 kB EEPROM, 4-channel DMA controller, 8 Universal Synchronous and Asynchronous Receiver and Transmitters (USARTs), 16-channel ADCs with 12-bit resolution and sampling rate of up to 2 MSps, External Bus Interface (EBI), Digital-to-Analog Converters (DACs), and multiple General Purpose I/O ports [27]. Multiple hardware USARTs are used to handle separate communication channels (i.e. data, commands and monitoring). Advanced ADC features are used to measure supply voltages and currents drawn by the ASICs in order to evaluate their power pulsing capabilities. The EBI allows accessing the data concentrator registers, including the data buffer, presented to the microcontroller as external RAM.

Analog regulators and multiplexers Together with ADC, DAC and I/O ports of the microcontroller, they are used to configure ASIC parameters and measure their operating conditions.

Communication interfaces FTDI Chip FT4232H serial-to-USB converter is installed, providing 4 independent channels with maximum data transmission rate of 12 Mbps each over standard USB 2.0 link [28]. Installed HDMI connnector was meant to provide a custom fast data and control link, but is currently unused.

LEMO connectors Used to connect additional control signals, such as external trigger, external clock, or the `busy` signal.

Chapter 3

The data acquisition firmware

In this chapter, the readout board firmware is presented. First, the general requirements are outlined. Then, the adopted conventions and development tools are described and justified. Following that, the basic coding framework is described, including the build system, architecture of unit tests, as well as multitasking and Input/Output subsystems. Finally, design and implementation of the three essential tasks of the firmware are described, followed by the summary of all developed firmware modules.

3.1 Requirements

The firmware controlled the operation of readout board for test beam measurements in July 2011. It runs on the AVR ATxmega128A1 8-bit RISC microcontroller. It communicates with data concentrator implemented in FPGA and connected to the microcontroller using external memory interface and an interrupt pin. It controls operation of other on-board devices through the digital and analog inputs and outputs. It communicates with PC computer through UARTs connected to USB to UART bridge, to accept commands and send out data.

In normal operation mode, the firmware is used by a physicist doing measurements on a test beam. The user does the following:

1. Configures board parameters, preferably using configuration file prepared beforehand.
2. Starts a run and watches histograms of the collected data on-line.
3. Repeats the measurements with the same or new configuration.
4. Performs off-line analyses of the data, usually combined with data from other detectors working on the same beam and, optionally, with metadata such as hardware operating conditions.

Of course, the firmware neither draws anything itself, as there is no graphical display on the readout board, nor it performs any analyses. Graphical user interface (eventually provided by

extending the EUDAQ framework—see Chapter 4) communicates with the firmware on behalf of the user. To allow that, the firmware must provide the following services:

Command interface Receiving commands, such as “start run” or “measure ADC temperature”, optionally responding with requested information (e.g. “45°C”) or an error message in case of failure. Design and implementation of this part is described in Section 3.9.

Event building and transmission Event rate should be as high as possible within hardware limitations. The mininum required event rate is about 100 Hz[1] (Section 3.10).

Monitoring Recording the operating conditions may improve results of data analysis. In the fast sampling mode, power pulsing characteristics of ASICs may also be measured (Section 3.11).

The readout board hardware and data concentrator soft core were developed simultaneously with the firmware. Their development required at least a minimum working command interface for testing and calibration, once the microcontroller is assembled on the board. Thus, the command interface should be created first.

Potential maintainers mostly use GNU/Linux as their work environment, so the build system should allow building the firmware and programming it into the target system from within GNU/Linux.

Additionally, the system will evolve to different platforms, so it was desirable to clearly separate parts that may be reused on different hardware in the future from the code specific to the current readout board.

3.2 Top level architecture

A layered architecture presented in Figure 3.1 was proposed for the firmware, with the intention to separate the layers using possibly simple interfaces. Such architecture was meant to make it easier to perform automatic tests without target hardware and to improve the firmware portability.

The need to run three tasks concurrently, with the requirement of possibly low latency on the event processing task, led to adopting a real-time kernel, which also made separation of tasks and communication between them easier. Many real-time kernels, such as FreeRTOS [29] or uC/OS-II [30], consist of a portable core and a set of ports to allow the core run on different machines. To reduce the dependency of the firmware on any particular kernel, a simple abstraction layer over the kernel was also introduced. Details are described in Section 3.6.

Atmel provides basic drivers and examples together with documentation of peripherals integrated in the ATxmega microcontroller [31]. Most of them consist of functions and macros that do nothing but set the value of a register or a group of registers. They are indicated in the

[1] Determined by expected performance of the MVD Telescope.

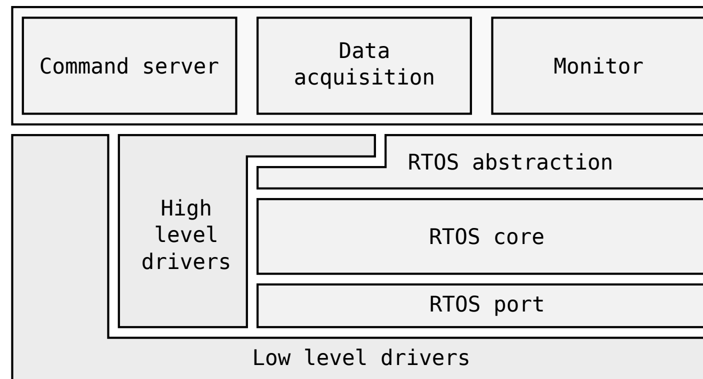


Figure 3.1: Top level architecture of readout board microcontroller firmware.

diagram as *low level drivers*. In some cases they were sufficient, but for USARTs and ADCs, more advanced drivers were written, which perform block transfers using dynamically allocated DMA channels. The latter are indicated in the diagram as *high level drivers*.

The drivers written for the data concentrator and on-board multiplexers may also be considered low level drivers, as they perform actions of similar level of complexity as the Atmel drivers.

Apart from the main firmware application, for some components small test applications were developed, used to confirm their functioning individually, by means of both unit tests (Section 3.5) and user interaction.

3.3 Implementation conventions

3.3.1 Language and toolchain

Compilers available for AVR platform include C and C++ compilers from both the GNU project [32] and IAR Systems [33]. The latter is a commercial tool running only in Windows, so it was not acceptable for this project and the GNU C compiler was chosen.

The code was written in C99 with GNU extensions and little assembly. Some useful GNU extensions to C language (besides the inline assembly syntax, which is also compiler-specific) include:

- statement expressions, i.e. statements and declarations inside expressions—particularly useful when defining complex macros,
- `noreturn` functions, saving a considerable amount of stack area for each task being an endless loop,
- built-in functions—GCC offers built-in functions for elementary tasks like byte swapping and counting bits, often being well-optimized for each target platform,
- variadic macros (also a C99 feature),

- inline functions (also C99), in many cases giving source code that is safer and easier to read than functionally equivalent macros, without losing efficiency.
- single-line (*C++-style*) comments (also C99) [34].

Use of these features might be considered a portability issue, since support for ISO C99 varies between different compilers and between different versions of the same compiler, and GNU extensions are obviously limited to GCC. However, GCC is available for most popular embedded platforms, including those planned for future versions of the readout system (PowerPC or MicroBlaze). It is also actively developed and popular in both open-source and commercial applications, so adhering to GCC should not be a serious limitation for the future.

The firmware described in this chapter was developed using AVR-GCC 4.3.4, AVR-Libc 1.6.7 [35] and GNU Binutils 2.20 [36]. AVRDUDE 5.10 [37] was used as a programming utility for the microcontroller.

3.3.2 Memory issues

The AVR ATxmega128A1 microcontroller has 8 kB of built-in data memory and 128 kB of program memory. Although the firmware was designed with portability in mind, these constraints influenced the way its components were implemented.

Constants in program memory

AVR-Libc provides macros and `typedefs` to support storing constant data objects in program memory to save limited RAM resources[2]. To allow for compiling and running the code using this feature on other platforms, a subset of macros and `typedefs` was chosen as a common interface, which for AVR uses AVR-Libc's definitions, and for other platforms may either use their own specific definitions, or emulate similar behaviour on normal data memory. In further text, functions that work on data in program memory have the `_P` suffix, and string literals in program memory are declared using the `S()` macro, e.g. `printf_P(S("Hello, world!"))`.

Dynamic allocation

Dynamic allocation of memory on the heap is generally non-deterministic, and in some cases fragmentation may lead to lack of possibility to allocate another block of memory, even if total amount of free memory could theoretically be sufficient to accomplish that [38]. Thus, the firmware does not use dynamic allocation at all, managing the objects lifetime by allocating them either statically or on stack.

[2]Another problem may arise from the fact that toolchain for AVR does not support optimizing identical constants to a single instance (`-f-merge-constants` option of `gcc`), even if they occur in the same compilation unit. The compiled and linked firmware eventually appeared to include multiple repetitions of identical strings, but since it safely fitted in the ATxmega flash memory, no solution for this was applied.

3.3.3 Error handling

The error handling convention adopted is very simple. All functions of which it is known that they may fail, return the object of type `result_t`, which is simply a pointer to an immutable (program-memory) string. If the function succeeds, it returns the `RESULT_OK` constant, defined as `((result_t)NULL)`. Otherwise, it returns a program-memory string containing an error message in the form that may be presented to the user. Listing 3.1 shows an example of this mechanism. It is also possible to catch an error and perform actions dependent on its type, for example by declaring global zero-length string constants and comparing their addresses. This haven't found application in the firmware, though.

```
1  #include "types.h" // result_t, RESULT_OK, S()
2  result_t divmod(int a, int b, int *quot, int *rem) {
3      if (b == 0)
4          return S("Division by zero");
5      if (quot == NULL || rem == NULL)
6          return S("Null pointer");
7      *quot = a / b; *rem = a % b;
8      return RESULT_OK;
9  }
10
11 int main() {
12     result_t res = divmod(1, 3, NULL, NULL);
13     if (res != RESULT_OK) {
14         fprintf(stderr, "error: %s", res);
15         return 1;
16     }
17     return 0;
18 }
```

Listing 3.1: Example of the error handling convention used in the firmware source code.

3.3.4 Object-oriented style

Although the code was written in C, some concepts of object-oriented design were employed in it, to make it easier to model and document.

As a means of decoupling software modules from each other, interfaces are used. An interface is defined by declaring a `struct` containing pointers to functions (methods) and a `void` pointer to the implementation-specific data. Each function accepts the data pointer as its first argument, which is effectively a rough equivalent of pointer or reference to `this` or `self` object in object-oriented languages.

Listing 3.2 shows an example of such declaration (the `Stream` interface and its implementations are further described in Section 3.7). An implementation should include a constructor, i.e. a function with the name conventionally ending with `_Init`, that not only intializes the data object, but also sets the function pointer members of the interface structure, as in Listing 3.3.

```c
typedef struct Stream_struct {
    void *pObj;
    result_t (*read)  (void *pObj, void *buf, size_t length, size_t *rdlength);
    result_t (*write) (void *pObj, const void *buf, size_t length, size_t *wrlength);
    result_t (*flush) (void *pObj);
    result_t (*close) (void *pObj);
}
Stream;
```

Listing 3.2: Example of the interface declaration convention used in the firmware source code.

```c
// FileStream - implements Stream interface for C standard library's FILE*
#include <stdio.h>

result_t FileStream_Read(void *pObj, void *buf, size_t length, size_t *rdlength) {
    size_t r = fread(buf, 1, length, (FILE *)pObj);
    if (rdlength)
        *rdlength = r;
    if (r != length && ferror((FILE *)pObj))
        return S("Error reading from file");
    return RESULT_OK;
}

// and so on for FileStream_Write(), FileStream_Flush(), FileStream_Close()...

// "constructor"
result_t FileStream_Init(Stream *pStream, FILE *fp) {
    pStream->pObj = (void *)fp;
    pStream->read = &FileStream_Read;
    pStream->write = &FileStream_Write;
    pStream->flush = &FileStream_Flush;
    pStream->close = &FileStream_Close;
    return RESULT_OK;
}
```

Listing 3.3: Example of the convention of implementing interfaces in the firmware source code. The declaration of the implemented interface `Stream` is shown in Listing 3.2.

Herb Sutter in [39] points out that functions defined by an interface may fulfill a double role. Apart from the category of functions that serve as customization points, i.e. they are meant to be implemented later, there may also be functions indended for use by client code. The latter often use the Template Method pattern [40] and call the abstract primitives when executing more advanced algorithms, as does the `Stream_Printf()` function in Listing 3.4. In C, this approach has an additional advantage in that it may be used to unify all calls to abstract member functions and save some typing (especially pointer dereferencing, which decreases code readability). In the example, this is done by functions `Stream_Close()`, `Stream_Read()` and `Stream_Write()`. The latter two also check the validity of the call. This way of defining interfaces is known among C++ and D programmers as the Non-Virtual Interface (NVI) idiom [39][41].

```
static inline
result_t Stream_Read(Stream *pStream, void *buf, size_t length, size_t *rdlength) {
    if (!pStream->read) // implementing read is optional
        return S("Stream not readable");
    return (*pStream->read)(pStream->pObj, buf, length, rdlength);
}

static inline
result_t Stream_Write(Stream *pStream, const void *buf, size_t length, size_t *wrlength) {
    if (!pStream->write) // implementing write is optional
        return S("Stream not writable");
    return (*pStream->write)(pStream->pObj, buf, length, wrlength);
}

static inline result_t Stream_Close(Stream *pStream) {
    return (*pStream->close)(pStream->pObj);
}

#include <stdarg.h>
result_t Stream_Printf(Stream *pStream, const char *fmt, ...) {
    char buffer[MAX_PRINTF_BUF];
    int res;
    va_list ap;
    va_start(ap, fmt);
    res = vsnprintf(buffer, MAX_PRINTF_BUF, fmt, ap);
    va_end(ap);
    if (res >= MAX_PRINTF_BUF)
        return S("Buffer overflow");
    if (res < 0)
        return S("Write error");
    return Stream_Write(pStream, pStream->printfbuf, pStream->printfbufPos, NULL);
}
```

Listing 3.4: The Non-Virtual Interface idiom applied to the `Stream` interface from Listing 3.2.

3.4 Build system and directory structure

3.4.1 Requirements

Several other applications were built besides the readout board firmware. They included mostly small programs to test individual components, like a program to measure performance of UART driver in multiple tasks, or automated unit tests (running on PC) described in Section 3.5.

Therefore, the build system had to provide a consistent and easy way to put different sets of components together and build them for different platforms.

3.4.2 Definitions

The build system is organized around building *configurations*[3]. A configuration is a set of software *modules* compiled for a particular *platform* and linked into a single file that may be loaded and executed under an operating system or programmed as firmware of a microcontroller.

[3]The idea was somewhat influenced by the U-Boot build system (see Section 5.6), which in turn borrowed it from the Linux kernel.

A module may be either an *application*, having a single default entry point, or a *library*, which only presents a programming interface to be used by applications or other libraries. Both applications and libraries may depend on other libraries, but no other module will usually depend on an application.

Differences between platforms usually exist at different levels, depending on operating system, CPU architecture and peripherals installed on a board or in a microcontroller. For example, the same microcontroller might be installed on two different boards, each one having different set of external peripherals. In this case, they could share drivers for peripherals integrated in the microcontroller, but have different drivers for their own specific devices. A set of source and header files, makefiles and linker scripts specific to a particular platform will be referred to as *port*.

3.4.3 Makefiles

GNU Make [42] is used as the only build tool. Despite many other tools exist, being either replacements for Make or makefile generators, features of Make itself are sufficient to create a scalable build system fulfilling the requirements. It is also well-documented and available on most modern build platforms, and most programmers should be familiar with it.

Following the advice in [43], there is only a single top-level `Makefile`[4], in which all common rules, definitions and templates are placed. Other makefiles, specific to individual configurations, modules or ports, are included conditionally by the top-level `Makefile`, either directly or indirectly.

This approach has several advantages over (more commonly used) recursively calling Make for modules placed in subdirectories and having their own complete makefiles:

- Easier control over dependencies, as Make always creates the complete dependency graph. In a structure with recursive makefiles, updating dependencies in a module required by other module(s) involved also manual updating dependencies in the latter. This would likely lead to incomplete dependency graphs or to rebuilding some modules always, only because something *could* change in required modules.
- Increased efficiency, because textual inclusion of other makefiles in a single process is usually less expensive than creating new processes for each module, often just to realize that there is nothing to rebuild.

`eval` and `call` functions of GNU Make are used extensively in order to reduce repetitiveness and make the creation of new configurations and modules as easy as writing a single line containing the name of the module or configuration and list of dependencies.

[4]Content of the top-level `Makefile` is included for reference in Appendix A.1.1

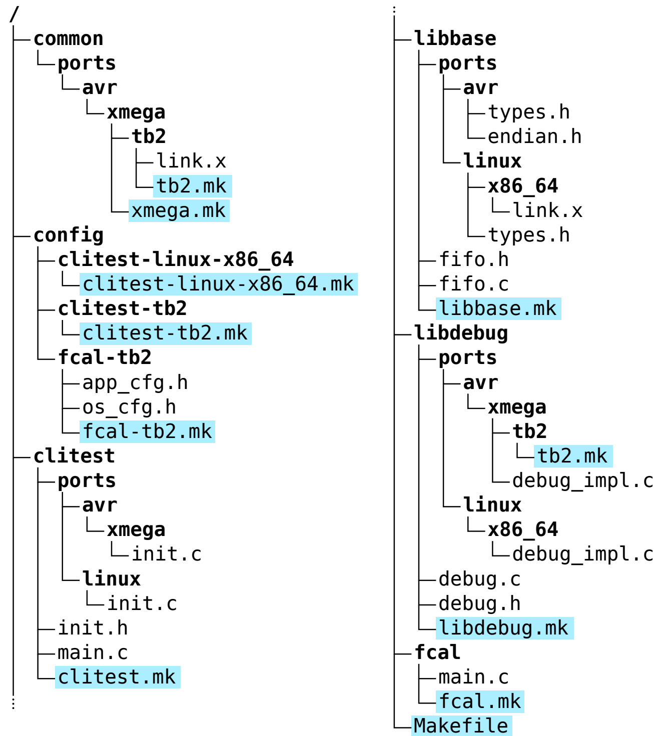


Figure 3.2: Subset of the source directory tree of the microcontroller firmware used in the explanation of the concepts of the build system.

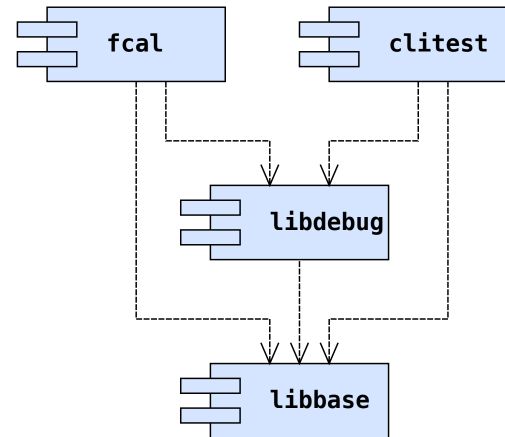


Figure 3.3: Dependencies between modules in the tree from Figure 3.2.

3.4.4 Modules and ports

The original directory hierarchy of the developed firmware is rather complex, so a small subset of it, shown in Figure 3.2, will be used to explain the build system concepts. In this subset of sources, there are two applications, `fcal` and `clitest`, and two libraries, `libbase` and `libdebug`. The dependencies between them are shown in UML 1.4 component diagram in Figure 3.3.

For each module, there is a separate directory in the root of source tree. Source and header files common to all platforms are placed directly in module's directory, whereas files specific to particular platforms are located under appropriate path inside `ports` directory.

Applications in this example may be built for two target platforms: `avr/xmega/tb2` (`tb2` is the name given to the current readout board in the source directory hierarchy) and `linux/x86_64`. Hierarchy of platforms may be defined arbitrarily, based on how particular source files or options may be reused for different groups of platforms. The directory hierarchy follows the general pattern *Operating System*/*CPU*/*System-on-a-Chip*/*Board*, where levels, at which there are no differences, are omitted. The only requirement is that the hierarchy must be consistent across all modules.

Differences between ports may be expressed in a number of ways. The `clitest` application contains header file `init.h`, defining a common interface that is implemented differently in `avr/xmega/init.c` and `linux/init.c`. Similar pattern is applied in the debug console, but part of the implementation (`debug.c`) is common for all platforms. This may be thought as a link-

time variant of the Strategy design pattern [40], where a single implementation is chosen based on its location in a particular subdirectory related to a hardware platform, instead of supplying implementations during runtime.

In `libbase`, the header file `types.h` is different in each port, allowing for tuning macro and type definitions to individual platforms. Real-time kernel abstraction (Section 3.6) also uses this pattern.

Each module directory contains also its own makefile, in which the only required part is the specification of the module name and the list of other modules that it depends on. Following the dependencies depicted in Figure 3.3, the makefiles for the example modules would include the following declarations:

```
# clitest/clitest.mk
$(eval $(call ef_app_template,clitest,libdebug libbase))
# fcal/fcal.mk
$(eval $(call ef_app_template,fcal,libdebug libbase))
# libbase/libbase.mk
$(eval $(call ef_library_template,libbase,))
# libdebug/libdebug.mk
$(eval $(call ef_library_template,libdebug,libbase))
```

Platform-specific options for Make, compiler, linker or assembler may be specified using makefiles located in port directories.

`ef_app_template` and `ef_library_template` are functions defined in the top-level `Makefile`. They produce rules to build the application or library specified as their first argument out of all source files under the directory of the module, automatically determining dependencies between source files and headers using the C preprocessor. They also include makefiles for all required libraries, as well as generate list of include directories for the compiler and list of libraries to the linker, based on the list of dependencies supplied in the second argument.

The directory `common` contains port-specific makefiles and linker scripts, which are used for all configurations and software modules.

3.4.5 Configurations

Valid configurations are listed in the top-level `Makefile` in a similar way modules are defined in their respective makefiles. Some of the defined configurations include:

```
bsuf := $(BUILD_OS)-$(BUILD_CPU)
bport := $(BUILD_OS) $(BUILD_CPU)
$(eval $(call ef_configuration_template, clitest-$(bsuf),     , clitest, $(bport)))
$(eval $(call ef_configuration_template, clitest-tb2    , avr-, clitest, avr xmega tb2))
$(eval $(call ef_configuration_template, fcal-tb2       , avr-, fcal    , avr xmega tb2))
```

The first argument to `ef_configuration_template` is the name of the configuration, the second one specifies toolchain prefix, the third one gives the name of the application to build,

and the fourth one specifies port. Variables `BUILD_OS` and `BUILD_CPU` are set automatically by the top-level `Makefile`, and they are used to configure the compiled application for the same platform on which it is built, which is the case for unit test programs (Section 3.5). Depending on the architecture of build machine, the name of the first configuration might be expanded for instance to `clitest-linux-x86_64`, and the port to `linux x86_64`.

The `ef_configuration_template` function does several things. It adds the configuration to the list of configurations displayed if Make is invoked without the default configuration selected before. It also produces a `config-`*`xxx`* target, which is used to set the default configuration, for example:

```
$ make config-fcal-tb2
```

The action for this target is to generate the file `genconfig.mk`, which is included in the top-level `Makefile` on next invocation of Make, and tells it to include the sub-makefile of the appropriate configuration, if one is available in `config/`*`config_name`* directory, and the makefile of the application to be built. The `ef_configuration_template` function also sets the toolchain prefix and port to be used in subsequent builds.

Invoking Make when the build is configured (i.e. file `genconfig.mk` exists) will cause it to include the application's makefile, involving also inclusion of makefiles of all required libraries, and then build the applications following the rules and options collected from all included makefiles.

Each configuration has its own directory `build/`*`config_name`*, to which produced object and library files are written. It is called the *build directory*, and is separated from the directories containing the source files. This allows for switching between configurations without the need to rebuild all objects, saving the time for example when unit tests are built and run alternately with the proper firmware.

3.4.6 Build options

Several options may be supplied by the user when invoking Make:

`VERBOSE=1` Normally, the exact actions executed by Make are hidden, and only abbreviated name of action followed by the path to the updated file is printed, e.g.:

```
DEP  libbase/fifo.c.dep
CC   libbase/fifo.o
AR   libbase/libbase.a
LD   fcal/fcal.exe
```

This makes any warnings or errors reported by invoked tools easy to notice. However, knowing the complete command line with all options passed to the tool may be useful in finding the source of potential problems, and this option makes it possible.

`TRACE=1` Causes Make to display names of included makefiles. Useful to trace the build system execution when adding new modules or refactoring existing ones.

`BUILDDIR=`*`path`* Changes build directory to given *`path`*. Useful if one does not want or has no rights to write under the source directory.

3.5 Unit testing

3.5.1 Motivation

Writing unit tests has been recognized a good programming practice, especially in agile and test-driven methodologies, where it is even obligatory. Unit tests will probably never ensure that software is entirely error-free, but have a set of other advantages instead:

- Modules are better isolated and their interfaces are more carefully thought out, which also improves the design and architecture of the whole developed system.
- Extended or refactored modules may be immediately tested for potential regression, and quickly reverted or fixed in case changes thought as improvement actually introduced a bug.
- Source code is equipped with unambiguous, up-to-date documentation showing its possible use cases [44].

Unit tests have another advantage in the context of writing software for embedded systems. They allow to test parts of code if the hardware is not available, either because it is in use by other developers, or it simply does not exist yet. And even if testing on the target hardware is possible, it may be faster to simply run a test script on the build machine, than wait until the microcontroller is programmed with updated test code and finishes executing it using its limited resources.

3.5.2 Implementation

Although many unit testing frameworks have been developed for C language[5], one may find the built-in language and standard library features sufficient to implement unit tests, saving time which otherwise would be spent trying some, if not all, possible choices, deciding which one to use, and then learning the new tool.

The implementation, inspired by the way unit tests are done in the D programming language, was simple and straightforward. In D, `unittest` statement allows for defining functions that are executed before the `main()` function is called, provided that the program is compiled with `-unittest` compiler switch. Body of such functions may contain any valid D code, but most

[5] At the time of this writing, Wikipedia lists as many as 36[45].

often it simply initializes an object or objects, calls the tested functions, and examines whether the defined contracts are satisfied, usually by means of the `assert()` expressions [41]. A similar technique is also feasible in C, using its preprocessor for conditional compilation and the `assert()` macro for checking.

```
1  int add(int x, int y) {
2      return x + y;
3  }
4  #ifdef UNITTEST
5  #include <assert.h>
6  void testAdd(void) {
7      assert(add(3, 4) == 7);
8      assert(add(-2, 0) == -2);
9  }
10 #endif // UNITTEST
```

Listing 3.5: An example illustrating the convention of implementing unit tests in the firmware source code.

A trivial example is shown in Listing 3.5, suppose it is a source file from a library called `libarithm`. To compile and run the unit test, a trivial application, say `unittest_libarithm`, would be created, that does nothing but calling `testAdd()` function from the library. The application's makefile would contain the following lines:

```
# compile in all unit tests
CPPFLAGS += -DUNITTEST
# link module under test into executable
$(eval $(call ef_app_template,unittest_libarithm,libarithm))
```

A configuration would be added to the top-level `Makefile`:

```
$(eval $(call ef_configuration_template, unittest_libarithm-$(BUILD_OS)-$(BUILD_CPU), ,\
                                         unittest_libarithm, $(BUILD_OS) $(BUILD_CPU)))
```

Finally the test application would be configured, built and executed:

```
$ make config-unittest_libarithm-linux-x86_64
$ make
$ ./build/unittest_libarithm-linux-x86_64/unittest_libarithm/unittest_libarithm
```

A simple shell script was created to automate building and running unittests, which only takes list of modules to test and options affecting Make execution, for example:

```
$ ./test.sh libbase libcli --clean --verbose
```

```
All tests passed
Coverage for ./build/unittest_libcli-linux-x86_64/unittest_libcli/unittest_libcli
File 'libcli/cmdarg_parsestring.c'
Lines executed: 100.00% of 52
libcli/cmdarg_parsestring.c:creating 'cmdarg_parsestring.c.gcov'

File 'libcli/cmdproc.c'
Lines executed: 96.82% of 440
libcli/cmdproc.c:creating 'cmdproc.c.gcov'

File 'libcli/cmdarg.h'
Lines executed: 100.00% of 4
libcli/cmdarg.h:creating 'cmdarg.h.gcov'

File 'libcli/editor.c'
Lines executed: 1.62% of 185
libcli/editor.c:creating 'editor.c.gcov'
```

Figure 3.4: Example results of test coverage analysis with gcov. Clearly, more tests should be written for `editor.c`.

One may observe that the `assert()` approach might be too restrictive, stopping immediately on first failed test and not allowing other tests to execute. In D, the problem could be solved by overriding behaviour of assert implemented in standard library. Here, similar result could be achieved by providing different definition of `assert()` macro, for example one that collects assertion results in a log file.

3.5.3 Code coverage

GCC provides also an easy way to measure amount of code covered by unit tests (number of times each line, including unit test code itself, gets executed, to be precise).

Code coverage analysis must be enabled by telling the compiler and linker to do so:

```
CFLAGS    += -fprofile-arcs -ftest-coverage
LDLIBS    += gcov
```

Now, running the test application would produce also a summary of coverage analysis, such as one presented in Figure 3.4.

From the generated `*.gcov` files one may discover which lines of code are executed and which are not, and think of writing tests that cover the latter. Also number of executions is counted, giving a basic idea for possible optimizations[6].

One must also be aware of the fact, that code coverage rate may be defined using a number of different criteria. Some pitfalls also exist, such as the trivial case with both branches of conditional instruction in a single line, that counts as executed, even if only one branch is ever executed.

[6]Detailed analysis is done using a profiler, but it is not described in this thesis.

3.6 Multitasking

During the analysis of requirements, three separate tasks were clearly identified, and the data acquisition task was expected to require to be allowed to execute immediately when an event to process appeared. To implement these tasks in a systematic way, a decision was made to use a real-time operating system (RTOS). The uC/OS-II real-time kernel was chosen first. It provides fixed priority pre-emptive scheduling, which means that it ensures that the processor always executes the highest priority task of all tasks which are ready to run [46]. Other characteristics of uC/OS-II include:

- It does not require purchasing license for non-commercial research and educational purposes.
- Full source code is available, including port for ATxmega.
- Features which are not used can be easily disabled, reducing usage of both code and data memory.

A *port* contains functions and macros required to save and restore the execution context when switching between tasks, as well as when entering an interrupt handler and returning from it. The ATxmega port was complete, but required removing unneeded features and fixing a small, but serious bug in functions that save and restore CPU status register on critical section entry and exit, respectively.

The multitasking support was arranged in two libraries. The first one, `libmt`, contains only multitasking kernel code and it does not depend on any hardware drivers. The second library, `libmtbsp`, is a *board support package*, which carries out any necessary initializations. For instance, the ATxmega port of this library initializes timer interrupt to provide system *tick*, enables interrupts, and passes control to the kernel, starting main task of the application.

3.6.1 Kernel abstraction

A simple abstraction was added for multitasking, in order to make drivers independent of any particular kernel. Some of them, such as DMA and serial port driver, use interrupts, which may be handled differently depending on the kernel. For example, serial port driver for ATxmega provides blocking I/O. If an interrupt caused by reception of some data occurs while CPU executes a low-priority task, proper handling of this interrupt should include calling scheduler at interrupt exit to pass control to the higher-priority task waiting for the data received. A way to call scheduler may be similar in principle, but will likely differ in implementation among different kernels. The blocking I/O in that driver is implemented based on semaphores, which also have seen variety of interfaces.

Another advantage of such abstraction is that multitasking kernel may be easily replaced, if the license of the currently used kernel changes or expires, or a smaller and/or faster kernel can be used.

The abstraction is provided as a C header file, `mt.h`, containing a set of macros calling the proper kernel-specific implementation. Therefore, it brings in little or no runtime overhead at all. Summary of macros defined in `mt.h` (or port-specific files included in it) is presented in Table 3.1. All services are optional and implementing them depends on kernel capabilities and actual needs. In the future, macros to create tasks or send and receive messages may be added if necessary.

Table 3.1: Macros defined in `mt.h`, implementing the real-time kernel abstraction layer used to make components of the firmware independent of a particular kernel.

Name	Description
`MT_ATOMIC_EXPR(expr)`	Atomic execution of an expression.
`MT_SEM_DECLARE(sem)`	Declare semaphore.
`MT_SEM_INIT(sem, value)`	Initialize semaphore.
`MT_SEM_PEND(sem, timeout)`	Acquire resource associated with the semaphore, possibly waiting until the resource is available.
`MT_SEM_POST(sem)`	Release resource.
`MT_SLEEPMS(ms)`	Wait specified number of miliseconds, possibly yielding the CPU to other tasks.
`MT_ISR(name)`	Register interrupt. Instance of this macro should be followed by the body interrupt handler function.

Both libraries (`libmt` and `libmtbsp`) accept a makefile-level configuration variable `cfg_libmt_kernel`, which is set to name of the kernel to be used in application. The variable selects a subdirectory under `libmt` and `libmtbsp` containing sources and headers for appropriate kernel. If the variable is not defined, default value `nomt` is assumed, so that no multitasking kernel is used at all. Some services may still be available, though. Interrupts may use objects that are also in use by main flow of code (in particular, thay may post semaphores for which the main program is waiting to acquire), so concepts of forcing atomic execution of code blocks and waiting for resource availability are still valid in seemingly single-threaded applications.

3.7 Input/Output

3.7.1 I/O streams

A simple I/O abstraction was needed to make the firmware tasks independent of any particular hardware. In normal operation, they communicate with PC using ATxmega UARTs, but for unit tests they should rather use buffers in memory, and ports for other platforms may use TCP connections, or even custom developed I/O devices.

The abstraction provided in `stdio.h` header file in the C standard library, i.e. the `FILE` structure and functions for manipulating files associated with `FILE` instances, is not quite sufficient. Although its implementation in AVR-Libc allows (or rather requires) supplying custom device drivers to deal with target system-specific I/O capabilities and requirements [38], the way of doing it is not a part of C standard library and therefore is not portable. Moreover, in AVR-Libc the way to provide custom stream implementations is by pointers to functions transferring single bytes. With such scheme it would be difficult to meet the requirement of fast transmission of relatively large blocks of data at once imposed by the data acquisition task.

Therefore, despite reinventing the wheel is usually a bad practice, a different I/O layer was created, being a compromise between simplicity, little overhead for large blocks of data, and extensibility.

The `Stream` class[7] has four abstract methods:

`Read()` Reads exactly the expected number of bytes, or returns with end-of-file or error condition.

`Write()` Initiates transfer of given data block to the output stream.

`Flush()` Returns only when all data requested to write by previous calls to `Write()` have finished.

`Close()` Closes the stream.

Non-abstract function `Printf_P` provides formatted output based on `Write()`, without referring to any particular implementation.

Behaviour of a `Stream` may be controlled using flags set during its initialization. Currently, only one flag is used, the `STREAM_MODE_TEXT` flag, which tells the `Printf_P` to precede every line feed control character in written text with carriage return character, which may be required by some terminals.

The `Stream` class and its implementations are shown in Figure 3.5. The `FifoStream` class uses a byte FIFO structure in memory as underlying stream, and was particularly useful in writing unit tests. `FDStream` wraps a Unix file descriptor (or a network socket) so that it can be used with functions that expect a `Stream`. `FILEStream` does the same with a C standard library stream (i.e.

[7] It is not a pure interface, because it also includes member variables for mode of end-of-line character conversion, as well as a per-object buffer for `Printf` function.

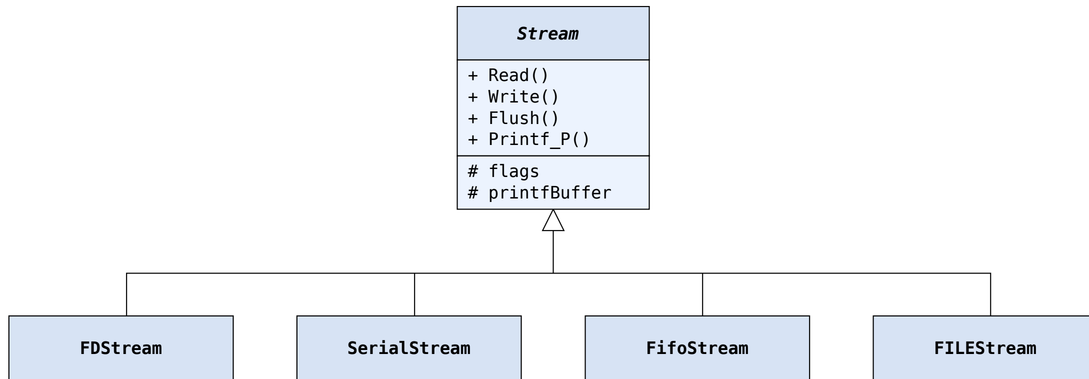


Figure 3.5: The `Stream` class and its implementations.

`FILE*`). Finally, `SerialStream` uses the common UART driver interface (see Section 3.7.3) to provide a stream associated with a serial port.

3.7.2 Debug console

To allow for displaying important error messages and warnings, as well as tracing program flow during debugging, a library providing the debugging and logging console was implemented. It was designed to be as simple and have as few dependencies as possible, so that other parts of code could be traced with it, and the chance that bugs in other parts of code could make it defective was minimized.

The only parts of the implementation that are platform-dependent are initialization and writing one character to the debug console. For ATxmega, the initialization consists in setting parameters of UART, which the library was configured to work with, whereas writing is implemented as waiting in a polling loop until the data register can accept new character and the writing to the register. This is highly inefficient, but works even if interrupts and DMA are disabled (intentionally or by accident) or code that is meant to handle them is broken. Implementation for Linux uses `stderr` to print out log messages.

Every message printed to the debug console belongs to one of five levels of logging, from severe error to program execution tracing. The minimum level of importance of messages to be printed is selected by setting the preprocessor macro `CFG_DEBUG_LEVEL`. Instructions printing the disabled messages are eliminated by the preprocessor, so that disabling debug console ensures that it does not affect firmware performance.

3.7.3 Serial port driver

The serial port driver consists of a platform-independent interface and a platform-specific implementation. What is currently included in both the interface and implementation is only what was needed by the firmware tasks, i.e. writing blocks of data (`Serial_Write()`) and reading single bytes (`Serial_Getc()`).

Implementation was written only for ATxmega and it uses interrupts and a software FIFO to receive bytes, and DMA and interrupts to write blocks of data. The latter appeared reasonably efficient in transmission of data from the readout chain (Section 3.10).

3.8 Drivers for readout board subsystems

The low-level interface to the ASICs and the data concentrator in its current form was unlikely to be reused in the future. Therefore, the highest useful level of abstraction here was just to wrap common sequences of accesses to I/O registers of all devices inside functions with meaningful names. This section describes mainly the digital interface to dedicated peripherals. Monitoring of their operating conditions by means of analog measurements is covered in Section 3.11.

3.8.1 Data concentrator

As stated before, the data concentrator is implemented in FPGA and makes its registers available to the microcontroller through the external memory interface, so that the CPU can access them using regular data memory read and write instructions.

Those registers were mapped into the AVR memory by means of C structures, using a pattern similar to the one used in the `avr/io.h` header file from AVR-Libc. There are several smaller structures associated with functional subsystems of the data concentrator, and they are all part of one top-level structure, `FCALDC_t`. It is not explicitly instantiated, but a macro is defined instead, in which an absolute address is cast to a pointer to this structure and then dereferenced. This is clarified in Listing 3.6.

On top of that, convenience functions that start and stop data acquisition, enable and disable power for front-end and ADC ASICs, and others were written.

3.8.2 Front-end and ADC ASICs

Front-end ASICs include two stages: preamplifier and shaper, and sensitivity of both stages may be switched independently, by choosing configuration of op-amp feedback circuits. The software simply changes the state of output pins of the microcontroller to achieve that. The front-end driver also uses the ATxmega DACs to set bias current for input transistors.

The ADC ASICs are controlled through a simple digital serial interface, somewhat similar to SPI. Commands to set their mode of operation (normal sampling or one of the test modes) or configure DACs controlling their analog part are transmitted in a loop which shifts bits and sets output pins appropriately. Hardware SPI in the ATxmega could not be used because the commands for the ADCs are of variable length.

The ADC ASICs are switched into the test mode when a command to test the digital interface between them and the data concentrator is issued by the user. In the test mode, the ADCs

```c
#ifndef FCALDC_BASE_ADDRESS
#warning "FCALDC_BASE_ADDRESS not configured. Setting to default value of 0x6000."
#define FCALDC_BASE_ADDRESS 0x6000
#endif

typedef volatile uint8_t reg8_t;
typedef volatile uint16_t reg16_t;
typedef volatile uint32_t reg32_t;
typedef volatile uint64_t reg64_t;

// ...

typedef struct FCALDC_TIME_struct {
    reg8_t  RESET;          // Write to this register causes reset of the timer.
    reg8_t  TAKE;           // Write to this register makes consistent copy of
                            //  TIME in SNAPSHOT.
    reg8_t  reserved[14];
    reg64_t TIME;           // Current value of timer, gives inconsistent read.
    reg64_t SNAPSHOT;       // Last timer snapshot taken.
}
FCALDC_TIME_t;
STATIC_ASSERT(sizeof(FCALDC_TIME_t) == 0x20, misaligned_TIME_registers);

// ...

typedef struct FCALDC_struct {
    reg8_t         reserved0[0x10];
    FCALDC_PWR_t   PWR;       // ASIC power management
    FCALDC_TIME_t  TIME;      // DAQ time (5ns ticks)
    FCALDC_EIO_t   EIO;       // Extra GPIOs
    FCALDC_CLK_t   CCLK;      // ADC clock adjustment
    reg8_t         reserved1[0x2000 - sizeof(FCALDC_PWR_t) - sizeof(FCALDC_TIME_t)
                               - sizeof(FCALDC_EIO_t) - sizeof(FCALDC_CLK_t)];
    FCALDC_DAQ_t   DAQ;       // Trigger settings, external DAQ interface
}
FCALDC_t;

#define FCALDC (*(FCALDC_t *)(FCALDC_BASE_ADDRESS))
```

Listing 3.6: Excerpts from `fcaldc.h` file, which contains mapping of the data concentrator registers to the AVR data address space.

produce predefined patterns (linear or simple pseudo-random sequences). In this mode, the data read from the ADCs through the data concentrator are not transmitted to PC, but they are compared against the expected pattern instead.

3.9 Command line interface

Suggested command protocol for the firmware was a simple exchange of textual messages, where the firmware receives a command, parses and executes it, and optionally prints the response, such as the value of a variable previously set, the value of a quantity just measured or an error message. The command syntax should be familiar to users that are familiar with laboratory devices using SCPI protocol. To make the calibration and configuration easier without building

a complete graphical configuration program, as well as to facilitate testing the hardware and development of the data concentrator, the editor should be an interactive console, providing command completion and history of a few recently used commands.

3.9.1 Implementation overview

The CLI module is implemented as a set of classes coupled by appropriate interfaces. They are shown on the class diagram in Figure 3.6.

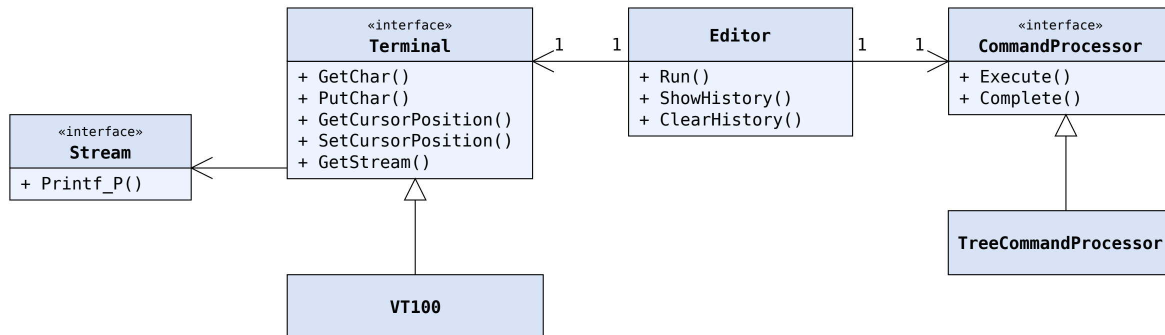


Figure 3.6: Class diagram for command line interface.

The main class is the `Editor` class, which allows for editing a command in a single line and browsing the command history. The `Editor` is independent of command syntax and any particular terminal standard. Command parsing, completion and execution are provided by implementing the `CommandProcessor` interface, whereas the terminal is handled by implementations of the `Terminal` interface. They translate special functions, such as moving the cursor or erasing a part of the current line, to control sequences specific for a given terminal. Currently the only implemented terminal is compatible with a subset of the VT100 terminal commands, based on the description in [47].

The editor displays the prompt (‘> ’) and waits for user input. The user may simply enter the command, which is echoed on the terminal, use control keys to edit the command (left and right arrows, Backspace), browse in command history (up and down arrows) or get a hint on available commands from the editor (the Tab key).

3.9.2 Syntax

The syntax for commands is influenced by the SCPI 1999 standard [48]. Commands are based on the hierarchical structure (tree). Each command is unambiguously identified by its full path in the tree, consisting of names of all nodes starting from the root, separated with ‘.’ (dot) or ‘:’ (colon). At each level of the tree, one node may be a *default* node, which means its name may be skipped in the path. Node name must begin with a letter. Following characters may be letters or digits. Command parser is case-insensitive.

Figure 3.7 shows an example of a command tree and the list of valid commands in this tree. Note that command HISTORY is equivalent to command HISTORY.SHOW, command MON.VDD is equivalent to MON.VDD.READ, and command MON.TEMP is equivalent to MON.TEMP.READ.

```
HISTORY
  [SHOW]
  CLEAR
MON
  VDD
    [READ]
    GAIN
    OFFSET
  TEMP
    [READ]
    GAIN
    OFFSET
```

```
HISTORY
HISTORY.SHOW
HISTORY.CLEAR

MON.VDD
MON.VDD.READ
MON.VDD.GAIN
MON.VDD.OFFSET
MON.TEMP
MON.TEMP.READ
MON.TEMP.GAIN
MON.TEMP.OFFSET
```

Figure 3.7: Example command tree (on the left) and list of valid commands in that tree (on the right).

The command path may be followed by parameters separated with space characters. Most commands follow the convention, in which the command invoked with no parameters reads the current value, while the same command invoked with parameter sets the value. In the above example, the command MON.TEMP might be used to measure and display the current temperature, the command MON.TEMP.GAIN might display the muliplier used to scale the raw value from the ADC to the temperature measured in degrees Celsius, and the command MON.TEMP.GAIN 5.13 might set the current value of that multiplier to 5.13.

3.9.3 Parsing and executing commands

To execute a command, the command processor must first find the function that is mapped to the entered textual command. Mappings are defined as arrays of structures containing the node name, and either a pointer to the function implementing the command, if the node is a leaf node (i.e. an actual command), or a pointer to an array of sub-nodes, otherwise. Details are shown in Listing 3.7.

Structures are arranged into arrays. The Execute() method simply does linear search on each level until it reaches a matching leaf node. It then calls the command function pointed to by that node, passing the remaining part of the entered command line as argument to this function. Algorithm of Execute() in Python-like pseudocode is shown in Listing 3.8.

3.9.4 Link-time composed arrays

Arrays of commands are not necessarily written as a whole in a single piece of source code, but they may be scattered over multiple files and composed during link time. This effectively inverts the dependency between the command entry and array to which it belongs, so that only the

```
// Command implementation are functions of this type.
// The same function may operate on different objects, passed by pObj.
// argstr is the remaining part of command line.
// pOut is a stream to write the command output, if any.
typedef result_t (*Command_func)(void *pObj, const char *argstr, Stream *pOut);

typedef struct CommandDelegate_struct {
    Command_func func; // command implementation
    void *pObj;        // pointer passed as first argument to func
}
CommandDelegate;

typedef struct Command_struct Command;

typedef struct CommandArray_struct {
    Command *commands;       // pointer to first entry in the array
    Command *defaultCommand; // pointer to default entry in the array
                             // or NULL, if no default entry is defined
}
CommandArray;

struct Command_struct {
    immutable_str name;      // command name or NULL pointer for a terminating entry.
    union {
        CommandDelegate command;  // command, if isLeaf == TRUE
        CommandArray subcommands; // array of nodes, if isLeaf == FALSE
    } body;
    bool isLeaf;             // TRUE, if this is actual command
};
```

Listing 3.7: Data types used to define the command tree structure.

```
def Execute(rootCommandArray, commandLine, pOut)
    commandArray = rootCommandArray
    defaultCommand = None
    # split command line to list of node names matching the [A-Z][A-Z0-9]* pattern
    # and separated with '.' or ':', and remaining part to be used as command argument
    # e.g. "MON.VDD.GAIN 123" -> ["MON", "VDD", "GAIN"], "123"
    (names, argstr) = splitCmdLine(commandLine)
    for name in names:
        for node in commandArray:
            if node.name == name:
                if name.isLeaf:
                    return node.func(node.pObj, argstr, pOut)
                else:
                    defaultCommand = node.defaultCommand
                    commandArray = node.commands
                    break
        else:
            raise Error('command ' + name + ' not found')
    if defaultCommand != None:
        return defaultCommand.func(defaultCommand.pObj, argstr, pOut)
    raise Error('command name expected')
```

Listing 3.8: Algorithm to parse and execute a hierarchical command.

programmer who implements the command needs to know where the command must be placed in the command tree, and only one file is compiled when a command is added or removed.

These link-time composed array are implemented by making each array having its own section in object files, and placing the (`static`) instances of the `Command` structure meant to be elements of that array in the appropriate section:

```
#define COMMAND_IN_ARRAY(_arrName) __attribute__((section(".ld_comp_array_" #_arrName "_data")))
Command command_HISTORY_SHOW COMMAND_IN_ARRAY(commandArray_HISTORY) =
{
  // initialize fields...
}
```

An array itself is declared as empty array placed in section with a name ending with `_begin`:

```
#define LD_COMP_ARRAY_DECLARE(_arrName, _type, ...) \
_type __attribute__((section(".ld_comp_array_" #_name "_begin"))) _name[] = { };
```

This causes the linker to assign an address of the first `_data` element to it, when the input sections are placed sorted by name in the `.text` output section in the linker script:

```
  .text   : {
    *(.vectors)
    KEEP(*(.vectors))
    *(SORT_BY_NAME(.ld_comp_array*)) /* place link-time composed arrays in program memory */
    *(.progmem.gcc*)
    *(.progmem*)
    /* ... other code & data sections ... */ }
```

3.9.5 Command completion

Because of memory constraints, the `Complete()` method cannot build a full list of matching completions in a buffer. Instead, it performs internal iteration over all matching completions, calling a supplied delegate for each of them. The algorithm is shown in Listing 3.9.

The editor calls this method when user presses the `Tab` key. If there is only one possible match at the current level, the editor uses it in the current line buffer. If there are more matches, editor lists them in subsequent lines, not changing the current line buffer.

3.9.6 Command history

A command entered in the editor is added to the command history buffer. The oldest commands from the history buffer may be deleted first, until the newly entered command fits entirely.

Browsing the history is done by using arrow keys in the terminal window, just like in most UNIX shells. Implementation simply searches for End-of-Line characters within the buffer to get

```
# Returns number of completions matched
def Complete(rootCommandArray, partialCommand, action)
    # Transform command to canonical form, e.g. "   mon   . vDD.  rE" -> "MON.VDD.RE"
    partialCommand = canonicalize(partialCommand)
    if foundAnyExtraCharacters: # i.e. not being valid separator or part of node name
        return 0
    commandArray = rootCommandArray
    names = splitCmdLine(partialCommand) # e.g. "MON.VDD.RE" -> ["MON", "VDD", "RE"]
    for name in names:
        # Count matching commands at current level, remember the first match, if any.
        (matchCount, firstMatch) = countNamesStartingWith(commandArray, name)
        if matchCount == 0:
            return 0
        else if matchCount == 1:
            if firstMatch.isLeaf:
                action(firstMatch)
                return 1
            else:
                commandArray = firstMatch.commands  # If not a leaf, search deeper
                continue
        else:
            for match in allNamesStartingWith(commandArray, name):
                action(match)
            return matchCount
```

Listing 3.9: Algorithm to iterate over possible completions of given initial part of a command path.

entire lines from it. The editor allows editing commands from the history—when the user starts typing in a line retrieved from the history, the command is copied from the history buffer to the current line buffer. The history buffer can also be displayed in its entirety.

3.10 Data acquisition

Following the scheme described in Section 2.2.2, the FPGA continuously reads the data from the ADC ASICs and stores them in a circular buffer. When a trigger condition is met, the FPGA keeps reading the data until it collects a predefined number of samples following the trigger (the *post-trigger* samples), and then enables the `busy` flag, inhibiting further triggers, and drives one of the input pins of ATxmega high, generating an interrupt in the firmware.

The interrupt routine releases the *wait-for-event* semaphore, on which the DAQ task is pending, and exits immediately. Then, since the DAQ task is given the highest priority in the system, scheduler passes the control directly to the DAQ task, which processes the event data, sends them out and resets the `busy` flag, allowing new triggers to be accepted by the system.

3.10.1 Pipelined buffers

Using the DMA, the serial transmission imposes relatively low load on the CPU, even at high bitrates. This led to the observation that if two buffers were used, processing of the next event could begin immediately after initiating transmission of the current one. If processing took shorter time

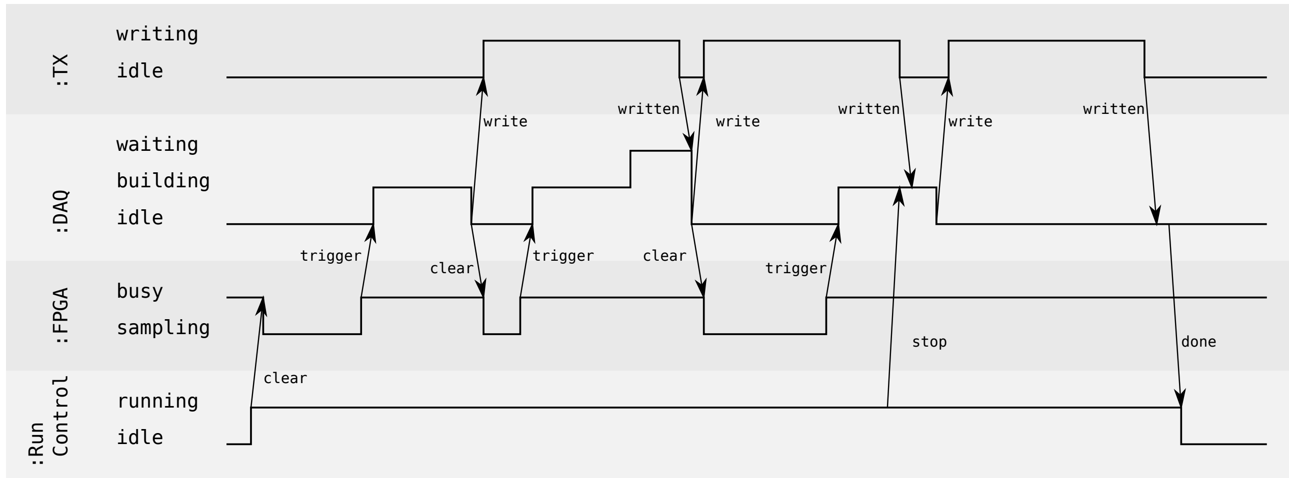


Figure 3.8: UML timing diagram presenting example DAQ session in which 3 events were sent.

than transmission, bus occupation rate could be close to 100%.

Consequently, two buffers were allocated, and a second task (the *TX* task) was created only to transmit the data. The DAQ task now builds the packet in the first buffer, waits until the TX task is complete with sending the contents of the second buffer, and passes the first buffer to the TX task, being now able to receive new data in the second buffer.

Figure 3.8 illustrates this scheme in a session consisting of three events. Starting a run clears the `busy` flag in FPGA, and soon the trigger condition occurs, causing the DAQ task to build the first data packet and pass it to the TX task, being able to process the next event in the second buffer while the first one is being transmitted. When the user requests stopping the run while an event is being processed, processing of that event completes, but the `busy` flag is not reset again when the TX task starts sending out the last event.

3.10.2 Data format

Event data are sent in packets. Each packet contains the data from one event and some metadata attached to it in the packet header. The metadata include the following information:

Sequence number A number incremented by one in each next event. Keeping track of sequence numbers allows for counting events lost due to transmission errors and computing overall event rate.

TLU trigger ID The 15-bit trigger sequence number generated by the Trigger Logic Unit (see Chapter 4) helps in synchronizing data coming from different sensors.

Timestamp With resolution of 5 ns (i.e. the reverse of the ADC clock frequency, equal to 200 MHz) allows for precisely determining intervals between events.

Number of samples How many samples from each channel are contained in the frame.

Channel mask Transmission of data from selected channels may be disabled to increase event rate. This field contains the information about which channels the data in the packet come

```
# seq: 20
# timestamp: 49117231
# smp chn00 chn01 chn02 chn03 chn28 chn29 chn30 chn31
    1   253   249   241   240   431   410   412   414
    2   301   259   238   242   438   421   415   413
    3   285   252   239   248   425   414   410   414
    4   265   253   245   246   430   416   414   414
```

Figure 3.9: Example event with 4 samples from each of 8 selected channels transmitted in text protocol.

from.

Trigger status Contains information about how the event was triggered—either by one of external trigger signals, sample value exceeding the threshold level, or requested by user with an appropriate command.

Two protocols were implemented:

- the text protocol, allowing for taking a quick look at the data without the need to implement and use any special decoder, thus being mainly used during development and calibration;
- the binary protocol, optimized for highest data rate.

3.10.3 Text protocol

In the text protocol, the format of transmitted data was designed so that it could be directly read by `gnuplot` [49]. Metadata are sent in comments (lines starting with '#') preceding the event data. An example of data transmitted for an event in the text protocol is shown in Figure 3.9.

With such protocol, data file can be created in a straightforward manner using `cat`, and plotted with `gnuplot`, for example:

```
$ cat /dev/ttyUSB0 >file.txt
$ gnuplot -e "plot 'file.txt' u 1:2"
```

3.10.4 Binary protocol

The binary protocol used in normal physics runs had to fulfill the following requirements:

- highest possible event rate (within hardware limitations),
- ability to unambiguously recover start and end of each packet in a continuous stream of bytes,
- integrity checking, to avoid accepting packets that were broken during transmission.

Each packet in the binary protocol consists of a series of raw samples from the ADCs preceded by the 30-byte header containing the metadata mentioned above. The structure of the

event packet is shown in Figure 3.10. The meaning of each field has been explained in previous sections.

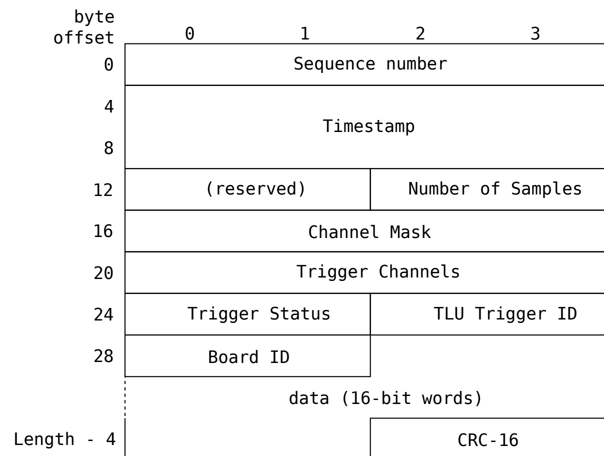


Figure 3.10: Structure of event packet in the binary protocol used to transmit data from the readout board to PC.

Integrity checking is provided by means of the CRC-16 code. Altough the data were transmitted over USB, which has its own mechanisms for ensuring data integrity, employing another layer of checking allowed detecting rare situations in which a relatively part of the data stream was lost, sometimes even causing the received frame to be composed of the beginning of one frame and ending of one of the consecutive frames. This problem could be possibly caused by an overflow in the internal FIFO of the USB-UART bridge, so that USB was carrying a data stream that was already incomplete.

To make the receiver able to detect packet borders, special sequences marking the start of the packet (SOP) and the end of the packet (EOP) were defined. Since any sequence of bytes could appear in the packet header, an additional special sequence (ESC) was needed to prevent decoding such sequences occuring in data as real special sequences.

The defined special sequences are listed in Table 3.2. The choice of special sequences was based on the observation, that in an array of 16-bit unsigned values from 10-bit ADCs, transmitted as series of bytes, six most significant bits of every second byte are always 0s. With special sequences consisting of two bytes, both having at least one of these bits always set, there is no need to scan the data from ADCs for occurrence of sequences of bytes defined as special ones, since they simply cannot occur there. Thus, only packet header needs *escaping*. The `FF FD` sequence is a variant of EOP that accounts for the case in which the last byte of the packet is `FF`.

3.10.5 Class design

The double-buffer DAQ scheme described earlier and the planned possibility to switch between protocols at runtime led to the design presented in Figure 3.11.

Table 3.2: Special sequences defined in the binary protocol used to transmit event data from the readout board to PC, to make the receiver able to unambiguously extract individual packets from a continuous stream of bytes.

bytes (hexadecimal)	name	description
`FF FA`	SOP	Start of Packet
`FF FC`	EOP	End of Packet
`FF FD`	FF,EOP	Single `FF` byte immediately followed by End of Packet
`FF FE xx`	ESC	Escape. Allows special sequences to appear in packet data. Decoded as two-byte sequence: `FF ~xx` (~ is a bitwise not operator, as in C).

An interrupt from the FPGA causes the `Event` structure to be filled with with appropriate data and metadata and passed to the `Prepare()` function of an implementation of the `EventWriter` interface, where preprocessing may take place (for example, `BinaryEventWriter` escapes the header and computes CRC code). The `EventWriter` is associated with an output `Stream`, to which its `Write()` method sends the data in an implementation-specific format.

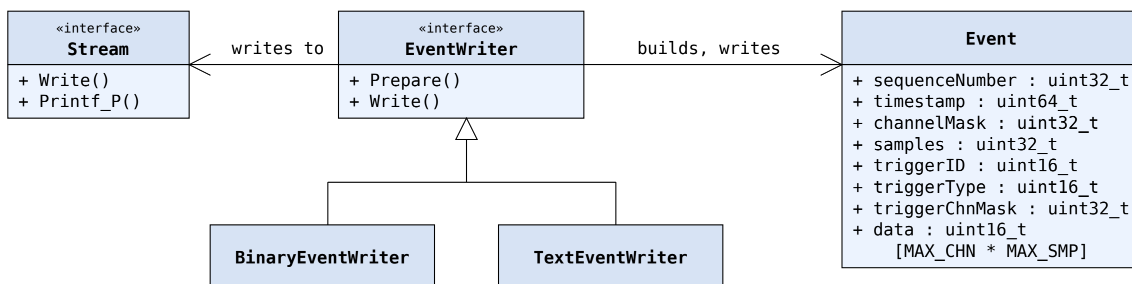


Figure 3.11: Diagram of classes implementing event building and transmission.

3.11 Hardware monitor and power pulsing measurements

3.11.1 Motivation

The firmware is capable of measuring operating conditions of hardware installed on board. The analog inputs that may be monitored include supply voltages and currents on front-end and the ADC ASICs, temperature of the ADC ASICs, as well as voltage and temperature of the microcontroller. The ATxmega includes two 12-bit ADCs. Each of them has an 8-input multiplexer and is capable of sampling with frequency of up to 2 MHz.

During regular physics runs, selected inputs are continuously measured in intervals of several seconds. The results of these measurements are transmitted over a separate channel in the USB

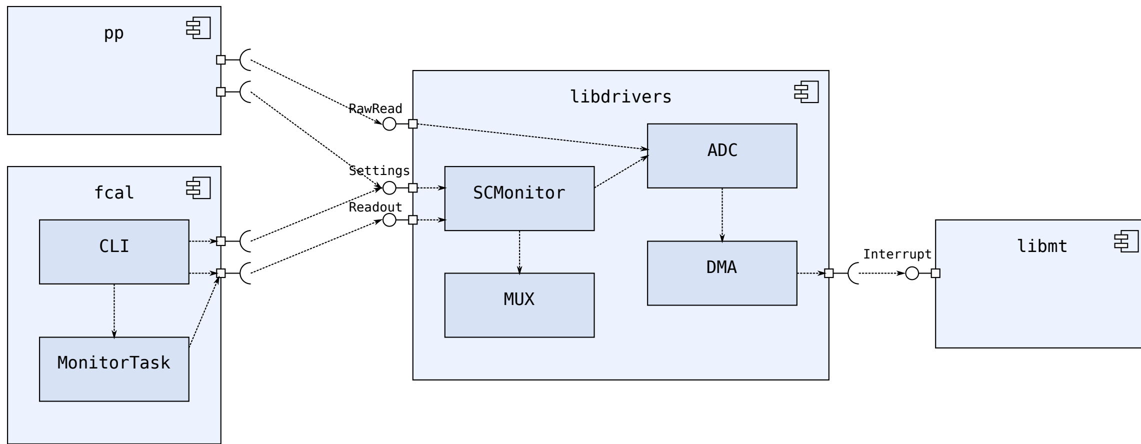


Figure 3.12: Firmware components involved in measuring operating conditions on-board integrated circuits.

connection, and logged on the PC as *tags* along with the data from the sensor. Such tags might help explaining anomalies in data, if they resulted from changing hardware operating conditions. For calibration purposes, firmware allows reading current value from arbitrary analog input upon user request, pausing the monitoring loop if necessary.

Another purpose of analog monitoring features is to test power pulsing capabilities of the ASICs, i.e. measuring the overall power consumption as their power supply is periodically switched on and off, as well as determining the time required after the power is switched on until the ASICs give correct output. For this purpose, another small firmware application named `pp` was created.

3.11.2 Implementation

Diagram of the components implementing the hardware monitoring features is shown in Figure 3.12.

Applying the DMA to the ADC readout allowed for taking arbitrary number of samples in configurable, regular intervals without significant load imposed on the CPU. The ADC driver contains two functions:

`ADC_Init()` Enables and configures both ATxmega ADC blocks.

`ADC_RawRead()` Reads a given number of samples into the supplied buffer. It first configures the internal multiplexers and the clock prescaler, then starts a DMA transaction and waits for an interrupt raised when the transaction is complete.

External analog signals are connected to microcontroller pins via on-board multiplexers and simple RC low-pass filters. Permanent configuration for all measurable analog signals on the readout board is stored in flash memory, and it includes settings for multiplexers (on-board and ATxmega internal), as well as readout mode (differential or single-ended).

Modifiable settings are stored in RAM, with an option to save them in EEPROM. These settings include: time required after switching external multiplexer for the signal to propagate through low-pass filter, number of samples to take at once, gain and offset. The latter three are used to compute average raw ADC output value (reducing impact of noise on readout) and convert it to units that make sense for a human, such as volts, milliamperes or degrees Celsius. The formula for obtaining the final value of measured input is:

$$value = \left(\frac{1}{N_{samples}} \sum_{i=1}^{N_{samples}} s_i \right) \times gain + offset \tag{3.1}$$

Commands related to analog monitoring include reading and setting the parameters mentioned above and readout upon user request, as well as enabling automatic monitoring of the selected input.

Monitor task is simply an endless loop that reads inputs for which automatic readout is enabled and writes results in the text form to the log output.

3.12 Results

3.12.1 DAQ performance

The data acquisition performance was measured by looking at the maximum event rate for given data payload per packet. The tests were carried out with triggers occurring sufficiently often that the time from cleaning the `busy` flag to receiving the next trigger could be ignored.

In these tests, the firmware showed a satisfying efficiency, giving effective data rate of about 378 kB/s at UART baudrate of 4 Mbits/s. This means bus occupation rate of nearly 95% and effective event rate of about 185 events per second, with 32 samples from each of 32 readout channels transmitted for each event. Cutting down the number of samples per event (by reading only selected channels or taking less samples per channel) reduces the time needed to transmit the event data and increases event rate, to the point in which cost of building the packet header and computing CRC causes the transmission of one buffer to complete before the second is ready to be sent. Figure 3.13 shows how the event rate changes with the number of samples per packet. For the minimum number of samples required to fully reconstruct the time and amplitude of registered pulses (i.e. 3 samples) from each of the 32 channels, the event rate is about 1000, i.e. 10 times the minimum required rate.

In real-world systems the effective event rate depends also on the average time from the moment when the `busy` flag is cleared to the next trigger.

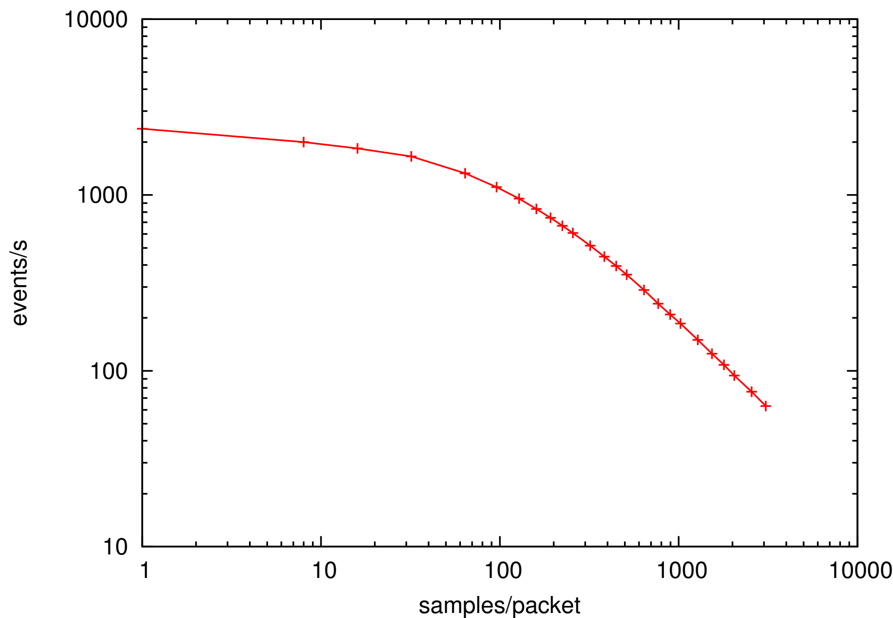


Figure 3.13: Data acquisition performance (number of events per second) as a function of number of samples per packet.

3.12.2 Power pulsing

Example results of measurements performed to evaluate power pulsing capabilities of ADC ASICs are presented in Figure 3.14, which shows how the supply voltages and currents changed in response to switching the power on and off. The ASICs are enabled at $t = 0$. Voltage and current become stable at about $t = 0.8$ ms, and the circuits stay enabled for another 1 ms, i.e. approximate time of a single bunch train in ILC.

This example shows that during normal operation, four 8-channel ADC ICs consume about 1.2 W power. In the stand-by mode, only digital interface is enabled, reducing the power consumption to less than 0.3 W. Given that only 5 such cycles will occur every second, the overall power consumption of ADC block may be reduced by a factor of 4 with the current version of the circuit. The detailed analysis with definite conclusions will be available in [50].

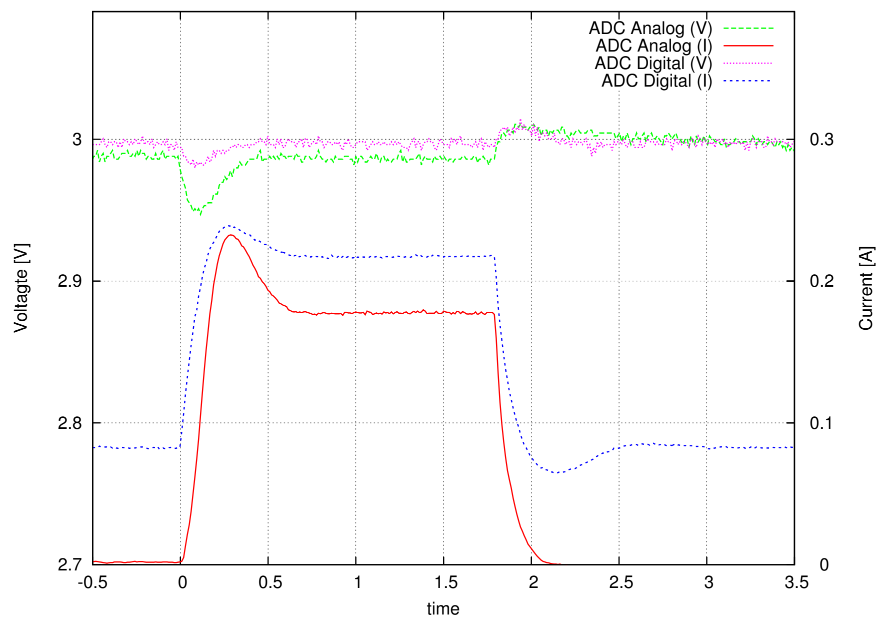


Figure 3.14: Supply voltages and currents drawn by analog and digital parts of ADC ASICs in response to switching the power on and off.

Chapter 4

Integrating with EUDAQ

This chapter describes how the support for the prototype readout board was added to an existing data acquisition framework for high-energy physics measurements—the EUDAQ framework [51]. A brief description of EUDAQ is given first, followed by the specification of requirements and the implementation details of the developed software modules required to read the data from the hardware and convert them to a representation suitable for further analyses. In the last section, some preliminary results are presented.

4.1 EUDAQ and its architecture

EUDAQ is a modular data acquisition framework developed primarily for the EUDET pixel telescope. Its design, however, makes it adaptable to other detectors. It is written in C++[1] [51].

The architecture of EUDAQ is shown in Figure 4.1. It is a set of separate processes communicating with each other over TCP connections, so that they do not necessarily have to be run on the same machine. The processes include:

Run Control Main process, used to control the whole system and monitor its status. All other processes connect to it, accept commands and report their statuses. Its graphical user interface is shown in Figure 4.2.

Log Collector Collects log messages from other processes, stores them in a single file and displays them in a graphical window (also visible in Figure 4.2) or in the text console.

Data Collector Collects data from all Producers, merges them into a single raw data stream, and writes to a disk file.

Producers Each detector has its own Producer process, which connects to the detector, configures it and reads data from it. The EUDAQ package already included Producers for two other detectors in the test beam setup, the Trigger Logic Unit, which generates and counts

[1]At the time of this writing, the EUDAQ source code was publicly available in the SVN repository at `http://svn.hepforge.org/eudaq`

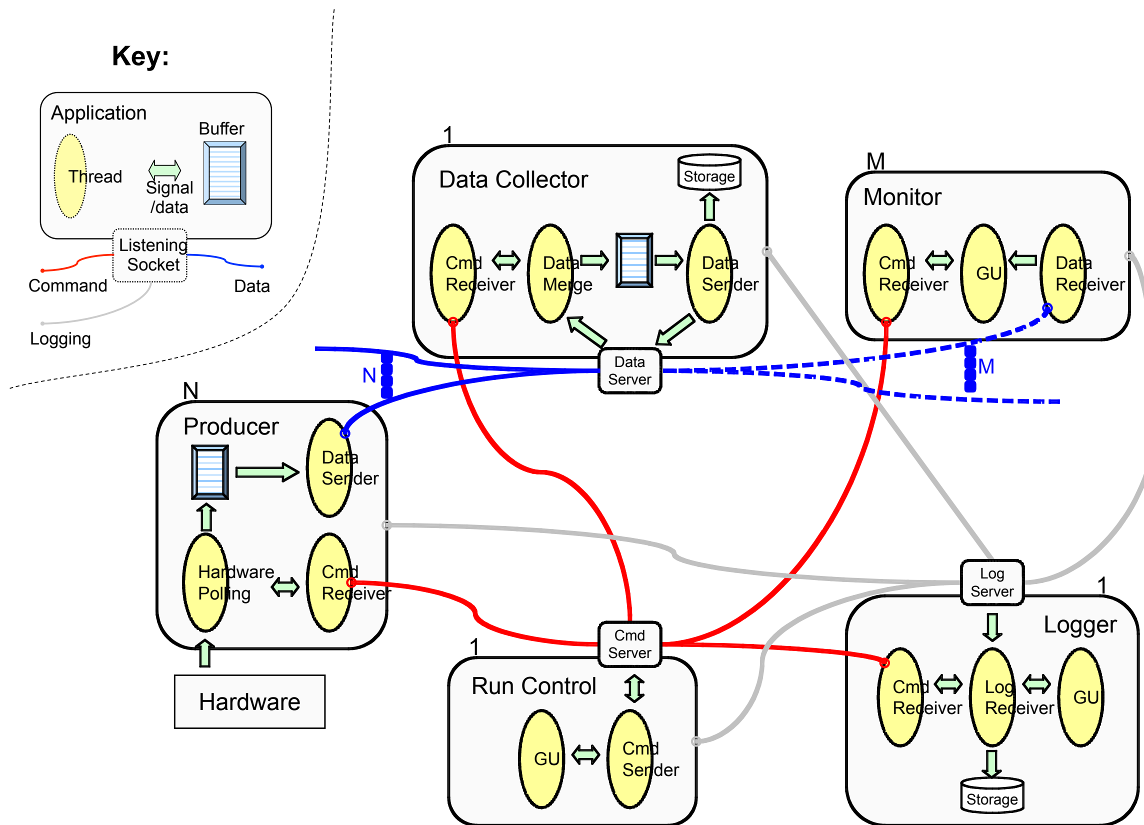


Figure 4.1: Schematic diagram of the EUDAQ architecture. Explanation of the notation is on the top left corner of the diagram. Source: [51].

triggers so that events from all detectors can be synchronized [24], and the MVD telescope, measuring the beam position [23].

Monitors Applications for on-line data display, such as the RootMonitor, which uses the ROOT framework [52] to plot histograms of the data collected previously or in the current run.

Making EUDAQ work with new detector required adding a Producer application, which would get the data from readout board, and a Data Converter Plugin, transforming the raw event data into representation that may be displayed by a Monitor and used for analyses.

4.2 The Producer

4.2.1 Overview

Following the instructions in [51], the Producer application was created by extending the `eudaq::Producer` class. When the application starts, a single object of the derived class `FCALProducer` is created, and its event processing loop called. In this loop, the data comming from the hardware and the commands received from the Run Control are handled.

The FCAL Producer has essentially only one use case, covered by the following procedure:

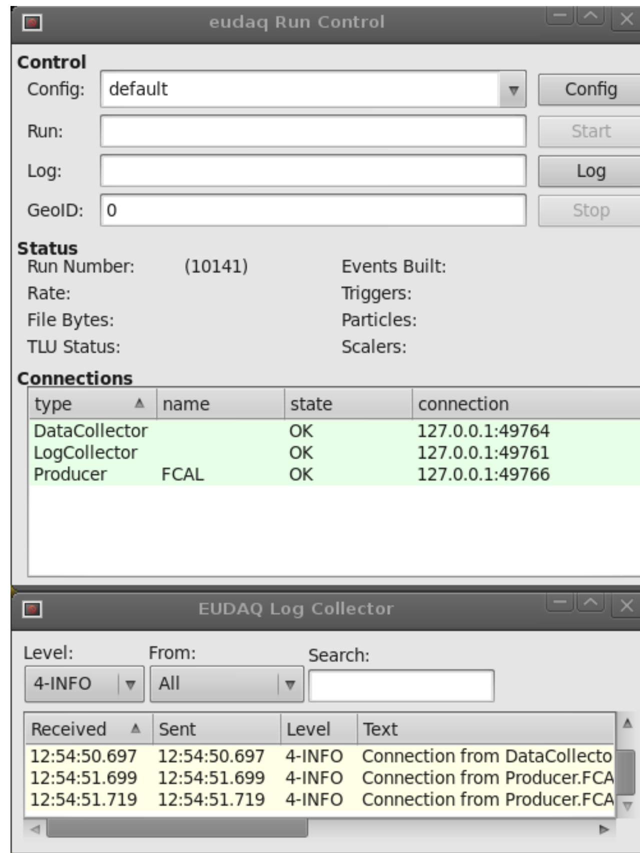


Figure 4.2: Screenshot of graphical interface of the EUDAQ processes: the Run Control and the Log Collector.

1. The user chooses a configuration file in the Run Control window and presses the `Config` button. All other processes, including the FCAL Producer, receive the `Configure` command with parsed contents of the configuration file. The FCAL Producer uses these parameters to connect to the readout board or boards and passes configuration commands to them.
2. The user presses the `Start` button in the Run Control. The FCAL Producer receives `Start Run` command and enables accepting triggers in the hardware.
3. The FCAL Producer processes events from the hardware and transmits the data to the Data Collector, which builds events and saves them to file. The user monitors the statistics of incoming data and decides to stop the run by pressing `Stop` button in the Run Control. The FCAL Producer receives `Stop Run` command and disables data acquisition in the hardware.
4. The user continues from the step 1. or 2., or closes the Run Control, which terminates all other processes, including the FCAL Producer.

The class diagram of the FCAL Producer is shown in Figure 4.3, and the design behind this diagram will be explained in the following paragraphs.

Figure 4.3: Class diagram of the FCAL Producer. Concrete `BoardCommands` and other minor classes are not shown.

4.2.2 The `FCALProducer` main loop

The FCAL Producer handles all requests within the event processing loop (i.e. the `run()` method) of a single object of `io_service` class of the Boost.Asio library [53]. Thanks to Boost.Asio, requests from multiple different sources could be served in a consistent, sequential manner.

Commands from the Run Control process are received in a separate thread, which is created by the `eudaq::Producer` class and maintains a TCP connection with the Run Control. These commands are translated to virtual function calls—including the `OnConfigure()`, `OnStartRun()`, `OnStopRun()` and `OnTerminate()` functions. These functions do not handle the commands directly in that thread. Instead, respective functors are created and `post()`-ed to the `io_service` object, except the `OnTerminate()` function, which calls `io_service::stop()` to break the loop.

Communication with the readout board or boards is done using the `serial_port` class from Boost.Asio. Handlers of Run Control commands initiate asynchronous reads and writes on serial ports, which are subsequently handled within the `io_service` object's event processing loop.

The FCAL Producer provides also a simple command relay, passing commands entered in the Producer console to the readout board.

4.2.3 Configuring the Producer and the readout board

The `OnConfigure()` method of the `FCALProducer` class takes an object of the `eudaq::Configuration` class, being the result of parsing the configuration file selected by the user. The configuration objects contains a set of named sections, and each section is an associative arrays, mapping properties to their values. An example configuration file is shown in Listing 4.1.

All configuration options meant for the FCAL Producer are in the `Producer.FCAL` section. They include the number of connected detector layers (i.e. pairs: readout board and sensor board) and connection parameters for the serial ports. The number of detector layers is practically limited only by the topology and throughput of the USB. There are three ports for each board, named `Cmd`, `Daq`, and `Mon`, for command interface, data acquisition, and monitoring, respectively. Only port names (`Port`) and baud rates (`Baud`) are included in the configuration, and for other parameters, fixed settings are used, i.e. 8 data bits, no parity, 1 stop bit and no flow control.

The configuration options include also commmands to send to the readout boards. Once all ports are opened and configured successfully, all keys in the `Producer.FCAL` section are compared against the `B`*`x`*`.`*`ord`*`.`*`cmd`* pattern, where:

- *`x`* is a sequential board identifier, starting from 0;
- *`ord`* is a number used to place the commands in a fixed order, if it is significant;
- *`cmd`* is a command, which is joined using a single space character with its parameter(s) from the value of the key, and passed directly to the readout board.

```
1  [RunControl]
2  RunSizeLimit = 1000000000
3  NoTrigWarnTime = 10
4
5  [Producer.FCAL]
6  Boards = 1
7  B0.Cmd.Port = /dev/ttyUSB0
8  B0.Cmd.Baud = 115200
9  B0.Daq.Port = /dev/ttyUSB1
10 B0.Daq.Baud = 4000000
11 B0.Mon.Port = /dev/ttyUSB2
12 b0.Mon.Baud = 115200
13
14 B0.10.TRIGGER.RJ45.ENABLE = on
15 B0.20.TRIGGER.LEMO.ENABLE = on
16 B0.30.TRIGGER.CHNS.ENABLE = off
17 B0.40.DAQ.SMPS = 32
18 B0.50.DAQ.POST = 12
```

Listing 4.1: EUDAQ configuration file containing options for the main EUDAQ process, the Run Control, as well as connection settings for the readout board and commands to send to it.

The commands sent to the readout board are encapsulated in implementations of the `BoardCommand` abstract class, objects of which are scheduled for execution in a queue maintained by the `BoardConnection` class. A command added to the queue contains a string with command to execute, and a timeout in seconds, within which the response should be received. The `BoardConnection` class reads the response asynchronously, and calls one of the three methods of the `BoardCommand` class instance, depending on the result of the operation:

`OnResponse()` is called when a complete response is received, i.e. the end-of-line character followed by the command prompt ("> ") is encountered.

`OnTimeout()` is called if nothing has been read from the command port of the board within the specified timeout period.

`OnError()` is called if an error occurs.

The above algorithm is summarized in the sequence diagram in Figure 4.4.

4.2.4 Reading and transmitting event data

A data acquisition run is started when the user presses the `Start` button in the Run Control UI. It causes an object of the `CommandStartRun` class to be created and `post()`-ed to the `io_service` object, in the same way as `CommandConfigure` was handled. A `BoardCommand` object with the `DAQ.BUSY.CLEAR` request, which clears the `busy` flag of the data concetrator, and allows accepting trigger pulses (see Section 3.10 for details), is created and sent to each board.

Once the commands complete successfully, the Begin-Of-Run Event (BORE) is sent to the Data Collector and the system is ready to accept data events. Asynchronous read is started on two other serial ports of each board, the DAQ port and the monitor/log console port. An instance

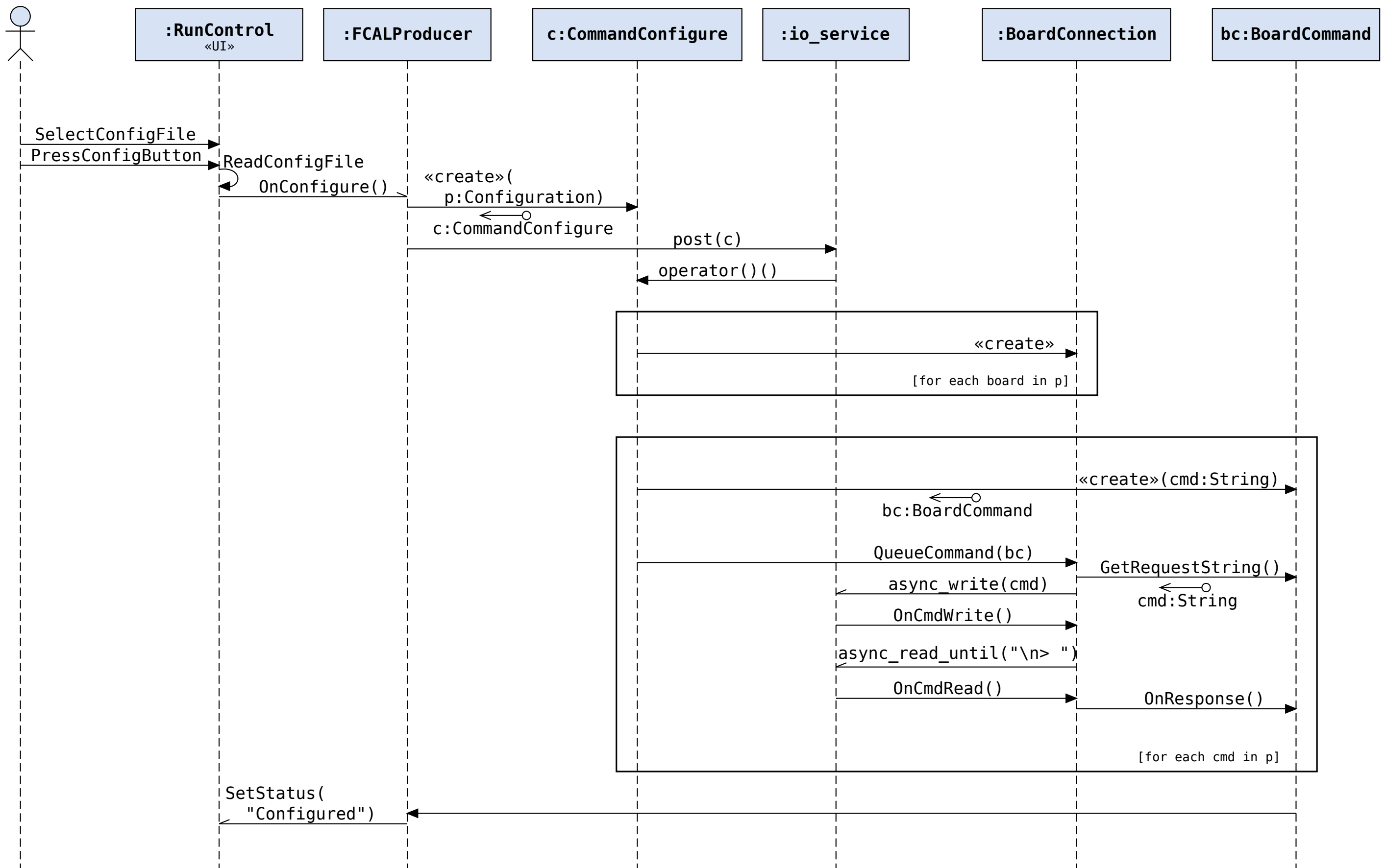


Figure 4.4: Sequence diagram of connecting to readout boards and sending configuration commands to them. To simplify the diagram, error handling is omitted.

of the `BoardEventHandler` class is created to handle complete event packets from the DAQ port and lines of text from the monitor port.

The asynchronous read handler for the DAQ port searches for Start-Of-Packet and End-Of-Packet special sequences in the received data and decodes escape sequences in order to extract complete data packets. For each packet, it then computes and checks the CRC-16 code, examines the sequence number to check if some of the previous events were lost, and, finally, calls the `OnEvent()` method of the `BoardEventHandler` instance. It takes the board identifier, the data packet and the number used to synchronize events (the TLU trigger ID) as arguments. The packets are stored in buffers (queues) until packets with the same trigger number are received from all readout boards. Then, they are moved from the buffers to a single `RawDataEvent` object, sent to the Data Collector process using the `eudaq::Producer::SendEvent()` method. The queues can accumulate a limited number of packets, and the oldest packets are removed from the queues if a complete event could not be built from them.

The asynchronous read handler for the monitor port simply reads complete lines of text and calls the `BoardEventHandler::OnMonitor()` method. Its implementation causes messages in the form of `@`*`quantity`*` = `*`value`*, i.e. periodical reports of hardware operating conditions, to be attached to the next transmitted `RawDataEvent` object as *tags*. Other messages are treated as log

messages and passed to the Log Collector process.

The data acquisition stops when the user presses the `Stop` button in the Run Control UI. The `DAQ.STOP` command is sent to all boards, and asynchronous reads on DAQ and monitor ports are cancelled. The Data Collector is sent an End-Of-Run Event (EORE), and the `Stopped` status is reported to the Run Control.

4.3 The Data Converter Plugin

Different detectors store their data in different formats, so to make the data analysis and monitoring software independent of these formats, Data Converter Plugins are used in EUDAQ to convert raw event data to a standardised representation.

One of these representations is defined by the `StandardEvent` class. Each `StandardEvent` object contains a run number, an event number, and may also contain a timestamp and tags. Data in `StandardEvent` are stored in an array of `StandardPlane` objects, where each `StandardPlane` contains charge values from one sensor. The data in one `StandardPlane` are stored as array of pixels, each pixel having two spatial coordinates, locating pixel on the sensor plane, and a time coordinate, which allows for storing multiple samples of the same pixel in consecutive *frames*.

The `FCALConverterPlugin` class derives from the `eudaq::DataConverterPlugin` abstract class and implements two functions:

`GetTriggerID()` Extracts TLU trigger ID from event data. Used to quickly synchronize events from different detectors without performing full data conversion.

`GetStandardSubEvent()` Fills the metadata in the `StandardEvent` object, and adds `StandardPlanes` to it, one plane per each connected detector layer.

A helper class (`FCALEvent`) was created to extract samples and metadata from raw events. Its public interface is simply a set of getter functions. It can be used outside the `FCALConverterPlugin`, for example when writing specialised analysis programs.

4.4 Results

Despite the relatively frequent crashes of the Run Control and Data Collector processes in longer runs, the FCAL Producer worked stably throughout the test beam measurements, and about 100 GB of data were collected in several configurations: 1 or 2 boards, with or without tungsten.

Figure 4.5 shows the histogram window of a single detector layer in the RootMonitor. Amplitude distributions for 6 selected channels are displayed in this window. Despite the high level of noise, one can notice that on most channels only pedestal distribution can be seen, whereas the two plots on the bottom left include smaller side peaks related to the signal coming from the particles being captured.

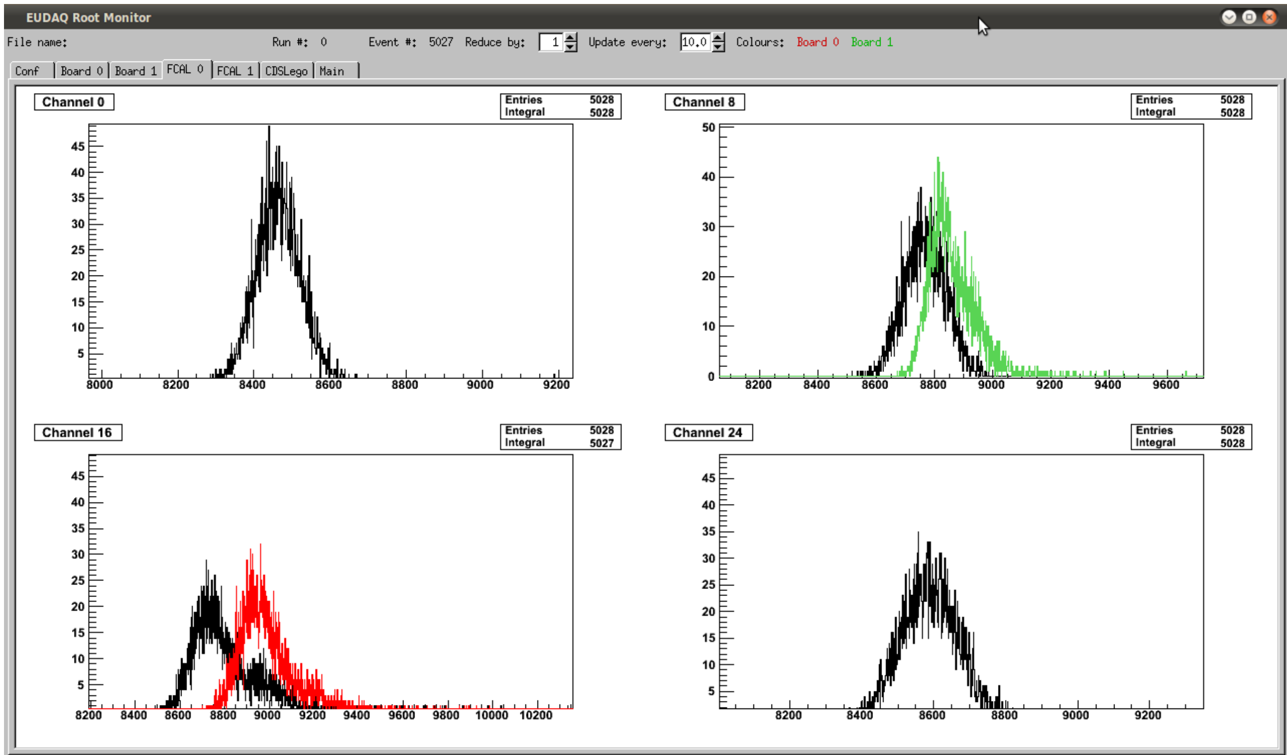


Figure 4.5: Screenshot of the RootMonitor plotting on-line distributions of amplitudes of pulses from selected channels during recording of the deposition of energy in the sensor by beam electrons.

The detailed analysis of the obtained data is beyond the scope of this thesis. However, for the sake of completeness, some preliminary results from the data collected with EUDAQ during test beam measurements in DESY will be presented. Figure 4.6 shows an example time series of a single event from one of the runs aimed at studying the electromagnetic shower development. All 32 channels are captured, and 32 samples are taken from each channel. Shaped pulses of different amplitudes can be seen on multiple channels, indicating that multiple charged particles deposited a part of their energies in the sensor pads.

Figure 4.7 shows an example amplitude distribution in about 100,000 thousand events without absorber. The green curve matches the pedestal distribution (Gaussian). The blue curve matches the distribution of energy of particles (Landau convoluted with Gaussian).

Baseline and systematic noise components were digitally subtracted in both figures. Good separation of signal from noise will allow for sufficiently precise reconstruction of time and amplitude of pulses in all events using the deconvolution procedure described in [54].

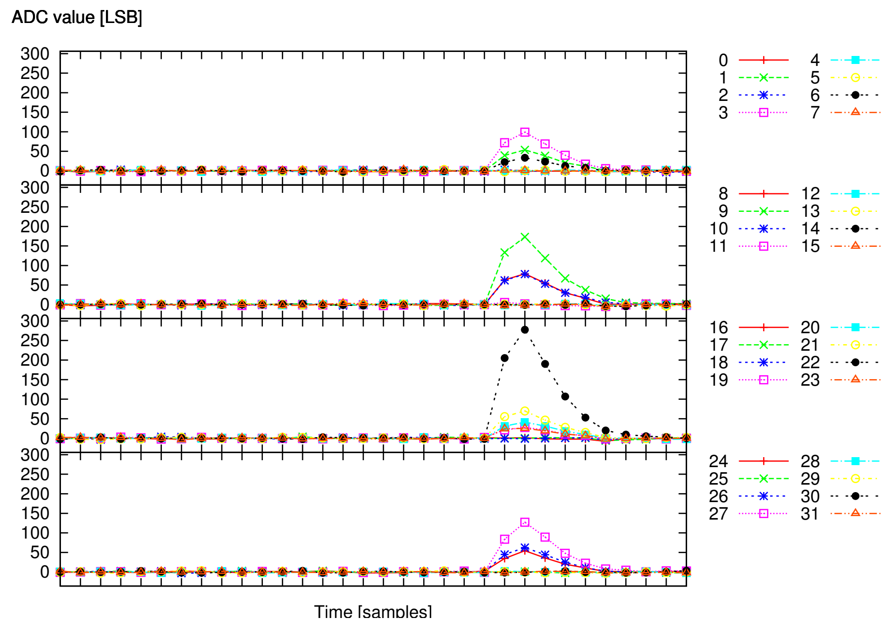


Figure 4.6: Example time series for a single event from electromagnetic shower development measurements.

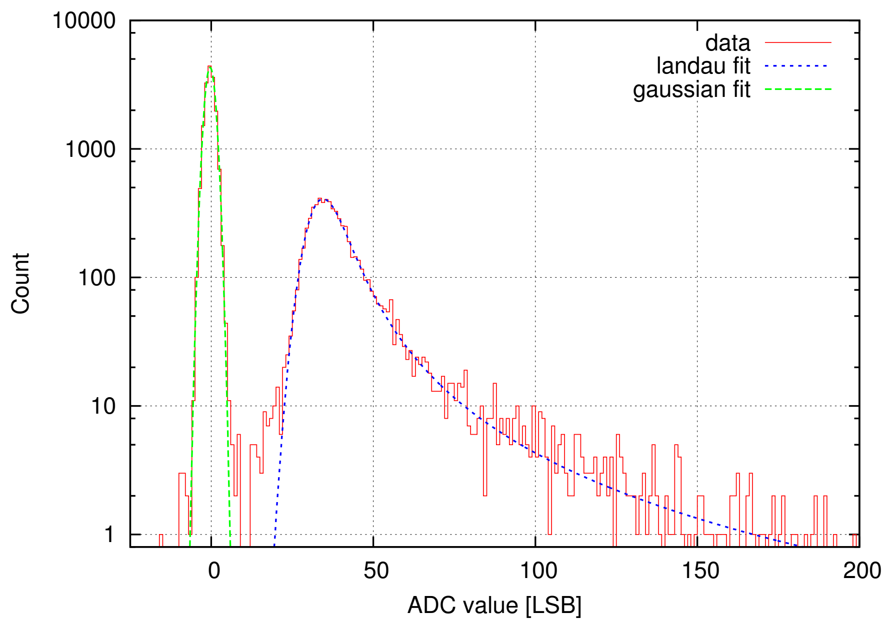


Figure 4.7: Amplitude distribution in about 100,000 events without absorber.

Chapter 5

Setting up Linux Operating System

5.1 Motivation

Linux has been successfully adopted in many embedded system applications, ranging from small consumer electronics devices, such as mobile phones, to large industrial and scientific process controllers. Main reasons for choosing Linux are that it is free and open source, configurable, and includes support for wide range of devices and protocols [55]. Also, there is a huge base of open-source and commercial software already developed for it.

Some FPGA devices available nowadays are powerful enough to install a CPU core able to run Linux System on them. Popular soft-core processors of this kind include Nios II, MicroBlaze and Xtensa architectures [56]. Some Xilinx devices, namely Virtex FXT series, even have one or two hardware PowerPC processor blocks integrated in one FPGA chip [57]. Having the CPU and programmable logic on the same chip, it is easier to decouple processing into software and hardware parts and take advantage of direct high-performance connection between them.

By combining these technologies, one can benefit from the advantages of both. Using the FPGA will allow for fast development of specialized hardware capable of processing high volumes of data, whereas writing applications for controlling the process and transferring the data for further analyses or storage can be significantly easier with Linux.

5.2 Installation process outline

The objective was to set up a Linux environment which would boot in a standalone fashion on an evaluation board with the Xilinx Virtex-5 FXT FPGA. Additional requirement was to make the installation easily modifiable in the future, by allowing adding new applications and changing configuration of existing ones.

The process of installing an embedded system based on Linux will differ among particular applications, but some generic steps and possible options typically considered at each step can be identified. They are summarized in the flowchart shown in Figure 5.1. The description in the

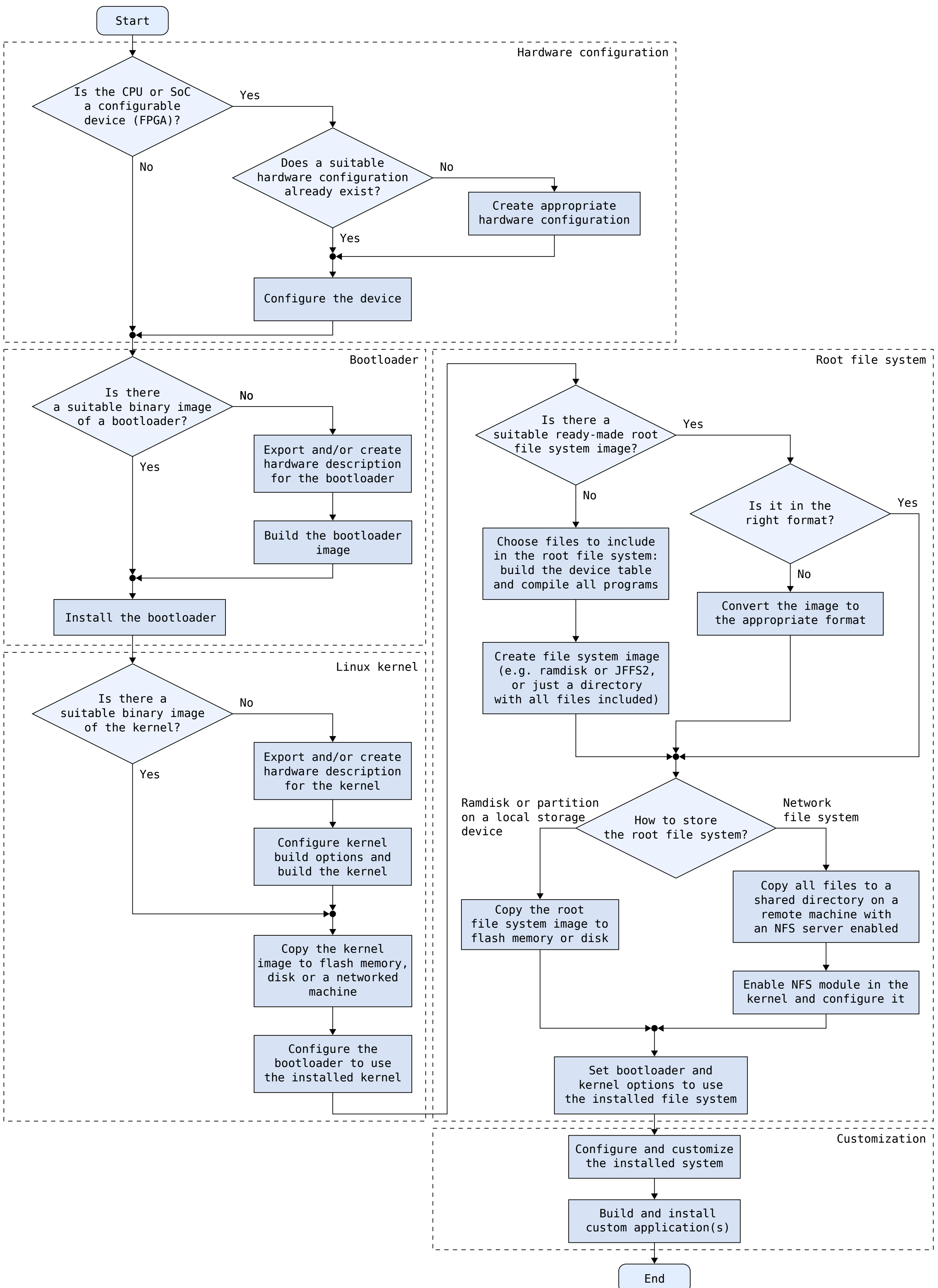


Figure 5.1: Flowchart showing the process of installation of an embedded system based on Linux.

following sections covers these steps in more detail in a case study for the Avnet Virtex-5 FXT Evaluation Kit [58]. The installation process that was carried, consisted of the following actions:

1. Obtaining information about hardware features and previous installations. Described in Section 5.3.
2. Installation of the required software on the build machine (PC)—Section 5.4.
3. Configuring the hardware (Section 5.5).
4. Building and installing the bootloader on the embedded device (Section 5.6).
5. Building and testing the kernel (Section 5.7).
6. Building the root file system image (Section 5.8).
7. Final installation of all images created in the previous steps (Sections 5.9 and 5.10).
8. Installing custom applications to implement the functionality for which the system is designed 5.11.

The description may seem too detailed, but it was intended to serve also as a reference allowing for repeating the process for the same or a different system.

5.3 Avnet Virtex-5 FXT Evaluation Kit

The board includes a Xilinx Virtex-5 XC5VFX30T-FF665 FPGA, 64 MB of DDR2 SDRAM and 16 MB of flash memory (128 blocks of 128 kB [59]), as well as two RS-232 serial ports (one of them being connected through a USB converter) and a 10/100/1000 Mbps Ethernet port [58]. Photograph of the board is shown in Figure 5.2.

Before beginning the installation of any software on the board, hardware tests described in the User Guide [58] were carried out. There are three test files available on the Avnet Design Resource Center website [60]: the "factory test design", which tests DDR2 SDRAM, flash memory, LEDs, switches, push-buttons and UART, and two other designs for testing USB-RS232 and Ethernet interfaces. The board passed all the tests successfully.

5.3.1 Configuration methods

The Avnet V5FXT board supports three methods of FPGA configuration: JTAG cable, System ACE Module, and flash memory through the BPI (Byte Peripheral Interface) [58]. The first method is especially suitable for testing new designs—the design can be programmed in very short time and it starts running immediately. However, the configuration is volatile, i.e. it is not retained after switching off the power supply. The second one allows for configuring the FPGA using bitstreams stored on CompactFlash cards, but it requires additional hardware module so it won't be described. The last method allows for loading the configuration from the BPI Flash each time the power is switched on or the "reconfigure" button is pressed, and this method was used.

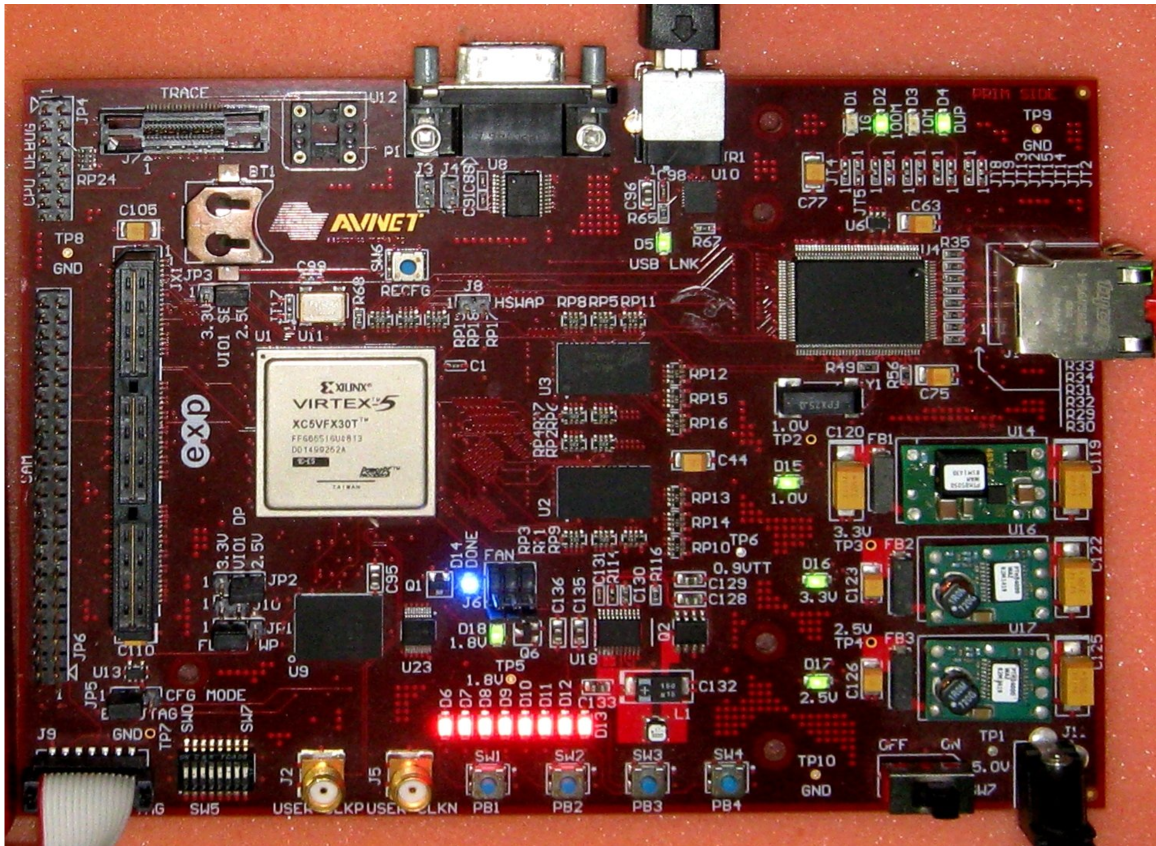


Figure 5.2: Photograph of the Avnet Virtex-5 FXT Evaluation Kit.

5.3.2 Previous Linux installations

Avnet has published a reference design for running Linux on V5FXT kit in the Design Resource Center [60]. However, there is no description provided regarding how the design was created and how the kernel and ramdisk images were built. Furthermore, this design can only be loaded using JTAG programmer and cannot be permanently installed on the board.

Similar designs have been made available on Xilinx Open Source Wiki website [61] along with step-by-step instructions for building Linux for Xilinx development boards, namely ML405, ML507 and ML510. Xilinx has also published a *how-to* document *Using U-Boot with the Xilinx ML507 Evaluation Platform* [62], describing installation in flash memory with the U-Boot bootloader, as well as other possible methods of booting, with high level of detail.

5.4 Software used in the installation

In the installation described here, several software packages were used: the tools needed to configure the FPGA and build the installed software, as well as the installed software itself.

The hardware design was created and programmed into the FPGA using the ISE Design Suite package version 12.1 from Xilinx [63]. The package includes the following programs:

ISE Project Navigator The main application managing all design files and tasks.

Xilinx Platform Studio (XPS) An application for creating embedded system configurations.

Xilinx Software Development Kit (SDK) The Eclipse IDE adapted to facilitate building software for MicroBlaze and PowerPC processors on Xilinx devices.

iMPACT A small utility for configuring FPGA and PROM devices connected with a programming cable.

The Linux kernel and the U-Boot bootloader were built using the Embedded Linux Development Kit (ELDK) 4.2 from DENX Software Engineering, which includes the GNU toolchain for PowerPC processors and the `mkimage` program for making images for the bootloader [64].

The root file system was created using Buildroot [65].

Finally, there were three source packages downloaded from Xilinx git repositories[1]: Linux, U-Boot, and the device tree generator. The first two were installed in the target system, whereas the last one is a simple script that was used to generate files for configuring both Linux and U-Boot sources for the target system.

5.5 Hardware configuration

5.5.1 Creating embedded system design

First, the FPGA must be configured to connect PowerPC CPU to peripherals and place their registers in specific areas of processor address space. Preparing such configuration is relatively easy using the Base System Builder (BSB)—a simple graphical tool (a part of XPS) for creating embedded system designs based on Xilinx FPGA devices. Board vendors may publish Xilinx Board Definition (XBD) files, containing definitions of peripherals installed on their boards, which allow the BSB to restrict the user to proper devices and correct configurations only. The XBD files used to create the design were downloaded from the Avnet website[2].

In the `New Project Wizard` of ISE Project Navigator, the FPGA device was specified using the following settings: `Family` was set to `Virtex5`, `Device` to `XC5VFX30T`, `Package` to `FF665` and `Speed grade` to `-1`.

[1]Following are the direct links to specific versions used in the installation:
U-Boot:
`http://git.xilinx.com/?p=u-boot-xlnx.git;a=commit;h=e094f2479ea339d7f48b6826f06f0be4984d9a98`,
Linux:
`http://git.xilinx.com/?p=linux-2.6-xlnx.git;a=commit;h=98ed71b684c0f06a7efcbc24ff3875e6380afae2`,
Device tree generator:
`http://git.xilinx.com/?p=device-tree.git;a=commit;h=db998c76803f5690972fdef46c40e3adf030dfb8`.
The list of repositories is available in [66]. Also, complete copy of these sources may be found on the DVD-ROM attached to this thesis, see Appendix B.

[2]The direct link to the package available at the time of this writing is `http://em.avnet.com/ctf_shared/sta/df2df2usa/avnet_edk12_2_xbd_files_21_08_2010_.zip` [67].

The next step was to use the BSB wizard in the XPS to create a configuration with a single PowerPC 440 processor and FPU enabled. Both the CPU frequency and bus frequency were set to their default value of 125 MHz. Both the instruction and data cache (32 kB each) were enabled and configured to cache only DDR2 SDRAM. Additional IP cores used in the design and their configuration are shown in Table 5.1.

Table 5.1: Peripherals (standard IP cores) implemented in the FPGA and their configuration.

Address range (hexadecimal)	IP core	Instance name	Options
00000000 - 03FFFFFF	ppc440mc_ddr2	DDR2_SDRAM_16Mx32	
81400000 - 8140FFFF	xps_gpio	Push_Buttons_3Bit	
81420000 - 8142FFFF	xps_gpio	LEDs_8Bit	
81440000 - 8144FFFF	xps_gpio	DIP_Switches_8Bit	
81480000 - 814FFFFF	xps_ll_temac	Hard_Ethernet_MAC	Use DMA, Use Interrupt
83E00000 - 83E0FFFF	xps_uart16550	RS232	Configure as UART 16550, Use Interrupt
83E20000 - 83E2FFFF	xps_uart16550	RS232_USB	Configure as UART 16550, Use Interrupt
FF000000 - FFFFFFFF	xps_mch_emc	FLASH_8Mx16	

This hardware configuration is only an example—any device (IP core) may be added, as long as the appropriate driver exists for it (either downloaded with kernel source or written by user).

One important thing here is the location of memories within the CPU address space. The processor starts execution of code at address 0xFFFFFFFC [68], and valid code must be available for it in that area during boot. It can be contained either in BRAM (Block Random Access Memory, i.e. the dedicated dual-port memory blocks in the FPGA, initial contents of which may be stored in the FPGA configuration file), or in flash memory. The second option was chosen, since the capacity of the installed BRAM is relatively small, so it might be better to save it for use by IP cores[3]. The DDR2 SDRAM was placed starting from the address 0, so that the default entry point for Linux did not have to be changed.

The BSB creates a User Constraints File (UCF), which may be used as a base when more cores are to be added later. Since in this design no other cores were used, this file was simply added to the ISE project and the programming file for the FPGA (called *bitstream*) was generated.

[3]The CPU cache is also instantiated in the BRAM.

5.5.2 Exporting Board Support Package

The next step was to generate a Board Support Package (BSP), i.e. a set of files configuring the software to run on previously created hardware design. In principle, this is done by converting text files between different formats, but eventually turned out to be rather complicated.

Following the instructions in [69], the Device Tree Generator downloaded from the git repository was copied to the new directory inside XPS project folder. Next, the hardware design was exported to SDK using the `Export Hardware Design to SDK` process in the ISE. In the SDK, the directory, to which the DTG was copied, was added to repositories using the `Xilinx Tools > Repositories` option.

A new `Xilinx Hardware Platform Specification` project was then created in an empty workspace in the SDK, using the previously exported hardware specification (which consists of a single XML file and documentation). Next, another new project, a `Xilinx Board Support Package` was created in the SDK, using the platform specification just created and `device-tree` selected as the Operating System. In the project options, `RS232` (name of the instance of serial port IP core) was set as console device.

From the set of files generated, two of them were used later. The file `xilinx.dts` contains the *device tree*—a description of all peripherals installed on the board. The kernel parses this file during boot and enables appropriate drivers to support the peripherals and configure them properly. The second file, `xparameters.h`, is a C header containing definitions of IP cores and their parameters for the bootloader.

5.6 The U-Boot bootloader

U-Boot is a universal bootloader program with a powerful console interface and ability to boot operating system from memory as well as from network. Other benefits of using the bootloader include ease of upgrading operating system or root file system images without any specialized hardware or programs (a terminal emulator with file transfer function or a TFTP server will be sufficient), as well as possibility to carry out some diagnostic tasks.

The procedure described below was based mainly on the Xilinx U-Boot how-to [62]. Other helpful resources include [70] and [55].

5.6.1 Creating board definition

Although number of supported boards in the U-Boot source tree in the git repository had already reached several hundreds [71], the Avnet Evaluation Kit was not included yet. However, several other Xilinx boards did, and one of them, the ML507, was a good starting point for creating a board definition for the Avnet board. It was so because all of the features of Avnet board are also present on ML507 (the opposite is not true, though), and the source tree contained

all the required drivers. The most significant differences between the two boards include the size of memories (64 MB of SDRAM and 16 MB of flash memory on the Avnet board, compared to 256 MB and 32 MB on ML507, respectively) and the I^{2}C EEPROM, which is absent on the Avnet board. The new board definition was created by copying definition for ML507 and making required changes to reflect these differences.

First, the existing directory `boards/xilinx/ml507` was copied to a new one, `boards/xilinx/avnet_v5fxt_evl`, and all file names were changed accordingly. The files inside the directory are:

`avnet_v5fxt_evl.c` (renamed from `ml507.c`) It had contained hardcoded size of SDRAM in functions `initdram()` and `testdram()`, which was changed to 64 MB. Also the function `checkboard()` was updated to show the right description of the board (i.e. `Board: Avnet Virtex-5 FXT Evaluation Kit`).

`Makefile` Only the include path was changed from `../ml507` to `../avnet_v5fxt_evl`. Build options are specified by the next two files, included by this `Makefile`.

`config.mk` Included by `Makefile`, contains address of the *text* (code) section for the linker. Original file for ML507 included two options, `TEXT_BASE = 0xFFFC0000` for booting from flash memory, and `TEXT_BASE = 0x02000000` for booting from SDRAM. Since the intent was to boot from flash, the first option was chosen.

`xparameters.mk` Also included by `Makefile`, defines whether additional drivers should be compiled or not. Values for `XPAR_IIC` (I^{2}C driver) and `XPAR_SYSACE` (SystemACE driver) variables were changed to “n”, i.e. the drivers were disabled.

`init.S` No changes were made.

`u-boot.lds.debug`, `u-boot-ram.lds` and `u-boot-rom.lds` Linker scripts. No changes were required.

`xparameters.h` It was substituted with the `xparameters.h` file generated in section 5.5.2.

Likewise, the file `include/configs/avnet_v5fxt_evl.h` was created by copying the file `include/configs/ml507.h`. The changes that were made included setting the address of the bootloader (`CFG_MONITOR_BASE`) to match the `TEXT_BASE` set before, changing console baud rate to 19200 baud, as well as removing references to I^{2}C EEPROM and configuring U-Boot environment to be stored in flash memory instead. The last change involved hardcoding the address and size of the environment in the header file. They were set to the flash memory base address + 0xFA0000, and 128 kB, respectively. Further explanation is given in Section 5.9.

Finally, the target for new board definition was added to the main `Makefile` by inserting lines shown in Listing 5.1. The definition was based on the similar target, `ml507_flash_config`.

5.6.2 Building and installing the U-Boot image

In order to build the U-Boot image, ELDK environment variables must be set properly, preferably using the `eldk_init` script, which is a part of the ELDK. Make is then used to configure and build

```
avnet_v5fxt_evl_flash_config:  unconfig
    @mkdir -p $(obj)include $(obj)board/xilinx/avnet_v5fxt_evl
    @cp $(obj)board/xilinx/avnet_v5fxt_evl/u-boot-rom.lds \
        $(obj)board/xilinx/avnet_v5fxt_evl/u-boot.lds
    @echo "TEXT_BASE = 0xFF400000" > $(obj)board/xilinx/avnet_v5fxt_evl/config.tmp
    @$(MKCONFIG) $(@:_flash_config=) ppc ppc4xx avnet_v5fxt_evl xilinx
```

Listing 5.1: Definition of Makefile target for building the U-Boot bootloader for the Avnet Evaluation Kit.

the image:

```
u-boot-xlnx$ make avnet_v5fxt_evl_flash_config
u-boot-xlnx$ make
```

In order to install the image in the flash memory, the bitstream generated in Section 5.5.1 had first to be programmed into the FPGA. It is required for the SDK option `Program Flash memory` to access the memory and, obviously, for the installed bootloader to run. Programming was done using the iMPACT tool.

The file `u-boot.bin` just created was programmed into the flash memory using the SDK option `Xilinx Tools > Program Flash memory`, with `Program at offset` set to 0xFC0000.

After that, pressing the CPU reset button (by default, this function is assigned to the PB1 button on the Avnet board) caused the U-Boot to load and print board configuration and command prompt on the console, as shown in Figure 5.3.

```
U-Boot 1.3.4-00327-ge094f24-dirty (May  2 2011 - 22:34:47)

CPU:   Xilinx PowerPC 440 UNKNOWN (PVR=7ff21912) at 125 MHz
       32 kB I-Cache 32 kB D-Cache
Board: Avnet Virtex-5 FXT Evaluation Kit
DRAM:  64 MB
FLASH: 16 MB
*** Warning - bad CRC, using default environment

In:    serial
Out:   serial
Err:   serial
=>
```

Figure 5.3: Console output from the U-Boot bootloader first run.

The message `*** Warning - bad CRC, using default environment` means that environment was not initialized. The environment is a set of key/value pairs, and it may be displayed using the `printenv` command of U-Boot, or saved using the `saveenv` command. Once the default environment had been saved to the flash memory, the message did not appear on the next boot.

5.7 Building and booting Linux

5.7.1 Configuring and building the kernel

The Linux source tree in git repository had already contained a configuration template for the Avnet Evaluation Kit (`arch/powerpc/configs/44x/avnet_v5fx30t_defconfig`), and it was used as a starting point to configure the kernel:

```
linux-2.6-xlnx$ make ARCH=powerpc 44x/avnet_v5fx30t_defconfig
linux-2.6-xlnx$ make ARCH=powerpc menuconfig
```

In the kernel configuration menu, the options essential for the setup with root file system in flash memory were enabled. Also the Xilinx GPIO driver was enabled and the I²C support was disabled. The following is the complete list of the options that were modified (i.e. enabled, unless otherwise noted):

- `Device Drivers`
  - `GPIO support`
    - `/sys/class/gpio/... (sysfs interface)`
    - `Xilinx GPIO support`
  - `I2C support` (disabled)
  - `Memory Technology Device (MTD) support`
    - `MTD partitioning support`
    - `Command line partition table parsing`
    - `Direct char device access to MTD devices`
    - `Caching block device access to MTD devices`
    - `RAM/ROM/Flash chip drivers`
      - `Detect flash chips by Common Flash Interface (CFI) probe`
      - `Support for Intel/Sharp flash chips`
    - `Mapping drivers for chip access`
      - `Flash device in physical memory map based on OF description`
- `File Systems`
  - `Miscellaneous filesystems`
    - `Journalling Flash File System v2 (JFFS2) support`

After saving the modified configuration, the kernel was built using Make:

```
linux-2.6-xlnx$ make ARCH=powerpc uImage
```

```
Image Name:   Linux-2.6.37+
Created:      Sun May 15 00:18:38 2011
Image Type:   PowerPC Linux Kernel Image (gzip compressed)
Data Size:    1845777 Bytes = 1802.52 kB = 1.76 MB
Load Address: 0x00000000
Entry Point:  0x00000000
```

Figure 5.4: Kernel image information displayed by `mkimage`.

The `uImage` is a `gzipped` kernel image preceded with a header containing additional information for U-Boot, such as the entry point. It is created by the `mkimage` tool, which displays the image properties as shown in Figure 5.4.

In order to boot the compiled kernel, the root file system and the device tree had yet to be prepared.

5.7.2 Root file system in ramdisk

One possible way to supply the root file system for the kernel is to use a ramdisk image, and a pre-built image was already available at the Xilinx Open Source Wiki website[4]. It was a gzip-compressed ext2 file system image, which could be turned into an image suitable for loading with U-Boot using `mkimage`:

```
u-boot-xlnx$ mkimage -A ppc -O linux -T ramdisk -C gzip -n 'Xilinx' \
> -d ramdisk.image.gz uramdisk
```

5.7.3 Compiling the device tree

The device tree for Linux must exactly match the board configuration. The tree in file `arch/powerpc/boot/dts/virtex440-avnet-v5fx30t.dts` was substituted with the new file (`xilinx.dts`) generated in Section 5.5.2. Since the root file system was first provided in a ramdisk image, the appropriate setting for the `bootargs` option had to be included in the device tree:

```
        bootargs = "console=ttyS0,19200 rootfstype=ext2 root=/dev/ram rw
mtdparts=ff000000.flash:16M(protect)ro";
```

The meaning of the above parameters is as follows:

- `console=ttyS0,19200` sets the first serial port to be used as console and configures its speed to 19200 baud;

[4]At the moment of this writing, the direct link to the image is `http://xilinx.wikidot.com/local--files/open-source-linux/ramdisk.image.gz` [72].

- `rootfstype=ext2 root=/dev/ram rw` sets the root file system type to ext2 and its location to ramdisk and tells the kernel that it is writable;
- `mtdparts=ff000000.flash:16M(protect)ro` marks the whole area of flash memory as read only in order to protect it from being erased.

Linux requires the device tree to be supplied in a special, binary form. It was compiled using the device tree compiler script available in the Linux source tree:

```
linux-2.6-xlnx$ scripts/dtc/dtc -b 0 -V 17 -R 4 -S 0x3000 -I dts -O dtb \
> -o avnet.dtb arch/powerpc/boot/dts/virtex440-avnet-v5fx30t.dts
```

5.7.4 Downloading and booting

U-Boot allows for downloading files from a TFTP server and booting from memory. The TFTP client in U-Boot requires at least three environment variables to be set: `ethaddr` (the MAC address of the board), `ipaddr` (the IP address of the board), and `serverip` (the IP address of TFTP server). The `ping` command can be used to test connectivity. For example:

```
=> setenv ipaddr 192.168.1.222
=> setenv serverip 192.168.1.55
=> setenv ethaddr 00:0a:35:c0:ff:ee
=> ping 192.168.1.55
host 192.168.1.55 is alive
```

All the three files—the Linux kernel image (`uImage`), the compiled device tree (`avnet.dtb`) and the ramdisk image (`uramdisk`)—were copied to a directory published by the TFTP server, and downloaded to the board's RAM using the following U-Boot commands:

```
=> tftp 2000000 uramdisk
=> tftp 2200000 uImage
=> tftp 2400000 avnet.dtb
```

Next, the `iminfo` and `fdt` commands in U-Boot were used to verify that the images and the device tree were downloaded correctly:

```
=> iminfo 2000000
=> iminfo 2200000
=> fdt addr 2400000
=> fdt list              (or fdt print)
```

Finally, Linux was booted using the `bootm` command:

```
=> bootm 2200000 2000000 2400000
```

This command accepts three arguments. The first argument is the address of the executable image to boot (i.e. the Linux kernel). The second one is the address of the ramdisk image, and it is optional—if no ramdisk is used, "-" should be supplied instead of the address. The last argument is the address of the compiled device tree.

When booting, U-Boot first verifies integrity of the images, then unpacks the Linux image and starts the kernel, which loads the drivers as they appear in the device tree, mounts the root file system, executes the startup script `/etc/rc.sh` and starts the `sh` shell in the console.

The above actions (i.e. downloading all images and booting the kernel) may be performed automatically by U-Boot. It is possible to execute commands stored in U-Boot environment variables. The variables were first set using the following commands:

```
=> setenv imgdl 'tftp 2000000 uramdisk; tftp 2200000 uImage; tftp 2400000 avnet.dtb'
=> setenv imgrun 'bootm 2200000 2000000 2400000'
=> setenv netboot 'run imgdl; run imgrun'
=> saveenv
```

Booting the system from a TFTP server could now be done using a single command:

```
=> run netboot
```

Renaming the latter variable, `netboot`, to `bootcmd` would cause U-Boot to download and run the whole system from the TFTP server automatically on system start or reset. This is particularly useful during development, and could be made even more flexible by providing the root file system on an NFS share.

5.8 Building the root file system

The ramdisk image could be installed in the flash memory, but it would be difficult to modify its contents in the running system, and it would occupy a significant amount of RAM, even if most of its contents were unused. These disadvantages may be overcome by installing a native flash file system, which can be easily build from scratch using Buildroot[5]. It is a set of scripts that automatically download source packages required to build a Linux-based embedded system, apply patches to them, and build various types of images that can be immediately programmed into the target device. As stated on its website, "Buildroot can generate any or all of a cross-compilation toolchain, a root file system, a kernel image and a bootloader image" [65]. Although in this installation it was used only to build the toolchain and the root file system, processes of

[5] Of course, Buildroot can also create ramdisk images.

configuring the kernel and the bootloader would not differ significantly if they were also built with Buildroot.

5.8.1 Configuration

Buildroot is configured in a similar way that the Linux kernel build is, including the possibility to use the author's preferred `menuconfig` tool. The following are the most important configuration options that were set:

- `Target Architecture`: `powerpc`,
- `Target Architecture Variant`: `440 with FPU`,
- `Toolchain`:
  - `Toolchain type`: `Buildroot toolchain`,
  - `Kernel Headers`: `Local Linux snapshot (linux-2.6.tar.bz2)` (required copying packed Linux source tree into the `dl/` directory),
  - `Binutils Version`: `binutils 2.21`,
  - `GCC compiler Version`: `gcc 4.5.x`,
  - `Build/install a shared libgcc?` (enabled),
- `System configuration`:
  - `/dev management`: `Dynamic using udev`,
  - `Port to run a getty (login prompt) on`: `ttyS0`,
  - `Baudrate to use`: `19200`,
- `Package Selection for the target`:
  - `Root FS skeleton`: `custom target skeleton`
  - `custom target skeleton path`: `fs/fcal_skeleton`, *(see below)*,
  - `Networking applications`:

    `ntp`, `ntpdate` (to set the time from the Internet),

    `openssh` (to enable secure remote access to the system over the network),
- `Filesystem images`:
  - `jffs2 root filesystem` (enabled).

5.8.2 Modifying the root file system

The root file system image generated by Buildroot can be customized in a variety of ways:

- by modifying, manually or using a script, the contents of the directory which contains all built files before it is packaged,
- by supplying a different *target skeleton*, i.e. the constant base to which programs and libraries are added when running Buildroot,

- by selecting the package `customize` in the configuration, which causes Buildroot to include additional files supplied by user in the directory `package/customize/source`,
- by extending Buildroot with new packages.

The second option was chosen to modify configuration files, so that the `eth0` interface was configured automatically, the proper time zone was set, and the time was synchronized with an NTP server once the Internet connection was available. The default skeleton, contained in the directory `fs/skeleton`, was copied to a new directory, `fs/fcal_skeleton`. The following changes were made in the new directory:

1. The following lines were added to the file `etc/network/interfaces` in order to configure the `eth0` interface automatically on each boot using DHCP:

```
auto lo eth0
iface eth0 inet dhcp
```

2. The `etc/init.d/S40network` startup script was modified by changing `/sbin/ifup -a`, which enables all network interfaces, to the following:

```
        if /sbin/ifup -a; then
            ntpdate `cat /etc/timeserver`
        fi
```

   The change would cause the system to set the current time from the network once the connection is configured. In the `etc/timeserver` file, the name of the NTP server was entered:

```
vega.cbk.poznan.pl
```

3. Default time zone was changed to the Central European Time (CET) with automatic switching to/from the Central European Summer Time (CEST) on the last Sundays of March and October, respectively. Contents of the `etc/TZ` file were changed to the following:

```
CET-1CEST-2,M3.5.0/02:00,M10.5.0/03:00
```

5.8.3 Building the image

Once the configuration and customization were done, Buildroot was run using `make`. All packages were downloaded, compiled and installed in the target directory, from which the root file system image was finally created. The image was available in file `output/images/rootfs.jffs2`.

5.9 Flash memory organization

5.9.1 Determining the bitstream location

In the BPI-Up configuration scheme adopted in the Avnet Evaluation Kit [58], the FPGA configuration bitstream is read from the flash memory starting from the address 0, in ascending order of addresses [73]. In addition, Virtex-5 FPGAs have the MultiBoot feature, which allows for storing up to four bitstreams in the flash memory. The bitstream to load is selected based on the state of the RS0 and RS1 (Revision Select) pins of the FPGA, which are in high impedance state during initial configuration, and may be driven using external pull-up and/or pull-down resistors [73]. On the Avnet board, RS0 is connected to pin A23 of the flash memory, which is the most significant addressing bit [59], and pulled up to 3.3 V [74]. RS1 is pulled down and connected to pin A24 of the memory, which is not used in the memory chip [59]. Following these conditions, the resulting start address for loading initial configuration is 0 with bit 23 set, i.e. 0x800000.

The total bitstream size for the XC5VFX30T is 13,517,056 bits [73], i.e. 1,689,632 bytes. It should then fit in 13 blocks of Flash (1664 kB or 0x1A0000 bytes). Therefore, the locations 0x800000—0x99FFFF of the flash memory were reserved for the bitstream.

5.9.2 Bootloader

As mentioned earlier in this chapter, the bootloader code (strictly speaking, the first instruction executed) must reside at the last word of the CPU address space. The U-Boot image for the flash memory is linked so that it occupies 256 kB and fits into the last two blocks of the memory. Additional block immediately preceding the bootloader was reserved for storing the U-Boot environment. This justifies the addresses used when building and installing U-Boot (Section 5.6).

5.9.3 Linux kernel, device tree and storage

The remaining part of the flash memory was allocated for the Linux kernel, the root file system and the device tree. The kernel image was 1.76 MB in size, but it is good to allocate more space to it, in case additional drivers were enabled later, hence 2 megabytes were reserved. A single flash block is enough for the device tree and so it was assigned.

Since the yet remaining area was separated into two parts by the FPGA bitstream, the smaller part was assigned to the root file system, containing system programs and settings, and the other one to the home file system, intended to hold user application(s) and data.

5.9.4 Passing the partition table to the kernel

The resulting organization of flash memory partitions is presented in Table 5.2. The location of the bootloader and the FPGA bitstream is constrained by hardware, whereas the other areas were

assigned arbitrarily.

Table 5.2: Proposed organization of flash memory partitions in the installed system.

Address	Offset	Size	Description
0xFF000000	0x000000	0x800000 (8 MB)	home file system (empty)
0xFF800000	0x800000	0x1A0000 (1664 kB)	FPGA bitstream
0xFF9A0000	0x9A0000	0x3E0000 (3968 kB)	root file system
0xFFD80000	0xD80000	0x200000 (2 MB)	Linux kernel
0xFFF80000	0xF80000	0x020000 (128 kB)	Device tree
0xFFFA0000	0xFA0000	0x020000 (128 kB)	U-Boot environment
0xFFFC0000	0xFC0000	0x040000 (256 kB)	U-Boot

The flash memory layout had to be passed to the kernel, and it was done using the `mtdparts` parameter in the `bootargs` option in the device tree. For partitions defined in this installation, it took the following form:

```
        bootargs = "console=ttyS0,19200 rootfstype=jffs2 root=/dev/mtdblock2 rw
mtdparts=ff000000.flash:8192k(home),1664k(bits)ro,3968k(rootfs),2048k(kernel)ro,
128k(devtree)ro,128k(env)ro,256k(uboot)ro";
```

Likewise, the `rootfs` was changed to point to a flash memory partition instead of the ramdisk. The device tree compilation procedure remained the same.

5.10 Programming the images

The FPGA bitstream can be programmed using the iMPACT tool and the programming cable. However, it is also possible to program it with U-Boot, and so it was done.

First, the bitstream file was converted to a raw binary file using the `promgen` command line tool (part of the ISE) with the following options:

```
$ promgen -w -p bin -c FF -o cpu.bin -u 800000 cpu.bit
```

After this conversion, the image was downloaded to the on-board SDRAM using TFTP, the appropriate flash memory area was erased, and the image was programmed:

```
=> setenv bsdl    'tftp 2000000 cpu.bin'
=> setenv bserase 'protect off ff800000 ff99ffff; erase ff800000 ff99ffff;
 protect on ff800000 ff99ffff'
=> setenv bsprog  'protect off ff800000 ff99ffff; cp.b 2000000 ff800000 ${filesize};
 protect on ff800000 ff99ffff'
=> setenv bsup    'run bsdl; run bserase; run bsprog'
=> run bsup
```

Similarly, the kernel image, the root file system image, and the device tree could be programmed with the Xilinx SDK, but U-Boot was used instead, by creating similar “scripts” in the environment variables for each image.

Booting images from the flash memory is done in a way similar to what was shown for booting from RAM in Section 5.7.4:

```
=> setenv bootcmd 'bootm ffd80000 - fff80000'
=> bootd
```

After setting the `bootcmd` variable as shown above, the installed operating system were ready to boot in a standalone fashion as it was intended.

On the first boot, post-install tasks, such as creation of the RSA and DSA key pairs for the SSH server, were executed. The network connection and time were properly configured. After a while, the login prompt appeared in the console, and the user could log in as `root` or `default`, the latter being a regular user account. First login did not require a password—the user should set the password immediately.

5.11 Porting the DAQ firmware to Linux

Initial attempts to compile and run parts of the firmware were made to confirm its portability. The firmware design allowed for easy replacing the I/O layer with what was available on the new platform, making the upper layer components working without any changes.

5.11.1 Data acquisition

The `libdaq` module of the firmware presented in Chapter 3 had already been tested before on the build machine with Linux to test its principle of operation separated from the hardware. Thanks to that, there was no need to write any new code to build it for the PowerPC Linux.

Instead of reading real data from the hardware, the packets were filled with a constant numeric pattern. The connection through the serial port used to transmit the data to PC was replaced with a TCP server. Using `netcat`, a few dozen megabytes were read from the stream produced by the server, and the time of the transmission was measured in order to evaluate the event rate. Based on the number of events contained in the received data and the time taken by the transmission, the event rate was estimated as about 1750 events per second with 32 channels and 3 samples per channel in each packet. The number is not impressive, compared to 1000 events per second with the same packet size for the 8-bit AVR running at 24 MHz, but numerous optimizations can be thought of.

First and foremost, the protocol implemented by the `BinaryEventWriter` makes no sense with the TCP connection, as the CRC becomes redundant, and simpler means of splitting the

stream into packets could be applied, such as preceding each packet with its length. On the other hand, the TCP itself may introduce a significant overhead, so replacing it with UDP or even a custom designed protocol on top of the physical layer only, could greatly improve the performance.

5.11.2 Command interface

The ported console interface also used TCP. Instead of a serial terminal emulator, `netcat` or `telnet` could be used to connect to it. Commands to list all available commands, print the command history and clear the history were interpreted and executed correctly. However, it was not possible to test the interactive features of the editor, i.e. browsing the history or using the command completion, because both `netcat` and `telnet` use a local buffer for an entire line of input.

5.12 Summary

The procedure of installing the Linux operating system described in this chapter, although closely related to one specific board, should be easy to adopt in many other systems. At every stage, possible options have been described, and the choices have been justified.

Parts of the data acquisition firmware described in Chapter 3 have been ported to the installed system and briefly tested. The tests have successfully proven the portability of the firmware components.

Summary and future work

The author's work on the development of the data acquisition system for prototype detectors has been described in this thesis. The three major goals achieved by the author are the following:

1. The firmware controlling the operation of the readout system developed for test beam measurements has been designed, implemented and tested.
2. The readout system has been integrated with an existing desktop data acquisition framework for experiments in high-energy physics (EUDAQ).
3. The complete Linux environment has been installed on the Avnet Evaluation Kit with Virtex-5 FXT FPGA, and parts of the firmware have been successfully compiled and run on that platform.

The first two goals were related directly to the development of the data acquisition system, covered by Chapters 3 and 4, and their realization included the following work done by the author:

- Understanding the principles of the system to be developed and collecting requirements.
- Proposing a general architecture of the firmware and the method of implementation.
- Implementing the build system which would allow for arranging software modules in different configurations and building these configurations for different platforms. The build system created this way helped also with implementing unit tests.
- Implementing the serial port driver and the debug console. Designing and implementing the extensible command interface, which helped the hardware engineer to test the FPGA core and other components of the readout board.
- Implementing part of the drivers for the readout board hardware components.
- Including the uC/OS-II real-time kernel in the project and implementing a simple abstraction layer to allow replacing it with a different kernel and testing software modules without the hardware.
- Extending the serial port driver to use DMA transfers and synchronization primitives of the real-time kernel to provide fast transmission in the background.

- Designing and implementing the data acquisition module and testing its performance. Observed event rate was over 10 times the required minimum, exceeding 1000 events per second for typical readout configurations allowing full reconstruction of time and amplitude of pulses on all channels.
- Designing and implementing the module for monitoring operating conditions of front-end and ADC ASICs.
- Extending the above module so that it would be capable of performing fast measurements of transient responses of the ASICs in order to evaluate their power pulsing capabilities.
- Writing the Data Producer module for the EUDAQ framework, which not only enabled a graphical user interface for the system, but also allowed for reliable synchronization between multiple detector layers (controlled by independently running copies of the firmware), as well as with the Trigger Logic Unit and the MVD Telescope.
- Writing the Data Converter Plugin for EUDAQ, which provided on-line data monitoring and conversion of the data to the representation suitable for analyses.

The author's effort and the complexity of the developed software may also be estimated by the amount of code written. The microcontroller firmware source archive, including tests, consists of about 30,000 lines of code, written mostly in C. Half of this number is the newly developed code, and the other half comes from Atmel's drivers and the uC/OS-II real-time kernel. Modules for EUDAQ comprise about 2,000 lines of new C++ code.

Installation of the Linux environment, reported in Chapter 5, required the author to create the hardware design, add support for the new board to the bootloader sources, and build the bootloader, the Linux kernel and the root file system. All the components were then installed on the board, and the command interface and data acquisition modules of the firmware were compiled and tested on the new platform. Finally, the whole process was described in detail, including explanation of all options chosen during the installation, to help the reader carry out a similar procedure.

Both the microcontroller firmware and the desktop software fulfilled all their functional and performance requirements. The quality and usefulness of the system was confirmed during the beam test in DESY, Hamburg, Germany in July 2011, where it worked continuously for several days without major problems, and collected over 100 GB of data. The obtained data are now being examined by the physicists working within the FCAL Collaboration. The current system will be used again in upcoming beam tests of LumiCal and BeamCal.

Despite the short time available for the development and limited resources of the hardware on which the firmware was run, successful efforts have been made by the author to ensure quality of the code itself. Individual parts are clearly separated from each other based on their responsibilities, and have proven to work in different configurations and on greatly dissimilar platforms. Key components have been equipped with unit tests. Code maintainability has been confirmed by author's colleagues, who have implemented their own extensions to the software.

The developed system has one minor issue, which may need some work in the future. Although the event rate is already sufficiently high in the current version of the system, it is limited by two factors. The maximum baud rate of standard USB bridge drivers is 4 Mbps, but could be theoretically increased to 12 Mbps. Optimizations in the code responsible for building event packets are also possible (part of this process can even be moved to the FPGA).

Another direction for future work is to build the next version of the system, which will be capable of reading many more channels and will reach higher event rates, using different hardware architecture. The author strongly believes that the code written so far can be successfully reused for that purpose.

The author's commitment has found additional confirmation in delivering a presentation about the system at one of the FCAL Collaboration workshops [75], and co-authoring an article covering the applicability of the system in a wider context [50]. He has also actively participated in the beam test, improving details of the software according to varying needs and having an opportunity to become its regular user.

Appendix A

Source code (selection)

A selection of source files is included in this appendix to serve as reference for implementations described in relevant chapters. Less important or large files are omitted or parts of them are removed in the following listings.

A.1 Firmware

A.1.1 Build system

Listing A.1 presents the top-level Makefile which implements all features of the build system described in Section 3.4.

```
### Top level Makefile for FCAL DAQ

comma    := ,
empty    := #

### Source and build paths.
ifneq ($(origin BUILDDIR),undefined)
srcdir   := $(realpath $(dir $(lastword $(MAKEFILE_LIST))))/
builddir := $(if $(realpath $(BUILDDIR)),$(realpath $(BUILDDIR)),$(error Invalid build directory specified: $(BUILDDIR)))
else
srcdir   := $(patsubst ./%,%,$(dir $(lastword $(MAKEFILE_LIST))))
builddir  = $(srcdir)build/$(config)
endif

### Static libraries that need to be linked using the --whole-libs option of ld
whole_libs =

### Text manipulation functions
f_chop      = $(wordlist 2,$(words $(1)),x $(1))
f_merge     = $(strip $(if $(word 2,$(2)),$(word 1,$(2))$(1)$(call f_merge,$(1),$(wordlist 2,$(words $(2)),$(2))),$(2)))

### File and directory functions
f_pwildcard = $(wildcard $(subst /./,/,$(foreach d,$(1),$(foreach subdir,. $(portdirs),$(srcdir)$(d)/$(subdir)/$(2)))))
f_portdirs  = $(strip $(if $(1),$(call f_portdirs,$(call f_chop,$(1)))) $(call f_merge,/,$(1)))
f_allsrcs   = $(call f_pwildcard,$(1),*.[cs])
f_to_objs   = $(subst /./,/,$(addprefix $(builddir)/,$(addsuffix .o,$(basename $(patsubst $(srcdir)%,%,$(1))))))
f_to_deps   = $(subst /./,/,$(addprefix $(builddir)/,$(addsuffix .dep,$(patsubst $(srcdir)%,%,$(1)))))
f_allobjs   = $(call f_to_objs,$(call f_allsrcs,$(1)))
f_incldirs  = $(patsubst %/.,%,$(wildcard $(foreach d,$(1),$(foreach subdir,. $(portdirs),$(srcdir)$(d)/$(subdir)))))
f_allscripts = $(call f_pwildcard,$(1),*.x)
f_config_mk = $(addsuffix .mk,$(join $(patsubst %,$(srcdir)config/%/,$(1)),$(1)))
f_module_mk = $(addsuffix .mk,$(join $(patsubst %,$(srcdir)%/,$(1)),$(1)))

f_info      = $(if $(TRACE),$(info $(1)))

```

```
36 ### $(eval ...) templates
37 #   Template: Include makefile once
38 define ef_once
39 $(foreach d,$(2),$(eval $(call ef_once_1,$(1),$(d))))
40 endef
41 define ef_once_1
42 ifndef makefile_$(2)_included
43 makefile_$(2)_included := T
44 $$(call f_info,Including $(2))
45 $(1) $(2)
46 endif
47 endef
48
49 #   Template: Create rules for generating and building configuration
50 #   Args:     1: Name of configuration
51 #             2: Toolchain prefix
52 #             3: Name of application module to build
53 #             4: Port specification
54 #   Example:  In config/tb2/tb2.mk:
55 #             $(eval $(call ef_configuration_template,tb2,avr-,fcal,avr xmega tb2))
56 #             In console:
57 #             $ make config-tb2
58 #             $ make tb2
59 define ef_configuration_template
60 config-$(strip $(1)):
61     @echo "Configuring for $(strip $(1))"
62     @echo "config = $(strip $(1))\nCROSS_COMPILE = $(strip $(2))\nAPP = $(strip $(3))\nPORT = $(strip $(4))\n" \
63         >genconfig.mk
64 .PHONY: config-$(strip $(1))
65
66 all_configs += $(strip $(1))
67 ifeq ($$(config),$(strip $(1)))
68 app_exe = $$($(strip $(3))_exe)
69 $$(eval $$(call ef_once,-include,$$(call f_config_mk,$(strip $(1)))))
70 $$(eval $$(call ef_once,include,$$(call f_module_mk,$(strip $(3)))))
71 $$(eval $$(call ef_once,include,$$(call f_pwildcard,common,*.mk)))
72 $(strip $(1)): $$($(strip $(3))_exe)
73 .PHONY: $(strip $(1))
74 endif
75 endef
76
77 define ef_add_module_subdir
78 $(1)_objs     += $$(call f_allobjs,$(addprefix $(1)/,$(2)))
79 $(1)_deps     += $$(call f_to_deps,$$(call f_allsrcs,$(addprefix $(1)/,$(2))))
80 $(1)_incldirs += $$(call f_incldirs,$(addprefix $(1)/,$(2)))
81 endef
82
83 #   Template: Create rules for building an application
84 define ef_app_template
85 $$(eval $$(call ef_once,include,$(call f_module_mk,$(2))))
86
87 $(1)_libs     = $(foreach lib,$(2),$$($(lib)_a) $$($(lib)_libs))
88 $(1)_objs     = $$(call f_allobjs,$(1))
89 $(1)_deps     = $$(call f_to_deps,$$(call f_allsrcs,$(1)))
90 $(1)_incldirs = $$(call f_incldirs,$(1)) $(foreach lib,$(2),$$($(lib)_incldirs))
91 $(1)_exe      = $$(builddir)/$(1)/$(1)$$(exesuf)
92
93 $$(eval $$(call ef_once,include,$$(call f_pwildcard,$(1),*.mk)))
94
95 $$($(1)_objs) $$($(1)_deps): INCLDIRS += $$($(1)_incldirs)
96 CLEAN += $$($(1)_objs) $$($(1)_deps)
97
98 $$($(1)_exe): $$($(1)_objs) $$($(1)_libs) $$(LDSCRIPTS)
99     $$(call f_do_ld,$$($(1)_objs) $$($(1)_libs) $$(LDSCRIPTS))
100 #    $$(DO_LD)
101 CLEAN += $$($(1)_exe)
102
103 $(call f_info,Including $(1)s dependencies)
104 -include $($(1)_deps)
105 endef
106
107
108 #   Template: Create rules for building a library
109 #   Args:     1: Library name
110 #             2: List of required libraries
111 #             3: List of subdirs to add
112 define ef_library_template
113 $$(eval $$(call ef_once,include,$$(call f_module_mk,$(2))))
114
115 $(1)_libs     = $(foreach lib,$(2),$$($(lib)_a) $$($(lib)_libs))
116 $(1)_objs     = $$(call f_allobjs,$(1))
117 $(1)_deps     = $$(call f_to_deps,$$(call f_allsrcs,$(1)))
```

```
118 $(1)_incldirs = $$(call f_incldirs,$(1)) $(foreach lib,$(2),$$($(lib)_incldirs))
119 $(1)_a        = $$(builddir)/$(1)/$(1).a
120 
121 # Include port-specific configuration
122 $$(eval $$(call ef_once,include,$$(call f_pwildcard,$(1),*.mk)))
123 ifneq ($(strip $(3)),)
124 $$(eval $$(call ef_add_module_subdir,$(1),$(3)))
125 endif
126 
127 $$($(1)_objs) $$($(1)_deps): INCLDIRS += $$($(1)_incldirs)
128 CLEAN += $$($(1)_objs) $$($(1)_deps)
129 
130 $$($(1)_a): $$($(1)_objs)
131     $$(DO_AR)
132 CLEAN += $$($(1)_a)
133 
134 $$(call f_info,Including $(1)s dependencies)
135 -include $$($(1)_deps)
136 endef
137 
138 ### Tools
139 
140 RM        = rm -f
141 MKDIR     = mkdir -p
142 
143 AS        = $(CROSS_COMPILE)as
144 LD        = $(CROSS_COMPILE)gcc
145 CC        = $(CROSS_COMPILE)gcc
146 CPP       = $(CROSS_COMPILE)cpp
147 AR        = $(CROSS_COMPILE)ar
148 NM        = $(CROSS_COMPILE)nm
149 OBJCOPY   = $(CROSS_COMPILE)objcopy
150 OBJDUMP   = $(CROSS_COMPILE)objdump
151 
152 CPPFLAGS  =
153 ASFLAGS   =
154 CFLAGS    = -Wall -std=gnu99
155 LDFLAGS   =
156 ARFLAGS   = cr
157 
158 LDSCRIPTS = $(call f_allscripts,common)
159 INCLDIRS  =
160 
161 LDFLAGS  += $(addprefix -Wl$(comma)--script=,$(LDSCRIPTS))
162 CPPFLAGS += $(addprefix -I,$(sort $(INCLDIRS)))
163 
164 tgt       = $(if $(VERBOSE),$@,$(patsubst $(builddir)/%,%,$@))
165 f_do      = $(if $(VERBOSE),,@echo "$(1)" && )mkdir -p $(dir $@) &&
166 DO        = $(call f_do,$(tgt))
167 
168 f_wrap_wholearchive = $(if $(filter-out %.x,$(1)),$(if $(filter $(1),\
169 $(whole_libs)),--Wl$(comma)-whole-archive $(1) --Wl$(comma)-no-whole-archive,$(1)),)
170 
171 f_do_ld   = $(call f_do,  LD   $(tgt)) $(LD) -o $@ $(LDFLAGS) \
172             $(foreach lo,$(1),$(call f_wrap_wholearchive,$(lo))) $(addprefix -l,$(LDLIBS))
173 DO_CC     = $(call f_do,  CC   $(tgt)) $(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $@
174 DO_AR     = $(call f_do,  AR   $(tgt)) $(AR) $(ARFLAGS) $@ $^
175 DO_AS     = $(call f_do,  AS   $(tgt)) $(CPP) $(CPPFLAGS) $< | $(AS) $(ASFLAGS) -o $@
176 DO_DEP    = $(call f_do,  DEP  $(tgt)) $(CPP) -MM $< $(CPPFLAGS) | sed "s,\(.*\)\.o[ :]*,$(@D)/\1.o $@ : ,g" >$@
177 
178 ### Simple build system detection.
179 
180 BUILD_OS  := $(if $(WINDIR),windows,$(shell uname -s | tr A-Z a-z))
181 BUILD_CPU := $(shell uname -m)
182 
183 exesuf    := $(if $(filter $(BUILD_OS),windows),.exe,)
184 
185 ### Configuration-related rules
186 
187 #   Try to load generated configuration...
188 -include genconfig.mk
189 ifndef config
190 all:
191     @echo "Build not configured.\nInvoke make with one of the following"
192     @echo "configurations as target or see README for details.\n"
193     @echo "$(subst $(empty) ,\n,$(addprefix config-,$(all_configs)))\n"
194     @exit 1
195 else
196 #    ...and redirect default goal to build that configuration.
197 $(call f_info,Active configuration: $(config))
198 portdirs  = $(addprefix ports/,$(call f_portdirs,$(PORT)))
199 INCLDIRS += $(addprefix config/,$(config))
```

```
all: $(config)
endif
.PHONY: all

unconfig:
	$(RM) genconfig.mk
.PHONY: unconfig

config-list:
	@echo "$(subst $(empty) ,\n,$(addprefix config-,$(all_configs)))\n"
.PHONY: config-list

### Rules for building objects and dependencies

$(builddir)/%.dep: %
	$(DO_DEP)

$(builddir)/%.o: $(srcdir)%.c
	$(DO_CC)

$(builddir)/%.o: $(srcdir)%.s
	$(DO_AS)

### Other

doc:
	doxygen
.PHONY: doc
CLEANDIR += doc/html

clean:
	$(RM) $(CLEAN)
	$(RM) -r $(CLEANDIR)
.PHONY: clean

cleanall: unconfig
	$(RM) -r $(srcdir)build
	$(RM) $(srcdir)*.gcov
.PHONY: cleanall

.DELETE_ON_ERROR:

### Define configurations
#   Each configuration includes makefiles for its own settings
#   and modules (app and libs) only when it is active.
#                                       | config name                               \
| toolchain | application       | port                      |
$(eval $(call ef_configuration_template, fcal-tb2                                   \
, avr-      , fcal              , avr xmega tb2            ))
$(eval $(call ef_configuration_template, clitest-tb2                                \
, avr-      , clitest           , avr xmega tb2            ))
$(eval $(call ef_configuration_template, clitest-$(BUILD_OS)-$(BUILD_CPU)-pty       \
,           , clitest           , $(BUILD_OS) $(BUILD_CPU)   ))
$(eval $(call ef_configuration_template, clitest-$(BUILD_OS)-$(BUILD_CPU)-tcp       \
,           , clitest           , $(BUILD_OS) $(BUILD_CPU)   ))
$(eval $(call ef_configuration_template, daqtest-tb2                                \
, avr-      , daqtest           , avr xmega tb2            ))
$(eval $(call ef_configuration_template, daqtest-$(BUILD_OS)-$(BUILD_CPU)           \
,           , daqtest           , $(BUILD_OS) $(BUILD_CPU)   ))
$(eval $(call ef_configuration_template, unittest_libbase-$(BUILD_OS)-$(BUILD_CPU) \
,           , unittest_libbase  , $(BUILD_OS) $(BUILD_CPU)   ))
$(eval $(call ef_configuration_template, unittest_libcli-$(BUILD_OS)-$(BUILD_CPU)  \
,           , unittest_libcli   , $(BUILD_OS) $(BUILD_CPU)   ))
$(eval $(call ef_configuration_template, unittest_libdaq-$(BUILD_OS)-$(BUILD_CPU)  \
,           , unittest_libdaq   , $(BUILD_OS) $(BUILD_CPU)   ))
```

Listing A.1: `Makefile`

A.1.2 Data acquisition

Implementation of the data acquisition core tasks, which are a part of the `libdaq` module, is shown in Listings A.2 and A.3. It requires two interfaces to be implemented, the `EventProducer` and the `EventWriter`. Implementation of the first one, that waits for hardware interrupt and

reads the data from the data concentrator, is shown in Listing A.4. Implementation of the `EventWriter` for the binary protocol is shown in Listing A.5. The declaration of the `Event` structure, used by all the above, is shown in Listing A.6.

```
1  /// \file daq_proc.h
2  #ifndef _DAQ_PROC_H__
3  #define _DAQ_PROC_H__
4
5  #include "types.h"
6  #include "event.h"
7  #include "event_writer.h"
8  #include "binary_event_writer.h"
9  #include "event_producer.h"
10 #include "mt.h"
11
12 /** \brief Should accommodate any supported event buffer structure.
13  *
14  *  Currently only BinaryEventWriter uses prepared buffer, and
15  *  therefore only BinaryEventBuffer appears here.
16  *
17  *  Unfortunately, this depends on the whole hierarchy of
18  *  event writers.
19  */
20 typedef union EventBuffer_union {
21     BinaryEventBuffer binary;
22 }
23 EventBuffer;
24
25 typedef struct DAQ_struct {
26     EventWriter eventWriter;
27     EventProducer eventProducer;
28
29     MT_SEM_DECLARE(txSem); ///< Output stream is ready to accept next event to send.
30     MT_SEM_DECLARE(eventSem); ///< Event is ready to be sent.
31     Event events[2];
32     EventBuffer eventBuffers[2];
33     bool stopped;
34     result_t result;
35 }
36 DAQ;
37
38 /** \brief Initalize DAQ.
39  *
40  *  \note After calling this function, initialize also event writer and event producer.
41  *
42  *  \param[in]  pDAQ    DAQ structure to initialize.
43  *  \param[in]  buffer  Buffer for at least 2 * \c CFG_EVENT_MAX_SAMPLES * \c CFG_EVENT_MAX_CHANNELS elements.
44  */
45 result_t DAQ_Init(DAQ *pDAQ, uint16_t *buffer);
46
47 void DAQ_SetSamplesPerEvent(DAQ *pDAQ, uint16_t nsamples);
48
49 void DAQ_SetChannelMask(DAQ *pDAQ, const uint32_t *channelMask);
50
51 #ifdef CFG_DAQ_PROC_NORETURN
52 #define DAQ_PROC_NORETURN __attribute__((noreturn))
53 #else
54 #define DAQ_PROC_NORETURN
55 #endif
56
57 void DAQ_ReaderProc(DAQ *pDAQ) DAQ_PROC_NORETURN;
58
59 void DAQ_WriterProc(DAQ *pDAQ) DAQ_PROC_NORETURN;
60
61 #endif // !_DAQTASKS_H__
```

Listing A.2: `libdaq/daq_proc.h`

```
1  #include "debug.h"
2  #include "daq_proc.h"
3
4  result_t DAQ_Init(DAQ *pDAQ, uint16_t *buffer) {
5      if (!MT_SEM_INIT(pDAQ->txSem, 0) || !MT_SEM_INIT(pDAQ->eventSem, 0))
```

```
6          return S("Could not init semaphores");
7      pDAQ->stopped = false;
8      pDAQ->result = RESULT_OK;
9
10     pDAQ->events[0].data = buffer;
11     pDAQ->events[1].data = buffer + CFG_EVENT_MAX_CHANNELS * CFG_EVENT_MAX_SAMPLES;
12     pDAQ->events[0].reserved = 0;
13     pDAQ->events[1].reserved = 0;
14     pDAQ->events[0].zero = 0;
15     pDAQ->events[1].zero = 0;
16
17     return RESULT_OK;
18 }
19
20 void DAQ_SetSamplesPerEvent(DAQ *pDAQ, uint16_t nsamples) {
21     pDAQ->events[0].samples = pDAQ->events[1].samples = htotb16(nsamples);
22 }
23
24 void DAQ_SetChannelMask(DAQ *pDAQ, const uint32_t *channelMask) {
25     for (size_t i = 0; i < CHANNEL_MASK_WORDS; ++i)
26         pDAQ->events[0].channelMask[i] = pDAQ->events[1].channelMask[i] = htotb32(channelMask[i]);
27 }
28
29 void DAQ_WriterProc(DAQ *pDAQ) {
30     result_t res = RESULT_OK;
31     uint8_t currentEvent = 0;
32 #ifdef CFG_DAQ_PROC_NORETURN
33     for (;;) {
34 #endif
35         for (;;) {
36             if (MT_SEM_POST(pDAQ->txSem) && MT_SEM_PEND(pDAQ->eventSem)) {
37 #ifndef CFG_DAQ_PROC_NORETURN
38                 if (pDAQ->stopped)
39                     return;
40 #endif
41                 if ((res = EventWriter_Write(&pDAQ->eventWriter, &pDAQ->events[currentEvent], &pDAQ->eventBuffers[currentEvent])) !=
42                     break;
43             } else {
44                 res = S("Semaphore problem");
45                 break;
46             }
47             currentEvent ^= 1;
48         }
49 #ifndef CFG_DAQ_PROC_NORETURN
50         pDAQ->result = res;
51         pDAQ->stopped = true;
52         (void)MT_SEM_POST(pDAQ->txSem);
53         return;
54 #else
55         if (res != RESULT_OK)
56             DPRINTF_ERROR("Error: %S\n", res);
57         MT_SLEEPMS(5000);
58         res = RESULT_OK;
59     }
60 #endif
61 }
62
63 void DAQ_ReaderProc(DAQ *pDAQ) {
64     uint8_t currentEvent = 0;
65     uint32_t seq = 0;
66     result_t res = RESULT_OK;
67 #ifdef CFG_DAQ_PROC_NORETURN
68     for (;;) {
69 #endif
70         for (;;) {
71             Event *pEvent = &pDAQ->events[currentEvent];
72             pEvent->seq = htotb32(seq);
73             if (!EventProducer_Next(&pDAQ->eventProducer, pEvent))
74                 break;
75             EventWriter_Prepare(&pDAQ->eventWriter, &pDAQ->events[currentEvent], &pDAQ->eventBuffers[currentEvent]);
76             if (MT_SEM_PEND(pDAQ->txSem) && MT_SEM_POST(pDAQ->eventSem)) {
77 #ifndef CFG_DAQ_PROC_NORETURN
78                 if (pDAQ->stopped)
79                     return;
80 #endif
81             } else {
82                 res = S("Semaphore problem");
83                 break;
84             }
85             seq++;
86             currentEvent ^= 1;
87         }
```

```
#ifndef CFG_DAQ_PROC_NORETURN
        pDAQ->result = res;
        pDAQ->stopped = true;
        (void)MT_SEM_POST(pDAQ->eventSem);
        return;
#else
        if (res != RESULT_OK)
            DPRINTF_ERROR("Error: %S\n", res);
        MT_SLEEPMS(5000);
        res = RESULT_OK;
    }
#endif
}
```

Listing A.3: `libdaq/daq_proc.c`

```
/// \file daq_producer.c
#include "daq_producer.h"
#include "event.h"
#include "event_producer.h"
#include "daq_proc.h"
#include "debug.h"
#include "fcaldc.h"
#include "mt.h"
#include "port_driver.h"
#include "pmic_driver.h"
#include "ucos_ii.h"
#include <avr/interrupt.h>

#include "static_assert.h"
STATIC_ASSERT(CFG_EVENT_MAX_CHANNELS == 32, current_hw_supports_only_exactly_32_channels);
STATIC_ASSERT(FCALDC_DAQ_MAX_CHANNELS == 32, fcaldc_and_event_must_be_configured_to_the_same_num_of_channels);

#define DETECTED_PORT     PORTQ
#define DETECTED_PIN      0
#define DETECTED_PORT_INT PORTQ_INT0_vect

OS_STK DaqTask_stack[DAQ_TASK_STACK_SIZE];
OS_STK TxTask_stack[TX_TASK_STACK_SIZE];

static MT_SEM_DECLARE(pTriggerSem);
static MT_SEM_DECLARE(pTestEventSem);
static DAQTASK_MODE_t DaqTask_mode = DAQTASK_MODE_NORMAL;

MT_ISR(DETECTED_PORT_INT) {
    PORT_ConfigureInterrupt0(&DETECTED_PORT, PORT_INT0LVL_OFF_gc, (1 << DETECTED_PIN));
    MT_SEM_POST(pTriggerSem);
}

static bool FCALEventProducer_Next(void *pObj, Event *pEvent) {
    for (;;) {
        if (!FCALDC.DAQ.STOPPING)
            FCALDC.DAQ.BUSYCLR = 0xFF;
        PORT_ConfigureInterrupt0(&DETECTED_PORT, PORT_INT0LVL_HI_gc, (1 << DETECTED_PIN));
        MT_SEM_PEND(pTriggerSem);
        if (DaqTask_mode == DAQTASK_MODE_NORMAL)
            break;
        else
            MT_SEM_POST(pTestEventSem);
    }

    uint16_t *buf = pEvent->data;
    uint8_t smps = FCALDC.DAQ.SAMPLES;

    pEvent->timestamp = htotb64(FCALDC.DAQ.TIME);
    pEvent->samples = htotb16((uint16_t)smps);
    pEvent->triggerType = htotb16(FCALDC.DAQ.TRIGSTAT);
    pEvent->triggerChns[0] = htotb32(FCALDC.DAQ.TRIGCHNS);

    uint8_t first = (FCALDC_DAQ_MAX_SAMPLES + FCALDC.DAQ.CNT + FCALDC.DAQ.POST - smps) % FCALDC_DAQ_MAX_SAMPLES;
    uint8_t last  = first + smps;

    // event data wrapped over buffer boundary
    if (last >= FCALDC_DAQ_MAX_SAMPLES) {
        last %= FCALDC_DAQ_MAX_SAMPLES;
        for (uint8_t s = first; s < FCALDC_DAQ_MAX_SAMPLES; s++) {
            volatile uint16_t *q = &FCALDC.DAQ.ADCVAL[0][s];
```

```
62             for (uint8_t chn = 0; chn < FCALDC_DAQ_MAX_CHANNELS; chn++) {
63                 *buf++ = htotb16(*q);
64                 q += FCALDC_DAQ_MAX_SAMPLES;
65             }
66         }
67         for (uint8_t s = 0; s < last; s++) {
68             volatile uint16_t *q = &FCALDC.DAQ.ADCVAL[0][s];
69             for (uint8_t chn = 0; chn < FCALDC_DAQ_MAX_CHANNELS; chn++) {
70                 *buf++ = htotb16(*q);
71                 q += FCALDC_DAQ_MAX_SAMPLES;
72             }
73         }
74     }
75     // event data continuous
76     else {
77         for (uint8_t s = first; s < last; s++) {
78             volatile uint16_t *q = &FCALDC.DAQ.ADCVAL[0][s];
79             for (uint8_t chn = 0; chn < FCALDC_DAQ_MAX_CHANNELS; chn++) {
80                 *buf++ = htotb16(*q);
81                 q += FCALDC_DAQ_MAX_SAMPLES;
82             }
83         }
84     }
85 
86     return true;
87 }
88 
89 /* we won't cancel daq
90 static void FCALEventProducer_Cancel(void *pObj)
91 {
92 }
93 */
94 
95 void FCALEventProducer_Init(EventProducer *pProducer) {
96     pProducer->next = &FCALEventProducer_Next;
97     pProducer->cancel = NULL;
98     pProducer->pObj = NULL;
99 
100     MT_SEM_INIT(pTriggerSem, 0);
101     MT_SEM_INIT(pTestEventSem, 0);
102 
103     // Configure PC0 as input, triggered on rising edge.
104     PORT_ConfigurePins(
105         &DETECTED_PORT,
106         (1 << DETECTED_PIN),
107         false,
108         false,
109         PORT_OPC_TOTEM_gc,
110         PORT_ISC_RISING_gc);
111 
112     PORT_SetPinsAsInput(&DETECTED_PORT, (1 << DETECTED_PIN));
113 
114     // Configure Interrupt0 to have medium interrupt level, triggered by pin 0.
115     PORT_ConfigureInterrupt0(&DETECTED_PORT, PORT_INT0LVL_OFF_gc, (1 << DETECTED_PIN));
116 
117     // Enable medium level interrupts in the PMIC.
118     PMIC_EnableHighLevel();
119 }
120 
121 static void TxTask(void *pArg) __attribute__((noreturn));
122 static void TxTask(void *pArg) {
123     DAQ_WriterProc((DAQ *)pArg);
124 }
125 
126 void DaqTask(void *pArg) {
127     static uint16_t buffers[2 * CFG_EVENT_MAX_CHANNELS * CFG_EVENT_MAX_SAMPLES] __attribute__((section(".extbss")));
128     static const uint32_t defaultMask = 0xFFFFFFFF;
129     DAQ daq;
130     DAQ_Init(&daq, buffers);
131     DAQ_SetChannelMask(&daq, &defaultMask);
132     BinaryEventWriter_Init(&daq.eventWriter, (Stream *)pArg);
133     FCALEventProducer_Init(&daq.eventProducer);
134     OSTaskCreate(&TxTask, (void *)&daq, &TxTask_stack[TX_TASK_STACK_SIZE - 1], TX_TASK_PRIO);
135     DAQ_ReaderProc(&daq);
136     if (daq.result != RESULT_OK)
137         DPRINTF_ERROR("%S\n", daq.result);
138 }
139 
140 void DaqTask_PendEvent(void) {
141     if (!MT_SEM_PEND(pTestEventSem)) {
142         DPRINTF_ERROR("Error in DaqTask_PendEvent\n");
143         MT_SLEEPMS(1000);
```

```
144     }
145 }
146
147 void DaqTask_SetMode(DAQTASK_MODE_t m) {
148     DaqTask_mode = m;
149 }
```

Listing A.4: `fcal/daq_producer.c`

```
1   #include <limits.h>
2   #include "binary_event_writer.h"
3   #include "crc16.h"
4
5   /// \brief Returns the number of 1-bits in x.
6   /// \note This is a gcc built-in function.
7   int __builtin_popcount(unsigned int x);
8
9   /// \brief Similar to __builtin_popcount, except the argument type is unsigned long.
10  /// \note This is a gcc built-in function.
11  int __builtin_popcountl(unsigned long x);
12
13  /// \brief Returns the number of 1-bits in x.
14  static inline int popcount32(uint32_t x) {
15  #if UINT_MAX == UINT32_MAX
16      return __builtin_popcount(x);
17  #elif ULONG_MAX == UINT32_MAX
18      return __builtin_popcountl(x);
19  #else
20  #error "No match for UINT32_MAX"
21  #endif
22  }
23
24  result_t BinaryEventWriter_Prepare(void *pObj, Event *pEvent, void *pBuffer);
25  result_t BinaryEventWriter_Write(void *pObj, Event *pEvent, void *pBuffer);
26
27  void BinaryEventWriter_Init(EventWriter *pWriter, Stream *pStream) {
28      pWriter->pObj = (void *)pStream;
29      pWriter->write = &BinaryEventWriter_Write;
30      pWriter->prepare = &BinaryEventWriter_Prepare;
31  }
32
33  #define PEB     ((BinaryEventBuffer *)pBuffer)
34  #define PSTREAM ((Stream *)pObj)
35  result_t BinaryEventWriter_Prepare(void *pObj, Event *pEvent, void *pBuffer) {
36      bool gotFF = false;
37      uint8_t *p;
38
39      inline void catByte(uint8_t b) {
40          if (gotFF) {
41              if ((b & 0xF8) == 0xF8) {
42                  *p++ = 0xFF;
43                  *p++ = 0xFE;
44                  *p++ = (uint8_t)~b;
45              } else {
46                  *p++ = 0xFF;
47                  *p++ = b;
48              }
49              gotFF = false;
50          } else {
51              if (b == 0xFF)
52                  gotFF = true;
53              else
54                  *p++ = b;
55          }
56      }
57
58      p = &PEB->headerBuf[0];
59  #if CFG_EVENT_MAX_CHANNELS <= 32
60      uint8_t channels = popcount32(tbtoh32(pEvent->channelMask[0]));
61  #else
62      uint16_t channels = 0;
63      for (size_t i = 0; i < CHANNEL_MASK_WORDS; ++i)
64          channels += popcount32(tbtoh32(pEvent->channelMask[i]));
65  #endif
66      uint16_t eventSamples = tbtoh16(pEvent->samples);
67  #ifdef BINARY_EVENT_WRITER_ENABLE_CRC
68      uint16_t crc = CRC16_Init();
```

```c
#endif
    pEvent->zero = 0;
    // write Start of Frame
    *p++ = 0xFF; *p++ = 0xFA;
    // write escaped event header in big endian
    for (uint8_t i = 0; i < EVENT_HEADER_SIZE; ++i) {
        uint8_t b = ((uint8_t *)pEvent)[i];
        catByte(b);
#ifdef BINARY_EVENT_WRITER_ENABLE_CRC
        crc = CRC16_Update(crc, b);
#endif
    }
    PEB->headerLen = p - &PEB->headerBuf[0];
    // add event data to crc
    uint16_t totalSamples = channels * eventSamples;
    PEB->totalSamples = totalSamples;
#ifdef BINARY_EVENT_WRITER_ENABLE_CRC
    for (uint16_t i = 0; i < totalSamples; ++i) {
        crc = CRC16_Update(crc, tbtoh16(pEvent->data[i]) & 0xFF);
        crc = CRC16_Update(crc, tbtoh16(pEvent->data[i]) >> 8);
    }
#endif
    // Add tail (crc and EoF).
    // Last byte of data is never 0xFF, so it's not possible
    // to detect false SoF when first byte of CRC is 0xFA.
    p = &PEB->tailBuf[0];
#ifdef BINARY_EVENT_WRITER_ENABLE_CRC
    catByte(crc & 0xFF);
    catByte(crc >> 8);
#else
    catByte(0);
    catByte(0);
#endif
    *p++ = 0xFF;
    *p++ = gotFF ? 0xFD : 0xFC;
    PEB->tailLen = p - &PEB->tailBuf[0];
    return RESULT_OK;
}

result_t BinaryEventWriter_Write(void *pObj, Event *pEvent, void *pBuffer) {
    result_t res;
    res = Stream_Write(PSTREAM, PEB->headerBuf, PEB->headerLen, NULL);
    if (res != RESULT_OK)
        return res;
    res = Stream_Write(PSTREAM, pEvent->data, sizeof(uint16_t) * PEB->totalSamples, NULL);
    if (res != RESULT_OK)
        return res;
    return Stream_Write(PSTREAM, PEB->tailBuf, PEB->tailLen, NULL);
}
#undef PEB
#undef PSTREAM
```

Listing A.5: `libdaq/binary_event_writer.c`

```c
/** \file event.h
 *  \par Options:
 *  - <tt>CFG_EVENT_MAX_CHANNELS</tt> Maximum number of channels in event.
 *    Determines length of \c channelMask.
 *  - <tt>CFG_EVENT_MAX_SAMPLES</tt> Maximum number of samples per channels in event.
 */
#ifndef _EVENT_H__
#define _EVENT_H__

#define CHANNEL_MASK_WORDS ((CFG_EVENT_MAX_CHANNELS + 31) / 32)

#include "types.h"
#include "endian.h"

/** \brief Returns x with the order of the bytes reversed;
 *  for example, 0xaabbccdd becomes 0xddccbbaa. Byte here always means exactly 8 bits.
 *
 *  \note This is a gcc built-in function.
 */
int32_t __builtin_bswap32(int32_t x);

/** \brief Similar to __builtin_bswap32, except the argument and return types are 64-bit.
 *
```

```
24   *  \note This is a gcc built-in function.
25   */
26  int64_t __builtin_bswap64(int64_t x);
27
28  static inline uint16_t htotb16(uint16_t hostuint16) {
29  #if __BYTE_ORDER == __LITTLE_ENDIAN
30      return hostuint16;
31  #else
32      return (hostuint16 >> 8) | (hostuint16 << 8);
33  #endif
34  }
35
36  static inline uint32_t htotb32(uint32_t hostuint32) {
37  #if __BYTE_ORDER == __LITTLE_ENDIAN
38      return hostuint32;
39  #else
40      return __builtin_bswap32(hostuint32);
41  #endif
42  }
43
44  static inline uint64_t htotb64(uint64_t hostuint64) {
45  #if __BYTE_ORDER == __LITTLE_ENDIAN
46      return hostuint64;
47  #else
48      return __builtin_bswap64(hostuint64);
49  #endif
50  }
51
52  static inline uint16_t tbtoh16(uint16_t tbuint16) {
53      return htotb16(tbuint16);
54  }
55
56  static inline uint32_t tbtoh32(uint32_t tbuint32) {
57      return htotb32(tbuint32);
58  }
59
60  static inline uint64_t tbtoh64(uint64_t tbuint64) {
61      return htotb64(tbuint64);
62  }
63
64  /** \brief Structure holding event header and pointer to
65   *         event data.
66   *
67   *  \note Order and sizes of fields of this structure
68   *        map directly to binary protocol.
69   *
70   *  \note All fields should be set in "tb" byte order.
71   *        Use htotb*() macros to set field values and
72   *        tbtoh*() macros to get field values.
73   *
74   *  \sa   htotb16() htotb32() htotb64()
75   *  \sa   tbtoh16() tbtoh32() tbtoh64()
76   */
77  typedef struct __attribute__((packed)) Event_struct {
78      /// Readout board sequence number.
79      uint32_t seq;
80      /// Data concentrator timestamp in 5ns intervals.
81      uint64_t timestamp;
82      uint16_t reserved;
83      /// Number of samples per channel.
84      uint16_t samples;
85      /// Only channels with corresponding bit set
86      /// are included in the data.
87      uint32_t channelMask[CHANNEL_MASK_WORDS];
88      /// Channels that triggered the event.
89      uint32_t triggerChns[CHANNEL_MASK_WORDS];
90      /// Trigger type.
91      uint16_t triggerType;
92      /// TLU trigger ID.
93      uint16_t tluNumber;
94      /// Board ID hash.
95      uint8_t boardId;
96      /// Reserved, must be 0.
97      uint8_t zero;
98      /// Array of samples*channels objects of type uint16_t, ordered
99      /// by samples first, then by channels, in ascending order.
100     /// channels is the number of bits set in channelMask.
101     /// Maximum number of elements is EVENT_MAX_SAMPLES*EVENT_MAX_CHANNELS.
102     uint16_t *data;
103 }
104 Event;
105
```

```
#define EVENT_HEADER_SIZE (sizeof(Event) - sizeof(uint16_t *))

#endif // !_EVENT_H__
```

Listing A.6: `libdaq/event.h`

A.2 FCAL Producer for EUDAQ

The following listings show the implementation of the classes described in Chapter 4. Source code of the `FCALProducer` class is presented in Listings A.7 and A.8. In Listings A.11 and A.12, the source code of the `CommandConfigure` class is shown. In Listings A.9 and A.10, the source code of the `BoardEventHandler` class is presented.

```
/// \file FCALProducer.hh

#ifndef _FCAL_FCALPRODUCER_H__
#define _FCAL_FCALPRODUCER_H__

#include "BoardConnection.hh"
#include "eudaq/Producer.hh"
#include "eudaq/Mutex.hh"
#include <boost/smart_ptr.hpp>
#include <boost/asio.hpp>

class FCALProducer : public eudaq::Producer {
public:
  FCALProducer(const std::string & runcontrol);

  ~FCALProducer();

  /// Reads data from hardware and forms events
  void ReadoutLoop();

  /// This gets called whenever the DAQ is configured
  virtual void OnConfigure(const eudaq::Configuration & param);

  /// This gets called whenever a new run is started
  /// It receives the new run number as a parameter
  virtual void OnStartRun(unsigned param);

  /// This gets called whenever a run is stopped
  virtual void OnStopRun();

  /// This gets called when the Run Control is terminating, we should also exit.
  virtual void OnTerminate();

  std::vector<boost::shared_ptr<fcal::BoardConnection> >& GetBoards() {
    return m_boards;
  }

  const std::vector<boost::shared_ptr<fcal::BoardConnection> >& GetBoards() const {
    return m_boards;
  }

  boost::asio::io_service& GetIoService() {
    return m_ioservice;
  }

  static const std::string EVENT_TYPE;
  static const std::string PRODUCER_NAME;

private:
  bool m_terminate;
  boost::asio::io_service m_ioservice;
  std::vector<boost::shared_ptr<fcal::BoardConnection> > m_boards;

#ifdef BOOST_ASIO_HAS_POSIX_STREAM_DESCRIPTOR
  void OnStdinRead(const boost::system::error_code& error, std::size_t length);
  void ReadUserInput();
```

```
57    boost::asio::posix::stream_descriptor m_stdin;
58    boost::asio::streambuf m_inputBuffer;
59  #endif // BOOST_ASIO_HAS_POSIX_STREAM_DESCRIPTOR
60  };
61  #endif // !_FCAL_FCALPRODUCER_H__
```

Listing A.7: `fcal/include/FCALProducer.hh`

```
1   /// \file FCALProducer.cxx
2
3   #include "FCALProducer.hh"
4
5   #include "BoardConnection.hh"
6   #include "CommandConfigure.hh"
7   #include "CommandStartRun.hh"
8   #include "CommandStopRun.hh"
9
10  #include "eudaq/RawDataEvent.hh"
11  #include "eudaq/Utils.hh"
12  #include "eudaq/Logger.hh"
13  #include "eudaq/OptionParser.hh"
14  #include "eudaq/counted_ptr.hh"
15  #include <iostream>
16  #include <ostream>
17  #include <cstdlib>
18  #include <map>
19  #include <sys/time.h>
20  #include <boost/asio.hpp>
21  #include <boost/lexical_cast.hpp>
22  #include <boost/bind.hpp>
23
24  using namespace boost::asio;
25  using boost::lexical_cast;
26  using eudaq::Mutex;
27  using eudaq::MutexLock;
28
29  using namespace fcal;
30
31  /// Name to identify the raw data format of the events generated
32  const std::string FCALProducer::EVENT_TYPE = "FCAL";
33
34  /// Name to indentify the producer
35  const std::string FCALProducer::PRODUCER_NAME = "FCAL";
36
37  FCALProducer::FCALProducer(const std::string & runcontrol)
38    : eudaq::Producer(PRODUCER_NAME, runcontrol), m_terminate(false)
39  #ifdef BOOST_ASIO_HAS_POSIX_STREAM_DESCRIPTOR
40    , m_stdin(m_ioservice, ::dup(STDIN_FILENO)), m_inputBuffer(1024)
41  #endif { // BOOST_ASIO_HAS_POSIX_STREAM_DESCRIPTOR
42    std::cout << "FCALProducer created.\n";
43  }
44
45  FCALProducer::~FCALProducer() {
46  #ifdef BOOST_ASIO_HAS_POSIX_STREAM_DESCRIPTOR
47    m_stdin.close();
48  #endif // BOOST_ASIO_HAS_POSIX_STREAM_DESCRIPTOR
49  }
50
51  #ifdef BOOST_ASIO_HAS_POSIX_STREAM_DESCRIPTOR
52  void FCALProducer::ReadUserInput() {
53    boost::asio::async_read_until(m_stdin, m_inputBuffer, '\n',
54                                  boost::bind(&FCALProducer::OnStdinRead, this,
55                                      boost::asio::placeholders::error,
56                                      boost::asio::placeholders::bytes_transferred));
57  }
58
59  void FCALProducer::OnStdinRead(
60    const boost::system::error_code& error, std::size_t bytes_transferred) {
61    (void)bytes_transferred;
62    if (!error) {
63      struct UserCommand : public BoardCommand {
64        UserCommand(FCALProducer& prod, const std::string& request)
65          : BoardCommand(request), producer(prod) {}
66
67        virtual void OnResponse() {
68          std::size_t nlines = m_response.size();
69          if (nlines == 0 || m_response[0] != m_request)
```

```
70            std::cout << "Warning, command not echoed." << std::endl;
71          for (std::size_t i = 1; i < nlines; ++i)
72            std::cout << m_response[i] << std::endl;
73          producer.ReadUserInput();
74        }
75
76        FCALProducer& producer;
77      };
78
79      std::istream is(&m_inputBuffer);
80      std::string line;
81      std::getline(is, line);
82      if (line.length() > 2 && line[0] == 'b' && line[2] == '.' && line[1] >= '0'
83        && line[1] <= '9') {
84        int addr = line[1] - '0';
85        if (m_boards[addr]) {
86          BoardCommand *pCmd = new UserCommand(*this, line.substr(3, std::string::npos));
87          m_boards[addr]->QueueCommand(boost::shared_ptr<BoardCommand>(pCmd));
88        } else
89          std::cout << "Board not connected" << std::endl;
90      } else {
91        std::cout << "Syntax error" << std::endl;
92        ReadUserInput();
93      }
94    } else {
95      OnTerminate();
96    }
97  }
98  #endif // BOOST_ASIO_HAS_POSIX_STREAM_DESCRIPTOR
99
100 void FCALProducer::ReadoutLoop() {
101 #ifdef BOOST_ASIO_HAS_POSIX_STREAM_DESCRIPTOR
102   ReadUserInput();
103 #endif // BOOST_ASIO_HAS_POSIX_STREAM_DESCRIPTOR
104   for (;;) {
105     m_ioservice.run();
106     if (m_terminate)
107       break;
108     m_ioservice.reset();
109     eudaq::mSleep(20);
110   }
111   return;
112 }
113
114 void FCALProducer::OnConfigure(const eudaq::Configuration & param) {
115   std::cout << "Configuring..." << std::endl;
116   m_ioservice.post(CommandConfigure(*this, param));
117 }
118
119 void FCALProducer::OnStartRun(unsigned run) {
120   std::cout << "Starting run #" << run << "..." << std::endl;
121   m_ioservice.post(CommandStartRun(*this, run));
122 }
123
124 void FCALProducer::OnStopRun() {
125   std::cout << "Stopping run..." << std::endl;
126   m_ioservice.post(CommandStopRun(*this));
127 }
128
129 void FCALProducer::OnTerminate() {
130   std::cout << "Quitting... ";
131   m_terminate = true;
132   m_ioservice.stop();
133 }
134
135 int main(int argc, const char** argv) {
136   (void)argc;
137   eudaq::OptionParser op("EUDAQ FCAL Producer", "0.1",
138                          "The Producer task for the FCAL readout board (v2.1)");
139   eudaq::Option<std::string> rctrl(op, "r", "runcontrol",
140                                    "tcp://localhost:44000", "address",
141                                    "The address of the RunControl application");
142   eudaq::Option<std::string> level(op, "l", "log-level", "NONE", "level",
143                                    "The minimum level for displaying log messages locally");
144   try {
145     op.Parse(argv);
146     EUDAQ_LOG_LEVEL(level.Value());
147     FCALProducer producer(rctrl.Value());
148     producer.ReadoutLoop();
149     std::cout << "Ok" << std::endl;
150     eudaq::mSleep(3000);
151   } catch (...) {
```

```
152     return op.HandleMainException();
153   }
154   return 0;
155 }
```

Listing A.8: `fcal/include/FCALProducer.cxx`

```
1  /// \file BoardEventHandler.hh
2  #ifndef _FCAL_BOARDEVENTHANDLER_H__
3  #define _FCAL_BOARDEVENTHANDLER_H__
4
5  #include "BoardConnection.hh"
6  #include "FCALProducer.hh"
7
8  namespace fcal {
9  class BoardEventHandler : public BoardEventListener {
10 public:
11   BoardEventHandler(FCALProducer& producer, unsigned run);
12   ~BoardEventHandler();
13
14   /// Called when a complete DAQ event frame arrives.
15   virtual void OnEvent(int addr, boost::uint16_t trn, const std::vector<boost::uint8_t>& rawData);
16
17   /// Called when a monitor line arrives.
18   virtual void OnMonitor(int addr, const std::string& str);
19
20   /// Called when an error occurs.
21   virtual void OnError(int addr, const std::string& str);
22
23 private:
24   FCALProducer& m_producer;
25   unsigned m_run;
26   unsigned m_ev;
27   std::vector<boost::shared_ptr<
28   std::map<boost::uint32_t, std::vector<boost::uint8_t> > > > m_buffers;
29   std::vector<boost::shared_ptr<boost::uint32_t> > m_trns;
30   boost::uint32_t m_newest;
31   std::multimap<std::string, std::string> m_tags;
32   std::size_t m_connectedBoards;
33 };
34 } // namespace fcal
35
36 #endif // !_FCAL_BOARDEVENTHANDLER_H__
```

Listing A.9: `fcal/include/BoardEventHandler.hh`

```
1  /// \file BoardEventHandler.cc
2  #include "BoardEventHandler.hh"
3
4  #include "eudaq/RawDataEvent.hh"
5  #include "eudaq/Logger.hh"
6
7  #include <boost/foreach.hpp>
8  #include <boost/lexical_cast.hpp>
9
10 using namespace fcal;
11 using boost::lexical_cast;
12
13 BoardEventHandler::BoardEventHandler(FCALProducer& producer, unsigned run)
14   : m_producer(producer), m_run(run), m_ev(0), m_newest(0), m_connectedBoards(0) {
15   const std::vector<boost::shared_ptr<BoardConnection> >& boards = producer.GetBoards();
16   std::size_t nboards = boards.size();
17   m_trns.resize(nboards);
18   m_buffers.resize(nboards);
19   for (std::size_t i = 0; i < nboards; ++i) {
20     if (boards[i]) {
21       m_buffers[i] = boost::shared_ptr<std::map<
22                      boost::uint32_t, std::vector<boost::uint8_t> > >(
23                        new std::map<boost::uint32_t, std::vector<boost::uint8_t> >());
24       boards[i]->StartRun(m_run);
25       ++m_connectedBoards;
26     }
```

```
27    }
28  }
29
30  BoardEventHandler::~BoardEventHandler() {
31    try {
32      m_producer.SendEvent(eudaq::RawDataEvent::EORE(m_producer.EVENT_TYPE, m_run, ++m_ev));
33      m_producer.SetStatus(eudaq::Status::LVL_OK, "Stopped");
34      BOOST_FOREACH(const boost::shared_ptr<BoardConnection>& board, m_producer.GetBoards()) {
35        if (board)
36          board->StopRun();
37      }
38    } catch (std::exception& e) {
39      std::cerr << "Exception " << e.what() << std::endl;
40    } catch (...) {
41      std::cerr << "Unknown exception\n";
42    }
43  }
44
45  void BoardEventHandler::OnEvent(
46    int addr, boost::uint16_t trn, const std::vector<boost::uint8_t>& rawData) {
47    // expand trigger # to 32 bits, (tlu trigger number is 15-bit)
48    boost::uint32_t current = trn;
49    if (!m_trns[addr])
50      m_trns[addr] = boost::shared_ptr<boost::uint32_t>(new boost::uint32_t(current));
51    else {
52      current += *m_trns[addr] & 0xFFFF8000;
53      if (trn < (*m_trns[addr] & 0x7FFF))
54        current += 0x8000;
55      *m_trns[addr] = current;
56    }
57
58    std::size_t completed = 0;
59    std::size_t nbufs = m_buffers.size();
60    for (std::size_t i = 0; i < nbufs; ++i) {
61      if (m_buffers[i] && m_buffers[i]->find(current) != m_buffers[i]->end())
62        completed++;
63    }
64
65    if (completed == m_connectedBoards - 1) {
66      // flush the event when have the complete set
67      // of blocks with the same trigger #
68      eudaq::RawDataEvent event(m_producer.EVENT_TYPE, m_run, m_ev);
69      // add blocks from queues of boards <addr
70      for (std::size_t i = 0; i < static_cast<std::size_t>(addr); ++i) {
71        if (m_buffers[i]) {
72          std::map<boost::uint32_t, std::vector<boost::uint8_t> >::iterator it =
73            m_buffers[i]->find(current);
74          event.AddBlock(i, it->second);
75          m_buffers[i]->erase(it);
76        }
77      }
78      // add block from current board
79      event.AddBlock(addr, rawData);
80      // add blocks from queues of boards >addr
81      for (std::size_t i = addr + 1; i < nbufs; ++i) {
82        if (m_buffers[i]) {
83          std::map<boost::uint32_t, std::vector<boost::uint8_t> >::iterator it =
84            m_buffers[i]->find(current);
85          event.AddBlock(i, it->second);
86          m_buffers[i]->erase(it);
87        }
88      }
89      // append recently collected tags to the event
90      std::multimap<std::string, std::string>::iterator it;
91      std::multimap<std::string, std::string>::iterator eit =
92        m_tags.end();
93      for (it = m_tags.begin(); it != eit; ++it)
94        event.SetTag(it->first, it->second);
95      m_tags.clear();
96      // and send the event
97      m_producer.SendEvent(event);
98      m_ev++;
99    } else {
100     // or store it and wait for matching events from other board(s)
101     if (m_buffers[addr])
102       m_buffers[addr]->insert(std::make_pair(current, rawData));
103   }
104
105   // update trigger # of newest event
106   // and remove blocks that are old enough to assume they'll never be complete.
107   if (current > m_newest) {
108     m_newest = current;
```

```
109     for (std::size_t i = 0; i < nbufs; ++i) {
110       if (m_buffers[i]) {
111         static const std::size_t SYNC_QUEUE_SIZE = 50;
112         if (m_buffers[i]->size() > SYNC_QUEUE_SIZE) {
113           m_buffers[i]->erase(
114             m_buffers[i]->begin(),
115             m_buffers[i]->lower_bound(m_newest - SYNC_QUEUE_SIZE));
116         }
117       }
118     }
119   }
120 }
121
122 void  BoardEventHandler::OnMonitor(int addr, const std::string& str) {
123   EUDAQ_INFO(std::string("Board ") + lexical_cast<std::string>(addr) + ": " + str);
124
125   static const boost::regex tagex("@(.*)=(.*)");
126   boost::smatch match;
127   if (boost::regex_match(str, match, tagex))
128     m_tags.insert(std::make_pair(match[1], match[2]));
129 }
130
131 void  BoardEventHandler::OnError(int addr, const std::string& str) {
132   EUDAQ_ERROR(std::string("Board ") + lexical_cast<std::string>(addr) + ": " + str);
133 }
```

Listing A.10: `fcal/src/BoardEventHandler.cc`

```
1  #ifndef _FCAL_COMMANDCONFIGURE_H__
2  #define _FCAL_COMMANDCONFIGURE_H__
3
4  #include "BoardConnection.hh"
5  #include "eudaq/Configuration.hh"
6  #include <boost/smart_ptr.hpp>
7
8  class FCALProducer;
9
10 namespace fcal {
11 struct ConfigureImpl;
12
13 class CommandConfigure {
14 public:
15   CommandConfigure(FCALProducer& producer, const eudaq::Configuration& param);
16   void operator()();
17 private:
18   boost::shared_ptr<fcal::ConfigureImpl> m_impl;
19 };
20 } // namespace fcal
21
22 #endif // !_FCAL_COMMANDCONFIGURE_H__
```

Listing A.11: `fcal/include/CommandConfigure.hh`

```
1  #include "CommandConfigure.hh"
2
3  #include "FCALProducer.hh"
4  #include "eudaq/Logger.hh"
5
6  #include <iostream>
7  #include <boost/lexical_cast.hpp>
8  #include <boost/foreach.hpp>
9
10 using boost::lexical_cast;
11 using namespace fcal;
12
13 namespace fcal {
14 struct ConfigureImpl : public boost::enable_shared_from_this<ConfigureImpl> {
15   ConfigureImpl(FCALProducer& producer, const eudaq::Configuration& param)
16     : producer(producer), param(param), boardsToConfigure(0) {}
17   void Execute();
18   boost::shared_ptr<fcal::BoardConnection> ConnectToBoard(int addr);
19   void StartConfiguration(int addr);
```

```
20    FCALProducer& producer;
21    eudaq::Configuration param;
22    std::size_t boardsToConfigure;
23  };
24
25  struct BoardCommandConfigure : public BoardCommand {
26    BoardCommandConfigure(boost::shared_ptr<ConfigureImpl> impl, const std::string& request)
27      : BoardCommand(request), impl(impl), last(false) {}
28
29    virtual void OnResponse();
30
31    virtual void OnTimeout() {
32      impl->producer.SetStatus(eudaq::Status::LVL_ERROR, "Timeout error");
33    }
34
35    virtual void OnError(const boost::system::error_code& error) {
36      impl->producer.SetStatus(eudaq::Status::LVL_ERROR, "Error: " + error.message());
37    }
38
39    boost::shared_ptr<ConfigureImpl> impl;
40    bool last;
41  };
42  } // namespace fcal
43
44  // functions below are ordered in the same order they are called
45  // 1. FCALProducer creates CommandConfigure and post()s it to the io_service
46  // 2. io_service calls CommandConfigure::operator()()
47  // 3. operator()() calls ConfigureImpl::Execute(), which calls ConnectToBoard and
48  // StartConfiguration for each board
49  // 4. StartConfiguration queues board commands in BoardConnection
50  // 5. received callbacks from BoardConnection to OnResponse, OnError or OnTimeout
51  // set the final status of the configuration
52
53  CommandConfigure::CommandConfigure(FCALProducer& producer, const eudaq::Configuration& param)
54    : m_impl(boost::shared_ptr<ConfigureImpl>(new ConfigureImpl(producer, param))) {}
55
56  void CommandConfigure::operator()() {
57    m_impl->Execute();
58  }
59
60  void ConfigureImpl::Execute() {
61    try {
62      std::cout << "Configuring (" << param.Name() << ")..." << std::endl;
63      std::vector<boost::shared_ptr<BoardConnection> >& boards = producer.GetBoards();
64      std::size_t nboards = param.Get("boards", 0);
65      boards.clear();
66      boards.resize(nboards);
67      std::size_t connected = 0;
68      for (std::size_t i = 0; i < nboards; ++i)
69        boards[i] = ConnectToBoard(i);
70      // start configuring all boards
71      for (std::size_t i = 0; i < nboards; ++i) {
72        if (boards[i]) {
73          StartConfiguration(i);
74          connected++;
75        }
76      }
77      std::cout << "Number of connected boards: " << connected << std::endl;
78      // the rest will be done asynchronously.
79    } catch (const std::exception & e) {
80      producer.SetStatus(eudaq::Status::LVL_ERROR, std::string("Exception: ") + e.what());
81    }
82  }
83
84  boost::shared_ptr<BoardConnection> ConfigureImpl::ConnectToBoard(int addr) {
85    std::cout << param.Name() << "\n";
86    std::string name = "b" + lexical_cast<std::string>(addr);
87    std::string empty;
88    std::string mon_port = param.Get(name + ".mon.port", empty);
89    std::string daq_port = param.Get(name + ".daq.port", empty);
90    std::string cmd_port = param.Get(name + ".cmd.port", empty);
91    boost::shared_ptr<BoardConnection> result;
92    if (mon_port.length() > 0 && daq_port.length() > 0 && cmd_port.length() > 0) {
93      try {
94        result = boost::shared_ptr<BoardConnection>(new BoardConnection(
95                     addr,
96                     producer.GetIoService(),
97                     mon_port, param.Get(name + ".mon.baud", 115200),
98                     daq_port, param.Get(name + ".daq.baud", 4000000),
99                     cmd_port, param.Get(name + ".cmd.baud", 115200)));
100       std::cout << "Connected to board " << name << std::endl;
101       EUDAQ_INFO("Connected to board " + name);
```

```
102       } catch (std::exception& e) {
103         EUDAQ_ERROR("Could not connect to board " + name);
104       }
105     } else {
106       std::cout << "No configuration found for board " << name
107                 << "\nProvide at least port names!\n" << std::endl;
108     }
109     return result;
110   }
111
112   void ConfigureImpl::StartConfiguration(int addr) {
113     const eudaq::Configuration::section_t& sect = param.GetSectionData();
114     std::vector<boost::shared_ptr<BoardCommandConfigure> > commands;
115     eudaq::Configuration::section_t::const_iterator it = sect.begin();
116     eudaq::Configuration::section_t::const_iterator eit = sect.end();
117     while (it != eit) {
118       static const boost::regex e("^B(\\d+)\\.(\\d+)\\.(.*)",
119                                   boost::regex::perl | boost::regex::icase);
120       boost::smatch match;
121       if (boost::regex_match(it->first, match, e)) {
122         int a = lexical_cast<int>(match[1]);
123         if (a == addr) {
124           std::string request = match[3];
125           if (it->second.length() > 0) {
126             request += ' ';
127             request += it->second;
128           }
129           boost::shared_ptr<BoardCommandConfigure> bcmd(
130             new BoardCommandConfigure(shared_from_this(), request));
131           std::cout << "Board " << addr << " config: '" << request << "'\n";
132           commands.push_back(bcmd);
133         }
134       }
135       ++it;
136     }
137     if (!commands.empty()) {
138       commands.back()->last = true;
139       boost::shared_ptr<BoardConnection> board = producer.GetBoards().at(addr);
140       BOOST_FOREACH(boost::shared_ptr<BoardCommandConfigure> bcmd, commands) {
141         board->QueueCommand(bcmd);
142       }
143       boardsToConfigure++;
144     }
145   }
146
147   void BoardCommandConfigure::OnResponse() {
148     try {
149       std::size_t nlines = m_response.size();
150       for (std::size_t i = 0; i < nlines; ++i)
151         std::cout << m_response[i] << "\n";
152       if (last) {
153         if (--impl->boardsToConfigure == 0) {
154           EUDAQ_INFO("Configured (" + impl->param.Name() + ")");
155           impl->producer.SetStatus(eudaq::Status::LVL_OK,
156                                    "Configured (" + impl->param.Name() + ")");
157         }
158       }
159     } catch (const std::exception & e) {
160       impl->producer.SetStatus(eudaq::Status::LVL_ERROR, std::string("Exception: ") + e.what());
161     }
162   }
```

Listing A.12: `fcal/sec/CommandConfigure.cc`

Appendix B

DVD-ROM contents

Sources, documentation and binary images of the firmware are located in the `daq` directory. It contains the following files:

`eudaq.tar.bz2` — Copy of SVN archive with EUDAQ sources, including the following files and directories related to `FCAL`:

- **`trunk/fcal`** — Source code of the `FCALProducer`.
- **`trunk/main/src/FCALConverterPlugin.cc`** — Source code of the `FCALConverterPlugin`.

`fw.tar.bz2` — Source code archive of the firmware, including tests and the initial Linux port.

`fw-doc.tar.bz2` — Documentation for the firmware sources generated with Doxygen.

`bintotext.py` — Script to decode binary data produced by the firmware.

`tb2_setup_manual.pdf` — User manual for the FCAL readout system.

Files related to Linux installation on Avnet Virtex-5 FXT Evaluation Kit covered in Chapter 5 are in the directory `v5fxt_linux`. It contains the following archives:

`buildroot-2011.05.tar.bz2` — Buildroot, including configuration used in the described installation and some downloaded packages.

`device-tree.tar.bz2` — Device tree generator.

`hardware_design.zip` — ISE project in which the FPGA configuration was created.

`images.zip` — Binary images ready to program into FPGA and flash memory:

- **`avnet-flash.dtb`** — Compiled device tree with flash memory partition set as root filesystem.
- **`avnet-ram.dtb`** — Compiled device tree with ramdisk set as root filesystem and flash partitions made available to the kernel.
- **`cpu.bin`** — FPGA bitstream in raw binary format, can be programmed into flash memory with U-Boot or SDK.
- **`cpu.bit`** — FPGA bitstream for direct programming into FPGA using iMPACT.
- **`cpu.mcs`** — FPGA bitstream in MCS format, can be used to program flash in iMPACT.

- **ramdisk.image.gz** — Original ramdisk image downloaded from Xilinx Open Source Wiki website.
- **rootfs.bin** — Root file system image (JFFS2) generated with Buildroot.
- **rootfs.tar** — Tar archive with the contents of the root file system.
- **u-boot.bin** — U-Boot image ready to program with SDK or update with U-Boot itself.
- **u-boot-env.bin** — U-Boot environment image with download and program “scripts”.
- **uImage** — Linux kernel image for U-Boot.
- **uramdisk** — U-Boot ramdisk image made from `ramdisk.image.gz`.
- **daqtest** — Compiled program used to test the data acquisition performance after porting to Linux on PowerPC.
- **clitest** — Compiled program used to test the command interface after porting to Linux on PowerPC.

linux-2.6-xlnx.tar.bz2 — Linux source tree from Xilinx git repository with updated device tree for Avnet board.

sdk_workspace.zip — Xilinx SDK projects used to generate device tree.

u-boot-xlnx.tar.bz2 — U-Boot source tree from Xilinx git repository with Avnet board definition added.

Finally, the electronic version of this thesis is included in file `thesis.pdf`.

Bibliography

[1] A. Djouadi, J. Lykken, K. Mönig, Y. Okada, M. Oreglia, and S. Yamashita, Eds., *International Linear Collider Reference Design Report*. ILC Global Design Effort and World Wide Study, Aug. 2007, vol. 2: Physics at the ILC. [Online]. Available: http://ilcdoc.linearcollider.org/record/6321/files/ILC_RDR_Volume_2-Physics_at_the_ILC.pdf [Accessed: 1 August 2011].

[2] *What is the ILC? – The project*. [Online]. Available: http://www.linearcollider.org/about/What-is-the-ILC/The-project [Accessed: 14 June 2011].

[3] *Compact Linear Collider website*. [Online]. Available: http://clic-study.org/ [Accessed: 12 Aug 2011].

[4] *What do we already know? The standard package*, CERN CMS website. [Online]. Available: http://cms.web.cern.ch/cms/Physics/StandardPackage/index.html [Accessed: 3 August 2011].

[5] *File:Standard Model of Elementary Particles.svg*. [Online]. Available: http://commons.wikimedia.org/wiki/File:Standard_Model_of_Elementary_Particles.svg [Accessed: 1 August 2011].

[6] *Energy scale*, Interactions.org – Particle Physics News and Resources, 2007. [Online]. Available: http://www.interactions.org/cms/?pid=2100&image_no=OT0101 [Accessed: 3 August 2011].

[7] *Finding the Higgs boson*, CERN CMS website. [Online]. Available: http://cms.web.cern.ch/cms/Physics/HuntingHiggs/CMS.html [Accessed: 3 August 2011].

[8] *Are there more particles left to find? Supersymmetry: uniting the forces*, CERN CMS website. [Online]. Available: http://cms.web.cern.ch/cms/Physics/Supersymmetry/index.html [Accessed: 3 August 2011].

[9] The ATLAS collaboration, *Combination of the Searches for the Higgs Boson in* $\sim 1\mathrm{fb}^{-1}$ *of Data Taken with the ATLAS Detector at 7 TeV Center-of-Mass Energy*, Aug. 2011. [Online]. Available: http://cdsweb.cern.ch/record/1375549 [Accessed: 8 Sep 2011].

[10] CMS Collaboration, *SM Higgs Combination*, Jul. 2011. [Online]. Available: http://cdsweb.cern.ch/record/1370076/ [Accessed: 8 Sep 2011].

[11] *Synchrotron Radiation*, Hyperphysics. [Online]. Available: http://hyperphysics.phy-astr.gsu.edu/hbase/particles/synchrotron.html [Accessed: 6 August 2011].

[12] N. Phinney, N. Toge, and N. Walker, Eds., *International Linear Collider Reference Design Report*. ILC Global Design Effort and World Wide Study, Aug. 2007, vol. 3: Accelerator. [Online]. Available: http://ilcdoc.linearcollider.org/record/6321/files/ILC_RDR_Volume_3-Accelerator.pdf [Accessed: 1 August 2011].

[13] *A schematic layout of the International Linear Collider*, Interactions.org – Particle Physics News and Resources, 2007. [Online]. Available: http://www.interactions.org/cms/?pid=2100&image_no=OT0104 [Accessed: 3 August 2011].

[14] T. Behnke, C. Damerell, J. Jaros, and A. Miyamoto, Eds., *International Linear Collider Reference Design Report*. ILC Global Design Effort and World Wide Study, Aug. 2007, vol. 4: Detectors. [Online]. Available: http://ilcdoc.linearcollider.org/record/6321/files/ILC_RDR_Volume_4-Detectors.pdf [Accessed: 1 August 2011].

[15] *The ILD web site*. [Online]. Available: http://www.ilcild.org/ [Accessed: 8 August 2011].

[16] *The ILD web site*. [Online]. Available: http://www.ilcild.org/ild-detector-systems [Accessed: 8 August 2011].

[17] *International Large Detector – Letter of intent*, The ILD concept group. [Online]. Available: http://www.ilcild.org/documents/ild-letter-of-intent/LOI.pdf [Accessed: 14 June 2011].

[18] *Particle Flow – Particle reconstruction*. [Online]. Available: http://www-flc.desy.de/hcal/basics/pfa.php [Accessed: 12 August 2011].

[19] *FCAL Collaboration*, FCAL Collaboration. [Online]. Available: http://fcal.desy.de/ [Accessed: 20 Aug 2011].

[20] M. Idzik, S. Kulis, and D. Przyborowski, “Development of front-end electronics for the luminosity detector at ilc,” *Nucl. Inst. and Meth.*, vol. A608, pp. 169–174, 2009.

[21] M. Idzik, K. Świentek, T. Fiutowski, S. Kulis, and P. Ambalathankandy, “A power scalable 10-bit pipeline adc for luminosity detector at ilc,” *JINST*, vol. 6 P01004, 2011.

[22] I. Melzer-Pellmann and N. Meyners, *Test Beams at DESY*, Mar. 2011. [Online]. Available: http://adweb.desy.de/~testbeam/ [Accessed: 12 Aug 2011].

[23] I.-M. Gregor, *ZEUS MVD Telescope*, Aug. 2011. [Online]. Available: http://www.desy.de/~gregor/MVD_Telescope/short_intro.html [Accessed: 12 Aug 2011].

[24] D. Cussans, *Description of the JRA1 Trigger Logic Unit (TLU), v0.2c*, Sep. 2009. [Online]. Available: http://www.eudet.org/e26/e28/e42441/e57298/EUDET-MEMO-2009-04.pdf [Accessed: 12 Aug 2011].

[25] J. Błocki, W. Daniluk, E. Kielar, J. Kotuła, A. Moszczyński, K. Oliwa, B. Pawlik, W. Wierba, L. Zawiejski, and J. Aguilar, *Silicon Sensors Prototype for LumiCal Calorimeter*, Dec. 2009. [Online]. Available: http://www.eudet.org/e26/e28/e42441/e68451/EUDET-MEMO-2009-07.doc [Accessed: 13 August 2011].

[26] *Spartan-3E FPGA Family Data Sheet*, Xilinx Inc., Aug. 2009. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds312.pdf [Accessed: 13 August 2011].

[27] *ATxmega64A1/128A1/192A1/256A1/384A1 Preliminary*, Atmel Corporation, Sep. 2010. [Online]. Available: http://www.atmel.com/dyn/products/product_card.asp?part_id=4298 [Accessed: 13 August 2011].

[28] *FT4232H Quad High Speed USB to Multipurpose UART/MPSSE IC*, Future Technology Devices International Ltd., 2010. [Online]. Available: http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT4232H.pdf [Accessed: 13 August 2011].

[29] *The FreeRTOS Project*, Real Time Engineers Ltd., 2011. [Online]. Available: http://www.freertos.org/ [Accessed: 18 Jun 2011].

[30] *μC/OS-II Kernel*, Micrium, 2011. [Online]. Available: http://www.micrium.com/page/products/rtos/os-ii [Accessed: 18 Jun 2011].

[31] *AVR XMEGA: Documents*, Atmel Corporation, 2011. [Online]. Available: http://www.atmel.com/dyn/products/documents.asp?category_id=163&family_id=607&subfamily_id=1965 [Accessed: 16 Jun 2011].

[32] *GCC, the GNU Compiler Collection*, Free Software Foundation, Inc., Sep. 2011. [Online]. Available: http://gcc.gnu.org/ [Accessed: 12 Sep 2011].

[33] *IAR Embedded Workbench for Atmel AVR*, IAR Systems. [Online]. Available: http://www.iar.com/website1/1.0.1.0/107/1/ [Accessed: 12 Sep 2011].

[34] R. M. Stallman and the GCC Developer Community, *Using the GNU Compiler Collection*, Free Software Foundation, Inc., 2008, For gcc version 4.3.6. [Online]. Available: http://gcc.gnu.org/onlinedocs/gcc-4.3.6/gcc.pdf [Accessed: 20 Aug 2011].

[35] *AVR Libc Home Page*, Feb. 2011. [Online]. Available: http://www.nongnu.org/avr-libc/ [Accessed: 12 Sep 2011].

[36] *GNU Binutils*, Free Software Foundation, Inc., Sep. 2011. [Online]. Available: http://www.gnu.org/software/binutils/ [Accessed: 12 Sep 2011].

[37] *AVRDUDE - AVR Downloader/UploaDEr*, Jan. 2010. [Online]. Available: http://www.nongnu.org/avrdude/ [Accessed: 12 Sep 2011].

[38] *avr-libc 1.7.1*, Free Software Foundation, Inc., Feb. 2011. [Online]. Available: http://savannah.nongnu.org/download/avr-libc/avr-libc-user-manual-1.7.1.pdf.bz2 [Accessed: 21 Aug 2011].

[39] H. Sutter, "Virtuality," *C/C++ Users Journal*, vol. 19(9), Sep. 2001. [Online]. Available: http://www.gotw.ca/publications/mill18.htm [Accessed: 26 Jul 2011].

[40] R. Helm, E. Gamma, J. Vlissides, and R. Johnson, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Nov. 1994.

[41] A. Alexandrescu, *The D Programming Language*. Addison-Wesley Professional, Jun. 2010.

[42] *The GNU Make Manual*, Free Software Foundation, Inc., 2010. [Online]. Available: http://www.gnu.org/software/make/manual/make.html [Accessed: 22 Aug 2011].

[43] P. Miller, "Recursive make considered harmful," *AUUGN Journal of AUUG Inc.*, vol. 19(1), pp. 14–25, 1998. [Online]. Available: http://aegis.sourceforge.net/auug97.pdf [Accessed: 14 Jul 2011].

[44] M. M. Robert C. Martin, *Agile Principles, Patterns, and Practices in C#*. Englewood Cliffs: Prentice Hall, 2007.

[45] Wikipedia, *List of unit testing frameworks — Wikipedia, The Free Encyclopedia*, 2011. [Online]. Available: http://en.wikipedia.org/w/index.php?title=List_of_unit_testing_frameworks&oldid=445735840 [Accessed: 20 Aug 2011].

[46] J. J. Labrosse, *MicroC/OS-II, The Real Time Kernel*, 2nd ed. Newnes, 2002.

[47] *VT100 User Guide*, 3rd ed., Digital, Jun. 1981, Electronic reproduction in 2001. [Online]. Available: http://vt100.net/docs/vt100-ug/ [Accessed: 24 Jul 2011].

[48] *Standard Commands for Programmable Instruments (SCPI)*, SCPI Consortium, May 1999. [Online]. Available: www.ivifoundation.org/docs/scpi-99.pdf [Accessed: 16 Aug 2011].

[49] *gnuplot homepage*, Mar. 2011. [Online]. Available: http://www.gnuplot.info/ [Accessed: 22 Aug 2011].

[50] S. Kulis, A. Matoga, M. Idzik, K. Świentek, T. Fiutowski, and D. Przyborowski, "A multichannel triggerless system for any-type detectors," 2011, To be published.

[51] E. Corrin, *EUDAQ Software User Manual*, Apr. 2010. [Online]. Available: www.eudet.org/e26/e28/e86887/e86890/EUDET-Memo-2010-01.pdf [Accessed: 28 Aug 2011].

[52] *ROOT: A Data Analysis Framework*. [Online]. Available: http://root.cern.ch/drupal/ [Accessed: 28 Aug 2011].

[53] C. M. Kohlhoff, *Boost.Asio*, Jul. 2011. [Online]. Available: http://www.boost.org/doc/libs/1_47_0/doc/html/boost_asio.html [Accessed: 19 Aug 2011].

[54] S. Kulis and M. Idzik, "Progress on signal processing techniques for lumical testbeam and clic detectors," in *Proceedings of the 18th FCAL Collaboration Workshop*, Jun. 2011, pp. 42–46. [Online]. Available: http://www.nipne.ro/fcal_2011/docs/Proceedings_FCAL_RO_2011.pdf [Accessed: 17 Sep 2011].

[55] K. Yaghmour, *Building Embedded Linux Systems*. O'Reilly Media, Apr. 2003.

[56] J. G. Tong, I. D. L. Anderson, and M. A. S. Khalid, "Soft-core processors for embedded systems," in *The 18th International Confernece on Microelectronics (ICM)*, 2006. [Online]. Available: http://www.reds.ch/share/cours/ReCo/documents/soft_core_processors.pdf [Accessed: 5 May 2011].

[57] *Virtex-5 Family Overview*, Xilinx. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf [Accessed: 1 May 2011].

[58] *Xilinx Virtex-5 FXT Evaluation Kit User Guide*, Avnet, May 2008. [Online]. Available: http://www.files.em.avnet.com/files/177/xlx_v5fxt_evl-ug-rev1.pdf [Accessed: 25 April 2011].

[59] *3 Volt Intel StrataFlash Memory Preliminary Datasheet*, Intel, Apr. 2001. [Online]. Available: http://www.datasheetcatalog.org/datasheets/400/354802_DS.pdf [Accessed: 7 May 2011].

[60] *Design Resource Center*, Avnet. [Online]. Available: http://www.em.avnet.com/drc [Accessed: 1 May 2011].

[61] *Xilinx Open Source Wiki*. [Online]. Available: http://xilinx.wikidot.com/ [Accessed: 27 April 2011].

[62] *Using U-Boot with the Xilinx ML507 Evaluation Platform*, Xilinx, Apr. 2009. [Online]. Available: http://xilinx.wdfiles.com/local--files/embedded-linux/uboot-ml507-howto.pdf [Accessed: 2 May 2011].

[63] *Xilinx Downloads*, Xilinx, Inc. [Online]. Available: http://www.xilinx.com/support/download/ [Accessed: 8 May 2011].

[64] *Embedded Linux Development Kit*, DENX Software Engineering, Nov. 2008. [Online]. Available: http://www.denx.de/wiki/DULG/ELDK [Accessed: 27 April 2011].

[65] E. Andersen and T. B. developers, *Buildroot: making Embedded Linux easy*, 2011. [Online]. Available: http://buildroot.uclibc.org/ [Accessed: 4 July 2011].

[66] *git.xilinx.com Git*. [Online]. Available: http://git.xilinx.com/ [Accessed: 27 April 2011].

[67] *Xilinx Board Definition (XBD) Files for Avnet's Xilinx Development Boards*, Avnet. [Online]. Available: http://www.em.avnet.com/sta/home/0,4610,CID%253D13747%2526CCD%253DUSA%2526SID%253DNoNav%2526DID%253DADA%2526LID%253D6448%2526BID%253DDF2%2526CTP%253DSTA,00.html [Accessed: 1 May 2011].

[68] V. Asokan, *Designing Multiprocessor Systems in Platform Studio*, Xilinx, Apr. 2007. [Online]. Available: http://japan.xilinx.com/support/documentation/white_papers/wp262.pdf [Accessed: 26 April 2011].

[69] *Device Tree Generator*. [Online]. Available: http://xilinx.wikidot.com/device-tree-generator [Accessed: 2 May 2011].

[70] *Das U-Boot – the Universal Boot Loader*. [Online]. Available: http://www.denx.de/wiki/U-Boot [Accessed: 28 April 2011].

[71] *U-Boot Design Principles*. [Online]. Available: http://www.denx.de/wiki/U-Boot/DesignPrinciples [Accessed: 28 April 2011].

[72] *PowerPC Linux*. [Online]. Available: http://xilinx.wikidot.com/powerpc-linux [Accessed: 2 May 2011].

[73] *Virtex-5 FPGA Configuration User Guide*, Xilinx, May 2010. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug191.pdf [Accessed: 7 May 2011].

[74] [Online]. Available: http://www.files.em.avnet.com/files/178/xlx_v5fxt_evl-sch-revb.pdf.pdf [Accessed: 1 May 2011].

[75] A. Matoga on behalf of AGH-UST and IFJ PAN, "Prototype readout system for beamtest of ILC forward calorimetry detector modules," in *Proceedings of the 18th FCAL Collaboration Workshop*, Jun. 2011, pp. 31–36. [Online]. Available: http://www.nipne.ro/fcal_2011/docs/Proceedings_FCAL_RO_2011.pdf [Accessed: 17 Sep 2011].

List of Listings

3.1 Example of the error handling convention used in the firmware source code. 33
3.2 Example of the interface declaration convention used in the firmware source code. 34
3.3 Example of the convention of implementing interfaces in the firmware source code. 34
3.4 The Non-Virtual Interface idiom applied to the `Stream` interface from Listing 3.2. . 35
3.5 An example illustrating the convention of implementing unit tests in the firmware source code. . . . 41
3.6 Excerpts from `fcaldc.h` file, which contains mapping of the data concentrator registers to the AVR data address space. . . . 48
3.7 Data types used to define the command tree structure. . . . 51
3.8 Algorithm to parse and execute a hierarchical command. . . . 51
3.9 Algorithm to iterate over possible completions of given initial part of a command path. . . . 53
4.1 EUDAQ configuration file containing options for the main EUDAQ process, the Run Control, as well as connection settings for the readout board and commands to send to it. . . . 68
5.1 Definition of Makefile target for building the U-Boot bootloader for the Avnet Evaluation Kit. . . . 81
A.1 `Makefile` . . . 100
A.2 `libdaq/daq_proc.h` . . . 101
A.3 `libdaq/daq_proc.c` . . . 103
A.4 `fcal/daq_producer.c` . . . 105
A.5 `libdaq/binary_event_writer.c` . . . 106
A.6 `libdaq/event.h` . . . 108
A.7 `fcal/include/FCALProducer.hh` . . . 109
A.8 `fcal/include/FCALProducer.cxx` . . . 111
A.9 `fcal/include/BoardEventHandler.hh` . . . 111
A.10 `fcal/src/BoardEventHandler.cc` . . . 113
A.11 `fcal/include/CommandConfigure.hh` . . . 113
A.12 `fcal/sec/CommandConfigure.cc` . . . 115

List of Figures

1.1 Fundamental particles of the Standard Model [5]. 16
1.2 Energy scale for the unification of fundamental interactions. 17
1.3 A schematic layout of the International Linear Collider. 18
1.4 ILC beam structure and timing. 18
1.5 Layout of ILD model used in simulations 20
1.6 Traces left in tracker and calorimeters by different particles. 21

2.1 Position of LumiCal within the very forward region. 24
2.2 Overview of the structure of LumiCal. 24
2.3 Schematic diagram of the test beam setup. 25
2.4 Photograph of the test beam setup (O. Novgorodova). 25
2.5 Signal and data flow in the FCAL DAQ system. 26
2.6 Data acquisition timing diagrams, showing relationships between signals for a single readout board and the whole setup. 26
2.7 Photograph of the sensor board and the readout board connected to each other. . . 27

3.1 Top level architecture of readout board microcontroller firmware. 31
3.2 Subset of the source directory tree of the microcontroller firmware used in the explanation of the concepts of the build system. 37
3.3 Dependencies between modules in the tree from Figure 3.2. 37
3.4 Example results of test coverage analysis with gcov. 42
3.5 The `Stream` class and its implementations. 46
3.6 Class diagram for command line interface. 49
3.7 Example command tree (on the left) and list of valid commands in that tree (on the right). 50
3.8 UML timing diagram presenting example DAQ session in which 3 events were sent. 54
3.9 Example event with 4 samples from each of 8 selected channels transmitted in text protocol. 55
3.10 Structure of event packet in the binary protocol used to transmit data from the readout board to PC. 56
3.11 Diagram of classes implementing event building and transmission. 57

3.12 Firmware components involved in measuring operating conditions on-board integrated circuits. 58
3.13 Data acquisition performance (number of events per second) as a function of number of samples per packet. 60
3.14 Supply voltages and currents drawn by analog and digital parts of ADC ASICs in response to switching the power on and off. 61

4.1 Schematic diagram of the EUDAQ architecture. 64
4.2 Screenshot of graphical interface of the EUDAQ processes: the Run Control and the Log Collector. 65
4.3 Class diagram of the FCAL Producer. 66
4.4 Sequence diagram of connecting to readout boards and sending configuration commands to them. To simplify the diagram, error handling is omitted. 69
4.5 Screenshot of the RootMonitor plotting on-line distributions of amplitudes of pulses from selected channels during recording of the deposition of energy in the sensor by beam electrons. 71
4.6 Example time series for a single event from electromagnetic shower development measurements. 72
4.7 Amplitude distribution in about 100,000 events without absorber. 72

5.1 Flowchart showing the process of installation of an embedded system based on Linux. 74
5.2 Photograph of the Avnet Virtex-5 FXT Evaluation Kit. 76
5.3 Console output from the U-Boot bootloader first run. 81
5.4 Kernel image information displayed by `mkimage`. 83

List of Tables

3.1 Macros defined in `mt.h`, implementing the real-time kernel abstraction layer used to make components of the firmware independent of a particular kernel. 44
3.2 Special sequences defined in the binary protocol used to transmit event data from the readout board to PC, to make the receiver able to unambiguously extract individual packets from a continuous stream of bytes. 57

5.1 Peripherals (standard IP cores) implemented in the FPGA and their configuration. . 78
5.2 Proposed organization of flash memory partitions in the installed system. 89