



WARSAW UNIVERSITY OF TECHNOLOGY
Faculty of Electronics and Information Technology
Institute of Electronic Systems

Maciej Lipiński

198420

Master Thesis

**Universal Measurement System
with Web Interface**

Supervisor:

Ph.D. Krzysztof Poźniak

Warsaw, 2009

Acknowledgements

*Multitude of gratitude to **Grzegorz Kasrowicz** for continuous, instant and thorough support throughout the project.*

* * *

*Many thanks to **Zbigniew Reszela**, my colleague at CELLS synchrotron, for theoretical help in the field of application design and data structures.*

Abstract

Universal Measurement System with Web Interface

Modern trends in measurement instrument development include miniaturization and remote control. Remote control interfaces offered by measurement devices increasingly often include Graphic User Interface(GUI) which reflects the appearance of local user interface (i.e. screen, buttons). The later is one of the limitations in miniaturization. More and more often, measurement devices are used only remotely, in which case, the local interface is unnecessary or even unwanted. Instruments controlled by remote GUI displayed using a PC or laptop seem to be a new direction in the development of measurement devices.

Universal Measurement System with Web Interface (UMSWI) was created for High Energy Physics, i.e. accelerator diagnostics. UMSWI's hardware (commercially available) is a small, modular, embedded system, designed according to modern trends. It incorporates powerful microprocessor (capable of running embedded operating system) and Field Programmable Gate Array (enabling fast, concurrent data processing). In order to efficiently use the hardware resources and create a state-of-art measurement instrument, which follows modern trends, a control system (software and FGPA logic) needed to be created, the creation preceded by in-depth research of existing solutions and available technologies.

This thesis is a theoretical and practical study of UMSWI's control system which enables to manage the hardware and conduct measurement providing web-based and Standard Commands for Programmable Instrumentation (SCPI) interfaces. The project included implementation of simple digital oscilloscope and spectrum analyzer functionalities and GUIs. The device's innovative character is determined by the fact that no dedicated server or client software is required to operate it. Since the measurement system is simultaneously a server, it can be connected directly to an intranet, Internet or PC/laptop and accessed using only web browser.

Moreover, the control system, which has been created, enables easy extensions (i.e. implementation of frequency counter) and the modular hardware architecture allows to change the quantities measured (i.e. instead of using recorder module with Analog-to-Digital Converter, a weather station can be installed). Further more, the control system architecture is platform-independent and the system can be ported to any microprocessor capable of running Embedded Linux. Such features highlight system's universality.

Streszczenie

Uniwersalny System Pomiarowy z Interfejsem Webowym (USPIW)

Znaczenie zdalnego sterowania jest coraz większe i permanentnie rozszerza się spektrum jego zastosowań. Urządzenia pomiarowe w tym względzie nie stanowią wyjątku. Zdalnie sterowane instrumenty pomiarowe są coraz bardziej popularne, a w niektórych sytuacjach niezbędne. Podczas wykonywania pomiarów w miejscach niebezpiecznych muszą być one stosowane. Jednocześnie wygoda wykonywania pomiarów z biura lub jakiegokolwiek miejsca na świecie staje się coraz bardziej atrakcyjna. Większość nowoczesnych instrumentów pomiarowych daje możliwość zdalnej kontroli. Poza standardami służącymi do programowania i obsługi instrumentów pomiarowych z poziomu aplikacji pomiarowych (*LabView*), coraz częściej zdalna kontrola obejmuje GUI (Graphic User Interface). W tym przypadku wirtualny interfejs imituje wbudowany panel frontowy urządzenia (np. e*Scope firmy Tektronix [7]). Przyrządy pomiarowe podlegają ciągłej miniaturyzacji. Jednym z elementów ograniczających zmniejszenie rozmiarów jest konieczność umieszczenia w urządzeniu pomiarowym panelu sterującego z ekranem. Coraz częściej zdarza się także, że przyrządy pomiarowe wykorzystywane są wyłącznie w trybie zdalnym. Wówczas wbudowany interfejs lokalny przyrządu pomiarowego staje się niepotrzebny lub wręcz niepożądany, gdyż, po pierwsze utrudnia dalszą miniaturyzację urządzenia i po drugie stanowi niepotrzebny wydatek.

Zdalna kontrola urządzeń pomiarowych w postaci GUI odzwierciedlającego panel frontowy urządzenia już nie tylko stanowi dodatkową funkcjonalność, ale może skutecznie konkurować z lokalnym interfejsem wbudowanym, a nawet przewyższać go możliwościami. Pozwala ona na obsługę wielu urządzeń z jednego stanowiska (PC/laptop) czy łatwe pozyskiwanie danych pomiarowych do dalszej analizy. Co więcej, zawsze istnieje możliwość wykorzystania interfejsu zdalnego lokalnie ustawiając PC/laptop obok urządzenia pomiarowego. Dlatego nowym i rozwojowym kierunkiem w dziedzinie takich urządzeń wydają się być przyrządy pozbawione wbudowanego interfejsu użytkownika. Urządzenia te mogą być atrakcyjne zarówno dla użytkowników jak i producentów, gdyż zmniejszają koszt produkcji (brak wyświetlacza, itp.) oraz pozwalają na większą uniwersalizację przyrządów. Funkcjonalność urządzenia jest w dużej mierze zależna od interfejsu, a jeśli ten jest wirtualny, jego wymiana wymaga jedynie nowego oprogramowania. Urządzenia sterowane wyłącznie za pomocą zdalnego GUI pojawiły się już na rynku (np. BitScope [11]).

Uniwersalny System Pomiarowy z Interfejsem Webowym jest urządzeniem stworzonym na potrzeby Fizyki Wielkich Energii, m.in. diagnostyki akceleratorowej. Baza sprzętowa instrumentu (dostępna komercyjnie) jest miniaturowym, modułarnym urządzeniem wbudowanym, które zostało zaprojektowane zgodnie z najnowszymi trendami. Urządzenie to łączy mikroprocesor o dużych możliwościach obliczeniowych z układem logicznym FPGA (Field Programmable Gate Array). Mikroprocesor (ARM9 [45]) pozwala na uruchomienie systemu operacyjnego, zaś FPGA (ALTERA Cyklon I [21]) umożliwi szybkie, równoległe przetwarzanie danych. Aby w pełni wykorzystać możliwości sprzętowe tego urządzenia i stworzyć produkt wpisujący się w nowoczesne trendy rozwoju systemów pomiarowych, konieczne było wykonanie odpowiedniego systemu sterującego (oprogramowanie i układ logiczny)

poprzedzone przeglądem istniejących rozwiązań i możliwych do wykorzystania technologii.

Bardzo ważną konkluzją z przeglądu istniejących rozwiązań jest fakt, iż oferowane przez producentów instrumentów pomiarowych wirtualne zdalne panele graficzne (GUI) wymagają instalacji oprogramowania po stronie klienta lub przygotowania dedykowanego serwera z odpowiednim oprogramowaniem podłączonego do urządzenia pomiarowego.

Celem tej pracy było wykorzystanie dostępnej komercyjnie bazy sprzętowej do stworzenia autonomicznego i uniwersalnego systemu pomiarowego ze zdalnym sterowaniem opartym na interfejsie WWW. Niniejsza praca stanowi opracowanie teoretyczne i realizację systemu sterującego USPIW. System ten obejmuje logikę do FPGA, Embedded Linux zoptymalizowany i skonfigurowany na potrzeby USPIW, a także szereg aplikacji i rozwiązań umożliwiających kontrolę urządzenia i wykonywanie pomiarów z poziomu strony WWW oraz wybranego interfejsu pomiarowego (Standard Commands for Programmable Instrumentation). W ramach pracy i zgodnie z wymaganiami początkowymi, zaimplementowano GUI i funkcjonalność umożliwiające na wykorzystanie urządzenia jako prostego oscyloskopu cyfrowego i analizatora widma.

Podstawą budowy systemu sterującego USPIW jest system operacyjny Linux, co pozwala na uniezależnienie architektury USPIW od platformy sprzętowej oraz wykorzystanie istniejących rozwiązań czy aplikacji. System sterujący USPIW może zostać uruchomiony na dowolnym mikroprocesorze, na którym możliwe jest uruchomienie Linux'a. Linux dla USPIW stworzony został w oparciu o jądro 2.6.19 odpowiednio zmodyfikowane i skonfigurowane. System plików zaimplementowany został jako *initramfs* – wkompilowany w obraz z jądrem, ładowany do pamięci RAM przy starcie systemu.

Układ logiczny opisany w języku VHDL (Very High Speed Integrated Circuit hardware description language) i zaimplementowany w FPGA ma dwa zadania: obsługa komunikacji z mikroprocesorem i zarządzanie akwizycją danych. Akwizycja danych kontrolowana jest przez mikroprocesor przy pomocy szeregu parametrów zapisywanych w rejestrach kontrolnych FPGA (wykorzystując logikę obsługującą komunikację). Logika zarządzająca akwizycją na bieżąco kontroluje stan rejestrów i reaguje w odpowiedni sposób na zmianę ich zawartości. Komunikacja w przeciwnym kierunku (logika akwizycji->mikroprocesor) działa analogicznie. Rejestry kontrolne umieszczone są w obszarze adresowym mikroprocesora. Podczas akwizycji danych wartości napięcia odczytane z przetworników analogowo-cyfrowych zapisywane są w pamięci SSRAM (Synchronous Static Random Access Memory). Po zakończeniu akwizycji następuje odczyt danych z SSRAM do mikroprocesora, przesłanie do klienta i przetworzenie do formy graficznej.

Komunikacja z FPGA po stronie mikroprocesora i systemu operacyjnego zapewniona jest przez stworzony do tego celu sterownik do Linux'a (Linux Device Driver). Sterownik ten pozwala na komunikację z logiką zaimplementowaną w FPGA na różnych poziomach abstrakcji (ogólny, wyspecjalizowany) i różnymi metodami (przez *ioctl* lub system plików */proc*).

Urządzenie obsługiwane jest z poziomu strony WWW. Głównymi jej składnikami są: Interfejs Oscyloskopu i Analizatora Widma oraz Interfejs Zarządzania Urządzeniem. Strona WWW Uniwersalnego Systemu Pomiarowego z Interfejsem Webowym dostarcza dodatkowo krótką informację o projekcie, instrukcje obsługi, oraz przykłady zastosowań (skrypty *Matlab*). Bardzo ważną kwestią podczas pracy nad projektem był wybór odpowiednich technologii do stworzenia Interfejsu

Oscyloskopu i Analizatora Widma. Spośród wielu możliwości rozwiązania tego zadania i technologii możliwych do zastosowania w jego realizacji, wybrano implementację GUI jako Apletu Java'owego. Komunikacja apletu ze sterownikiem Linux'owym, a w konsekwencji z logiką FPGA, odbywa się z wykorzystaniem Common Gate Interface (CGI). Zastosowanie Apletu Java'owego, który wykonywany jest w przeglądarce na komputerze klienta oraz CGI pozwoliło na przeniesienie wymagań na moc obliczeniową z ograniczonego w zasobach mikroprocesora USPIW na komputer klienta. Takie zadania jak generacja grafiki, interakcja z użytkownikiem, przechowywanie danych pomiarowych odbywają się po stronie klienta, nie obciążają USPIW i redukują ilość przesyłanych informacji między serwerem (znajdującym się w USPIW) i klientem. Architektura Apletu Java'owego oparta jest o wzór Model-View-Controller (MVC) [61], który umożliwia dokonywanie zmian w każdym z trzech komponentów architektury (modelu danych, interfejsie użytkownika, logice sterowania) niezależnie. Sprawia to, iż aplet może zostać łatwo rozszerzony o nowe funkcje lub wykorzystany do implementacji całkiem nowych zadań.

Interfejs Zarządzania Urządzeniem wykorzystuje CGI do wywoływania funkcji systemowych lub uruchamiania aplikacji oraz Java Script do weryfikacji danych wejściowych.

Zastosowane technologie oraz fakt, iż urządzenie pomiarowe jest jednocześnie serwerem, pozwoliły uwolnić użytkownika od konieczności instalowania dedykowanego oprogramowania lub stosowania specjalnego serwera podłączonego do urządzenia. Użytkownikowi nie potrzebne są specjalne uprawnienia, aby obsługiwać USPIW. Pod tym względem stworzony system wyprzedza oferowane komercyjnie rozwiązania i może być nazwany innowacyjnym

Aby USPIW mógł zostać zintegrowany w większym systemie pomiarowym lub być obsługiwany przez aplikacje pomiarowe (np. w celu zaprogramowania przebiegu pomiaru), zaimplementowano Interfejs do Zdalnych Pomiarów. Istnieje wiele standardów pozwalających na realizację tego zadania. Bardzo powszechnym i często stosowanym jest Standard Commands For Programmable Instruments (SCPI) [57]. Standard ten określa składnię i strukturę poleceń do kontroli programowalnych instrumentów pomiarowych. W USPIW zaimplementowany został jako serwer socket'owy. Składa się on z interfejsu użytkownika, analizatora składni, dekodera poleceń, interfejsu ze sterownikiem oraz systemu zapisywania informacji o pracy serwera (logowanie). Wdrożony serwer realizuje prosty słownik poleceń dla oscyloskopu. Budowa Serwera SCPI pozwala na jego łatwe rozszerzenie o nowe funkcje. Interfejs przetestowany został przy pomocy aplikacji *Matlab*. Odpowiednie skrypty użyte w tym celu i pozwalające na połączenie się z USPIW oraz przeprowadzenie pomiarów dostępne są na stronie USPIW.

Dzięki innowacyjnej budowie i architekturze systemu, do jego obsługi niepotrzebny jest dedykowany serwer (np. w postaci komputera PC), ani specjalne oprogramowanie klienckie. Urządzenie może zostać podłączone bezpośrednio do intranetu, Internetu lub komputera osobistego. Zwykła przeglądarka internetowa umożliwia bezpośredni dostęp do wbudowanego interfejsu WWW, który pozwala na zarządzanie urządzeniem i przeprowadzanie pomiarów. Stworzony system sterujący USPIW daje możliwość łatwego rozszerzenia funkcjonalności urządzenia (np. o funkcjonalność częstotściomierza). Architektura GUI (Aplet Java'owy) pozwala na łatwe dodawanie nowych paneli kontrolnych przy wykorzystaniu uniwersalnych metod komunikacji ze sprzętem. Modułarna budowa bazy sprzętowej umożliwia zmianę mierzonych wartości. Można zamontować, np. stację meteorologiczną, zamiast modułu z przetwornikami analogowo-cyfrowymi, a następnie wykorzystać

istniejące rozwiązania do stworzenia odpowiedniego interfejsu. Co więcej, architektura i rozwiązania zastosowane w USPIW są niezależne od platformy sprzętowej. Dzięki temu mogą stanowić podstawę do stworzenia interfejsu sterującego dla dowolnego urządzenia (jeśli mikroprocesor pozwala na uruchomienie Linux'a), które ma być zarządzane zdalnie za pomocą połączenia Ethernet.

System spełnił wszystkie wymagania początkowe, a nawet przewyższył je pod względem uniwersalności. Pomyślnie przeszedł on testy w warunkach laboratoryjnych, a następnie został wykorzystany do pomiarów w Europejskiej Organizacji Badań Jądrowych (CERN). Pomiary przeprowadzone zostały w akceleratorze PS (Proton Synchrotron). Obejmowały analizę kształtu i widma sygnału elektrycznego z czujników pomiarowych detekujących przyspieszane protony.

Table of Contents

Acknowledgements	2
Abstract	3
Streszczenie	4
Table of Contents	8
1. Introduction	11
1.1 Remote control of measurement instruments	12
1.2 Web User Interface to control hardware	12
1.3 Examples of commercially available solutions	13
1.3.1 Tektronix.....	13
1.3.2 Agilent.....	14
1.3.3 BitScope	15
1.4 Hardware solutions for measurement systems	15
1.5 Universal Measurement System with Web Interface (UMSWI)	16
1.5.1 UMSWI hardware architecture and dataflow	17
1.5.2 Embedded Operating System	18
1.6 The Thesis Project Genesis and Objective	18
1.7 Requirements	18
2. Architecture	20
2.1 Embedded Operating System – Linux	21
2.1.1 Embedded Linux System for UMSWI	21
2.2 FPGA logic	22
2.3 Hardware-software communication layer	24
2.3.1 FPGA configuration.....	24
2.3.2 Communication between ARM and FPGA	24
2.4 Web User Interface	25
2.4.1 Oscilloscope and Spectrum Analyzer	25
2.4.1.1 Java Applet architecture.....	27
2.4.2 UMSWI Management Interface.....	28
2.5 Remote Measurement Interface	29
2.5.1 A Note on SCPI Compliance.....	31
2.6 Summary	31
3. Design and Implementation	33
3.1 Development environment	33
3.2 Embedded Linux Operating System	35
3.2.1 Components	35
3.2.2 Configuration	36
3.2.3 System boot and startup	37

3.2.4	UMSWI utilities organization	40
3.3	Implementation of the FPGA logic in VHDL.....	40
3.3.1	Communication logic.....	41
3.3.2	Acquisition Management Logic	45
3.3.3	Trigger detection	48
3.4	Linux Device Driver	49
3.4.1	Abstract layer.....	50
3.4.1.1	Debugging	52
3.4.1.2	/proc filesystem.....	53
3.4.1.3	ioctl	58
3.4.2	Physical layer.....	62
3.5	Binding Web Interface to Device Driver with CGI.....	64
3.6	Web Interface	65
3.6.1	Oscilloscope and Spectrum Analyzer GUI	66
3.6.1.1	Model.....	67
3.6.1.2	View.....	71
3.6.1.3	Controller	73
3.6.2	UMSWI management and configuration.....	74
3.7	Measurement Interface.....	75
3.7.1	User interface	76
3.7.2	Pre-parser.....	76
3.7.3	Parser	76
3.7.4	Commands decoder.....	77
3.7.5	Command logic.....	78
3.7.6	Hardware interface.....	79
3.7.7	Logfile interface	79
3.7.8	Extendibility.....	80
4.	Testing	82
4.1	Development test.....	82
4.1.1	Embedded Linux Operating System.....	82
4.1.2	Linux Device Driver.....	82
4.1.3	FPGA debugging	83
4.1.4	Applet tests.....	84
4.1.5	SCPI server tests.....	84
4.2	Final tests.....	85
4.2.1	Test set-up.....	85
4.2.2	Vertical axis measurements	86
4.2.3	Horizontal axis measurements.....	89
4.2.4	Frequency domain	90
4.2.5	Boundary conditions tests	92
4.2.5.1	Hardware-wise.....	92
4.2.5.2	Software-wise	93
4.2.6	UMSWI parameters	93
5.	System Applications.....	94
5.1	European Organization for Nuclear Research (CERN).....	94

5.2	Potential applications.....	96
6.	Conclusions.....	98
	Appendix A – Additional information.....	99
1.	UMSWI hardware analysis.....	99
1.1	Data acquisition hardware architecture	99
2.	Review of available technologies	100
2.1	Embedded Operating Systems.....	100
2.2	Remote Measurement Interfaces.....	100
2.2.1	Physical layer.....	101
2.2.2	Abstract layer.....	101
2.3	Web technologies to control hardware	102
2.4	Web Graphic User Interfaces	104
2.5	Web servers	104
3.	Descriptions of chosen solutions.....	104
3.1	General architecture of embedded Linux.....	104
3.2	Model-View-Controller (MVC) design pattern	106
3.3	Observer-Observable paradigm	107
3.4	Standard Commands for Programmable Instruments (SCPI).....	107
4.	Parameters of digital oscilloscope	109
	Appendix B – FPGA – ARM interface.....	111
	Appendix C – Example Manual.....	114
	Appendix D – Developer’s web page	118
	Appendix D – Additional Materials on the Accompanying CD	123
	Appendix E – List of Figures	124
	Appendix F – List of Tables	127

1. Introduction

The significance of remote control is increasing in the entire spectrum of applications, measurement is not an exception. Remotely controlled measurement instruments are both popular and needed. When acquisition is made in a dangerous place, remote control is necessary. However, the convenience of performing measurement from the office or any location in the world is becoming increasingly important and appealing nowadays. That is why most of the vendors of measurement equipment offer their instruments with remote control. It is a standard for good and expensive measurement devices, i.e. oscilloscopes, to offer control via USB, Ethernet, GPIB, etc. Such devices can be remotely controlled using measurement applications, special software provided by the vendors or web interface. Remote control interfaces provide functionality at least equal to the functionality of local interfaces.

In the measurement devices (in principle, any device) which are used only-remotely, local interface (i.e. buttons, screen, knobs) can introduce unnecessary overheads in size and costs. On the other hand, measurement devices which are used “on the spot”, can be controlled through remote interface as well. What is more, using remote interface locally can be preferred since it provides more functionality and enables to control many measurement device using single PC/laptop.

It may lead to a conclusion that nowadays, remote interface can become a substitute or competition for local interface. It seems that remote control has many advantages over traditional control. It allows management of many instruments from one station (PC, laptop), i.e. using single application (*LabView*, *Matlab*). It also enables to easily export data for further analysis. The costs of production and development can be significantly reduced by eliminating local interface. It can also remove minimal size constraints resulting in significant size reduction, since there is no need to place screen, buttons, etc on the device. Importantly, such devices (without local interface) do not lack any of the functionality of standard instruments. In contrary, they are much more functional than devices with only local interface.

Elimination of local interface can be also advantageous for equipment vendors. User interface of only-remotely controlled devices can be easily changed by upgrading the firmware. This is a great asset. One of the factors which determines application of a device is its user interface. The possibility to easily change interface enables device to be universal within hardware limitations.

1.1 Remote control of measurement instruments

There are many ways a measurement device can be controlled remotely. In principle, the solutions are divided according to the medium of communication and the software interface. A detailed description is provided in **Appendix A: 2.2**. Among 7 most commonly implemented mediums of communication in measurement devices (*GPIB, R-232, VXI, LXI, PXI, USB and Ethernet*), USB and Ethernet connections are becoming increasingly important. The software interfaces are divided into two categories.

First category (i.e. *VISA, SICL, VXI-11*) enables to program measurement devices and control them from measurement applications (i.e. *LabView, Matlab*). It is available via most of the mediums of communication and is implemented in most of the measurement devices.

Second category provides control with Graphic User Interface (GUI) which is meant to resemble local interface. It is a new trend among measurement instruments vendors to provide such interface. Only USB or/and Ethernet links are used in this category. The GUIs are either implemented as stand-alone applications which connect with the device over USB/Ethernet or Web User Interfaces which use browsers and Ethernet connection to control measurement instruments.

Web-based remote control via Ethernet seems especially attractive because it does not require installation of any special software. Ethernet card and web browser are enough to operate the device. This requirement is met by the majority of standard PCs, laptops and some models of mobile phones. It also enables the measurement instrument, without additional efforts (i.e. special server), to be connected to the Internet and controlled from any place in the World (unlike USB based control).

1.2 Web User Interface to control hardware

Web-based remote control of measurement instruments via Ethernet is an example of Web User Interface which enables to control hardware. This form of hardware control is increasingly popular not only among measurement devices.

User Interface (UI), in computing, is defined as a set of means which allow interaction (mutual exchange of information) between the user and the system (i.e. application). If the mean of interaction is a web page which is transmitted via the Internet (Ethernet connection) from the system (web server) to the user (web client) who views it using web browser, the UI is referred to as *Web User Interface (WUI)*.

The rapid increase of Internet's popularity resulted in widespread usage of Web User Interfaces in new range of applications. The greatest advantage of Web User Interface is the fact that its only hardware (Ethernet card) and software (web browser) requirements are met by overwhelming majority of modern PCs and laptops. One of the applications of WUI is online control of hardware. Web User Interface to control hardware is a web page which directly reflects state of the hardware and enables the user (client) to alter this state. One of the examples of WUI which controls hardware is the administrator's control web page of routers such as LinkSys [1] or Livebox [2]. Nowadays, web-based control of hardware finds increasing number of applications in the following fields:

- Intelligent buildings - WUI enables to access intelligent building's control panel and manage it from any location in the World (i.e. office),

- Measurement device control – WUI, which resembles the instrument, enables to perform measurement remotely, or view measurement results by many research teams spread around the World
- Internet remote laboratories – WUI enables to perform experiments and measurements remotely using laboratory sets,

Web User Interfaces can range from very simple HTML pages which are controlled by clicking appropriate hyperlinks or inputting values into forms, to sophisticated web applications which provide Graphics Interface (i.e. resembles actual device being controlled). A detailed description of various technologies which enable web client to interact with hardware is enclosed in **Appendix A: 2.3**. What distinguishes such a technology is the possibility to make system calls, start/stop applications or read/write files on server side (which are the means to control hardware) as a consequence of web client's request. Since web server is the recipient of web client's requests, it needs to be able to perform such actions. Most servers, if not all, embed Common Gate Interface (CGI)[3]. It is an old mechanism which enables the server to execute scripts (shell, Perl, Python, etc) or even applications. Such scripts or applications can, in turn, access and control hardware. A newer technology which enables hardware control, by providing file access and special functions to run shell commands, is PHP [4]. PHP is a server-side scripting language which produces dynamic web pages. It requires a PHP parser installed along with the web server. There is a number of other technologies which enable to access hardware. The more sophisticated technology (Java Servlets, ASP.NET) the more requirements needs to be satisfied by the server. Very often the web server is embedded in the device which is being controlled. It means that the server is run on an embedded system with limited resources which does not allow to use sophisticated technologies. An example of such device is a measurement instruments which provides web-based control.

1.3 Examples of commercially available solutions

Among few commercially available solutions which enable web-based control of measurement instruments, most is based on web servers embedded into the devices. A background research of commercially available Web User Interfaces for measurement instruments revealed that the number of such solutions is not great.

1.3.1 Tektronix

Two implementations of remote control over Ethernet are offered by the Tektronix measurement instruments. First solution uses special application, available for Windows operating system, which needs to be installed on a PC. It is described in [5] application note for TG700 Tektronix device. The Tektronix's application connects to measurement instrument in order to send commands and retrieve data.

e*Scope is the second remote control GUI provided by Tektronix. It is a web-based interface which provides access to all front panel oscilloscope's controls and presents a faithful reproduction of the oscilloscope screen. Tektronix website [6] states that:

*“With the new e*Scope web based remote control feature, a common network browser, and Internet connection, the engineer in Beijing can see exactly what the designer in Berlin is seeing on the TDS3000B screen-at the same time.”*

e*Scope, described in [7], operates in two modes: basic and advanced. The basic mode is available directly after connecting a PC with LAN cable to Tektronix instrument. The e*Scope home page, which is housed in the device, enables the user to control oscilloscope by typing in commands. To run the advance mode, a special website provided by Tektronix [8] needs to be accessed or a “e*Scope Software” needs to be downloaded (to avoid connecting to Tektronix web page). The advanced mode enables user to control oscilloscope through graphic user interface (Figure 2).

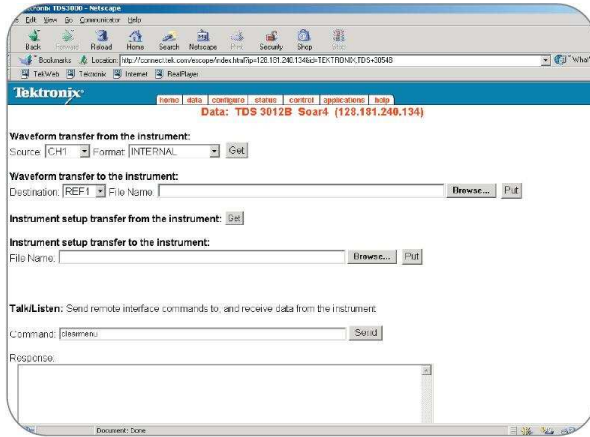


Figure 1 e*Scope basic mode

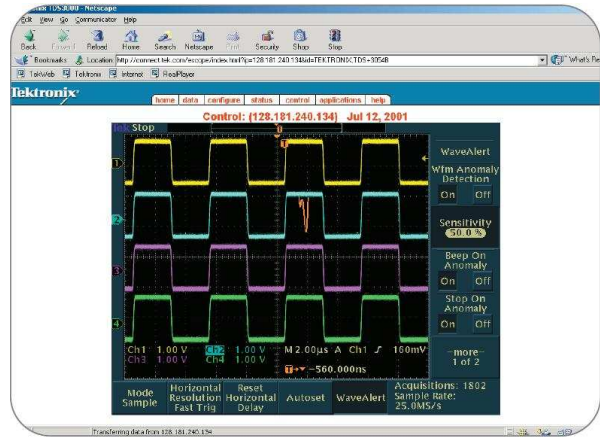


Figure 2 e*Scope advanced mode

An investigation of “e*Scope Software”, which is available for download from Tektronix web page, enabled to learn the technology and design of e*Scope solution. It uses JavaScript to send commands to the instrument and retrieve screen views. A screen view is generated in the Tektronix device and send to the browser as an image in .png format. “e*Scope Software” turned out to be a simple JavaScript.

1.3.2 Agilent

BenchLink Web Remote Control [9] is a software that provides remote control for Agilent’s spectrum analyzers. The software is installed on a local server computer which is connected to the instrument via GPIB or LAN interface (Figure 3). Multiple users can access the analyzer simultaneously from the intranet or Internet. The server requirements include: Windows XP, 100MB free disk space, PCI expansion for PCI-GPIB card or PCMCIA in case of using laptop or configuration to run a LAN-GPIB gateway. Only Web browser is required from the client to operate the analyzer. The software can be tested on Agilent web site [10] which provides a limited-features simulation of spectrum analyzer Figure 4).

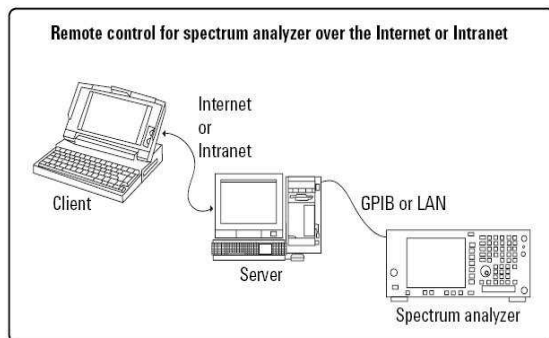


Figure 3 Remote control of Agilent Analyzer

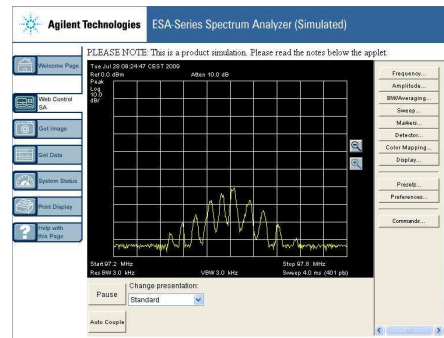


Figure 4 BenchLink applet

1.3.3 BitScope

BitScope is the only commercially available product found by the author*, which offers only remote interface. BitScopes can be controlled via Ethernet or/and USB (depending on the model). BitScopes are Mixed Signal Oscilloscopes, which means that they capture and display one, two or four analog signals and eight logic or timing signals, simultaneously. Regardless of the connection type, BitScope products are controlled by BitScope DSO Virtual Instrument Application which needs to be installed on a PC. It integrates Digital Storage Oscilloscope, Mixed Signal Oscilloscope, Spectrum Analyzer, Logic Analyzer, Data recorder and Networking. The software is available for Windows and Linux workstations. BitScope Model 100 is presented in

Figure 5. This model is USB-controlled. It is the only BitScope model which is “*user programmable and software extendable*”[11] which is possible though BitLib Application Programming Library. The library can be used with “*several different programming languages and numeric analysis environments*”[11]. It can be used to operate BitScope from MatLab or LabView as well as for writing applications with Visual Studio or Borland Delphi.

Figure 6 provides an insight into BitScope Model 100 architecture and the manual of BitScope Model 50 [12] provides details of BitScope’s hardware. It is controlled by PIC microcontroller (PIC16F877) and uses Complex Programmable Logic Device (M4A5-TQFP44). 8bit Analog-to-Digital Converters are used enabling 100MHz bandwidth and 2mV~40mV analogue sensitivity.



Figure 5 BitScope instrument and GUI

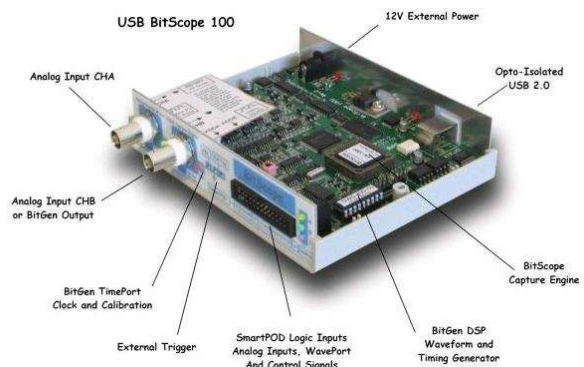


Figure 6 BitScope Model 100 architecture

1.4 Hardware solutions for measurement systems

The multitude of features, remote measurement interfaces, sophisticated local interfaces (i.e. touch-screens) or the ability to be controlled only remotely result in excessive hardware requirements towards modern measurement systems. In particular, nowadays most (if not all) measurement instruments include microprocessors which control virtually every circuit in the measurement devices. Since modern microprocessors can be very powerful (i.e. PowerPC [13], ARM [14]) and the requirements on instrument’s features are increasingly demanding, more and more measurement instruments employ embedded operating systems (in case of less sophisticated and cheaper devices) or even normal operating systems (very sophisticated and expensive, i.e. Agilent Infiniium Oscilloscopes [15] work on Windows XP Pro). Such solution allows for great flexibility.

* The author cannot guarantee that there is no other similar product on the market

On the other hand, the example of BitScope (1.3.3) shows a tendency of using programmable logics in measurement devices. Complex Programmable Logic Devices (CPLDs) as well as Field Programmable Gate Arrays (FPGAs) are used for signal processing (i.e. FFT) and other operations where massive parallelism is needed. Such tasks are performed much faster in FPGAs or CPLDs than in microprocessors. It reflects a general recent trend in electronic device development to combine the logic blocks and interconnections of traditional FPGAs with embedded microprocessors and related peripherals. FPGA is especially popular in custom-made or low-volume systems, since it is re-programmable providing easy bugs fixing and short time to market. Among providers of CPLDs and FPGAs are Altera [16], Xilinx [17], ATMEL [18] or Lattice Semiconductors [19].

1.5 Universal Measurement System with Web Interface (UMSWI)

Responding to a demand from European Organization for Nuclear Research (CERN), Creotech Ltd. [20] high-tech company produced a prototype of modular embedded measurement device. In the configuration provided, it is an ARM based microcomputer with data acquisition daughterboard. It consists of 3 modules: main board, ARM computer and recorder. The most important parameters and features of the hardware are presented in Table 1.




Module name	Module application	Module's components	Size [mm]	Photo
Main board	Hosts power supply, peripherals and other modules	Switched-mode Power Supply Graphic controller Sound controller I2C interface <i>Peripherals:</i> USB, RS232, Ethernet, output for built-in LCD-TFT and for VGA monitor	100x80	
ARMputer	Single Board Computer	Processor: ARM9 (AT91RM9200) [45] 128MB SDRAM Ethernet interface 10/100 Mbit FLASH 8MB SD/MMC reader, Interfaces: 2 x Serial ports, 2x USB hub and device	60x70	
Recorder	Acquisition	ALTERA Cyclone I FPGA [21] 2 x fast, 105MS/s. 10 bit ADCs [22] SSRAM – 128K x 32 b [23]	100x80	

Table 1 Hardware components of UMSWI

The device is meant to be a measurement instrument for High Energy Physics i.e. used in accelerators for diagnostics. However, the number of possible application is far greater, alternatively UMSWI can be used for data acquisition in any dangerous or hard-to-reach place, as a remote monitoring system of industrial parameters, reconfigurable measurement system or an element of distributed measurement system .

In such places as accelerators, measurements are done remotely due to the possible radiation danger. Once settled in the measurement location, the instrument is operated from a safe place. Therefore, development of a control system which enables remote management of the device and remote data acquisition was necessary. The device is equipped with Ethernet peripheral to enable remote control via Ethernet connection. Since Web User Interface (1.1) seems to be the new trend in measurement instruments' remote control, which demands the least requirements on the client and is a very flexible solution, it was decided that such interface should be developed. Remote control using Web User Interface is especially suitable for operation at CERN, since it does not require dedicated client software. Due to the fact that four different operating system platforms are used at CERN (Linux, Windows, Mac and UNIX), it would be very time-consuming and expensive to create client's software for each of them. Web User Interface is client's platform-independent.

In order to enable UMSWI to be a part of a larger system (i.e. Tango [24] or EPICS [25]) or to take part in experiments where measurement instruments are controlled with applications such as *LabView* or *MatLab*, more "traditional" control is needed, therefore Remote Measurement Interface (1.1, first category) was required to be implemented.

The author was given an opportunity to choose and adapt embedded operating system, develop a prototype control software, FPGA logic and interfaces for the provided hardware.

1.5.1 UMSWI hardware architecture and dataflow

Design and development of control system for hardware requires thorough understanding of architecture, data flow and limitations of the provided hardware. **Figure 7** presents general overview of UMSWI's architecture and dataflow.

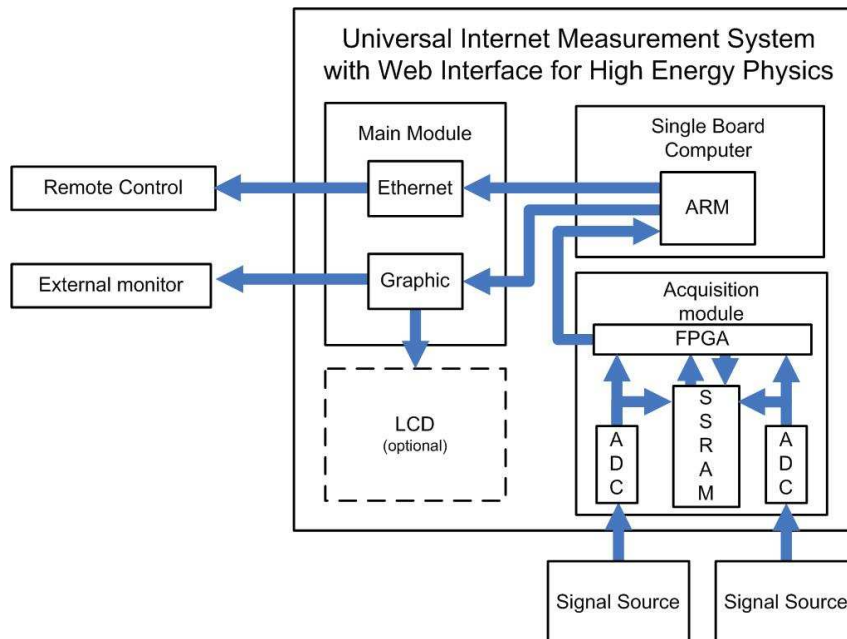


Figure 7 UMSWI architecture and dataflow

The acquired signal is converted by 10 bits Analog-to-digital Converters (ADCs) and saved into Synchronous Static Random Access Memory (SSRAM). The acquisition is controlled by the Field Programmable Gate Array (FPGA). Alternatively, instead of saving data in SSRAM, it can be directly read by FPGA, computed and later saved in SSRAM. It is also possible to save data in SSRAM and read in FPGA simultaneously. Once the acquisition has finished, data can be read by the processor. Readout process is managed by FPGA and controlled by microprocessor. Data processing can be performed in FPGA logic as well as in application running on microprocessor. From the processor data is transported to the user by the Ethernet or can be displayed locally on LCD/VGA monitor.

Hardware Architecture of acquisition module is described in details in **Appendix A: 2.1**. It was particularly important to familiarize with acquisition module, since its layout has the greatest influence on architecture and design of UMSWI control system.

1.5.2 Embedded Operating System

The UMSWI was intentionally provided with a powerful ARM9 microprocessor to enable usage of embedded operation system. In fact, the microprocessor (AT91RM92000) is very popular among embedded systems. It is, of course, possible to develop applications directly for this processor. However, much better and more popular solution is running embedded operating system. It makes the system flexible and allows re-use or adaptation of already existing solutions. A review of embedded operating systems and general description of Embedded Linux architecture (Linux was chosen to be the operating system on UMSWI) is provided in **Appendix A: 2.1** and **3.1**

1.6 The Thesis Project Genesis and Objective

Following a demand by High Energy Physics for a small remotely controlled diagnostic measurement device to be used in accelerator tunnels, UMSWI hardware was created. The hardware was designed following modern trends in measurement instrument development and having in mind broader applications (than accelerator diagnostics). Such universal and trendy hardware needed equally featured control system[†] which could not be provided by *Creotech Ltd.* The UMSWI control system's development required to be preceded by research to establish current trends, tendencies and technological possibilities in the field - a task in line with interests, knowledge and experience of the author who decided to take this challenge, thus this Master Thesis project was started.

The goal of this Master Thesis is the utilization of commercially available hardware (*Creotech Ltd.*) to create an autonomous, universal measurement system with remote Web-based control.

1.7 Requirements

An employee of European Organization for Nuclear Research (CERN), who has experience in the area of exploitation and usage of commercial measurement instruments, presented demand for a measurement device with the following interfaces:

- Web User Interface (WUI) –providing web-based Graphic User Interface to enable the user to remotely control UMSWI, perform data acquisition and display acquired data in graphical form.

[†] A collective of systems, applications, interfaces, etc. that were designed and developed by the author is called in this thesis **control system of UMSWI**.

- Remote Measurement Interface (RMI) – implementation of one of the standard measurement protocols to allow remote control from measurement applications level (i.e. *Matlab*, *LabView*)

The device was required to enable functionality of simple digital oscilloscope and spectrum analyzer with further possibility of other measurement system implementations.

Installation of a reasonably powerful microprocessor in the provided device was intended for usage of embedded operating system, therefore an operating system should be chosen and developed.

The main limitations to this project were imposed by the provided hardware. The following measurement system features were determined:

- Signal source – two digital ADCs determines number and type of signal source
- Sampling speed – determined by the speed of ADCs:100MHz
- Sample maximal length – determined by the SSRAM memory size (128k of samples)
- Communication: Ethernet, USB, RS-232

2. Architecture

The architecture of UMSWI's control system was created dividing the system into the following components

- Operating system
- FPGA logic
- Web User Interface (WUI)
- Remote Measurement Interface (RMI)
- Connotation between Remote Interfaces and FPGA logic

The division was determined by hardware architecture (1.5.1), requirements (1.7) and technologies (**Appendix A: 2**) needed to develop each part of the system. General architecture of entire system was created before design and implementation of component (**Figure 8**).

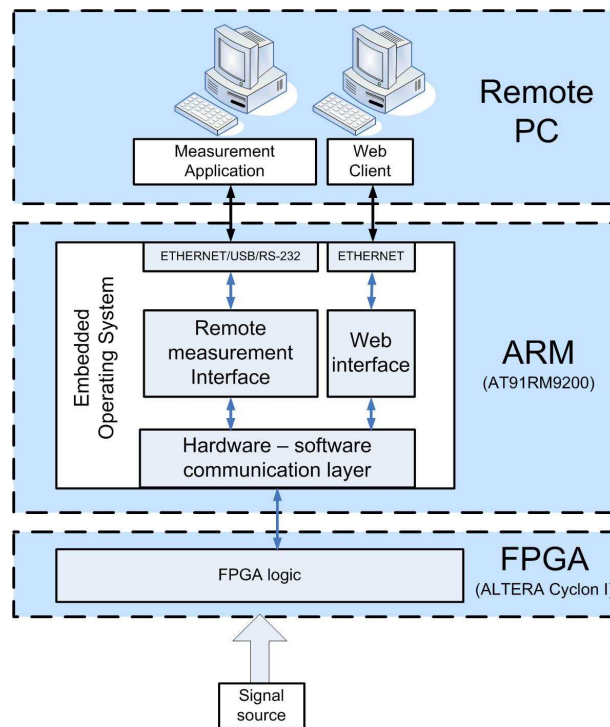


Figure 8 General UMSWI architecture

According to the requirements, the UMSWI control system is based on an embedded operating system. Such solution enables high level of flexibility which was decided to be utilized to the benefit of the system's flexibility, robustness, simplicity of further extensions and modifications. The architecture and design of the entire system and each component were prepared having in mind reusability, extendibility and universality.

Each of the required interfaces (WUI & RMI) is placed in the embedded operating system environment. Both interfaces need to communicate with FPGA logic, therefore a common hardware-software communication layer can be provided. The layer is designed according to operating system rules and adjusted to underlying hardware specification. It provides communication with FGPA logic on various levels of abstraction to enable creation of additional interfaces and control of different FPGA logic or even different hardware

Remote Measurement Interface (RMI) needs to implement standard interface which can be connected to (on the physical level) and understood (on the abstract level) by third-party measurement applications running on the remote PC. Therefore the client application does not influence the system’s architecture, unlike in the Remote User Interface (RUI). In RUI, depending on the technology choice, the application running on the remote PC can be either an integral part of the UMSWI control system, or can have substantial influence on the system’s architecture.

The FPGA logic provides logic to control acquisition process and communication interface to exchange data between ARM and FPGA. The communication interface is universal to enable control of custom-made acquisition control logic (i.e. extended to implement computation algorithms like FFT).

2.1 Embedded Operating System – Linux

The choice of embedded operating system was preceded by background research on available solutions suited for AT91RM9200 architecture. The review is summarized in **Appendix A: 2.1**.

All the proprietary solutions were discarded since they increase the costs of UMSWI and bring licensing issues in further extensions or modification of the system. Furthermore, the number of users of proprietary embedded operation systems is smaller and the exchange of information between them not as public as in the case of open source embedded operating systems. Therefore, the choice of non-proprietary embedded Linux, the most popular among embedded open source operating system with the strongest developer’s support . There is a vast number of books, articles and forums describing it’s usage and development on ARM mikroprocessors: [26, 27, 28, 29]. The author of [27] in the chapter “Reasons for Choosing Linux” as well as the author of [28] in the chapter “Why Embedded Linux” devote few pages pointing out advantages of using Linux. Among others are: “availability of code”, “hardware support”, “available tools”, “Community support” and many others. Running Linux enables using a great number of open source programs and support of a strong and numerous community of Linux and embedded Linux developers. It was decided not to use any of the open source Embedded Linux distributions to ensure systems wide portability. If the system was developed for particular embedded Linux distribution, using the UMSWI’s control system on the architecture not supported by the chosen distribution might pose a problem. Therefore, “vanilla” kernel was used making UMSWI control system potentially usable on any distribution. “Vanilla” kernel is the Linux kernel version maintained by Linux Torvalds (the creator of Linux) himself. It servers as a reference point for all the distributions and ports of Linux. Many Linux operating system vendors modify the kernels of their product, i.e. to add support for drivers and features not officially released as stable. All the embedded Linux distributions are include versions of “vanilla” kernel.

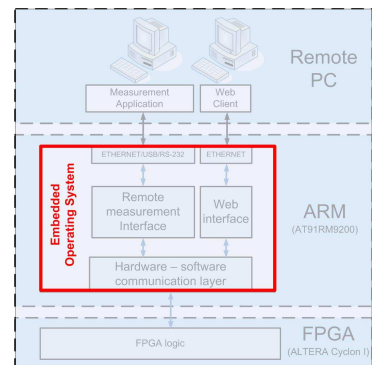


Figure 9 UMSWI architecture - Linux

2.1.1 Embedded Linux System for UMSWI

The architecture of Embedded Linux System for Universal Measurement System with Web Interface follows general rules of embedded Linux system architecture which are described in details in **Appendix A: 3.1**. The architecture is determined by three factors :

- Hardware restrictions (storage size, RAM, peripherals),

- System requirements and applications,
- Potential portability to other architectures.

One of the main hardware restrictions in Embedded Linux Systems is storage size. Frequently, the biggest challenge for embedded developers is to fit the system into limited memory space. 8 MB of flash memory provided by UMSWI is more than sufficient to hold embedded Linux image with compiled kernel and root file system providing basic utilities. It turned out that the flash memory is also sufficient to hold all the utilities of UMSWI. However, it was decided to store UMSWI utilities on the MMC/SD memory card. Actually, UMSWI utilities are stored in both locations (root file system on flash memory and MMC/SD). It allows the system to be much more flexible. If the system is to be ported to an architecture with limited flash memory, MMC/SD can be used. If MMC/SD slot is not provided, UMSWI utilities are read from flash. On the startup, the system tries to find UMSWI utilities on MMC/SD card in the first place. If it fails, utilities stored in flash are used. It makes the system robust and enables easy upgrades. The user upgrades or modifies UMSWI utilities stored on MMC/SD. If the upgrade fails, or the user's modifications are erroneous, the system is still useful provided the MMC/SD card is not inserted.

The Low-level interface is appropriately ported to match the AT91RM9200 architecture and ARMputer peripherals.

2.2 FPGA logic

FPGA logic architecture is composed of two parts: Communication Logic (CL) and Acquisition Management Logic (AML). Communication Logic is used to exchange information between Acquisition Management Logic and Remote Interfaces (in principle: operating system user space):

- acquisition parameters
- state of acquisition
- measurement data

Acquisition Management Logic is meant to manage data acquisition, in particular:

- collecting data from ADCs
- storing data in SSRAM (during data acquisition)
- reading data from SSRAM (during data readout)
- data processing
- trigger management and detection

Such architecture makes it easier to further extend the system (i.e. with different data processing algorithm) or adapt it to different hardware. Thanks to the separation, in case of system extension or adaptation, the main modifications are performed in the Acquisition Management Logic, while the communications remains unchanged or requires very small modifications.

The Acquisition Management Logic is controlled, through Communication Logic, by the user. The control includes starting/stopping the acquisition and determining the acquisition characteristics (parameters). The parameters were determined studying operation and control of an oscilloscope:

- Sampling time (t_s) – the minimum sampling time ($t_{s_{min}}$) is determined by ADCs' sampling frequency (100MHz), sampling time can be a multiple of $t_{s_{min}}$ only,
- Record length (l) – number of samples stored in SSRAM after trigger. Maximum value of record length (l_{max}) is limited by SSRAM size (128K 32bit-words),

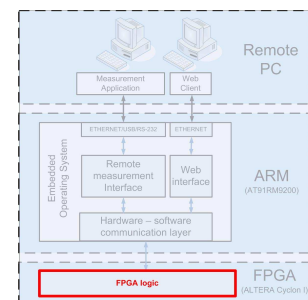


Figure 10 UMSWI architecture -

- Delay time (t_d) – the interval between trigger detection and start of record length counter,
- Trigger source (SRC)– there are three types: external signal, signal from ARM9 (user), signal level.
- Trigger level,
- Trigger edge,

It was decided to store each of the parameters in FPGA as a register mapped into separate address of ARM9 microprocessor memory space. Therefore, the task of Communication Logic is to:

- recognize the operation (read/write),
- decode the address presented on the address bus ,
- read data from ARM data bus and write it into appropriate register in case of ARM write operation,
- read data from appropriate register and present it on the data bus, in case of ARM read operation,

Acquisition Management Logic updates the registers with acquisition state or measured data and controls the content of other registers. Such architecture allowed to solve the problem of two clock domains described in details in **Appendix A: 1.1**.

Another hardware obstacle, which influenced FPGA logic architecture (described in **Appendix A: 1.1**), is the fact that the memory address space mapped to FPGA by ARM is smaller than SSRAM size. Therefore, the entire memory space of SSRAM is mapped to one FPGA register of one word size (16bits) called readout register. This register is, in turn, mapped to certain address in ARM address space. Each time the processor reads readout register, new data from SSRAM is provided by Acquisition Control Logic. Because data width of bus between ARM and FPGA (16bits) is twice smaller than SSRAM word (32bits), it takes two readout operations for ARM read entire SSRAM word. The FPGA logic architecture is presented in **Figure 11**.

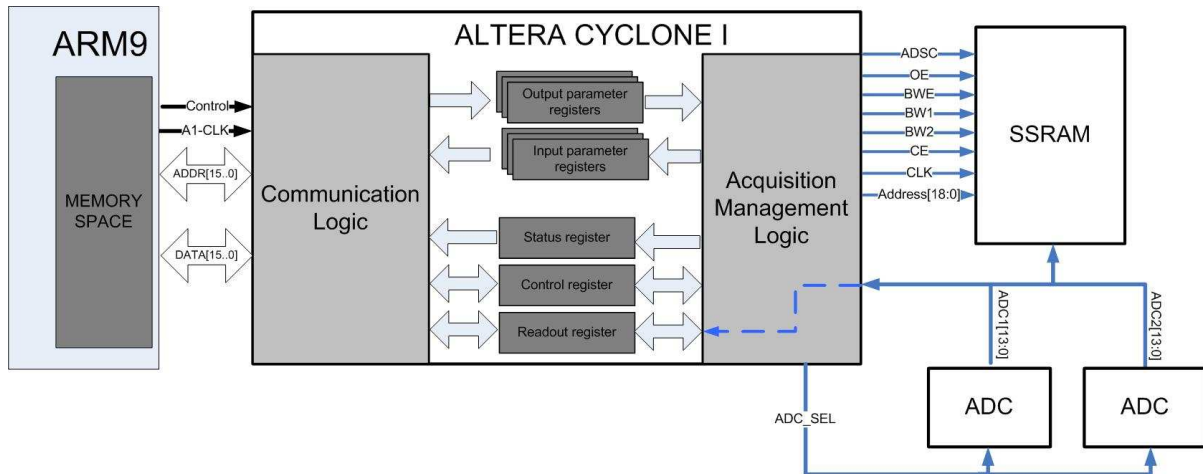


Figure 11 Acquisition and readout control and dataflow

2.3 Hardware-software communication layer

The hardware-software communication layer is understood as an interface between Linux user space and FPGA. Its main purpose is to provide communication between Interfaces and FPGA. However, the configuration of FPGA (sending a binary stream to FPGA) is also included into the layer's tasks.

Since Linux is used as operating system, there are two possible approaches to interface hardware connected to microprocessor and mapped into memory address space:

- Mapping appropriate address in User Space – slower, easier to implement,
- Writing Linux Device Driver to access appropriate address in Kernel Space – faster and much more efficient for data transfers, enables interrupt implementation, not a trivial task.

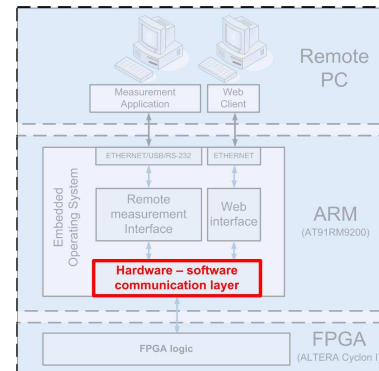


Figure 12 UMSWI architecture - driver

The communication and configuration were decided to be approached separately, choices of the appropriate technique for each of them are discussed below in separate subchapters. The result is summed up in the **Figure 13**.

2.3.1 FPGA configuration

In normal operation of UMSWI, FPGA configuration is done during the startup of the system. Reconfiguration during system operation is only done in the development phase. Therefore, the speed of configuration process is not crucial and it has been decided to use simpler and faster to develop User Space mapping to implement FPGA configuration.

2.3.2 Communication between ARM and FPGA

The communication between ARM and FPGA is crucial for system's operation and is one of its basic components. Up to 265 Kb of measurement data needs to be transferred from FPGA. To comply with the intention to create universal and extensible system, the communication is provided on two levels of abstraction:

- Interface suited to the implemented Interfaces and Acquisition Logic
- General interface enabling to extend Acquisition Logic or use the existing Logic in a non-standard way.

Such solutions enables flexibility on both sides of the communication layer. It is possible to extend the already existing applications (alternatively, create new applications) to use the same Acquisition Logic in non-standard way or to use modified or entirely different Acquisition Logic.

The communication layer along with FPGA configuration are the only software parts of the UMSWI control system which are directly hardware dependent. While the FPGA configuration is a simple operation, communication layers is more advanced and its architecture needed to be created having in mind easy porting to other hardware configurations. Therefore, it was decided to create a Linux Device Driver [30] with architecture clearly divided into abstract and physical layer. Such separation enables the driver to be easily portable to other hardware configuration and also makes it easier to extend the driver with additional functionality.

Abstract layer

Abstract layer of the driver is responsible for communication between the driver and user space (in which all applications are run) and implements the driver’s logic. It is hardware independent.

Physical layer

Physical layer implements the communication between driver and hardware as well as hardware’s initialization.

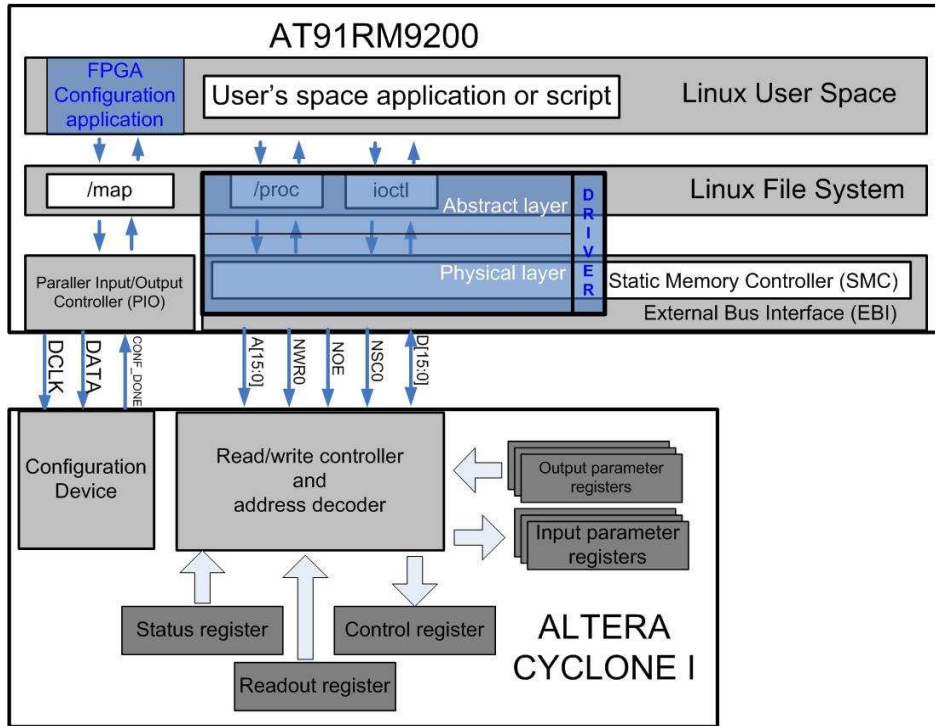


Figure 13 Communication between FPGA and ARM

2.4 Web User Interface

Web User Interface is clearly divided into UMSWI Management Interface (UMI) and Oscilloscope & Spectrum Analyzer Graphic User Interface (O&SA GUI). The former allows configuration of UMSWI’s parameters such as IP address,. The later is meant to perform measurement and present the results in graphical form. Additionally, information about system (manuals) are provided by the interface. Since Oscilloscope and Spectrum Analyzer GUI is much more demanding (in terms of development effort and system requirements), sophisticated and crucial to the system, it was decided to make the choice of technologies-to-be-used according to its requirements and adjust the implementation of UMSWI Management Interface to the chosen solutions.

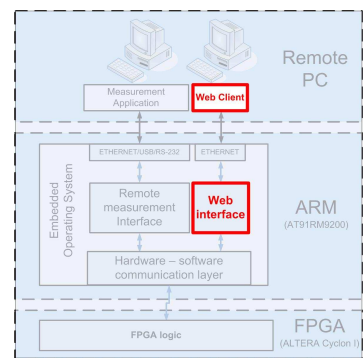


Figure 14 UMSWI architecture - WUI

2.4.1 Oscilloscope and Spectrum Analyzer

Architecture of Web User Interface depends greatly on chosen technologies. In general three components of the interface can be distinguished:

- Graphic User Interface (displayed in the client's browser)
- Web Server
- Interface between Web Server and hardware (in particular, Linux Device Driver)

For each of the components, a decision of technology-to-be-used needed to be taken considering choices of technologies for the other components. A review of possible solutions for each of the components is included in **Appendix A: 2.3, 2.4 and 2.5**

The choices were made taking into account two criteria: simplicity and limited resources. Simplicity of solutions is important for the development and further extensions to the system. Resource limitations:

- Processor speed
- RAM size

Simplicity:

- the less tools need to be cross-compiled the better – some tools, applications are not trivial to port to embedded architecture
- less sophisticated solutions are easier to test and debug

To move much of the workload (i.e. graphic generation, user interface handling) from the embedded system to the client PC (far more powerful unit), Java Applet technology was chosen for implementation of Web Graphic User Interface. Java Applet is a web application which is downloaded and executed in the client's browser. Since the interaction with the client is managed by the applet locally on the client's machine (unlike in PHP where client's interaction is handled by the server), network traffic can be reduced by communicating with the server only during hardware interfacing. Therefore, the role of the server is limited to simply passing information/data from/to the driver. This eliminates many server requirements imposed by other technologies (i.e. support for PHP).

The server's capabilities influence the choice of technology used to interface hardware and vice versa (hardware interface impose requirements on the server). If the Web Server embeds scripts interpreter (i.e. PHP) or enables Java Servlets, hardware (through Linux Device Driver) can be accessed directly by opening its file representation. However, since the server's requirements from the Web Graphic Interface (web applications) were minimized, it is reasonable to choose hardware interface with minimum Web Server's demands as well. Such choice enables to use the simplest Web Server. Therefore, Common Gate Interface (CGI) was chosen as an interface between Web Server and Linux Device Driver (which implements /proc file system – very convenient for CGI access) and consequently the simplest and smallest (9K, [29]) Web Server provided by BusyBox could be used. The choice to use CGI in UMSWI is supported by the following advantages:

- CGI is well known, well developed and it is still being used by many web pages and applications (ex. hotmail.com),
- It is implemented by most of web servers, does not involve any additional tools to be cross-compiled,
- It allows to execute "CGI scripts" written in many different scripting languages as well as compiled programs,
- Any distribution of Linux enables writing scripts for CGI interface, it means that as long as the most basic version (even very old) of Linux is ported for a platform, and the most basic HTTP server is available, CGI can be used. As a consequence, using CGI makes the whole system very flexible and platform independent.

Except for the communication with the driver, CGI scripts can be used to manage UMSWI control system by performing system calls (i.e. to configure Ethernet Interface) or starting/stopping applications (i.e. SCPI Server). Choosing BusyBox Web Server provides portability, since such server is available for majority of embedded Linux systems. The choice of technologies is summarized in **Figure 15**.

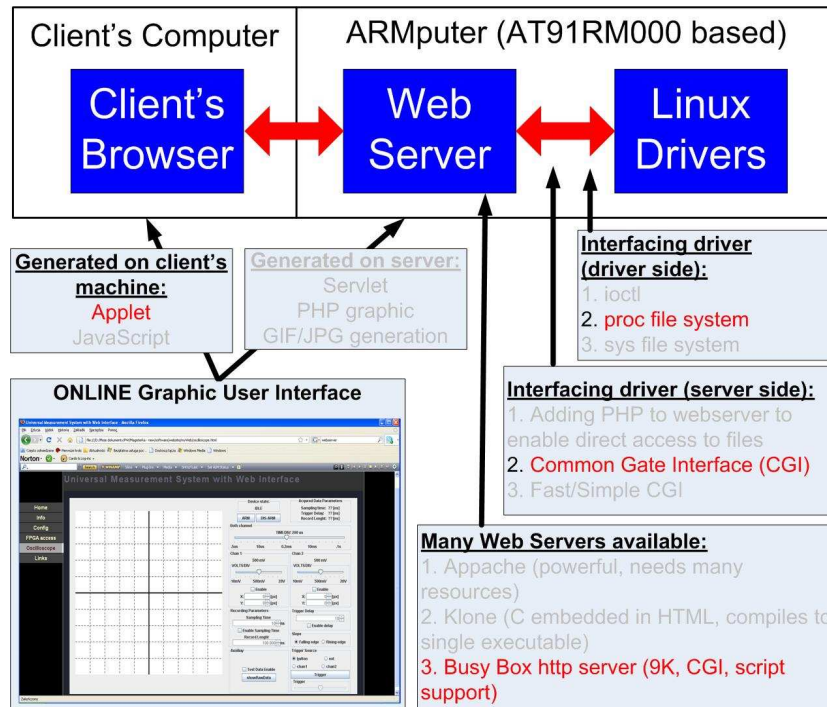


Figure 15 Choice of technologies for Web Interface of UMSWI [31]

Once the choice of technologies was done, the Web Interface architecture could be created (Figure 16). The Web Server stores UMSWI website and Java Applet binaries, and provides Common Gate Interface. Once the web site embedding Java Applet is opened, the applet is downloaded to the web browser and executed. The applet communicates with hardware (Linux Device Driver) through CGI interface. CGI is also used by the UMSWI configuration web page.

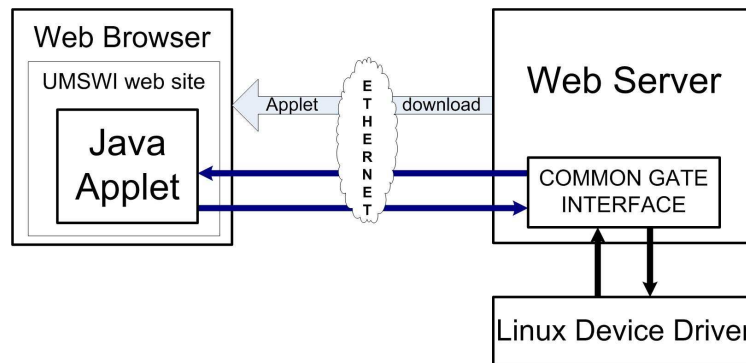


Figure 16 Oscilloscope and Spectrum Analyzer Web architecture

2.4.1.1 Java Applet architecture

One of the system patterns which helps in application design on the abstract, architectural level is Model-View-Controller, described in details in **Appendix A: 3.2**. It is a language-independent pattern which is widely used. It was chosen because it allows easier and independent modification of visual appearance or underlying business rules. Thus, it enables easy extensibility and reusability. It divides the application into three logical components: model, view and controller making it easy to customize or modify each part. An architecture of UMSWI's applet organized according to MVC paradigm is presented in **Figure 17**.

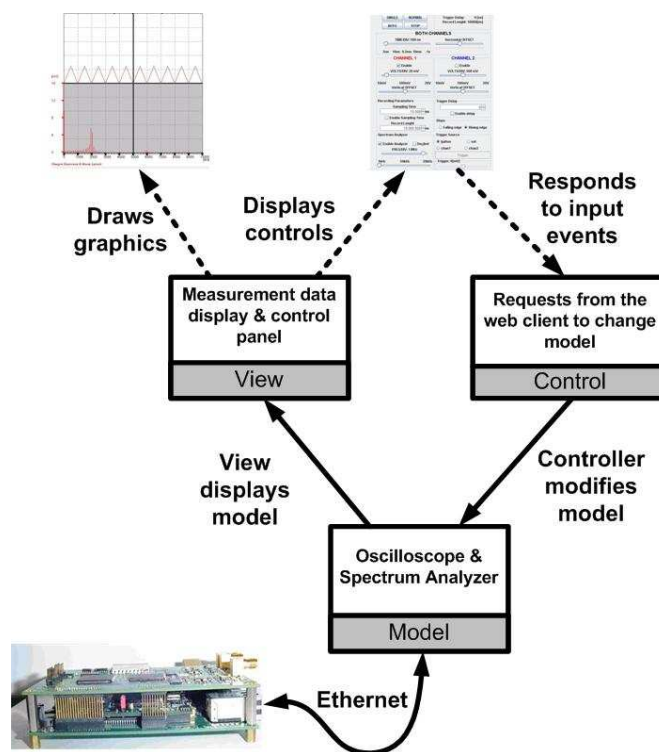


Figure 17 UMSWI's architecture according to MVC

Model represents Oscilloscope and Spectrum Analyzer, it reflects their state. Thus, *Model* communicates with the hardware and changes its settings. *View* is responsible for displaying data provided by *Model*. Control panel, which enables to change the *Model*, is also displayed by *View*. Any changes made by user on the control panel are detected by the *Control* component which updates the *Model*. The control panel enables to adjust two kinds of settings:

- Hardware settings – parameters which can be used to control acquisition logic (sampling time, trigger delay, trigger source, record length, trigger level),
- Display settings – parameters which control the way data is displayed and whether it is displayed (Volts/Div, Time/Div, Freq/Div, enable chan1/chan2),

It also enables control of the device state (start/stop acquisition) and display of the device parameters

2.4.2 UMSWI Management Interface

The UMSWI Management Interface is kept simple on demand of CERN's employee who required the device. It includes only the most necessary configuration:

- IP address (setting current and saving default)
- Mask (setting current and saving default)
- Port of SCPI Server
- SCPI Server on/off

The default IP address is saved in the memory and the system is started with such address. The possibility of setting of IP address or SCPI Server Port is important when the system is integrated into a Local Network Area (LAN) infrastructure or when many UMSWIs create distributed measurement system. It was decided to enable starting and stopping SCPI Server to save UMSWI's resources when SCPI Server is not used.

Common Gate Interface scripts allows to perform the above-mentioned configuration. It would be also possible to use Graphic User Interface developed as Java Applet to do

the configuration. However, it was concluded that Java Applet is too heavy weighted-technology for such trivial task. Therefore it was decided to prepare simple webpage and use Java Script for data verification and calling CGI scripts. The architecture of UMSWI Management Interface is very similar to Oscilloscope and Spectrum Analyzer's architecture. **Figure 18** presents architecture of entire Web User Interface.

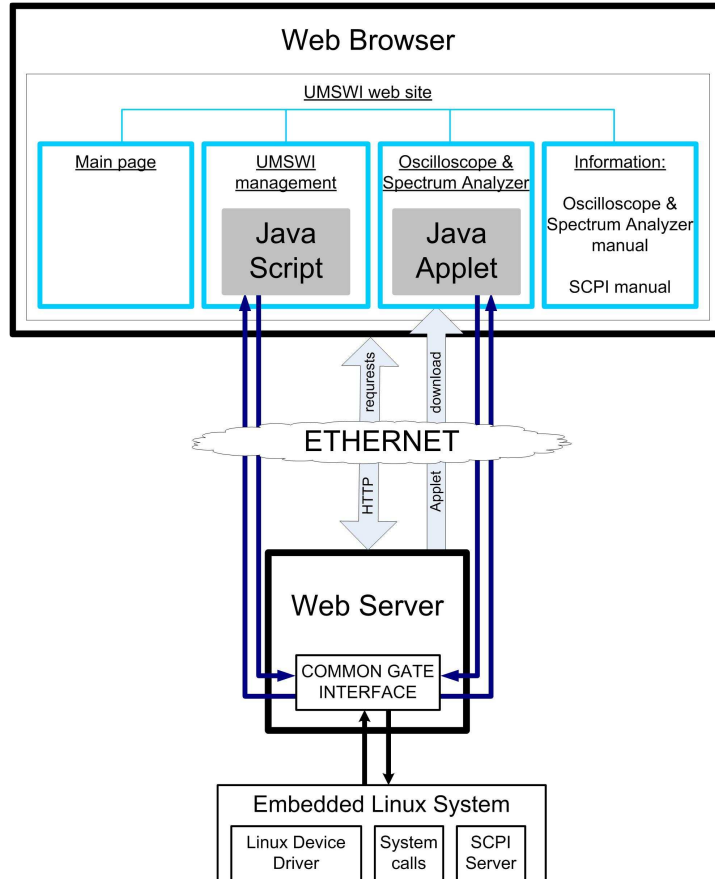


Figure 18 Web User Interface architecture

2.5 Remote Measurement Interface

Remote Measurement Interface (RMI) is described in this thesis as an interface which enables UMSWI to be controlled remotely by measurement applications (i.e. *LabView*, *Matlab*). The medium to be used, is determined by the UMSWI's hardware: Ethernet. An in-depth investigation was conducted to choose an appropriate interface for implementation. A review of possible solutions can be found in **Appendix A: 2.2** The following requirements were taken into consideration:

- Well defined and widely used,
- Modern,
- Simple,
- Physical layer: Ethernet.

It seems that most of the measurement instrument vendors (i.e. Agilent, Tektronix, HP) offer new high-tech devices with many remote measurement interfaces. However, Standard Commands for Programmable Instruments (SCPI) seems to be the most widely

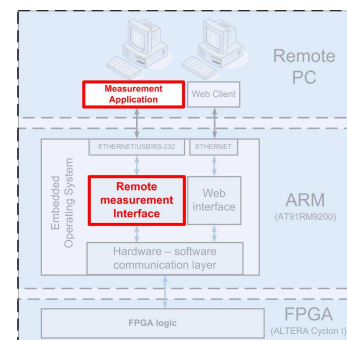


Figure 19 UMSWI architecture

implemented. The medium of data transfer has changed from GPIB or RS to Ethernet and USB, however the SCPI standard is still alive. What is more, SCPI can be used by most of the popular measurement applications (like *LabView*) and applications which can connect with measurement instruments to retrieve measurement data(i.e. *Matlab*). Therefore, it was decided to implement Standard Command for Programmable Instruments (SCPI).

The SCPI standard is shortly described in **Appendix A: 3.3**. The standard defines command's structure and syntax but does not specify underlying hardware or software solutions. **Figure 20** presents example SCPI command and its architecture.

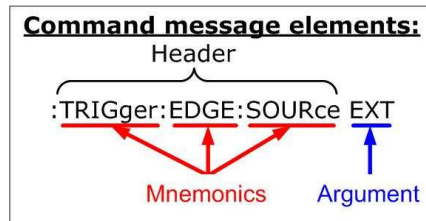


Figure 20 SCPI example command

In principal, an input to the RMI application is a string of characters consisting of a command (message), or a set of commands separated by (;)semi-colons. Each command message is composed of a sequence of mnemonics separated by (:) colon and an argument. The path determined by mnemonics unequivocally determines what action shall be performed.

Remote User Interface needs to be an application which implements TCP/IP socket server (called SCPI server in this thesis), it accepts and responds to the request from measurement application clients. The architecture of SCPI Server is presented in **Figure 21**. Interpretation of SCPI commands consists of two phases: parsing and decoding. It is followed by command execution using Hardware Interface. Parsing is device-independent. It depends on the syntax which is common for all the SCPI command. After the command has been divided into mnemonics and argument, the command must be decoded and executed. The decoding depends on the commands dictionary (which is based on the controlled hardware capabilities) Execution is device dependant. It needs to be implemented for the particular device. This is why the architecture of SCPI server is modular.

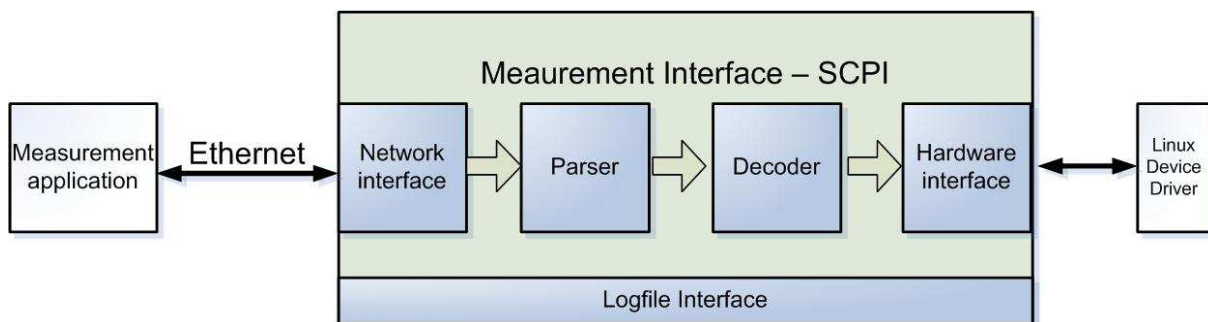


Figure 21 SCPI server architecture

SCPI standard strictly determines server to the following messages:

1. Device settings information if a query is inputted,
2. Measurement data, if data acquisition is turned on.
3. Error codes (a digit) if an error occurs.

To provide user with more information about RMI application performance and detailed error messages, a logfile Interface is used. All the messages about performance of each component of the application and detailed error messages are written to a file.

2.5.1 A Note on SCPI Compliance

When implementing SCPI command interface two approaches are possible:

- Full SCPI compliancy
- SCPI “look and feel” commands

Full SCPI compliancy requires to follow strictly SCPI Standard documentation which defines what certain commands should do, what commands to include for certain instrument classes, etc. Often, full SCPI compliancy is not implemented. Instead, by giving the user the “look and feel” of SCPI, the user will be immediately familiar with the equipment’s control. This approach is extremely common amongst instrument manufacturers. Studying Tektronix’s [7], RIGOL’s [6] and other companies’ programmer’s manuals of digital oscilloscopes, it was noticed that some of the SCPI commands found in the manuals do not comply with SCPI standard but seem useful and reasonable, while mandatory SCPI commands are not implemented because they are not necessary.

The “look and feel” approach was taken in the implementation of SCPI standard for Measurement Interface of Universal Measurement System.

2.6 Summary

Figure 22 summarizes the architecture of UMSWI. For each of the required interfaces (measurement and web interface) a server is provided. Remote Measurement Interface server implements Standard Command for Programmable Instruments (SCPI), thus it is called SCPI Server. The Web Server is provided by Busybox [32]. SCPI and WEB servers communicate with hardware (in order to control acquisition process and retrieve measurement data) using Linux Device Driver. Since SCPI server is developed from scratches, it implements communication with Linux Device Driver. Web Server needs Common Gate Interface (CGI) to communicate with Linux Device Driver.

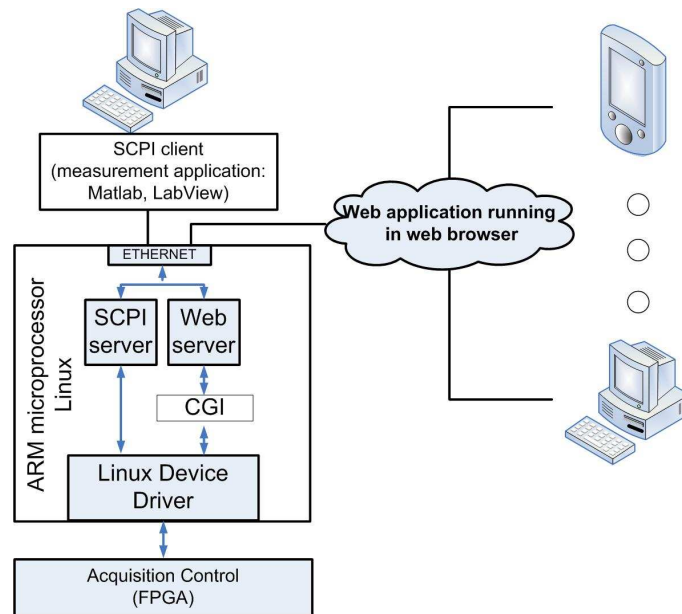


Figure 22 UMSWI architecture [31]

A client to Remote Measurement Interface Server, in principle, is any measurement application (i.e. LabView, Matlab) which enables control of remote instruments via TCP/IP using SCPI commands. A client to Web Server is a web browser with Java Script and Java

Applet enabled. Java Applet requires Java Virtual Machine installed on the client. All the UMSWI-related software is stored on SD/MMC card enabling easy update and modifications.

A careful choice of technologies and well-thought planning resulted in very portable, flexible and easily extensible software architecture. The requirements towards Linux utilities are very basic. The HTTP server needed for the system to operate is provided by Busybox (used by most of the embedded Linux distributions) and only adds 9K, which is not much ever for the Linux distributions with strong memory constraints. In principle, the HTTP server with CGI interface is the only requirement for Linux distribution to run the system.

The only hardware dependant parts of the system are: Linux Device Driver and application which configures FPGA. Only these two components need to be changed to run the system on different hardware.

Application of MVC architecture in applet should result in easy extensions or changes.

3. Design and Implementation

3.1 Development environment

The Universal Measurement System with Web Interface (UMSWI) was being developed for 2 years. It took considerably long time to establish the most convenient development environment, tools and workstation.

A typical cross-development environment according to [26] is presented in **Figure 23**. A *host* is a development workstation, a PC or Laptop, running Linux distribution.

„Webster's defines nonsense as "an idea that is absurd or contrary to good sense." It is my opinion that developing embedded Linux platforms on a non-Linux/UNIX host is nonsensical.” [26]

A *target* is referred to embedded hardware platform (UMSWI). Thus, native development is understood as building of applications on and for the host system. On the contrary, cross-development means the compilation and building of applications on the host system that are supposed to run on the embedded system.

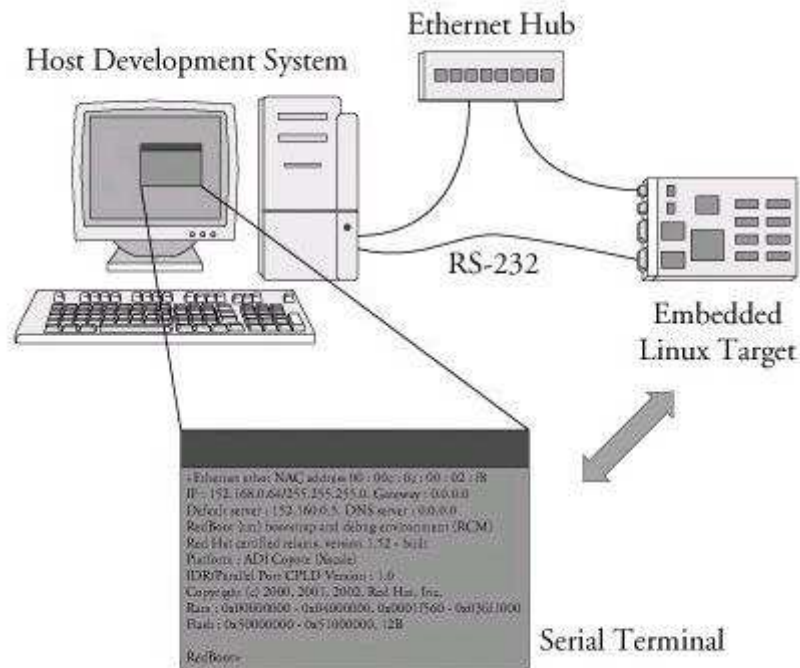


Figure 23 Layout of cross-development environment [26].

The configuration presented in **Figure 23** was used during most of the development of UMSWI. In the final stage of the development, setup was extended to the one presented in **Figure 24**. The host development system was connected to a target board via RS-232 and Ethernet. A serial terminal program (minicom) was used to communicate with the target board via RS-232. The u-boot bootloader, which is stored in the target's flash memory, was started automatically after the power-up. It is a very powerful tool which enabled the image of Kernel along with root filesystem to be downloaded to target board using TFTP protocol over Ethernet. Once downloaded the image was run. During development, NFS root mount

for target board was used. Linux ran on the target board mounted the root filesystem located on the host over NFS. There are many advantages of such a solution :

- Root file system is not size-restricted,
- Any changes to application under development are available to target system immediately, the same files are available to target and host system simultaneously,
- Kernel can be debugged and booted before developing proper root file system,
- It makes development much faster and easier.

A second development computer running Windows XP was used for development and debugging of VHDL design of FPGA logic. This computer is called FPGA development and debugging workstation. Both workstations were connected using NFS file system. It made file exchange very convenient. Altera Quartus II software tool was used for development and debugging of FPGA logic. The debugging was performed using Quartus II tool called Signal Tap II Embedded Logic Analyzer and Byte Blaster II cable. Signal Tap II is a system-level debugging tool which enables to capture and display real-time signals in any FPGA design. Signal Tap II connects via Byte Blaster II download/upload cable with JTAG connection to device under test. The Windows workstation was also used for website development and partly for Applet development with Eclipse. The Eclipse KDE was run on both workstations.

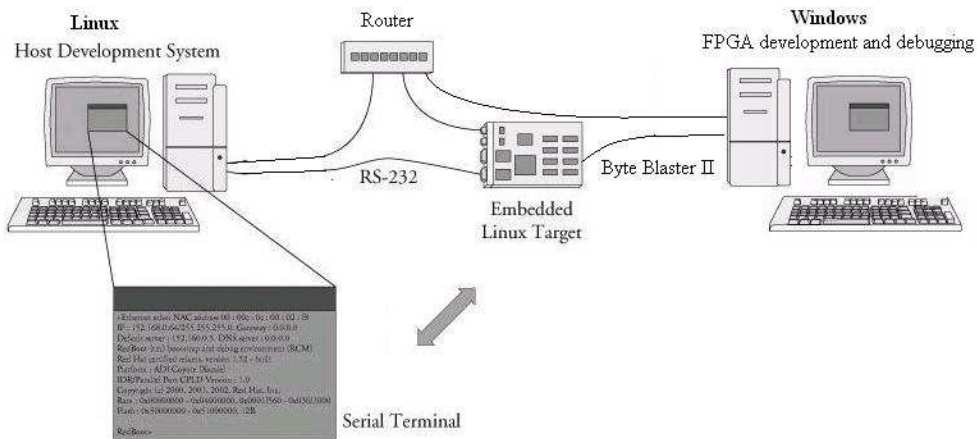


Figure 24 UMSWI development setup

The setup (applications) used for development of Universal Measurement System with Web Interface is summarized in **Table 2**. It was learnt painfully by the author that the most crucial was the choice of Linux distribution for the workstation. During the first year of development SUSE10.2 Linux distribution was used. Under SUSE, few cross-compilation toolchains were tested, e.g. Buildroot and Dan Kegel's crosstool. These toolchains were troublesome to build and not satisfactory in embedded Linux development. Many books about Linux embedded systems, i.e. [33] and [34], mention that Debian distribution is very convenient for embedded system development. Therefore, Debian Linux was installed on the development workstation. It was a very positive change for the UMSWI project development. A considerable number of packages, including ARM development tools (i.e. cross-compilation toolchain), is available for Debian (and Debian-related Linux distributions). The package installation is easy and fast. A cross-compilation toolchain provided as a Debian package by Free Electrons [35] was used during development of Linux, *fpga* driver and *SCPI Server*. The toolchain is based on uClibc library popular among embedded Linux systems developers. The usage of uClibc library allows to save memory space (details in **Appendix A: 3.1**).

UMSWI part	tool/KDE	Operating System
Embedded Linux	ARM cross-toolchain (Debian package)	Debian Linux
Linux Device Driver		Debian Linux
SCPI Server		Debian Linux
Java applet	Eclipse	Windows XP/Debian Linux
Website	-	Windows XP
FPGA logic	Alera Quartus II	Windows XP

Table 2 UMSWI development tools

3.2 Embedded Linux Operating System

The Linux, which is used on UMSWI, is based on TWarm Project [36]. Since the ARMputer module and TWarm board are very similar, the hardware configuration and ports could be applied to Embedded Linux System on UMSWI (with necessary modifications).

3.2.1 Components

Root filesystem

The root file system is based on Filesystem Hierarchy Standard (FHS)[37]. The FHS was trimmed, removing the directories used to provide an extensible multiuser environment, such as: /opt/, /home, /mnt and /root. Only the essential directories were left.

```

|-bin
|-dev
|-etc
|---init.d
|-lib
|-proc
|-sbin
|-usr
|---ARMscope
|---bin
|---sbin
|-var

```

Figure 25 root filesystem hierarchy

Kernel

The main component of the Embedded Linux is the kernel. Kernel used in UMSWI is based on TWarm Project kernel. It is a 2.6.19 “vanilla” kernel [38] patched with AT91 Linux 2.6 appropriate patch [39] with necessary changes to Ethernet PHY.

Busybox

Busybox 1.00 was used to accommodate the root file system with necessary Unix tools which are all symlinks to a single Busybox executable.

C Library

The library was provided by the cross-compilation toolchain which links the cross-compiled applications against *uClibc*, instead of GNU C library (*glibc*). *uClibc* is a special C library for embedded systems which is very popular and supports many platforms (i.e. ARM, MIPS, PPC). It provides most of GNU C library functionality. Most of the applications that can be compiled against *glibc*, should also compile and run using *uClibc*. It substantially reduces embedded systems' size. Only the most necessary library files were copied to the root file system.

3.2.2 Configuration

Kernel

The most important features of kernel's configuration (**Figure 27**) include:

- Initial RAM filesystem and RAM disk (initramfs/initrd) support
- Initramfs enabled with source from a give directory
- Ethernet (10 or 100Mbit) for AT91RM9200 support
- Configured for AT91RM9200 processor (ARCH_AT91RM9200) with support for AT91RM9200-DK Development Board and AT91RM9200-EK Evaluation Kit
- Boot command: "mem=64M root=/dev/mem rw console=ttyS0, 115200" which means
 - "mem=65M" – force usage of a specific amount of memory,
 - "root=/dev/mem rw" – specifies root filesystem
 - "console=ttyS0, 115200" – use serial port number 0 as output console device, baud rate: 115200
- USB support enabled
- Ext2 and VFAT file system support
- /proc file system support
- NFS boot support (during development)
- AT91 SC/MMC Card Interface support

Busybox

The most important of Busybox's configuration (**Figure 26**) include:

- Build Busybox as a static binary (no shared libs)
- Support reading inittab
- httpd Web Server enabled
- Support for Common Gateway Interface (CGI)
- ifconfig enabled with "hw" option
- telnetd, tftp enabled
- support for mounting NFS file systems

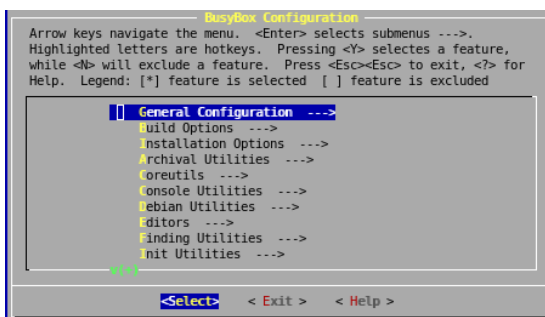


Figure 26 Busybox configuration

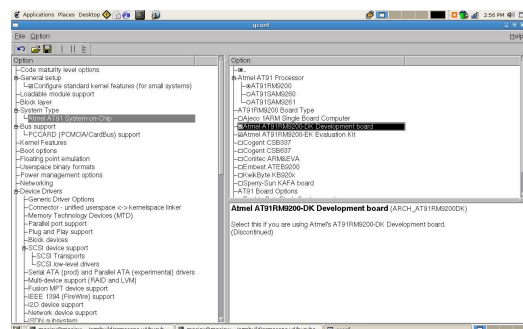


Figure 27 Linux kernel configuration

3.2.3 System boot and startup

Bootling sequence of Embedded Linux implementing *initramfs* is presented in **Figure 28**.

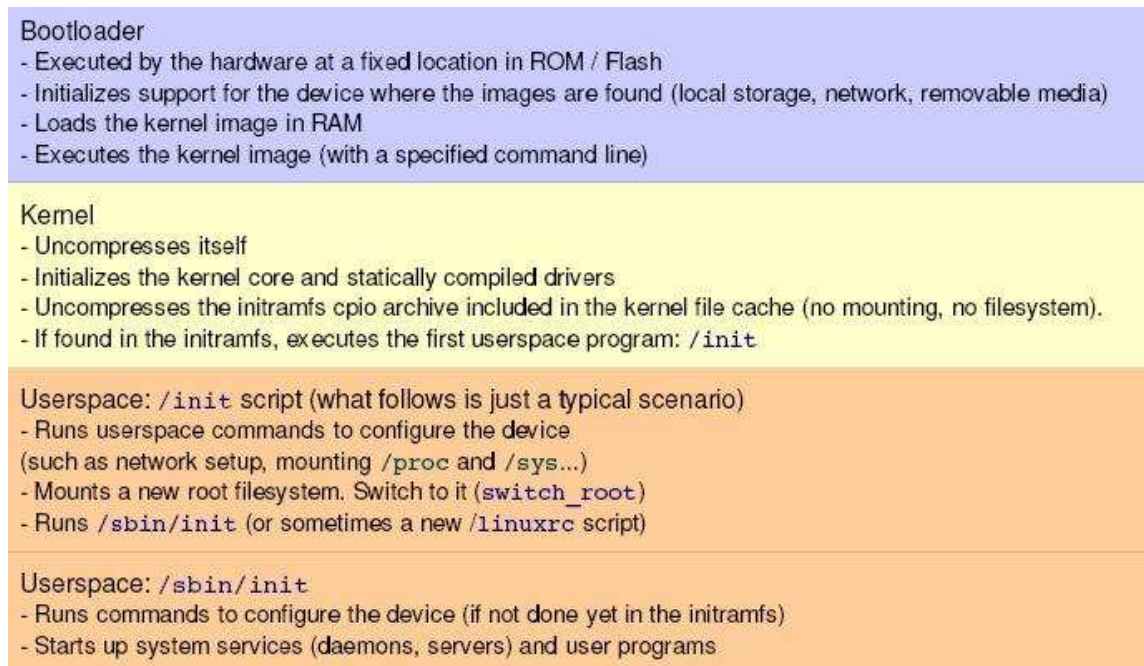


Figure 28 Bootling sequence with *initramfs* [40, page 73]

Bootloader

At the startup, the bootloader is executed automatically from a given location, usually with very little space. Therefore, 2 stages are implemented [29]:

- 1st stage bootlader** – offers minimum functionality and is meant to access and execute the 2nd stage bootloader on a bigger location,
- 2nd stage bootlader** – offers the full bootloader functionality, it can be even an operating system itself.

According to [28] the important features of bootloader include:

- Support for embedded hardware
- Storage footprint
- Support for networking
- Support for flash booting
- Console UI availability
- Upgrade solutions availability
- Argument passing from the boot loader to Linux kernel
- Memory Map
- Calling PPRM routines from the kernel

Three bootloaders suitable for ARM-based embedded systems [29] :

- Das U-Boot: Universal Bootloader from Dentx Software [41]
- RedBoot: eCos based bootloader from Red-Hat [42]
- uMon: MicroMonitor general purpose, multi-OS bootloader [43]

In TWarm project, Darrell Harmond's bootloader [44] (with necessary modifications) is used as a 1st stage bootloader (called loader in this thesis) and Das U-boot (with necessary modifications) is used as 2nd stage bootloader (called u-boot in this thesis). During the development phase, both loaders were used like in TWarm project. U-boot provides many useful utilities, it allows to download kernel image with Trivial File Transfer Protocol

(TFTP), it passes to kernel boot parameters and PHY parameters (i.e. MAC address), it also enables different kinds of booting (from network, MMC/SD card, etc). However, since it was decided to boot the kernel and root file system from flash memory, it turned out that u-boot is not necessary in the normal boot process of UMSWI, provided some modifications are made in Linux start-up script, BusyBox's configuration and Darrell Harmond's bootloader.

A tool enabling MAC address to be set from Linux (Networking Utilities ---> ifconfig/Enable option "hw" (ether only)) was added in BuysBox configuration and Linux start-up script (*/etc/inittab*) was appended to set up MAC address. The loader was modified to perform default Linux start after short delay and BusyBox start on request. Modified loader's menu is presented in **Figure 29**. It was decided to leave the possibility of starting u-boot, since it can be useful for further development and there is enough space in the flash memory. However, a modification was made to the address in which the u-boot is started.

```

Initializing SDRAM
Universal Measurement Sytem - by Maciex
32bit SDRAM 2xHynix HY57V561620C

1: Upload loader to Dataflash with vector 6 modification.
2: Upload u-boot to Dataflash.
3: Upload linux to Dataflash
4: Start U-boot
5: Start linux
6: Start u-boot and linux
7: SDRAM test
    
```

Figure 29 Modified loader's menu

Userspace

UMSWI specific startup operations are done in three steps:

1. The MMC/SD card with UMSWI utilities is attempted to be mounted in */usr/ARMScope/location*. The */usr/ARMScope/* folder hold all the custom-made UMSWI utilities. The mounting is done in */etc/init.d/rcS* (**Figure 30**) system initialization script
2. */usr/ARMScope/start* (**Figure 32**) script is called (in */etc/inittab*) . This script is used for the UMSWI utilities initialization and can be modified by the user easily. It starts the following initialization (by calling appropriate scripts):
 - Configures FPGA (*config_FPGA* script)
 - Loads FPGA driver (*load_driver* script)
 - Starts SCIP server (if option enabled)
 - Sets the default IP (*set_IP* script)
3. httpd web server is started as "respawn" (*/etc/inittab* file, **Figure 31**)

```

1  #!/bin/sh
2
3  mount -t proc none /proc
4  #mount -t devpts none /dev/pts
5
6  #echo 'mounting /usr/ARMScope/'
7  sleep 2
8  mount -t vfat /dev/mmcblk0 /usr/ARMScope/
9  sleep 2
    
```

Figure 30 */etc/init.d/rcS* system initialization script

```

1 # /etc/inittab
2 #
3 # Copyright (C) 2001 Erik Andersen <andersen@codepoet.org>
4 #
5 # Note: BusyBox init doesn't support runlevels. The runlevels field is
6 # completely ignored by BusyBox init. If you want runlevels, use
7 # sysvinit.
8 #
9 # Format for each entry: <id>:<runlevels>:<action>:<process>
10 #
11 # id      == tty to run on, or empty for /dev/console
12 # runlevels == ignored
13 # action  == one of sysinit, respawn, askfirst, wait, and once
14 # process == program to run
15
16 # Startup the system
17 null::sysinit:/sbin/ifconfig eth0 hw ether 00:08:03:7a:3e:16
18 null::sysinit:/sbin/ifconfig lo 127.0.0.1 up
19 null::sysinit:/sbin/route add -net 127.0.0.0 netmask 255.0.0.0 lo
20 null::sysinit:/sbin/ifconfig eth0 192.168.1.101 up
21 null::sysinit:/sbin/route add -net 192.168.1.101 netmask 255.255.255.0 eth0
22
23 # main rc script
24 ::sysinit:/etc/init.d/rcS
25
26 #start ARMScope utilities
27 null::sysinit:/usr/ARMScope/start
28 null::respawn:/usr/sbin/httpd -h /usr/ARMScope/www/
29
30 # Put a getty on the serial port
31 ttyS0::respawn:/sbin/getty -L ttyS0 115200 vt102
32
33 # Stuff to do for the 3-finger salute
34 ::ctrlaltdel:/sbin/reboot
35
36 # Stuff to do before rebooting
37 null::shutdown:/bin/umount -a -r
38
39

```

Figure 31 */etc/inittab* file

```

1 #!/bin/sh
2
3 # configure FPGA
4 /usr/ARMScope/scripts/config_FPGA
5
6 # load FPGA driver
7 /usr/ARMScope/scripts/load_driver
8
9 # start SCPI (if enabled)
10 /usr/ARMScope/scripts/start_scpi
11
12
13 # set default IP
14 /usr/ARMScope/scripts/set_ip

```

Figure 32 */usr/ARMScope/start* script

3.2.4 UMSWI utilities organization

Tools and data which are used by UMSWI are stored in `/usr/ARMscope` folder in root file system and on MMC/SD card. Its organization is presented in **Figure 33**.

```

|-ARMscope
|---data
|---FPGAconfig
|---FPGAdriver
|---scpi_server
|---scripts
|---www
|----cgi-bin
|-----oscilloscope
|-----systemConfig
|----data
|----images
|----oscilloscope
    
```

Figure 33 UMSWI utilities organization

FPGAconfig holds the `.rbf` file with FPGA logic configuration and a small application which configures FPGA.

FPGAdriver holds FPGA Linux Device Driver compiled as a loadable module

scpi_server holds SCPI server application

www holds:

- the UMSWI website (in `www/`),
- CGI scripts (in `www/cgi-bin/`), two kinds:
 - used in applet-driver communication (in `www/scripts/oscilloscope/`)
 - used for system configuration (in `www/scripts/systemConfig/`)
- oscilloscope and spectrum analyzer applet (in `www/oscilloscope/`)
- data available on the website, i.e. Matlab scripts (in `www/downloads/`)
- images used on the website (in `www/images/`)

data holds information which needs to be stored between boots, i.e. default IP

start is a script which starts UMSWI utilities

3.3 Implementation of the FPGA logic in VHDL

Data acquisition and readout are managed by Field Programmable Gate Array (FPGA). The logic for FPGA was created in Very High Speed Integrated Circuits Hardware Description Language (VHDL) using Altera Quartus II programmable logic device design software.

ADCs require low-jitter clock while ARM needs an independent clock for data readout. Therefore there are two clock domains (see **Appendix A: 1.1**) and different parts of the logic needed to be divided according to the clock domain affiliation (**Table 3**).

90 MHz	100 MHz
<ol style="list-style-type: none"> 1. Communication with ARM <ol style="list-style-type: none"> a. Control register b. Readout 2. SSRAM control during processor readout 	<ol style="list-style-type: none"> 1. ADC control 2. SSRAM control during data acquisition 3. ONLINE data analysis 4. Data acquisition parameters implementation (delay, length, sampling time)

Table 3 FPGA logic design components according to frequency affiliation

In terms of clock domain, there is a clear division between FPGA->ARM communication (Communication Logic) and the rest of the logic (Acquisition Management Logic) which is reflected in the architecture of entire FPGA logic. Unfortunately, SSRAM needs to be operated with two different clocks depending on the state (acquisition/readout). General design of FPGA logic is presented in **Figure 34**.

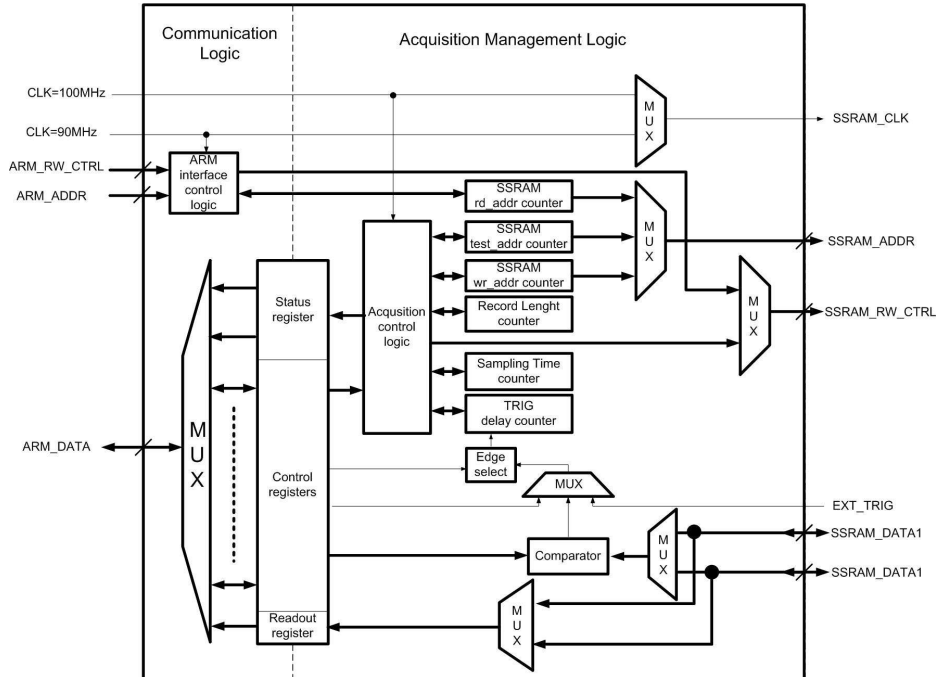


Figure 34 Data acquisition and readout design

3.3.1 Communication logic

Data acquisition is controlled by parameters described in chapter 2.2. Additionally, the following parameters were added during development:

- Readout start address – enables to set the address from which data readout is started. By default, readout starts from the address where first acquisition data was stored (at the moment of trigger detection or after delay)
- Test mode – enables and controls tests of SSRAM.

Parameters returned by acquisition process:

- Status - indicates state of acquisition process,
- Start address pointer - indicates the first address in SSRAM where acquisition data is stored. If delay time is zero, it is the address of the sample during which trigger occurred. If delay time is greater than zero, it is address of the sample stored when the appropriate delay time was counted down.
- Stop address pointer – indicates the last address of the acquisition data stored in SSRAM.

The parameters are stored in FPGA registers which are mapped into ARM address space. The readout register, which enables the acquired data to be transferred from SSRAM to ARM, is also implemented as FPGA registered mapped into ARM address space and managed by the same logic.

Signals responsible for communication between FPGA and ARM are connected on the ARM's side to Static Memory Controller (SMC) which is a part of External Bus Interface (EBI). SMC generates signals that control access to external static memory or peripheral devices (up to 8 devices chosen by chip select lines NCSx). It is fully

programmable and can address up to 512 M bytes. On the FPGA side write/read controller and address decoder are implemented to manage communication with ARM. **Figure 35** presents datasheet schema which was used to connect ARM pins with FPGA pins. It determines the type of communication.

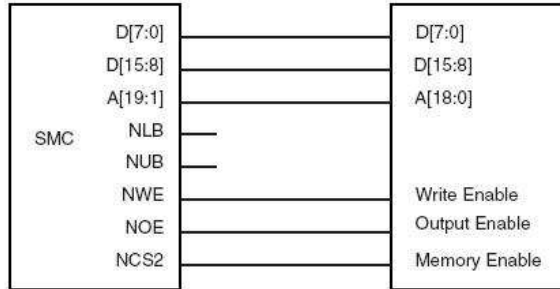


Figure 35 Shows how to connect a 16-bit device without byte access on NSC2 [45]

The following signals enable communication between ARM and FPGA:

- D[15:0] – bidirectional data bus
- A[15:0] – address bus
- NSC0 – chip select number 0
- NWE – write enable
- NOE – output enable signal

The communication protocol is defined in ARM datasheet [45] and can be adjusted by manipulating several parameters (**Table 4**).

Name	Description	Value	SMC setting
Wait select enable	Enables/disables wait states (additional cycles during which NWE/NOE pulse is held low)	enabled	WSEN =1
Number of wait states	Defines the read (NOE) and write (NWE) signal pulse length from 1 cycle to 128 cycles	1	NWS = 1
Data read protocol	Standard or Early Read Protocol	Standard	DRP = 0
Setup delay	Time between the moment when address is available on the bus and write/read enable pulse is set.	1 cycle	RWSETUP = 1
Hold delay	Length of the read/write enable pulse	1 cycle	RWHOLD = 1
Pulse delay	Time between the end of read/write pulse and moment when data ceases to be valid on data bus		

Table 4 Communication SMC settings

Graphic interpretation of the parameters mentioned in **Table 4** is provided in **Table 5** and in **Figure 36**.

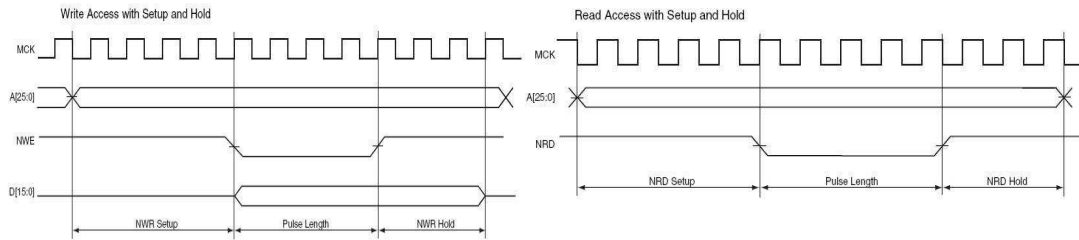


Figure 36 Interpretation of NRD/NWR Setup, Pulse Length and NWR/NRD Hold parameters

Number of Wait States	Read Access	Write Access
0		
1	<p>(1) Early Read Protocol (2) Standard Read protocol</p>	

Table 5 Interpretation of Wait State parameter

In principle, the write/read controller is activated when Chip Select signal (NCS) goes low. The address then is decoded. NWE and NOE signals are monitored, depending which signal goes low, appropriate action is performed (reading/writing). The algorithm is presented in **Figure 37**.

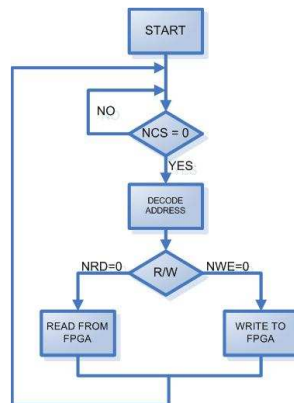


Figure 37 Communication Logic flowchart

The choice of SMC parameters, which is presented in **Table 4**, was made through tests using Signal Tap II Embedded Logic Analyzer. Signal Type II is a tool included with Altera Quartus II software which helps debugging an FPGA design by probing the state of internal signals in the design. Example test of read access is presented in **Figure 38**. The figure shows correct readout. However, write access presented in **Figure 39** indicates that the parameters are inappropriate – only half of the word is written.

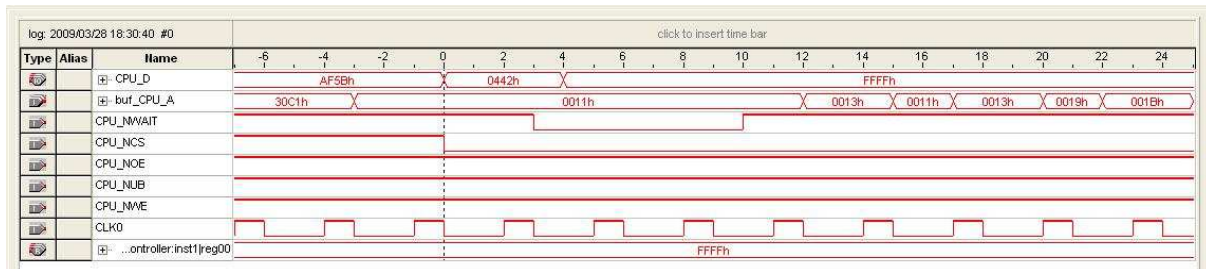


Figure 38 FPGA-ARM communication test

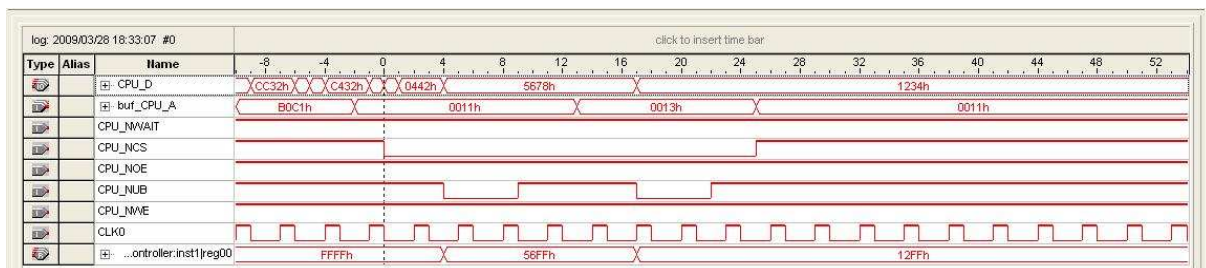


Figure 39 FPGA-ARM communication test

The communication interface between FPGA and ARM is summarized in **Figure 40**.

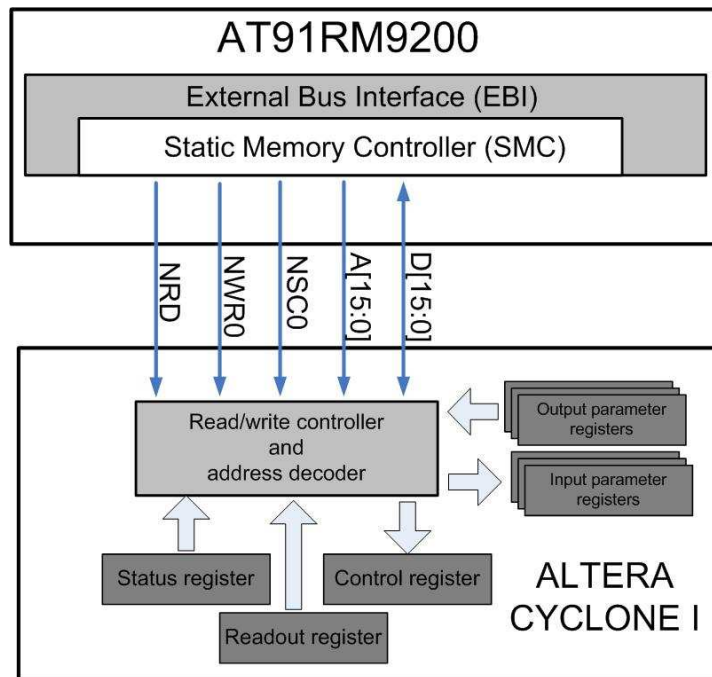


Figure 40 ARM-FPGA interface

3.3.2 Acquisition Management Logic

The process of acquiring data can be divided into three phases:

- Idle – no writing/reading to/from SSRAM, parameters can be set,
- Armed – storing data in SSRAM continuously, waiting for trigger,
- Acquisition – storing required number of samples in SSRAM after trigger occurred (and trigger delay was applied)

It seems reasonable to store the data read from ADCs continuously in SSRAM (Armed phase). The memory is treated as circular buffer. When acquisition is ought to start (trigger detected and delay time counted), the address of appropriate sample is remembered. The processor readout by default starts from this address. Such a solutions enables the user to view data which occurred before the trigger signal (as long as the record length is not equal to 128k, which is the SSRAM size). The proposed acquisition process is summarized in **Table 6**.

N°	Description	State name
1	Parameters are stored in FPGA registers: sampling time, record length, delay, trigger source, trigger level	I D L E
2	Data is stored continuously to SSRAM with programmed frequency (sampling time). If the trigger by signal level has been chosen, simultaneously the level of acquired signal is compared with the trigger level value stored in FPGA register.	A R M E D
3	When trigger signal is detected, the delay counter is activated	T R I G
4	After appropriate delay has been counted, the SSRAM address is stored in FPGA register and data acquisition is started by activating sample length counter.	D E L A Y
5	After appropriate number of samples is stored in SSRAM, the end address is saved in FPGA register and the bit indicating that data has been acquired is activated. This is a signal for ARM processor that data is ready for readout.	A C Q U I R E

Table 6 Acquisition process

Moore finite state machine (FSM) was designed to control data acquisition and readout (**Figure 41**). State machine is in 100 MHz domain. However, it is controlled by registers which communicate with ARM in 90 MHz clock domain. Communication between control registers and microprocessor is available regardless of the FSM state.

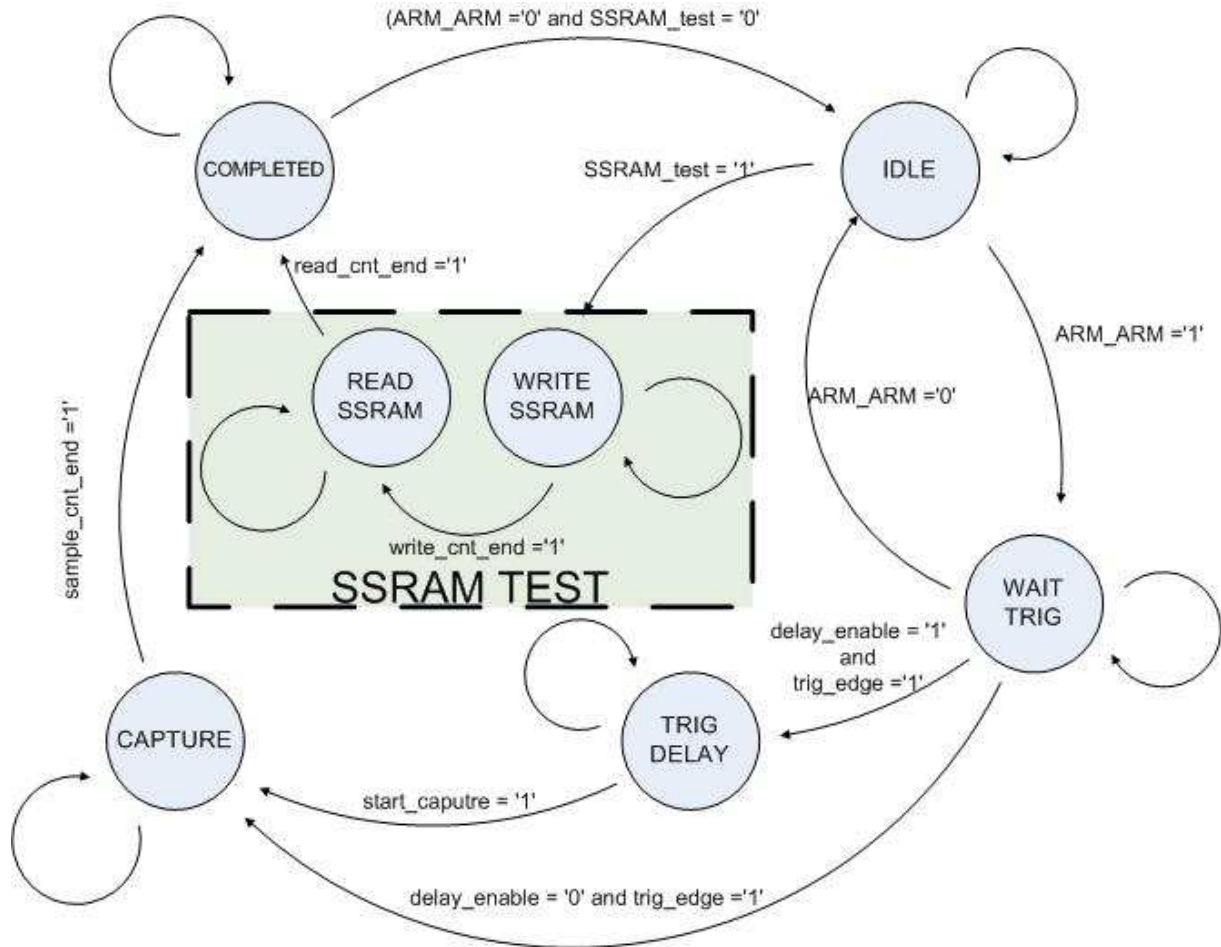


Figure 41 Finite state machine

FSM consists of five main states and two SSRAM test states. Testing features were added in the debugging phase of the project. It was necessary in order to test SSRAM and data buses at the working frequency. Detailed description of all the FSM states is provided in **Table 7**.

State	Description
IDLE	No acquisition, no data readout, all the acquisition parameters are recommended to be set in this state
WAIT TRIG	<p>Clock domain of SSRAM is switched to 100 MHz. Data is read from ADCs and written to SSRAM in sampling time intervals (multiples of 10ns) continuously. SSRAM works as a round buffer. FSM is in this state until trigger is detected or device is “dis-armed” by the user.</p> <p>The following parameters are loaded from control registers during this state: delay time and sample length. It means that change of this parameters by microprocessor in the subsequent states, will not affect current data acquisition process.</p>
TRIG DELAY	<p>If trigger delay is not enabled by the user (trigger delay time equals 0), FSM skips this state. In this state, data is acquired with the appropriate frequency (sampling time). Time set by the user (delay time, multiple of 10ns) is counted down, starting from the trigger occurrence. After appropriate time has collapsed, SSRAM address is stored as start address pointer in the control register. By default, readout starts from this address. However, the address from which readout shall be started, can be set by the user.</p>
CAPUTRE	Data stored to SSRAM with appropriate frequency (sampling time) for user-defined time (number of samples).
COMPLETED	Acquisition is stopped and SSRAM clock is switched to 90 MHz. Flag bit in status control register is set to indicate that data is read for readout. The default readout start address can be changed. This state is maintained even after the readout is completed. So there is possibility to read data multiple times. Return to IDLE state is possible only after ARM bit in control register is set to zero (device “dis-armed”).
WRITE SSRAM	<p>Data is written to SSRAM as if during acquisition. Instead of writing data from ADCs, data is generated by FPGA. There are few test modes which determine what data shall be written to SSRAM:</p> <ul style="list-style-type: none"> • writing address to the memory indicate by the address, • writing 0x5555 to even and 0xAAAA to odd addresses on channel 1 and 0x0000 to channel 2, • writing 0x5555 to even and 0xAAAA to odd addresses on channel 2 and 0x0000 to channel 1, • writing 0x0000 to both channels and all addresses
READ SSRAM	Data is read from the memory (only for Signal Tab II observation)

Table 7 Description of FSM states.

Data (2 x 10bits) is read from both Analogue to Digital Converters (ADCs) simultaneously and written to the same SSRAM word (32bits) under the address indicated by the address counter. Data and address bus width between ARM and FPGA are both 16 bits. Therefore, it is possible to access from ARM directly only $2^{16} = 64$ K of 16-bit words. Since SSRAM has 128 K 32-bit words, it is only possible to access directly 25 % of SSRAM.

Since microprocessor access to SSRAM data is always performed by reading consecutive words starting from a given address, it was decided to solve the problem by mapping entire SSRAM memory into single 16-bit register (readout register). Each time the readout register is read, the address counter is incremented. Since data bus between FPGA and ARM is only 16-bits, one SSRAM word (32-bit) is read in two turns. The least significant bit (LSB) of readout counter indicates which half of the SSRAM word is read (high or low). Readout counter is incremented each time read operation is detected on the readout register. The idea is presented in **Figure 42**.

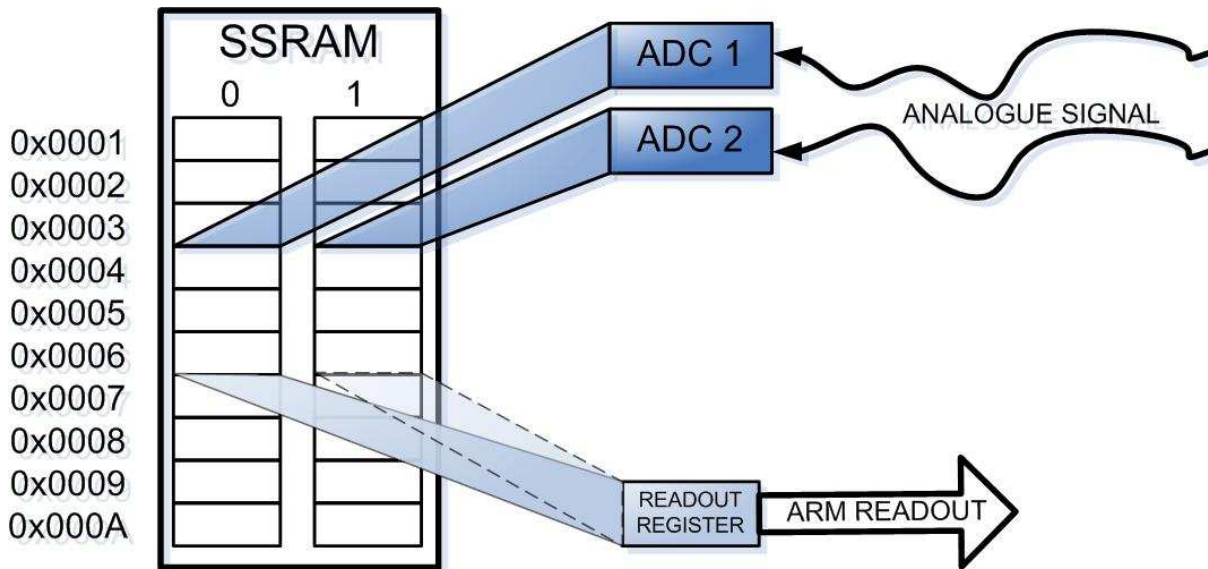


Figure 42 Measured data flow

3.3.3 Trigger detection

There are four possible trigger sources

- a) ARM/AUTO – the user triggers acquisition, it is done by writing appropriate bit in control register
- b) External trigger – signal connected to special trigger input
- c) Channel 1 – trigger by level of input signal to ADC on channel 1
- d) Channel 2 – trigger by level of input signal to ADC on channel 2

These trigger sources fall into two categories:

1. Trigger source is a signal from ADC (c & d)
2. Trigger source is a binary signal (a & b)

In both cases, trigger is detected in so-called edge detector by analyzing four consecutive samples of a binary signal and recognizing appropriate trigger edge (falling/rising). In further case (2), the source is a binary signal which can be directly an input to edge detector (bit in control register, TRIG IN). In former case (1), the source is an analogue signal translated by ADCs to vector discrete values. Thus it needs to be translated into a binary

signal which, in turn, is an input to the edge detector. The translation is done by a comparator. Signal value is compared with set (by the user) trigger level. If signal value is greater than trigger level, high level ('1') is set on the comparators output, otherwise low level ('0') is outputted (**Figure 43**).

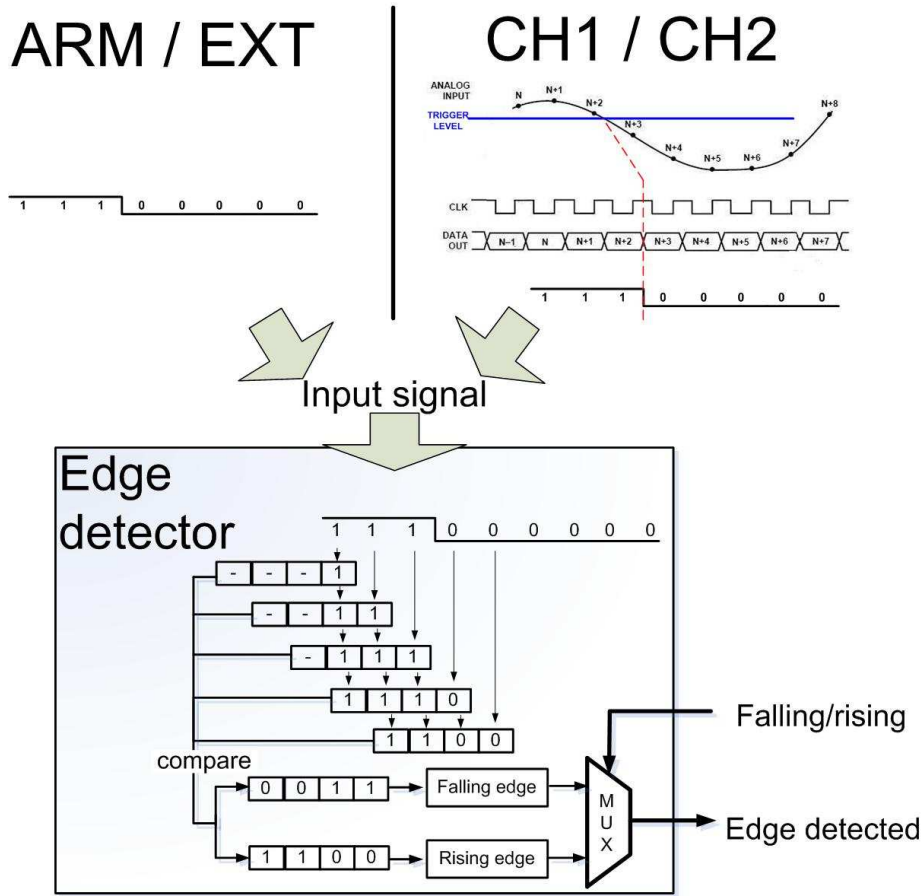


Figure 43 Trigger detection process

3.4 Linux Device Driver

“Device drivers take on a special role in the Linux kernel. They are distinct “black boxes” that make a particular piece of hardware respond to a well-defined internal programming interface; they hide completely the details of how the device works. User activities are performed by means of a set of standardized calls that are independent of the specific driver; mapping those calls to device-specific operations that act on real hardware is then the role of the device driver. “ [30]

Device drivers allow to interact with hardware devices from user space. They provide an abstract layer between hardware and the application/script, thus the higher-level application code can be written independent of the underlying hardware.

In this project device driver is needed to provide interface between Linux user space application (SCPI server) or scripts (CGI) and logic implemented in FPGA. Since the system is meant to be universal (logic implemented in FPGA can be customized or replaced completely), the driver needs to provide flexibility. The driver is also clearly divided into logical and physical layers to make porting to other architectures as simple as possible (**Table 8**).

Functionality	Description	Layer	File name
Driver logic	Hardware independent implementation of driver's logic, uses custom read/write functions to access hardware	abstract	oscilloscope.c oscilloscope.h
Hardware configuration	Called during driver's initialization, used to configure hardware (by writing appropriate registers with appropriate settings) and map physical address into virtual memory address space. Hardware dependent because the configuration needs to be appropriate to the underlying hardware. No direct hardware access.	physic	FPGA_config.c FPGA_config.h
Read/write function implementation	Functions used to access hardware directly by reading or writing appropriate address.		k_IOfpga.c

Table 8 Drivers structure

3.4.1 Abstract layer

The *fpga* driver was developed as character device driver. Char devices are accessed through names in the filesystem. Those names are called device files or nodes and are conventionally located in the */dev* directory. Char nodes are identified by a "c" in the first column of the output of *ls -l*. This command prints also the information about device numbers (major and minor). The major number identifies the driver associated with the device. The minor number is used to determine exactly which device is being referred to. *fpga* driver implements two kinds of device numbers allocation. By default, it allocates device number dynamically since there is a constant effort within Linux kernel development community to move over to the use of dynamically-allocated device numbers and a randomly picked major number can lead to conflicts and troubles if the driver is more widely used. In case a static allocation is desired, it is possible to specify device number at the load time.

The *fpga* implementation uses a global variable, *fpga_major*, to hold the chosen number (there is also a *fpga_minor* for the minor number). The variable is initialized to *FPGA_MAJOR_NUMBER*, defined in *fpga.h*. The default value of *FPGA_MAJOR_NUMBER* in the distributed source is 0, which means "use dynamic assignment." The user can accept the default or choose a particular major number, either by modifying the macro before compiling or by specifying a value for *fpga_major* on the *insmod* command line.

The *fpga* driver connects four basic operations with the reserved device numbers through *file_operations* structure (defined in *<linux/fs>*). The structure is a collection of function pointers. Each open file is associated with its own set of functions which are in charge of implementing the system calls. A *file_operation* structure in *fpga* driver is called *fpga_fops* (according to convention). There are for fields in the structure which point to the functions in the driver that implement the following specific operations (**Figure 44**):

*struct module *owner*

Not an operation but a pointer to the module that “owns” the structure. This field is used to prevent the module from being unloaded while its operations are in use. It is simply initialized to THIS_MODULE, a macro defined in <linux/module.h>.

*int (*fpga_ioctl) (struct inode *, struct file *, unsigned int, unsigned long);*

Implements *ioctl* system call which offers a way to control device.

*int (*fpga_open) (struct inode *, struct file *);*

The first operation performed on the device file. It is used to track the number of opened device references

*int (*release) (struct inode *, struct file *);*

This operation is invoked when the file structure is being released. It is used to track the number of opened device references.

```

134  /** File operation structure.
135      Associates the driver (major number) with device operations.
136      The structure is a collection of function pointers. Each open file is associated with its own set of functions.
137  */
138  static struct file_operations fpga_fops =
139  {
140      .owner      = THIS_MODULE,
141      .ioctl     = fpga_ioctl,
142      .open      = fpga_open,
143      .release   = fpga_release
144  };
145

```

Figure 44 File operations structure

Internally, *fpga* represents each device with a structure of type *struct fpga_Dev* (Figure 45), its components are described below.

```

82  typedef struct {
83      struct cdev          cdev;          /**< Character device structure embedded in device structure. */
84      void *pFPGAaddr;    /**< Virtual address of FPGA. */
85      int reccount;       /**< Keeps the number of opened references to the device (opened files/nodes in /dev folder). */
86      int irq;           /**< Interrupt line number. */
87      char device_name[10]; /**< Name of the device representing it in the user space (both /dev and /proc/driver): ex pscip0, pscip1,... */
88      parameters_t parameters; /**< Oscilloscope parameters. */
89      cmd cmd;
90      unsigned int state; /**< Status. */
91      unsigned int startAddr; /**< Start readout addr. */
92      unsigned int stopAddr; /**< Stop readout addr. */
93      unsigned int startRDaddr;
94      char dataAcquiredFlag;
95      unsigned int readout_number;
96      unsigned int test_register;
97  } nfpga_Dev, *fpga_Dev;

```

Figure 45 Structure which represents FPGA device.

Struct cdev

The kernel uses structures of type *struct cdev* to represent char devices internally. Before the device’s operations can be invoked by the kernel, *cdev* structure must be allocated and registered. The structure and its associated helper functions are defined in <linux/cdev.h> which needs to be included in the driver’s code. Since the *cdev* structure is embedded within *fpga_Dev* structure, it is allocated using *cdev_init()* function.

*void *pFPGAaddr*

A pointer to the beginning of virtual address of FPGA address space. This pointer is used as a base address of all the I/O functions (read/write). It is obtained using *ioremap_nocache* function.

int refcount

Keeps the number of opened references to the device. It is incremented each time *fpga_open* function is called, and decremented each time *fpga_release* function is called.

char device_name[10]

Name of the device, represents the driver in user space (*/proc/drivers/fpga*) and kernel space

parameters_t parameters

A structure storing acquisition parameters (**Figure 46**).

```

73
74  typedef struct {
75      unsigned int    recordLen;    /* Record Length*/
76      unsigned int    time;        /* time/frequency*/
77      unsigned int    delay;      /* delay after trigger*/
78
79  } parameters_t;

```

Figure 46 Structure storing acquisition parameters

unsigned int cmd

unsigned int state

unsigned int startAddr

unsigned int stopAddr

unsigned int startRDaddr

char dataAcquiredFlag

unsigned int readout_number

unsigned int test_register

Variables which reflect the content of appropriate registers in the FPGA logic.

3.4.1.1 Debugging

During the driver’s development debugging was done extensively. Kernel programming brings its own, unique debugging challenges. Kernel code cannot be easily executed under debugger, nor can it be easily traced, because it is a set of functionalities not related to a specific process. Driver errors can bring down the entire system, thus destroying much of the evidence that could be used to track them down.

printk

There are few ways to debug Linux Device Driver, the most commonly used (in general and during development of *fpga* driver) is monitoring, which in applications’ programming is done by calling *printf* at suitable points. In Kernel debugging, the same can be accomplished using *printk*. The *printk* function behaves similarly to the standard C library function *printf*. It is defined in Linux kernel. The kernel needs its own printing function because it runs by itself, without the help of the C library. The kernel messages are appended to */var/log/messages* or printed to the current console.

During the development *printk* was very useful, however, in the final release of the driver printing messages to the console or log file is unnecessary and unwanted. On the other hand, *printk* functions can be found useful if a bug is detected or during further

development of the driver is needed. Therefore, all the *printk* functions which were used for debugging purposes, are included in the pre-processor `#ifdef DEBUG` condition and can be enabled/disabled necessary.

/proc

Another technique used for debugging during the driver's development was querying the system which can be done by creating a file in the `/proc` file system. It is a special, software-created filesystem that is used by the kernel developers to export information to the world. The content of the files under `/proc` is generated on the fly by functions tied to each file.

This solution is heavily used in the Linux system by many utilities such as *ps*, *top* and *uptime* to get their information. Some drivers export information via `/proc`. The `/proc` filesystem can be very conveniently used with CGI scripts to export information from the device and even control the device. Therefore, it is the *fpga* driver's main mean data information exchange and control. This solution, however, has an important disadvantage, which needs to be mentioned but does not overweight advantages. Adding files under `/proc` is discouraged by the kernel developers as `/proc` filesystem is seen as "a bit of an uncontrolled mess that has gone far beyond its original purpose" [30].

3.4.1.2 /proc filesystem

In order to create a read-write `/proc` file, the driver must implement two functions: a function to produce the data when the file is read and a function to read and interpret the data when a file is written to. When a process (application) reads from *fpga* driver's `/proc` file, the kernel allocates a page of memory. The data written to the page by the driver in read-function, is returned to user space. The function presented in **Figure 47** assumes that there will never be a need to generate more than one page of data (it returns value of one control register: 16 bits) and so ignores the start and offset values. All of `proc_read` functions in *fpga* driver, but one, can be implemented in such a simple way, since the amount of data returned by them is precisely known and is less than a page.

```

620 static int cmd_read_proc(char *page, char **start, off_t offset, int count, int *eof, void *data)
621 {
622     oscilloscope_Dev dev = &oscilloscope_devices;
623
624     int len = 0;
625     unsigned int cmd;
626     #ifdef __DEBUG__
627     printk(KERN_ALERT "DEBUG_M: cmd write- beginning\n");
628     #endif
629
630     IOFpga_read_word(dev->pFPGAaddr + CMD_ADDR, &cmd);
631     len += sprintf(page+len, "\n%d\n", CMD_MASK & cmd );
632
633     #ifdef __DEBUG__
634     printk(KERN_ALERT "DEBUG_M: cmd write - end\n");
635     #endif
636     return len;
637 }

```

Figure 47 Function which generates data when `/proc/fpga/cmd` file is read

The only exception is the function that returns the measurement data. The amount of data returned is not constant, it depends on the acquisition parameters. It is very likely that more than one page of data is returned. Thus implementation of multiple pages `/proc` file was necessary. It was done using *seq_file* interface. This interface provides a set of functions for implementation of large kernel virtual files. It assumes that the `/proc` virtual file steps

through a sequence of items that must be returned to the user space, therefore, an “iterator” object needs to be created. `Seq_file` needs four iterator methods called `start`, `next`, `stop` and `show`.

The `start` function is always called first (**Figure 48**). It reads `record length` FPGA register to find out how many double words is to be read. It also reads `test` FPGA register to check whether the reading is performed in normal mode or in test mode. If a specific test mode is on, the data is outputted in a special form. `pos` is an integer position indicating how many double words have been read.

```

1016  /* executed always at the beginning of the readout sequence. */
1017  static *readresult_seq_start(struct seq_file *sfile, loff_t *pos)
1018  {
1019      unsigned int readCount;
1020      unsigned int readTest;
1021      oscilloscope_Dev dev = &oscilloscope_devices;
1022      #ifdef __DEBUG__
1023      printk(KERN_ALERT "DEBUG_M: readresult_seq_start \n");
1024      #endif
1025      IOFpga_read_2words(dev->pFPGAaddr + REC_LEN_ADDR, &readCount);
1026      dev->readout_number = readCount;
1027      IOFpga_read_word(dev->pFPGAaddr + TEST_ADDR, &readTest);
1028      dev->test_register = readTest;
1029
1030      if (*pos >= readCount || *pos >= MAX_RECORD_LEN) //check if you read all the data
1031          return NULL;
1032      return (1 + *pos); //pos needs to be not NULL to work
1033  }
    
```

Figure 48 Implementation of start method in the `seq_file` interface

The `next` function should move the iterator to the next position (**Figure 49**). It increments the `pos` variable and checks it against the expected number of words to be read. It returns `NULL` if there is nothing left in the sequence.

```

1035  static void *readresult_seq_next(struct seq_file *s, void *v, loff_t *pos)
1036  {
1037      #ifdef __DEBUG__
1038      printk(KERN_ALERT "DEBUG_M: readresult_seq_next \n");
1039      #endif
1040      (*pos)++;
1041      if (*pos >= oscilloscope_devices.readout_number || *pos >= MAX_RECORD_LEN) //check if you read all the data
1042          return NULL;
1043      return (1 + *pos);
1044  }
    
```

Figure 49 Implementatin of `seq_next`

When all the acquired data is read and the kernel is done with the iterator, it calls `stop` function. There is no action required in fpga driver implementation, so the function is empty. In between these calls, the kernel calls `show` (**Figure 50**) method to output measurement data to the user space. This method creates output for the item in the sequence. It uses special function for `seq_file` output (`seq_printf`). The `show` function reads two words (16bits). It performs two consecutive accesses to the address of readout register. Each access increments the SSRAM address counter in FPGA logic. One call to `show` function causes one SSRAM word (32bits) to be read. Each SSRAM word consist of the measurement from 2 ADCs.


```

1051  /* creates data sent to user space */
1052  static int readresult_seq_show(struct seq_file *s, void *v)
1053  {
1054      oscilloscope_Dev dev = &oscilloscope_devices;
1055      signed int first_word;
1056      signed int second_word;
1057      signed int voltage_1;
1058      signed int voltage_2;
1059      int * number = (int *) v;
1060      #ifdef __DEBUG__
1061      printk(KERN_ALERT "DEBUG_M: readresult_seq_show \n");
1062      #endif
1063      //read from device
1064      IOFpga_read_word(dev->pFPGAaddr + DATA_ADDR, &first_word);
1065      IOFpga_read_word(dev->pFPGAaddr + DATA_ADDR, &second_word);
1066      //calculate voltage in mV
1067
1068      voltage_1 = (1000 * VPP * 1000 * (first_word - 512))/1024;
1069      voltage_2 = (1000 * VPP * 1000 * (second_word - 512))/1024;
1070
1071
1072      voltage_1 = voltage_1/1000;
1073      voltage_2 = voltage_2/1000;
1074      //show to the user
1075
1076      if(TEST_ON_MASK & (dev->test_register))
1077          seq_printf(s, "0x%x ->> 0x%x: 0x%x\n", number, first_word, second_word);
1078      else
1079          seq_printf(s, "%d: %d\n", voltage_1, voltage_2);
1080
1081      #ifdef __DEBUG__
1082      printk(KERN_ALERT "DEBUG_M: readresult_seq_show - end \n");
1083      #endif
1084  }
1085
    
```

Figure 50 *seq_file* show method which outputs measurement data to user space

The data read from SSRAM is in Binary Offset format, which is determined by the ADC's hardware mode setting. To make the driver universal and user-friendly, it was decided that the output format of the data should be readable. Since only natural numbers are allowed in device drivers, and the resolution of ADCs is approximately 1mV (10 bits and 1ppV gives 1000mv/1024), it was decided to output data in milivolts. The *show* function performs the necessary conversation. To make SSRAM testing more convenient, the data is outputted in the hex format during test mode.

All the iterator operations (*start*, *stop*, *next*, *show* functions) are packaged up and connected to a file in */proc* by filling in a *seq_operations* structure (**Figure 51**) and creating a file implementation. The connection to */proc* is made creating *file_operations* structure (**Figure 52**) and necessary *open* method (**Figure 53**), which connects the file to the *seq_file* operations.

```

165  /** Used for packaging set of the interior operations of seq_file interface.
166
167  */
168  static struct seq_operations readresult_seq_ops = {
169      .start    = readresult_seq_start,
170      .next     = readresult_seq_next,
171      .stop    = readresult_seq_stop,
172      .show    = readresult_seq_show
173  };

```

Figure 51 Seq_operations structure

```

153  /** File operation structure.
154      Defines sequence operations - used in seq_file interface
155      seq_file interface is to read large procFS files
156
157  */
158  static struct file_operations readresult_proc_ops = {
159      .owner    = THIS_MODULE,
160      .open    = readresult_proc_open,
161      .read    = seq_read,
162      .llseek  = seq_lseek,
163      .release = seq_release
164  };

```

Figure 52 File operations structure

```

1007  /* Connects the file structure with our sequence operations */
1008  static int readresult_proc_open(struct inode *inode, struct file *file)
1009  {
1010      #ifdef __DEBUG__
1011      printk(KERN_ALERT "DEBUG_M: readresult_proc_open \n");
1012      #endif
1013      return seq_open(file, &readresult_seq_ops);
1014  }

```

Figure 53 Proc open method

The FPGA logic (i.e acquisition) can be controlled by writing appropriate data to */proc* files. This is possible by binding *write_proc* functions with */proc* files. **Figure 54** presents implementation of function called when */proc/fpga/cmd* file is read.

```

595  static int cmd_write_proc(struct file *filp, const char *buffer, unsigned long count, void *data)
596  {
597      oscilloscope_Dev dev = &oscilloscope_devices;
598      unsigned int cmd;
599
600      #ifdef __DEBUG__
601      printk(KERN_ALERT "DEBUG_M: cmd write- beginning\n");
602      #endif
603
604      sscanf(buffer, "%d", &cmd); //get data from the user
605
606      dev->cmd = CMD_MASK & cmd; //filter data
607
608      IOfpga_write_word(dev->pFPGAaddr + CMD_ADDR, CMD_MASK & cmd); //write to FPGA
609
610      #ifdef __DEBUG__
611      printk(KERN_ALERT "DEBUG_M: cmd write - end\n");
612      #endif
613      return count;
614  }

```

Figure 54 Implementation of *write_proc* function

buffer is a pointer to a page of data retrieved from user space. This data is read and interpreted by the function. Then appropriate hardware access is performed.

All the `read_proc` and `write_proc` functions need to be connected to entries in the `/proc` hierarchy using. This is done in `procfs_register` function (**Figure 55**). In the first place, the function allocates memory for the proc device data structure which is used to pass information between `read_proc` and `write_proc` of `readXwords /proc` entry. Then the appropriate `/proc` path is created (`/proc/driver/fpga`) and registered. Finally, all the `/proc` entries are connected to appropriate functions (`read_proc` and/or `write_proc` accordingly). As an example, **Figure 55** presents how `cmd /proc` entry is bound with appropriate functions.

```

1395  /**
1396      Register proc filesystem entries. It creates the appropriate directory in /proc/driver/ and fills it with nodes representing each IP
1397      It registers functions implemented for proc_fs interface
1398  */
1399  static int fpga_procfs_register(void)
1400  {
1401      struct proc_dir_entry *procfs_file_cmd;
1402      struct proc_dir_entry *procfs_file_state;
1403      struct proc_dir_entry *procfs_file_dataAcquired;
1404      struct proc_dir_entry *procfs_file_reset;
1405      struct proc_dir_entry *procfs_file_params;
1406      struct proc_dir_entry *procfs_file_pAddr;
1407      struct proc_dir_entry *procfs_file_startRDaddr;
1408      struct proc_dir_entry *procfs_file_readSingleData;
1409      struct proc_dir_entry *procfs_file_read2words;
1410      struct proc_dir_entry *procfs_file_readXwords;
1411      struct proc_dir_entry *procfs_file_readresult;
1412      struct proc_dir_entry *procfs_file_config;
1413      struct proc_dir_entry *procfs_file_info;
1414      struct proc_dir_entry *procfs_file_test;
1415      struct proc_dir_entry *procfs_file_trig_level;
1416      struct proc_dir_entry *procfs_file_fpga;
1417
1418      char root_dir[30]="driver", model_dir[30];
1419      char module_name[30]="fpga"; //module directory
1420
1421      #ifdef _DEBUG_
1422      printk(KERN_ALERT "DEBUG_M: pscip_procfs_register - beginning\n");
1423      #endif
1424      // allocate proc device data structure
1425      fpga_procdev = kmalloc(sizeof(nfpga_Proc), GFP_KERNEL);
1426      if (!fpga_procdev)
1427          goto err_kmalloc;
1428      memset(fpga_procdev, 0, sizeof(nofpga_Proc));
1429      //create dir path
1430      sprintf(model_dir,"%s/%s",root_dir,module_name);
1431
1432      #ifdef _DEBUG_
1433      printk(KERN_ALERT "DEBUG_M: model_dir: %s\n", model_dir);
1434      #endif
1435      // register proc entry under "driver" entry
1436      proc_model_dir = proc_mkdir(model_dir, NULL);
1437      if (proc_model_dir == NULL)
1438          goto err_proc_mkdir;
1439      //cmd
1440      procfs_file_cmd = create_proc_entry("cmd", S_IRUGO|S_IWUSR, proc_model_dir);
1441      if (procfs_file_cmd == NULL)
1442      {
1443          remove_proc_entry(model_dir, NULL);
1444          goto err_create_proc_entry;
1445      }
1446      procfs_file_cmd->write_proc = cmd_write_proc;
1447      procfs_file_cmd->read_proc = cmd_read_proc;
1448
1449      .....
1450
1451      .....
1452
1453      .....
1454
1455      .....
1456
1457      .....
1458
1459      .....
1460
1461      .....
1462
1463      .....
1464
1465      .....
1466
1467      .....
1468
1469      .....
1470
1471      .....
1472
1473      .....
1474
1475      .....
1476
1477      .....
1478
1479      .....
1480
1481      .....
1482
1483      .....
1484
1485      .....
1486
1487      .....
1488
1489      .....
1490
1491      .....
1492
1493      .....
1494
1495      .....
1496
1497      .....
1498
1499      .....
1500
1501      .....
1502
1503      .....
1504
1505      .....
1506
1507      .....
1508
1509      .....
1510
1511      .....
1512
1513      .....
1514
1515      .....
1516
1517      .....
1518      return -ENOMEM;
1519  }
1520  }
    
```

Figure 55 `procfs_register` function

3.4.1.3 ioctl

Most drivers implement *ioctl* system call, which supports user space requests via the *ioctl* method. In the user space, the *ioctl* system call have the following prototype:

```
int ioctl(int fd, unsigned long cmd, ...)
```

The dots in the prototype represent a single optional argument. In *fpga* driver implementation, a pointer to a structure is mostly used since it enables to exchange any amount of data with user space. The driver's header file defines structures which are passed to the *ioctl* system call as the third argument. User programs must include that header file to control the driver. The header defines also symbolic names representing commands' numbers. **Figure 56** presents an example implementation of *ioctl* command called "cmd". *Ioctl* commands need to copy data to or from the user address space. It is done by the following kernel functions, which copy an arbitrary array of bytes and sit at the heart of the *ioctl* implementation.

```
unsigned long copy_to_user(void __user *to, const void *from, unsigned long count);  
unsigned long copy_from_user(void *to, const void __user *from, unsigned long count);
```

The usage of these functions can be seen in *cmd* implementation on **Figure 56**. In *CMD_WRITE*, the data (number representing a command) is copied from the user-space and written to the hardware. In *CMD_READ*, the data is read from the hardware, filtered and copied to the user-space. This schema is repeated with the implementation of the other *ioctl* command.

Universal Measurement System with Web Interface

```

246  /**
247      Used to read-write I/O operations and IP configuration
248
249      @param *inode    - contains cdev structure, allows to get our device structure
250      @param *filp     - pointer to a file structure
251      @param cmd       - command to be executed
252      @param arg       - data passed from/to kernel space
253
254  */
255  static int fpga_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg)
256  {
257      singleReg_t singleReg;
258      parameters_t parameters;
259      dataX2_t dataX2;
260      dataX_t dataX;
261      int i = 0;
262      unsigned int rdData;
263      unsigned int count;
264      oscilloscope_Dev dev = filp->private_data;
265
266      #ifdef __DEBUG__
267      printk(KERN_ALERT "DEBUG_M: oscilloscope_ioctl() beginnig \n");
268      #endif
269
270      switch (cmd)
271      {
272          //Write command (ARM, Trigger)
273          case CMD_WRITE :
274
275              #ifdef __DEBUG__
276              printk(KERN_ALERT "DEBUG_M: --\t CMD writing -beginning\t--\n");
277              #endif
278              if (copy_from_user (&singleReg, (void *) arg, sizeof (singleReg_t)))
279              {
280                  return (-EFAULT);
281              }
282
283              singleReg.data = 0x028 & singleReg.data;
284              IOFpga_write_word(dev->pFPGAAddr + CMD_ADDR, (0xFFFF & singleReg.data));
285
286              #ifdef __DEBUG__
287              printk(KERN_ALERT "DEBUG_M: --\t CMD writing -finished\t--\n");
288              #endif
289
290              break;
291
292          // Read commands (ARM, Trigger)
293          case CMD_READ :
294
295              #ifdef __DEBUG__
296              printk(KERN_ALERT "DEBUG_M: --\t CMD reading -beginning\t--\n");
297              #endif
298
299              IOFpga_read_word(dev->pFPGAAddr + CMD_ADDR, &singleReg.data);
300              singleReg.data = 0x0028 & singleReg.data; //filter out only command
301              if (copy_to_user ((void*) arg, &singleReg, sizeof (singleReg_t)))
302              {
303                  return (-EFAULT);
304              }
305
306              #ifdef __DEBUG__
307              printk(KERN_ALERT "DEBUG_M: --\t CMD Reading -finished\t--\n");
308              #endif
309
310              break;
311
312          break;
313          default:
314              return -EINVAL;
315      }
316
317      #ifdef __DEBUG__
318      printk(KERN_ALERT "DEBUG_M: oscilloscope_ioctl() end \n");
319      #endif
320      return 0;

```

Figure 56 ioctl driver method

/proc and ioctl functionality

/proc file entries and *ioctl* commands implemented by *fpga* driver provide the same functionality. Therefore, both can be used interchangeably. **Table 8** presents the driver's interface (/proc and ioctl). *ioctl*'s third argument is a pointer to data structure, depending on the command, 5 different data structures are used to exchange data between user-space and the driver. The structures are presented in **Figure 57**.

```

69  /* ioctl data struct */
70  typedef struct {
71      unsigned int    data;    /* CMD, status, addresses*/
72  } singleReg_t;
73
74  typedef struct {
75      unsigned int    recordLen;    /* Record Length*/
76      unsigned int    time;        /* time/frequency */
77      unsigned int    delay;      /* delay after trigger */
78  } parameters_t;
79
80
81  typedef struct {
82      unsigned int    data1;    /* lower address data */
83      unsigned int    data2;    /* higher address data*/
84  } dataX2_t;
85
86  typedef struct {
87      unsigned int    addr;    /* address */
88      unsigned int    data;    /* data*/
89  } fpga_t;
90
91
92  typedef struct {
93      unsigned int    data1[131072];    // data array 128*1024
94      unsigned int    data2[131072];    // data array 128*1024
95      unsigned int    count;    // number of valid data
96  } dataX_t;
97
    
```

Figure 57 ioctl data structures

The driver interface is divided into:

- general purpose – can be used to access FPGA with any configuration, prepared for custom-made configuration,
- FPGA control logic specific – specific for the control logic implemented in FPGA,
- oscilloscope specific – for oscilloscope/spectrum analyzer implementation.

Command name	ioctl		/proc filesystem			
	ioctl command	Arg	File mane	format	R/W	Example commands
Reset	RESET	-	reset	1	W	Echo 1 > reset
General FPGA access	FPGA	Fpga_t	fpga	r/w Addr data (hex)	R/W	Echo w 0x220 0x33 > fpga Cat fpga
Configure	CONFIG_READ	singleReg_t	Config	Number (decimal)	R/W	Echo 1 > config Ext. trigger, rising slope, sampling time disabled
	CONFIG_WRITE					
Command	CMD_READ	singleReg_t	cmd		R/W	Echo 16 > cmd Echo 32 > cmd Echo 48 >cmd ARM TRIGGER ARM & TRIGGER
	CMD_WRITE					
Status	STATE_READ	singleReg_t	state		R	Cat state 1 2 3 Data acquired ARMED Data acquired and armed
Parameters	PARAM_READ	Parameters_t	parameters	Len:time:delay (decimals)	R/W	Echo 20:0:0 > parameters
	PARAM_WRITE					
Custom start read address pointer	START_RD_ADDR_READ	singleReg_t	startRDaddr	startAddr (hex)	R/W	Echo 200 > startRDaddr
	START_RD_ADDR_WRITE					
Address pointers (start & stop)	ADDR_POINTERS_READ	dataX2_t	addressPointers	StartAddr stopAddr (hex)	R	200 500
Read single data	DATA_1_READ	singleReg_t	readSingleData	0xdata (no conversion, hex)	R	Cat readSingleData
Read two words (one SSRAM word)	DATA_2_READ	dataX2_t	read2words	Voltage_ADC_1 Voltage_ADC_2 (converted, decimal, [mV])	R	Cat read2words
Read x SSRAM words	DATA_X_READ	dataX_t	readXwords	Volt_adc1, volt_adc2 (converted, decimal, [mV])	R/W	Echo 20 > readXwords Cat readXwords
Read entire measurement data			readresult	Volt_adc1, volt_adc2 (converted, decimal, [mV])	R	Cat readresult
Test	TEST	singleReg	test		R/W	Echo 21 > test Cat test

Table 9 ioctl/proc interface

3.4.2 Physical layer

Physical layer comprises of the function which are hardware dependent. In particular there are two kinds of such functions:

- Read/write,
- Hardware configuration and memory mapping.

Read/write functions are wrappers of special kernel I/O memory access functions (provided via `<asm/io.h>`). Read/write wrappers enable to abstract physical layer from logical layer. During development they allowed to test various hardware access solutions without changing logical layer. Read/write wrappers implement the specific types of I/O access which are needed in logical layer, namely:

- Read/write 16 bits from/to FPGA
- Read/write 32 bits from/to FPGA
- Read/write 32 bits from/to ARM SMC register

All the wrapper functions embed debugging facilities and memory barriers which prevent compiler optimization.

Since the data bus width between ARM and FPGA is 16 bits, 32-bit access to FPGA is performed as two 16-bit accesses. Moreover, due to hardware problems, the least significant byte of the address is ignored, when addressing control registers. Therefore, 32-bit access to FPGA is implemented in the following way (**Figure 58**):

```
49 void IOFpga_read_2words(void *address, unsigned int *val)
50 {
51     unsigned int temp;
52
53     *val = ioread16(address);
54     temp = ioread16(address + 0x10);
55     *val = *val + (temp << 16);
56     rmb();
57
58     #ifdef __DEBUG__
59     printk(KERN_ALERT "\t\t DEBUG_K: ioread32(0x%x, 0x%x)\n", (unsigned int)address, (unsigned int)*val);
60     #endif
61 }
62 void IOFpga_write_2words(void *address, unsigned int val)
63 {
64     iowrite16( (0xFFFF & val) ,address);
65     iowrite16( (0xFFFF & (val >> 16)),address + 0x10);
66     rmb();
67
68     #ifdef __DEBUG__
69     printk(KERN_ALERT "\t\t DEBUG_K: iowrite32(0x%x, 0x%x) \n", (unsigned int)address, (unsigned int)val);
70     printk(KERN_ALERT "\t\t addr as int: %d\n", (int)address);
71     #endif
72 }
```

Figure 58 2 words (32-bits) FPGA IO functions

The functions presented on **Figure 58** are used, for example, when accessing acquisition parameters (sample length, sampling time, trigger delay). These parameters are more than 16-bit wide, thus they are stored in two consecutive control registers. However, only one wrapper function needs to be used to read their value (**Figure 59**).

```
405
406 IOFpga_read_2words(dev->pFPGAaddr + REC_LEN_ADDR, &parameters.recordLen);
407 IOFpga_read_2words(dev->pFPGAaddr + TIME_ADDR, &parameters.time);
408 IOFpga_read_2words(dev->pFPGAaddr + DELAY_ADDR, &parameters.delay);
409
```

Figure 59 Using FPGA IO functions

On the other hand, the wrapper function which is used for setting configuration registers in ARM can access 32 bits at once. Therefore, the implementation is much simpler (**Figure 60**).

```

63 void reg_read(void *address, unsigned int *val)
64 {
65     *val = ioread32(address);
66     rmb();
67     #ifdef __DEBUG__
68         printk(KERN_ALERT "\t\t DEBUG_K: ioread32(0x%x, 0x%x)\n", (unsigned int)(address), *val);
69     #endif
70 }
71
72 void reg_write(void *address, unsigned int val)
73 {
74     iowrite32(val, address);
75     wmb();
76     #ifdef __DEBUG__
77         printk(KERN_ALERT "\t\t DEBUG_K: iowrite32(0x%x, 0x%x) \n", (unsigned int)(address), (unsigned int)val);
78     #endif
79 }

```

Figure 60 Implementation of read/write ARM register functions

The main task of **configuration and memory mapping function** are

- Map control registers of External Bus Interface (EBI) User Interface, Static Memory Controller (SMC) and Power Management Controller (PMC) into virtual address space,
- initialize EBI, SMC and PMC with appropriate parameters,
- map I/O physical address into virtual address space.

The EBI Chip Select Assignment Register is used to determine to which Chip Select pin the FPGA is connected. FPGA address bus and data bus are connected to Static Memory Controller(SMC). SMC controls access to external static memory and peripheral devices. It is fully programmable by setting appropriate parameters in the SMC Chip Select Register (**Table 10**). The frequency on which SMC works is set by writing PMC Programmable Clock Register (PMC_PCK) and PMC System Clock Enable Register (PMC_SCER).

Name	Description	Value	SMC setting
Data width	Determines data bus width	16 bits	DBW = 1
Data float time	External bus is marked occupied and cannot be used by another chip select during TDF cycles	0	TDF = 0
Byte access type	Used if data width is 16 or 32 bits, defines whether chip select line is connected to two/four 8-bit wide devices or one 16 bit wide device		BAT = 0
Wait select enable	Enables/disables wait states (additional cycles during which NWE/NOE pulse is held low)	enabled	WSEN =1
Number of wait states	Defines the read (NOE) and write (NWE) signal pulse length from 1 cycle to 128 cycles	1	NWS = 1
Data read protocol	Standard or Early Read Protocol	Standard	DRP = 0
Setup delay	Time between the moment when address is available on the bus and write/read enable pulse is set.	1 cycle	RWSETUP = 1
Hold delay	Length of the read/write enable pulse	1 cycle	RWHOLD = 1

Table 10 SMC configuration

3.5 Binding Web Interface to Device Driver with CGI

The communication between applet and hardware (more precisely Linux Device Driver) is performed using Common Gate Interface (CGI).

“CGI is the part of the Web server that can communicate with other programs running on the server. With CGI, the Web server can call up a program, while passing user-specific data to the program (such as what host the user is connecting from, or input the user has supplied using HTML form syntax). The program then processes that data and the server passes the program's response back to the Web browser” [3].

A short explanation how CGI scripts work is included **Appendix A: 2.3. Table 11** presents structure of CGI requests.

Action	GET request send by the applet to the server
Read from hardware	<i>/cgi-bin/oscilloscope/get_name_.cgi</i>
Write to hardware	<i>/cgi-bin/oscilloscope/get_name_.cgi?param_value</i>

Table 11 GET requests: *_name_* is the name of hardware parameter [31].

In Universal Measurement System with Web Interface CGI is used in a non-standard way. It is called from the applet application and the output is never shown to the user directly. It is either ignored (when data is send to the hardware) or stored in applet variable for further processing. The applet provides special universal functions to read/write data from/to hardware. Such functions create appropriate requests to the server, send them and read the answer. The method used to pass the data to the server is GET. Since GET is used, server stores the argument of the request (everything after ‘?’) in environmental variable QUERY_STRING which can be read in the script. Example “get” and “set” scripts are presented in **Figure 61**, detailed description is also provided.

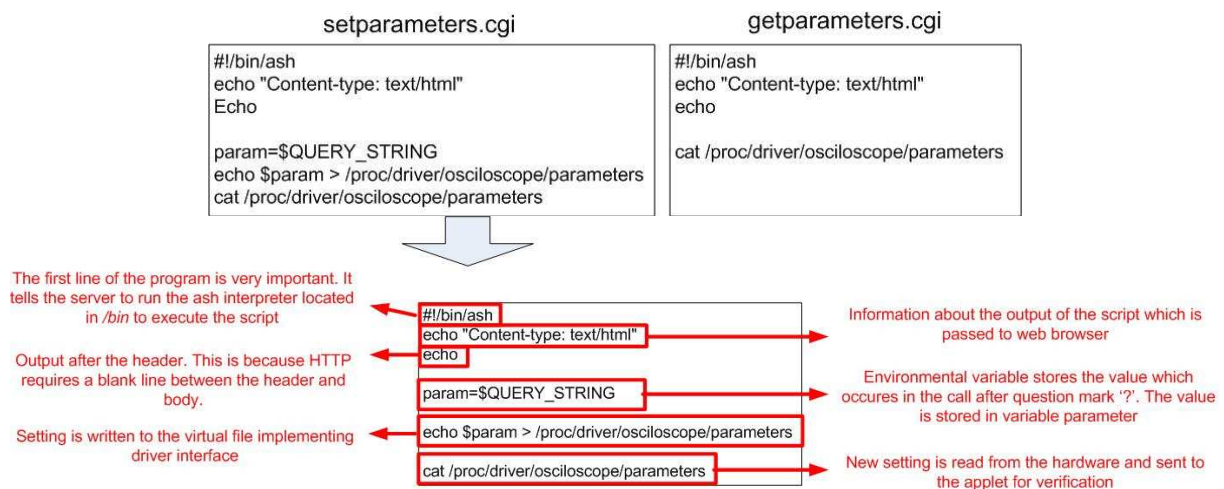


Figure 61 Example CGI scripts with a detailed description [31].

3.6 Web Interface

The Web Interface of Universal Measurement System with Web Interface is based on a simple website which enables to navigate through the utilities provided by UMSWI:

- UMSWI system management and configuration interface (Java Script),
- Oscilloscope and Spectrum Analyzer Graphic User Interface (Java Applet),
- Information about the project,
- Manuals (including example scripts in Matlab).

The layout of the website is meant to be simple and intuitive. The structure of UMSWI’s web page and its layout is presented in **Figure 62**.

Universal Measurement System with Web Interface

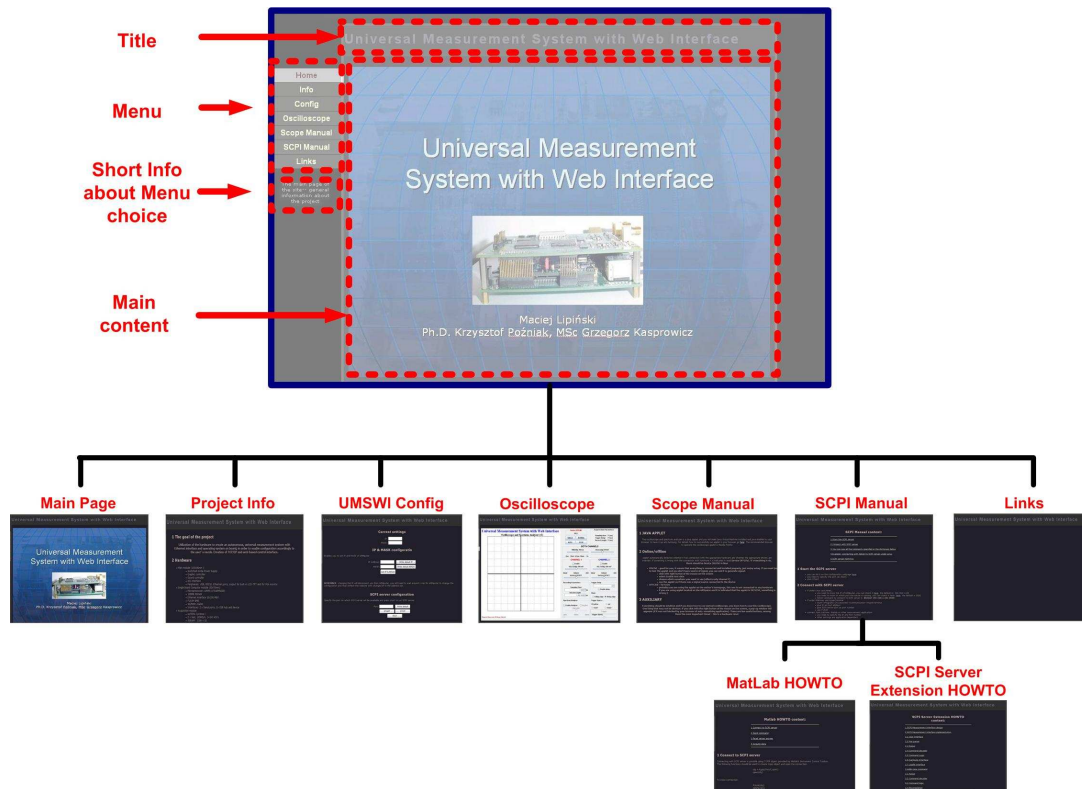


Figure 62 Design of UMSWI web site layout and structure.

The website employs Cascading Style Sheets (CSS) [46] to separate the presentation layer from the document's structure and content. Using CSS makes development of the website easier in terms of presentation consistency and its changes, since visual effects of entire website (all the web pages) can be controlled from one location: *myStyleSheet.css* document. The style sheet document is imported by all the web pages of the website. If the look of the website's pages need changes, modification in *myStyleSheet.css* is propagated throughout the entire server – automatically. CSS makes also development of new pages much easier, since import of the style sheet document by the new web page makes its “look and feel” identical as the rest of the rest of the website. The drawback of the CSS is its inconsistent browser support. Therefore, special attention needs to be paid to test the website in various browsers and, if necessary, implement so-called CSS “hacks” to achieve consistent layout among different browsers. However, this drawback is a minor problem for UMSWI's website because it's layout is not sophisticated.

There is a developer's website of UMSWI on the authors home page [47]. It provide an extended version of the website which is available on UMSWI. The extension include sections which provides HOWTO including detailed description about system development, source codes, binaries and information how to use them.

3.6.1 Oscilloscope and Spectrum Analyzer GUI

Oscilloscope and Spectrum Analyzer GUI is implemented as Java Applet designed according to Model-View-Controller (MVC) paradigm [61]. One of the methods of implementing MVC in Java is the Observer-Observable pattern, which is described in **Appendix A: 3.3**. It was decided that the Observer-Observable pattern would be used in relation between model and view. Thus, the model implements observable and all the views implements observer interface. If a model parameter is updated, all the registered

observers are notified. The following constraints were established for the applet's design, which is presented in **Figure 63**:

- All the parameters representing hardware and view settings are stored in the model (only),
- The main model (*DevModel*) is a holder of oscilloscope (*OscilloscopeModel*) and spectrum analyzer (*AnalyzerModel*) models,
- *DevModel* implements observable, and creates an interface to access *OscilloscopeModel* and *AnalyzerModel* which does not implement observable,
- *DevModel* accesses hardware through *FpgaUtils*,
- *FpgaUtils* class is used to interface hardware and does not store any parameters (unless in offline mode),
- To minimize transfer between client and server, hardware parameters are set only when the oscilloscope is being armed (the acquisition is started),
- Views implement observer interface,
- Different panels of Control Panel are implemented as independent observers (called control widgets)

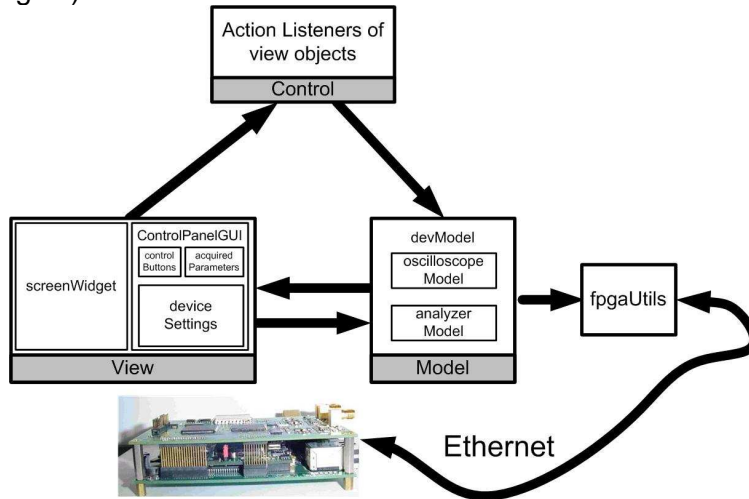


Figure 63 MVC implementation design

3.6.1.1 Model

Figure 64 presents simple UML class diagram of classes which constituted *Model* and classes associated with it.

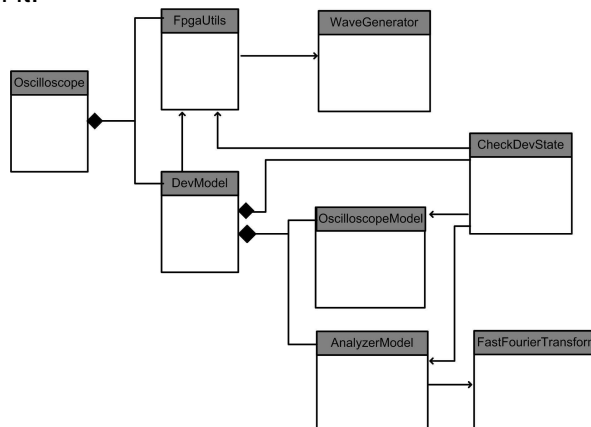


Figure 64 Class diagram of *Model* related classes

Device Model (class *DevModel*)

The *DevModel* class is the main *Model* class which takes the role of a container of the other model classes. Each device (oscilloscope, spectrum analyzer,...) is represented by its own class that implements functions specific to that device. However, *View* & *Controller* components of MVC architecture interface only *DevModel* and have no knowledge of the other classes. It means that all the methods of *OscilloscopeModel* or *AnalzyerModel* which need to be called by *View* or *Controller* need to be represented in *DevModel*. Only *DevModel* implements observable, thus it is responsible for notifying observers about parameters changes. This class manages also communication with hardware (through *FpgaUtils*) and stores hardware parameters as well as the attributes representing device state. In order to eliminate unnecessary communication between the client and the server, changes of hardware parameters made by the user on the Control Panel are not instantly followed by writing new parameters to the device. The hardware parameters are sent to the device, only before the acquisition is started. Once the acquisition is finished, the hardware parameters are read from the device and saved in variables representing hardware setting during the latest data acquisition. Also the raw data from the device is saved in the *Model*. It means that *DevModel* stores two representations of hardware settings:

- Hardware parameters to be written to the device when the acquisition is started, they determine the settings of a new acquisition
- Hardware parameters which were set when the latest acquisition took place.

Such solution solves the problem of using device by multiple clients or from multiple browsers. However, it does not solve the concurrency problem (when two measurements are done from different clients simultaneously). Since the parameters are set just before data acquisition is started, each client can set different parameters and change it independently.

Oscilloscope Model (class *OscilloscopeModel*)

It holds all the parameters representing view settings of the oscilloscope (i.e. time/div, volt/div, x-axis & y-axis start positions of the signal) and implements methods used for calculations connected with oscilloscope display. The *View* is only displaying data and perform no calculation. It is *OscilloscopeModel* class responsibility to provide *View* with positions in which data needs to be displayed on the screen (*screen vectors*). *Model* implements functions which perform the following actions:

- Calculate screen vectors according to current display settings (time/div, volts/div),
- Retrieve current screen vectors,
- Calculate distance between "ticks",
- Set and get values of all the display parameters.

Analyzer Model (class *AnalyzerModel*)

It holds all the parameters representing view settings of the spectrum analyzer (i.e. freq/div, spectrum start position), instantiates class responsible for FFT calculation and implements methods used for calculations connected with spectrum display. The *View* is only displaying data and perform no calculation. It is *AnalyzerModel* class responsibility to provide *View* with positions in which data needs to be displayed on the screen (screen vectors) and scaling factors to print appropriate scales on the display margins. Analyzer model implements methods which perform the following actions:

- Prepare data for FFT calculation, instantiates *FastFourierTransform* object and uses it to calculate FFT on the prepared data and returns the spectrum,
- Look for spectrum maximum value (used for scaling view and scales)
- Calculate spectrum scales,

- Convert spectrum to decibels,
- Calculate screen vectors for spectrum (in volts and decibels),
- Look for maximum spectrum frequency,
- Set and get screen parameters.

Fast Fourier Transform (*class FastFourierTransform*)

The FFT algorithm was not implemented by the author. Open source implementation by Tsan-Kuang Lee from University of Pennsylvania is used [48].

Check Device State (*class CheckDevState*)

This class implements *Runnable* interface which enables its instances to be executed by a thread. The thread is started when acquisition is initiated (pressing : “SINGLE”, “NORMAL” or “AUTO”). The task of CheckDevState depends on the “mode” of acquisition and trigger type:

- **AUTO** mode – it triggers acquisition, checks device state until the data is ready for readout, calls readout function and re-starts the cycle (triggering acquisition),
- **SINGLE** or **NORMAL** modes
 - User-defined trigger – checks whether the “Trigger” button was pressed, once the button has been pressed, it triggers acquisition, checks device state until the data is ready for readout, calls readout function and stops acquisition (and re-starts the cycle),
 - Channel or external trigger – checks device state until the data is ready for readout, calls readout function and stops acquisition (in SINGLE mode) or repeats the cycle (in NORMAL mode).

FPGA Utilities (*class FpgaUtils*)

The class implements communication with hardware through CGI scripts. This class provides two kinds of methods:

- Universal hardware set/get methods which enables to call any CGI script on the server,
- Oscilloscope implementation specific methods which enables to set/get acquisition parameters – they use universal methods in their bodies.

The communication, in universal hardware get/set methods, is implemented using HTTP Tunneling and GET requests described in [49]. They allows to communicate with the server through HTTP socket connection on port 80. This way, the firewalls can be bypassed and server-side programs do not have to return complete HTML documents, instead only data can be returned. The limitations to this method include the fact that the requests responses are received by the applet directly, not the browser and the only server the applet can tunnel to, is the server from which the applet was downloaded. The limitations are acceptable for the methods implementation in UMSWI.

An example *FpgaUtils* method enabling to get data from hardware is presented in **Figure 65**. The method uses `URLConnection` class provided by *java.net*. package. The class contains methods which enable to communicate with URL over the network from the applet.


```

190 private String getHardwareSetting(String what) {
191     if(OFFLINE) { //applet in offline mode
192         return "00";
193     } else { //HTTP Tunneling and GET Requests
194         URLConnection connection = null;
195         String protocol = currentPage.getProtocol(); //gets the protocol name of the URL from which
196                                                     //applet was downloaded
197         String host = currentPage.getHost(); //gets hosts address
198         int port = currentPage.getPort(); //gets port number
199         String scriptName="get" + what + ".cgi"; //creating http request adres
200         String urlSuffix = "/cgi-bin/oscilloscope/" + scriptName; //creating http request adres
201         java.net.URL dataURL = null;
202         try {
203             dataURL = new java.net.URL(protocol, host, port, urlSuffix); //creating URL object referring to applet's host
204         } catch (MalformedURLException e) {
205             e.printStackTrace();
206         }
207         try {
208             connection = dataURL.openConnection(); //Creating URLConnection object
209             connection.setUseCaches(false); //Instructing browser not to cache URL data
210             connection.setRequestProperty("header", "value"); //setting HTTP headers
211         } catch (IOException e) {
212             e.printStackTrace();
213         }
214         BufferedReader in = null;
215         try {
216             in = new BufferedReader(new InputStreamReader
217                 (connection.getInputStream())); //creating an input stream
218         } catch (IOException e) {
219             e.printStackTrace();
220         }
221         String line = new String();
222         String output = new String();
223         try {
224             while ((line=in.readLine())!=null) { //reading data from the server
225                 output=output + line + ":"; //creating methods output string
226             }
227         } catch (IOException e) {
228             e.printStackTrace();
229         }
230         try {
231             in.close(); //closing input stream
232         } catch (IOException e) {
233             e.printStackTrace();
234         }
235         return output.substring(0, output.length()-1); //cutting out last :
236     }

```

Figure 65 Implementation of HTTP Tunneling and GET requests

The method responsible for setting data to the hardware is very similar to the get method. The main difference is the parameter passed in the URL address (**Figure 66**)

```

137         String scriptName="set" + what + ".cgi?" + input;
138         String urlSuffix = "/cgi-bin/oscilloscope/" + scriptName;

```

Figure 66 Forming URL request which sends parameter to the hardware

FpgaUtils enables the applet to be used offline. The applet is offline, if it has no access to CGI scripts. Such situation happens when it is not run from the target machine (i.e. in Eclipse's Applet Viewer). *FpgaUtils* implements function which automatically, during its initialization, checks whether the applet is offline/online. In offline condition, communication with hardware is simulated. All the hardware parameters are written to variables instead of writing them to hardware. Consequently, parameters are read from the variables rather than from hardware. The measurement data, instead of being read from the hardware, is generated by a *WaveGenerator* class. Such solution was designed to make the development easier and faster by enabling running the applet in Applet Viewer or on authors homepage[47].

Wave Generator (class *WaveGenerator*)

Generates sine, cosine, triangle, square and sawtooth waveforms with user-defined parameters (frequency, amplitude, sampling rate, DClevel). Source: [48]

3.6.1.2 View

The applet was initially design to be an oscilloscope only, therefore it attempts to resemble a traditional oscilloscope front panel. The graphic user interface is divided into a screen widget (*OscilloscopeScreen*) and a control panel (*controlPannelGUI*). The screen is meant to present acquired data according to view settings. The control panel enables to change the device state, display device parameters and adjust two kinds of settings:

- Hardware settings – parameters which can be used to control acquisition logic (sampling time, trigger delay, trigger source, record length, trigger level),
- Display settings – parameters which control the way data is displayed and whether it is displayed (Volts/Div, Time/Div, Freq/Div, enable chan1/chan2),

Throughout the applet’s development, the GUI look evolved. The changes were caused by the user feedback and addition of functionalities to the applet, i.e. the spectrum analyzer was added in the final state of applet’s development. The newest version of the applet was (and still is) available on the author’s home page [47]. Since the applet is designed to work “offline”, it could be put on the faculty’s server and tested by users.

The addition of the functionalities was based on users feedback and project requirements (spectrum analyzer). The final GUI design is presented in **Figure 67**. Since some functions are not used during the normal applet usage and due to the space limitations, an auxiliary panel displayed in a separate window was introduced. The Auxiliary Panel is opened upon user’s request by clicking right mouse button on the screen. Auxiliary panel includes:

- Enabling test data and setting the kind of test data,
- Displaying raw data,
- Scaling factor setting.

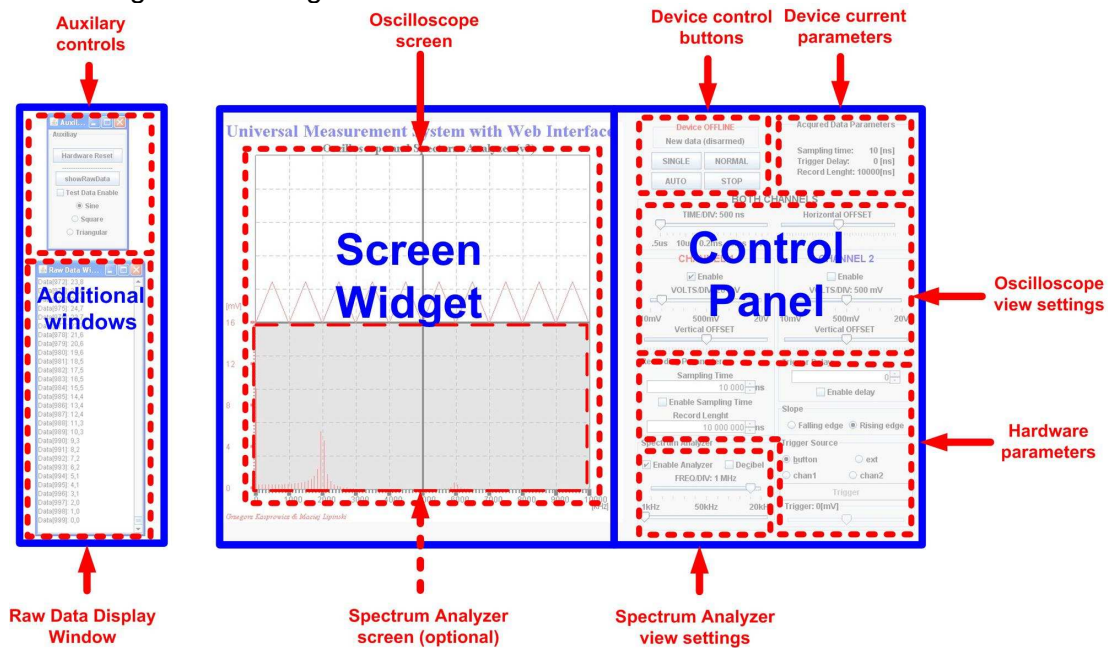


Figure 67 Final GUI design

A detailed UML Class Diagram of View-related classes is presented in **Figure 68**. All the View –related classes implement Observer interface. They register to observable *Model*. View-relate classes hold no data. All the data retrieved from the user is stored in the Model. All the data displayed by View-related classes is retrieved from the Model. Therefore, the View is never out-of-date.

Universal Measurement System with Web Interface

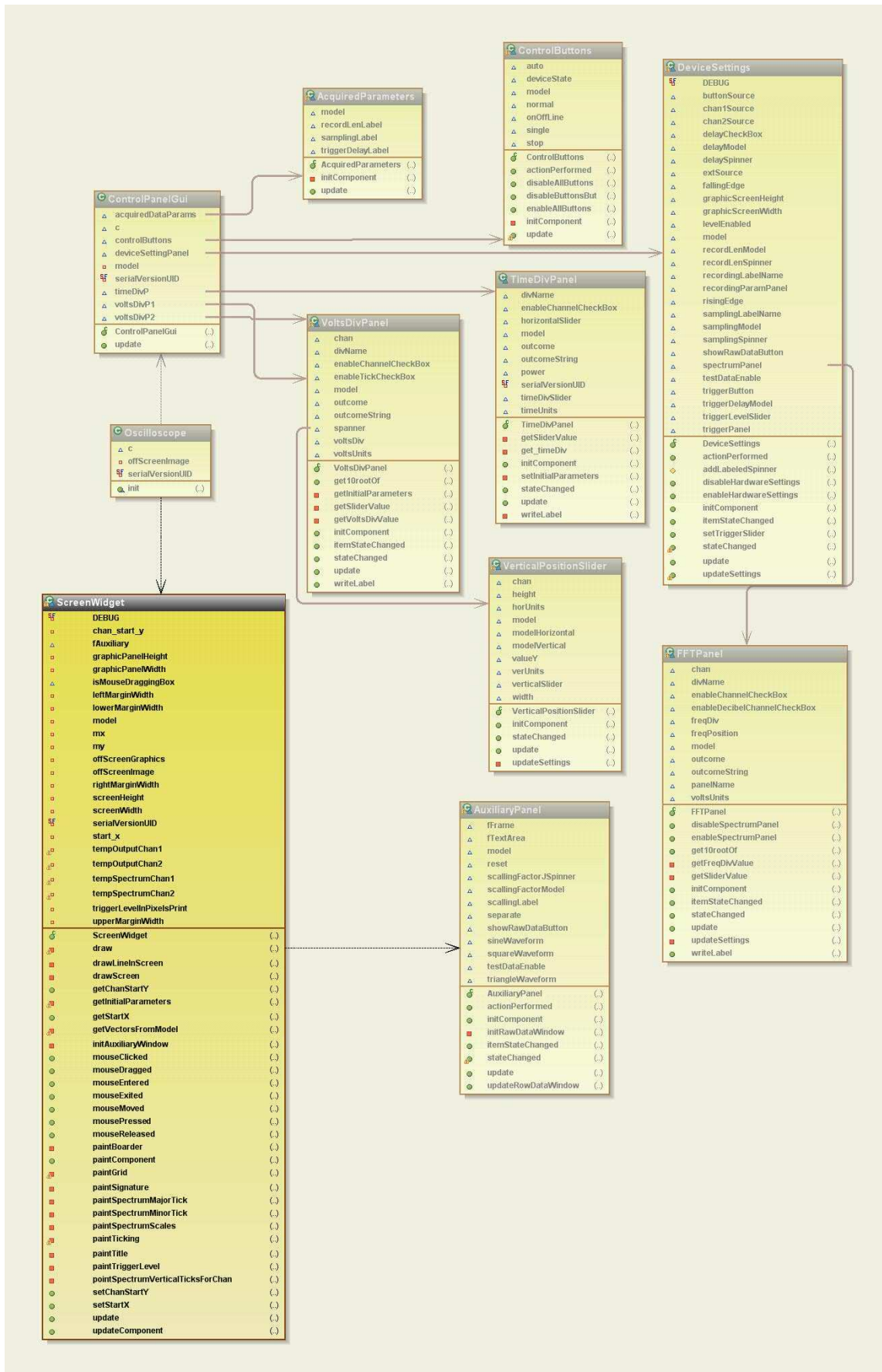


Figure 68 UML Class Diagram of View-related classes

Screen Widget (*class ScreenWidget*)

ScreenWidget is responsible for generation of the applet’s screen image. It uses so-called double-buffering. This means that drawing is done to an *offscreen* image in the first place. When generation of the *offscreen* image is finished, it is drawn on the screen. Such solution reduces screen flickering. The tasks of *ScreenWidget* includes:

- Drawing grid, title and all the other constant components of the screen,
- Drawing measurement data from channel 1 or/and 2 (if enabled) starting from appropriate position (the screen view can be moved by dragging it with a mouse or changing position on control panel),
- Drawing spectrum and its scales,
- Drawing “ticks” (similar to oscilloscope cursors),
- Showing Auxiliary Panel.

Auxiliary Panel (*class AuxiliaryPanel*)

Implements control of auxiliary functions:

- Hardware reset - triggers reset of FPGA logic,
- Scaling factor – data read from the device is multiplied by this value
- Show Raw Data – displays data read from the device (scaled by scaling factor)
- Test Data Enable – it is possible to force offline behaviour of the applet which results in generation of waveforms

Control Panel GUI (*class ControlPanelGUI*)

This is nothing more than a container for widgets implementing control panels, in particular: *AcquisitionParameters*, *ControlButtons*, *DeviceSettings*, *FFTPanel* and *TimeDivPanel*. All the control widgets enable to set device parameters, display settings, or change device state. None of the values are stored in the widgets, a value retrieved from the user is instantly used to update the *Model*.

3.6.1.3 Controller

In Java, controllers are the listeners in Java event structure. Each component that interacts with the user needs to implement some kind of event listener. Such method updates appropriate value in the model. It is important that the neither *View* nor *Controller* stores any data internally. This way the view is never “out of date”, since it displays data retrieved from the model.

Figure 69 presents simplified UML diagram explaining how hardware parameters are set in UMSWI.

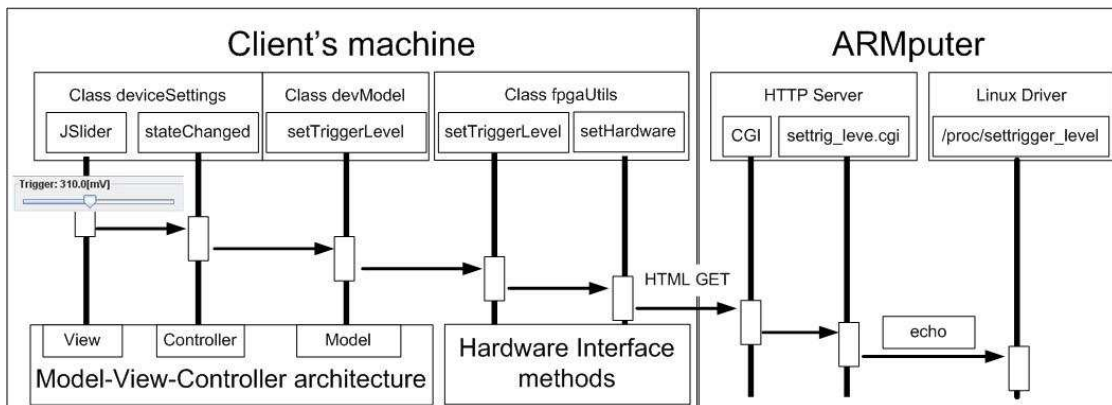


Figure 69 UML Diagram describing applets’ hardware interfacing [50]

3.6.2 UMSWI management and configuration

The management and configuration interface of UMSWI is implemented using HTTP forms, Java Script and Common Gate Interface (CGI). HTTP forms provide buttons and input fields. Java Script functions verify input data and call CGI scripts. CGI scripts perform system calls to change system configuration or start/stop SCPI Server. The web page layout is presented in **Figure 70**.

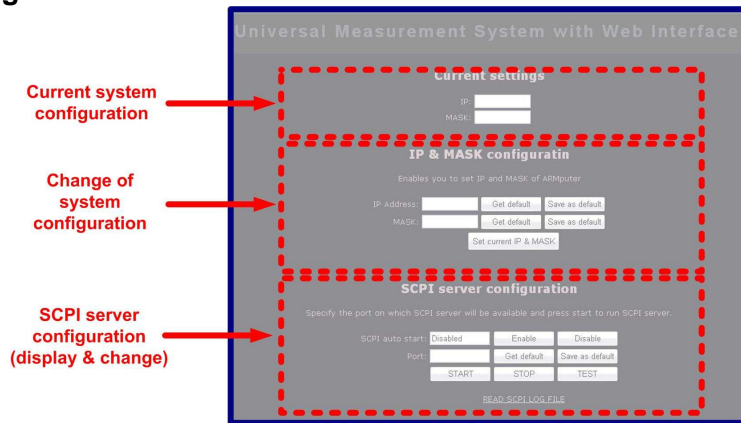


Figure 70 UMSWI configuration and management web page layout

The webpage is divided into three parts:

1. Current system configuration information – it reads current IP address and net Mask when the webpage is loaded. Java Script function *checkIP()* calls *getipaddress.cgi* script which make *ifconfig* system call. The output of the call is interpreted by *checkIP()* to get IP address and net Mask. *checkIP()* is presented in **Figure 71**.

```

377  function checkIP()
378  {
379      //define configuration parameters
380      var IPnumber;
381      var MASKnumber;
382      //create httpRequest
383      var httpRequest = new XMLHttpRequest();
384      try{
385          var result = httpRequest.open('GET', '/cgi-bin/systemConfig/getip.cgi', true);
386      }catch(error){
387          document.getElementById('currentIP').value = "Can't read";
388          document.getElementById('currentMASK').value = "Can't read";
389          return;
390      }
391      //each time the HTTP ready state has changed, this function is called
392      httpRequest.onreadystatechange = function (evt)
393      {
394          //request is complete
395          if (httpRequest.readyState == 4)
396          {
397              //Response text of the request
398              var message = httpRequest.responseText;
399              //get IP out of the text
400              var index = message.indexOf("inet addr:") + 10;
401              IPnumber = message.substring(index, index + 15);
402              var index2 = IPnumber.indexOf(" ");
403              IPnumber = IPnumber.substring(0, index2);
404              //get Mask out of the text
405              index = message.indexOf("Mask:") + 5;
406              MASKnumber = message.substring(index, index + 15);
407              document.getElementById('currentIP').value = IPnumber;
408              document.getElementById('currentMASK').value = MASKnumber;
409          }
410      };
411      //send the request
412      httpRequest.send(null);
413      return IPnumber;
414  }

```

Figure 71 Example Java Script script using CGI

2. Change of system settings – enables to get default, set current and store in memory as default IP and Mask.
 - i. **Get default** – reads IP/Mask value from *default_ip/default_mask* file stored in */usb/ARMScope/data* folder. Default IP/Mask is set on the start-up of the device by *set_ip* script.
 - ii. **Save as default** - it gets the value of IP/Mask from the form input field, verifies the input data correctness (IP/Mask has special format) and saves the IP/Mask inputted in form field in *default_ip/default_mask* file. Default IP/Mask is set on the start-up of the device by *set_ip* script.
 - iii. **Set current IP & Mask** - it gets the value of IP and Mask from the form input fields, verifies the input data correctness (IP and Mask have special format) and sets the IP and Mask calling *setnewip.cgi* script. The script uses *ifconfig* system call to set the new system configuration
3. SCPI server configuration – it enables to set the system to start SCPI automatically on device start-up, it is also possible to start/stop the server, get default and store in memory port number.
 - i. **Enable/disable SCPI auto start** – it modifies the *default_scpi_autostart* file stored in */usb/ARMScope/data* folder. If auto start is enabled, the file is written with “Enable”, otherwise it holds “Disable”. *start_scpi* script, which is called during system start-up, reads *default_scpi_autostart* file and starts SCPI server if “Enable” is read, otherwise SCPI Server is not started,
 - ii. **Get default** – calls the *getdefaultport.cgi* script which reads the *default_port* file from */usr/ARMScope/data* folder,
 - iii. **Save as default** – it gets the value of Port from input field, verifies the input data (Port has special format) and saves the Port number in *default_port* file. Default Port number is used by the *start_scpi* script on system start-up to run SCPI Server, provided the automatic SCPI server start is enabled,
 - iv. **START** - it gets the value of Port from input field, verifies the input data (Port has special format) and calls *startscpiserver.cgi* which starts the SCPI Server on the provided port,
 - v. **STOP** – calls *stopscpiserver.cgi* which stops the server process,
 - vi. **TEST** – calls *testscpiserver.cgi* which calls *ps* system command and looks for SCPI Server process,
 - vii. **READ SCPI LOG FILE** – calls *getscpilofile.cgi* script which opens the *log_file* located in */usr/ARMScope/data*. SCPI Server writes to *log_file* detailed information about its performance, especially errors.

3.7 Measurement Interface

Measurement Interface is implemented as a SCPI Socket Server with commands interpreter and hardware interface using C language. An information about SCPI standard and its syntax can be found in **Appendix A: 3.4**. **Figure 72** presents example SCPI message and its elements. **Figure 73** presents design of SCPI Measurement Interface. It is a small application which takes as an input argument the kind of user interface (server or local). If the application is started as server, the second argument needs to be provided, the argument is the number of port on which server is listening. Details of program implementation of each of the application’s components are described in the following subchapters.

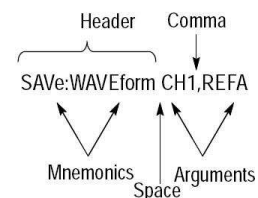


Figure 72 SCPI command message elements

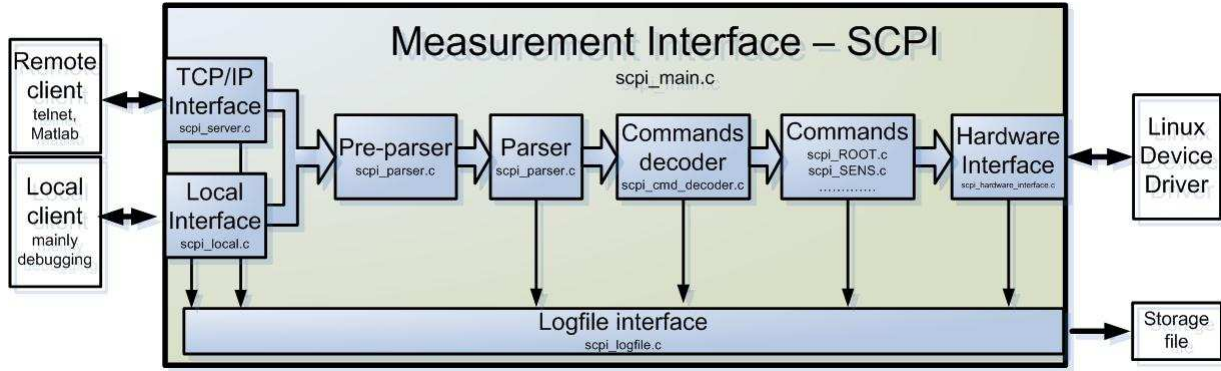


Figure 73 SCPI Server design

3.7.1 User interface

The main user interface of the SCPI Measurement Interface application is a socket server. It implements internet stream sockets which are characterized by IP Address and port number. Stream sockets use TCP to provide reliable two-way connected communication. A local interface was needed during development. It is a simple command line interface.

3.7.2 Pre-parser

Pre-parser is a single function (*pre_parse_cmd()*) which receives data from the user as a single string of characters. It extracts separate commands (command messages) by looking for semicolons. The outcome of this function is a dynamically allocated linked list of separated commands.

3.7.3 Parser

For each element of the linked list returned by pre-parser, parser function (*parse_cmd()*) is run. It extracts and recognizes elements (mnemonics, argument) which compose the command. Mnemonics are checked against a list of known mnemonics. If the extracted mnemonic is not found on the list, an error is returned.

As SCPI syntax allows full names and abbreviation of the mnemonics and determines that the parser is not case sensitive, the following approach was taken. For each mnemonic, an abbreviation and full name needs to be placed in the list of available shortcuts. The names are divided into several lists according to the abbreviation length. The extracted mnemonic is converted into uppercases and checked against the lists (starting with the list with the longest shortcuts). If the mnemonic is found, it is added to the head of a linked list associated with the command which is being parsed. If it is not found on the list, an error is returned.

Consecutive commands do not have to start each time from the root (":"). It means that, if a command is executed (i.e. :SENS:SWE:TIME 1) and the consecutive command has the same path (i.e. :SENS:SWE:POIN 100), SCPI standard says that it is enough to input the last mnemonic (i.e. POINT 100) instead of the full path. This is why the parser, before starting to extract mnemonics, checks whether the full path is provided (starting with ":").

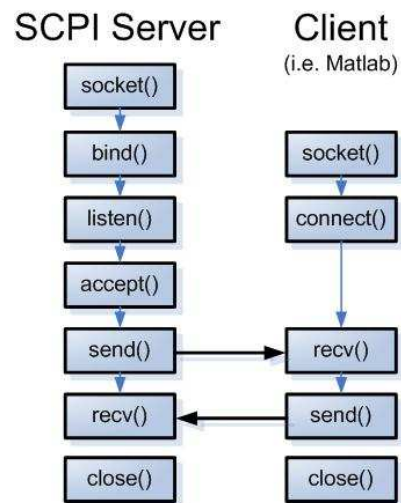


Figure 74 Communication layers

If colon is not detected at the beginning of the command, linked list from the previously performed parsing is taken deleting only the last element (head) of the list. The outcome of the parser function is a linked list of consecutive mnemonics which compose the header and command's argument.

3.7.4 Commands decoder

A dictionary of available commands was translated into a data tree (Figure 76). Each node of the tree is associated with a mnemonic and is represented by a structure that holds (Figure 75):

- mnemonic's name,
- list of pointers to child-nodes,
- pointer to a function associated with the node.

```

137 struct cmd_struct{
138     char      *name;
139     struct cmd_struct *cmds[30];
140     int  (*func)(char*);
141 };
    
```

Figure 75 Command structure

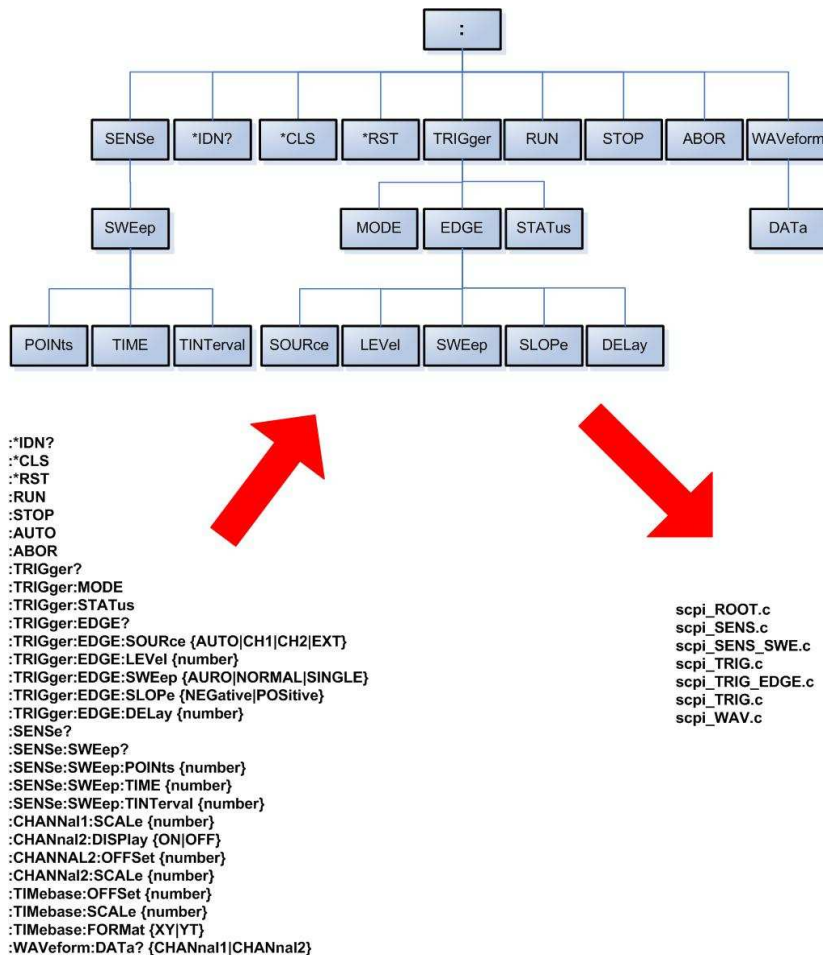


Figure 76 C implementation of SCPI dictionary

Based on the data tree, a set of C-files defining nodes' relations and command functions were created. A C-file representing the parent node (i.e. TRIGGER) defines its connections with children nodes (i.e. MODE, SENSE, STATus) and functions associated with children nodes (Figure 77).


```

1  /*! \file scpi_TRIG.c
2      \brief TRIG implementation.
3  */
4  #include "scpi_main.h"
5
6  int init_TRIG_cmds()
7  {
8
9      //////////// TRIG ////////////
10     TRIG.cmds[0] = &TRIG_MODE;
11     TRIG.cmds[1] = &TRIG_STAT;
12     //-----
13     TRIG.cmds[2] = &TRIG_SLOPE;
14     TRIG.cmds[3] = &TRIG_EDGE;
15
16     TRIG.name="TRIG";
17     TRIG.func=&fun_TRIG;
18
19     TRIG_MODE.name="MODE";
20     TRIG_MODE.func=&fun_TRIG_MODE;
21
22     TRIG_STAT.name="STAT";
23     TRIG_STAT.func=&fun_TRIG_STAT;
24 }
    
```

Figure 77 Defining nodes relations and function associations

To make development and further extensions easier, the following file naming convention has been established:

```

scpi_NODE1_NODE2_...._NODEx.c
scpi_NODE1_NODE2_...._NODEx.h
    
```

3.7.5 Command logic

Command logic is implemented for each node in the function associated with the node. A pointer to this function is held in the data tree. SCPI Standard requires all the commands (except: `:*CLS`, `:*RST`, `:RUN`, `:STOP`, `:AUTO`) to answer a query. Query is defined as a header with question mark “?” at the end (argument). For the end nodes (nodes without children), a query returns value of the setting associated with the node. For the middle nodes (nodes with children), a query returns settings associated with all the children nodes. To make the implementation of command logic easier and faster, a special function has been defined which takes as an input a list of possible arguments (i.e. `?, AUTO, CH1, CH2, EXT`). It recognizes the argument and returns its index in the list. In principle, a function which implements a command logic has structure presented in **Figure 78**.

```

4  int fun_NODE1_NAME(char *input_value)
5  {
6      if( is_not_empty(input_value) )      //check if input value was provided
7      {
8          int option_number;
9
10         //define possible input_value options
11         char possible_options[][2][10] = {
12             {"XXX", "XXXend"},
13             {"YYY", "YYYend"},
14             {"?", "?"};
15
16         //check which options has been provided
17         option_number = get_option(input_value, possible_options, 3);
18
19         if(option_number == 0)
20         {
21             //execute option 0 logic
22         }
23         else if(option_number == 1)
24         {
25             //execute option 0 logic
26         }
27         else if(option_number == 2) //query
28         {
29             //execute query
30         }
31         else
32         {
33             //error, wrong input_value
34         }
35     }
36     else
37     {
38         //error, input value is empty
39     }
40 }
    
```

Figure 78 Template of function implementing command's logic

3.7.6 Hardware interface

Hardware interface uses Linux Device Driver to control FPGA. The driver is interfaced by reading and writing appropriate files in */proc/driver/fpga* directory.

3.7.7 Logfile interface

Each time SCPI Interface application is opened, a new logfile is created. All the messages concerning application performance are written to the logfile. A special function (*print_to_logfile*) is defined to make the process simple and efficient. Studying a *logfile*, the entire process of command parsing, decoding and execution can be followed step-by-step. It makes much easier finding errors SCPI commands send by the user (**Figure 79**).

```

2  SCPI_PARSER & INTERPRETER LOG FILE
3  -----
4  -----
5  ----- NEW REQUEST FROM CLIENT -----
6  -----
7  pre_parsing result:
8  :sens:swe:tint 100
9  :sens:swe:poi 10
10
11 -----
12 parsing result:
13 SENS
14 SWE
15 TINT
16 argument value: 100
17 ~~~~~
18
19 Executing :SENS:SWE:TINT
20
21 SYNTAX_ERROR(102): Failed to parse the input data, probably incorrect command
22 Error in: ->POI<-
23

```

Figure 79 Example SCPI log file

3.7.8 Extensibility

New commands (nodes) can be added to the SCPI Measurement Interface. In order to do that, the following actions needs to be taken:

1. PARSER - entry needs to be added to the list of recognized mnemonics and its abbreviations in the *scpi_parser_data.h*
2. COMMAND DECODER –a node in the data tree needs to be added by:
 - a. Declaring new node (appropriate header, depending on the node’s location in data tree),
 - b. Creating a pointer in the parent node (appropriate C-file, depending on the node’s location in data tree)
 - c. Creating a pointer to the function implementing command's logic (appropriate C-file, depending on the node’s location in data tree),
3. COMMAND LOGIC – an appropriate action associated with the new command needs to be implemented (appropriate C-file, depending on the node’s location in data tree)
4. recompilation.

A detailed instruction describing how to extend SCPI Measurement Interface with is included in SPCI Manual available on the UMSWI website

Figure 80 summarizes parsing and decoding process for “:sens:swe:poi 10 ; :trig:edge:sour auto ; :run ” input string.

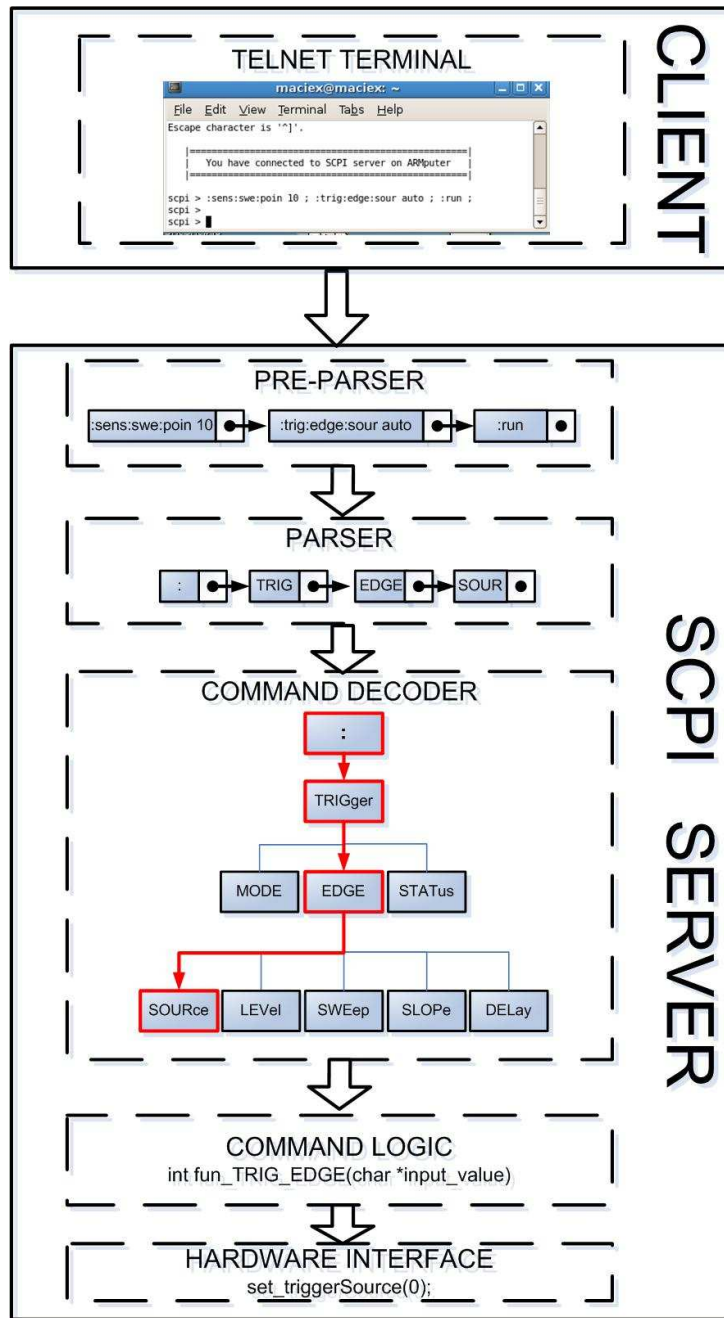


Figure 80 Explanation of parsing and decoding process

4. Testing

4.1 Development test

Testing was performed for each of the system's components separately. Once a part of the system was proven to work correctly, it was added to another correctly working component, to eventually, create a working system. Such management made the development easier and more efficient. The order of development and tests was following:

1. Development and tests of Embedded Linux,
2. Development and simulation of FPGA logic,
3. Development and tests of Linux Device Driver without interfacing hardware,
4. Development and tests of Java Applet without interfacing Linux Device Driver,
5. Tests of Java Applet which interfaces Linux Device Driver without interfacing hardware (FPGA logic),
6. Development and tests of SCPI Server without interfacing Linux Device Driver,
7. Tests of SCPI Server and Linux Device Driver without interfacing hardware (FPGA logic),
8. Tests of FPGA logic and Linux Device Driver (interfacing hardware),
9. Tests of Java Applet interfacing Linux Device Driver interfacing hardware (FPGA logic),
10. Tests of SCPI Server interfacing Linux Device Driver interfacing hardware (FPGA logic),
11. Development and tests of UMSWI management and configuration interface.

During the development, when a test analogue input signal was required, two sources of signal were used:

- Stabilized power supply for constant input,
- Music card output, waveforms generated with *Cool Edit 2000*, waveform generator.

4.1.1 Embedded Linux Operating System

Tests of Embedded Linux were conducted by checking whether the required by project utilities and peripherals work correctly:

- Ethernet,
- MMC/SD card ,
- USB (optional),
- httpd (web server).

4.1.2 Linux Device Driver

The driver was initially tested without interfacing hardware. The driver's architecture separates hardware interface from the driver's logic and the actual communication between the driver and hardware is limited to reading and writing registers at appropriate address. Therefore, it was possible to test thoroughly driver's logic by substituting the operation of reading/writing hardware by reading/writing variables and outputting information about the operation to the terminal/log file. Once logic was proven to work correctly, the hardware interface was tested by examining Static Memory Controller (SMC) control signals on the microprocessor's pins. Further testes of the driver were done along with Java Applet, SCPI Server and FPGA tests.

4.1.3 FPGA debugging

FPGA logic was firstly tested using Quartus II Simulator tool which enables functional and timing simulation. Only simulation was done at the beginning, since the author was not provided with recorder module.

When the hardware was available, the logic was tested during its operation (in real time) using Signal Tap II tool provided by Quartus II. The SignalTap II Embedded Logic Analyzer [51] enables to debug an FPGA design. It does not require changes to the design or external probes in order to capture the internal nodes' or I/O pins. The device memory is used to store the captured data.

FPGA testing was started with FPGA-ARM Communication Logic. It was needed to work correctly before starting tests of Acquisition Management Logic which is controlled from ARM. SignalTap II enabled to see the signals coming from ARM microprocessor, therefore it was possible to determine whether the hardware part of the driver was working correctly. Thanks to the Signal Tap, it was also possible to establish the right SMC parameters used in FPGA-ARM communication and described in **3.3.1 Communication logic**. During the tests of FPGA-ARM communication, a multimeter turned out to be useful as well. Some of the problems encountered during attempts of communication were caused by minor faults in the hardware of UMSWI. In particular, unconnected pins of address and data bus. After this experience, to avoid tedious debugging of the FGPA logic done in vain, the hardware was always tested first. Therefore the control logic of acquisition process was extended to allow tests of SSRAM and the address (data)bases. The following tests were performed to proof SSRAM reliability:

- Instead of storing in SSRAM data read from ADC, data was generated FGPA was written to SSRAM and than read, two kinds of data were generated:
 - Data equal to the address of writing,
 - 0xAAAA and 0x5555 in subsequent addresses,
- Electrical values of the pins were measured - two missing connections were detected.

SignalTap was also used to debug and improve the acquisition control logic, mainly the trigger and delay timing to make sure it is correct.

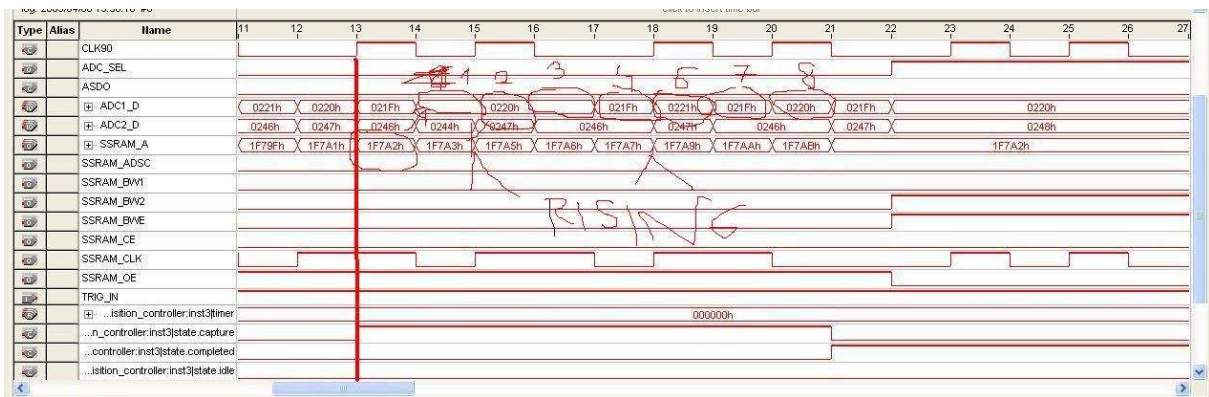


Figure 81 Debugging FPGA

Final tests of FPGA logic and Linux Device Driver where done applying a signal (from known source) and performing entire measurement using /proc file system interface of fpga driver. The acquired data begin compared with the source (reference).

4.1.4 Applet tests

The applet was developed and tested along with CGI scripts. In the first place, they were tested without interfacing Linux Device Driver. The scripts were configured to read and write ordinary files (including the readout data). Once the applet's CGI interface and CGI scripts were proved to work correctly, an OFFLINE mode was introduced in the applet. In OFFLINE mode the data is prevented from being sent with CGI scripts to server (on the lowest possible level of applets' architecture). The parameters are written to variables and read from variables, the measurement data is generated. This enabled easy development of the applet in Eclipse Development Environment. Before being tested as a part of entire system, the applet was run with the driver which does not interface hardware. Once interfacing hardware, the applet with all its functionalities could be tested. It also allowed further tests of other system components, .i.e FPGA logic (Figure 82 presents trigger tests).

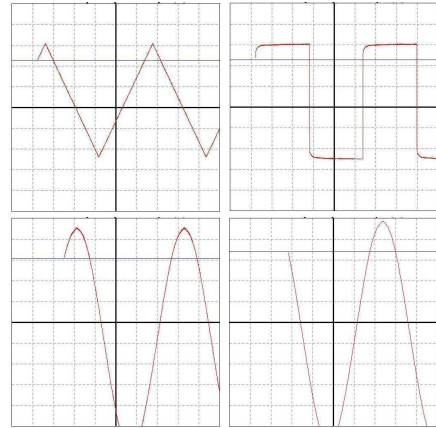


Figure 82 Trigger tests

4.1.5 SCPI server tests

Tests of SCPI server were done in the similar way as the tests of applet. In the first place, information about the hardware operations to-be-made was printed, secondly, a “fake driver” (which does not interface hardware) was used, lastly SCPI server was connected to the hardware. For the testing purpose a local interface for SCPI Server was developed. It was done to be able to exclude the possibility that error is caused due to wrong implementation of socket protocol. Finally, tests of socket server and all other components, using telnet application as client, were conducted.

SCPI Server was tested with Matlab application[52]. Matlab uses TCPIP object to connect with remote instruments via TCPIP protocol. “fwrite()” function is used to send messages and “fread()” function is used to receive responses. Special m-files were created to simplify communication between Matlab and UMSWI.

- SCPIopen()* – opens connection with UMSWI,
- SCPIidentify(t)* – identifies device,
- SCPImeasureTest(t)* – performs example measurement
- SCPImeasureAUTO(t, sample_number, chan)* – performs measurement allowing to choose sample time and channe,
- SCPIclose(t)* – closes connection.

Figure 83 presents measurement of the same waveform using Java Applet and Matlab.

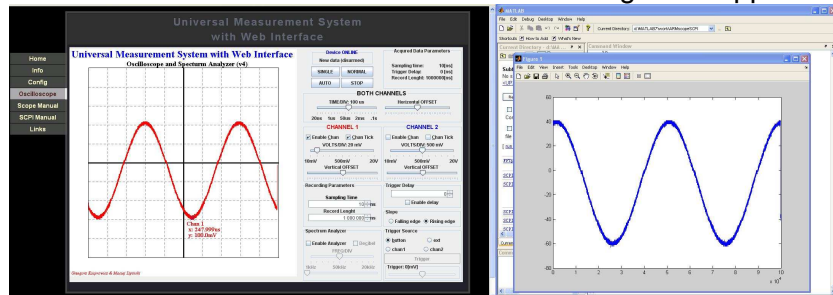


Figure 83 Matlab test of SCPI Server

4.2 Final tests

Tests were conducted in ELPHA/PERG laboratory. The aims of final tests included:

- verification of measurement accuracy,
- specification of UMSWI's parameters and features,
- observation of system's behavior in boundary and beyond-boundary conditions.

4.2.1 Test set-up

The parameters of devices needed to conduct tests were determined by UMSWI's theoretical parameters and practical methods of its verification.

Frequency

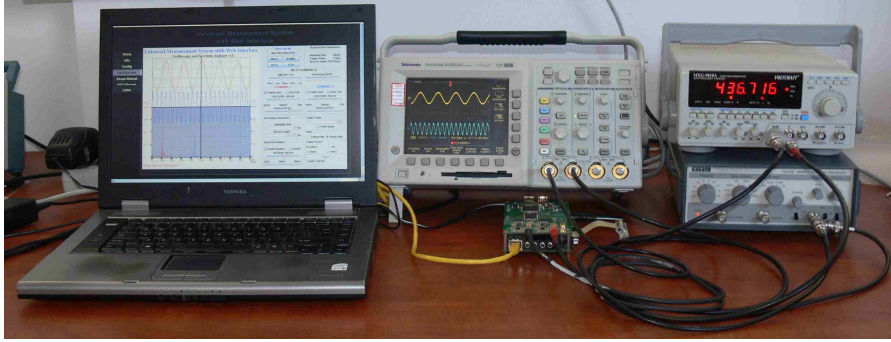
Since the sampling rate of UMSWI's Analogue-to-Digital Converters is 100MHz, the highest frequency of an analogue signal which, theoretically, can be reconstructed from samples (according to Nyquist-Shannon sampling theorem) is 50MHz. Therefore, theoretically, the range of digital oscilloscope used for reference measurement should be at least 50MHz and the generator should produce signals in the range 50MHz-0Hz. However, in practice, the frequency value for which an oscilloscope is considered accurate is significantly smaller than theoretical value and is called "frequency range" (**Appendix: 4**). It is indicated by the frequency at which measured signal is attenuated by 3dB. Since preliminary tests showed that frequency range falls between 15-20MHz, 20MHz functional generation was considered sufficient. On the other hand, to receive quality reference measurement, it is recommended that the reference measurement device is significantly more precise than the device under test (DUT). Therefore the reference oscilloscope frequency range should be 100MHz or more.

Voltage

The resolution of UMSWI's ADCs is $1V/1024bits \approx 1mV$, therefore the vertical sensitivity of 1mV/div should be sufficient to verify the amplitude of the smallest signal that could be detected by UMSWI. Devices which were used to conduct test measurements are listed in **Table 12**. The test setup-up is presented in **Figure 84**.

Name	Model	Parameters	Function
Digital Phosphor Oscilloscope	Tektronix TDS 3054B	Range: 500MHz Sampling: 5GS/s Vertical sensitivity: 1mV/div	Provided reference measurement
Function generator 1	TG120 20MHz	Max freq: 20MHz	Input signal
Function generator 2	MXG-9810A	Max freq: 7MHz	Input signal

Table 12 Devices used during tests


Figure 84 Test set-up

The measurements were taken on the reference oscilloscope using “measure” function. On ARMScope, measuring was performed using “ticks” to receive time of one period and signal’s amplitude. It can be assumed that readout error of a period (or voltage) on AMRscope is approximately one pixel. The screen is 500-pixel wide (and high), and was always fitted to show less than 2 periods (or less than 2 amplitudes). Therefore, the readout error can be estimated as:

$$error_{1x} \leq error \leq error_{2x} \quad (\text{Eq. 1})$$

where

$$error_{1x} = \frac{1px * 0.1 * x \frac{time_or_mV}{div}}{500px * 0.1 * x \frac{time_or_mV}{div}} = \frac{1}{500} = 0.2\% \quad (\text{Eq. 2})$$

is the error when period (or amplitude) is equal to screen width (height), and

$$error_{2x} = \frac{1px * 0.1 * x \frac{time_or_mV}{div}}{250px * 0.1 * x \frac{time_or_mV}{div}} = \frac{1}{250} = 0.4\% \quad (\text{Eq. 3})$$

is an error when two periods (or amplitudes) are equal to screen width (or height). Therefore:

$$0.2 \leq error \leq 0.4$$

All the Matlab scripts used to present measurement results are included in the attached CD.

4.2.2 Vertical axis measurements

Initial measurements of amplitude accuracy were taken within moderate frequency and voltage range to avoid errors of low- and high-frequencies. Two measurement series were taken. First measurement included constant frequency and various amplitude values (from 100mV to 1V with 100mV intervals). Second measurement included constant amplitude value and various frequencies (from 1Hz to 1MHz with logarithmic increment). Results of the initial measurements of amplitude accuracy are presented in **Figure 85** in charts 1 & 2. Charts 5 & 6 present the measurement error. It is quite apparent that the error is constant. Therefore, it was decided to introduce scaling factor. A simulated effect of scaling factor on the amplitude accuracy is presented in charts 3 & 4, and the decreased accuracy error can be seen on charts 5 & 6. The scaling factor was calculated as an average

of ratios between reference amplitudes and measured amplitudes (scaling_factor = 1.0946). Green lines in charts 3, 4, 5 & 6 show results of the amplitude accuracy measurements conducted with scaling factor applied. The results prove that introducing scaling factor was a good decision, the relative error (in per cent) dropped from 8.62% to 0.69% which is close to measurement readout error. The standard deviation of the amplitude is small and equals 0.44 .

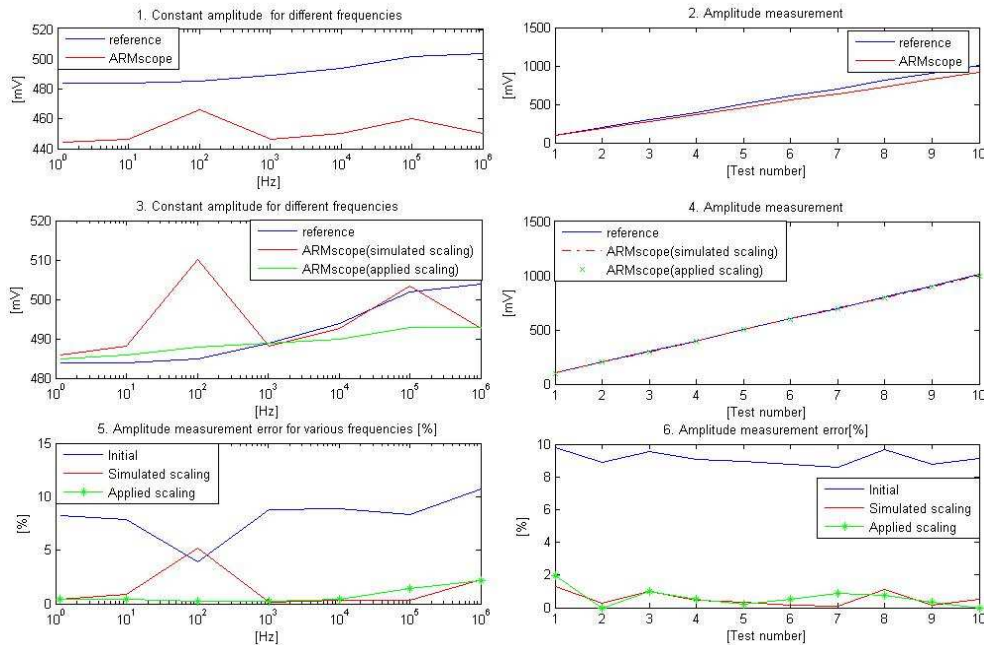


Figure 85 First amplitude accuracy test (*final_test_1.m*)

Once it was proven that amplitude accuracy is stable for reasonable frequencies, a measurement was conducted to verify the range of amplitude accuracy. The measurement focused on high frequencies. The results are presented in **Figure 86**. As described in **Appendix A: 4**, -3dB attenuation determines the frequency range of a device. The results show that the actual frequency range is approximately 12MHz. The attenuation is flat until 1MHz and almost drops below -3dB for 10MHz. Therefore, it seemed reasonable to state that the frequency range of ARMscope is 10MHz, while the actual frequency range is slightly higher and reaches 12MHz. The results clearly show that there is no point in conducting measurements for frequencies higher than 20MHz.

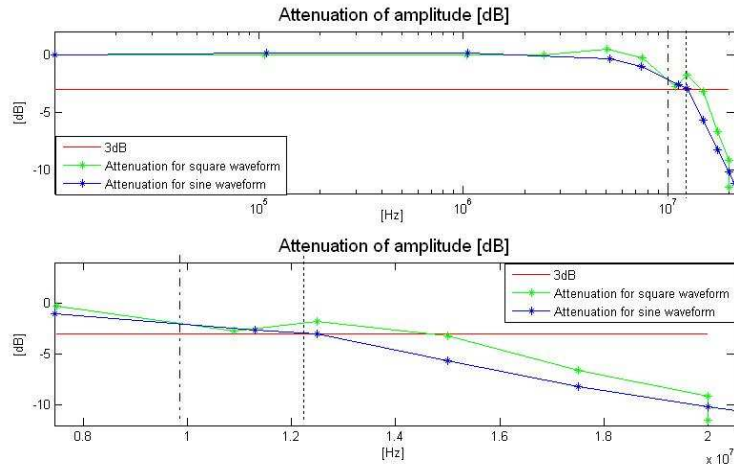


Figure 86 Amplitude attenuation for high frequencies (*final_test_2.m*)

Figure 87 presents attenuation of various amplitude values for frequency 10MHz and the relative error (in per cent) of amplitude accuracy at such frequency. The attenuation does not go beyond -3 dB which means that the proposed frequency range of 10MHz seems to be the good choice.

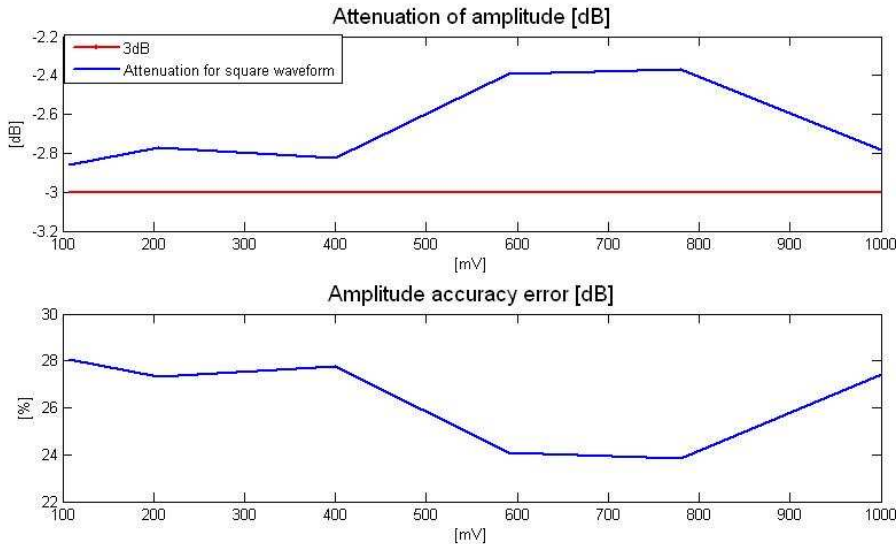


Figure 87 Amplitude attenuation at 10Mhz for various amplitude values (*final_test_3.m*)

Figure 88 presents offset error for 10KHz square signal. Offset accuracy indicates how well the device handles low-frequency issues. The average error of 2.62% shows that this device is not perfect for low frequencies.

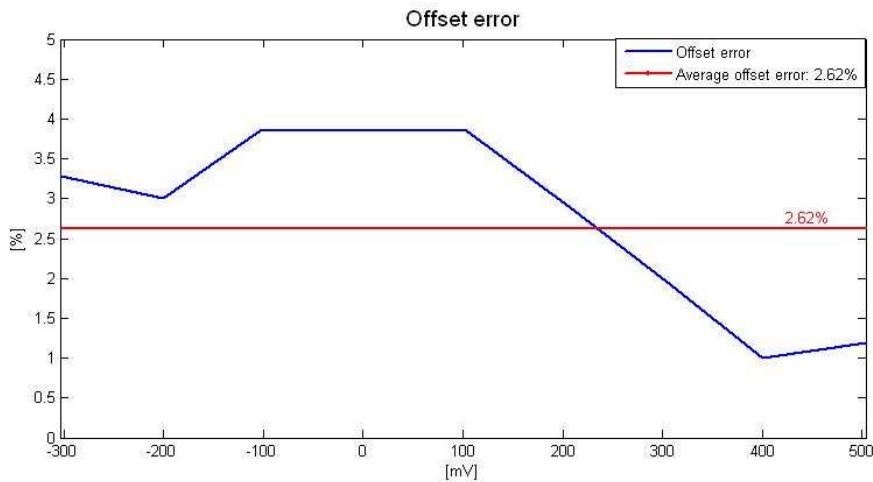


Figure 88 Offset error

Although the resolution of ADCs is ~1mV (1V/1025 bits), due to the noise, the minimal amplitude which can be detected and measured by ARMScope was observed to be 5mV. **Figure 89** and **Figure 90** present example test screen shots.

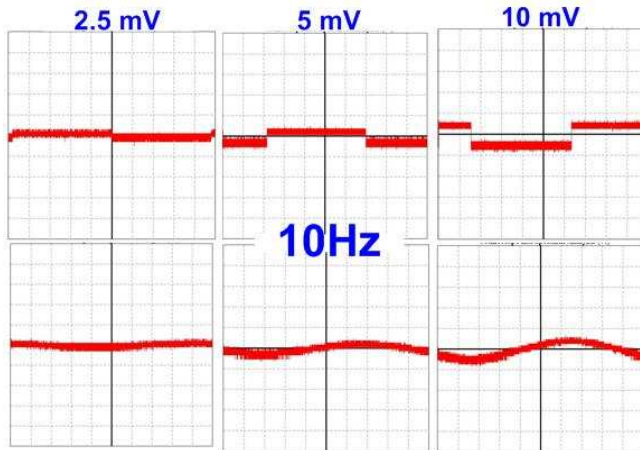


Figure 89 Minimal input voltage test at 10 Hz

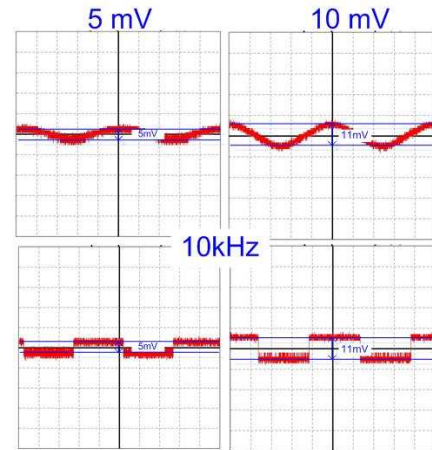


Figure 90 Minimal input voltage test at 10 kHz

4.2.3 Horizontal axis measurements

Relative error of signal frequency and period are presented in **Figure 91**. The measurements show that frequency error stable in the frequency range:100Hz-10MHz. The error is on the level of measurement error: 0.2% - 0.4%. This is a big error if compared with data sheets of commercial digital oscilloscopes. However the error is determined by the readout error and the error of reference measurement. It is very probable that the actual frequency error is much lower.

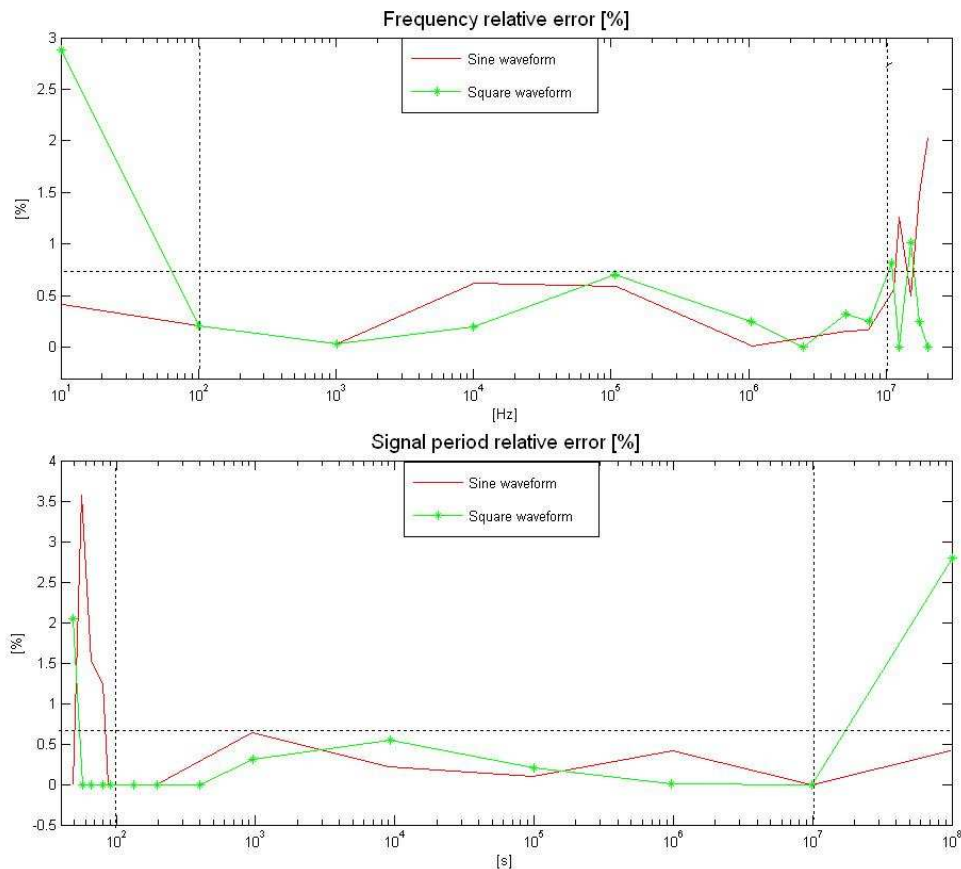


Figure 91 Signal frequency and period relative error(*final_test_4.m*)

The results prove the upper limit of UMSWI's accurate measurement (established in **4.2.2**) and sets limit for low frequency measurement to 100Hz. However, the low frequency

limitation can be questionable, since the instability of reference oscilloscope for 10Hz measurement is approximately 1-2%.

Figure 92 presents measurement of rising time. Rising time is described in **Appendix A: 4.**

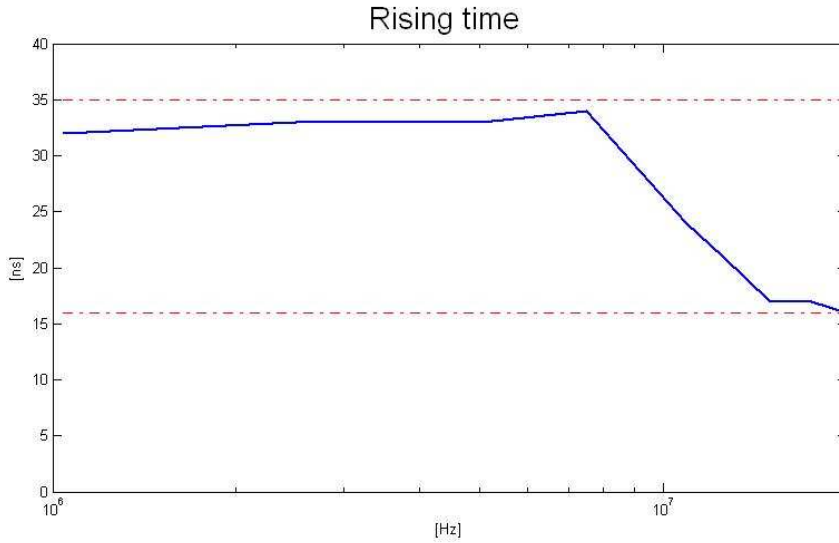


Figure 92 Rising time measurement (*final_test_5.m*)

4.2.4 Frequency domain

Performance of UMSWI spectrum analyzer in terms of frequency measurement was tested by reading frequency of the main harmonic displayed by the UMSWI’s spectrum analyzer and frequency measurement from reference oscilloscope. Since the UMSWI spectrum analyzer is not suitable for accurate reading of frequency, this test was only to prove rough accuracy of the FFT algorithm and scale display. **Table 13** and **Figure 93** present measurement results. It is clear that FFT algorithm works correctly in terms of frequency.

Sine waveform	
Reference frequency [Hz]	Spectrum analyzer reading [Hz]
10100	10000
109000	110000
1055000	1050000
5200000	5200000
7450000	7500000
11300000	10125000
12500000	12500000
15000000	15000000
17500000	17500000
20000000	20500000
21200000	21000000

Square waveform	
Reference frequency [Hz]	Spectrum analyzer reading [Hz]
10020	10000
107000	107000
1040000	1040000
2500000	2500000
5060000	5100000
7500000	7500000
10900000	11000000
12500000	12500000
15000000	15000000
17500000	17500000
20000000	20000000

Table 13 Test of Spectrum analyzer

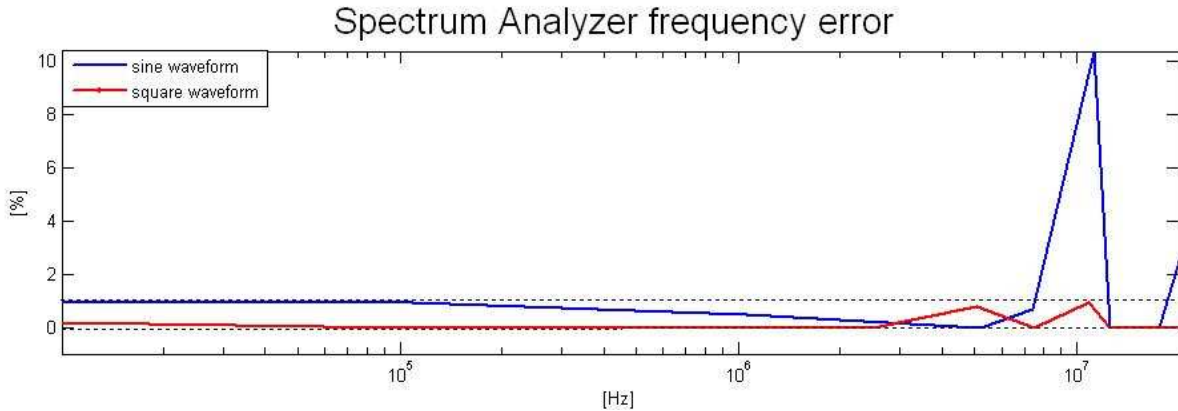


Figure 93 Spectrum analyzer test (*final_test_6.m*)

The performance of UMSWI's spectrum analyzer in terms spectrum's amplitude value (in mV and dB) was tested with the help of Matlab and using ability to perform measurement with UMSWI from Matlab. To connect from Matlab to UMSWI SCPI Server and perform measurements, scripts provided on UMSWI website were used. The measurement connection with UMSWI was started with *SCPIopen.m*. Another script (*SCPImeasure.m*) was used to retrieve data with appropriate parameters. Matlab connection with SCPI Server is closed using another script: *SCPIClose.m*.

Spectrum analysis of the same signal were done using UMSWI Java Applet (**Figure 95**) and Matlab scripts (**Figure 94**), the results compared. This analysis proved that SCPI Server works correctly.

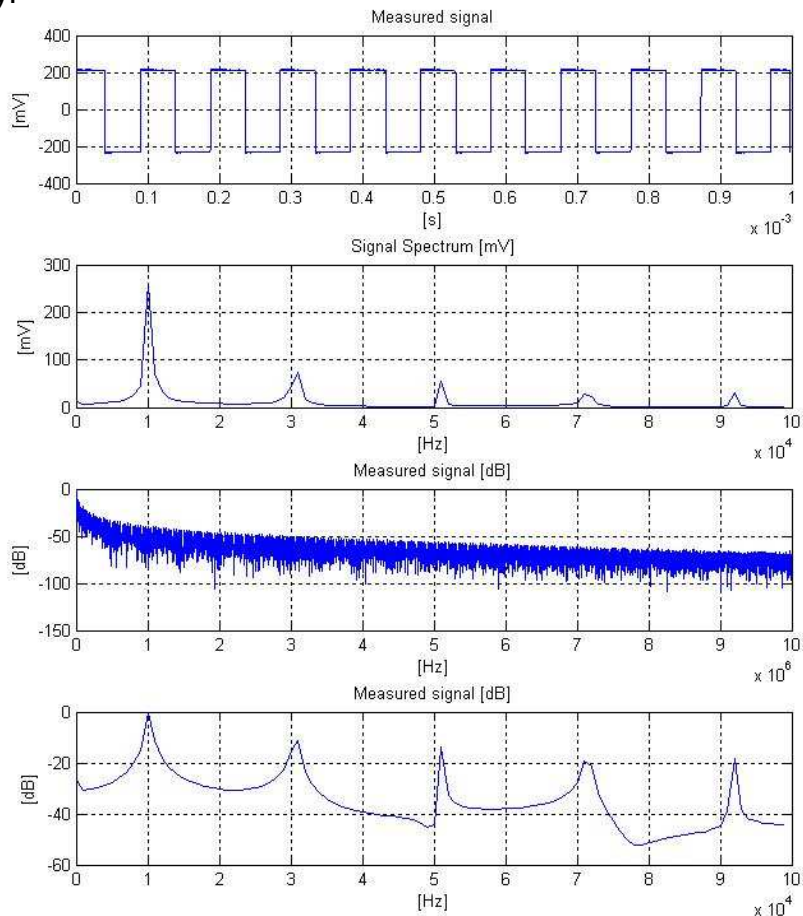


Figure 94 Frequency analysis done with Matlab script (*myFFTplot_1.m*)

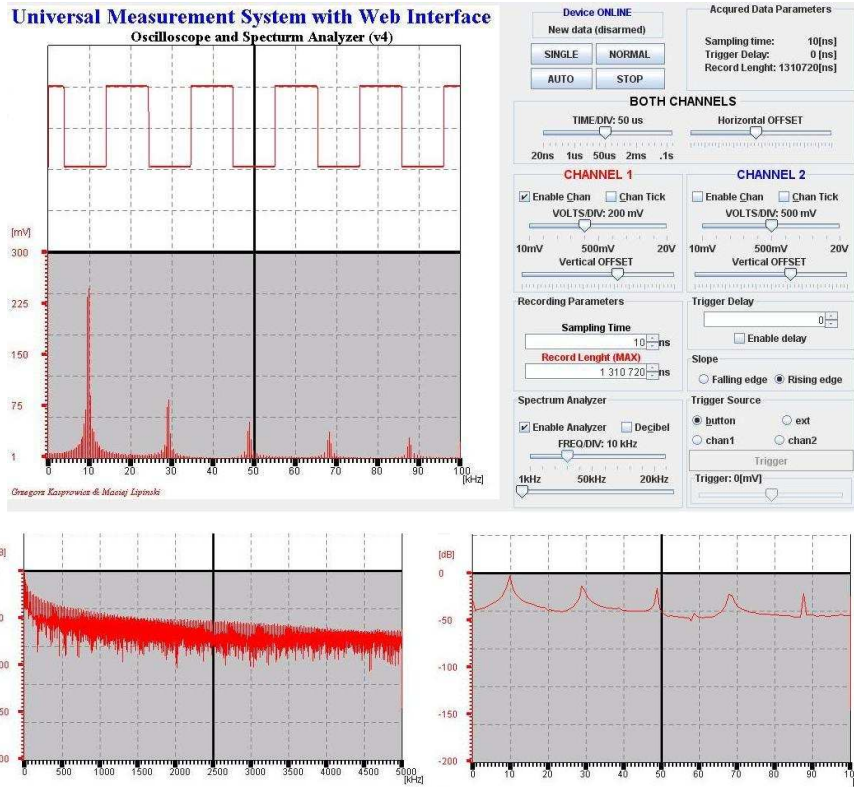


Figure 95 Frequency analysis conducted with UMSWI Spectrum Analyzer

4.2.5 Boundary conditions tests

4.2.5.1 Hardware-wise

Waveforms captured at bandwidth frequency and beyond bandwidth frequency are presented in **Figure 96**. The sine wave is of reasonable quality at 10MHz. Since, there are only 5 samples per division at 20MHz, the sine signal is more similar to triangle.

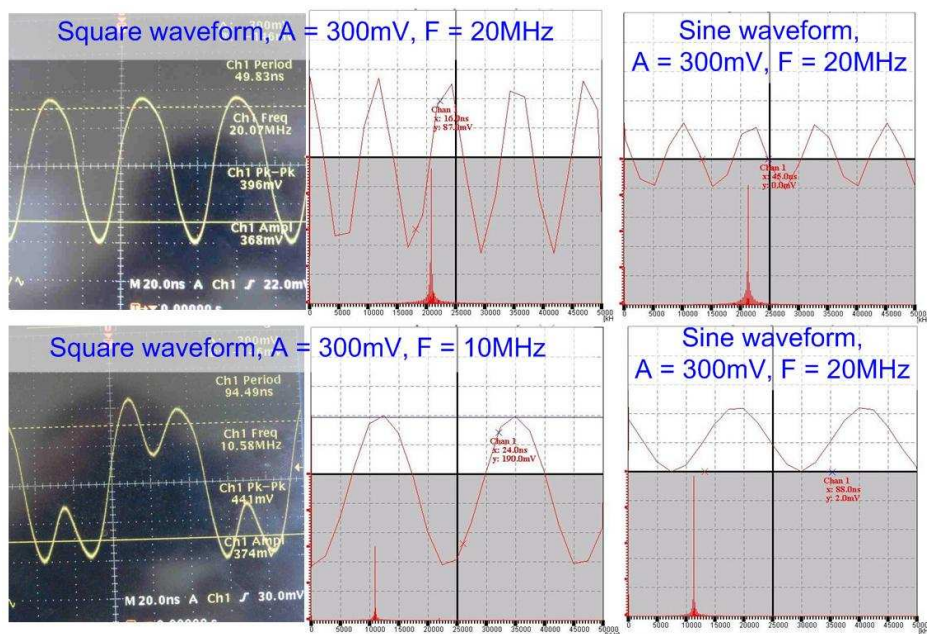


Figure 96 Sine and square signal measurement at 10 MHz and 20 MHz

Because of a hardware filter at 30MHz, the square wave measured at 10MHz does not have square shape. It can be clearly seen from the spectrum that the second, third and other harmonics were cut off by the hardware filter causing signal deformation.

When the input signal amplitude exceeds 1V, or the offset causes the signal to go beyond +500mV or - 500mV (if scaling factor applied, the values may be different), the measured signal is cut off. Exceeding the input voltage range is not recommended due to possible hardware damage. **Figure 97** presents measurement of input signal with 1.2V amplitude. The signal is obviously cut off.

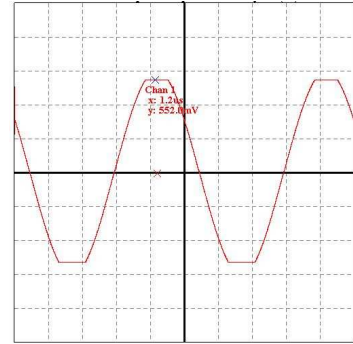


Figure 97 Input signal exceeding voltage

4.2.5.2 Software-wise

During tests in the ELPHA/PERG laboratory the UMSWI was used continuously for 6 hours without necessity of hardware reboot or software reset. The time of measurement taken using Applet application depends on the number of samples. In case of maximum memory usage (128 K words), it reaches average of 5 seconds. When the number of samples equals screen resolution (500px), the measurement time drops to less than 1s, the refresh rate in auto mode equals 0.85 times / s.

4.2.6 UMSWI parameters

Parameter name	Value
Bandwidth	10 MHz
Memory Depth	128K points (Single and Dual Channel)
Channels	Dual Channels + External Trigger
Sample Rate	100MS/s
Rising Time	25ns
Time Base Range	20ns/div to 200ms/div
Trigger models	Edge, Auto, Manual
Trigger source	CH1, CH2, Ext, Manual
Vertical Sensitivity	10mV to 1V
Vertical Resolution	10 bits
Dynamic Range	46 dB
Input Voltage	1V
Input coupling	DC
Measurement time of 128K samples	5s
Auto mode screen refresh when sample number equals screen resolution	0.85 times/s
Time base accuracy	4000 ppm
DC Vertical Accuracy	± 2.6%

Table 14 UMSWI parameters

5. System Applications

5.1 European Organization for Nuclear Research (CERN)

Universal Measurement System with Web Interface is currently used at European Organization for Nuclear Research (CERN)[53].

UMSWI was used at Proton Synchrotron (PS) to observe proton bunches. PS is a 28 GeV accelerator used as an injector for other CERN's facilities: the Super Proton Synchrotron (SPS) and the Large Hadron Collider (LHC). One of the PS accelerator parameters is harmonic number (h) – the number of proton packages being accelerated. Bunches (groups) of protons are transported in buckets. The idea is explained in **Figure 98**. The harmonic number of CERN's Proton Synchrotron ranges from 1 to 23. The frequency at which protons circulate in PS (frequency of turn) varies from 430kHz to 470kHz. The change of frequency from 430kHz to 470kHz increases protons' energy from 800MeV to 26GeV.

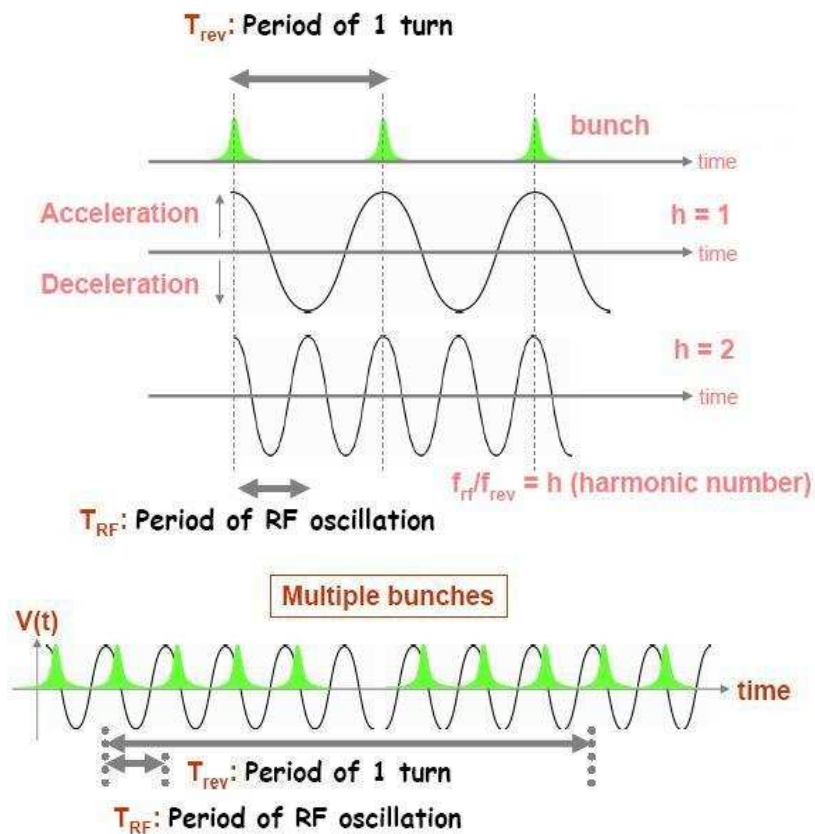


Figure 98 Acceleration of particles with AC voltage radio frequency RF [54].

Figure 99 presents measurement of a beam of protons filling 4 out of 7 buckets ($h=7$). In all the measurements, channel 1 is connected to measurement transformer, channel 2 is connected to Wall Current Monitor.

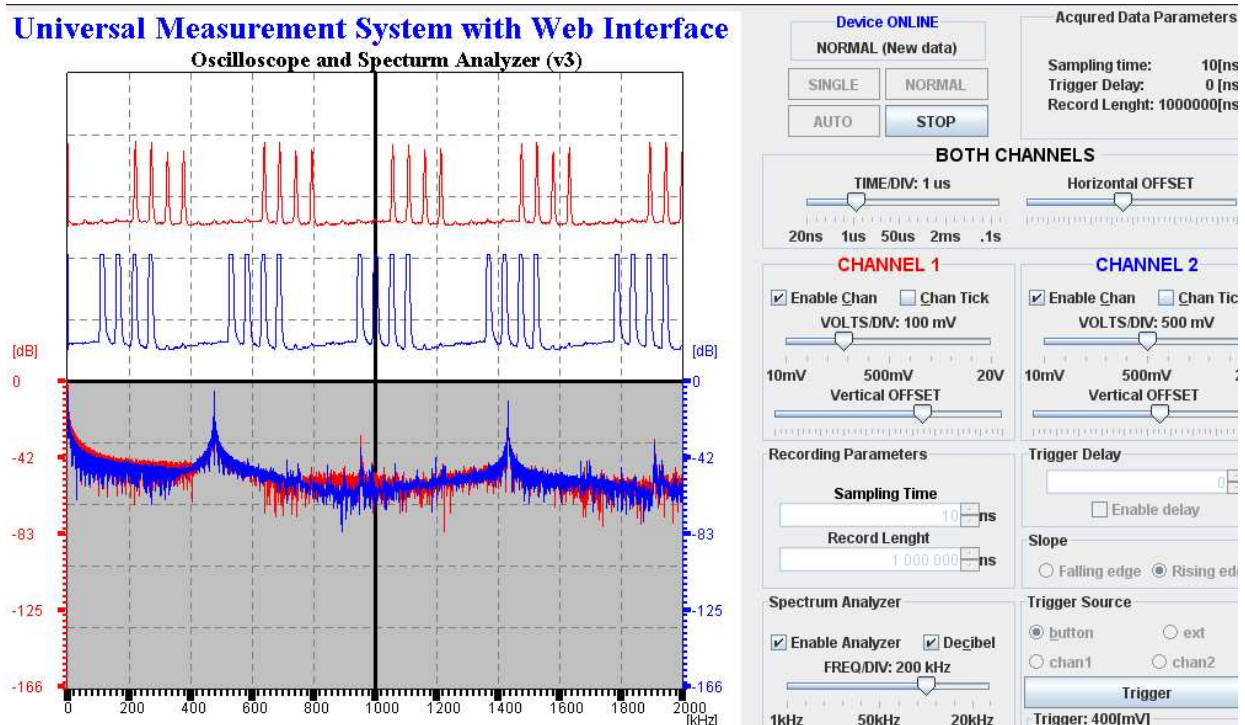


Figure 99 Four bunches of protons, $h=7$

Figure 100 presents proton beam with harmonic number of 8. All buckets are filled with protons. The energy of each bunch is slightly different, therefore periodic amplitude variation can be noticed every each picks.

A phenomena called bunch splitting takes place during harmonic number change from 7 to 21. The division of bunches during bunch splitting is presented in Figure 101.

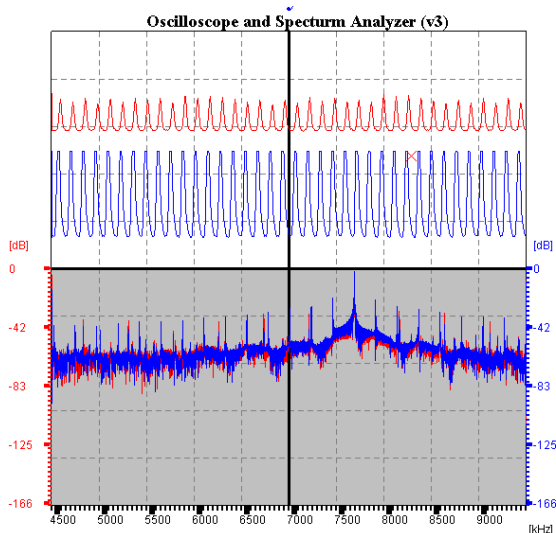


Figure 100 Eight protons in bucket, $h=8$

Universal Measurement System with Web Interface

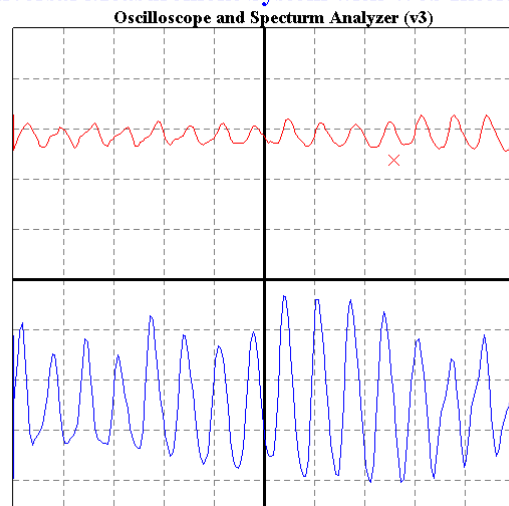


Figure 101 Bunch splitting

Figure 102 presents a situation when only there is only one bucket ($h=1$) while in Figure 103 the harmonic number is 16 and all the buckets are filled. Figure 104 presents two buckets filled with protons of different energy ($h=4$).

Universal Measurement System with Web Interface

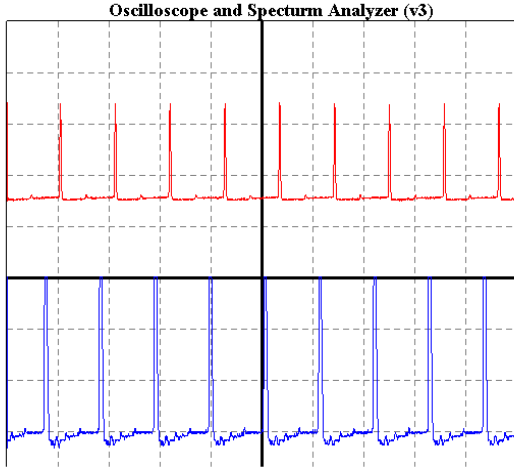


Figure 102 Single bunch, $h=1$

Universal Measurement System with Web Interface

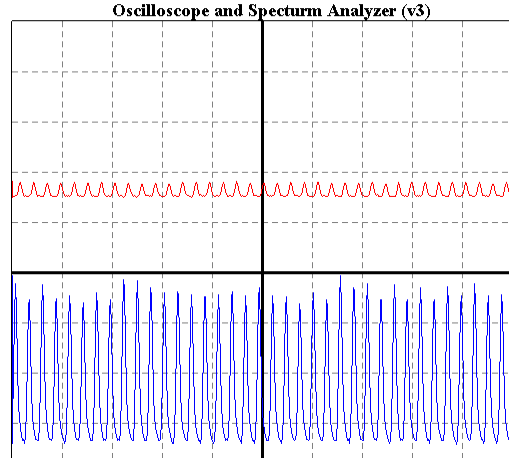


Figure 103 All 16 buckets full

Universal Measurement System with Web Interface

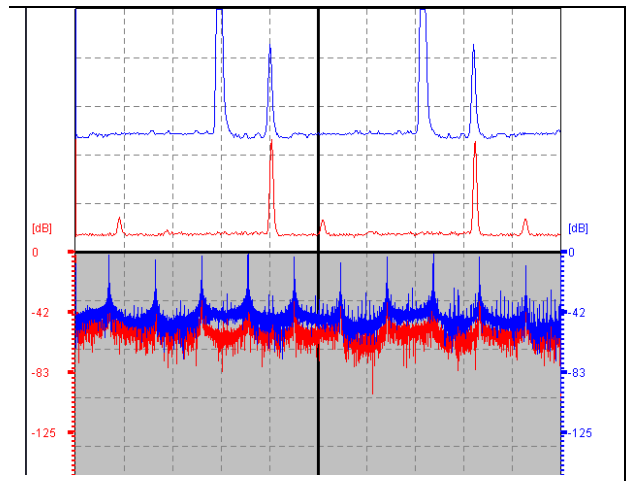
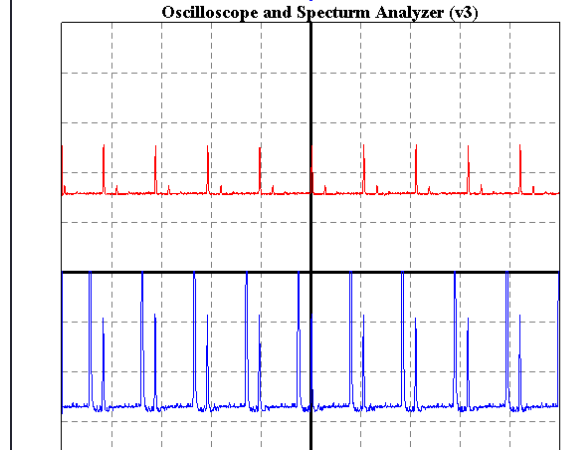


Figure 104 Two buckets filled with bunches of varied proton number

5.2 Potential applications

Potential applications of the outcome of this Master Thesis can be divided into three categories:

- Application of the system as is (without hardware or software modifications) ,
- Application of the system with modifications of software and/or configuration (content of MMC/SD card),
- Application of the control system (measurement platform with web and SCPI interfaces) on new or modified hardware platform.

Without hardware modifications, the UMSWI can be used as a very cheap (~150 EURO) oscilloscope with remote screen and measurement interface (i.e. to observe protons in Photon Synchrotron). It allows diagnostic measurements in accelerator tunnels where data acquisition needs to be done remotely due to possible radiation danger. However, it can be used to perform measurement in any dangerous places where remote data acquisition is required, i.e. areas where explosion danger is high (mines, factories), high health-risk zones (chemistry) or radioactive areas (power plants).

Thanks to the design consisting of microprocessor (running Embedded Linux) connected with FPGA, the same hardware with modified configuration files and applet (the content of MMC/SD card) can be used to perform the following measurements tasks:

- Advanced digital oscilloscope – appropriate functions need to be implemented in Java Applet,
- real time spectrum analyzer - implementing FFT algorithm in FPGA,
- software defined radio,
- 2 channels correlator,
- frequency counter,
- any device which use ADCs to measure input.

The UMSWI is also suitable for monitoring. The possibility of data processing (in FPGA or microprocessor) enables UMSWI to be configured for self-decision making (i.e. deciding whether to set up an alarm based on measured values). It is also possible to concurrently process (in FPGA) data received from ADCs and store the outcome in memory. It can be used to implement in FPGA algorithms for estimation of intensity or trajectory of particles beam in accelerators. The advantage of UMSWI over ordinary digital oscilloscopes is the fact that calculations (i.e. trajectory, intensity) can be done on UMSWI in real time. In oscilloscopes, *lag time* disables real time calculations. Modification of old and addition of new FPGA algorithms is very easy – an appropriate file on MMC/SD card needs to be replaced.

Since the hardware used to build UMSWI is modular and because the control system of UMSWI was designed to be as much platform independent as possible, there are many possible applications of UMSWI which involve hardware modification. In such applications, UMSWI is understood as a measurement platform with web and SCPI interface which enables ready-made mechanism for implementation of control GUI. The recorder module of UMSWI can be replaced by any other measurement board, thus a new measurement device with web interface is created. The recorder module can be replaced by board with radio antenna, water parameters measurement device, weather station, etc. Additionally, UMSWI can be used to create a distributed system of measurement devices.

6. Conclusions

Universal Measurement System with Web Interface (**Figure 105**) was created for diagnostic purposes in High Energy Physics having in mind current technology trends and market requirements to enable its wild usage in other places than accelerator tunnels as well. UMSWI required design and development of flexible, well-thought and easily extensible system. The objective was achieved. The system fulfilled all the initial requirements and after being successfully tested by the author in laboratory conditions, it was sent to European Organization for Nuclear Research (CERN) for further tests and operation.

The essence and main advantage of UMSWI is its build-in web interface and web server which make the device autonomous, plug & play and very convenient remotely controlled measurement system. Unlike most of the measurement devices, UMSWI does not require dedicated and separate server to be controlled via Ethernet. There is also no need for special client software. Everything is included in the device and the client needs no more than a web browser to operate it.

The system performance could be further increased introducing optimization in terms of data acquisition speed and graphic generation. However, such optimisation would introduce visible improvement only when handling large numbers of samples close to memory limits (2 x 128K samples).

Development of Universal Measurement System with Web Interface resulted in creating a control system which is vertically and horizontally flexible. Vertical flexibility is recognized by the fact that the UMSWI control system can be ported to different platforms (various microprocessors) with minor effort (Linux Device Driver porting). Horizontal flexibility means that the existing control system can be easily extended to perform other measurements as well as changed to control different hardware. Thus, the “universal” in device’s name is justified. Simplicity of extensibility was proved during the development, when the oscilloscope interface was extended by adding spectrum analyzer.

Moreover, the design and solutions used in control system of UMSWI can be a good basis for developing remote control of any system which needs to be controlled over the Ethernet. The core of the system can be reused and adapted easily. The software architecture is platform independent and requires very little resources.

Production of a measurement system based on similar hardware and UMSWI’s control system is planed by Creotech Ltd.



Figure 105 Universal Measurement System with Web Interface

Appendix A – Additional information

1. UMSWI hardware analysis

1.1 Data acquisition hardware architecture

Data acquisition and readout is managed by the FPGA. **Figure 106** presents a general overview of acquisition architecture and data flow. The data acquired from ADCs can be read by FPGA or written directly to SSRAM. It can be also written to SSRAM and read by FPGA simultaneously. After being processed in FPGA, the data can be send to microprocessor or/and written to SSRAM. The access of microprocessor to the data stored in SSRAM is possible only indirectly through FPGA.

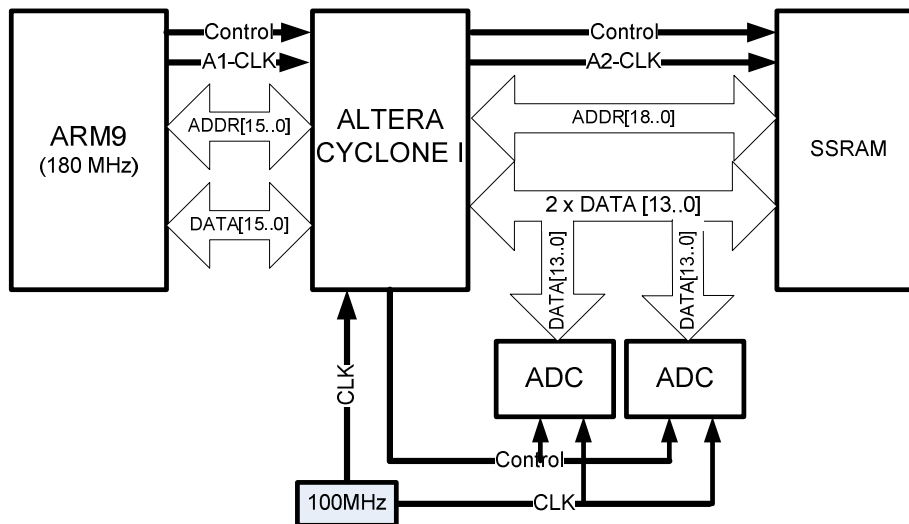


Figure 106 Acquisition hardware architecture

An important issue, which can be noticed in **Figure 106** is the fact that there are different sources of clock signal. **CLK** defines clock signal generated by oscillator which is connected to ADC and FPGA. This is a low-jitter clock signal which is required by ADCs. **A1-CLK** stands for adjustable clock provided by ARM. This is an independent clock signal for ARM's data readout. This clock can be derived by dividing the main ARM clock (180MHz) by the power of two. **A2-CLK** stands for adjustable clock generated by FPGA which can be virtually anything, in particular can be equal to **CLK** or **A1-CLK**. Therefore, the following clock domains:

- Clock domain imposed by 100MHz clock connected to ADCs - used during data acquisition,
- Clock domain imposed by ARM clock (90MHz) - used during data readout

The clock of SSRAM needs to be switched between 100MHz and 90 MHz appropriately. Two clock domains disable direct reading of data from ADCs to ARM. It is necessary to store the data first with the ADC domain frequency (or division) in SSRAM or FPGA memory. After desired number of data samples have been saved, the data can be read by microprocessor in it's clock domain. In theory, during either operation (writing to SSRAM or readout) and in between, the data can be processed in FPGA (i.e. FFT). Processing data in FPGA during acquisition is the least efficient if we want to store the outcome in SSRAM. This is because, when writing data to SSRAM without processing, data can be written to SSRAM directly from

ADCs, without going through FPGA. This results in the minimal delay, data can be written with 100MHz. If the data is processed in FPGA, one data bus needs to be switched between reading data from ADC to FPGA, and writing data from FPGA to SSRAM. It results in two times slower process and much greater delay. Data processing or analysis can be done simultaneously with data writing to SSRAM from ADCs. This is used to implement trigger by signal level when the signal level is interpreted while writing data to SSRAM.

Another issue indicated in **Figure 106** is the fact that address bus between ARM and FPGA is not as wide as address bus between FPGA and SSRAM. Thus not entire SSRAM memory space can be directly accessed from ARM.

2. Review of available technologies

2.1 Embedded Operating Systems

The ARM processor (AT91RM92000) installed on the Single Board Computer module is very popular among embedded systems. It is, of course, possible to develop applications directly for this processor. However, much better and more popular solution is running embedded operating system (OS). Developing applications for embedded system running OS does not require extensive, processor-specific knowledge. It is exactly the same as on standard PC, just the compilation must be performed for ARM architecture and the consideration of limited resources must be taken into account. ARM9 processors are so popular for embedded platforms that there are a few operating systems available for this processor:

- Linux
 - Distributions: uLinux, Denx, Embedian, BlueCat, Cadenux (open source/proprietary)
 - “vanilla” kernel + patches (open source)
- Windows CE (proprietary)
- Symbian OS (proprietary)
- Palm OS (proprietary)

Linux open source distributions:

uClinux – it supports many architectures and forms basis of many network routers, security camera, DVD or MP3 players.

Cadenux – specialized in Linux for no-MMU ARM7 and ARM9 processors. The distribution is build around uClinux.

Denx – open source distribution in form of Embedded Linux Development Kit (ELDK). It provides software development environments for real-time and embedded systems.

Embedian – a smaller version of Debian, to be used on embedded systems, it retains good features of Debian (i.e. packaging system).

2.2 Remote Measurement Interfaces

A clear distinction needs to be done between physical layer and abstract layer remote control standards. The former standards define construction and electrical parameters as well as communication protocol of physical communication link. General Purpose Interface Bus (GPIB), Recommended Standard 232 (RS-232), Universal Serial Bus (USB), VME eXtensions for Instruments (VXI) or Ethernet are means of physically connecting

the controller with the measurement instrument. Different abstract layers can be used to communicate via this physical connections.

2.2.1 Physical layer

Description based on [55].

General Purpose Interface Bus (GPIB), IEEE-488 – standard developed in 1960s by Hewlett Packard to facilitate communication between computers and instruments. It provides specification and protocol for the communication. It is a parallel bus which sends data in bytes encoded as ASCII characters. It's maximum data rate is up to 8MB/s, it allows up to 15 devices within the range of 20 m.

Serial Communication (RS-232) – a popular mean of data transfer between a computer and peripheral devices (i.e. programmable instrument). It uses a transmitter for sending data one bit at a time via single communication line to a receiver. It is used for data transfers when the speed is not crucial or when the distance is long. Unlike GPIB which needs special board plugged into the computer to enable communication, most of the PCs are equipped with serial port (however, it is less and less common). Its speed is up to 115.2kb/s (synchronous: 1Mb/s). Range: 15 m.

Universal Serial Bus (USB) – increasing popular serial bus standard which enables to connect device to a host computer. It is plug and play, enables to connect up to 127 devices to one host. It enables fast transfers (USB 2.0: 480 Mb/s).

VME eXtension for Instruments (VXI) – base on VME standard (IEEE 1014), consists of mainframe chassis with slots holding modular instruments on plug-in boards. It is popular in analysis for research/industry control application and data acquisition that require substantial number of channels (hundreds of thousands).

LAN eXtensions for Instrumentation (LXI) – standard for an instrumentation platform based on Ethernet technology. It is meant to be modular, flexible, and well-suited for small- and medium-size systems.

PCI eXtensions for Instrumentations (PXI) – standard based on PCI similarly as LXI and VXI.

Ethernet – frame-based standard in computer networking technologies for local area networks (LANs).

2.2.2 Abstract layer

Virtual Instrumentation Software Architecture (VISA) – is an API for communication with measurement instruments from PC. It is an industry standard implement in products of such companies as Agilent Technologies and National Instruments. The standard includes communication over physical links such as GPIB and VXI. VISA cannot be used directly to control instrument over LAN, however, it is used by Ethernet-enabled standards, such as VXI.

VXI-11 is an instrument protocol specification which defines a network protocol for controller-device communication over a TCP/IP network. In principle, it allows an application (client) to call procedures in the remote measurement instrument (server) as if they were local. Remote procedures are identified by the client using a unique number. Each message, along with the argument, encodes this number. According to [56] VXI-11 Devices can be programmed in two ways:

- Calling VXI-11 compliant VISA library, preferably Windows users, such libraries are available from National Instruments and Agilent
- Installing the VXI-11's Remote Procedure Call Library (RPCL) and writing programs with RPC calls, preferably Unix-like OS users

Standard Instrument Control Library (SICL) can be used to control measurement instruments over GPIB, VXI, RS-232, LAN and other physical links. It is a communication library that can be used by application written in C or C++ on various operation systems. Examples of C programs that use SICL, which can be found in [57], show that SICL is mean of communicating with measurement instruments using Standard Commands for Programmable Instruments (SCPI).

Interchangeable Virtual Instruments (IVI) defines instrument drivers standard. It builds on the VXIplug&play specifications. However, it additionally incorporates new features that address such issues as performance, development flexibility, instrument interchangeability. It can communicate with instruments across GPIB, VXI, PXI, Serial, Ethernet and USB.

Standard Commands for Programmable Instruments (SCPI) defines syntax and structure for programmable measurement and test instruments. It does not define underlying physical or software layer. It happens that instrument control interfaces are simple wrappers of SCPI commands, i.e. SICL.

2.3 Web technologies to control hardware

In order to control measurement instrument, a web server needs to interface hardware (in case of UMSWI, via Linux Device Driver). Depending on the web server's capabilities and the technology chosen there are few possibilities. If the web server embeds script interpreter (ex. PHP), driver can be accessed directly by opening its file representation. Otherwise, Common Gate Interface (CGI) can be used to call script (written in any language, i.e. shell script, perl script) which performs required action.

Common Gate Interface (CGI)

CGI enables to communicate with programs running on the server from the webpage. With CGI, the Web server can call up a program and pass user-specific data to the program. The program then processes that data and the server passes the program's response back to the Web browser. Most servers expect CGI scripts to reside in special directories (i.e. *cgi-bin*) and have special extensions (.cgi). When a user opens an URL associated with CGI script, the client sends a request to the server asking for the file. When the server recognizes that the address being requested is a CGI program, the server does not return the file content verbatim. Instead, the server tries to execute the script. The process is explained in **Figure 107**. It is worth mentioning that:

“CGI has the advantage of being a more-or-less platform-independent way to produce dynamic web content. Other well-known technologies for creating web applications, such as ASP and server-side JavaScript, are proprietary solutions that work only with certain web servers” [58]

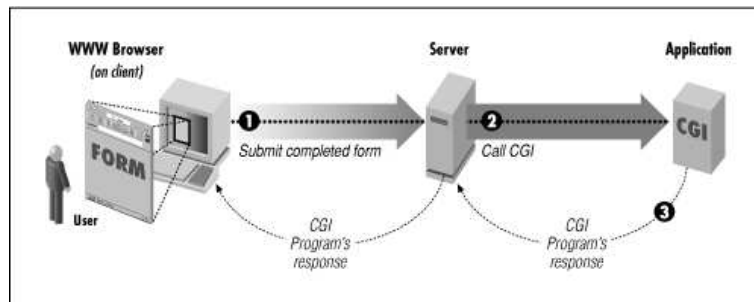


Figure 107 CGI process explanation

PHP

According to [4], “PHP is the most widely used programming language on the Web, with over 40 percent of all web applications written in PHP”. It is a server-side scripting language designed to create dynamic web content. PHP parser needs to be added to web server to generate HTML pages based on PHP. PHP is very flexible, many libraries are available which provide ready-made solutions. It is also well suited for Web Graphic generation. However, in interfacing hardware, the most important is the fact that PHP provides functions to access, read and write server-side files. It means that hardware can be controlled directly from PHP scripts by accessing files in /proc or /dev. PHP provides also functions to execute server-side applications or shell commands (i.e. exec(), system()) which can also be used to access and control hardware. Since PHP is a server-side scripting language, it is run on server and the workload of user interfacing, graphic generation or data processing is on server side.

Java Servlet

According to [58], “servlets provide an elegant, efficient alternative” to CGI and “an easy-to-connect-to, Java-based agent on the server” for Java applets. A servlet is a Java class which can be loaded dynamically to expand server’s functionality. It is run on the server inside Java Virtual Machine (JVM), therefore is portable and safe. Java Servlets do not require Java support in the web browser but they do need such support on server side. Java Servlet can control hardware by reading/writing file (i.e. in /proc file system) or using special library (i.e. JavaComm) as described in the article [59].

Active Server Pages (ASP)

Microsoft produced technology for generating dynamic web content. It enables HTML pages to contain embedded code (usually VBScript or Jscript). ASP uses COM components which are necessary for ASP’s correct performance. ASP support for other servers than Microsoft Internet Information Server Version 3.0 is commercial. Therefore, ASP can be placed among not-very-platform-independent.

JavaServer Pages (JSP)

Unlike ASP, JSP is an open standard which is implemented by many vendors across all platforms. JSP’s syntax is similar to ASP’s except that the scripting language is Java. It is closely tied with Java servlets.

2.4 Web Graphic User Interfaces

The graphic User interface can be generated on the server using Java Servlet technology or creating graphics using PHP. It can be also done by generating a graphic image from the data (on the device) and updating the image on the website. The advantage of such a solution is the fact that the user does not need to have special applications installed or browser's plug-ins enabled. However, there are at least three disadvantages:

- appropriate technology has to be ported (cross compiled) for ARM microprocessor,
- the work connected with data computation, interaction with user, etc is done on a limited-resources device (ARM microprocessor),
- all the user's requests are answered by server directly, the exchange of information between client and server is constant and heavy.

Another solution is to move most of the work to the client. Any computer used by the client is far more powerful than the ARM microprocessor, so the limitations are less strict. Moving the work to the client's side also means that the web server can be very simple. JavaScript or Java Applet enable graphic generation and user interaction handling on the client's side. The drawback of such a solution is the fact that the user needs to have web browser configured appropriately in case of Java Script. To use Java Applet, a Java Virtual Machine needs to be installed.

2.5 Web servers

The choice between generating GUI on the server or on the client is directly connected with the choice of the web server. The former solution needs a good web server with necessary tools (ex, PHP, Perl, Java, etc) the latter needs simple web server. Among many solutions tested were three worth mentioning:

- **Apache Web Server** which was successfully cross-compiled for the ARM. Apache is a widely used server, probably the first choice when developing web applications on standard computers for websites accessed by man users simultaneously,
- **KLone Web Server** is a peculiar web server developed especially for embedded systems. It allows to create dynamic pages by embedding C language in HTML. What is even more interesting, the KLone server along with developed website is compiled to a single executable. Such a solution seems quite appealing, however it has few drawbacks:
 - development can become troublesome because any change needs recompilation of the server, especially that the web server needs to be cross-compiled for ARM,
 - Bugs in the C code embedded in HTML can cause the entire server to crash.
- **Web Server provided by Busybox** - a very small web server with basic functionalities (i.e. CGI).

3. Descriptions of chosen solutions

3.1 General architecture of embedded Linux

General architecture of embedded Linux system is the same as architecture of any Linux system. At this level of abstraction all Linux system are equal. **Figure 108** presents all the components of generic Linux system architecture. Kernel is the core component of the operating system.

“Its purpose is to manage the hardware in a coherent manner while providing familiar high-level abstractions to user-level software (such as the POSIX APIs and the other de facto, industry-standard APIs against which applications are generally written)”[27]

Thanks to such architecture, applications which use the APIs provided by a kernel are portable among the various architecture. Within Linux kernel, the low-level interfaces are the part of the kernel which is platform-dependant and needs to be ported to specific architecture. Low-level services typically handle CPU-specific operations, Basic interfaces to devices and architecture-specific memory operations. Low-level hardware-dependant interfaces are managed and controlled by hardware-independent Application Programming Interfaces (APIs) of High-level abstractions.

“Above the low-level services provided by the kernel, higher-level components provide the abstractions common to all Unix systems, including processes, files, sockets, and signals. Since the low-level APIs provided by the kernel are common among different architectures, the code implementing the higher-level abstractions is almost constant, regardless of the underlying architecture”.[27]

File systems and Network protocols are good examples of components used by the kernel to understand and interact with coming from or going to certain devices structured data.

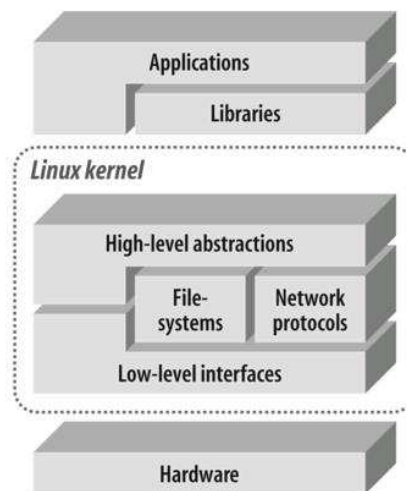


Figure 108 Architecture of a generic Linux system [27]

At least one properly structured filesystem is needed for kernel's proper operations – root filesystem. Kernel loads the first program to run on the system from root filesystem. It can be either loaded during the system on start-up into RAM and operated from there, or stored and operated from hardware storage device.

“It [kernel] also normally relies upon this filesystem [root filesystem] for certain further operations, such as loading modules and providing each process with a working directory (though these activities might take place on other filesystems mounted within the tree that begins with the root filesystem).” [27]

Very often regular application do not interface kernel directly because kernel's services are unfit to be used directly by applications. Therefore, libraries and system daemons are provided to interact with kernel on behalf of applications. One of the main libraries used in Embedded Linux Systems (instead of GNU C library used in "normal" systems, called *glibc*) is *uClibc* library. Błąd! Nie można odnaleźć źródła odwołania. presents comparison between *glibc* and *uClibc*. It is apparent that usage of *uClibc* allows to save very precious in embedded system memory space.

<i>C</i> program	Compiled with shared libraries		Compiled statically	
	<i>glibc</i>	<i>uClibc</i>	<i>glibc</i>	<i>uClibc</i>
Plain "hello world"	4.6 K	4.4 K	475 K	25 K
Busybox	245 K	231 K	843 K	311 K

Figure 109 Benefits of using *uClibc* library [29]

In most of Embedded Linux Systems, the daemons and Unix utilities (most Unix commands) are provided by a toolset called BusyBox. It is a very small-size application, single executable, which provides great functionality.

"BusyBox even includes a DHCP client and server (udhcpd and udhcpc), package managers (dpkg and rpm), a vi implementation with most of its features, and last but not least, a web server. This server should satisfy the typical needs of many embedded systems, as it supports HTTP authentication, CGI scripts, and external scripts (such as PHP). Configuring support for this server with all its features adds only 9 KB to BusyBox 1.5.0" [27]

3.2 Model-View-Controller (MVC) design pattern

"The Model-View-Controller (MVC) paradigm is a way to partition your user interface so it's easier to write and maintain. The idea is that you start with a model—a set of classes representing the data you're working with. Next, you construct various views of the data—classes that display the data on the screen. Finally, you create a controller object that accepts user input and updates the model or view." [60]

When an application uses the MVC architecture, it employs three elements to help it bridge the data and visual models that it uses. These three elements must be created and managed by the program (**Figure 110**):

- View: visible GUI which is seen by the user,
- Model: abstraction used in the program logic, represents state and nature of visual objects presented on the screen,
- Controller: enables communication between the model and view components. It updates the model according to the changes resulting from the interaction with the user.

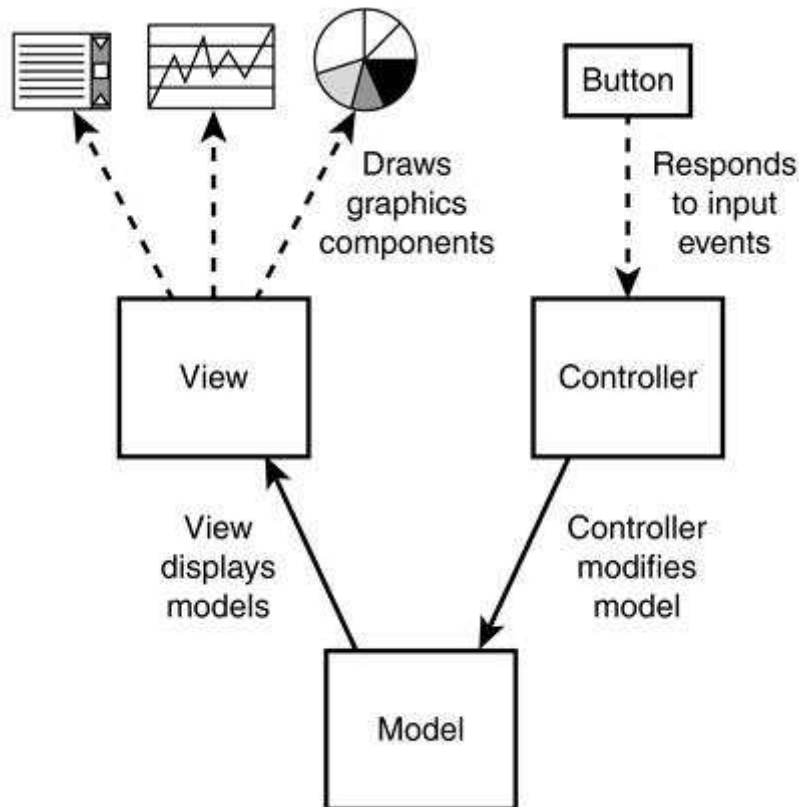


Figure 110 MVC architecture [60]

3.3 Observer-Observable paradigm

The Observer-Observable ([60, 61]) pattern consists of Observer listener interface and Observable base class which are provided by *java.util* package. An object (i.e. view) implementing observer interface registers itself as an observer of the object (i.e. model) which is an observable. Each time the model (observable) changes, all the registers observers (there can be many views) are updated.

According to [61], Observer-Observable pattern can be used in relation between the following parts of the application:

- view and controller - the changes in the view cause response in the controller,
- model and view – all the registered views are notified about model's state change.

The same book mentions different view implementations:

- “model push” vs. “view pull” – the model sends updates to the registered views or views get information from the model, when it's needed,
- Multiple view targets – more than one view can be registered to the model, thus the same data can be represented in many ways,
- “Look but don't touch” views – when the view does not provide interaction with the user.

3.4 Standard Commands for Programmable Instruments (SCPI)

SCPI defines a set of commands to control programmable test and measurement devices in instrumentation systems. It specifies command structure and syntax, it does not define underlying hardware and software. Vertical and horizontal programming consistency

is promoted by the standard. Message consistency between instruments of the same class (vertical) and between instruments with the same functional capabilities (horizontal) are defined. An example of vertical consistency is using the same command for reading DC voltage from several different multimeters. Horizontal consistency is using the same command to control similar functions across instrument classes. SCPI defines specific command sets for a given measurement functions (i.e. frequency or voltage), Thus, frequency measurement can be made in the same way in two oscilloscopes made by different manufacturers. It is also possible for a SCPI counter to make a frequency measurement using the same commands as an oscilloscope.

SCPI commands consist of set commands and query commands (simply called commands and queries). Commands change instrument settings or perform a specific action. Queries cause the instrument to return data or information about its status. Most commands have both forms. The query form is the same as the set form except that it ends with a question mark. A command message is a command or query name, followed by any information the instrument needs to execute the command or query. It consists of five element types defined in **Table 15** and presented in **Figure 111 [62]**

Symbol	Meaning
<Header>	Command name. Command is a query if the header ends with a question mark. It may begin with a colon (:) character.
<Mnemonic>	A header sub function. Most of headers consist of many Mnemonics separated by colon (:)
<Argument>	A quantity, quality, restriction, or limit associated with the header. Some command have no argument while others have multiple arguments. Arguments are separated from the header by a <Space>. Multiple arguments are separated from one another by <Comma>.
<Comma>	A single comma between arguments of multiple-argument commands
<Space>	A white space character between command header and argument.

Table 15. Command message elements

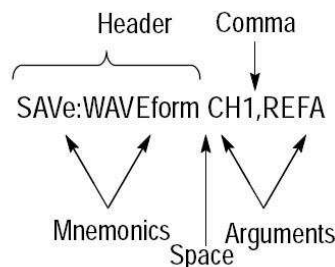


Figure 111. Command message elements

SCPI requirements concerning mnemonics' names:

- Each mnemonic has both a long and a short form. A SCPI instrument shall accept only the exact short and the exact long forms,
- The instrument shall accept both upper and lowercase characters without distinction between cases.

SCPI commands are based on a hierarchical structure created according to the style guidelines. The most important of the requirements are listed below:

1. The lowest node should have the broadest base possible,
2. Tree should be as shallow as possible,
3. A complete tree path shall be unique,
4. In general, parameters should only appear at the leaf nodes of the tree.

For an instrument a “dictionary” of commands is implemented.

4. Parameters of digital oscilloscope

Bandwidth – frequency range in which the oscilloscope measurement is accurate. It is indicated by the frequency at which the displayed signal is attenuated by -3dB (reduces to 70.7%). Well-designed oscilloscopes (i.e. Tektronix, Hewlett-Packard) tend to have flat bandwidth in entire frequency range. It means that the attenuation of the signal is close to 0 even at the specification bandwidth. Often, the specification bandwidth of such oscilloscopes is much less than its actual bandwidth.

The theoretical bandwidth is based on Nyquist-Shannon sampling theorem which says that “a signal can be reconstructed [from samples] exactly if the signal is band-limited and the sampling frequency is greater than twice the signal bandwidth”[63]. It means that theoretically, to avoid bandwidth degradation in measured signal, oscilloscope must have a sampling rate two times greater than it’s nominal bandwidth. In practice, high performance oscilloscopes manage to accommodate sampling rate of 2.5 times bandwidth. However, the mainstream oscilloscopes usually oversample the bandwidth by a multiple of 4x.

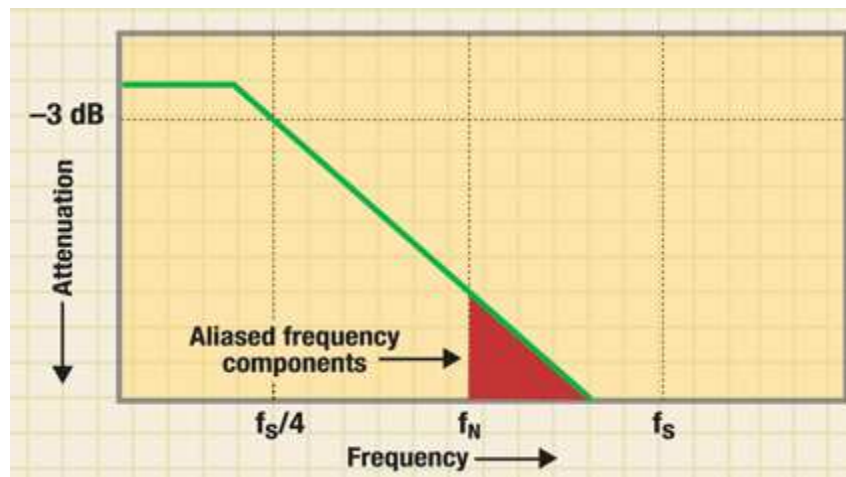


Figure 112 Relation between sampling rate and bandwidth [63]

Vertical Resolution – it is the minimal detectable voltage change, determined by ADC’s resolution and the input signal range,

Vertical Sensitivity – the smallest voltage the oscilloscope can detect (typically, 2mV/Div)

Dynamic Range – refers to how well the measurement device can detect small signals in presence of large signals, it is expressed in decibels (dB)

$$\text{Dynamic Range (dB)} = 20\log(V_{\max}/V_{\min})$$

V_{\max} – maximum voltage being acquired.

V_{\min} – minimum resolution that can be seen.

Rule of thumb: 1 bit of resolution \sim 6 dB of dynamic range (10-bit instrument’s theoretical maximum dynamic range is 60dB)

Accuracy - ability of an instrument to represent the true value of a signal. The achievable accuracy of an oscilloscope (any measurement instrument) is limited by the resolution of

the ADC. However, the high resolution does not guarantee high accuracy. Factors that reduce accuracy mostly occur at high and low frequencies.

Gain accuracy – accuracy of amplitude measurement, determines how well oscilloscope handles high-frequency noise

Offset accuracy – accuracy of offset in DC coupling mode, determines how well oscilloscope handles low-frequency errors

Horizontal axis resolution – limited by sampling rate. 100MS/sec acquisition rate can achieve a time resolution of $1/(100\text{MS}/\text{sec}) = 10\text{ns}$. Accuracy of horizontal axis can be reduced by low- and high-frequency errors alike vertical axis. However, these errors are usually insignificant compared to problems with accuracy of vertical axis.

Time base accuracy – specifies frequency/period measurement instability, is expressed in parts-per-million (ppm)

Rise time – time needed by the signal to go from a specified low value to a specified high value. The low value is usually specified as 10% of set height and the high value is specified by 90% of set height. It describes useful frequency range on an oscilloscope. Pulses with rise time faster than oscilloscope's rise time cannot be displayed accurately.

Memory – in most oscilloscopes sampling rate and memory size are intertwined since they want to fill entire (fixed-size) window with the waveform. In some settings configuration it may lead to situation where both time and memory are maximized. Since maintaining sampling rate is more important, entire memory is used. Usually, sampling rate is sustained as long as the scope does not run out of memory to fill the display, otherwise sampling rate is decreased (bandwidth is therefore memory dependant as well). In case of UMSWI, sampling rate is always maintained and, if required, the waveform is smaller than screen display.

Appendix B – FPGA – ARM interface

Example commands		Output/input from/to driver data format	Proc_fs	R/W	address	Register name	Variable in FPGA	comment					
Echo 1 > reset		1	reset	W	0x...00		ARM_reset	asynchronous					
Echo 16 > cmd Echo 32 > cmd Echo 48 > cmd		Number(decimal) ARM Trig ARM + Trig	Config cmd	R/W	0x...10	Reg01[1..0]	ARM_trig_src[1..0]	0-ARM; 1-ext; 2- chan1; 3- chan2					
						Reg01[2]	ARM_time_en	0-disable; 1-enable					
						Reg01[3]	ARM_slope	0-rising; 1-falling					
						Reg01[4]	ARM_ARM	0- idle; 1 - ARM					
						Reg01[5]	ARM_trigger	0- not trigger; 1- trigger					
1 2 3	Data acquired ARMED Data acquired and armed		state	R	0x...20	Reg02[0] Reg02[1]	ARM_DATA_ACQUIRED ARM_ARMED	0 – not; 1 - yes 0 – not; 1 – yes					
Echo 20:0:0 > parameters		Len:time:delay	parameters	R/W	0x...30	Reg03	ARM_record_len[15..0]						
						Reg04	ARM_record_len[18..16]						
					0x...50	Reg05	ARM_timer[15..0]						
						Reg06	ARM_timer[23..16]						
					0x...70	Reg07	ARM_delay[15..0]						
						Reg08	ARM_delay[31..16]						
							startAddr	startRDaddr	R/W	0x...90	Reg09	ARM_start_RD_addr[15..0]	Problem with reading, MASK=0x0FFF
											Reg0A	ARM_start_RD_addr[18..16]	
		StartAddr stopAddr	addressPointers	R	0x...B0	Reg0B	ARM_start_addr_pointer[15..0]						
						Reg0C	ARM_start_addr_pointer[18..16]						
					0x...D0	Reg0D	ARM_stop_addr_pointer[15..0]						
						Reg0E	ARM_stop_addr_pointer[18..16]						
			readSingleData	R	0x...100		RD_DATA						
			read2words	R			WR_DATA						

Universal Measurement System with Web Interface

		readXwords	R/W				
Cat readresult		readresult	R				
<p>Test 0 – special states for writing and reading from memory</p> <p>IIde->write_ssram->read_ssram->completed(waiting for data to be read)</p> <p>For this test data outputted by “readresult” is displayed differently than normal</p> <p>0x01 – writing address to the memory</p> <p>0x11 – writing 0x5555 to even and 0xAAAA to odd addresses on channel 1 and 0x0000 to channel 2</p> <p>0x21 – writing 0x5555 to even and 0xAAAA to odd addresses on channel 2 and 0x0000 to channel 1</p> <p>0x31 – writing 0x0000 to both channels and all addresses</p>	0xNumber	test	R/W	0x...110	regT[0]	ARM_SSRAM_test_0	<p>Test 0</p> <p>Cat readresult: addr ->> chan1: chan2 0x1 ->> 0x2aa: 0x8</p>
<p>Test 1 – everything works as normal, but instead of reading data from ADC, the data is read from FPGA (and written to memory),</p> <p>For this test data outputted by “readresult” is displayed differently than normal</p> <p>0x02 – writing address to the memory</p> <p>0x12 – writing address to channel 1 and 0x0000 to channel 2</p> <p>0x22 – writing addresses to channel 2 and 0x0000 to channel 1</p> <p>0x32 – writing 0x0000 to both channels and all addresses</p>	0xNumber				regT[1]	ARM_SSRAM_test_1	<p>Test 1</p> <p>Cat readresult: addr ->> chan1: chan2 0x1 ->> 0x2aa: 0x8</p>
<p>Test 2 – everything works as normal, but instead of reading data from ADC, the data is read from FPGA (and written to memory)</p> <p>For this test data outputted by “readresult” is displayed</p>	0xNumber				regT[2]	ARM_SSRAM_test_1	<p>Test 1</p> <p>Cat readresult: Chan1: chan2</p>

Universal Measurement System with Web Interface

<p>normal , can be used with applet</p> <p>0x04 – writing address to the memory</p> <p>0x14 – writing address to channel 1 and 0x0000 to channel 2</p> <p>0x24 – writing addresses to channel 2 and 0x0000 to channel 1</p> <p>0x34 – writing 0x0000 to both channels and all addresses</p>							
					regT[7..4]	ARM_test_kind	Depending on the test kind

Appendix C – Example Manual

Below, Oscilloscope and Spectrum Analyzer Manual is presented. This manual is available on the UMSWI's website and is presented here as an example. The website provides also SCPI Manual (with example scripts in Matlab) and information concerning UMSWI's configuration. The website is included in the CD. It can be also found on author's homepage [47]

1. JAVA APPLET

The Oscilloscope and Spectrum Analyzer is a Java Applet and you will need Java Virtual Machine installed and Java enabled in your browser to have it up and running. For details how to successfully run applet in your browser see [64]. The recommended browser to operate the oscilloscope applet is Mozilla Firefox

2. ONLINE/OFFLINE

Applet automatically detects whether it has connection with the server. If something is wrong with the connection, it is indicated by red sign Device OFFLINE. If everything is ok, there should be Device ONLINE in blue.

- ONLINE - good for you, it means that everything is connected and installed properly, just enjoy using. If you want just to test the applet and you don't have any source of signal, you can ask it to generate signal:
 - i. click with right button of the mouse on the screen
 - ii. select Enable test data
 - iii. choose which waveform you want to see (affects only channel 2)
 - iv. use the applet as if there was a signal source connected to the device

- OFFLINE - for tests
 - i. probably you are using the applet on the author's homepage, this one is not connected to any hardware
 - ii. if you are using applet located on the ARMputer and it is indicated that the applet is OFFLINE, something is wrong :(

3. COMPONENTS OF THE GUI

When you open the Oscilloscope Web page, you will see Screen and Control Panel. These are all you need during normal operation. Control panel enables you to set acquisition parameters, start/stop acquisition, adjust the view of the results and decide what should be displayed on the screen. Screen presents results of measurement acquired with the parameters you wanted. When you start the applet, the screen is empty. It will stay empty even after the acquisition if you do not enable any of the channels. Except of screen and control panel, an auxiliary window can be opened by clicking the screen with right button of the mouse. Auxiliary Panel provides functions which are rarely, i.e. it enables you to see raw data. Raw data is are the voltage values which were received from the hardware, scaled by the factor indicated. Spectrum raw data, is the outcome of Fast Fourier Transform calculation performed on the raw data.

Universal Measurement System with Web Interface

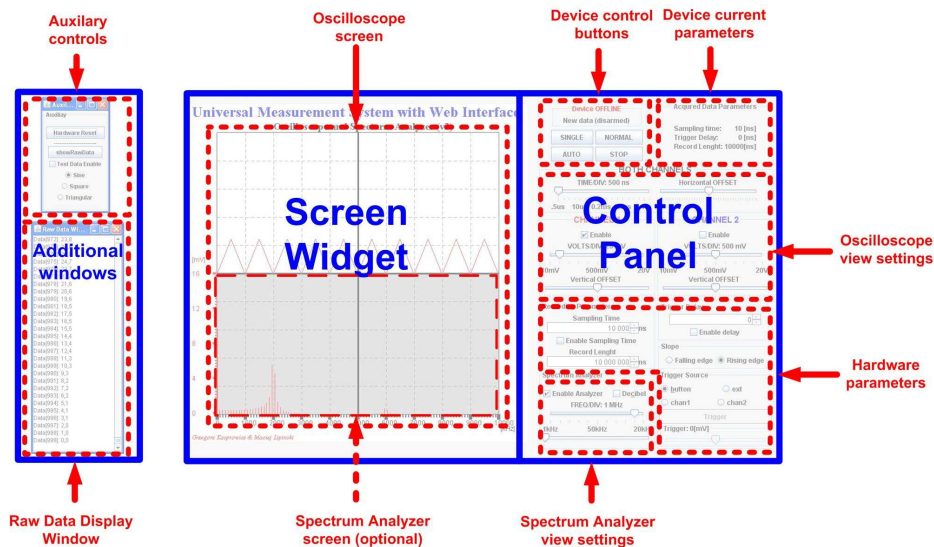


Figure 113 Oscilloscope & Spectrum Analyzer GUI

4. SCREEN

The screen has multiple usage. In initial state it displays nothing but the grid in oscilloscope-like window. What is currently displayed on the screen depends on the control panel settings, in general the screen can show:

- Nothing – when none of the channels is enabled
- Input signal to channel 1 or/and 2
- Spectrum of input signal to channel 1 or/and 2 along with input signal to channel 1 or/and 2

Spectrum of a given channel is displayed only if the channel is enabled. Along with spectrum chart, an appropriate spectrum scales on the screen margin is displayed (depending on the kind of spectrum, it is either mV or dB scale).

By dragging the screen (pressing left button of the mouse and moving the mouse), the horizontal and vertical position of signals can be changed.

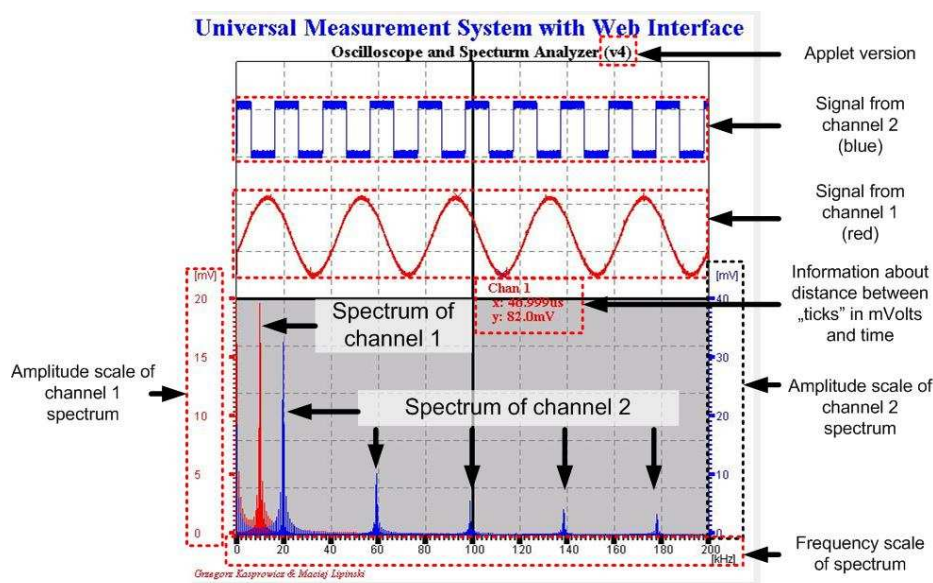


Figure 114 Oscilloscope & Spectrum Analyzer Screen

The oscilloscope enables to measure distance in mV and time between two points on the screen (called “ticks”). It can be enabled on the Control Panel. The mechanism is explained in the figure below.

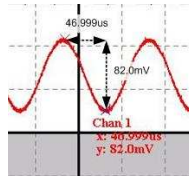


Figure 115 Oscilloscope's “Ticks”

Once the “ticks” are enabled (for specific channel), the cursor of the mouse is followed on the screen by “X”. The first “X” is red, it is the starting point for distance calculation. When a place on the screen is clicked, the red “X” is left in this place and a blue “X” appears. The blue “X” is accompanied with the information about the channel for which the “ticks” are enabled (different channels can have different volts/div settings, thus the measurement of distance is different), the voltage and time measured in the way explained in the figure above. In working with “ticks” the following rolls must be remembered:

- The tick which follows mouse cursor can be set in a place on the screen by clicking the screen.
- When both ticks are set in a position on the screen (the mouse cursor is “free”):
 - If red tick is clicked with the mouse cursor, positions of both “ticks” are reseted and red “X” starts to follow the cursor
 - If any place, except red “X”, is clicked, position of blue “X” is reseted, and it starts following the mouse cursor.
- When the left mouse button is kept pressed, the position of displayed signals can be changed

5. CONTROL AND AUXILIARY PANELS

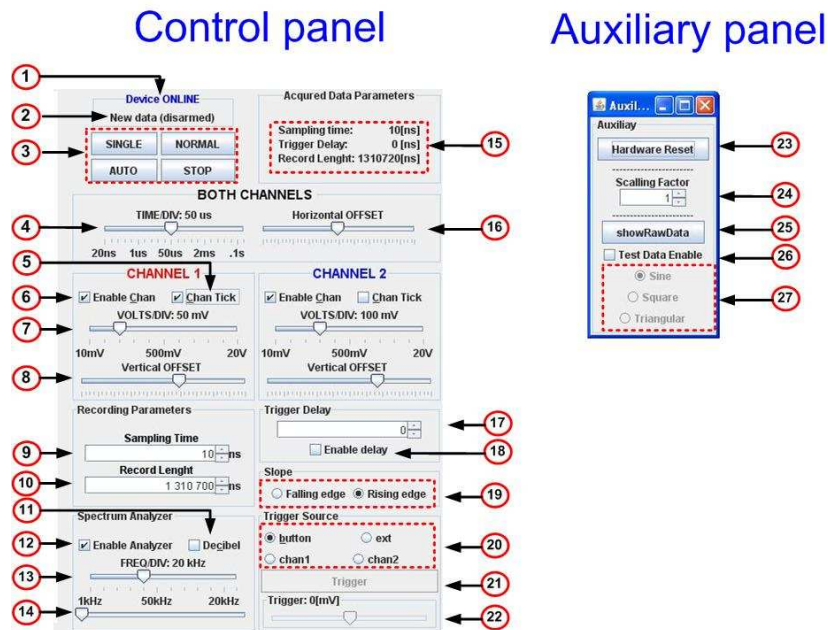


Figure 116 Oscilloscope and Spectrum Analyzer's control and auxiliary panel

1. Shows whether Applet is connected to the server and hardware (see 1).
2. State of the device:
 - a. IDLE – initial state,
 - b. SINGLE – acquisition is running in single mode,
 - c. AUTO – acquisition is running in auto mode,
 - d. NORMAL – acquisition is running in normal mode,
 - e. STOPPED – when acquisition was forced to stop by clicking STOP,
 - f. Acquiring data – when data is being sent from the server to the applet,
 - g. New data (disarmed) – Acquisition stopped after successfully acquiring data.
3. Enable to control acquisition:
 - a. SINGLE – data is acquired one time after trigger occurred,
 - b. NORMAL – data is acquired each time the trigger occurs until stopped with STOP button,
 - c. AUTO – data is acquired continuously, regardless of the trigger occurrence, until stopped with STOP button,
 - d. STOP – stops data acquisition.
4. Set TIME/DIV for both channels.
5. Enables measurement of voltage and time with “ticks” for channel 1.
6. Enables display of channel 1.
7. Sets VOLTS/DIV for channel 1.
8. Sets vertical position of channel 1 signal .
9. Sets sampling time (range: 10ns – 250us).
10. Sets record length (maximum record length depend on the sampling time: $recLen = 131072 * samplingTime$).
11. Changes spectrum scale from mV to dB.
12. Enables spectrum analyzer – spectrums of enabled channels are displayed on the screen .
13. Sets FREQ/DIV .
14. Changes horizontal position of spectrum.
15. Displays parameters set during latest acquisition – the data which is displayed on the screen was acquired with this parameters set.
16. Changes horizontal position of signals.
17. Sets delay time in [ns] – delay time is an interval between trigger occurrence and acquisition start.
18. Enables delay – it is not enough to set the delay time in 17, it needs to be enabled to here.
19. Sets the edge on which trigger should occur.
20. Trigger source:
 - a. Button – acquisition is started when “Trigger” button is pressed,
 - b. Chan2 – acquisition is started when signal on channel 2 fulfils “trigger conditions” (trigger level and edge),
 - c. Ext - acquisition is started when falling/rising edge is detected in external signal.
21. Starts acquisition when trigger source is set to “button”.
22. Sets trigger level when trigger source is set to “chan 1” or “chan 2”.
23. Send reset signal to FPGA logic.
24. Data read from the device is multiplied by this factor.
25. Displays “raw data” in a separate window.
26. Enables to test the applet without any source of signal – signal is generated by the applet itself as if when offline.
27. enables to choose kind of generated signal on channel 2 (only).

Appendix D – Developer’s web page

This is a content of the web page existing on author’s web site [65] which is meant for UMSWI developers. It contains all the codes and binaries used by the system and explains how to used them. It is included on the CD as well.

1 Downloads:

- Binaries (Download):
 - loaderML.bin - Bootloader, source code (modified from TWarm, which in turn was modified from Flabio Ribeiro (etc.) - this bootloader was modified to start the Linux directly. It means that, in normal operation, the zImage is copied from flash and run by bootloader (not U-Boot), it is recommended to upload new zImage to flash using bootloader. In case of development U-Boot can be started by choosing menu option 4
 - zImageML - kernel + rootfs in one zImage (Linux-2.6.19, taken from TWarm, modified configuration, customized rootfs, includes all the UMSWI's utilities in /usr/AMRscope folder)
- Configuration (Download):
 - busybox.config - configuration used for Busybox-1.00
 - kernel.config - configuration used for Linux-2.6.19 (patched and modified by Poles in TWarm)
- Oscilloscope and Spectrum Analyzer Java Applet source code (Download) - the project was developed in Eclipse, it needs javac at least 1.5. It was painfully learnt that earlier version are not enough. The compilation should be done under Linux.
 - src/ - folder with sources
 - bin/ - folder with compiled classes
- Bootloader's source code (Download)
- SCPI server (Download):
 - src/ - folder with sources
 - doc/ - documentaton generated by doxygen, available here as well
 - scpi_server - binary (to run SCPI Server on port 2020 : ./scpi_server -s 2020)
- Linux Device Driver providing communication with FPGA (Download):
 - src/ - folder with sources
 - fpga.ko - driver compiled as module (to instal issue: insmod fpga.ko)
- FPGA Logic (Download) - it is a project in ALTERA Quartus II - the entire VHDL code is in the file: acquisition_controller.vhd, the binary file is here
- FPGA configuration application (Download)
 - src/ - folder with sources
 - config - compiled binary (to load mag_fpga.rbf: ./config mag_fpga.rbf)
- MMC/SD content(Download) - Content of MMC/SD card is initially equal to /usr/ARMscope folder in rootfs,
- Root file system (Download)

2 Development environment

The project is being developed with the workstation running Debian distribution of Linux (GOOD BLESS Debian :). The cross-compilation tool used during the development was intalled as a debian package using **Synaptic Package Manager**. The package was prepared by Free Electrons. I followed this, see Lab 3- Cross-compilation. Remember to export:

```
export PATH=/usr/local/uclibc-0.9.28-2/arm/bin:$PATH
export CROSS_COMPILE=arm-linux-
export ARCH=arm
```

3 MMC/SD card content

Tools and data which are used by the system are stored in /usr/ARMscope folder and on MMC/SD card. Its organization is presented below:

```

-ARMscope
---data
---FPGAconfig
---FPGAdriver
---scpi_server
---scripts
---www
-----cgi-bin
-----oscilloscope
-----systemConfig
-----data
-----images
-----oscilloscope
    
```

Figure 117 MMC/SD card content

FPGAconfig/ holds the .rbf file with FPGA logic configuration and a small application which configures FPGA.

- **FPGAdriver/** holds FPGA Linux Device Driver compiled as a loadable module and a script which loads the driver and creates entry in /dev
- **scpi/_server** holds SCPI server application
- **www/** - the UMSWI website
 - **cgi-bin/** - CGI scripts
 - **oscilloscope/** - used in applet-driver communication
 - **systemConfig/** - used for system configuration
 - **oscilloscope/** - oscilloscope and spectrum analyzer applet
 - **data/** - data available on the website, i.e. Matlab scripts
 - **images/** - images used on the website
- **data/** - holds information which needs to be stored between boots, i.e. default IP
- **start** is a script which starts UMSWI utilities

4 Linux start-up

During the development phase, both loaders (Bootloader and U-boot) were used. U-boot passes to kernel boot parameters and PHY parameters (i.e. MAC address). To enable booting the kernel and root file system from flash memory without U-Boot, modifications in BusyBox's configuration and TwARM's bootloader were needed. A tool enabling MAC address to be set when Linux is on (Networking Utilities --> ifconfig/Enable option "hw" (ether only)) was added in BusyBox configuration and Linux start-up script (/etc/inittab) was appended with the line which sets up MAC address. The loader was modified to include default Linux start after short delay. Modified loader's menu is presented in the picture . Starting u-boot is still possible , since it can be useful for further development and there is enough space in the flash memory. However, a modification was made to the address in which the u-boot is started.

```

Initializing SDRAM
Universal Measurement Sytem - by Maciejx
32bit SDRAM 2xHynix HY57V561620C

1: Upload loader to Dataflash with vector 6 modification.
2: Upload u-boot to Dataflash.
3: Upload linux to Dataflash
4: Start U-boot
5: Start linux
6: Start u-boot and linux
7: SDRAM test
    
```

Figure 118 Bootloader's menu

UMSWI specific startup operations are done in three steps:

- 1. The MMC/SD card with UMSWI utilities is attempted to be mounted in /usr/ARMScope/ location. The /usr/ARMScope/ folder hold all the custom-made UMSWI utilities. The mounting is done in /etc/init.d/rcS system initialization script
- 2. httpd web server is started as "respawn" (/etc/inittab file)
- 3. /usr/ARMScope/start script is called (in /etc/inittab) . This script is used for the UMSWI utilities initialization and can be modified by the user easily. It starts the following initialization (by calling appropriate scripts):
 - Configures FPGA (config_FPGA script)
 - Loads FPGA driver (load_driver script)
 - Starts SCPI Server if Enabled (start_scpi script)
 - Sets the default IP (set_IP script)

**The following Linux start-up were prepared appropriately
/etc/inittab**

```
-----
# /etc/inittab
#
# Copyright (C) 2001 Erik Andersen <andersen@codepoet.org>
#
# Note: BusyBox init doesn't support runlevels. The runlevels field is
# completely ignored by BusyBox init. If you want runlevels, use
# sysvinit.
#
# Format for each entry: <id>:<runlevels>:<action>:<process>
#
# id      == tty to run on, or empty for /dev/console
# runlevels == ignored
# action  == one of sysinit, respawn, askfirst, wait, and once
# process == program to run

# Startup the system
null::sysinit:/sbin/ifconfig eth0 hw ether 00:08:03:7a:3e:16
null::sysinit:/sbin/ifconfig lo 127.0.0.1 up
null::sysinit:/sbin/route add -net 127.0.0.0 netmask 255.0.0.0 lo
null::sysinit:/sbin/ifconfig eth0 192.168.1.101 up
null::sysinit:/sbin/route add -net 192.168.1.101 netmask 255.255.255.0 eth0

# main rc script
::sysinit:/etc/init.d/rcS

#start ARMScope utilities

null::sysinit:/usr/ARMScope/start
null::respawn:/usr/sbin/httpd -h /usr/ARMScope/www/
```

```

# Set up a couple of getty's
#tty1::respawn:/bin/runterm.sh
#tty2::respawn:/sbin/getty 38400 tty2

# Put a getty on the serial port
ttyS0::respawn:/sbin/getty -L ttyS0 115200 vt102
#::respawn:/bin/sh
#::respawn:/bin/login -- root

#run application
#ttyS0::once:/mnt/flash01/startup

# set up stuff for logging
#tty4::respawn:/usr/bin/tail -f /var/log/messages

# Stuff to do for the 3-finger salute
::ctrlaltdel:/sbin/reboot

# Stuff to do before rebooting
null::shutdown:/bin/umount -a -r

```

/etc/init.d/rcS

```

#!/bin/sh

mount -t proc none /proc
#mount -t devpts none /dev/pts

#echo 'mounting /usr/ARMScope/'
sleep 3
mount -t vfat /dev/mmcbk0 /usr/ARMScope/
sleep 3

```

Sleep is needed to allow Linux to "see" the mmcbk0 device and later to mount it.

/etc/fstab

```

# /etc/fstab: static file system information.
#
# <file system> <mount point> <type> <options> <dump> <pass>
/dev/root / auto defaults,errors=remount-ro 0 0
proc /proc proc defaults 0 0
/dev/mmcbk0 /usr/ARMScope vfat defaults 0 0

```

5 ARMScope package

Everything to build zImage should be available here: Download(270MB !!!!)
This package is located (not entirely legaly) on EITI's server mion, the transfer is not good and it may be deleted by admin at any time

If you manage to download it (CONGRATULATIONS), this is how the zImage can be created:

5.1 Development environment

See 2 *Development Environment* to know how to install cross compilation toolchain. Don't remember to export environmental variables:

```
$export PATH=/usr/local/uclibc-0.9.28-2/arm/bin/:$PATH
$export CROSS_COMPILE=arm-linux-
$export ARCH=arm
```

Things will need to be done as root (\$su). The ARMScope package needs to be untarred in convenient location, in my case in /home/maciex/armbuild (\$tar -xvzf armscope.v8). The package contains the following stuff:

- **linux-2.6.19** - patched, appropriately modified and configured kernel
- **busybox-1.00** - configured Busybox
- **config** - configuration files for Linux and Busybox
- **loader_ML** - source code of modified Bootloader
- **root_fs** - root file system, the one which is compiled into zImage
- **SD_card** - content which should be copied to SD card
- **binaries** - compiled Bootloader and zImage

5.2. Configuration and compilation of busybox

If Busybox needs to be compiled (usually it's not the case), the location of its installation needs to be indicated.

```
$cd armscope.v8/busybox-1.00/
$make menuconfig
```

Set location of root_fs (in my case: /home/maciex/armbuild/armscope.v8/root_fs) in *Installation Options*

```
$make clean
$make
```

5.3. Configuration and compilation of kernel

Before compiling kernel, its configuration needs to be change, so that the root_fs folder location is indicated

```
$cd armscope.v8/linux-2.6.19/
$make xconfig
```

Go to: *General setup* --->*Initramfs source file(s)*: and set the location (in my case: /home/maciex/armbuild/armscope.v8/root_fs)
Save changes.

```
$make clean
$make
.....Wait.....
zImage is in: arch/arm/boot
```

Appendix D – Additional Materials on the Accompanying CD

Location		Content	
Bibliography/		All the articles, datasheets, information which is in the “Bibliography” list and could be legally downloaded	
Binaries/		Binaries for UMSWI	
SD_card/		Content of MMC/SD card which should be inserted to the device	
Tests		Test data, Matlab scripts with results interpretation	
Development/		Set of tools, codes etc which can be used to further develop UMSWI	
	Environment/	Patched and configured Linux kernel, Busybox, u-boot, Bootloader, prepared root file system. With very few changes in configuration (setting the right paths) it can be used to create binaries for UMSWI	
	FPGAconfig/	Application used for FPGA configuration	
	bin/	Binary	
		src/	Source code
	FPGAdriver/	FPGA Linux Device Driver – used for communication between Linux User Space and FPGA Logic	
	bin/	Cross-compiled Linux Module	
		src/	Source code
		doc/	Doxygen generated documentation
	FPGAlogic/	FPGA logic including Communication and Data Acquisition Management Logics	
	bin/	rbf file	
		src/	VHDL source code
		QuartusProject/	Project in Quartus used for FPGA logic development (with pins assigned)
	JavaApplet/	Oscilloscope and Spectrum Analyzer Java applet	
	bin/	Binaries (compiled under Linux with java version “1.5.0_14”)	
		src/	Source code
		EclipseProject/	Project in Eclipse used for applet development
		doc/	Javadoc documentation
	SCPIserver/	SCPI Server	
	bin/	Crosscompiled for ARM	
		src/	Source code
		doc/	Doxygen generated documentation
	WWWforDevelopers/	UMSWI website as is on author’s homepage, it includes additional page for developers	
	WWWforUMSWI/	UMSWI website embedded in the device	
MaciejLipinski.doc		Master Thesis (MS World)	
MaciejLipinski.pdf		Master Thesis (pdf)	
UMSWIwebsite		Shortcut to UMSWI website (as is provided by the device)	
UMSWIdevelopersWebsite		Shortcut to developers’ UMSWI website (as is on the authros’s website)	

Appendix E – List of Figures

- Figure 1 e*Scope basic mode
- Figure 2 e*Scope advanced mode
- Figure 3 Remote control of Agilent Analyzer
- Figure 4 BenchLink applet
- Figure 5 BitScope instrument and GUI
- Figure 6 BitScope Model 100 architecture
- Figure 7 UMSWI architecture and dataflow
- Figure 8 General UMSWI architecture
- Figure 9 UMSWI architecture
- Figure 10 UMSWI architecture
- Figure 11 Acquisition and readout control and dataflow
- Figure 12 UMSWI architecture
- Figure 13 Communication between FPGA and ARM
- Figure 14 UMSWI architecture
- Figure 15 Choice of technologies for Web Interface of UMSWI []
- Figure 16 Oscilloscope and Spectrum Analyzer Web architecture
- Figure 17 UMSWI's architecture according to MVC
- Figure 18 Web User Interface architecture
- Figure 19 UMSWI architecture
- Figure 20 SCPI example command
- Figure 21 SCPI server architecture
- Figure 22 UMSWI architecture [31]
- Figure 23 Layout of cross-development environment [26].
- Figure 24 UMSWI development setup
- Figure 25 root filesystem hierarchy
- Figure 26 Busybox configuration
- Figure 27 Linux kernel configuration
- Figure 28 Booting sequence with *initramfs* [, page 73]
- Figure 29 Modified loader's menu
- Figure 30 */etc/init.d/rcS* system initialization script
- Figure 31 */etc/inittab* file
- Figure 32 */usr/ARMscope/start* script
- Figure 33 UMSWI utilities organization
- Figure 34 Data acquisition and readout design
- Figure 35 Shows how to connect a 16-bit device without byte access on NSC2 []
- Figure 36 Interpretation of NRD/NWR Setup, Pulse Length and NWR/NRD Hold parameters
- Figure 37 Communication Logic flowchart
- Figure 38 FPGA-ARM communication test
- Figure 39 FPGA-ARM communication test
- Figure 40 ARM-FPGA interface
- Figure 41 Finite state machine
- Figure 42 Measured data flow
- Figure 43 Trigger detection process
- Figure 44 File operations structure
- Figure 45 Structure which represents FPGA device.
- Figure 46 Structure storing acquisition parameters
- Figure 47 Function which generates data when */proc/fpga/cmd* file is read

- Figure 48 Implementation of start method in the seq_file interface
- Figure 49 Implementatin of seq_next
- Figure 50 seq_file show method which outputs measurement data to user space
- Figure 51 Seq_operations structure
- Figure 52 File operations structure
- Figure 53 Proc open method
- Figure 54 Implementation of write_proc function
- Figure 55 procfs_register function
- Figure 56 ioctl driver method
- Figure 57 ioctl data structures
- Figure 58 2 words (32-bits) FPGA IO functions
- Figure 59 Using FPGA IO functions
- Figure 60 Implementation of read/write ARM register functions
- Figure 61 Example CGI scripts with a detailed description [31].
- Figure 62 Design of UMSWI web site layout and structure
- Figure 63 MVC implementation design
- Figure 64 Class diagram of Model related classes
- Figure 65 Implementation of HTTP Tunnelling and GET requests
- Figure 66 Forming URL request which sends parameter to the hardware
- Figure 67 Final GUI design
- Figure 68 UML Class Diagram of View-related classes
- Figure 69 UML Diagram describing applets' hardware interfacing []
- Figure 70 UMSWI configuration and management web page layout
- Figure 71 Example Java Script script using CGI
- Figure 72 SCPI command message elements
- Figure 73 SCPI Server design
- Figure 74 Communication layers
- Figure 75 Command structure
- Figure 76 C implementation of SCPI dictionary
- Figure 77 Defining nodes relations and function associations
- Figure 78 Template of function implementing command's logic
- Figure 79 Example SCPI log file
- Figure 80 Explanation of parsing and decoding process
- Figure 81 Debugging FPGA
- Figure 83 Matlab test of SCPI Server
- Figure 84 Test set-up
- Figure 85 First amplitude accuracy test (final_test_1.m)
- Figure 86 Amplitude attenuation for high frequencies (final_test_2.m)
- Figure 87 Amplitude attenuation at 10Mhz for various amplitude values (final_test_3.m)
- Figure 88 Offset error
- Figure 89 Minimal input voltage test at 10 Hz
- Figure 90 Minimal input voltage test at 10 kHz
- Figure 91 Signal frequency and period relative error(final_test_4.m)
- Figure 92 Rising time measurement (final_test_5.m)
- Figure 93 Spectrum analyzer test (final_test_6.m)
- Figure 94 Frequency analysis done with Matlab script (myFFTplot_1.m)
- Figure 95 Frequency analysis conducted with UMSWI Spectrum Analyzer
- Figure 96 Sine and square signal measurement at 10 MHz and 20 MHz
- Figure 97 Input signal exceeding voltage range
- Figure 98 Acceleration of particles with AC voltage radio frequency RF [].

- Figure 99 Four bunches of protons, $h=7$
- Figure 100 Eight protons in bucket, $h=8$
- Figure 101 Bunch splitting
- Figure 102 Single bunch, $h=1$
- Figure 103 All 16 buckets full
- Figure 104 Two buckets filled with bunches of varied proton number
- Figure 106 Acquisition hardware architecture
- Figure 107 CGI process explanation
- Figure 108 Architecture of a generic Linux system [27]
- Figure 109 Benefits of using *uClibc* library [29]
- Figure 110 MVC architecture [60]
- Figure 111. Command message elements
- Figure 112 Relation between sampling rate and bandwidth [63]
- Figure 113 Oscilloscope & Spectrum Analyzer GUI
- Figure 114 Oscilloscope & Spectrum Analyzer Screen
- Figure 115 Oscilloscope's "Ticks"
- Figure 116 Oscilloscope and Spectrum Analyzer's control and auxiliary panel
- Figure 117 MMC/SD card content
- Figure 118 Bootloader's menu

Appendix F – List of Tables

- Table 1 Hardware components of UMSWI
- Table 2 UMSWI development tools
- Table 3 FPGA logic design components according to frequency affiliation
- Table 4 Communication SMC settings
- Table 5 Interpretation of Wait State parameter
- Table 6 Acquisition process
- Table 7 Description of FSM states.
- Table 8 Drivers structure
- Table 9 ioctl/proc interface
- Table 10 SMC configuration
- Table 11 GET requests: `_name_` is the name of hardware parameter
- Table 12 Devices used during tests
- Table 13 Test of Spectrum analyzer
- Table 14 UMSWI parameters
- Table 15. Command message elements

Bibliography

- [1] Linksys by Cisco web site: www.linksysbycisco.com
- [2] Livebox by TP.SA web site: www.tp.pl
- [3] S. Gundavaram, *CGI Programming on the World Wide Web*, First Edition, O'Reilly, 1996
- [4] K. Tatroe, R. Lerdorf, P. MacIntyre, *Programming PHP, 2nd Edition*, O'Reilly, 2006
- [5] Tektronix, *TG700 Remote Control and Connectivity*
- [6] Tektronix web site: <http://www.tek.com>
- [7] Tektronix, *e*Scope Remote Control Puts Network-Connected Oscilloscope on Your PC Desktop*
- [8] Tektronix's e*Scope Server page: <http://connect.tek.com/escope>
- [9] Agilent, *BenchLink Web Remote Control Software for the PSA Series Spectrum Analyzers, ESA-E and ESA-L Series Spectrum Analyzers, E7400A Series EMC Analyzers * Option 230*
- [10] Agilent web site providing BenchLink Web Control Software trial version : <http://wireless.agilent.com/videos/econtent/Remote/>
- [11] BitScope official web site, <http://www.bitscope.com/>
- [12] BitScope, *BitScope 50 Pocket Analyzer*
- [13] PowerPC, <http://www.power.org/home>
- [14] ARM <http://www.arm.com/>
- [15] Infiniium 800 Series Oscilloscopes
- [16] Altera Ltd. <http://www.altera.com/index.jsp>
- [17] Xilinx, <http://www.xilinx.com/>
- [18] ATMEL <http://www.atmel.com/>
- [19] Lattice Semiconductor Corporation, <http://www.latticesemi.com/>
- [20] Creotech Ltd. www.creotech.pl , Indiry Gandhi 35/226, 02-776 Warszawa
- [21] ALTERA, *Cyclone FPGA Family*, 2003
- [22] Analog Devices, 10-bit, 65/80/105 MSPS, 3V A/D Converter
- [23] ISSI, 128K x 32, 128K x 36 Synchronous Pipelined Static RAM, 2004
- [24] TANGO, website: www.tango-controls.org
- [25] Experimental Physics and Industrial Control System, website: www.aps.anl.gov/epics
- [26] C. Hallinan, *Embedded Linux Primer: A practical, Real-World Approach*, Prentice Hall, 2006
- [27] Karim Yaghmour, *Building Embedded Linux Systems*, O'Reilly, 2003
- [28] P.Raghavan, A. Lad, S. Neelakandan, *Embedded Linux System Design and Development*, Auerbach Publications, 2006
- [29] Thomas Petazzoni, Michael Odenacker, *Embedded Linux kernel and driver development*, Free Electrons, 2008, <http://free-electrons.com/>
- [30] J. Corbet, A. Rubini, G. Kroah-Hartman, *Linux Device Drivers, Third Edition*, USA 2005
- [31] Lipinski M. and Kasproicz G. (2009). Universal Measurement System with Web Interface. R.S.Romaniuk, K.S.Kulpa (ed.), *Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2009*; vol.7502. Proc.SPIE, USA
- [32] Busybox Project homepage www.busybox.net/
- [33] Craig Hollabaugh, *Embedded Linux: Hardware, Software, and Interfacing*, Addison Wesley, 2002
- [34] Wookey and Paul Webb, *Guide to ARMLinux for Developers*, Aleph One Ltd., 2002
- [35] Free Electrons Embedded Linux Experts, <http://free-electrons.com/>
- [36] Pelos, *TwARM Atmel AT91RM9200 Eval Board*, www.twarm.pelos.pl
- [37] R. Russell, D. Quinlan, C. Yeoh, *Filesystem Hierarchy Standard*, FHS Group, 2004

- [38] The Linux Kernel Archives, <http://kernel.org>
- [39] AT91 Linux 2.6 Patches, http://maxim.org.za/at91_26.html
- [40] M. Odpenacker, T. Petazzoni, *Embedded Linux kernel usage*, Free Electrons, 2009, <http://free-electrons.com/>
- [41] Das U-Boot – the Universal Boot Loader, <http://www.denx.de/wiki/U-Boot>
- [42] RedBoot Debug and Bootstrap Firmware, <http://www.ecoscentric.com/ecos/redboot.shtml>
- [43] MicroMonitor, <http://microcross.com/html/micromonitor.html>
- [44] Darrel Harmon's Homepage: <http://dlharmon.com/>
- [45] ATMEL, *ARM920T-based Microprocessor: AT91RM9200*
- [46] Eric A. Meyer, *CSS: The Definitive Guide, 3rd Edition*, O'Reilly Media, Inc.
- [47] Development website of UMSWI: <http://home.elka.pw.edu.pl/~mlipins1/myWeb/index.html>
- [48] Tsan-Kuang Lee Sound Spectrum Java Demo, <http://www.ling.upenn.edu/~tklee/Projects/dsp/>
- [49] Customized J2EE training, *Using Applets as Front Ends to Server-Side Programs*, Marty Hall, 2007
- [50] Lipinski M. and Kasprowicz G. (2009), *Control of "Universal Measurement System with Web Interface" as an example of universal embedded system control*, paper and presentation at ICSE2009 Conference, UK
- [51] *Quartus II Handbook Version 9.0*, Altera Corporation, 2009
- [52] MatLab – produced by The MathWorks Company, www.mathworks.com
- [53] European Organization for Nuclear Research, <http://www.cern.ch>
- [54] Shin-ichi Adachi, *Pump-Probe Experiment*, Cheiron 2007
- [55] J. Travis, J. Kring, *LabVIEW for everyone: graphic Programming Made Easy and Fun*, Prentice Hall, 2006
- [56] *VXI-11 TUTORIAL and RPC Programming Guide*, ISC Electronics
- [57] *Standard Instrument Control Library, User's Guide*, Test & Measurement Systems Inc., 2003
- [58] Jason Hunter, *Java Servlet Programming, 2nd Edition*, O'Reilly Media, Inc., 2003
- [59] D. Riekhonf, K. Fligg, *How to Control a Robot Over the Internet*, Sys-Con Media, 2000
- [60] Mark Wutka, *Special Edition Using Java 2 Enterprise Edition*, QUE, 2001, <http://csis.pace.edu/~bergin/mvc/mvcgui.html>
- [61] Stephen Sterling, Olav Masse, *Applied Java Patterns*, Printice Hall, 2001
- [62] Tektronix Prorammer Manual, TDS 200-Series Digital Real-Time Oscilloscope
- [63] Phil Stearns, Sampling rate's impact on oscilloscope bandwidth, Electronic Products, www.electronicproducts.com
- [64] How to run Java Applet in your Web browser, <http://web.cecs.pdx.edu/~ps/CapStone03/dynvis/getplugin.htm>
- [65] UMSWI developer's web page, <http://home.elka.pw.edu.pl/~mlipins1/myWeb/developers.html>