WARSAW UNIVERSITY OF TECHNOLOGY
Electronics and Information Technologies Faculty
Research Group on Biocybernetic Aparatus
Institute of Electronic Systems

Tomasz Bolesław CEDRO
Index number: 188597

Master of Science Diploma Thesis

CeDeROM Brain Computer Interface

Thesis Supervisor:
prof. dr hab. Antoni GRZANKA

November 10, 2011
Warsaw, Poland

CeDeROM Brain Computer Interface
Abstract

Brain Computer Interface is a biomedical equipment used to perform user interaction with computer equipment based on a brain activity mesurement. Modular Research System presented in this document is a prototype aimed for supporting the research groups with versatile and flexible hardware platform for BCI research, but also easy construct and verification of commercial products that can find its use in solving various real life problems.
Modular design consists of control boards based on STM32 ARM–Cortex M3 for mobile applications and FPGA for real time DSP, safe power supply with galvanic separation, general purpose hex 24–bit SigmaDelta ADC, integrated biological signal acquisition frontend with 24–bit SigmaDelta ADC and SPI interface, standard EEG electrodes connector board, user interaction with LED and push–buttons, and finally the demonstration expansion board with computer joystic interface to control external hardware.
System is equipped with USB2.0 FullSpeed Device Unit (ARM) and 1GBit Ethernet Controller (FPGA). System was designed to be a low–cost solution based on Free and/or Open–Source Software. Example usecases are also presented in this document.

Keywords: CeDeROM BCI (Brain Computer Interface), Neural Interface, Biocybernetics, Biomedical, Neurofeedback, Biofeedback, ARM, Cortex, FPGA, ADC, DSP, OpenEEG, Open–Source, FreeBSD, USB, Matlab Driver, BCIOP, Atari, PONG.

Interfejs Mózg–Komputer „CeDeROM BCI”
Streszczenie

Interfejs Mózg–Komputer (Brain Computer Interface) jest urządzeniem biomedycznym nakierowanym na interakcję użytkownika ze sprzętem komputerowym za pomocą pomiaru aktywności elektrycznej mózgu. Modularny System Badawczy zaprezentowany w niniejszym dokumencie to prototyp, którego zadaniem jest wsparcie grup badawczych wszechstronną i elastyczną platformą sprzętową do badań nad BCI, ale również łatwe opracowanie i weryfikacja komercyjnych urządzeń, które mogą być pomocne w rozwiązywaniu problemów z życia codziennego.
Na modułową konstrukcję składa się jednostka sterująca oparta o mikrokontroler STM32 z rdzeniem ARM–Cortex M3 dla zastosowań mobilnych oraz układ Xilinx Spartan FPGA dla przetwarzania sygnałów w czasie rzeczywistym, bezpieczny moduł zasilania z separacją galwaniczną, ośmiokanałowy przetwornik analogowo–cyfrowy 24–bit SigmaDelta, zintegrowany układ pomiaru i akwizycji sygnałów biologicznych z przetwarzaniem SigmaDelta 24–bit oraz interfejsem SPI, standardowy moduł połączeniowy dla elektrod EEG, układ interakcji z użytkownikem w postaci zestawu przycisków i diod świecących, a także demonstracyjna płytka z modułem interfejsu joysticka sterującego zewnętrznym sprzętem komputerowym.
System wyposażony jest w interfejsy komunikacyjne USB2.0 (moduł ARM) oraz 1GBit kontroler Ethernet (moduł FPGA). System został zaprojektowany jako niedrogie rozwiązanie korzystające z wolnego oprogramowania (Free–Software) o swobodnym dostępie do kodu źródłowego (Open–Source). Przykładowe zastosowania przedstawiono w końcowej części dokumentu.

Słowa kluczowe: BCI, Interfejs Mózg Komputer, Interfejs Neuronalny, Biocybernetyka, Inżynieria Biomedyczna, Neurofeedback, Biofeedback, ARM, Cortex, FPGA, ADC, DSP, OpenEEG, Open–Source, FreeBSD, USB, Ethernet, Matlb/Octave/SciLab, BCIOP, Atari, PONG.

To My Family, To My Friends
Thank You! I Love You! :-)

OM MANI PEME HUNG

Foreword

Human organism and its information processing abilities are limited. It is also still very susceptible to cellular malfunction, organic diseases and mechanical injuries. There are situations were even whole parts of the body become inactive when others are still functional. In this case interconnecting biological organism with biomedical equipment is the only way to maintain missing functionality or the human life itself.

What is the purpose of life with no communication ability, especially with other people, family, friends, when we cannot share our knowledge, experiences and ideas. Sometimes the information exchange is impossible because of some dysfunction, sometimes the human body limitations itself are the boundaries of our cognition.

The science–fiction unfortunately is still far from reality – today we cannot visit a harsh environment within a cybernetic avatar, or become a spaceship probe controlled directly with our perception from Earth. This however could be possible with advancement of Brain Computer Interfacing technology serving as a gateway between world of biology and technology.

There are two general types of modern BCI – invasive and noninvasive – depending on the techniqie applied for signal acquisition from the tissue. Noninvasive devices acquire biological signals from outside of the human body. This can be done with use of various tomography methods (such as fMRI), or simply electrodes being sticked with conducting paste on top of the skin reading electrical activity of the inside organ situated below – just like ElectroCardioGraphy (ECG) reads heart's work cycle, or ElectroEncephaloGraphy (EEG) reads electrical activity of a brain. Invasive methods requires electrodes (or more general sensors) to be implanted directly into the organ/tissue, therefore it is impossible in amateur applications, especially in an early stage of academic research. Both approaches requires great amount of interdisciplinary knowledge, well coordinated team of highly skilled enthusiasts, legal support, proper funding and laboratory equipment.

Because high–end equipment is not available for amateur/academic research due to extremely high prices, while free and open solution does not provide minimal functional level, this document presents results of a research aimed at creating inexpensive but versatile modular hardware BCI platform. This work is a proof of concept that such system can become a reality with use of commercial off the shelf components and the Open–Source software tools, some basic laboratory equipment, determination and knowledge.

Research results clearly show that there is still even more to accomplish than already has been done. Some parts of this research are pioneering solutions that allow some other solutions to exist in the first place, but has not yet been done before, especially in open manner. Modular design allows easy system reconfiguration for both scientific research and the commercial implementation. Experience gained during all those experiments allowed to identify and overcome some problems at this stage, but also forced me to leave some tasks for closer inspection in near future. All small failures and successful achievements finds my acceptance and submission because they help me to better realize facts, understand surrounding world, and hopefully produce better results.

Contents

List of Figures

Chapter 1

Know-How

1.1 Reasons, Problems, Solutions.

Having planned tasks that will bring research, or any kind of computer–related project, into an end with no obstacles has become a wishful thinking. Unfortunately there are too much independent variables that makes planning a modern art of risk management rather than precise work navigation along timeline. Especially when advanced solution is created from scratch, there are even more complications and problems to solve, often possible solutions bring even more problems.

This process is hard to understand by the final user, as he/she wants to simply have cheap, working, stable and good looking, preferably one–click solution. Users nowadays already lost insight into system internals, some of them even don't want to know how things work as they tend to think that everything can be already bought (or ordered online). It has become a reality that for most people science or engineering is not much different from black magic. Marketing tricks makes things even worse because they model a happy unaware end–user that can posses years of work of thousands of intellectuals for a dollar, so users start to think they posses all those skills and knowledge as they own its results, never even wondering how it was possible to make it work. Therefore real value and importance of this knowledge is far more valueable and fundamental.. and it has never been only about new products.

The dilema of balancing between time and final cost is usually about what can be bought and integrated into final solution to shorten time–to–market but increase final price. This is also crucial factor for project planning, management and development. There is no need to manufacture single transistors or integrated cirtuit at some point when cheaper and faster application can be found with no significant impact on the income and result, but on the other hand there is no innovation in rebranding existing products. Real innovation begins where challenge is set to do something that never been done before in a way that was previously unknown or unsuccessful. Most manufacturers however exploit unaware users to produce simple low–quality products, enslave them with habits to manipulate future sale with fake innovation.

Independence is very important for me, even at cost of longer project timeline, not

only because of limited resources as those are always too small, mainly because I am not only a model consumer anymore, I want to consciously create my own solutions to share or sale with benefit. This chapter contains technical difficulties I have found during my research and description of invented solutions. Most of them were caused because I have decided to use only *Commercial Off The Shelf (COTS)* elements and Free Open Source Software and avoid repacking existing expensive solutions stating they are mine results.

Sharing experience with other individuals, using pieces of their work for my results, saving resources, gaining new skills, having independent solution with full insight unlimited with restrictive licenses – these are only few reasons why I have chosen this implementation path. Know–How presented in this chapter will also allow estimate work amount necessary to create such simple project from scratch. It may also help estimate real cost of a solution and realize complexity standing behind the scenes.

1.2 Biological Signal Amplifiers

According to [9] biological amplifiers are described by following parameters:

- *Amplifier Gain* is the output voltage level referenced to the voltage acquired by the amplifier circuit on its inputs. Usually the gain value is at least 1000 in linear scale, it can be also measured in decibels (dB):

$$Gain_{lin} = \frac{U_{out}}{U_{in}}$$

$$Gain_{dB} = 20log_{10}Gain_{lin}$$

- *Frequency Response* is a characteristics in frequency domain describing useful bandwidth together with other frequencies and their relation in acquired signal. Useful bandwidth should contain all frequencies present in the electrophysiological signal generated by a tissue or organ of interest, while other frequencies should be limited accordingly to disallow impact of low frequency (f_l) drift and high frequency (f_h) interferences caused by surrounding technical environment on the target signal. Both low and high frequency cutoff is set at 0.707 value of midfrequency plateau, that is half of the useful signal power (because $(0.707)^2 = 0.5$), this is why they are also called $-3dB$ points and the difference between f_h and f_l is the *3dB bandwidth*.

$$-3dB = 20log_{10}0.707$$

- *Common Mode Rejection* is very important parameter in differential (U_d) signal measurement where two electrodes are used for single source measurement to discard common signals (U_c). This method is very useful in biological signals measurement where magnitudes of useful signal are thousands times smaller than external interferences caused by the power network or electronic equipment having similar potential across the human body acting as antenna.

$$CMMR_{lin} = \frac{U_c}{U_d}$$

$$CMMR_{dB} = 20log_{10}CMMR_{lin}$$

- *Noise and Drift* are the additional and unwanted signals in the measurement result produced by the amplifier circuit, electrode–skin contact, etc. Drift refers to signals below $0.1Hz$ and it is a peak–to–peak variation of the baseline, while noise above $0.1Hz$ measured in microvolts peak–to–peak (μV_{p-p}) or microvolts root mean square (μV_{RMS}).

- *Recovery Time* is the time necessary for amplifier circuit return from saturation to normal operation.Saturation can be caused by electrodes contact malfunction, stimulation currents, defibrillation pulses, or any other stimuli that pushes amplifier output voltage to reach the maximum/minimum offset voltage of constant value that prevents device from proper measurement.

- *Input Impedance* of the amplifier circuit should be high enough to prevent attenuation of useful signal produced by a tissue. Each electrode–tissue has its own characteristic impedance related to many factors such as the electrode–skin contact quality, electrolyte substrate and its temperature, are of the electrode, tissues between source tissue and the electrode, etc. The average impedance values are in range of 200kΩ for 1Hz and 200Ω for 1MHz.

- *Electrode Polarization* is the *half–cell polarization* caused by ion–electron exchange between metal electrode and electrolyte paste or simply skin perspiration that result in a DC component in a measured signal. Association for the Advancement of Medical Instrumentation (AAMI) specify that ECG equipment should tolerate $\pm 300mV$ DC component resulting from the electrode–skin contact.

There are many more different configurations of biological signals amplifiers designs and theory with practical examples presented in [9] therefore it is highly advised to get familiar with this great book, as there is no sense to rewrite its contents in this small document.

1.3 System Configuration and Methodology

Each area of life seems to have a trend that leads the (research) directions in a defined period of time until something better is invented and become a new trend. This includes electronic design that nowadays tend to shift from analog design into digital domain with analog circuity reduced to absolute minimum. This allows better miniaturization, lower system cost, higher availability with smaller POF (Point Of Failure), better scalability and system reconfiguration in future without total redesign now replaced with firmware upgrade. Personally I don't think such blind trends–follower attitude is glorifying, but in this particular case it is very close to my intuition.

Biomedical equipment also use this „digitalization" trend to minimize analog components amount in the design and use better digital components that provide equivalent

or better capabilities when implemented as software algorithms in programmable logic. *BurrBrown* company, now owned by *Texas Instruments*, leading world class manufacturer of components for biomedical instrumentation has released a series of documents and devices to pioneer these new fields of biomedical technologies. Analog filtering is replaced by DSP (Digital Signal Processing), analog components are replaced by high–end digital solutions, devices become more and more integrated.

I have joined this trend not only because I have better experience in digital electronics design, but also in my opinion they are more agile and give better flexibility in commercial application of different services and solutions. It is always possible to hire someone highly–skilled that will design the subsystem component, but the big–picture is more important for me – to create working solution. I have presented below some documents found in the knowledge base, they cover various aspects of modern design, they are available for free download on the BurrBrown/Texas Intruments website and I will place their contents in the Appendix. There are whole books being written on how to build analog amplifiers. I will use those information and results as modules for creating my solution instead. Documents presented below are also very important and valuable source of information for any designer of biocybernetic instrumentation:

- „Improving Common-Mode Rejection Using the Right-Leg Drive Amplifer" [10]
- „High Speed Data Conversion" [11]
- „Analog-to-Digital Converter Grounding Practices Affect System Performance" [12]
- „Interleaving Analog-to-Digital Converters" [13]
- „Thermal Noise Analysis in ECG Applications" [14]
- „Principles of Data Acquisition and Conversion", [15]
- „Analog Front-End Design for ECG Systems Using Delta-Sigma ADCs" [16]
- „A Glossary of Analog-to-Digital Specifications and Performance Characteristics" [17]
- „What Designers Should Know About Data Converter Drift" [18]
- „Power Management for Precision Analog" [19]

My solution will be based on high–resolution analog–to–digital conversion with minimal analog components amount. One usecase contain fully integrated device for biological signals acquisition (ADS1298 chip), another usecase contain general purpose high–resolution analog–to–digital converter that can be attached to any compatible analog frontend. For more information please follow the „Solution Approach" (Chapter 2.1).

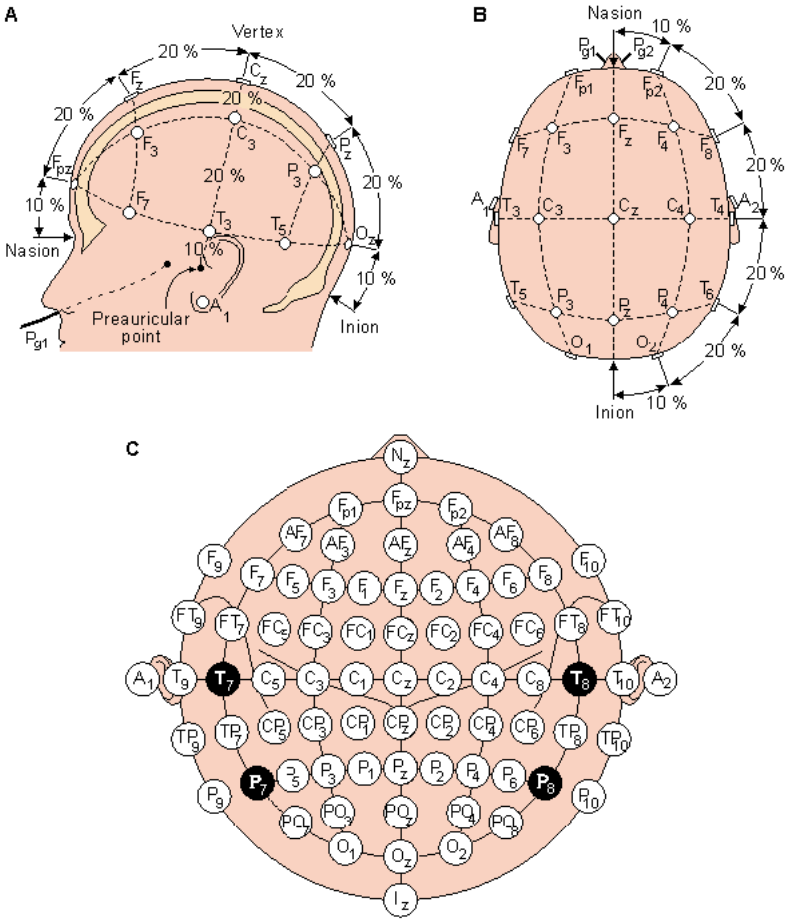


Figure 1.1: 10–20 EEG Electrode Placement Diagram [85].

1.4 EEG 10–20 system

The *10–20 system* is a standard introduced to describe electrodes placement on top of the scalp (head skin) during the EEG measurement. This is important to reproduce and compare measurements over time on different subjects, because selected test points mark regions of underlying cerebral cortex that produces the internal signal received by the electrodes. The name 10–20 actually comes from the distance parts in left–right (10%) front–back (20%) length of the skull (Figure 1.1).

As described in [22] each site has a letter to identify the lobe and a number to identify the hemisphere location. The letters F, T, C, P and O stand for Frontal, Temporal, Central, Parietal, and Occipital, respectively. Note that there exists no central lobe, the "C" letter is only used for identification purposes only. A "z" (zero) refers to an electrode placed on the midline. Even numbers (2,4,6,8) refer to electrode positions on the right hemisphere, whereas odd numbers (1,3,5,7) refer to those on the left hemisphere. Two anatomical landmarks are used for the essential positioning of the EEG electrodes: first, the nasion which is the point between the forehead and the nose; second, the inion which

(a) NeuroSky (b) Emotiv

Figure 1.2: Commercial low–cost BCI devices for home use.

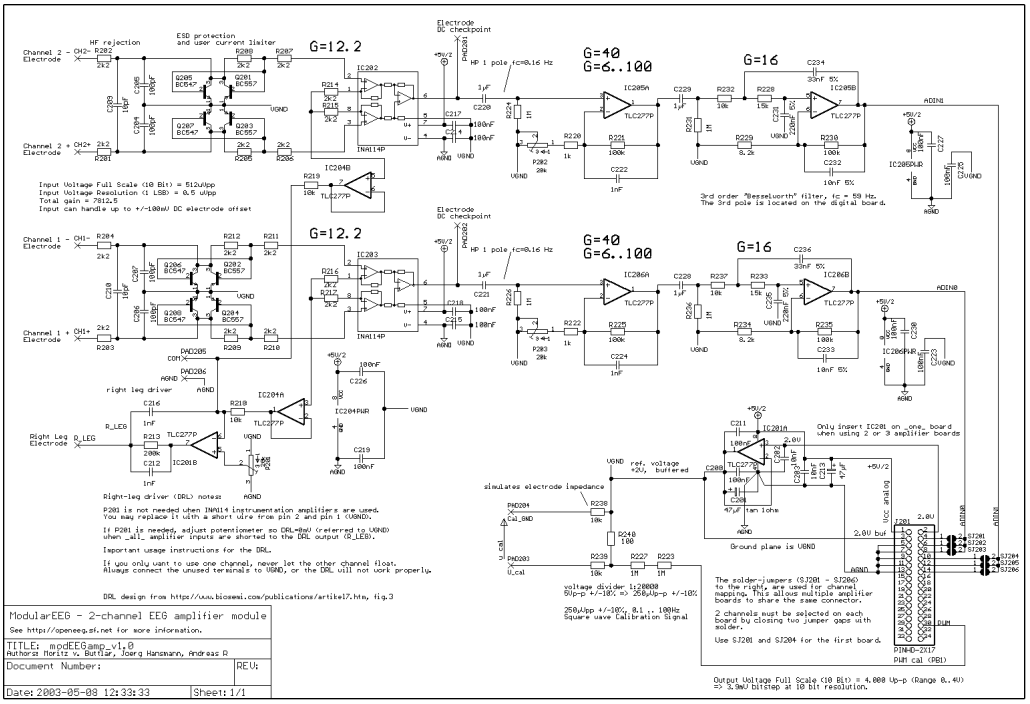
is the lowest point of the skull from the back of the head and is normally indicated by a prominent bump. When recording a more detailed EEG with more electrodes, extra electrodes are added utilizing the spaces in-between the existing 10-20 system. This new electrode-naming-system is more complicated giving rise to the Modified Combinatorial Nomenclature (MCN). This MCN system uses 1, 3, 5, 7, 9 for the left hemisphere which represents 10%, 20%, 30%, 40%, 50% of the inion-to-nasion distance respectively. The introduction of extra letters allows the naming of extra electrode sites. Note that these new letters do not necessarily refer to an area on the underlying cerebral cortex.

1.5 OpenEEG and other BCI Platforms

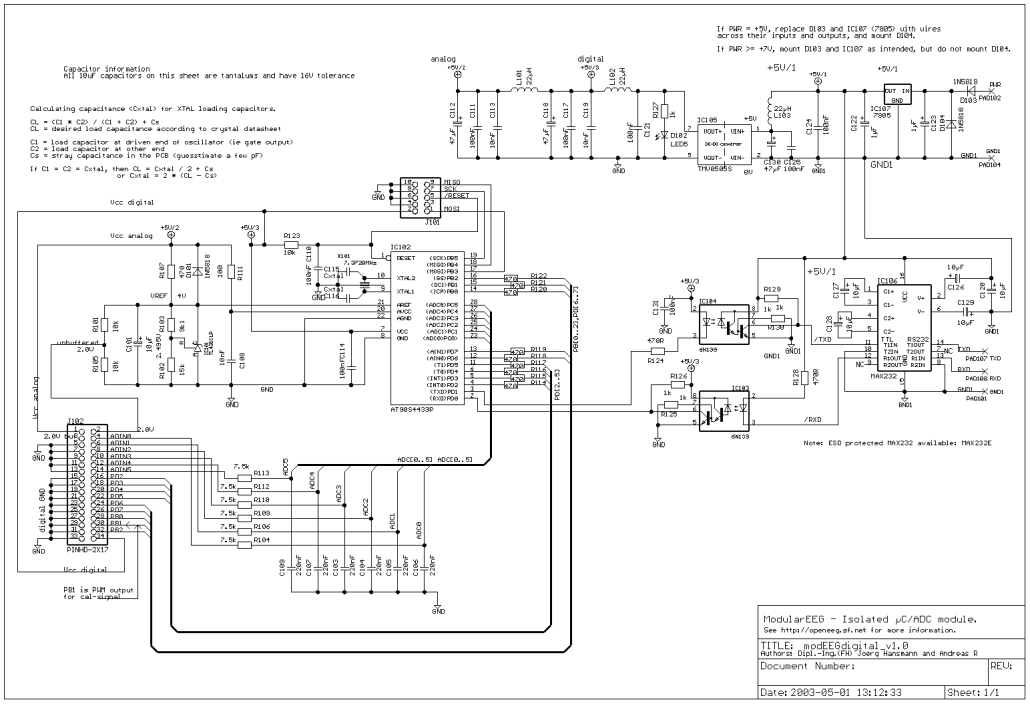
OpenEEG [84] is the most popular low–cost platform for amateur BCI/Neurofeedback research with fully open schematics (Figure 1.3) and internal support available to the community that gathered around this project for the past 8 years. There are some disadvantages of the platform. Hardware design is already outdated as it use 8–bit microcontroller with 8–bit ADC and RS232 for data transmission (modern computrs does not have this port even available). Analog amplifier is not equipped with 50/60Hz notch filter so the power interferences are sometimes blocking the measurement, or the device needs voodoo style shielding. This is however the reference platform for many other designs, including one presented in this doument.

Some alternative to the OpenEEG system may be provided by the commercial BCI devices from NeuroSky [88] or Emotiv [89] companies. Because they are commercial and closed source they seem to be poor research equpiment with their limited and non–customizable design. Commercial high–end equipment is provided by gTec.at [90] for well funded BCI research groups around the world to have standarized working environment.

There are several applications that can work with OpenEEG system, including OpenViBE [86] for BCI interaction in Virtual–Reality environment, BCI2000 [85] well known BCI framework, BrainBay [87] and many more. One of the biggest disadvantages of these programs however is the fact that they work only on Windows, some of them are not



(a) Analog Part Schematics



(b) Digital Part Schematics

Figure 1.3: OpenEEG Schematics.

open–source so it is not possible to fix bugs in relatively old releases, some of them are restricted by licensing constraints. These programs however are the best (often also the only) choice home or academic user can get to experiment with BCI systems.

1.6 FreeBSD – Operating System of a choice

There are many computer operating systems on the market. Some of them free, some commercial, all protected with some kind of licensing agreement. Each one of them have some advantages and features making is usable more in some specific applications than the others. The system of a choice for my research is a well known Berkeley Software Distribution Unix Operating system named FreeBSD [34]. It is free, open source, compatible with most open source applications, and it has very simple license that allows closing final application. The FreeBSD can be considered a distribution of BSD system, just like OpenBSD, NetBSD and others. Commercial example of BSD application is very popular Apple Mac OS X for desktop machines and iPhoneOS for mobile machines, also lots of network and embedded equipment use this operating system for closed source applications.

Unlike Linux, FreeBSD is better organized and more self–compatible with other installations (one command or configuration file will work among all FreeBSD installations, what can't be said about Linux distributions, as each one of them is different, even among releases of one distribution, almost each new kernel release breaks existing drivers, etc). This is very important to have stable and independent working environment, especially in long term projects, where its configuration and internals does not make your work obsolete after few weeks or constantly require additional fees. Maintenance is very important part of the project life cycle. Microsoft exactly knows that and offers easy to configure Windows platform with multiple development tools. All those tools allow rapid development with no need to know details of implementation, but they are also commercial, relatively expensive and low quality, they require constant upgrades to non–backward compatible formats, so user ends up constantly spending money for something that is a black–box and cannot be expanded or customized. Other operating systems are usually dedicated for backend specific tasks, therefore require dedicated hardware platform what makes them unavailable and often useless for average computer user.

FreeBSD is free and less resource hungry, so it can run even on simple embedded systems, also does not double price of the final product. Its license does not put any constrains on application and the source code distribution. Although well organized structure and stable standard, the main disadvantage of the FreeBSD STABLE line is the lag behind modern multimedia features available only to the commercial systems. Also drivers for new gadgets and hardware are hard to find or simply waiting to be written. On the other hand this non–bleeding–edge development line is what makes it stable and straightforward solution. People that want to bring new features should consider developing CURRENT branch, where like in Linux, things change day by day and are not guaranteed to work or maintain backward compatibility. After new features are well tested and comply internal standards, they are (then and only then) integrated

into STABLE line.

FreeBSD has proven to be most innovative and well organized solution for many years being ahead of competitors in some areas even driving development of new features for many other operating systems including Linux and Windows. Its very logical organization, maybe even raw in its simplicity, makes this system good candidate for stable working environment that can be controlled with a simple set of text configuration files and standard Unix/POSIX environment and utilities. This system does not hide anything from user in a form of invisible automation unless explictly asked to do so. It is also free of external dependencies, even on Free Software, so neither deveopers nor users are forced to buy upgrade or pay additional money for trivial additional features. There are also nice features available uncommon to any other operating systems, for instance binary emulation of other operating systems including BSD and Linux allowing for transparent binary runtime execution. Therefore all futher chapters will assume FreeBSD operating system was used unless stated otherwise.

1.7 GNU ARM Toolchain

GNU ARM Toolchain [25] is a free of charge and Open Source set of tools to develop software that will run on ARM machine. The toolchain consists of the GNU binutils, compiler set (GCC) and debugger. Newlib is used for the C library. The toolchain includes the C and C++ compilers. It comes fully documented – with online books for developers, and system man pages for each of the tools provided. Users familiar with Unix or Unix-like operating systems will have no problem with GNU ARM tools, as they are the same as the ones used to build programs on x86, x86_64, mips, and many other platforms:

- arm-elf-gcc – is a GNU compiler of a C programming language. Object files produced by this program contains binary code executed by ARM microprocessor.
- arm-elf-as – is an GNU assemmbly language compiler. It uses AT&T UNIX syntax (different than Intel) to parse input and produces binary code of almost any microprocessor architecture.
- arm-elf-ld – is a GNU Linker program that produces final executable or even whole system image from a separate object files. This program requires special configuration file (**.ld*) with memory map (rom+flash+ram) of a target ARM system to produce **.hex* or **.bin* image matching configuration of a specific silicon chip.
- arm-elf-gdb – is a GNU Debugger. This tool allows watching target program execution instruction–after–instruction and helps finding bugs in code. This program also allows remote debug of a physical target plaform connected via JTAG interface to the host running debug daemon software (ie. OpenOCD). Remote debug can be performed with use of TCP/IP network, so the user is not limited to the direct neighbourhood of the device.

- gnu make – this program is not a part of the GNU ARM Toolchain, but is common GNU tool available on most Unix or Unix-like environments. This program controls the generation of executables and other non-source files of a program from the program's source files. It is indispensable program that automates build process of a final executable code or system image. It reads instructions, on how to react on command line parameters, from a special configuration file called Makefile.

There are also few others commercial Software Development Kits (SDK) for ARM family, but I will not discuss them here, as they are expensive and windows platform dependent in most cases. GNU SDK is a standard set of tools to develop software in a free and platform independent way. Also system requirements for this software is far beyond expectations of the modern commercial toolkits. Once the developer learn how to work with GNU tools, it will be no difference whether he/she writes software for a computer or embedded system, and what hardware platrofm is inside – it will be only a matter of few configuration files and program switches. What is more, the whole development process takes place in a shell (command line) environment, so it can be performed remotely via simple terminal program, from any place linked to an internet. For those who prefer graphical and windowed development environment, there is a special release called YAGARTO [27] to work with Eclipse Integrated Desktop Environment (IDE) [26].

All this is for free, so the real price goes for knowledge and skills of a user. This fact is really worth appreciation to the developers of the GNU Project [24] and The Free Software Foundation [23].

1.8 Free Real Time Operating System

1.8.1 Introduction

FreeRTOS is a free real-time operating system ported to many different hardware platforms, targeted to use with embedded systems. FreeRTOS gives developer ability to create multitasking environment with multiple tasks or co-routines that can be sorted by priority, sharing resources by queues, synchronising by semaphore and mutex technique. Within its small footprint (approximately 25kB) there are both aviable simple dynamic memory allocation (malloc) and Interrupt Servicing Routine (ISR) mechanisms. Please refer for project homepage `http://www.freertos.org` for detailed information.

1.8.2 API Fundamentals

The kernel is the core component within an operating system. Operating systems such as Linux employ kernels that allow users access to the computer seemingly simultaneously. Multiple users can execute multiple programs apparently concurrently. Each executing program is a task under control of the operating system. If an operating system can execute multiple tasks in this manner it is said to be multitasking. The use of a multitasking operating system can simplify the design of what would otherwise be a complex software application:

- The multitasking and inter-task communications features of the operating system allow the complex application to be partitioned into a set of smaller and more manageable tasks.
- The partitioning can result in easier software testing, work breakdown within teams, and code reuse.
- Complex timing and sequencing details can be removed from the application code and become the responsibility of the operating system.

1.8.2.1 Task

FreeRTOS versions prior to V4.0.0 allow a real time application to be structured as a set of autonomous 'tasks' only. This is the traditional model used by an RTOS scheduler.

A real time application that uses an RTOS can be structured as a set of independent tasks. Each task executes within its own context with no coincidental dependency on other tasks within the system or the scheduler itself. Only one task within the application can be executing at any point in time and the real time scheduler is responsible for deciding which task this should be. The scheduler may therefore repeatedly start and stop each task (swap each task in and out) as the application executes. As a task has no knowledge of the scheduler activity it is the responsibility of the real time scheduler to ensure that the processor context (register values, stack contents, etc) when a task is swapped in is exactly that as when the same task was swapped out. To achieve this each task is provided with its own stack. When the task is swapped out the execution context is saved to the stack of that task so it can also be exactly restored when the same task is later swapped back in.

Task Summary:

- Simple.
- No restrictions on use.
- Supports full preemption.
- Fully prioritised.
- Each task maintains its own stack resulting in higher RAM usage.
- Re-entrancy must be carefully considered if using preemption.

1.8.2.2 Co-routine

FreeRTOS version V4.0.0 onwards allows a real time application to optionally include co-routines as well as, or instead of, tasks. Co-routines are conceptually similar to tasks but have the following fundamental differences (elaborated further on the co-routine documentation page): Stack usage. All the co-routines within an application share a single

stack. This greatly reduces the amount of RAM required compared to a similar application written using tasks. Co-routines use prioritised cooperative scheduling with respect to other co-routines, but can be included in an application that uses preemptive tasks. The co-routine implementation is provided through a set of macros. The reduction in RAM usage comes at the cost of some stringent restrictions in how co-routines can be structured.

Co-Routine Summary:

- Sharing a stack between co-routines results in much lower RAM usage.
- Cooperative operation makes re-entrancy less of an issue.
- Very portable across architectures.
- Fully prioritised relative to other co-routines, but can always be preempted by tasks if the two are mixed.
- Lack of stack requires special consideration.
- Restrictions on where API calls can be made.
- Co-operative operation only amongst co-routines themselves.

1.8.2.3 API Selection

Only those API functions specifically designated for use from within an ISR (Interrupt Service Routine) should be used from within an ISR, as ISR stands for Interrupt Service Routine and is designed only to serve IRQ. Tasks and co-routines use different API functions to access queues. A queue cannot be used to communicate between a task and a co-routine or visa versa. Intertask communication can be achieved using the full featured API functions, the alternative API functions, and the light weight API functions (those with "FromISR" in their name). Use of the light weight functions outside of an ISR requires special consideration, as described under the heading "Performance tips and tricks - using the light weight API". For more detailed information please browse the Documentation at `http://www.freertos.org`.

1.9 Universal Serial Bus

1.9.1 Introduction

USB stands for Universal Serial Bus, a serial interface to exchange data between computer and peripherials. It uses Plug'n'Play Host-Device architecture, where Host is responsible for all data manipulation on the bus, device identification and driver load/unload, while Device can receive or send data when asked by the Host. Device can be dynamicaly connected and disconected without need to turn off or restart computer. Devices are

identified by PID and VID number pair (Product and Vendor ID) along with Identifier String. There are many different Transfer Modes to fit needs of a specific project. USB Device can have more than only one interface using common physical connection.

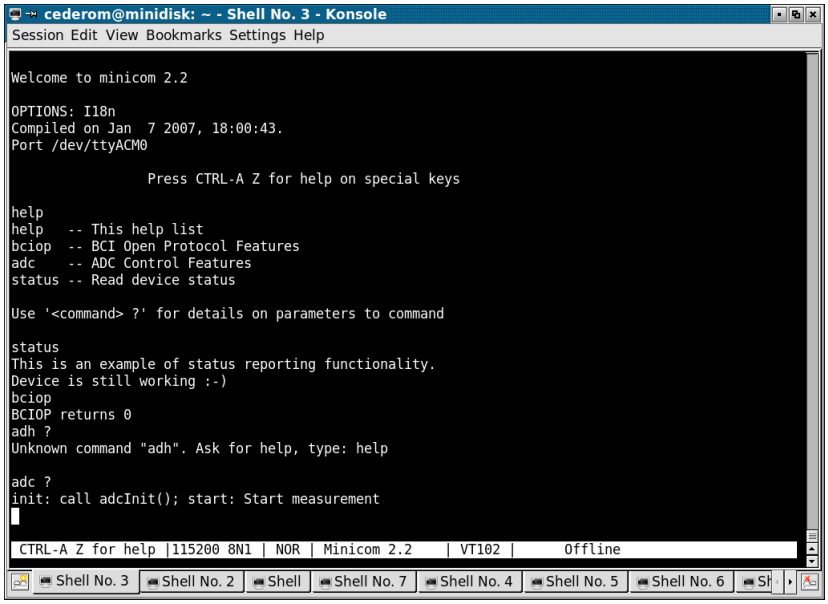


Figure 1.4: Command Line Interface (CLI) implemented on LPC2148 ARM–based microcontroller with built–in USB Device Controller using OpenSource programs.

This chapter is just an overview of the USB Bus features, to get more detailed information, but not yet reading full specification that is really really big, I highly recommend reading „USB in a NutShell" [33] freely available e–book.

1.9.2 Standards

Current version of USB standard is 3.0, but it is not yet widely deployed in computer hardware, so we will still use version 2.0, the succesor of the USB 1.0 and 1.1 specification [32]. It uses EHCI (Encanced Host Controller Interface) specification. There can be up to 127 devices on the bus, connected one Port via HUB (star architecture). HUB can perform current mesurement and dynamic load/unload devices that interrupts the BUS. Devices connected this way shares bandwith of the Port. If more bandwith or devices are to be connected, Host with more Ports should be used.

There are two types of connectors used in USB: A for upstream/Host, B for downstream/Device. Data Cable consists of four signals: 5V, D+, D-, GND. Data is transferred differentially on D+ and D- lines, using NRZI (Non Return to Zero Inverted) and bit stuffing. D+ and D- lines are also used as control lines (nondifferential) at reset and enumeration phase, just after connecting Device to the Host.

USB 2.0 Device can support one or two of three speed modes:

- Low Speed 1.5MBit/s, accuracy 1.5 % or 15000ppm

- Full Speed 12MBit/s, accuracy 0.25 % or 2500ppm
- High Speed 480MBit/s, accuracy 500ppm

1.9.3 Physical Signalling

USB trasnmitter defines differential '1' by pulling D+ over 2.8V with 15k resistor tied to ground, and pulling D- under 0.3V with 1.5k resistor tied to 3.6V. Logical '0' is inversion of logical '1'. The receiver identifies '1' when D+ is 200mV higher than D- line, and '0' when D- line is 200mV above D+. The polarity of the signals might be inverted depending on the current Speed Mode - so called 'J' state for Low Speed is represented by differential '0', while it is differential '1' in High Speed mode.

It is important to select proper series resistors for impedance matching for D+ and D- lines. Low and Full Speed Modes has characteristic impedance of 90ohms with 15% accuracy. High Speed also uses 17.78mA constant current for signalling to reduce noise.

Device indicates its speed mode by pulling high (3.3V via 1.5k resistor) appropriate data line: D+ to mark Full/High Speed or D- to mark Low Speed. High and Full Speed modes are selected the same way electrically, but the reset and enumeration phase selects proper protocol.

Device can support only Low and Full Speed modes and no High Speed mode, to make product cheaper. When device supports High Speed then it also support Full Speed but no Low Speed.

1.9.4 Power Management

USB Devices can be powered from within the Host requiring no externel power supply. Device should know how much current it will consume and give this value to the Host during Enumeration Phase in 2mA quants. USB also specifies Unit Load as 100mA. This divides Devices into three categories:

1. Low-Power: can draw only one unit load, and must work within 4.4V to 5.25V range measured on upstream plug
2. High-Power: can draw maximum 5 unit load (500mA) after enumeration phase but 1 unit load before enumeration, and Vbus ranging from 4.75V to 5.25V.
3. Self-Powered: may draw 1 unit load from the bus to allow detection without external power turned on; must have external power supply to operate.

No device can drive Vbus - after Host turns off Vbus, Device must deactivate pullups on the D+ and D- lines that are used for speed identification.

Device must support Suspend Mode. Global Suspend will work for each Device on the bus, while Selective Suspend can work only for selected ones. One unit load stands for 500uA suspend current. Attention should be taken that pullup resistors sinks 200uA all the time.

Device should put itself in Suspend Mode when there is no activity in the bus for 3 miliseconds and has 7 ms to acomplish this task. After 10ms of silence on the bus, the Device should not sink more than designated Suspend Current, but the Host can send special kepalive packets to the Device to avoid suspend on the bus with no data:

- High Speed: micro-frames sent every 125us +-62.5ns
- Full Speed: frames sent every 1ms +-500ns
- Low Speed: EOP (End of Packet) sent every 1ms in case of data absence on the bus

Device should resume operation after receiving any non idle signaling, and should report to the Host if it has been resumed from any other reason.

1.9.5 USB Procotol

USB is more complicated than standard UART where pure bytes of data are being sent and received transparently one after another. USB consists of advanced logical structure that determines many different functions that can be implemented in a single device. That is why there must be an USB Stack implemented in a Software, supported by a Hardware Block that performs low level transactions and timings conforming to the USB Bus standard [32].

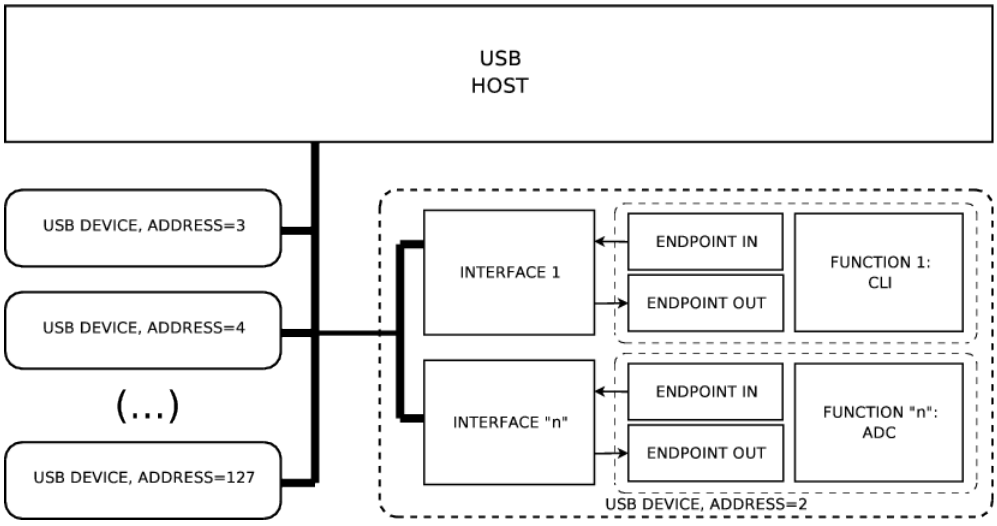


Figure 1.5: Host centric USB bus organisation

Transfer with a Device is always initiated by a Host and the „IN" transfer direction is always directed towards Host. „OUT" transfer means „from the Host to the Device", while „IN" transfer means „from the Device to the Host". Each transaction consists of three parts of data:

- Token Packet – says what kind of transfer is being performed (IN, OUT, SETUP)

- Data Packet – is optional packet containing the payload
- Status Packet – provides acknowgledge or error correction data (ACK, NAK, STALL)

There can be as many as 128 devices connected to a single bus in a slave manner – that is Device cannot transmit anything itself. If a specific Function of the Device wants to transmit or receive any data it must use „End Point" (EP) as a kind of transmit buffer and wait for the Stack/Hardware and Host to perform Bus Transaction. But to use and End Point, the Device Stack has to be initialised and setup with proper parameters (Transfer Mode, Interface number, buffer length, etc) described by a set of an USB Descriptors.

1.9.6 USB Transfer Modes

1.9.6.1 Interrupt Transfer

Works similar to an interrupt on the computer – when the USB Device needs to report some activity it waits until being polled by a Host, and then transfers some small amount of data. It guarantees small latency, it is bidirectional (Stream Pipe) and supports error detection and retransfer in next period. The maximum data payload is

- 8 bytes for Low Speed
- 64 bytes for Full Speed
- 1024 bytes for High Speed

1.9.6.2 Isochronous Transfer

This mode can be used to transfer data periodicaly or continously in time sensitive applications like audio or video streaming. No delivery or retry is guaranted, and error correction is provided only by CRC, although USB bandwidth is guaranted with bounded latency. This transfer mode is unidirectional using Stream Pipe. Maximum data packet size is 1024 bytes for High Speed and 1023 for Full Speed mode. Low Speed mode does not support this transfer mode. Also there is no handshake (no Handshake Packet), so errors and STALL/HALT cannot be reported.

1.9.6.3 Bulk Transfer

This mode can be used to transfer bursts of data that do not need low latency. Because bulk transfer mode uses unallocated bandwidth of the USB Bus, it should not be used in time sensitive applications, as it will have to wait for others isonchronous and interrupt transfers with preallocated bandwidth. This mode is Unidirectional using Stream Pipes, and supports error detection based on CRC and retransmission if necessary.

1.9.7 Bandwidth Mangement

It is possible to allocate bandwith for an USB Device at enumeration stage of the Isonchronous and Interrupt Endpoints. Host is then responsible for managing the requested bandwidth. Maximum 90% for Full Speed Bus and 80% for High Speed Bus can be selected. The rest is reserved for Control and Bulk Transfers.

1.9.8 USB Descriptors

Descriptors are defined by the software as a part of the logical structure of the USB Device. Besides dedicated silicon hardware that performs bit–level communication with the Host, Device Descriptors are the most important part of the Device itself, because they define all of its internal organisation – name, identification and manufacturer, bus version support, available configurations, internal interfaces, transfer modes, buffer lengths, and many more.

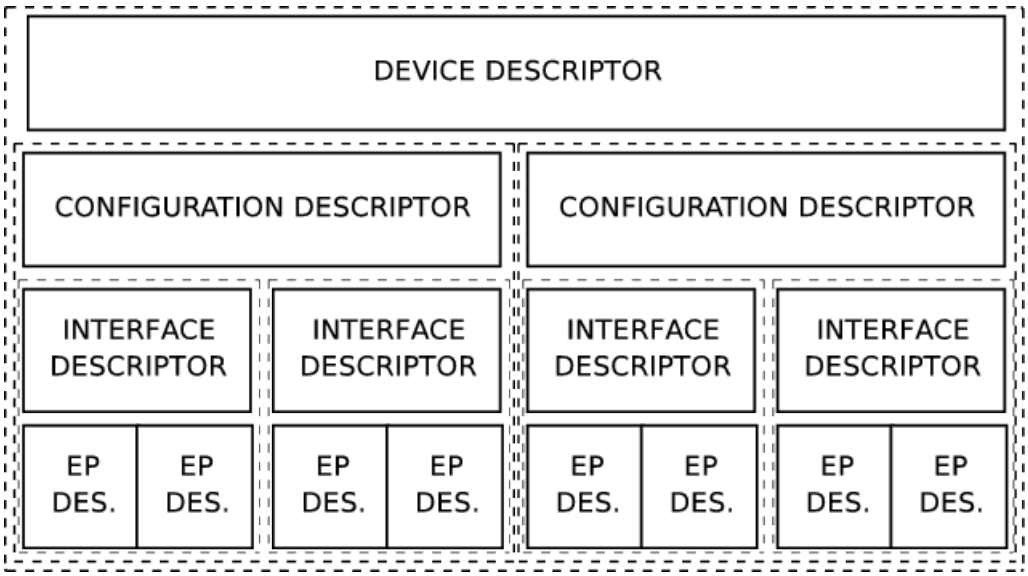


Figure 1.6: USB Device Descriptors organisation diagram

Descriptors perform following functions:

- Device Descriptor – it contains information such as USB Bus compliance, Product and Vendor ID (PID/VID pair to recognise device and load apropriate drivers), and number of possible configurations.
- Configuration Descriptor – tells how much power does the current configuration requires and how many interfaces it has. One device can have many different configurations. At the enumeration stage host can iterate all of these possible configurations (maximum amount is defined by the Device Descriptor) to choose one that is possible to achieve or selected by the end user. Only one configuration can be choosen at a time.

- Interface Descriptor – groups endpoints (EP) based on a specific function of the USB Device. Interface can be described as one of many logical devices inside a one hardware, because one USB Device can perform many different functions at the same time.
- EndPoint Descriptor – specifies type of transfer, direction, packet size, polling interval. EndPoint is a source or sink of data. Endpoint number zero is a control endpoint and has no descriptor.
- String Descriptor – is optional and contains human readable information encoded in Unicode format. Strings can contain multilanguage contents.

1.10 Device Drivers in Matlab

1.10.1 Introduction

Designing advanced data acquisition systems requires whole set of methods to verify a prototype and the data that it produces. This can be achieved by use of Matlab [29] , or free Matlab-like software (ie. Octave [30], SciLab [31], or similar). With its wide spread of signal spectrum analysis, filter design and data modelling techniques, this is perfect environment for research related activities. Data can be transferred from device hardware into matlab environment by use of commercial Data Acquisition Toolbox and standard data bus, or design specific dedicated data bus and self written device driver. This section describes how to create a dedicated device driver in Matlab, from sctratch.

1.10.2 How Matlab handles execution

Matlab can generate binary executables basing on its standard m-files (Compiler Toolbox), but can also use external binaries or dynamic libraries to perform required operations. We can use these features to write our own driver. Please read matlab manual page named „External Interfaces" for detailed information. Input-Output routines and cooperation with external applications are available thanks to:

- dynamic libraries (so/dll) support
- external C of Fortran procedure calls within MEX file
- creating MEX files in C or Fortran language
- result data import/export with MAT files
- including Matlab code fragments in C or Java applications

Device Driver is usualy a set of functions performing device configuration and data transfer. In Matlab environment this can be implemented as set of MEX files or standard dynamic library. MEX files are Matlab specific, can be written in C or Fortran language,

and they are easier in later use. Dynamic libraries can be written in any programming language, used by any other application, and they are just a bit harder in later use with Matlab. Dynamic libraries are preffered method in this document. Please take a look at *„MATLAB Interface to Generic DLLs"* Matlab Manual section for detailed information.

1.10.3 Dynamic Libraries Matlab Intefrace

Dynamic library is a kind of container that holds many functions in a predefined manner. There are few specific initialisation functions and the others can operate on data passed in as pointers. Because all library contents can be dynamicaly loaded at runtime, there can be many functions with the same name grouped in a different libraries performing different functions. This is how the plugins work - required content is loaded into runtime memory only when its needed.

Functions included in a dynamic library can be loaded into Matlab runtime memory and become accesible directly from the interpreter commadline. In most cases data cast is automatic and Matlab types are preffered. Dynamic Libraries can also be written in languages other than C, but the library interface must conform to the C dynamic library standard.

1.10.4 Using Dynamic Libraries

1.10.4.1 Opening the library

To acces functions included in a dynamic library, use `loadlibrary` routine:

```
loadlibrary('library_name', 'header_filename')
```

where:

- `library_name` – is a dynamic library filename (`*.so` or `*.dll`)
- `header_filename` – is a header filename (`*.h`) with all of the function names and data types used by the dynamic library

1.10.4.2 Closing the library

To close a dynamic library that is already open, use `unloadlibrary` routine:

```
unloadlibrary library_name
```

1.10.4.3 Browsing the library

To display a dynamic library contents, use `libfunctions` or `libfunctionsview` routines:

```
libfunctions('library_name')
libfunctionsview('library_name')
```

where:

- `libfunctions` – returns string array with function names from selected library
- `libfunctionsview` – function names from selected library are displayed as table in a separate window

Both functions can use `-full` switch to display additional functions information (ie. parameter list, data types, etc.)

1.10.4.4 Calling the library functions

To call a function that is included in a opened dynamic library, use `callib` routine:

```
calllib('library_name', 'function_name', arg1, ..., argN)
```

where:

- `library_name` – is an opened dynamic library
- `function_name` – is a function that we want to call
- `arg1,..,argN` – is a function argument list

1.10.5 Example

As an example we want to use three functions included in one dynamic library. Two of them will return string (character array) and the last one return sum of passed arguments.

The header file `test.h`:

```
1 char* test();
2 char* test2();
3 int test_add(int a, int b);
```

The source file `test.c`:

```
1  #include "test.h"
2  char* test(){
3          return "test function 1 result\n";
4  }
5  char* test2(){
6          return "test function 2 result\n";
7  }
8  int test_add(int a, int b){
9          return a+b;
10 }
```

Building the library with GNU C Compiler:

```
1 gcc -shared -o test.so test.c
```

Now the Matlab part:

```
>> cd path_to_our_library
>> loadlibrary test.so test.h
>> calllib('test', 'test')
ans =
test function 1 result
>> calllib('test', 'test2')
ans =
test function 2 result
>> calllib('test', 'test_add')
??? Error using ==> calllib
No method with matching signature.
>> calllib('test', 'test_add', 1, 2)
ans =
     3
>> unloadlibrary test
```

1.11 Serial Wire Debug

Serial Wire Debug [43] is a new low–level embedded system access introduced by ARM Corporation [42] in their new CPU design `ARMv7` named `Cortex`. It is compliant to ARM Debug Interface version 5 [44] that specifies all requirements and capabilities of this transport. We will call it transport, because its purpose is to transport commands between debug software on the host computer and debug port on the target system, just as JTAG does, but in a different fashion. JTAG use state machine design, while SWD use packed–based half duplex serial link with lower pin count than JTAG.

ARM is a company that designs CPU and license that design as IP (Intellectual Property) to a silicon manufacturer company that puts some additional peripherials around and sell this as a physical chip for a device of some kind. `ARM Cortex` devices appeared on the market around year 2008 bringing new quality to mobile world, somehow synchronized with Google Android OS entrance to the market. In year 2011 we have `Cortex-A8` family dominating the market with commercially available product and this year `Cortex-A9` comes to light with multicore CPU support. There are plans for `Cortex-A15` in 2012 having even more computational power, more cores, multimedia peripherials, etc. `Cortex` family is on its way to the top, being manufactured by silicon gigants such as Samsung [49], Qualcomm [50], Texas Instruments [51], ST Microelectonics [52] and others. It will soon fill the growing market of tablet, laptop, set–top–box and other embedded devices.

Having already insight and tools to work with this new Serial Wire Debug transport gives an apportunity to work with upcoming devices in near future and gain skills on already existing ones. Creating development tools before developers can have it is also great benefit. The LibSWD I have created is the **first in the world open implementation** of the Serial Wire Debug Open Framework already integrated with UrJTAG [37] and OpenOCD [36] low–level embedded systems access software utilities.

There are still a lot of things to do in order to access other, deeper, parts of the CPU and trace/debug facilities, but the first step has been done – recognising the transport mechanism logical and physical construction of Serial Wire Debug, creating LibSWD [38] and integrating it with existing software utilities, attracting the community around the subject for further development stimulation.

1.11.1 Serial Wire Debug Technical Reference

1.11.1.1 Introduction

This section contains essential information to understand how SWD transport works, how it is organized, what bus commands are available, what is the Debug Access Port (DAP), Debug Port (DP), Access Port (AP), and how all this works. Information contained in this section are based on copyrighted by ARM Corporation [42] *Arm Debug Interface version 5* manual [44] and more specific to our example device *ARM Cortex-M1 Reference Manual* [45]. For detailed information refer to the *ARM Info Center* [46].

1.11.1.2 Signalling

SW-DP (Serial Wire Debug Port) operates with a synchronous serial interface. This uses a single bidirectional data signal SWDIOTMS and a clock signal SWCLK (and the GROUND signal ofcourse). Each sequence of operations on the wire consists of two or three phases:

- Packet request – The external host debugger issues a request to the debug port. The debug port is the target of the request.
- Acknowledge response – The target sends an acknowledge response to the host.
- Data transfer phase – This phase is only present when either:
  - Data Read or Data Write request is followed by a valid (OK) – acknowledge response.
  - ORUNDETECT flag is set to 1 in the CTRL/STAT Register (if CTRL/STAT=1 then data transfer is required on all responses).

The data transfer is one of:

- target to host, following a read request (RDATA)
- host to target, following a write request (WDATA).

The SW-DP uses a serial wire for both host and target sourced signals. The host emulator drives the protocol timing. Only the host emulator generates packet headers. Both the target and host are capable of driving the bus HIGH and LOW, or tristating it. Time required to switch from transmission into receiving mode is called Turnaround time.

The SW-DP clock, SWCLKTCK, can be asynchronous to the device/system CLK. SWCLKTCK can be stopped when the debug port is idle, but host must continue to clock the interface for a number of cycles after the data phase of any data transfer – this ensures that the transfer can be clocked through the SW-DP. 100k high–pullup resistor is recommended at target on SWTMSIO line, therefore this line should be driven high before entering low power mode.

1.11.1.3 Interface Reset and Synchronization

Interface reset or resynchronization occurs at 50 clocks with data line set high and then IDCODE read ended with OK response. Device will signal request for reset by not driving the data line at response stage, after two bad data sequences in a row target locks out and requests reset sequence described before. Additionally host should give target some time for command processing to return a payload, host can request IDCODE read and when it fails it should reset Target.

Note: After reset and jtag–to–swd sequence SWDIOTMS *line must be driven low with at least one clock pulse on* SWCLK *line, otherwise target will not respond to any request!* JTAG-TO-SWD *sequence presented by ARM is incomplete and will not work without* IDLE *cycle. It is good to implement* IDLE *command that consists of* SWDIOTMS *line set low with 8 clock pulses on* SWCLK *line (kind of request with* 0x00 *payload) that will allow* SW-DP *to successfully complete commands execution when appended at the end of long queue or reset sequences.*

1.11.1.4 SWD Packet Construction

- Start – single start bit, with value 1.
- APnDP – single bit, indicating whether the DP or the AP Access Register is to be accessed. This bit is 0 for a DPACC access, or 1 for an APACC access.
- RnW – single bit, indicating whether the access is a read or a write. This bit is 0 for an write access, or 1 for a read access.
- A[2:3] – two bits, giving the A[3:2] address field for the DP or AP Register Address (shifted out LSB first):
  - APACC access, the register being addressed depends on the A[3:2] value and the value held in the SELECT register.
  - DPACC access, the A[3:2] value determines the address of the register in the SW-DP register map.
- Parity – single parity bit for the preceding packet.
- Stop – single stop bit. In the synchronous SWD protocol this is always 0.
- Park – single bit. The host must drive the line high before tristating the line. The target reads this bit as 1.
- TRN (Turnaround) – this is a period when the line is not driven and the state of the line is Undefined. The length of the turnaround period is controlled by the TURNROUND field in the Wire Control Register. The default setting is a turnaround period of one clock cycle. By default turnaround time is one cycle.
- ACK – 3-bit target-to-host response. Transmitted LSB first on the wire.

- `WDATA[0:31]` – 32 bits of write data, from host to target.
- `RDATA[0:31]` - 32 bits of read data, from target to host.

In the SWD protocol, a simple parity check is applied to all packet request and data transfer phases. Parity bit appears on the wire immediately after the `A[2:3]` bits (`ACK[0:2]` bits are never included in the parity calculation). Even parity is used:

- Packet requests – parity check is made over the APnDP, RnW and A[2:3] bits (when the number of bits set to 1 is odd then the parity bit is set to 1, when the number of bits set to 1 is even then the parity bit is set to 0).
- Data transfers (`WDATA` and `RDATA`) - parity check is made over the 32 data bits, `WDATA[0:31]` or `RDATA[0:31]`. If, of these 32 bits (if the number of bits set to 1 is odd then the parity bit is set to 1, if the number of bits set to 1 is even then the parity bit is set to 0).

1.11.1.5 Successful Write Operation

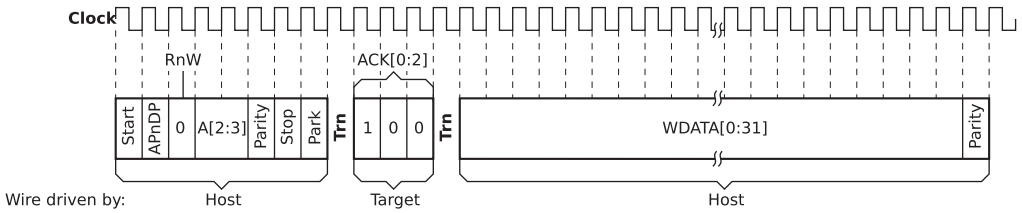


Figure 1.7: Successful write operation [46].

A successful write operation consists of three phases:

- 8-bit write packet request, from the host to the target
- 3-bit OK acknowledge response, from the target to the host
- 33-bit data write phase, from the host to the target

By default, there are single–cycle turnaround periods between each of these phases. The `OK` response only indicates that the debug port is ready to accept the write data. The debug port writes this data after the write phase has completed. The response to the debug port write itself is given on the next operation. There is no turnaround phase after the data phase. The host is driving the line and can start the next operation immediately. SW-DP can buffer writes to the `APBUS`.

The SW-DP implements a write buffer that enables it to accept write operations even when other transactions are still outstanding. The debug port issues an OK response to a write request if it can accept the write into its write buffer. This means that an OK response to a write request, other than a write to the `DP ABORT` Register, indicates only

that the write has been accepted by the debug port. It does not indicate that all previous transactions have completed.

If a write is accepted into the write buffer but later abandoned, the WDATAERR flag is set in the CTRL/STAT Register, see Control/Status Register (CTRL/STAT). A buffered write is abandoned if:

- A sticky flag is set by a previous transaction.
- A debug port read of the IDCODE or CTRL/STAT Register is made. Because the debug port is not permitted to stall reads of these registers, it must:
  - perform the IDCODE or CTRL/STAT Register access immediately
  - discard any buffered writes, because otherwise they would be performed out-of-order.
- A debug port write of the ABORT Register is made. This is because the debug port cannot stall an ABORT Register access.

This means that if you make a series of access port write transactions, it might not be possible to determine which transaction failed from examining the ACK responses. However, it might be possible to use other enquiries to find which write failed. For example, if you are using the auto-address increment (AddrInc) feature of a Memory Access Port (AHB-AP), then you can read the Transfer Address Register to find which was the final successful write transaction. See AHB-AP Transfer Address Register, TAR, 0x04 and AHB-AP register summary for more information.

The write buffer must be emptied before the following operations can be performed:

- any access port read operation
- any debug port operation other than a read of the IDCODE or CTRL/STAT Register, or a write of the ABORT Register.

Attempting these operations causes WAIT responses from the debug port until the write buffer is empty. If you have to perform a SW-DP read of the IDCODE or CTRL/STAT Register, or a SW-DP write to the ABORT Register immediately after a sequence of access port writes, you must first perform an access that the SW-DP is able to stall. In this way you can check that the write buffer is cleared before performing the SW-DP register access. If this is not done, WDATAERR might be set and the buffered writes lost.

1.11.1.6 Successful Read Operation

A successful read operation consists of three phases:

- 8-bit read packet request, from the host to the target
- 3-bit OK acknowledge response, from the target to the host

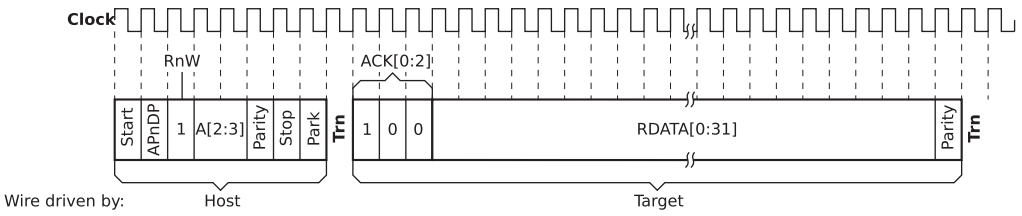


Figure 1.8: Successful read operation [46].

- 33-bit data read phase, where data is transferred from the target to the host.

By default, there are single–cycle turnaround periods between the first and second of these phases and after the third phase. However, there is no turnaround period between the second and third phases.

The `SW-DP CTRL/STAT` register includes a `READOK` flag, bit `[6]`. This register is described in Control/Status Register, `CTRL/STAT`. The `READOK` flag is updated on every access port read access and on every `RDBUFF` read request. When the `SW-DP` initiates the access port access it clears the `READOK` flag to `0` and, when the `SW-DP` target gives an `OK` response to the read request, it sets the `READOK` flag to `1`. This means that if a host receives a corrupted `ACK` response to an access port or `RDBUFF` read request it can check whether the read actually completed correctly. The host can read the `DP CTRL/STAT` Register to find the value of the `READOK` flag:

- If the flag is set to `1` then the read was performed correctly. The host can use a `RESEND` request to obtain the read result, see `Read Resend Register`, `RESEND` (`SW-DP` only).
- If the flag is set to `0` then the read was not successful. The host must retry the original access port or `RDBUFF` read request.

Read accesses to the access port are posted. This means that the result of the access is returned on the next transfer. If the next access you have to make is not another access port read then you must insert a read of the `DP RDBUFF` Register to obtain the posted result. When you must make a series of access port reads, you only have to insert one read of the `RDBUFF` Register:

- On the first access port read access, the read data returned is Undefined. You must discard this result.
- If you immediately make another access port read access this returns the result of the previous access port read.
- You can repeat this for any number of access port reads.
- Issuing the last access port read packet request returns the last-but-one access port read result.
- You must then read the `DP RDBUFF` Register to obtain the last access port read result.

1.11.1.7 WAIT response to Read or Write operation request

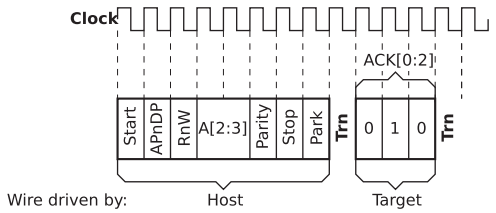


Figure 1.9: WAIT response to Read or Write operation request [46].

A WAIT response to a read or write packet request consists of two phases:

- 8-bit read or write packet request, from the host to the target
- 3-bit WAIT acknowledge response, from the target to the host.

By default, there are single–cycle turnaround periods between these two phases and after the second phase. If Overrun Detection is enabled then a data phase is required on a WAIT response. Writing to the ABORT register after receiving a WAIT response enables the debugger to access other parts of the debug system.

A WAIT response is issued by the SW-DP if it is not able to immediately process the request from the debugger. However, a WAIT response must not be issued to the following requests. SW-DP must always be able to process these three requests immediately: IDCODE, CTRL/STAT, ABORT. With any request other than those listed, the SW-DP issues a WAIT response, with no data phase, if it cannot process the request. This happens if a previous access port or debug port access is outstanding, or if the new request is an access port read request and the result of the previous AP read is not yet available.

Normally, when a debugger receives a WAIT response it retries the same operation. This enables it to process data as quickly as possible. However, if several retries have been attempted, and time permitted for a slow interconnection and memory system to respond, if appropriate, the debugger might write to the ABORT register. This signals to the active access port that it must terminate the transfer that it is currently attempting. An access port implementation might be unable to terminate a transfer on its ASIC interface. However, on receiving an ABORT request the access port must free up the SWD interface.

1.11.1.8 FAULT response to Read or Write operation request

A FAULT response to a read or write packet request consists of two phases:

- 8–bit read or write packet request, from the host to the target.
- 3–bit FAULT acknowledge response, from the target to the host.

By default, there are single–cycle turnaround periods between these two phases and after the second phase. If Overrun Detection is enabled then a data phase is required on a FAULT response.

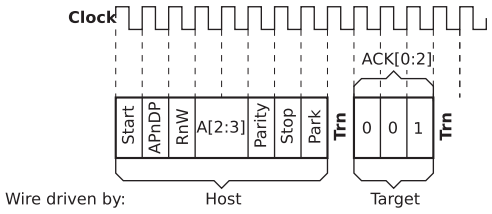


Figure 1.10: FAULT response to Read or Write operation request [46].

SW-DP does not issue a FAULT response to an access to the IDCODE, CTRL/STAT or ABORT registers. For any other access, the SW-DP issues a FAULT response if any sticky flag is set in the CTRL/STAT Register. Use of the FAULT response enables the protocol to remain synchronized. A debugger might stream a block of data and then check the CTRL/STAT register at the end of the block. The sticky error flags are cleared by writing bits in the ABORT register.

1.11.1.9 Protocol Error Sequence

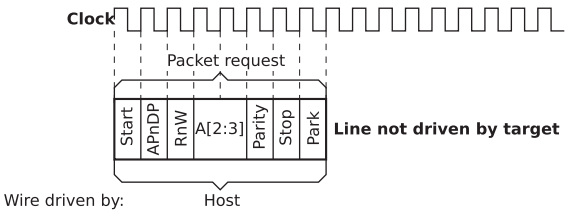


Figure 1.11: Protocol error sequence [46].

A protocol error occurs when a host issues a packet request but the target fails to return any acknowledge response. If the SW-DP detects a parity error in the packet request it does not reply to the request.

When the host receives no reply to its request, it must back off, in case the SW-DP has lost frame synchronization for some reason. After this, it can issue a new transfer request. In this situation it must read the IDCODE register – this is mandated by this specification because a successful read of the IDCODE register confirms that the target is operational. If there is no response at the second attempt, the debugger must force a line reset to ensure frame synchronization and valid operation. This is necessary because the SW-DP is in a state where it only responds to a line reset. After the line reset the debugger must read the IDCODE register before it attempts any other operations.

If the transfer that resulted in the original protocol error response was a write, you can assume that no write occurred. If the original transfer was a read, it is possible that the read was issued to an access port. Although this is unlikely, you must consider this possibility because reads are pipelined and the debug port might implement a write buffer.

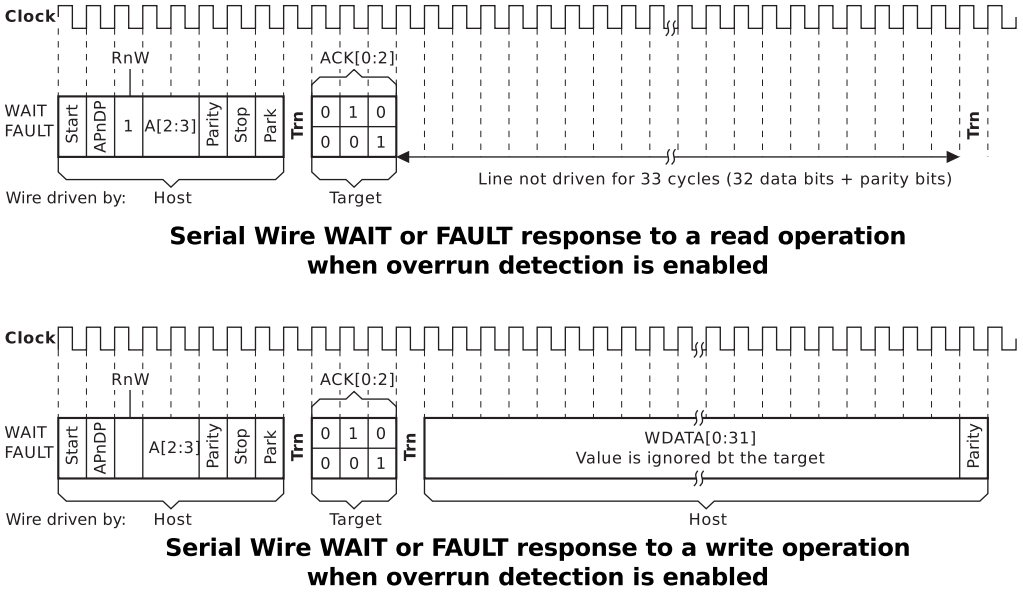


Figure 1.12: Protocol error sequence [46] when Sticky Overrun Detection is enabled.

1.11.1.10 SW-DP Registers

`SW-DP` registers:

- address `b00`:
  - R, `APnDP=b0`: Identification Code Register, `IDCODE`
  - W, `APnDP=b0`: Abort Register, `ABORT`
- address `b01`:
  - `CTRLSEL=b0`: R/W: Control/Status Register, `CTRL/STAT`
  - `CTRLSEL=b1`: R/W: Wire Control Register, `WCR` (`SW-DP` only)
- address b10:
  - R: Read Resend Register, `RESEND` (`SW-DP` only)
  - W: AP Select Register, `SELECT`
- address `b11`:
  - R: Read Buffer, `RDBUFF`

Note: There is a bug in ARM documentation stating that `IDCODE` *and* `ABORT` *register should be accessed with* `APnDP=1`*, but this points to the Access Port (*`AP`*) registers, not Debug Port (*`DP`*) registers, while these registers clearly belongs to the* `DP`*.*

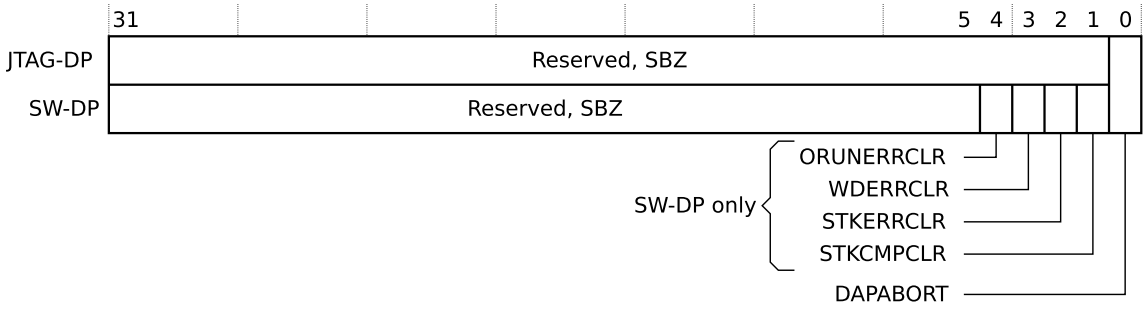


Figure 1.13: `ABORT` register map [46].

1.11.1.11 ABORT Register

The Abort Register is always present on all debug port implementations. Its main purpose is to force a `DAP` abort. On a `SW-DP`, it is also used to clear error and sticky flag conditions. A write–only register. Always accessible and returns an `OK` response if a valid transaction is received. Abort Register accesses always complete on the first attempt.

Bit description:

- `[31:5]` – Reserved, SBZ.
- `[4] ORUNERRCLR` – Write b1 to this bit to clear the `STICKYORUN` overrun error flagb (SW-DP only).
- `[3] WDERRCLR` – Write b1 to this bit to clear the `WDATAERR` write data error flagb (SW-DP only).
- `[2] STKERRCLR` – Write b1 to this bit to clear the `STICKYERR` sticky error flagb (SW-DP only).
- `[1] STKCMPCLR` – Write b1 to this bit to clear the `STICKYCMP` sticky compare flagb (SW-DP only).
- `[0] DAPABORT` – Write b1 to this bit to generate a DAP abort. This aborts the current access port transaction. This must only be done if the debugger has received `WAIT` responses over an extended period.

DP Aborts – Writing `b1` to bit `[0]` of the `Abort Register` generates a debug port abort, causing the current `AP` transaction to abort. This also terminates the `Transaction Counter`, if it was active. From a software perspective, this is a fatal operation. It discards any outstanding and pending transactions and leaves the access port in an unknown state. However, on a `SW-DP`, the sticky error bits are not cleared. You use this function only in extreme cases, where debug host software has observed stalled target hardware for an extended period. Stalled target hardware is indicated by `WAIT` responses.

After a debug port abort is requested, new transactions can be accepted by the debug port. However, an access port access to the access port that was aborted can result in

more `WAIT` responses. Other access ports can be accessed, however, the state of the system might make it impossible to continue with debug.

Clearing error and sticky compare flags – When a debugger, connected to a `SW-DP`, checks the `Control/Status Register` and finds that an error flag is set, or that the sticky compare flag is set, it must write to the Abort Register to clear the error or sticky compare flag. You can use a single write of the Abort Register to clear multiple flags, if this is necessary. After clearing the flag, you might have to access the debug port and access port registers to find what caused the flag to be set. Typically:

- For the `STICKYCMP` or `STICKYERR` flag, you must find which location was accessed to cause the flag to be set.
- For the `WDATAERR` flag, after clearing the flag you must resend the data that was corrupted.
- For the `STICKYORUN` flag, you must find which debug port or access port transaction caused the overflow. You then have to repeat your transactions from that point.

1.11.1.12 IDCODE, Identification Code Register

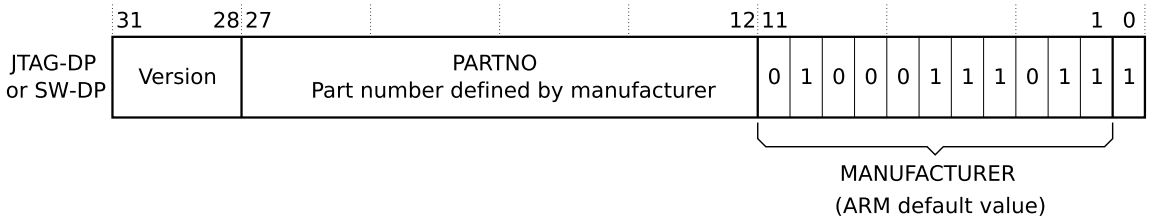


Figure 1.14: `IDCODE` register map [46].

The Identification Code Register is always present on all debug port implementations. It provides identification information about the ARM Debug Interface. It is at address `0b00` on read operations when the `APnDP bit=1`. It is a read-only register and always accessible.

Bits description:

- `[31:28]` Version code: `JTAG-DP=0x3`, `SW-DP=0x2`
- `[27:12]` `PARTNO` – Part Number for the debug port. Current ARM-designed debug ports have the following `PARTNO` values: `JTAG-DP=0xBA00, SW-DP=0xBA10`
- `[11:1]` `MANUFACTURER` – JEDEC Manufacturer ID, an 11-bit JEDEC code that identifies the manufacturer of the device. The ARM default value for this field is `0x23B`.
- `[0]` – Always `0b1`.

`JEDEC Manufacturer ID` codes are assigned by the JEDEC Solid State Technology Association, see *JEP106M, Standard Manufacture's Identification Code.* This code is also described as the *JEP-106 manufacturer identification code* and can be subdivided into two fields:

- Continuation code, 4 bits, `[11:8]: b0100, 0x4`
- Identity code, 7 bits, `[7:1]: b0111011, 0x3B`

1.11.1.13 CTRL/STAT, Control/Status Register

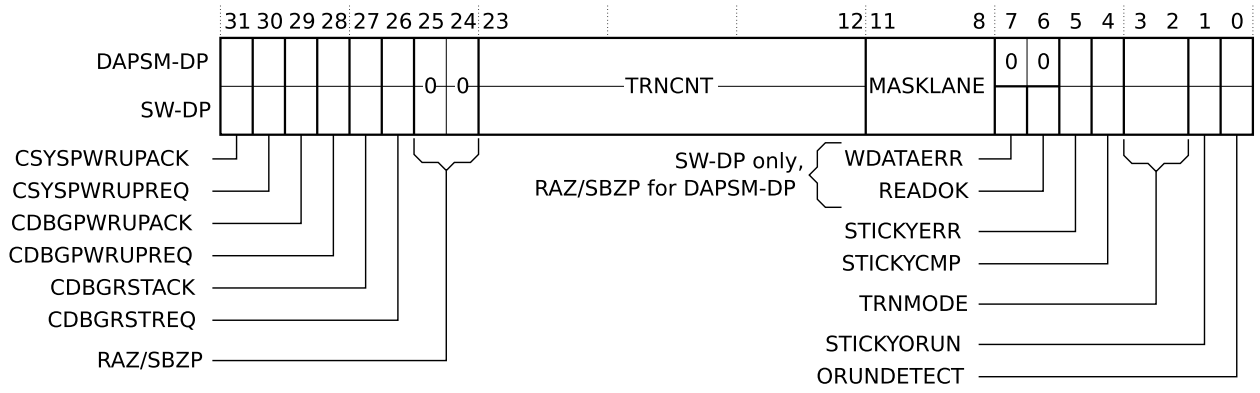


Figure 1.15: `CTRL/STAT` register map [46].

The `Control/Status Register` is always present on all debug port implementations. It provides control of the debug port and status information about the debug port. It is located at address `0b01` on read and write operations when the `APnDP=b1` and the `CTRLSEL=b0` in the `Select Register`. It is a read–write register, in which some bits have different access rights. It is implementation–defined whether some fields in the register are supported – below is an obligatory bit list and description.

Bits description:

- `[31]`, RO: `CSYSPWRUPACK` – System power–up acknowledge.
- `[30]`, R/W: `CSYSPWRUPREQ` – System power–up request. After a reset this bit is `LOW` (`0`).
- `[29]`, RO: `CDBGPWRUPACK` – Debug power–up acknowledge.
- `[28]`, R/W: `CDBGPWRUPREQ` – Debug power–up request. After a reset this bit is `LOW` (`0`).
- `[27]`, RO: `CDBGRSTACK` – Debug reset acknowledge.
- `[26]`, R/W: `CDBGRSTREQ` – Debug reset request. After a reset this bit is `LOW` (`0`).

- [25:24] – Reserved, RAZ/SBZP.
- [21:12], R/W: TRNCNT – Transaction counter. After a reset the value of this field is Unpredictable.
- [11:8], R/W: MASKLANE – Indicates the bytes to be masked in pushed compare and pushed verify operations. After a reset the value of this field is Unpredictable. The MASKLANE field, bits [11:8] of the CTRL/STAT Register, is only relevant if the Transfer Mode is set to pushed verify or pushed compare operation. In the pushed operations, the word supplied in an access port write transaction is compared with the current value of the target access port address. The MASKLANE field lets you specify that the comparison is made using only certain bytes of the values. Each bit of the MASKLANE field corresponds to one byte of the access port values. Therefore, each bit is said to control one byte lane of the compare operation:
  - b1XXX: Include byte lane 3 in comparisons (0xFF---)
  - bX1XX: Include byte lane 2 in comparisons (0x-FF--)
  - bXX1X: Include byte lane 1 in comparisons (0x--FF-)
  - bXXX1: Include byte lane 0 in comparisons (0x---FF)
- [7], RO[1]: WDATAERR[1] – This bit is set to 1 if a Write Data Error occurs. This bit can only be cleared by writing b1 to the WDERRCLR field of the Abort Register. After a power–on reset this bit is LOW (0). It is set if:
  - there is a parity or framing error on the data phase of a write
  - a write that has been accepted by the debug port is then discarded without being submitted to the access port.
- [6], RO[1]: READOK[1] – This bit is set to 1 if the response to a previous access port or RDBUFF was OK. It is cleared to 0 if the response was not OK. This flag always indicates the response to the last access port read access. After a power–on reset this bit is LOW (0).
- [5], RO[2]: STICKYERR – This bit is set to 1 when the processor receives a bus error on the system AHB-Lite bus. When STICKYERR is set, no transaction is passed from the JTAG or SW interfaces to the debug AHB system bus. Any read that is performed when STICKYERR is set results in data that is Unpredictable. To clear this bit write b1 to the STKERRCLR field of the Abort Register. After a power–on reset this bit is LOW (0).
- [4], RO[2]: STICKYCMP - This bit is set to 1 when a match occurs on a pushed compare or a pushed verify operation. To clear this bit write b1 to the STKCMPCLR field of the Abort Register, see Abort Register, ABORT. After a power-on reset this bit is LOW (0).

3:2 , R/W: TRNMODE - This field sets the transfer mode for access port operations. After a power-on reset the value of this field is Unpredictable. In normal operation, access port transactions are passed to the access port for processing. In pushed verify and pushed compare operations, the debug port compares the value supplied in the access port transaction with the value held in the target access port address. Below is a list of the permitted values of this field and their meaning:

  - `b00`: Normal operation
  - `b01`: Pushed verify operation
  - `b10`: Pushed compare operation
  - `b11`: Reserved

- `[1]`, RO`[2]`: `STICKYORUN` – If overrun detection is enabled (see bit `[0]` of this register), this bit is set to `1` when an overrun occurs. To clear this bit write `b1` to the `ORUNERRCLR` field of the Abort Register, `ABORT`. After a power–on reset this bit is `LOW` (`0`).

- `[0]`, R/W: `ORUNDETECT` – This bit is set to `b1` to enable overrun detection. After a reset this bit is `Low` (`0`).

1.11.1.14 SELECT, AP Select Register

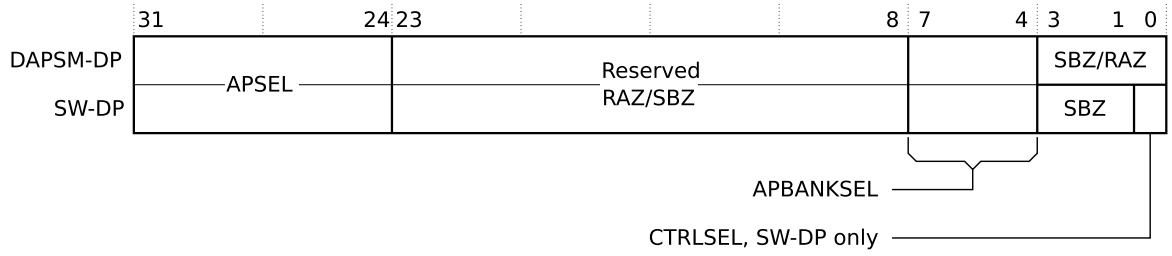


Figure 1.16: `SELECT` register map [46].

The `AP Select Register` is always present on all debug port implementations. Its main purpose is to select the current Access Port (`AP`) and the active four–word register window in that access port. On a `SW-DP`, it also selects the `Debug Port` address bank. It is at address `0b10` on write operations when the `APnDP=b1` and is a write–only register. Access to the AP Select Register is not affected by the value of the `CTRLSEL` bit.

Bits description:

- `[31:24]`, `APSEL`: Selects current access port. Note: Because the processor has only one access port, `APSEL` must be `8'b00000000`. The reset value of this field is Unpredictable.

- `[23:8]`, – Reserved. SBZ/RAZ[1].

- `[7:4]`, `APBANKSEL` – Selects the active 4-word register window on the current access port. The reset value of this field is Unpredictable.
- `[3:1]`, – Reserved. SBZ/RAZ[1].
- `[0]`, `CTRLSEL`: `SW-DP` Debug Port address bank select, `SW-DP` only. After a reset this field is `b0`. However the register is WO so you cannot read this value. The `CTRLSEL` field, bit `[0]`, controls which debug port register is selected at address `b01` on a `SW-DP`. Meaning of the different values of `CTRLSEL` is as follows:
  - `0`: `CTRL/STAT`, see Control/Status Register
  - `1`: `WCR`, see Wire Control Register (`SW-DP` only)

1.11.1.15 RDBUFF, Read Buffer

The `32-bit Read Buffer` is always present on all debug port implementations. However, there are significant differences in its implementation on `JTAG` and `SW Debug Ports`. On `SW-DP` it is at address `0xC` on read operations when the `APnDP=b1` and is a read–only register. Access to the Read Buffer is not affected by the value of the `CTRLSEL` bit in the `SELECT` Register.

On a `SW-DP`, performing a read of the Read Buffer captures data from the access port, presented as the result of a previous read, without initiating a new access port transaction. This means that reading the Read Buffer returns the result of the last access port read access, without generating a new AP access. After you have read the Read Buffer, its contents are no longer valid. The result of a second read of the Read Buffer is Unpredictable.

If you require the value from an access port register read, that read must be followed by one of:

- A second access port register read. You can read the Control/Status Register (`CSW`) if you want to ensure that this second read has no side effects.
- A read of the DP Read Buffer. This access, to the access port or the debug port depending on which option you used, stalls until the result of the original access port read is available.

1.11.1.16 WCR, Wire Control Register

The Wire Control Register is always present on any `SW-DP` implementation. Its purpose is to select the operating mode of the physical serial port connection to the `SW-DP`. It is a read/write register at address `0b01` on read and write operations when the `CTRLSEL=b1` in the Select Register. For information about the `CTRLSEL` bit see AP Select Register `SELECT`. Note: When the `CTRLSEL=b1`, to enable access to the `WCR`, the DP Control/Status Register is not accessible. Many features of the Wire Control Register are implementation–defined!

Bits description:

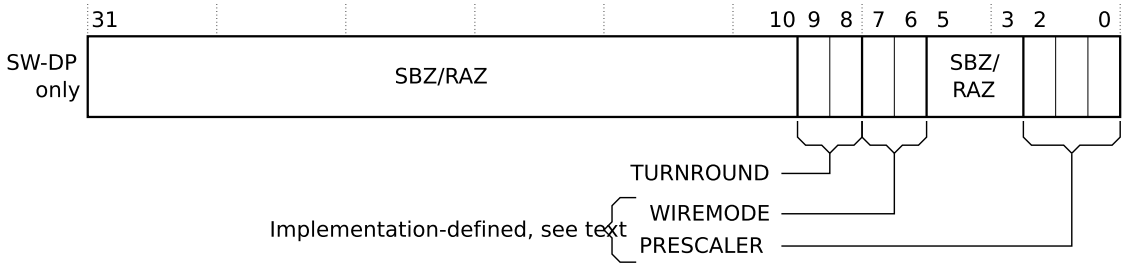


Figure 1.17: `WCR` register map [46].

- `[31:10]` – Reserved. SBZ/RAZ.
- `[9:8]`, `TURNROUND`: Turnaround tristate period. After a reset this field is `b00`. This field defines the turnaround tristate period. This turnaround period allows for pad delays when using a high sample clock frequency. The possible values of this field and their meanings is presented below:
  - `b00`: 1 sample period
  - `b01`: 2 sample periods
  - `b10`: 3 sample periods
  - `b11`: 4 sample periods
- `[7:6]`, `WIREMODE`: Identifies the operating mode for the wire connection to the debug port. After a reset this field is `b01`. This field identifies `SW-DP` as operating in Synchronous mode only. This field is required. The possible values of the field and their meanings is presented below:
  - `b00`: Reserved
  - `b01`: Synchronous (no oversampling)
  - `b1X`: Reserved
- `[5:3]` – Reserved. SBZ/RAZ.
- `[2:0]`, `PRESCALER`: Reserved. SBZ/RAZ.

1.11.1.17 RESEND, Read Resend Register

The Read Resend Register is always present on any `SW-DP` implementation. Its purpose is to enable the read data to be recovered from a corrupted debugger transfer, without repeating the original `AP` transfer. It is a `32-bit` read-only register at address `0b10` on read operations. Access to the Read Resend Register is not affected by the value of the `CTRLSEL` bit in the `SELECT` Register.

Performing a read to the `RESEND` register does not capture new data from the access port. It returns the value that was returned by the last `AP` read or `DP RDBUFF` read.

Reading the RESEND register enables the read data to be recovered from a corrupted transfer without having to re–issue the original read request or generate a new DAP or system level access. The RESEND register can be accessed multiple times. It always returns the same value until a new access is made to the DP RDBUFF register or to an access port register.

1.11.2 LibSWD – Serial Wire Debug Open Library

1.11.2.1 Introduction

To be able to work with SWD transport on my target ARM–Cortex device I had to create a library that would allow to operate on SWD bus and integrate easily into existing applications that did not mention SWD at design time. LibSWD [38] was designed to be a standalone transport framework with both high–level and low–level API to work with SWD devices and allow extending its capabilities in future. Library was written by Tomasz CEDRO from Orange Labs Warsaw (Poland) under 3–clause BSD license [35].

LibSWD is integrated with external software using „driver bridge" – a simple set of functions that drive physical signall of the physical interface, where interface itself can be driven with existing drivers of that applications, making LibSWD very robust and universal.

Basic integration with UrJTAG [37] and OpenOCD [36] was done – a free and Open–Source utilities previously used only for JTAG operations, just waiting to extend its existing potential and usability with new features, such as SWD transport. This however was very time consuming and difficult task because it required redesign of existing software architecture without destroying existing design to maintain backward–compatibility. Open–Source gives this opportunity. On the other hand applications created by dozens of developers without clear lidership not always represent the perfect coding style and project organization. There are still lots of things to be done in those software utilities, but the benefits and its potential are now clearly visible and proven.

1.11.2.2 LibSWD Reference

LibSWD has its own website at http://www.libswd.sf.net with remote code repository and download section. LibSWD is documented with Doxygen, an inline source code documentation system, providing always up–to–date documentation of the source code, even between releases.

LibSWD is an Open-Source framework to deal with with Serial Wire Debug Port in accordance to ADI (Arm Debug Interface, version 5.0 at the moment) specification. It is released under 3-clause BSD license.

1.11.2.3 SWD basics

Serial Wire Debug is an alternative to JTAG (IEEE1149.1) transport layer for accessing the Debug Access Port in ARM-Cortex based devices. LibSWD provides methods for

bitstream generation on the wire using simple but flexible API that can reuse capabilities of existing applications for easier integration. Every bus operation such as control, request, turnaround, acknowledge, data and parity packet is named a "command" represented by a `swd_cmd_t` data type that builds up the queue that later can be flushed into real hardware using standard set of (application–specific) driver functions. This way LibSWD is almost standalone and can be easily integrated into existing utilities for low–level access and only requires in return to define driver bridge that controls the physical interface interconnecting host and target. Drivers and other application–specific functions are `extern` and located in external file crafted for that application and its hardware. LibSWD is therefore best way to make your application SWD aware.

1.11.2.4 Context

The most important data type in LibSWD is `swd_ctx_t` structure type, a context that represents logical entity of the swd bus (transport layer between host and target) with all its parameters, configuration and command queue. Context is being created with `swd_init()` function that returns pointer to allocated virgin structure, and it can be destroyed with `swd_deinit()` function taking the pointer as argument. Context can be set only for one interface–target pair, but there might be many different contexts in use if necessary, so amount of devices in use is not limited.

1.11.2.5 Function Organization

All functions in general operates on pointer type and returns number of processed elements on success or negative value with `swd_error_code_t` on failure. Functions are grouped by functionality that is denoted by function name prefix (ie. `swd_bin*` are for binary operations, `swd_cmdq*` deals with command queue, `swd_cmd_enqueue*` deals with creating commands and attaching them to queue, `swd_bus*` performs operation on the swd transport system, `swd_drv*` are the interface drivers, etc).

Standard end–users are encouraged to only use high level functions (`swd_bus*, swd_dap*, swd_dp*`) to perform operations on the swd transport layer and the target's `DAP` (Debug Access Port) and its components such as `DP` (Debug Port) and the `AP` (Access Port). More advanced users however may use low level functions (`swd_cmd*, swd_cmdq*`) to group them into new high–level functions that automates some tasks (such as high–level functions does).

Functions of type `extern` are the ones to implement in external file by developers that want to incorporate LibSWD into their application. Context structure also has void pointer in the `swd_driver_t` structure that can hold address of the external driver structure to be passed into internal swd drivers (`extern swd_drv*` functions) that wouldn't be accessible otherwise.

1.11.2.6 Commands

Bus operations are split into *commands* represented by `swd_cmd_t` data type. They form a bidirectional command queue that is part of `swd_ctx_t` structure. Command type, and so its payload, can be one of: `control` (user defined 8–bit payload), `request` (according to the standard), `ack`, `data`, `parity` (data and parity are separate commands!), `trn`, `bitbang` and `idle` (equals to control with zero data). Command type is defined by `swd_cmdtype_t` and its code can be negative (for `MOSI` operations) or positive (for `MISO` operations) – this way bus direction can be easily calculated by multiplying two operation codes (when the result is negative bus will have to change direction), so the libswd "knows" when to put additional `TRN` command of proper type between enqueued commands.

Payload is stored within union type and its data can be accessed according to payload name, or simply with `data8` (`char`) and `data32` (`int`) fields. Payload for write (`MOSI`) operations is stored on command creation, but payload for read (`MISO`) operations becomes available only after command is executed by the interface driver.

There are 3 methods of accessing read data – flushing the queue into driver then reading queue directly, single stepping queue execution (flush one–by–one) then reading context log of last executed command results (there are separate fields of type `swd_transaction_t` in `swd_ctx_t`'s log structure for read and write operations), or providing a double pointer on command creation to have constant access to its data after execution.

After all commands are enqueued with `swd_cmd_enqueue*` function set, it is time to send them into physical device with `swd_cmdq_flush()` funtion. According to the `swd_operation_t` parameter commands can be flushed one–by–one, all of them, only to the selected command or only after selected command. For low level functions all of these options are available, but for high–level functions only two of them can be used – `SWD_OPERATION_ENQUEUE` (but not send to the driver) and `SWD_OPERATION_EXECUTE` (all unexecuted commands on the queue are executed by the driver sequentially) – that makes it possible to perform bus operations one after another having their result just at function return, or compose more advanced sequences leading to preferred result at execution time.

Because high–level functions provide simple and elegant manner to get the operation result, it is advised to use them instead dealing with low–level functions (implementing memory management, data allocation and queue operation) that exist only to make high–level functions possible.

1.11.2.7 Drivers

Calling the `swd_cmdq_flush()` function leads to execution of not yet executed commands from the queue (in a manner specified by the operation parameter) on the SWD bus (transport layer between interface and target, not the bus of the target itself) by `swd_drv_transmit()` function that use application specific `extern` functions defined in external file (ie. `libswd_drv_urjtag.c`) to operate on a real hardware using drivers from existing application.

LibSWD use only `swd_drv_{mosi,miso}_8,32` (separate for 8–bit char and 32–bit int data cast type) and `swd_drv_{mosi,miso}_trn` functions to interact with drivers, so it is

possible to easily reuse low–level and high–level devices for communications, as they have all information necessary to perform exact actions – number of bits, payload, command type, shift direction and bus direction. It is even possible to send raw bytes on the bus (control command) or bitbang the bus (bitbang command) if necessary.

`MOSI` (Master Output Slave Input) and `MISO` (Master Input Slave Output) was used to clearly distinguish transfer direction (from master–interface to target–slave), as opposed to ambiguous read/write statements, so after `swd_drv_mosi_trn()` master should have its buffers set to output and target inputs active. Drivers, as most of the LibSWD functions, works on data pointers instead data copy and returns number of elements processed (bits in this case) or negative error code on failure.

1.11.2.8 Example program

Below is the simplest possible example program source code to show how to initialize libswd, read the `IDCODE` register out of the target device, then deinitialize and quit:

```
1  #include <libswd.h>
2    int main(){
3     swd_ctx_t *swdctx;
4     int res, *idcode;
5     swdctx=swd_init();
6     if (swdctx==NULL) return -1;
7     //we might need to pass external driver structure to swd_drv* functions
8     //swdctx->driver->device=...
9     res=swd_dap_detect(swdctx, SWD_OPERATION_EXECUTE, &idcode);
10    if (res<0){
11     printf("ERROR: %s\n", swd_error_string(res));
12     return res;
13    } else printf("IDCODE: 0x%X (%s)\n", *idcode, swd_bin32_string(idcode));
14    swd_deinit(swdctx);
15    return 0;
16   }
```

1.11.3 LibSWD in practice

This section documents practical steps already done with LibSWD, real–life hardware and software. LibSWD is still under heavy development, but soon it will become most popular utility in the embedded development world of ARM Cortex devices.

All work regarding SWD implementation is described in details at `http://stm32primer2swd.sf.net`. This is first in the world universal and open SWD implementation!

1.11.4 LibSWD integration with UrJTAG

This section documents integration of LibSWD with UrJTAG application [37] for low level accessing digital systems equipped with IEEE1149.1 JTAG interface. At the time when project was started SWD was not supported by UrJTAG and there was no library nor idea on how to support that bus, so it had to be invented from scratch, just as libswd itself.

UrJTAG was the first Open–Source program for accessing JTAG system and boundary scan. It had also simple driver for flash access and *SVF* file format player (it allows

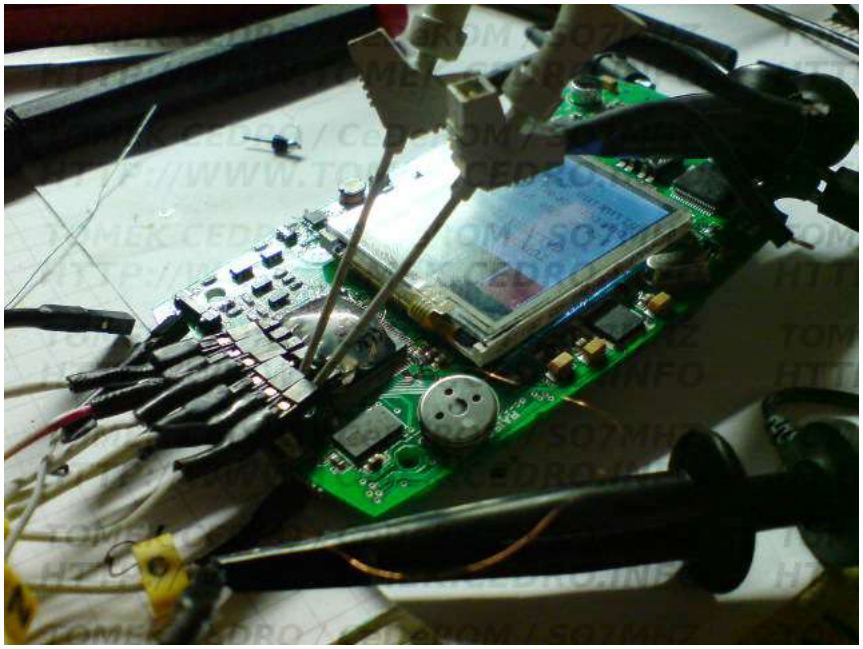


Figure 1.18: Tapping jtag/swd interface into physical signals.

replaying JTAG bus operations from file), that made is perfect tool for board testing, uploading/downloading firmware and acting as bridge between target microelectronic device and the computer software working on its registers. UrJTAG however does not have any built–in logic for debugging, flash memory support on many different targets, etc. It was simply meant to work on target JTAG registers. This is both weakness of the application because itself it cannot do anything useful, and the strong part as it has no predefined target device, any device can be virtually created and supported based on *BSDL* file (describes internal register map of a target device) provided by manufacturer.

The first question asked here should be „why to make JTAG application talk SWD" anyway? The answer is simple – because it has already developed into mature, well tested application supporting multiple features and interfaces, so we don't need to create everything from scratch. Extending functionality of Open–Source application is also faster and cheaper. Interfaces are different for JTAG and SWD, but the driver infrastructure could be reused and the code organization turned out to be good enough to make it happen, everything else had to be invented and implemented with code. It was also fun to try.

Note that all code change did not destroy or impact existing program behavior in any negative way. Everything is perfectly backward–compatible with new features available. This is very important and big achievement with such great functionality change. To make it happen additional abstraction layer has been created named `transport`, being something between `cable` and the `target` (usually it was JTAG).

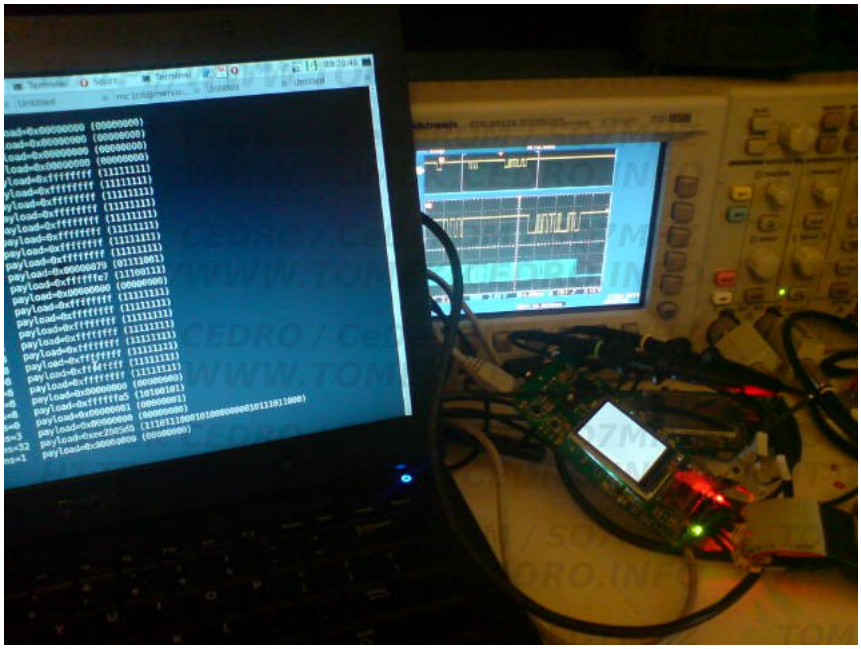


Figure 1.19: LibSWD communicating with Stm32Primer2 using UrJTAG drivers.

1.11.4.1 Creating KT-LINK driver

KT–LINK was the first cheap interface on the market, supporting both JTAG and SWD, designed in polish company Kristech [47] based on new FT2232H (`H` suffix stands for 480MBit USB Hi–Speed) chip from FTDI Chip Ltd. [53]. Its total cost of 50EUR and well known/supported chip made it perfect candidate for prototyping of SWD interface. It was also the only interface at time to work with, but it was so new that it had no driver, so we had to write it!

Luckily the UrJTAG application had the backend framework to work with the bitstream on the interface, so the driver implementation was in fact limited to hardware initialization and setup/cleanup routines. It was much harder with OpenOCD application where there was no generic functionality to work with the interface hardware except JTAG, also the level of internals complexity and bad code practices made this task harder than planned – it took few months to create from scratch generic bistream/bitbang framework and integrate it with existing solutions, also to make new transport work it was necessary to create transport framework...

1.11.4.2 Functionality verification

The successful integration ended with reading the `IDCODE` register out of the target CPU using SWD bus. Still there is a need to create user friendly commandline interface to work on bus by hand. Note that logging level change and proper error handling is also implemented. Please see the examples below as shown on using the UrJTAG application (the OpenOCD driver is now functional as well):

- Default JTAG driver invocation:

```
UrJTAG 0.10 #1864
Copyright (C) 2002, 2003 ETC s.r.o.
Copyright (C) 2007, 2008, 2009 Kolja Waschk and the respective authors
UrJTAG is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
There is absolutely no warranty for UrJTAG.
jtag.c:518 main() Warning: UrJTAG may damage your hardware!
Type "quit" to exit, "help" for help.
jtag> cable kt-link
Transport not selected or unsupported. Default transport is: JTAG
Connected to libftdi driver.
nSRST pin state is high...
KT-LINK JTAG Mode Initialization OK!
jtag> pod reset=0
jtag> frequency 10000
Setting TCK frequency to 10000 Hz
jtag> detect
IR length: 9
Chain length: 2
Device Id: 00111011101000000000010001110111 (0x3BA00477)
 Unknown manufacturer! (01000111011) (/mnt/stuff/tmp/swd/target/share/urjtag/MANUFACTURERS)
Device Id: 00000110010000010100000001000001 (0x06414041)
 Unknown manufacturer! (00000100000) (/mnt/stuff/tmp/swd/target/share/urjtag/MANUFACTURERS)
```

- Transport selection is what tells the program whether JTAG or SWD will be used for the session. The old driver selection was made by calling `cable` and the interface/driver name. Interface selection is the first command executed after program starts. Additional parameter to `cable` is the `transport` that can be one of `jtag` or `swd` (default is `jtag` to maintain backward compatibility when the parameter was not required).

```
jtag> cable kt-link help
Usage: cable KT-LINK [vid=VID] [pid=PID] [desc=DESC] [TRANSPORT]
VID        USB Device Vendor ID (hex, e.g. 0abc)
PID        USB Device Product ID (hex, e.g. 0abc)
DESC       Some string to match in description or serial no.
TRANSPORT  Setup cable transport mode (jtag, swd, ...).
Default:   vid=403 pid=bbe2 driver=ftdi-mpsse
jtag> cable kt-link swd
Connected to libftdi driver.
nSRST pin state is high...
KT-LINK SWD Mode Initialization OK!
jtag> cable kt-link jtag
Connected to libftdi driver.
nSRST pin state is high...
KT-LINK JTAG Mode Initialization OK!
jtag> cable kt-link
Transport not selected or unsupported. Default transport is: JTAG
Connected to libftdi driver.
nSRST pin state is high...
KT-LINK JTAG Mode Initialization OK!
```

- Detecting target device with `detect` command is the second call just after `cable` selection. On success we should have the `IDCODE` printed in hex and binary:

```
jtag> cable kt-link swd
Connected to libftdi driver.
nSRST pin state is high...
KT-LINK SWD Mode Initialization OK!
```

```
jtag> detect
IDCODE=0x0EE2805D8 (11101110001010000000010111011000)
```

- It is possible that detect will fail, so we should get the error message. As we can see the error reporting is very informative and helpful in finding problem location:

```
jtag> detect
Error: detect.c:566 urj_tap_detect_swd() Cable<->DAP transport error: swd_dap_detect() failed ([SWD_ERROR_ACK] acknowled
```

- In case of application troubles we can see the function call order by increasing the verbosity level to `detail`:

```
jtag> debug detail
jtag> detect
Detecting SWD devices...
SWD_I: Executing swd_dap_activate(0x2833ccc0, SWD_OPERATION_EXECUTE)
SWD_I: Executing swd_dap_reset(0x2833ccc0, SWD_OPERATION_EXECUTE)
SWD_I: Executing swd_dp_read_idcode(0x2833ccc0, SWD_OPERATION_EXECUTE)
SWD_I: swd_dp_read_idcode() succeeds, IDCODE=EE2805D8 (11101110001010000000010111011000)
IDCODE=0x0EE2805D8 (11101110001010000000010111011000)
```

- It is even possible to track the bitstream on the bus having additional insight into function call, data locations for debugging and to reduce additional equipment necessary such as digital oscilloscope with memory:

```
jtag> debug debug
swd_log_level_inherit(): SWD Context not (yet) initialized...
Return in urj_parse_line r=0
jtag> detect
Detecting SWD devices...
swd_init() OK
SWD_I: Executing swd_dap_activate(0x2833cb40, SWD_OPERATION_EXECUTE)
SWD_D: swd_drv_transmit(0x2833cb40, 0x2832b920) bits=0  cmdtype=UNDEFINED    returns=0   payload=0x00000000 (00000000)
SWD_D: swd_drv_transmit(0x2833cb40, 0x2832b960) bits=1  cmdtype=MOSI_TRN     returns=1   payload=0x00000000 (00000000)
SWD_D: swd_drv_transmit(0x2833cb40, 0x2832bc00) bits=8  cmdtype=MOSI_CONTROL returns=8   payload=0xffffffff (11111111)
SWD_D: swd_drv_transmit(0x2833cb40, 0x2832bc20) bits=8  cmdtype=MOSI_CONTROL returns=8   payload=0xffffffff (11111111)
SWD_D: swd_drv_transmit(0x2833cb40, 0x2832bc40) bits=8  cmdtype=MOSI_CONTROL returns=8   payload=0xffffffff (11111111)
SWD_D: swd_drv_transmit(0x2833cb40, 0x2832bc60) bits=8  cmdtype=MOSI_CONTROL returns=8   payload=0xffffffff (11111111)
SWD_D: swd_drv_transmit(0x2833cb40, 0x2832bc80) bits=8  cmdtype=MOSI_CONTROL returns=8   payload=0xffffffff (11111111)
SWD_D: swd_drv_transmit(0x2833cb40, 0x2832bca0) bits=8  cmdtype=MOSI_CONTROL returns=8   payload=0xffffffff (11111111)
SWD_D: swd_drv_transmit(0x2833cb40, 0x2832bcc0) bits=8  cmdtype=MOSI_CONTROL returns=8   payload=0xffffffff (11111111)
SWD_D: swd_drv_transmit(0x2833cb40, 0x2832bce0) bits=8  cmdtype=MOSI_CONTROL returns=8   payload=0xffffffff (11111111)
SWD_D: swd_drv_transmit(0x2833cb40, 0x2832bd00) bits=8  cmdtype=MOSI_CONTROL returns=8   payload=0x00000079 (01111001)
SWD_D: swd_drv_transmit(0x2833cb40, 0x2832bd20) bits=8  cmdtype=MOSI_CONTROL returns=8   payload=0xffffffe7 (11100111)
SWD_I: Executing swd_dap_reset(0x2833cb40, SWD_OPERATION_EXECUTE)
SWD_D: swd_drv_transmit(0x2833cb40, 0x2832bd40) bits=8  cmdtype=MOSI_CONTROL returns=8   payload=0xffffffff (11111111)
SWD_D: swd_drv_transmit(0x2833cb40, 0x2832bd60) bits=8  cmdtype=MOSI_CONTROL returns=8   payload=0xffffffff (11111111)
SWD_D: swd_drv_transmit(0x2833cb40, 0x2832bd80) bits=8  cmdtype=MOSI_CONTROL returns=8   payload=0xffffffff (11111111)
SWD_D: swd_drv_transmit(0x2833cb40, 0x2832bda0) bits=8  cmdtype=MOSI_CONTROL returns=8   payload=0xffffffff (11111111)
SWD_D: swd_drv_transmit(0x2833cb40, 0x2832bdc0) bits=8  cmdtype=MOSI_CONTROL returns=8   payload=0xffffffff (11111111)
SWD_D: swd_drv_transmit(0x2833cb40, 0x2832bde0) bits=8  cmdtype=MOSI_CONTROL returns=8   payload=0xffffffff (11111111)
SWD_D: swd_drv_transmit(0x2833cb40, 0x2832be00) bits=8  cmdtype=MOSI_CONTROL returns=8   payload=0xffffffff (11111111)
SWD_D: swd_drv_transmit(0x2833cb40, 0x2832be20) bits=8  cmdtype=MOSI_CONTROL returns=8   payload=0xffffffff (11111111)
SWD_D: swd_drv_transmit(0x2833cb40, 0x2832be40) bits=8  cmdtype=MOSI_CONTROL returns=8   payload=0x00000000 (00000000)
SWD_I: Executing swd_dp_read_idcode(0x2833cb40, SWD_OPERATION_EXECUTE)
SWD_D: swd_drv_transmit(0x2833cb40, 0x2832be60) bits=8  cmdtype=MOSI_REQUEST returns=8   payload=0xffffffa5 (10100101)
SWD_D: swd_drv_transmit(0x2833cb40, 0x2832be80) bits=1  cmdtype=MISO_TRN     returns=1   payload=0x00000001 (00000001)
SWD_D: swd_drv_transmit(0x2833cb40, 0x2832bea0) bits=3  cmdtype=MISO_ACK     returns=3   payload=0x00000004 (00000100)
SWD_D: swd_drv_transmit(0x2833cb40, 0x2832bec0) bits=32 cmdtype=MISO_DATA    returns=32  payload=0xee2805d8 (11101110001
SWD_D: swd_drv_transmit(0x2833cb40, 0x2832bee0) bits=1  cmdtype=MISO_PARITY  returns=1   payload=0x00000000 (00000000)
SWD_I: swd_dp_read_idcode() succeeds, IDCODE=EE2805D8 (11101110001010000000010111011000)
swd_dap_detect() OK
```

```
38 IDCODE=0x0EE2805D8 (11101110001010000000010111011000)
39 Return in urj_parse_line r=0
```

1.11.5 LibSWD integration with OpenOCD

1.11.5.1 Transport Infrastructure

In order to allow access to non-JTAG devices, transport layer has been created by David Brownell to abstract operations on the bus interconnecting interface and target system. Implementation for SWD transport in OpenOCD based on LibSWD was created by Tomasz Cedro. A new interface `ft2232_swd` has been created reusing existing code for FT2232 devices. The code was developed and tested on KT–LINK interface, a first on the market inexpensive FT2232H–based design by Krzysztof Kajstura supporting both JTAG and SWD. Everything created around SWD is backward compatible and make (re)use of existing code and design, so it should be fairly easy to add new inetrfaces and transports now. Accomplishing this goal was quite a challenge, but it turned out to be possible.

1.11.5.2 Interface Signal and Bitbang Infrastructure

OpenOCD had only jtag–specific `flush_queue()` function to transfer logical data into interface hardware that was not usable for transfering bitstream for other transports. In order to implement transports other than jtag at interface level it was necessary to create bit–bang functionality allowing free control over read/write access to the interface electrical signals. Because OpenOCD has integrated TCL interpreter I have created `interface_signal` and bitbang frontend for mentioned underlying interface signal operations. Example bitbanging is presented below, it can be also used for full interface configuration at script/configuration level, so there is no need to hard–code complex routines anymore for new interfaces or transports:

```
1  %telnet localhost 4444
2  Trying 127.0.0.1...
3  Connected to localhost.
4  Escape character is '^]'.
5  Open On-Chip Debugger
6  > interface_signal list
7        Interface Signal Name      |    Mask    |   Value
8  ------------------------------------------------------------
9                              RnW | 0x00001000 | 0x00001000
10                             LED | 0x00008000 | 0x00008000
11                            srst | 0x00000A00 | 0x00000A00
12 > bitbang led=hi
13 LED=0x00008000
14 > bitbang led=lo
15 LED=0x00000000
16 > bitbang led=hi
17 LED=0x00008000
18 > interface_signal list
19       Interface Signal Name      |    Mask    |   Value
20 ------------------------------------------------------------
21                             RnW | 0x00001000 | 0x00001000
22                             LED | 0x00008000 | 0x00008000
23                            srst | 0x00000A00 | 0x00000A00
24 > bitbang led=lo
```

```
LED=0x00000000
> interface_signal list
      Interface Signal Name      |    Mask    |   Value
-------------------------------------------------------------
                             RnW | 0x00001000 | 0x00001000
                             LED | 0x00008000 | 0x00000000
                            srst | 0x00000A00 | 0x00000A00
> interface_signal
Bad syntax!
interface_signal (add|del|list) signal_name [mask]
in procedure 'interface_signal'
>
```

No need to mention that such commands can be grouped in functions and then called for some target-specific operations (i.e. SPI memory access). This wont be as fast as built-in transport but definitely will come handy for testing.

1.11.5.3 Interface configuration file and device specific scripting

`interface_signal` and bitbang brings new possibilities to interface scripting. Signals can be added dynamically at runtime with no need to hardcode them or recompile program. Signals can be driven read or write with selected values and masks allowing unlimited control over interface pinout. It is now possible to easily write external scripts for additional interface hardware, such as ADC or DAC, that was not possible before. Below is an example configuration file for KT–LINK interface (...and its example usage presented above):

```
interface ft2232_swd
ft2232_device_desc "KT-LINK"
ft2232_layout ktlink_swd
ft2232_vid_pid 0x0403 0xBBE2
interface_signal add RnW 0x1000
interface_signal add LED 0x8000
interface_signal add SRST 0x0a00
```

Later on I have also added some additional signals that comes handy in SWD transport and general driver/target testing:

```
interface_signal add SRSTin 0x0040
interface_signal add CLK 0x01
interface_signal add MOSI 0x02
interface_signal add MISO 0x04
interface_signal add nSWDsel 0x20
```

1.11.5.4 Transport initialization and IDCODE read

This is the program invocation and debug messages output:

```
%./openocd -c noinit
Open On-Chip Debugger 0.5.0-dev-g7ad8d2e-dirty (2011-07-25-15:13)
Licensed under GNU GPL v2
For bug reports, read
  http://openocd.berlios.de/doc/doxygen/bugs.html
Info : accepting 'telnet' connection from 4444
Info : only one transport option; autoselect 'swd'
```

```
Info : New SWD context initialized at 0x2843B280
10 kHz
Info : KT-LINK SWD-Mode initialization complete...
Info : max TCK change to: 30000 kHz
Info : clock speed 10 kHz
SWD_I: Executing swd_dap_activate(0x2843b280, SWD_OPERATION_EXECUTE)
SWD_I: Executing swd_dap_reset(0x2843b280, SWD_OPERATION_EXECUTE)
SWD_I: Executing swd_dp_read_idcode(0x2843b280, SWD_OPERATION_EXECUTE)
SWD_I: swd_dp_read_idcode() succeeds, IDCODE=EE2805D8 (11101110001010000000010111011000)
Info : SWD transport initialization complete. Found IDCODE=0xEE2805D8.
Warn : gdb services need one or more targets defined
User : 42 75811 command.c:557 command_print(): debug_level: 3
Debug: 43 78630 command.c:151 script_debug(): command - ocd_command ocd_command type ocd_transport init
Debug: 44 78630 command.c:151 script_debug(): command - ocd_transport ocd_transport init
Debug: 46 78630 transport.c:263 handle_transport_init(): handle_transport_init
Debug: 47 78630 swd.c:129 oocd_swd_transport_init(): entering function...
SWD_I: Executing swd_dap_activate(0x2843b280, SWD_OPERATION_EXECUTE)
Debug: 48 78630 swd_libswd_drv_openocd.c:174 swd_drv_mosi_trn(): OpenOCD's swd_drv_mosi_trn(swdctx=@0x2843B280, bits=1)
Debug: 49 78630 interface.c:41 oocd_interface_signal_find(): Searching for signal "RnW"
Debug: 50 78630 interface.c:62 oocd_interface_signal_find(): Signal RnW already exists.
Debug: 53 78631 swd_libswd_drv_openocd.c:61 swd_drv_mosi_8(): OpenOCD's swd_drv_mosi_8(swdctx=@0x2843B280, cmd=@0x28432C
Debug: 70 78646 swd_libswd_drv_openocd.c:61 swd_drv_mosi_8(): OpenOCD's swd_drv_mosi_8(swdctx=@0x2843B280, cmd=@0x28432C
Debug: 87 78662 swd_libswd_drv_openocd.c:61 swd_drv_mosi_8(): OpenOCD's swd_drv_mosi_8(swdctx=@0x2843B280, cmd=@0x28432C
Debug: 104 78778 swd_libswd_drv_openocd.c:61 swd_drv_mosi_8(): OpenOCD's swd_drv_mosi_8(swdctx=@0x2843B280, cmd=@0x28432
Debug: 121 78792 swd_libswd_drv_openocd.c:61 swd_drv_mosi_8(): OpenOCD's swd_drv_mosi_8(swdctx=@0x2843B280, cmd=@0x28432
Debug: 138 78808 swd_libswd_drv_openocd.c:61 swd_drv_mosi_8(): OpenOCD's swd_drv_mosi_8(swdctx=@0x2843B280, cmd=@0x28432
Debug: 155 78824 swd_libswd_drv_openocd.c:61 swd_drv_mosi_8(): OpenOCD's swd_drv_mosi_8(swdctx=@0x2843B280, cmd=@0x28432
Debug: 172 78840 swd_libswd_drv_openocd.c:61 swd_drv_mosi_8(): OpenOCD's swd_drv_mosi_8(swdctx=@0x2843B280, cmd=@0x28432
Debug: 189 78856 swd_libswd_drv_openocd.c:61 swd_drv_mosi_8(): OpenOCD's swd_drv_mosi_8(swdctx=@0x2843B280, cmd=@0x28432
Debug: 206 78872 swd_libswd_drv_openocd.c:61 swd_drv_mosi_8(): OpenOCD's swd_drv_mosi_8(swdctx=@0x2843B280, cmd=@0x28432
SWD_I: Executing swd_dap_reset(0x2843b280, SWD_OPERATION_EXECUTE)
Debug: 223 78888 swd_libswd_drv_openocd.c:61 swd_drv_mosi_8(): OpenOCD's swd_drv_mosi_8(swdctx=@0x2843B280, cmd=@0x28432
Debug: 240 78904 swd_libswd_drv_openocd.c:61 swd_drv_mosi_8(): OpenOCD's swd_drv_mosi_8(swdctx=@0x2843B280, cmd=@0x28432
Debug: 257 78920 swd_libswd_drv_openocd.c:61 swd_drv_mosi_8(): OpenOCD's swd_drv_mosi_8(swdctx=@0x2843B280, cmd=@0x28432
Debug: 274 78936 swd_libswd_drv_openocd.c:61 swd_drv_mosi_8(): OpenOCD's swd_drv_mosi_8(swdctx=@0x2843B280, cmd=@0x28432
Debug: 291 78952 swd_libswd_drv_openocd.c:61 swd_drv_mosi_8(): OpenOCD's swd_drv_mosi_8(swdctx=@0x2843B280, cmd=@0x28432
Debug: 308 78968 swd_libswd_drv_openocd.c:61 swd_drv_mosi_8(): OpenOCD's swd_drv_mosi_8(swdctx=@0x2843B280, cmd=@0x28432
Debug: 325 78984 swd_libswd_drv_openocd.c:61 swd_drv_mosi_8(): OpenOCD's swd_drv_mosi_8(swdctx=@0x2843B280, cmd=@0x28432
Debug: 342 79000 swd_libswd_drv_openocd.c:61 swd_drv_mosi_8(): OpenOCD's swd_drv_mosi_8(swdctx=@0x2843B280, cmd=@0x28432
Debug: 359 79016 swd_libswd_drv_openocd.c:61 swd_drv_mosi_8(): OpenOCD's swd_drv_mosi_8(swdctx=@0x2843B280, cmd=@0x28432
SWD_I: Executing swd_dp_read_idcode(0x2843b280, SWD_OPERATION_EXECUTE)
Debug: 376 79032 swd_libswd_drv_openocd.c:61 swd_drv_mosi_8(): OpenOCD's swd_drv_mosi_8(swdctx=@0x2843B280, cmd=@0x28432
Debug: 393 79048 swd_libswd_drv_openocd.c:201 swd_drv_miso_trn(): OpenOCD's swd_drv_miso_trn(swdctx=@0x2843B280, bits=1)
Debug: 404 79060 swd_libswd_drv_openocd.c:131 swd_drv_miso_8(): OpenOCD's swd_drv_miso_8(swdctx=@0x2843B280, cmd=@0x2843
Debug: 469 79124 swd_libswd_drv_openocd.c:159 swd_drv_miso_32(): OpenOCD's swd_drv_miso_32(swdctx=@0x2843B280, cmd=@0x28
Debug: 470 79124 swd_libswd_drv_openocd.c:160 swd_drv_miso_32(): OpenOCD's swd_drv_miso_32() reads: 0xEE2805D8
Debug: 473 79126 swd_libswd_drv_openocd.c:131 swd_drv_miso_8(): OpenOCD's swd_drv_miso_8(swdctx=@0x2843B280, cmd=@0x2843
SWD_I: swd_dp_read_idcode() succeeds, IDCODE=EE2805D8 (11101110001010000000010111011000)
Info : 474 79126 swd.c:147 oocd_swd_transport_init(): SWD transport initialization complete. Found IDCODE=0xEE2805D8.
```

This is the user action that makes the above output – for demonstration purpose this is done by hand with OpenOCD's remote TCL interpreter using Telnet client program:

```
%telnet localhost 4444
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
> source [find interface/kt-link-swd.cfg]
only one transport option; autoselect 'swd'
New SWD context initialized at 0x2843B280
> adapter_khz 10
10 kHz
> init
```

```
12 KT-LINK SWD-Mode initialization complete...
13 max TCK change to: 30000 kHz
14 clock speed 10 kHz
15 SWD transport initialization complete. Found IDCODE=0xEE2805D8.
16 gdb services need one or more targets defined
17 > transport init
18 SWD transport initialization complete. Found IDCODE=0xEE2805D8.
```

1.12 JTAG / IEEE1149.1

JTAG stands for *Joint Test Action Group* and is a common name for *IEEE1149.1* protocol used for various testing mechanisms in silicon manufacturing, printed circuit board (PCB) connection verification, early stage embedded system development, and many more. JTAG requires target system to have special connector with standard–defined signals (TDI, TDO, TCK, TMS, TRST) and special silicon block residing on integrated circuit that allows JTAG operations. Connection is possible using dedicated interface that interconnects target system with a personal computer running software that can interact with the silicon block using JTAG as command transport system. Internal register access can be used to verify proper functionality (i.e. after fresh silicon fabrication), connections on the board and communication with other devices such as flash memory, driving the input–output pins, or even debugging program execution when dedicated In–Circuit–Emulation is embedded into CPU...

JTAG was adopted as IEEE [8] standard in 1990, also that was the year when 80486 CPU was released by Intel Corporation [48] becoming first JTAG–aware device ever. Since then the number of devices supporting this standard was constantly growing. Nowadays almost all CPU are equipped with JTAG port, however not all of them are freely accessible as JTAG is only a transport interconnecting user software with complex subsystems hidden in silicon and protected by NDA (Non Disclosure Agreement) by their manufacturers. Also number of tools is limited by their accessibility, very high price and narrow target device family orientation.

JTAG is the last resort of rescue in case of logical device damage as it may allow to restore broken firmware to a bricked device, companies use it for early stage development or servicing purposes, it is also very dangerous backdoor into device security as it allows access to information at lowest possible level – hardware registers and operating system internals.

1.12.1 JTAG Technical Reference

1.12.1.1 Introduction

This section shows the internal organization of JTAG subsystem, how it is organized at physical and logical level, and how it can be used to access internal target or CPU registers in order to perform actions such as input–output or memory access. JTAG is general transport mechanism, therefore we will not cover accessing the registers that are out of scope of this standard, as they are usually device specific and differs among different

targets. Also the implementation of the JTAG Port itself may vary across different devices, as not features are supported or necessary in each case.

JTAG can be accessed usually at dedicated physical connector that gives access to the Test Access Port (TAP) or more general Debug Access Port (DAP), depending on device family and architecture. This standard is not limited to some specific manufacturer or its device. Successful connection to TAP/DAP gives access to TAP/DAP internal registers. There are only two kind of registers: Data and Instruction. Manipulating values of that registers can trigger actions on internal functional block registers responsible for specific functionality such as I/O, memory, or debug actions. Those functional blocks are usually vendor specific and each of them require separate implementation as a tool that can make use of its features. This is why having JTAG access not always result in successful operation on device internals – because they are device specific, often kept in secret by the manufacturer. We will base on publicly available description of the ARM JTAG Debug Port being part of the *ARM Debug Interface* [44] standard.

1.12.1.2 Signalling

The JTAG connection requires at least five standard signals:

- TDI (Test Data Input) – bitstream input to the target system
- TDO (Test Data Output) – bitstream output from the target system
- TCK (Test ClocK) – clock signal for synchronous state machine
- TMS (Test Mode Select) – control signal selecting mode of operation
- TRST (Test ReSeT) – additional debug logic reset (not system reset)

Additionaly there are two power signals necessary (Vcc, GND) and sometimes SRST (System ReSeT) is also available for easier system reset during development. Because JTAG is a synchronous state machine it requires clocking signal, but its frequency range may vary depending on the target system state (i.e. in power down the acceptable clock may be far slower than usual), so the *Adaptive Clocking* teqnique was introduced using additional RTCK (Return CLK) signal returning clock pulse after each bit has been shifted. This was it is possible to maintain optimal connection speed. Sometimes it is also possible to emulate TRST pin behavior with special sequence of TMS and TCK signals, but some chips will not respond at all on the JTAG port when TRST is not connected (i.e. HTC Nexus One).

The power supply is often necessary to be connected to the interface logic to provide reference voltage for input–output buffers that are placed between interface CPU and target CPU/TAP. Target can use different power supply voltage (i.e. 1,8V) than interface (i.e. 3,3V) or the user computer system (i.e. 5V). Buffers are then used as voltage–shifters or negotiators preventing pernament damage to the systems that are interconnected at different voltage levels. Note that different interfaces can have different capabilities of

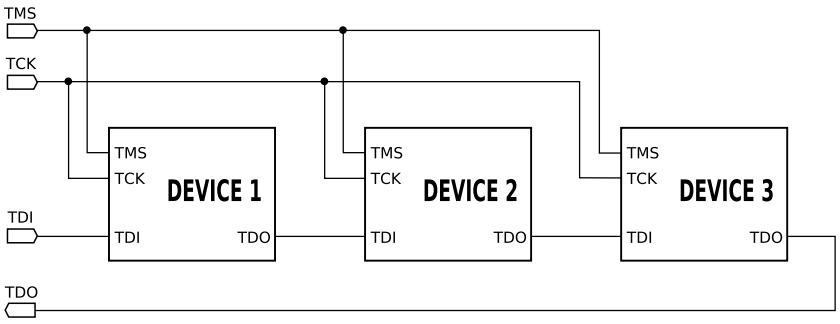


Figure 1.20: Daisy–chaining JTAG multiple devices [22].

such voltage translation, some of them does not have this feature at all, so make sure what logic is used before connecting to the target system.

Therefore physical JTAG connection requires at least 6 wires to work (TDI, TDO, TCK, TMS, VCC, GND), sometimes 7 (when TRST is obligatory) and sometimes even 8 (when RTCK is available).

1.12.1.3 Daisy Chaining

Because JTAG operation is a bistream shift through internal device registers, it is possible to connect output of one device into another device input (and so on) to create a serial daisy–chain connection with multiple devices available to work. Such devices must be equipped with JTAG compliant connector and internal logic. Speed of such connection is then decreased and only one device can be active with other switched into `BYPASS` mode simply passing input to output (with one bit delay).

1.12.1.4 JTAG Debug Port

`JTAG-DP` contains a debug port state machine (JTAG) that controls the `JTAG-DP` operation, including controlling the scan chain interface that provides the external physical interface to the `JTAG-DP`. It is based closely on the JTAG TAP State Machine, see *IEEE Std 1149.1 (2001)*.

`nTRST` signal (*n* stands for „negative", that is „active low" Target ReSeT signal activated with 0) asynchronously takes the JTAG state machine logic to the Debug–Logic–Reset state. Debug–Logic–Reset state can also always be entered synchronously from any state by a sequence of five `TCK` cycles with `TMS` high, however depending on the initial state of the JTAG this might take the state machine through one of the Update states, with the resulting **side effects**.

When the JTAG goes through the Capture–IR state, a value is transferred onto the Instruction Register (`IR`) scan chain. The `IR` scan chain is connected between `TDI` and `TDO`. While the JTAG is in the Shift–IR state, and for the transition from Capture–IR to Shift–IR, the `IR` scan chain advances one bit for each tick of `TCK`. This means that on the first tick, the LSB of the `IR` is output on `TDO`, bit `[1]` of the `IR` is transferred to bit `[0]`,

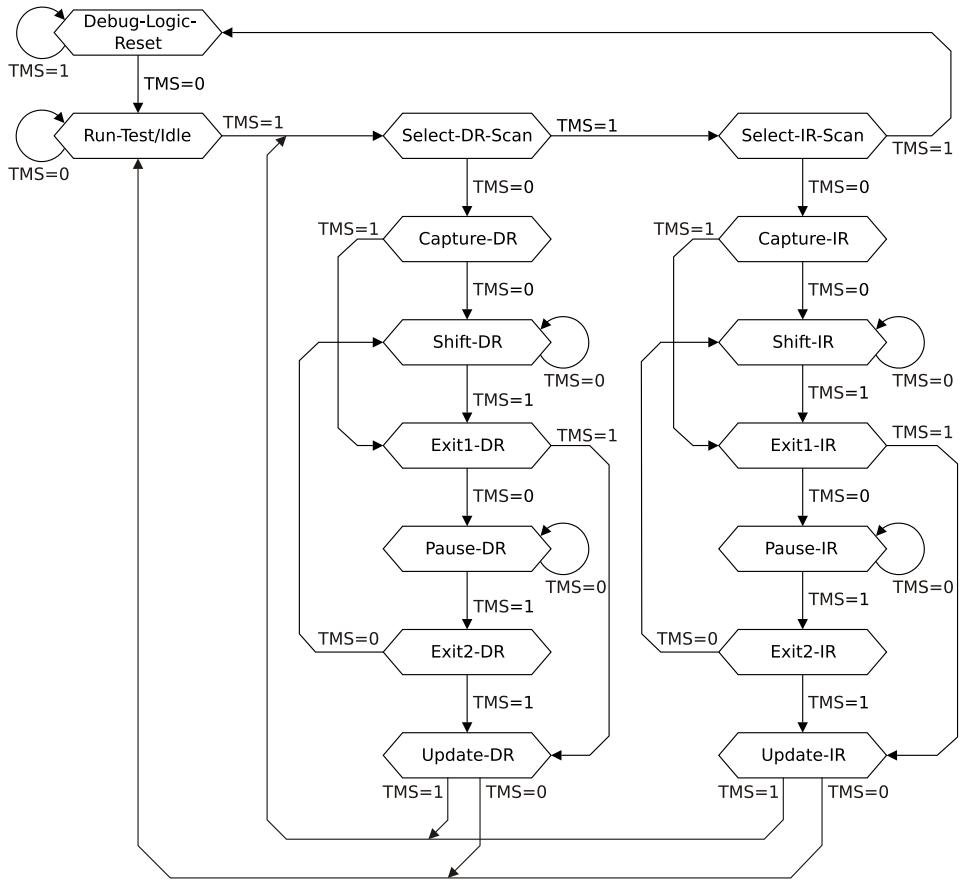


Figure 1.21: JTAG State Machine [46].

bit [2] is transferred to bit [1], for example. The MSB of the IR is replaced with the value on TDI.

When the JTAG goes through the Update–IR state, the value scanned into the scan chain is transferred into the Instruction Register. When the JTAG goes through the Capture–DR state, a value is transferred from one of a number of Data Registers (DR) onto one of a number of Data Register scan chains, connected between TDI and TDO. This data is then shifted while the JTAG is in the Shift–DR state, in the same manner as the IR shift in the Shift–IR state. When the JTAG goes through the Update–DR state, the value scanned into the scan chain is transferred into the Data Register. When the JTAG is in the Run–Test/Idle state, no special actions occur. Debuggers can use this as a true resting state.

1.12.1.5 JTAG Debug Port Registers

The JTAG–DP registers are only accessed when the Instruction Register (IR) for the DAP access contains the IDCODE, DPACC, or ABORT instruction. The JTAG–DP register accessed depends on both Instruction Register (IR) value for the DAP access and the address field

of the DAP access. Each CPU or other device can use different set of registers with different functionality! JTAG is onaly a transport mechanism to access those registers and use them according to the device specification. Registers presented below are specific to ARM ADIv5 architecture.

1.12.1.6 JTAG Instruction Register (IR)

JTAG `IR` is a 4–bit register that holds the current DAP (Debug Access Port) Controller instruction. On debug logic reset, `IDCODE` becomes the current instruction. If the `IR` register is set to `IR` instruction value out of scope (that is not implemented, or reserved), then the Bypass Register is selected.

- ABORT (b1000, DR: 1bit) – Access the DP Abort Register, to force a DAP abort.
- DPACC (b1010, DR: 35bits) – JTAG DP Access Registers – Initiate a Debug Port (`DP`) or Access Port (`AP`) access, to access a debug port or access port register. The `DPACC` and `APACC` are used for read and write accesses to registers – `DPACC` to access the `CTRL/STAT`, `SELECT` and `RDBUFF` registers, while `APACC` is used to access all of the access port registers.
- APACC (b1011, DR: 35bits) – JTAG AP Access Registers – The `DPACC` and `APACC` scan chains have the same format. See `DPACC` (above).
- IDCODE (b1110, DR: 32bits) – JTAG Device ID Code Register – Device identification. The Device ID Code value enables a debugger to identify the debug port to which it is connected. Different debug ports have different Device ID Codes, so that a debugger can make this distinction.
- BYPASS (b1111, DR: 1bit) – JTAG Bypass Register – Bypasses the device, by providing a direct path between `TDI` and `TDO`.

1.12.2 JTAG Data Register (DR)

There are five physical `DR` registers: `BYPASS, IDCODE, DPACC, APACC, ABORT`. There is a scan chain associated with each of these registers – the `IR` register determines which of these scan chains is connected to the `TDI` and `TDO` signals.

- ABORT (IR: b1111, DR: 1bit) – Access the DP Abort Register, to force a DAP abort. The debugger must scan the value `0x0000008` into this scan chain. This value writes the `RnW` bit as `0`, `A[3:2]` field as `b00`, `1` into bit `0`, the `DAPABORT` bit, of the Abort Register.
- – IDCODE (IR:b1110, DR: 32bit) – For the STM32 ARM–Cortex family these fields are as shown below. Please note that this string should be different for every device family (even hardware revision). Version, bit `[31:28]`, is set to `3`. Part number, bit `[27:12]`, is set to `0xBA00`. Manufacturer ID, bit `[11:1]`, is set to `0x23B`. Reserved, bit `[0]`, is set to `1`.

- DPACC/APACC (DPAC_IR:b1010, APAC_IR:b1011, DR: 35bit) – Initiate a debug port or access port access, to access a debug port or access port register. In the Capture–DR state, the result of the previous transaction, if any, is returned, together with a 3–bit `ACK` response. Only two `ACK` responses are implemented (all others reserved):
  - `b010 OK/FAULT` response to a `DPACC` or `APACC` access: If the response indicated by `ACK[2:0]` is `OK/FAULT`, the previous transaction has completed. The response code does not show whether the transaction completed successfully or was faulted. You must read the `CTRL/STAT` register to find whether the transaction was successful:
    - If the previous transaction was a read that completed successfully, then the captured `ReadResult[31:0]` is the requested register value. This result is shifted out as `Data[34:3]`.
    - If the previous transaction was a write, or a read that did not complete successfully, the captured `ReadResult[31:0]` is Unpredictable. If `Data[34:3]` is shifted out it must be discarded.
  - `b001 WAIT` response to a `DPACC` or `APACC` access: A `WAIT` response indicates that the previous transaction has not completed. The host should retry the `DPACC` or `APACC` access. Normally, if software detects a `WAIT` response, it retries the same transfer. This enables the protocol to process data as quickly as possible. However, if the software has retried a transfer a number of times, permitting enough time for a slow interconnect and memory system to respond, it might write to the `ABORT` register, to cancel the operation. This signals to the active access port that it can terminate the transfer it is currently attempting and permits access to other parts of the debug system. An access port might not be able to terminate a transfer on its ASIC interface. However, on receiving an `ABORT`, the access port must free its JTAG interface.

Operation in the Update–DR depends on whether the `ACK[2:0]` response was `OK/FAULT` or `WAIT`. The two cases are described in:

- Update–DR operation following an `OK/FAULT` response.
- Update–DR operation following a `WAIT` response: No request is generated at the Update–DR state and the shifted–in data is discarded. The captured value of `ReadResult[31:0]` is Unpredictable. You can detect a `WAIT` response without shifting through the entire `DP` or `AP` Access Register.

If `IR=DPACC` then read/write to `DP`, `IR=APACC` then read/write to `AP` register. `RnW=0` write into `DATAIN[31:0]`, `RnW=1` read from addressed register in next read cycle: `DPACC` selects `A[3:2]`, `APACC` selected by `A[3:2]` and the `SELECT DP` register. Register accesses can be pipelined, because a single `DPACC` or `APACC` scan can return the result of the previous read operation at the same time as requesting another register access. At the end of a

sequence of pipelined register reads, you can read the DP RDBUFF Register to return the result of the final register read.

If the current IR instruction is APACC, causing an APACC access:

- If any sticky flag is set in the DP CTRL/STAT Register, the transaction is discarded. The next scan returns an OK/FAULT response immediately.
- If pushed compare or pushed verify operations are enabled then the scanned-in value of RnW must be 0, otherwise behavior is Unpredictable. On Update–DR, a read request is issued and the returned value compared against DATAIN[31:0]. The STICKYCMP flag in the DP CTRL/STAT register is updated based on this comparison.
- The AP access does not complete until the access port signals it as completed.

Sticky overrun behavior on DPACC and APACC accesses: At the Capture–DR state, if the previous transaction has not completed a WAIT response is generated. When this happens, if the Overrun Detect flag is set, the Sticky Overrun flag, STICKYORUN, is set. While the previous transaction remains not completed, subsequent scans also receive a WAIT response. When the previous transaction has completed, any more APACC transactions are abandoned and scans respond immediately with an OK/FAULT response. However, debug port registers can be accessed. In particular the CTRL/STAT register can be accessed, to confirm that the Sticky Overrun flag is set and to clear the flag after gathering any required information about the overrun condition.

Most common situations handling:

- Read Operation returns OK/FAULT – Capture read data.
- Write Operation returns OK/FAULT – No more action required.
- Read or Write Operation returns WAIT – Repeat the same access until either an OK/FAULT ACK is received or the wait timeout is reached. If necessary, use the DAP ABORT register to enable access to the AP.
- Read or Write Operation returns Invalid ACK – Assume a target or line error has occurred and treat as a fatal error.

1.13 Brain Computer Interface Open Protocol

1.13.1 Introduction

This section describes first approach to create free, open–source and compact protocol for data exchnage with Brain Computer Interface class devices. Because there are already few places on Earth where BCI are being develpoed, but none of them conforms to well known standard, I have decided to start new project called *Brain Computer Interface Open Protocol*, so many different designs could exchange information in a well known and

documented manner. This is fresh project and a lot of work has to be put to make it what it is supposed to be, but the first step is being made right now.

Because a C programing language based program can work on almost any hardware this computer language is used to create sources of the BCIOP, that can be later included into a bigger project source tree, exported as a static library or connected to an external application as dynamic library. Currently it is a part of the protytype source code.

1.13.2 BCIOP Overwiew

To minimise data overhead TLV was used to transfer data in binary manner, that is Tag–Length–Value builds one data packet. Tag determines function, Length determines data lenght with octet (octet = 8bits) as base unit. Value is the payload.

BCIOP does not care about packet routing or addressing – this should be done by lower layers protocols. BCIOP is intended for experiments and prototyping, but also production. Each TLV packet by default is forwarded by `10101010b` (`0xAA`) octet to allow sychronization and debugging. This can be turned off to maximize thruput. Also data format can be set or changed in a runtime. Simple error correction packet can be appended to ensure valid transmission (off by default). If device detects some data on the bus, but cannot recognise them, it should send bursts of SYNC packets to help host detect transmission speed and perform auto–baudrate procedure.

To initiate connection, host sends `INIT` packet with `REQUEST` payload to the device, and receives `INIT` packet with `RESPONSE` payload as response with the protocol number used for communication. Then data format and time format has to be determined with `DATA FORMAT` and both `TIMESTAMP FORMAT` and `TIMESTAMP QUANT` requests containing appropriate payload. Depending on the hardware used data can be signed or unsigned, 8-bit (1 octet), 16-bit (2 octets), 24-bit (3 octets), NKB, U2, Gray, etc. If device supports this feature, timestamping can be used for futher synchronisation. To make it easier for programmers to perform data casting in their software, device suggests target data type (signed, unsigned, char, integer, long, etc).

Commands are divided into categories – setup (Tag MSB set to zero) and transfer (Tag MSB set to one). When host sends transfer packet to the device with length byte set to zero, this is a request for specific transfer – for example data burst with timestamp, or simple timestamp to verify synchronisation.

Device or host can send `ERROR` packet in case of failure condition – that is when no initialisation was performed, no data format was selected, buffer overflow, or similar. When the condition is not critical, `ERROR` packets serves only as additional information and can be supressed to avoid interference with data. This can be useful in case of minimalistic hardware with unidirectional interface, when only partial functionality is implemented - host should receive and interpret data properly, as long as TLV tags are properly used and protocol specification is maintained. Delivery acknowlegement can be turned on in response to a data packet, but any packet with erratic length or value field (ie. out of boundaries) must be discarded without concern or error message – this will allow detect errors by timeout and avoid packet flood.

1.13.3 BCIOP Packet Details

BCIOP PACKET		
TAG (T) 1 octet length	LENGTH (L) 1 octet length	VALUE (V) L octets length

1.13.3.1 TAG 0x01: ERROR

LENGTH {0,1}:
L=0: Last error code request, L=1: Value octet holds error information

VALUE:

- V[1]: Error information
  - Fatal errors (MSB set to one):
    - 0x80: UNKNOWN, internal error
    - 0x90: ADC, no communication
    - 0x91: ADC, internal error
    - 0xA0: DAC, co communication
    - 0xA1: DAC, internal error
  - Non-Fatal errors (MSB cleared to zero):
    - 0x00: No INIT performed
    - 0x01: No DATA FORMAT request was performed
    - 0x02: No TIMESTAMP FORMAT request was perormed
    - 0x03: No TIMESTAMP QUANT set was perormed
    - 0x04: No SAMPLING FREQIENCY request was performed
    - 0x05: Feature not supported
    - 0x10: Internal Buffer empty, wait for data
    - 0x11: Internal Buffer full, read more data

1.13.3.2 TAG 0x01: INIT

LENGTH {0,1}:
L=0: Initialisation request, L=1: Initialisation response

VALUE:

- V[1]: Current protocol version as BCD pair number (ie. 0x10 for 1.0 release)

1.13.3.3 TAG 0x02: IDENT

LENGTH{0,N}:
L=0: Identification request, L=N: N value octets hold identification response

VALUE:

- V[n]: N bytes ASCII of identification string

1.13.3.4 TAG 0x03: DATA FORMAT

LENGTH {0,2}:
L=0: Data format request, L=2: 2 value octets hold data format information

VALUE:

- V[1]: Number of octets per Data Sample: 1,2,3,..,8
- V[2]: (1000 0000 AND DataSign) OR (0111 0000 AND DataCode) OR (0000 1111 AND DataCast)
  - DataSign: 0x80: Signed, 0x00: Unsigned
  - DataCode: 0x01: Binary (NKB), 0x02: Gray, 0x03: Two-complement (U2)
  - DataType: 0x01: char (1 octet), 0x02: short integer (2 octet), 0x03: integer (4 octet), 0x04: long (8 octets)

1.13.3.5 TAG 0x04: TIMESTAMP FORMAT

LENGTH {0,1}:
L=0: Timestamp format request, L=1: Value octet holds timestamp format information response

VALUE:

- V[1]: Number of Octets per Data: 1,2,3,4 (max. 4). Unsigned.
- V[2]: Data Format: (1000 0000 AND TimestampSign) OR (0111 111 AND TimestampCast)
  - TimestampSign:
    - 0xFF: Signed
    - 0x00: Unsigned
  - TimestampCast:
    - 1: char (1 octet)

  - 2: short integer (2 octet)
  - 3: integer (4 octet)
  - 4: long (8 octets)

1.13.3.6 TAG 0x05: TIMESTAMP QUANT

LENGTH {0,1,4}:
L=0: Timestamp quant request, L=1 or L=4 Value octets hold timestamp quant value

VALUE:

- if L=1 then V[1]: Time quant (unsigned) is set to multiply of 10 microseconds:

$$10\mu s * \{1..255\}$$

.

- if L=4 then V[1..4]: Time quant (unsigned) is set to multiply of 1 nanosecond:

$$1ns * \{1..2^{32}\}$$

.

1.13.3.7 TAG 0x06: TIMESTAMP

LENGTH {0,N}:
L=0: Timestamp value request, L=N: N value octets hold current timestamp information response

VALUE:

- V[n]: N octets of current device timestamp value conforming to TIMESTAMP FORMAT

1.13.3.8 TAG 0x07: SAMPLING FREQUENCY

LENGTH {0,1}: L=0: Sampling frequency request, L=1: Value octet holds sampling frequency information response/request

VALUE:

- V[1]=n: Sample every n-th timequant.

1.13.3.9 TAG 0x10: ACQUISITION STATUS

LENGTH {0,1}
L=0: Acquisition status request, L=1: Acquisition status response/command

VALUE:

- V[1]: Acquisition Status
  - 0x00: Not running
  - 0x01: Single shot mode
  - 0x02: Free-runing mode
  - 0xFF: Error

Note: Single Shot mode is activated after device receives Acquisition Status packet with L=1 and V=0x01. Free-runing mode starts working after device receives Acquisition Status packet with L=1 and V=0x02.

1.13.3.10 TAG 0x80: CHANNEL

LENGTH {0,1}:
L=0: Current channel number request, L=1: Value octet holds channel number information response/requests

VALUE:

- V[1]: Current channel number. Unsigned.

1.13.3.11 TAG 0x81: DATA

LENGTH {0,N}:
L=0: Data request, L=N: N value octets hold data response

VALUE:

- V[n]: N octets of data conforming to DATA FORMAT

1.13.3.12 TAG 0x82: DATA BURST

LENGTH {0,N}:
L=0: Data burst request, L=N: N-1 Value octets hold burst of data reply

VALUE:

- V[1]: 1 octet with unsigned number of data samples

- V[2..N]: (N-1) octets of data conforming to DATA FORMAT

Note: Number of samples depends on data format and cannot exceed 254 (in case of 1 octet data)

1.13.3.13 TAG 0x83 DATA+TIMESTAMP

LENGTH {0,N}:
L=0: Data+Timestamp request, L=N: N value octets hold data+timestamp response

VALUE:

- V[X]: X octets of timestamp value conforming to current TIMESTAMP FORMAT
- V[N-X]: (N-X) octets of data conforming to DATA FORMAT

Note: This command cannot be send or received before timestamp and data format is set - this is should produce failure condition.

1.13.3.14 TAG 0x84: (DATA+TIMESTAMP) BURST

LENGTH {0,N}:
L=0: Data burst request, L=N: N value octets hold data+timestamp burst

VALUE:

- V[1]: 1 octet with unsigned number of data+timestamp pairs
- V[N-1]: burst of (N-1) octets of timestamp conforming to TIMESTAMP FORMAT + Y octets of data conforming to DATA FORMAT

Note: Number of samples depends on data and timestamp format and cannot exceed 127 (in case of 1 octet data and 1 octet timestamp)

1.13.3.15 TAG 0x85: DATA+TIMESTAMP+CHANNEL

LENGTH {0,N}:
L=0: Data+Timestamp+Channel request, L=N Data+Timestamp+Channel response

VALUE:

- V[1]: 1 octet of channel value (unsigned)
- V[2..5]: 4 octets of timestamp value (unsigned)
- V[6..N]: (N-5) octets of data conforming to DATA FORMAT

1.13.3.16 TAG 0x86: (DATA+TIMESTAMP+CHANNEL) BURST

LENGTH {0,N}
L=0: Data + timestamp + channel packs burst request

VALUE:

- V[1]: 1 octet with unsigned number of data+timestamp+channel packs
- V[1..N]: burst of (N-1) octets of: timestamp conforming to TIMESTAMP FORMAT + 1 octet of channel number + Y octets of data conforming to DATA FORMAT

Note: Number of samples depends on data and timestamp format and cannot exceed 85 (in case of 1 octet data 1 octet channel and 1 octet timestamp)

1.13.3.17 TAG 0x87: TIMESTAMP+(DATA+CHANNEL) BURST

LENGTH {0,N}:
L=0: One timestamp + data and channel packs burst request, L=N: N value octets holding response

VALUE:

- V[1]: 1 octet with unsigned number of data+channel packs
- V[2..M]: (M-2) octets with timestamp conforming to TIMESTAMP FORMAT
- V[M+1..N]: burst of 1 octet of channel number + Y octets of data conforming to DATA FORMAT.

Notes: This packet is intended to transfer only one measurement of multichannel sampling device acquired simlutaneously (with no use of multiplexer). Device logic should remember timestamp for each sample. When timestamp of one sample is different than the others, error packet should be send to the host, becuase phase errors will occur. If device uses multiplexer and do not perform simultaneous multichannel sampling, (DATA + TIMESTAMP + CHANNEL) BURST should be used. Number of samples depends on data and timestamp format and cannot exceed 126 (in case of 1 octet data 1 octet channel and 1 octet timestamp) CODE NAME LENGTH VALUE

1.14 Xilinx Software and Hardware

This section presents basic knowledge required to setup a working environment to program FPGA devices.

1.14.1 Introduction to FPGA programming

In order to use Xilinx (or other) FPGA devices a *bitfile* must be created to tell the internal organization of the device logic cells. Bitfile is a kind of firmware program that must be preprogrammed into internal or external nonvolatile ROM/Flash memory and then uploaded into internal FPGA's gate array at powerup to designate its functions. Memory type depends on target device but in general it can use serial interface (for smaller footprint and pin consumption) or parallel interface (for faster access) with command set and internal organization supported both by software tools and the target itself.

Bitfile can be created with dedicated software development toolkit, in our case *Xilinx ISE* [57], that is available for free after registration with its basic functionalities which can be further extended with additional commercial software components. I will use free of charge *Xilinx ISE Web Pack* for all further actions concering project componenets based on FPGA technology. All techiques mentioned above are presented in more detail in following sections.

1.14.2 Known issues

Bitfile can be uploaded into nonvolatile Flash memory or directly into FPGA configuration memory using JTAG interface. When using flash memory it may use non–linear address space so first it needs to be converted into *PROM File* which is a *bitfile* but remapped to match physical memory organization. Only few memory types are supported by Xilinx ISE, also official Xilin Developmen Board that I have used (`Spartan 3A-DSP`) had a memory installed that was not supported by Xilinx ISE [61] so I had to purchase different chip and replace it manually otherwise pretty expensive board would have no use.

Another problem a developer will encounter is the JTAG programming done by Xilinx USB Platform Cable which is pretty expensive and works practically only under Windows in default setup. It is possible to overcome driver limitations using Open–Source softare, but the environment will differ a bit from the one available for Windows platform, but it should work on all other systems instead. It is possible to use different, cheap and popular, JTAG interfaces for FPGA programming – the solution is presented later in this section. Although I will work on FreeBSD platform using Linux binaries, I recommend to download bundled Xilinx ISE package with Linux and Windows installer as it may come useful for advanced testing or Open–Source solutions verification and sometimes, unfortunately, the solution for Windows is still the only one available.

1.14.3 Installing Linux version of Xilinx ISE on FreeBSD OS

Xilinx ISE is available only for Windows and Linux operating systems. Because I use FreeBSD [34] that can emulate Linux binaries, this section describes procedure for installation and configuration of software on alternative operating system with some troubleshooting guide related to system environment diferrenes and some issues with ISE binaries. This is very interesting and innovative possibility to natively run binaries from

other operating system. My solution is also presented publicly on FreeBSD Forums [59] and linked on Xilinx Forums [58] related threads.

At the time of writing this document current version of ISE is 13.1 available to download as `Xilinx_ISE_DS_13.1_O.40d.1.1.tar` file. We need to obtain the package, extract files, setup Linux dynamic libraries on FreeBSD platform and start the installation:

- Obtain and extract the installer.

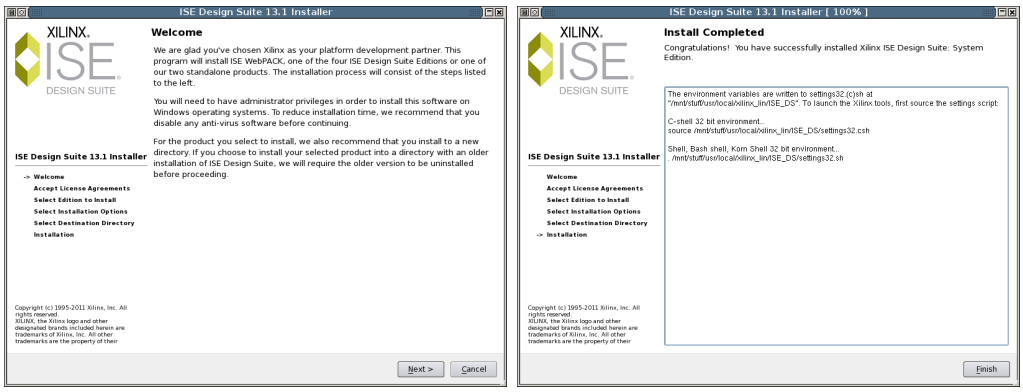
```
%tar xvf Xilinx_ISE_DS_13.1_O.40d.1.1.tar
x Xilinx_ISE_DS_13.1_O.40d.1.1/
x Xilinx_ISE_DS_13.1_O.40d.1.1/Microsoft.VC90.CRT/
x Xilinx_ISE_DS_13.1_O.40d.1.1/Microsoft.VC90.CRT/Microsoft.VC90.CRT.manifest
x Xilinx_ISE_DS_13.1_O.40d.1.1/Microsoft.VC90.CRT/msvcm90.dll
...
```

- Check for Linux dependencies and their availability on FreeBSD platform, install if necessary. Do not use wrapper scripts (such as top level `xinfo` or `xsetup`).

```
% cd Xilinx_ISE_DS_13.1_O.40d.1.1/
% ./bin/lin/xsetup
/mnt/stuff1200/usr/local/Xilinx_ISE_DS_13.1_O.40d.1.1/bin/lin/_xsetup:
 error while loading shared libraries: /usr/X11R6/lib/libgthread-2.0.so.0: ELF file OS ABI invalid
```

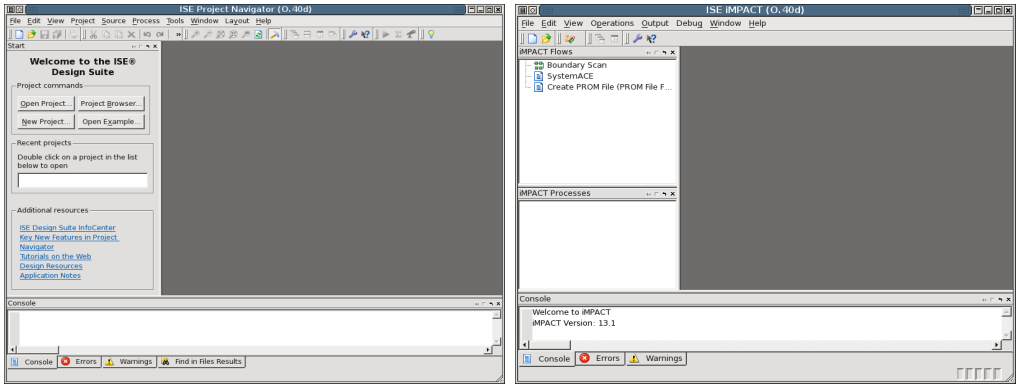
  The problem is with libraries (i.e. `libgthread-2.0.so.0`) not being found in `/usr/X11R6/lib` (this Linux location in fact is `/usr/compat/linux/usr/X11R6/lib` in our FreeBSD host system) as the path is unfortunately hardcoded into Xilinx binaries (normally the loader program search for dynamic libraries in a system dependent fashion as specified by `ld` and `ldconfig` so it should find dynamic libraries when they are on search path and this is why I don't like Linux anymore because it keep no standard at all). If the libraries are not on your system get them automatically using `port` subsystem (`man ports`). The libraries will be installed in `/usr/lib` instead `/usr/X11R6/lib` (from Linux perspective), so they need to be relinked. Because in FreeBSD `/usr/compat/linux/usr/X11R6/lib` contains no libraries it is simpler to relink directory with necessary libraries in place of empty one, then again link some libraries that are placed in other places. This can be done using the following script (as `root`) from the Xinix ISE installer path (it will also copy and add bundled libraries to the emulation layer, but again unfortunately Xilinx does not provide all dependencies anymore with the installation package).

```
#!/bin/sh
#(C) 2011 Tomasz Boleslaw CEDRO (http://www.tomek.cedro.info)
mkdir /usr/compat/linux/usr/lib/xilinx
cp lib/lin/* /usr/compat/linux/usr/lib/xilinx/
echo "/usr/compat/linux/usr/lib/xilinx" >> /usr/compat/linux/etc/ld.so.conf.d/xilinx.conf
mv /usr/compat/linux/usr/X11R6/lib /usr/compat/linux/usr/X11R6/lib_orig
echo "/usr/compat/linux/usr/X11R6/lib_orig" >> /usr/compat/linux/etc/ld.so.conf.d/x11r6_orig.conf
ln -s /usr/compat/linux/usr/lib /usr/compat/linux/usr/X11R6/lib
ln -s /usr/compat/linux/lib/libgthread-2.0.so.0 /usr/compat/linux/usr/X11R6/lib/libgthread-2.0.so.0
ln -s /usr/compat/linux/lib/libglib-2.0.so.0 /usr/compat/linux/usr/X11R6/lib/libglib-2.0.so.0
ln -s /usr/compat/linux/lib/libuuid.so.1 /usr/compat/linux/usr/X11R6/lib/libuuid.so.1
/usr/compat/linux/sbin/ldconfig -r /usr/compat/linux -i
```



(a) Installation start. (b) Installation complete.

Figure 1.22: Xilinx ISE Design Suite installation spash screen. Linux binary working on FreeBSD operating system.



(a) ISE project manager and editor. (b) iMpact FPGA programming tool.

Figure 1.23: Xilinx ISE Design Suite components readu for use on FreeBSD.

- Start the installation with `sudo ./bin/lin/xsetup` in the installation top directory. You need to use `sudo` (install with `pkg_add -r sudo` if not present then `man sudo` and edit the configuration to allow selected users running binaries ar `root`) because installer needs the windows from the Xorg server and this cannot be use by `root` when you are logged in as an ordinary user.

- During the installation it is possible to select license that enable some commercial modules. The basic ISE Web Pack offers compiles, simlator and programming utilities for free. It is also possible to have 30–dat free trial of additional commercial components (after online registration).

- When the installation is complete ISE (project manager and editor, command `ise`) and iMPACT (programming tool, command `impact`) tools are ready to use from `ISE_DS/ISE/bin/lin/` subdirectory of a selected install location.

- Unfortunately drivers for JTAG interfaces are the worst part of the Xilinx software and in fact there is no support for JTAG Xilinx USB Platform Cable at this moment in default Xilinx ISE configuration on FreeBSD/Linux as this requires compilation of deidcated Linux Kernel Driver that does not build even on Linux itself. There are however some tricks to overcome this limitation, as presented later in this section, based on Open–Source `fxload` application to upgrade cable firmware and `xcs3prog` utility to program Xilinx FPGA devices using some selected interfaces, both available in the FreeBSD port tree. If you need to have full functionalities of iMPACT you need to use Windows version of software, there is also separate option in the setup program to install only programming utilities.

1.14.4 Programming the FPGA target device

Xilinx ISE is delivered with iMPACT utility for programming the target devices, flash memories, JTAG chains, etc. Because Xilinx USB based JTAG interface drivers use `windrv` there are problems using them on platforms other than Windows. This is why other mechanisms have to be used in order to upload bitfile into target device or its memory.

1.14.4.1 Recording actions to SVF file

First and most universal way to upload bitfile into target device when no USB support in iMPACT is available relies on using virtual cable which redirects JTAG actions generated by iMPACT into *SVF* (Serial Vector Format [67]). SVF file can be then „replayed" in any program that supports this format, for instance Open–Source UrJTAG [37], with any JTAG interface supported by that program (i.e. not necessairly Xilinx USB Platform Cable).

This soltion can be used when some specific feature/action of iMPACT must take place. Free SVF players however seems to be buggy and not always work as expected. This is also the only choice when no other solution is available.

1.14.4.2 Using user–space driver

iMPACT since release 9 supports userland drivers for JTAG interface. There is an Open–Source project lead by Michael Gernoth [68] that use multiplatform `libusb` for communicaton with JTAG device instead proprietary kernel driver (that does not build), but this seems to be still unreliable solution for many platforms including FreeBSD therefore I will not discuss it here. It is mainly concentrated on making Xilinx cable usable (which should be the Xilinx's task) on platforms other than Windows, but there is experimental support for popular and cheap FT2232–based interfaces.

1.14.4.3 Bitfile programming with XC3SPROG

The best way as it turned out in practice was to use standalone Open–Source application `xc3sprog` [60] that completely removes requirement for using iMPACT, because it can

directly program Xilinx FPGA device with pure bitfile, programming externel SPI Flash memory is also supported. There is no need to use iMPACT at all! Application not only supports all Xilinx JTAG cables but also cheap and popular FT2232–based devices. Because Xilinx ISE environment work with no problem on all platforms (as it does not use any hardware), even using emulation or virtualization, it is possible to create project and export its bitfile that can be later uploaded into target FPGA using xc3sprog that works on all platforms.

In order to use Xilinx JTAG interface to work with xc3sprog, a `fxload` utility is required first to upload apropriate firmware into interface. Both utilities are available in FreeBSD's port tree, interface firmware files are distributed with Xilinx ISE (in `common/bin/lin` directory). I have created simple helper script to find which firmware file is suitable for cable that we own. User should supply firmware filename as parameter, after successful flashing a red (target off) or green (target on) LED should light.

```
#!/bin/sh
# Xilinx USB Platform Cable firmware flashing helper script.
# (C) Tomasz Boleslaw CEDRO (http://www.tomek.cedro.info)
#
# This script helps finding proper firmware for your device.
# You need to have FXLOAD utility installed to use this script.
# It may be necessary to change vid/pid parameter and -t to match your device.
#
# After successful flasihng your cable should have LED turned on (green when
# cable is connected to a powered on target, otherwise red).

if [ ! $1 ]; then echo "Usage: ./script <firmware>"; exit 0; fi

fxload -v -D vid=0x03fd,pid=0x0007 -t fx2 -s xusbdfwu.hex -I $1
```

Because interface firmware is uploaded into volatile memory, it needs to be uploaded into interface each time it is reconnected. Flashing interface also changes its USB descriptors, so the same device can be seen with two different VID/PID pairs after and before flashing. This is why I have created another helper script `flashit.sh` that first uploads firmware into blank JTAG interface or reflashes already programmed interface, then uploads bitfile given as first script parameter directly into volatile FPGA Core Memory for faster testing. If script is given third parameter `spi` the bitfile is programmed into the SPI Flash memory so the bitfile remains after power–down, but this operation requires placing `bscan_spi.bit` bitfile that it dedicated to a specific FPGA device and act as a bridge between JTAG scan chain and the SPI Flash memory. Script automates most of the tasks and checks for required files to make developer work faster, some variables needs to be adjusted first to match the hardware being used.

```
#!/bin/sh
# Xilinx FPGA bitfile upload automation script using fxload and xc3sprog.
# (C) 20110827 Tomasz Boleslaw CEDRO (http://www.tomek.cedro.info)

CABLEFW=xusb_xup.hex
VID=0x03fd
PID=0x0007
ALTVID=0x03fd
ALTPID=0x0008
DELAY=10
```

```
FXLOAD=`whereis -bq fxload`
XC3SPROG=`whereis -bq xc3sprog`

if [ ! $FXLOAD ]; then echo "You need to install 'fxload' port first..."; exit 1; fi
if [ ! $XC3SPROG ]; then echo "You need to install 'xc3sprog' port first..."; exit 1; fi
if [ ! $1 ]; then echo "Usage: script <bitfile> [spi]"; exit 1; fi
if [ "$2" = "spi" ]; then
 if [ ! -f bscan_spi.bit ]; then
  echo "You need to provide bscan_spi.bit that will match your FPGA device!"
  echo "The bscan_spi.bit acts as bridge between JTAG chain and SPI.."
  echo "Visit xc3sprog website for more information: http://xc3sprog.sf.net"
  exit 1
 fi
fi

echo "Flashing Xilinx Cable Firmware (you may need to set it up first)..."
# First try the cable VID/PID with no firmware.
echo "Flashing blank cable..."
fxload -D vid=$VID,pid=$PID -t fx2 -s xusbdfwu.hex -I $CABLEFW
# Then if necessary try reprogramming existing xilinx cable firmware.
if [ $? -ne 0 ]; then
 echo "Trying to reflash existing xilinx cable firmware..."
 fxload -D vid=$ALTVID,pid=$ALTPID -t fx2 -s xusbdfwu.hex -I $CABLEFW
fi

if [ $? -eq 0 ]; then
 echo "Waiting for jtag interface firmware to settle..."
 sleep $DELAY
 if [ "$2" = "spi" ]; then
  echo "Uploading bitfile into SPI Flash (using provided bscan_spi.bit)..."
  xc3sprog -v -c xpc bscan_spi.bit
  xc3sprog -v -c xpc -I $1
 else
  xc3sprog -c xpc $1
 fi
else
 echo "Flashing Xilinx JTAG Cable failed, exiting..."
 exit 0
fi
```

The example working session is following:

```
%./flashit.sh ../pong_top.bit
Flashing Xilinx Cable Firmware (you may need to set it up first)...
Flashing blank cable...
no device foundNo such file or directory : vid=0x03fd,pid=0x0007
Trying to reflash existing xilinx cable firmware...
Waiting for jtag interface firmware to settle...
XC3SPROG (c) 2004-2010 xc3sprog project $Rev: 449 $ OS: FreeBSD
Free software: If you contribute nothing, expect nothing!
Feedback on success/failure/enhancement requests:
  http://sourceforge.net/mail/?group_id=170565
Check Sourceforge for updates:
  http://sourceforge.net/projects/xc3sprog/develop

firmware version = 0x0406 (1030)
CPLD version = 0x0012 (18)
DNA is 0xd14100d525220801
```

1.15 Schematics and PCB design with Eagle CAD

Eagle CAD [62] is an inexpensive (free for basic non commercial use [63]) and multiplatform commercial solution for electronic circuit design with great support from community

and the software vendor. Working on Windows, Linux (also FreeBSD) and Mac, having lots of components libraries and footprints, scriptable, being able to export to common indurstry standard formats, Eagle is flexible, versatile and easy to use solution for all electronics engineers that need to create schematics of their circuit and design a printed circuit board for manufacturing.

Both schematics and board layout editors are fully integrated, so implementing change in schematics also implements change in board layout. There is no need to produce, export and import component netlists anymore (that is list of components and their connections to other components). It is also impossible to implement a change that would break a design (i.e. deleting component in board layout without deleting it on schematics). Creating a design is therefore as simple as placing proper components on the schematics, creating their electrical relation to other components (using wires or bus/network names), then creating their physical image layout of the PCB.

Eagle helps in design with many features that help and speed–up the process (i.e. displaying unconnected nodes and signals), also prevents from making errors and at the end perform project verification, that is layout against schematics, mechanical constrains assumed by DRC (Design Rules Check) and electrical constrains assumed by ERC (Electrical Rules Check).

Although Eagle is relatively mature product, it is still under constant development, new features are being introduced with new releases. I have even proposed some new features that could improve the design process (i.e. selecting only specific type layout objects, changing parameters of design already converted to production matrix, etc). Additional features can be obtained with scripting functionalities. Eagle technical support is really fast, accurate and responsive (below 24h). This is why I consider this software and organization as professional – because it works well and efficient at low cost – which seems to be rare nowadays.

1.15.1 Creating new components and libraries

Eagle CAD use components libraries to draw elements on schematics (so called *symbols*) and printed circuit board (so called *footprints*). With vast basic library of components, and even larger user supplied free–of–charge components and scripts repository [64], it is possible to design every circuit.

Sometimes however it is necessary to create new components that are not yet present in the libraries (I have created lots of them and shared with other Eagle users throug public repository [64]). Eagle includes integrated component and library editor for this purpose. Components can be groupped in libraries (i.e. of one manufacturer or function type). Each component is represented by *Device* – a logical entity that has its own *Symbol* as visible on schematics and various *footprints* as visible on the PCB Layout (one integrated circuit can have multiple package variants). It is possible to copy existing elements from other libraries not to duplicate out work (i.e. for a common IC package). *Device* is ready after connecting *Symbol* pins with *Footprint* pads, so the relation is fulfilled between logical symbol and physical device.

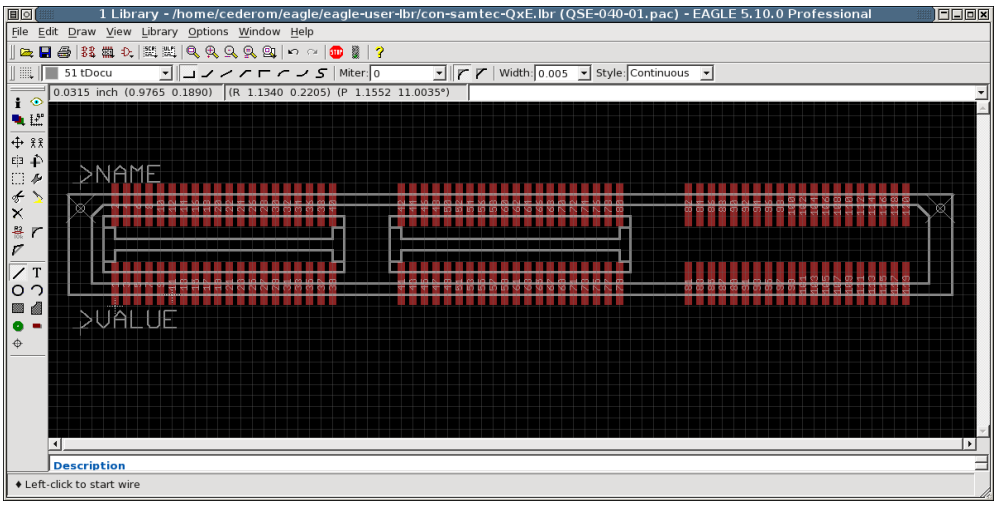


Figure 1.24: Creating new components for Eagle CAD with integrated components editor.

1.15.2 Exporting design for manufacturing

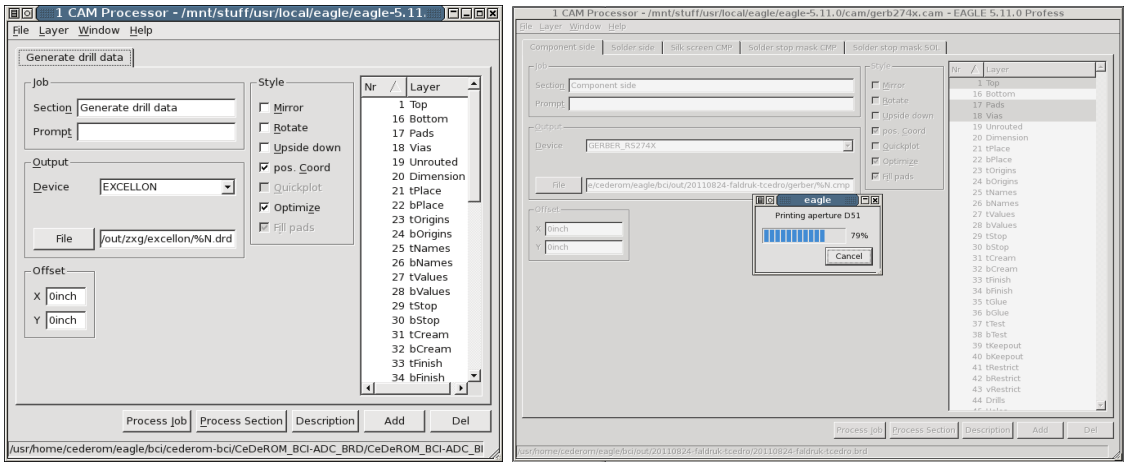
The most common starndard for PCB manufacturing is the *Excellon* format for drilling and *Gerber* for shapes definition (i.e. copper, stop/soldermask and legend/silkscreen layer images). Both of these formats are supported by Eagle and available for use in `Control Panel / CAM Jobs` as script `excellon.cam` and `gerb274x.cam`.

Generating project documnetation for manufacture process is very simple and fully automated with presented scripts – user just needs to run the script that will show nice graphical window full of tweaks and features to select, load the design to export, select output folder and start the process. Files are usually ready for use with default settings, no mirroring, etc. It is important to remeber to select output directory for all layers otherwise results will be stored together with the selected project (which is fine in most cases).

Each project also has its own `CAM Processor` utility to work on specific area of design (i.e. printed circuit board) and export this design into vast number of formats (i.e. EPS) which is very useful feature for general documentation, not necessairly manufacturing.

1.15.3 Running Linux Eagle CAD on FreeBSD

As mentioned before Eagle CAD is available for many different platforms including Linux. Because my system of a choice is FreeBSD which can run Linux binaries natively it was no problem to run Linux Eagle on my FreeBSD machine with `Linuxlator` kernel module enabled and standard Linux dynamic libraries installed. It is very interesing and unique functionality to run natively binaries from other operating systems, where FreeBSD again seems to be a pioneer. Eagle is properly compiled and find all necessary dynamic libraties on their default location (some Linux applications like Xilinx ISE has library location hardcoded that provokes some additional operations on user before application can run).



(a) Excellon generation (b) Gerber generation.

Figure 1.25: Generating Eagle CAD project documentation for manufacturing.

```
% ldd eagle
eagle:
  libXrender.so.1 => /usr/lib/libXrender.so.1 (0x28e8e000)
  libXrandr.so.2 => /usr/lib/libXrandr.so.2 (0x28e97000)
  libXcursor.so.1 => /usr/lib/libXcursor.so.1 (0x28e9e000)
  libfreetype.so.6 => /usr/lib/libfreetype.so.6 (0x28ea8000)
  libfontconfig.so.1 => /usr/lib/libfontconfig.so.1 (0x28f3b000)
  libXext.so.6 => /usr/lib/libXext.so.6 (0x28f6a000)
  libX11.so.6 => /usr/lib/libX11.so.6 (0x28f7b000)
  libdl.so.2 => /lib/libdl.so.2 (0x2907c000)
  libXi.so.6 => /usr/lib/libXi.so.6 (0x29081000)
  libpng12.so.0 => /usr/lib/libpng12.so.0 (0x2908a000)
  libpthread.so.0 => /lib/libpthread.so.0 (0x290b1000)
  librt.so.1 => /lib/librt.so.1 (0x290cb000)
  libjpeg.so.62 => /usr/lib/libjpeg.so.62 (0x290d6000)
  libstdc++.so.6 => /usr/lib/xilinx/libstdc++.so.6 (0x290f9000)
  libm.so.6 => /lib/libm.so.6 (0x291d9000)
  libgcc_s.so.1 => /lib/libgcc_s.so.1 (0x29202000)
  libc.so.6 => /lib/libc.so.6 (0x29210000)
  libz.so.1 => /lib/libz.so.1 (0x29388000)
  libXfixes.so.3 => /usr/lib/libXfixes.so.3 (0x2939d000)
  libexpat.so.1 => /lib/libexpat.so.1 (0x293a2000)
  libXau.so.6 => /usr/lib/libXau.so.6 (0x293c9000)
  libxcb-xlib.so.0 => /usr/lib/libxcb-xlib.so.0 (0x293cc000)
  libxcb.so.1 => /usr/lib/libxcb.so.1 (0x293ce000)
  /lib/ld-linux.so.2 (0x28e66000)
  libXdmcp.so.6 => /usr/lib/libXdmcp.so.6 (0x293eb000)
```

1.16 PCB Crafting

The Printed Circuit Board is the mechanical backbone of an electronic device covered with thin layer of metal (copper) to flow current between components. The topology of a PCB is designed nowadays in computer programs (such as Eagle CAD, Protel DXP, Pads, etc) and then transfered into physical board using chain of technological processes. This section describes methods that give best possible precision of PCB crafting at home

for single or small series of prototyping.

1.16.1 Photo–litography

Photoligography is a process which can be used to transfer PCB layout image onto the blank circuit board to create a matrix for etching the copper. Photoresist protects the copper against etching and it can be removed with UV light exposition to create necessary board layout image.

The image must be first printed on a thin translucent photocopy foil or tracing paper with high resolution printer such as laser printer. Color laser printer gives better black levels than black–while printers because black color in that case is made with three colour layers not one. The best printout however is obtained with DTP digital print that is not very expensive and offers unique printout quality both in contrast, black levels and resolution (even 3200dpi). For standard home made matrix DTP with true 1200dpi printout is sufficient even for precise BGA designs (that I have verified in practice).

When the printout is ready we need to have blank PCB covered with photoresist that protects the copper layer. Prefabricated PCB of this kind can be bought in electronics store, but also it can be created with standard blank PCB and *Photopositiv 20* spray emulsion available from *Kontakt Chemie* [93] and some practice/patience necessary to create thin layer of constant width and then make it solid by drying in max. 80 celcius degrees temperature for about 20 minutes.

When both PCB with photoresist and the matrix printout is ready, we need the UV light source strong enough to radiate the photoresist and change its chemical properties. I have verified some of the UV fluorescent light sources to be good enough for this process, but the Philips Halogen Dental UV lamp model 412447 75W/12V turned out to be the best solution.

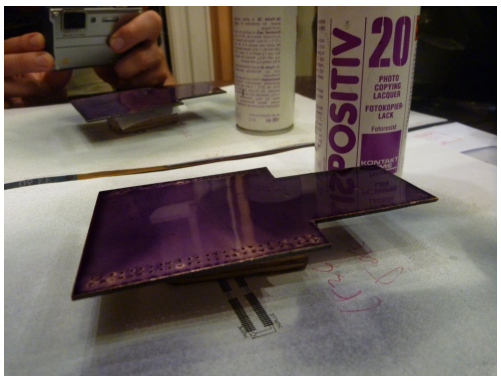
Photoresist changes its chemical properties when exposed to UV light. Normally it acts as shield for copper during the etching process, but when exposed to the UV radiation it melts within NaOH just like the photography. Creating the layout image on the PCB is as simple as covering the PCB behind the image and exposing it to the UV light source so the martix is transferred onto the photoresist layer. After NaOH induction PCB is ready for etching.

1.16.2 Copper Etching

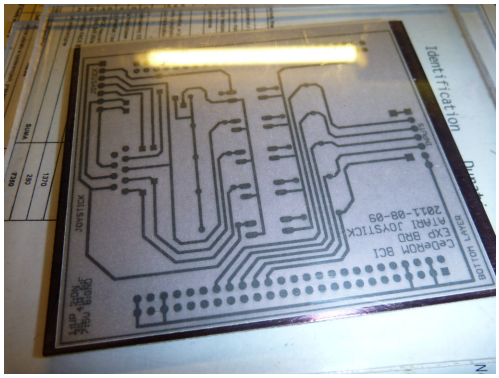
Etching process is necessary to remove copper from blank PCB in order to create isolation areas between tracks connecting together different components and pins of the components that transfer voltages/signals. Tracks are created by covering some parts of copper with some substance resistant for etching so the copper remains intact.

The most common method of etching is with Ferric Chloride. Although this is very hazard substance both for environment, human body and human clothes (it leaves stains that cannot be removed) it seems to have best lifetime measured in etching cycles and the precision of etching itself, because it can be „weaker" to etch slower but does not remove layers that should stay intact.

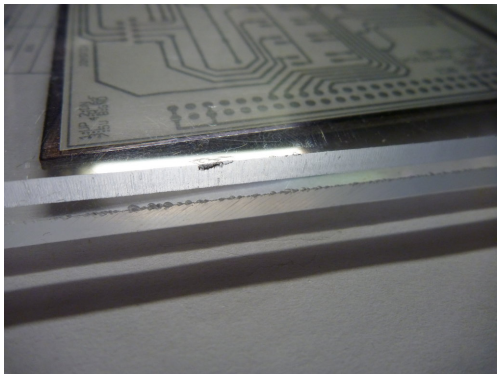
(a) UV Lamp DIY.



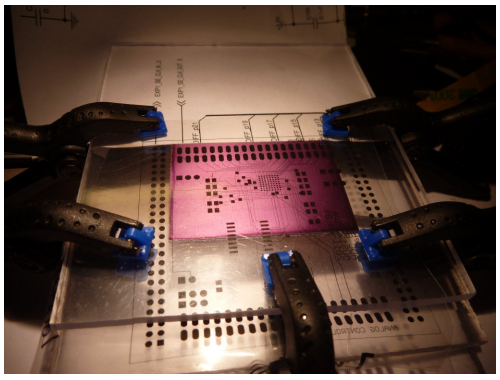
(b) Photoresist in spray.



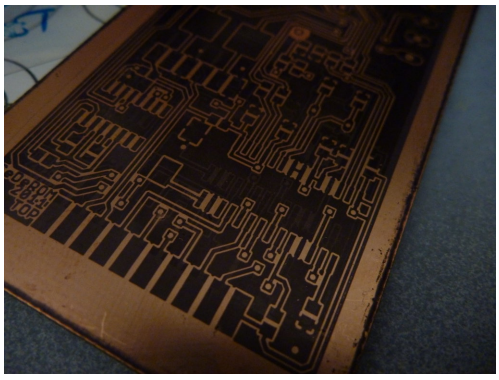
(c) Matrix transfer.



(d) Mechanical adjustment.

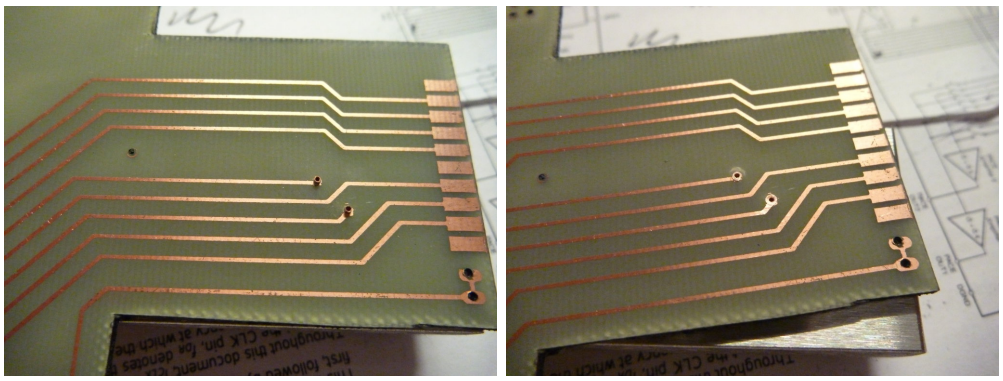


(e) UV radiation.



(f) Developed board.

Figure 1.26: Home made photo–litography.



(a) Rivets on place. (b) After riveting.

Figure 1.27: Home made mechanical metalization.

Another etching substance often used is the B327 ($Na_2S_2O_8$) and it perform its task faster, but sometimes it can over–etch the protected layers and it has far shorter lifetime than ferric chloride and can be used only to etch few PCB...

After etching the PCB should be cleaned out with Acetone from photo matrix, it can be also protected with resin mixed with alcohol that will prevent copper oxidation and helps soldering. Such resin resembles professional soldermask on a factory boards, but I don't know a method yet on how to perform this operation at home (this would help soldering the BGA devices).

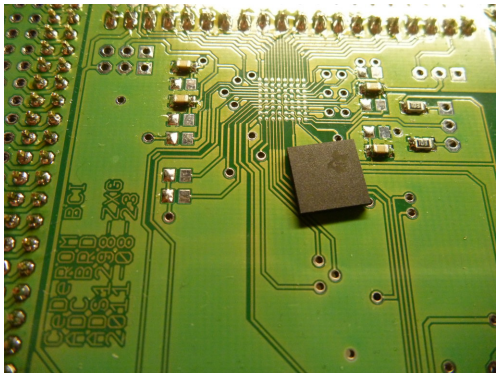
1.16.3 Drills Metalization

Printed circuit boards can be one, two, and multilayer, that means there are multiple layers of copper that can be etched to create a track matrix. Popular home made boards are only one and two layer because multilayer boards require specialized and expensive machinery and processing. Often one signal use two layers to route a track between components. In this case a drill must be made and both layers needs to be connected with metal. This can be done with a thin wire, but when the drill is used by a component pin that covers one of the pads it is impossible to solder both of the tracks.

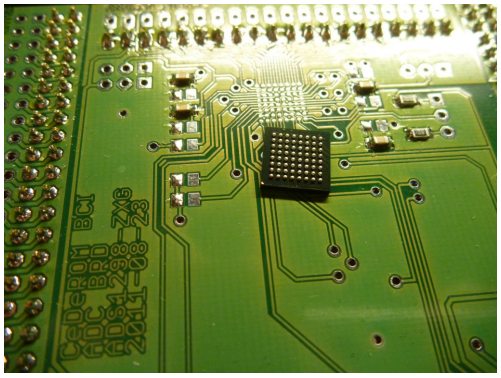
The metalization at home seems to be impossible because normally it requires complicated electrochemical processing. I have found a nice solution for mechanical metalization from LPKF GmbH [94] that use riveting with small copper parts. This is not the cheapest method but is allows metalization at home that allows later component placement in a through hole fashion.

1.16.4 BGA Soldering

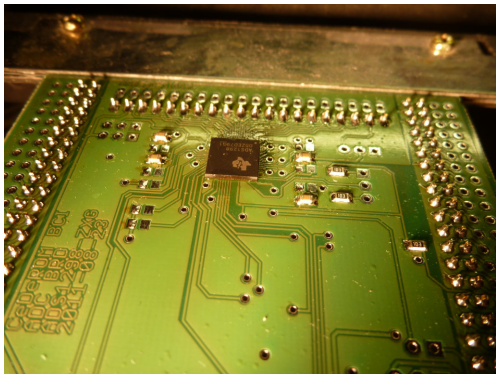
Standard through–hole or surface–mount components can be soldered with an ordinary soldering station and tin wire (for SMD I recommend tin of $\phi 0.25mm$), but BGA (Ball Grid Array) devices must be soldered with special hot air soldering station that forces



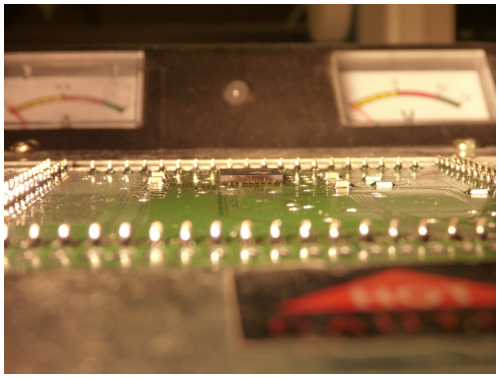
(a) Soldermask helps soldering.



(b) Tin balls are on the IC!



(c) Balls melted IC and cooper.



(d) Side view photo.

Figure 1.28: Soldering the BGA device.

hot air (around 400 celcius degrees) flow around the chip and the board making the tin balls to melt and connect copper wires with contact points below the Integrated Circuit casing.

Chapter 2

Solution Approach

2.1 Solution Approach

2.1.1 Introduction

There are few inexpensive commercial BCI solutions that showed up on the market recently, or their predecessors for Neurofeedback/Biofeedback being present for many years already. Free and Open–Source low cost projects also have some share in the field but in most cases they are incredibly outdated. Professional research project with billion dollar funding can be only seen in serious scientific publications or popular science TV programs. The truth about progress in the field of Neural Interfacing is not that impressive and we still cannot achieve its complexity and functionality as expected from the sci–fi novells [83] because we don't understand how brain works and we cannot reproduce its complexity with current technology, so it seems impossible that we successfully talk in this complex language of nature, yet.

On the other hand it is not really necessary to simulate whole brain functionality to perform simple operations such as cursor movement or activity detection in some outer parts of the brain. The technology for reading biological signals of brain activity in non–invasive manner is already available. Computational capabilities of modern computer systems are good enough to perform simple signal processing and classification that can be used for decission making to drive actions „by thoughts". It is probably the good algotirhm that will eventually solve the problem, or method of reading and stimulating brain activity in a closed loop feedback used for external object control. The algorithm however cannot exist without a hardware platform that makes it possible to operate and interact with real world data. Because commercial solutions are too expensive and free also cost some money but does not provide enough efficiency, I have decided to create such solution myself from scratch, both in software and hardware domain. This solution is relatively inexpensive, use commercial off–the–shelf components (COTS) and free/open–source software for licensing freedom, cost reduction and better insight into the black boxes being used.

The main goal of solution presented in this document is to provide hardware platform

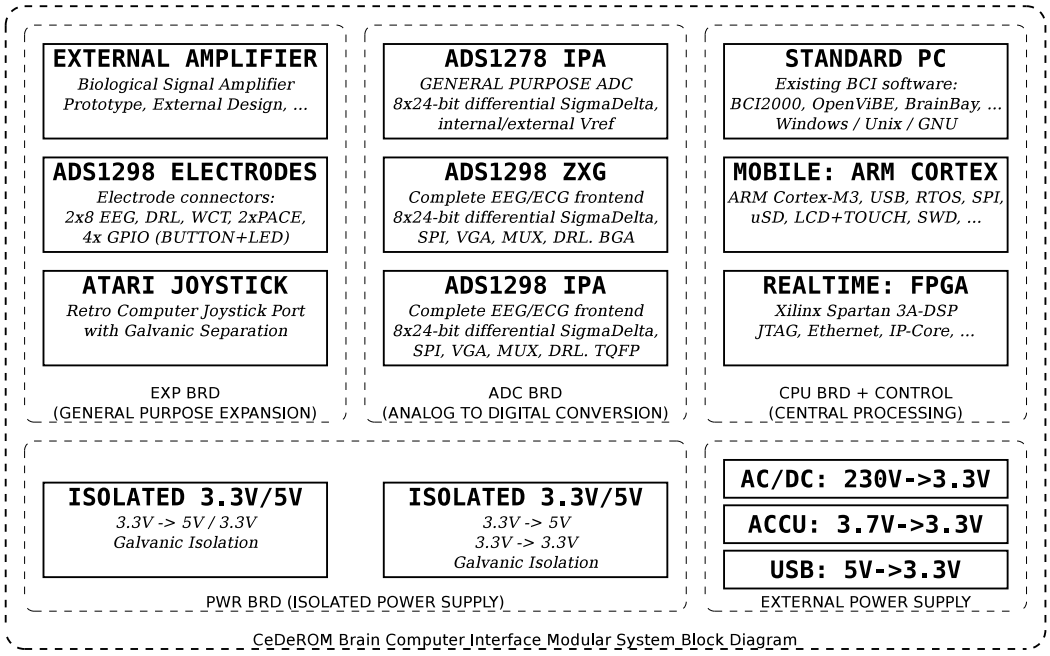


Figure 2.1: CeDeROM BCI Block Diagram.

for BCI research, that will be inexpensive but provide superior possibilities as compared to existing solutions – it must be better than gadget–like and open–source solutions, but also allow to perform tasks available only to expensive and closed–source solutions, even if this means creating solutions from scratch. Modular design is a very important factor because single component or whole configuration will be available for verification in short period of time with no need to redesign whole system which also means low research expenses. The software part is aimed to reuse existing open–source applications that are well known to the community, so the hardware will be just as easily applicable as all existing solutions before. Open–Source allows modification freedom, better testing and worldwide feddback from the community, no licensing restricions, better insight into details and many more advantages at cost of time necessary to create such solutions (often from scratch).

2.1.2 Similar solutions

The reference point for my research is a well known OpenEEG / Modular EEG Project [84] with open documentation and total cost of hardware less than $500. There are some modern commercial devices available for this price as well (i.e. Emotiv EPOC [89] or [88]) but these are restricted with licenses and does not provide detailed technical documentation of its comopnents. Commercial and professional research equipment's prices start at $10000 (i.e. from gTec [90]).

2.1.3 Block Diagram

Figure 2.1 presents block diagram of the *CeDeROM Brain Computer Interface* platform. It consists of many modules that can be connected together to perform as selected configuration. Modular design allows easy reconfiguration and practical verification of theoretical ideas or different system components. The system is divided into following functional areas:

- *Processing and Control* (`CPU_BRD`) that allows signal processing, decission making, results presentation and analysis, etc. At the moment those functionalities are provided by Personal Computer, Mobile Device, and Programmable Logic (FPGA). Each of the provided hardware platform has its advantages that offer unique possibilities of final solution application. Personal Computer is a very popular platform for design, development and simple experimentation. Mobile experiments with constant monitoring is important in holter–like applications, it can be also equipped with GSM/GPS module to allow realtime interaction no matter the location. FPGA solution is best for real–time high computational complexity applications that cannot be implemented on a PC, also the final verification of the integrated circuit can be performed here before manufacturing process, or any other standalone application that would have required a computer otherwise.

- *Analog–To–Digital Conversion* (`ADC_BRD`) provides bistream out of the analog signalls provided to the system. Different types and configurations of converters can be applied, tested and parametrized this way. High–resolution and multi–channel conversion requires powerfull enough control modules.

- *Analog Amplification* (`AMP_BRD`) provides analog front–end for the analog–to–digital modules. It is possible to verify different operational amplifiers, their parameters and configurations during the research of a new design, but also commercial amplifiers can be connected and used for comparison or reference.

- *Power Supply* (`PWR_BRD`) provides supply for both analog and digital circuits. In most cases also the galvanic separation can be implemented in this module, because of biomedical equipment safety requirements. It is important to remember that data lines also need the galvanic bareer when separation is applied on the power supply unit.

- *Adapters* (`ADP_BRD`) provide mechanical and electrical connection between modules and the control systems, so the modules can be connected to different hardware platforms.

- *Expansion Boards* (`EXP_BRD`) provide various expansions for the platform that can be used for user interaction (i.e. audio–visual signallization, push buttons), signal acquisitioni (i.e. electrodes), information exchange and interconnect with other systems(i.e. joystick).

2.1.4 Hardware Implementation

Figure 2.2 presents photos of assembled modules boards stacked on top of each other and then connected to control unit. Top board contains push–buttons, LED and electrode connectors. Electrodes lead signal to the analog board below or the analog–to–digital conversion unit with analog frontend provided internally. Below the ADC unit there is a power supply mounted directly on the adapter board. Bottom board is the control unit (FPGA or ARM based) that provides both power supply and input/output lines but these are not isolated as isolation must be applied on the target board level (i.e. power supply or the ADC board).

In the configuration presented on the images there is a joystick board (also isolated from target with optocouplers) stacked below the isolated power unit. This configuration should provide safe operations in biomedical sector, but also allow testing interoperation of different components. It is also possible however to connect ADC board directly to the CPU board and ommit the galvanic separation if necessary. All modules have jumper configuration to select between isolated and non–isolated supply.

2.1.5 Software Implementation

Open–Source was used for solution design, development, implementation and application, unless absolutely necessary and noted accordingly (i.e. inexpensive Eagle CAD, or free version of Xilinx ISE). Programs available for other free and open BCI solutions should work with design provided in this document with no additional configuration or environment change. I belive that this approach, although more time consuming, brings more benefit not only to this project itself but also for many other designers that can reuse tools created in this project and give additional features in return.

2.2 Modules Description

2.2.1 CPU_BRD: Xilinx Spartan–3A DSP FPGA

FPGA technology gives ability to create dedicated digital solutions that can contain whole computer system and real–time digital processing systems in one integrated circuit with rapid implementation and reconfiguration time. It is possible therefore to create and verify in practice new architectures that are not yer available in commercially available integrated circuits at cost reduced only to the FPGA device and the necessary extrnal components. Project synthesis for large scale production can be verified at functional level with FPGA and then processed to the topology verification and manufacturing. Small scale production designs, especially targeted at research, needs the FPGA to exist because they cannot be realised using other methods or it would take too much time to redesign physical device, while reconfiguration of FPGA takes only seconds. It is also possible to use free and open–source software components do tesing and compile design, just like in this project, so the cost is limited to minimum (only hardware), there are no constrains on

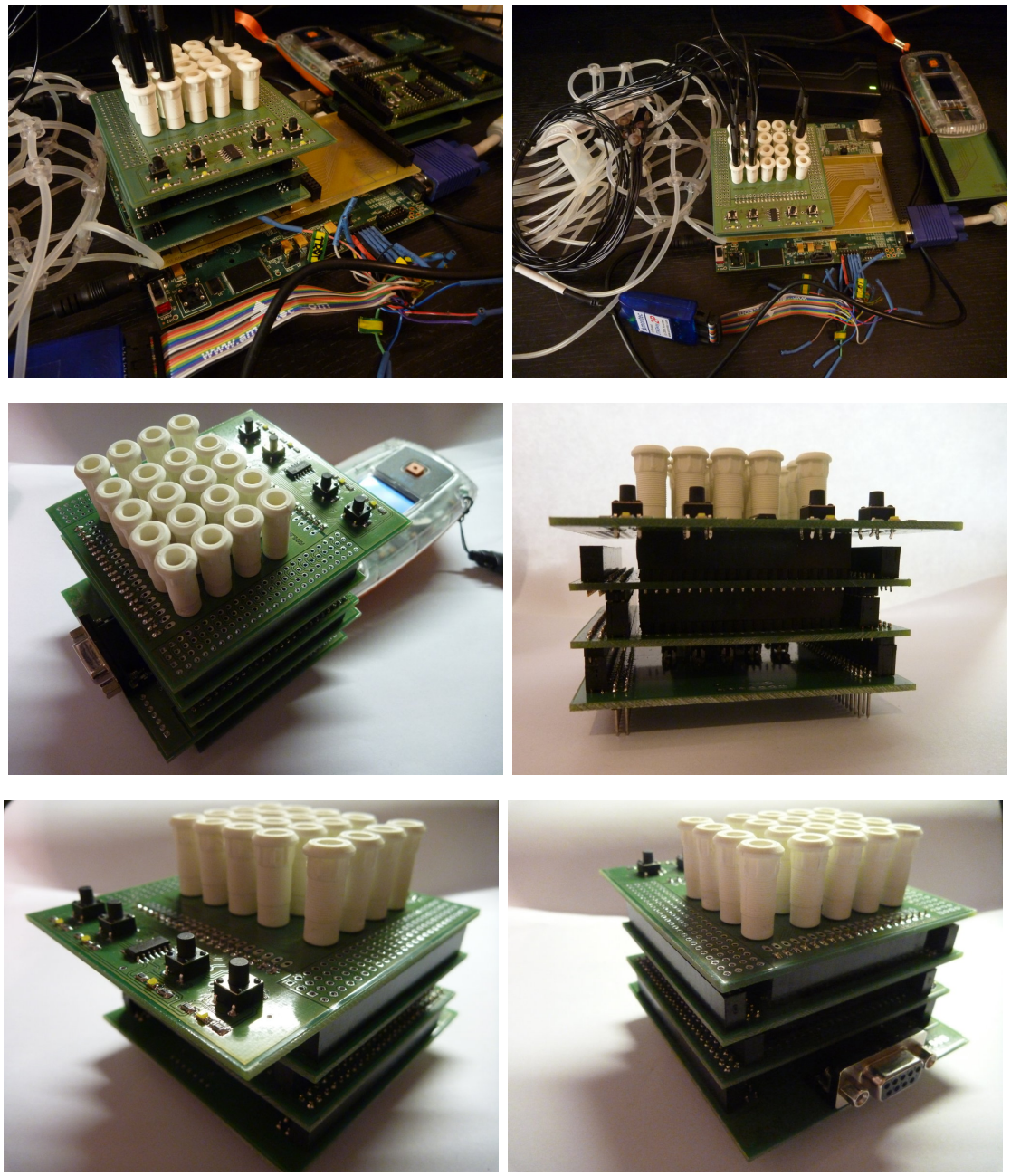


Figure 2.2: CeDeROM BCI Assembled Circuit Boards.

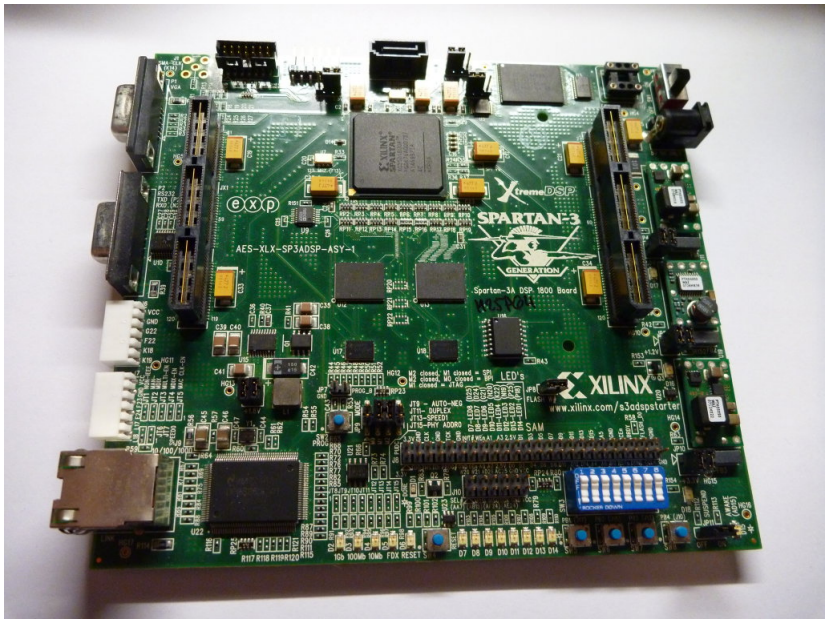


Figure 2.3: Xilinx Spartan 3A–DSP Development Board.

licensing, components can be freely customized, at the additional cost of time necessary to perform these tasks. FPGA is therefore the best platform for advanced research where big part of the solution logic can be reconfigured in seconds with no additional cost.

Figure 2.3 presents picture of the Xilinx Spartan 3A–DSP Development Board. This is the high–capacity device for DSP (Digital Signal Processing) applications that can contain both digital components such as CPU (i.e. ARM, AVR, PowerPC, etc) with operating system (i.e. FreeRTOS, FreeBSD, Linux, etc) and various signal processing circuits using dedicated adder/multiplier logical blocks present in this chip. Development Board is equipped with 10/100/1000Gbit Ethernet controller, JTAG Port, RS–232 Serial Port, low cost VGA output, 128MB DDR2 RAM, 16Mx8 BPI Flash, 64MBit SPI Flash, built–in voltage regulators, and many bipolar/unipolar input/output lines to general use. This equipment makes it perfect candidate for research and prototyping of various BCI applications that require realtime and/or computational power exceeding capabilities of an ordinary microprocessor devices but easily applicable in programable logical devices.

2.2.2 CPU_BRD: Stm32Primer2 (ARM Cortex–M3)

Stm32Primer2 [39] is a Development Kit based on ARM Cortex–M3 CPU with built–in 400mAh accumulator, color graphical LCD display with backlight, push buttons, micro SD card slot, USB 2.0 Device connector, RLink JTAG/SWD Debug Interface, audio codec with speaker and microphone. All those features make it perfec candidate for mobile applications, also the integration level of STM32 microcontroller allows external components count minimalization and therefore target application and device minimal dimensions. Figure 2.9 presents Stm32Primer2 device with adapter board installed for

use with CeDeROM BCI modules.

I have also chosen Stm32Primer2 because of its good support by the community gathered around its creator, Raisonance company [40], with dozens of example applications with source code and schematics available. The software components should be easy to integrate and combine, just as the hardware modules I have designed, resulting in rapid solution development.

Because RIDE development environment is commercial and closed source application with some licensing restrictions, there are ongoing efforts to bring SWD support for open–source applications such as UrJTAG [37] and OpenOCD [36] for target ARM–Cortex CPU programming and debugging. For this purpose I have created LibSWD [38], a first in the world Serial Wire Debug Open Framework, and introduced transport other than JTAG for these low–level embedded access applications. The work should be finished soon opening a door for new devices into open–source development world.

2.2.3 ADP_BRD: QSE to Goldpin Adapter

Xilinx Spartan 3A–DSP FPGA Development Kit use specialized `QSE-060` connectors from Samtec Inc. [54] that are relatively expensive (over 20 times more than average Goldpin connector) and hard to find on out local market, so I have decided to design adapter board that would reroute signals from QSE connectors into popular and cheap *Goldpin* sockets. This will also allow to stack more boards on top of each other and make modular design easier to implement. Similar adapter can be designed for other kind of hardware to allow use of existing *CeDeROM BCI* system modules.

Figure 2.4 presents schematics of the adapter, Figure 2.5 PCB layouts with component placement, Figure 2.6 presents photo of assembled board mounted on the target device. I have also created from scratch a library with Samtec connector for Eagle CAD [62] and shared it with user community using central components public repository [64] so it is possible to draw a schematic and design a PCB with this device that was not possible before.

There are two Goldpin connectors for each QSE connector and there are two QSE connectors on the FPGA Development Board, therefore it is possible to use two *CeDeROM BCI* base modules at time. Board was designed this way that one of the Goldpin connectors groups bipolar (differential) IO pins and clock sources, while second Goldpin groups unipolar IO pins and clock sources.

2.2.4 ADP_BRD: Stm32Primer2

This module is a straightforward adapter to mach mechanical dimensions and electrical signalling of CeDeROM BCI into Stm32Primer2 development kit equipped with ARM Cortex–M3 CPU (as described in section 2.2.2). At this point it provides only support for hardware SPI interface, some control and joystick signalls.

Because Stm32Primer2 is equipped with USB Device Controller it is possible to use it as a bridge between computer and external modules (such as creating USB HID device

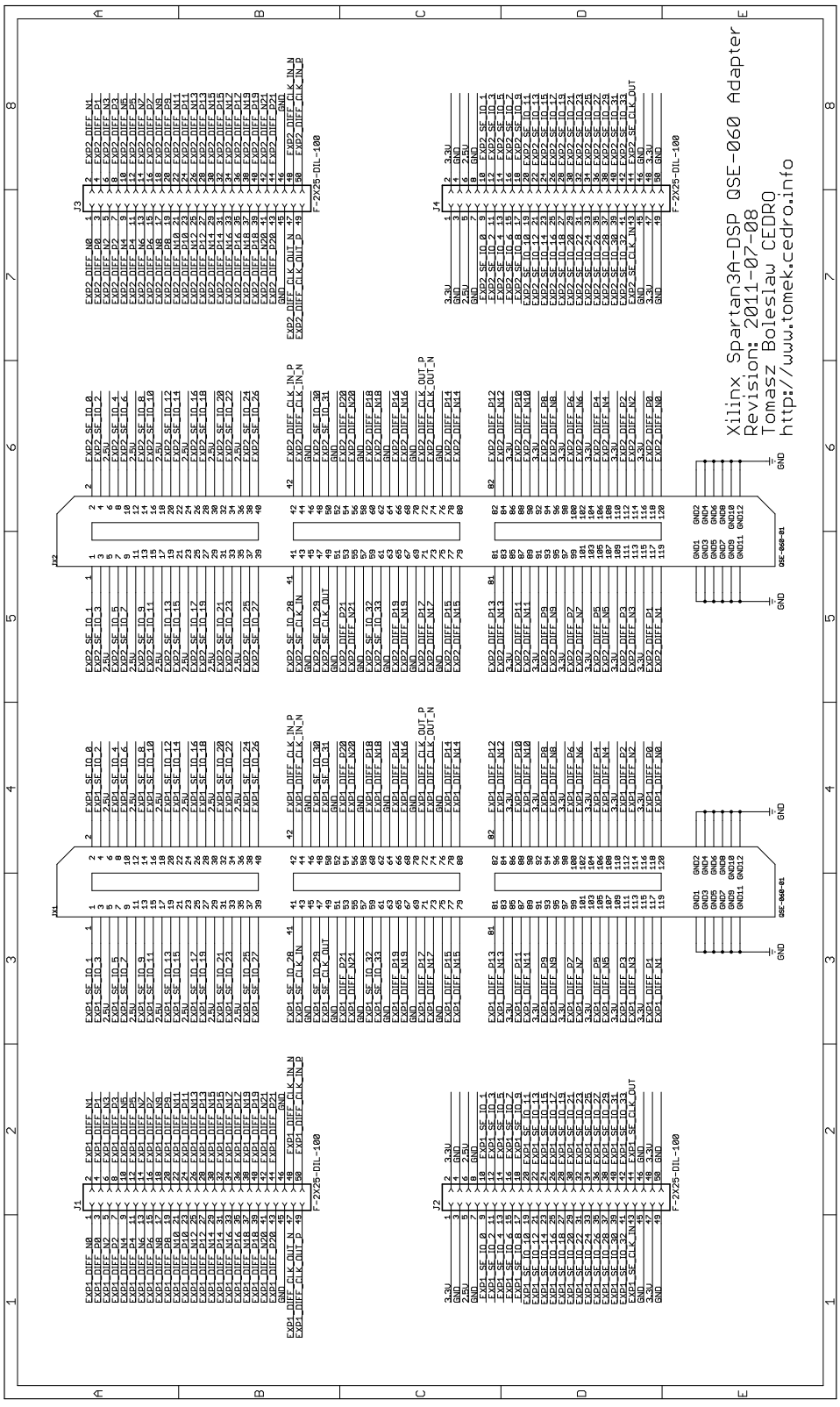
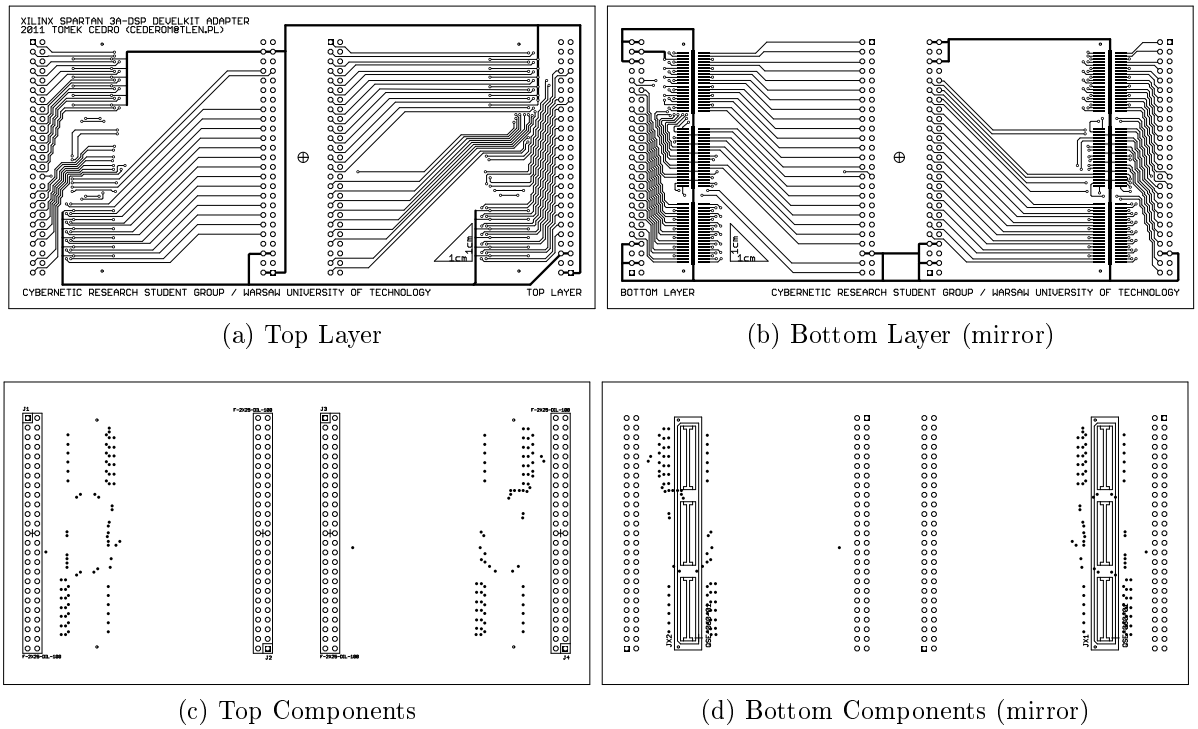


Figure 2.4: Xilinx Spartan 3A–DSP Development Board QSE–to–Goldpin adapter schematics.



(a) Top Layer

(b) Bottom Layer (mirror)

(c) Top Components

(d) Bottom Components (mirror)

Figure 2.5: CeDeROM BCI Xilinx Spartan 3A–DSP Development Board QSE–to–Goldpin adapter PCB design.

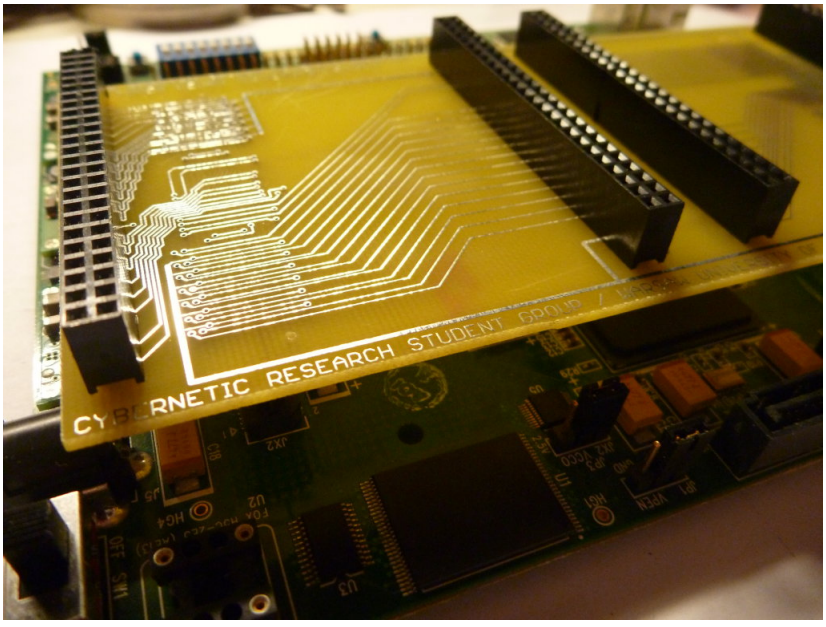


Figure 2.6: Assembled QSE–to–Goldpin CeDeROM BCI ADP_BRD for Xilinx Spartan 3A–DSP Development Board.

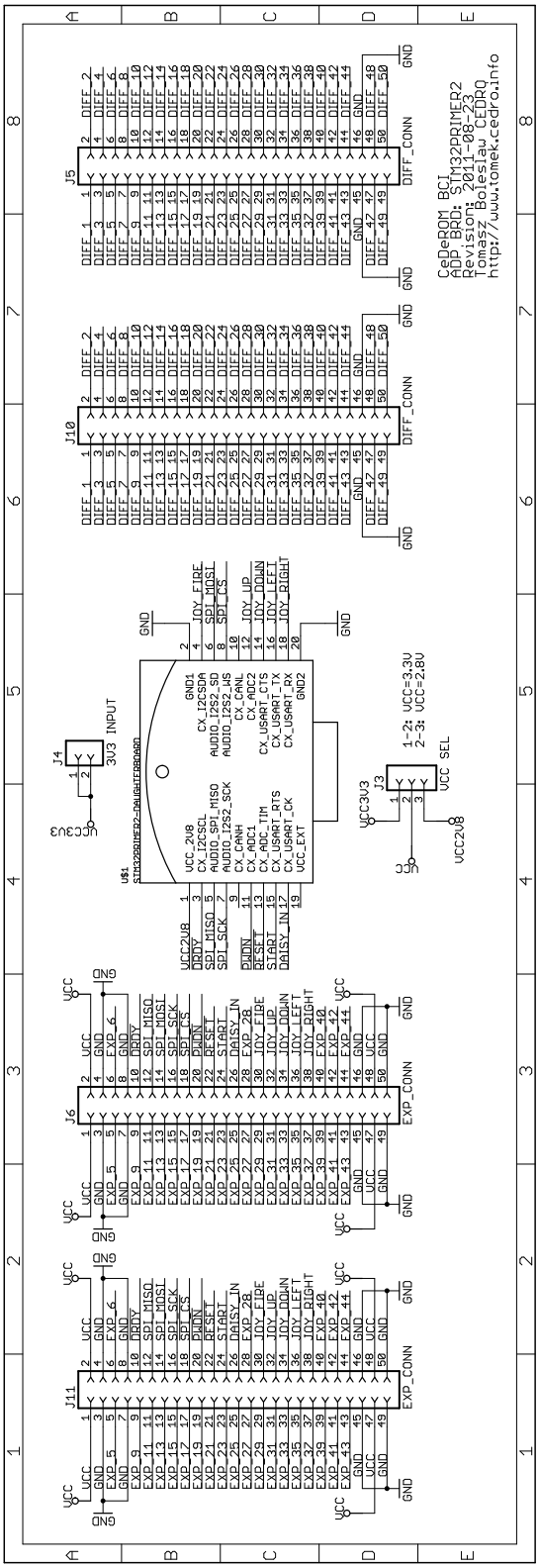
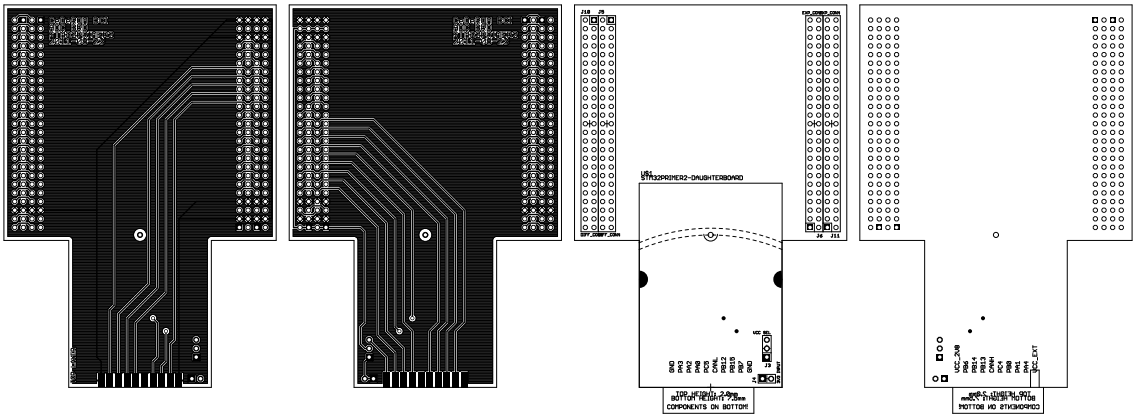


Figure 2.7: CeDeROM BCI Stm32Primer2 Adapter Board.



(a) Top Layer (b) Bottom Layer (mirror) (c) Top Components (d) Bottom Components (mirror)

Figure 2.8: CeDeROM BCI Stm32Primer2 Adapter Board PCB design.

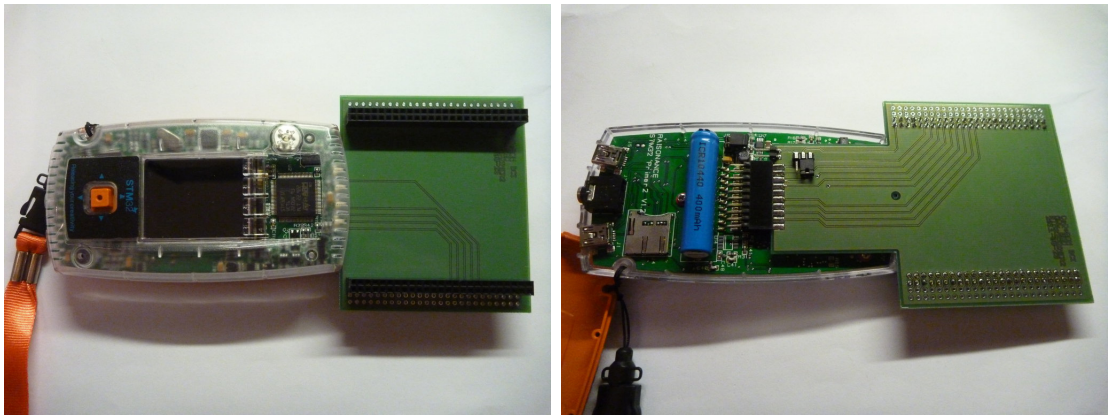


Figure 2.9: CeDeROM BCI Stm32Primer2 Adapter Board, assembled.

with physical joystick capabilities). Also the standard connector contains only 2.8V power supply signal, that is too low for DC–DC converters implemented in `Isolated PWR_BRD` (section 2.2.5), so small modification is necessary to connect 3.3V supply from LCD Display voltage regulator onboard Stm32Primer2. Presented module has jumper setting option if the power supply for other CeDeROM BCI modules should have 3.3V to match standard needs, or 2.8V that will fit other modules without DC–DC converters demanding 3.3V for proper operation.

Figure 2.7 presents schematics of the module, while Figure 2.8 presents PCB layout and component placement. Figure 2.9 presents photography of assembled circuit boards.

I have also created from scratch a library with Stm32Primer2 DaughterBoard connector, board dimensions and schematic symbol for Eagle CAD [62] and shared it with user community using central components public repository [64] so it is possible to draw a schematic and design a PCB with this device that was not possible before.

2.2.5 PWR_BRD: Isolated 3.3V/5V

Biomedical equipment requires galvanic separation between patient body and the equipment connected to a power network, so in case of malfunction human body remains safe. This is why `Power Board Modules` are necessary – to provide safe power supply to modules directly connected to a human body.

Standard digital connectors of the system provides non–isolated 3.3V from the `CPU_BRD`, but there are also separate power supply pins on the analog connectors (`DVDD`, `DGND` for digital circuity and `AVDD`, `AGND`, `AVSS` for bipolar analog circuity) that can provide „safe" isolated voltage. The `PWR_BRD` function is to provide safe power supply on the analog pins if necessary. This safe voltage is converted from digital lines and passed on analog lines using isolated DC–DC converters, but user can also choose with proper jumpers if he/she wants to use safe of unsafe voltage as required by experiments they are going to conduct, or simply to compare power quiality (i.e. noise, stability, etc).

Figure 2.10 presents module schematics, Figure 2.11 presents PCB layout and components placement. Figure 2.12 presents photography of assembled circuit board.

I have also created from scratch a library with XP Power/Murata DC–DC converter modules for Eagle CAD [62] and shared it with user community using central components public repository [64] so it is possible to draw a schematic and design a PCB with this device that was not possible before.

Two boards were designed for *XP_Power* [69] and *Murata* [70] modules which are pin–to–pin compatible components so the board layouts are identical. Boards contain two DC–DC converters that produce 3.3V and 5V out of the 3.3V. The galvanic isolation is quaranteed for 1kV but 3kV version components are also available. Additionally, boards can work only with 5V converter as 3.3V LDO Voltage Regulator is included in the design. Also the simple resistor voltage divider that creates virtual ground on half supply voltage potential (in this case around 2.5V) is included to be buffered by analog board and generate symmetrical power supply for bipolar design. This approach should be safe enough for integrated mixed signal solutions (such as ADS1298 or ADS1278 implemented in `ADC_BRD` modules) because not all components allow analog and digital ground potential to differ more than 0.3V.

2.2.6 ADC_BRD: ADS1298

Analog–to–Digital Conversion Boards (`ADS1298 ADC_BRD`) are based on a ADS1298 [77] integrated circuit from BurrBrown / Texas Instruments [76] is a full frontend for biological signal measurement with variable–gain input amplifiers (VGA), advanced multiplexer (MUX), DRL negative feedback reference potential generator, electrode lead–off detection and 8 independent differential 24–bit Sigma–Delta analog–to–digital converters with SPI (Serial Peripheral Interface) for interfacing with control circuit (`CPU_BRD` in out case). There is even a special ADS1298R (`R` suffix) version of the device that allows respiratory measurement based on body impedance variation. Device contains internal oscillator and voltage references, so it is a perfect candidate for single chip mobile applications where no advanced analog circuity is necessary. It is possible to use external analog circuit with

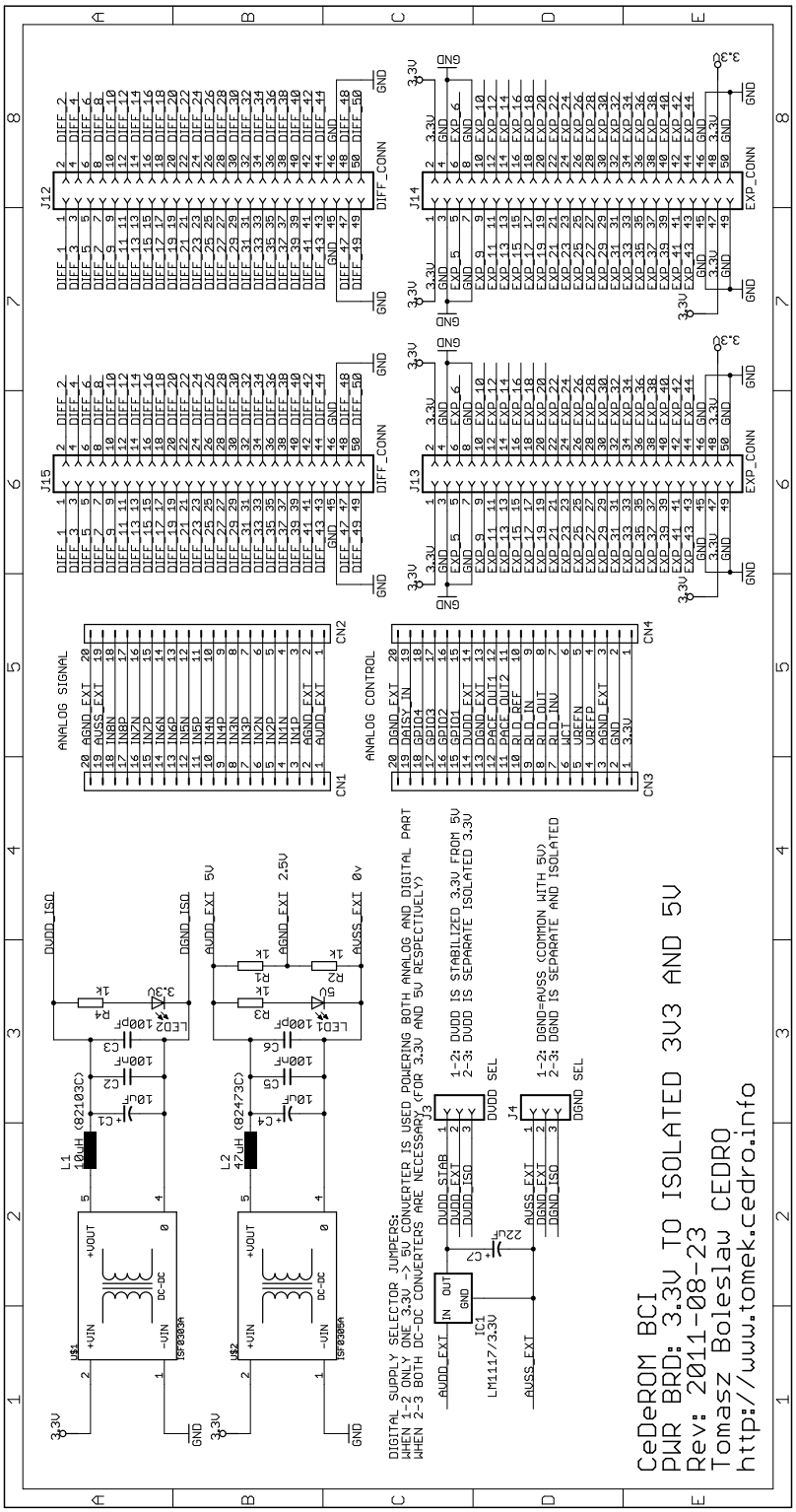
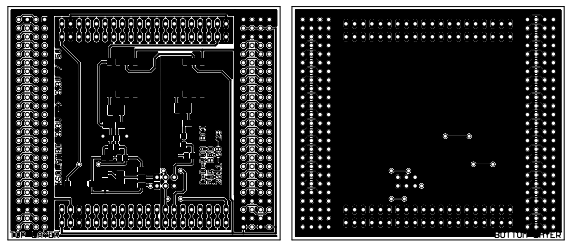
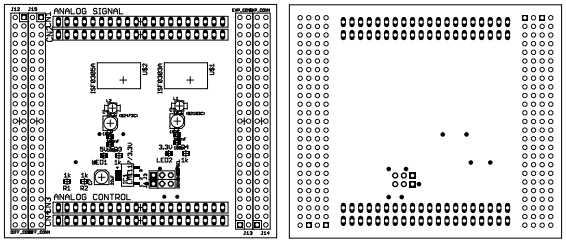


Figure 2.10: CeDeROM BCI Power Board: 3.3V to isolated 3.3V and 5V Converter.



(a) Top Layer (b) Bottom Layer (mirror)



(c) Top Components (d) Bottom Components (mirror)

Figure 2.11: CeDeROM BCI Power Board PCB design.

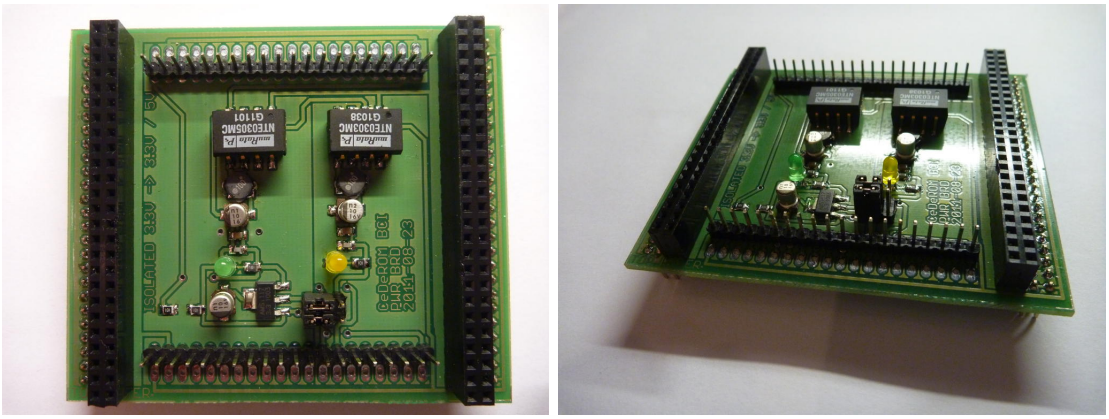


Figure 2.12: CeDeROM BCI Power Board: 3.3V to isolated 3.3V and 5V Converter, assembled.

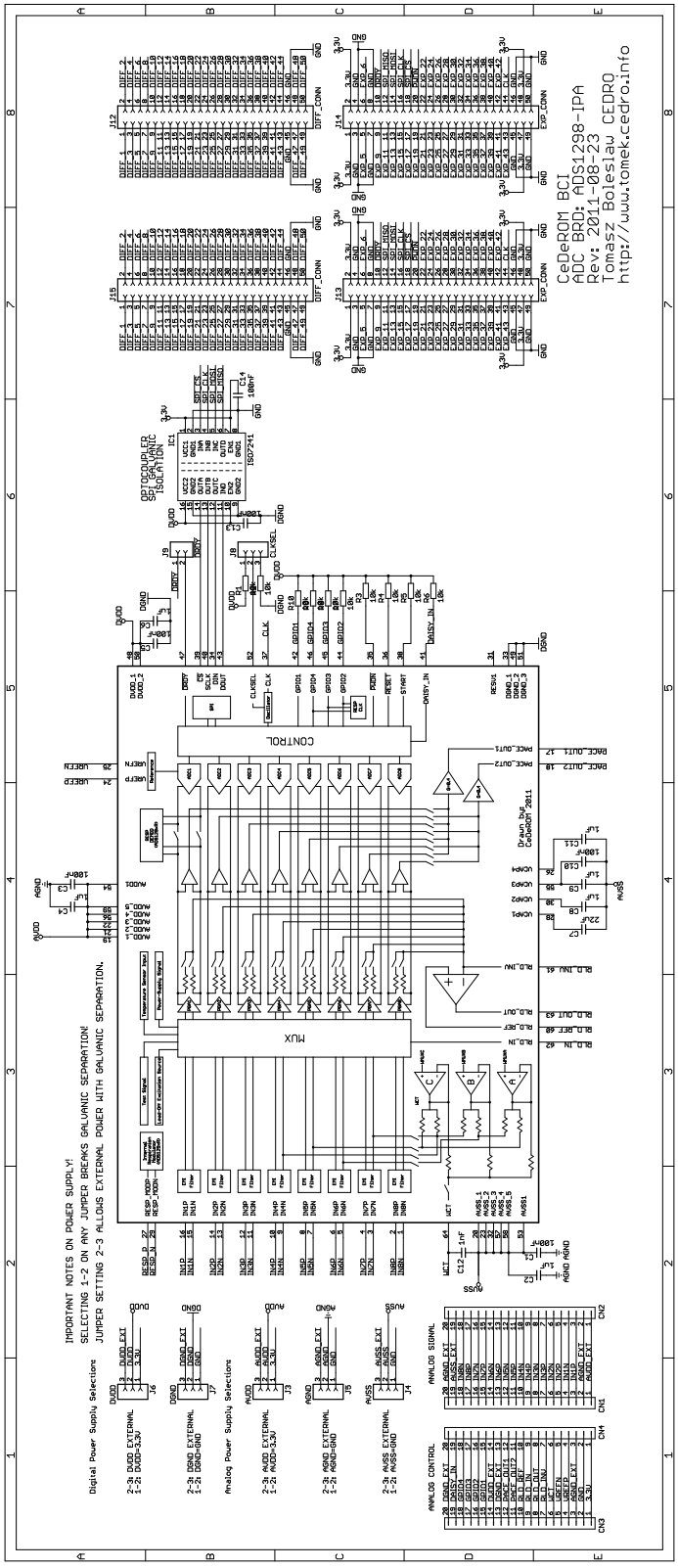


Figure 2.13: CeDeROM BCI Analog–To–Digital Conversion Board: ADS1298-IPA (TQFP footprint).

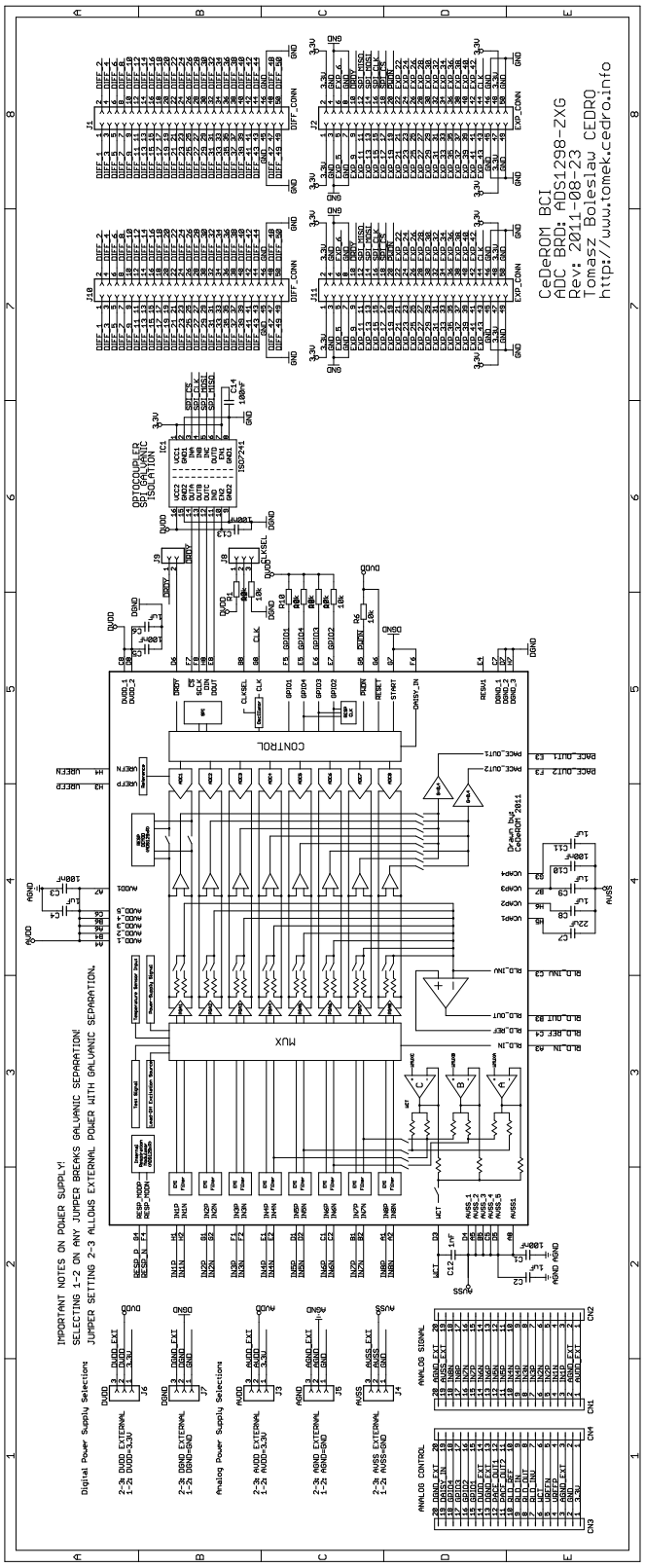
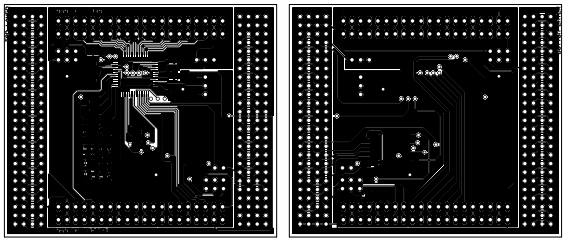
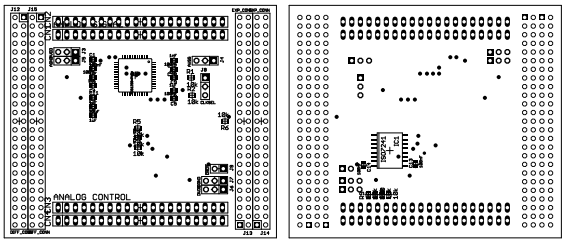


Figure 2.14: CeDeROM BCI Analog–To–Digital Conversion Board: ADS1298-IPA (BGA footprint).



(a) Top Layer (b) Bottom Layer (mirror)



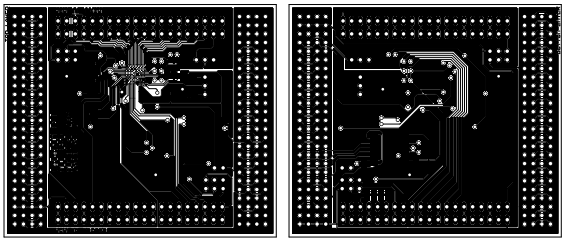
(c) Top Components (d) Bottom Components (mirror)

Figure 2.15: CeDeROM BCI Analog–To–Digital Conversion Board based on TQFP ADS1209–IPA, PCB design.

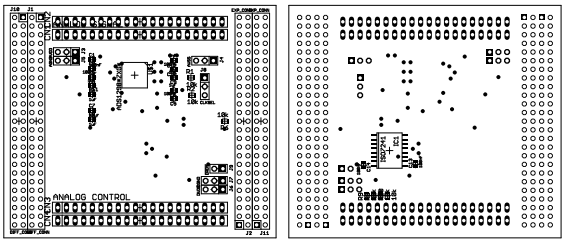
additional filters and amplifiers but this will decrease some features performance such as electrode lead-off detection. This one IC contains all features that I wanted once to design myself, so even without sophisticated analog frontend it should suffice for many experimental and production applications. It is ofcourse possible to attach additional `AMP_BRD` analog module or simply use `ADS1298 Electrodes EXP_BRD` (section 2.2.7) to connect standard EEG electrodes.

Figure 2.13 presents schematics of ADS1298–IPA based design with TQFP [78] IC footprint, while Figure 2.14 is based on ADS1298–ZXG version of the IC using BGA [79] footprint. Figure 2.17 presents photo of assembled circuit boards.

Both designs differ slightly as BGA version use lower count of additional control pins – only SPI and PowerDown are available because of small package and limited technology of PCB manufacturing as well as IPA/TQFP device stock inaccesibility – this is why also the PCB layout and component placement is also different (Figure 2.19 and 2.16). Both versions can be powered from „safe" `Isolated PWR_BRD` but also directly from digital lines – the configuration is selectable physically using jumpers. SPI interface is protected with optocouplers, while less important lines (i.e. `!DRDY` and `!PWDN` that are not essential for `SPI` operations) are connected directly to the digital IO lines and should be cutt–off after initial recognition of ADS1298 behavior is already known to developers, they are simply unnecessary or the galvanic separation is required (remember that CeDeROM BCI is targeted for early stage research and prototyping where such features might be necessary). Please keep in mind that connecting even one line to the other parts of device powered by mains will break galvanic separation and make whole device potentially unsafe



(a) Top Layer (b) Bottom Layer (mirror)



(c) Top Components (d) Bottom Components (mirror)

Figure 2.16: CeDeROM BCI Analog–To–Digital Conversion Board based on BGA ADS1209–ZXG, PCB design.

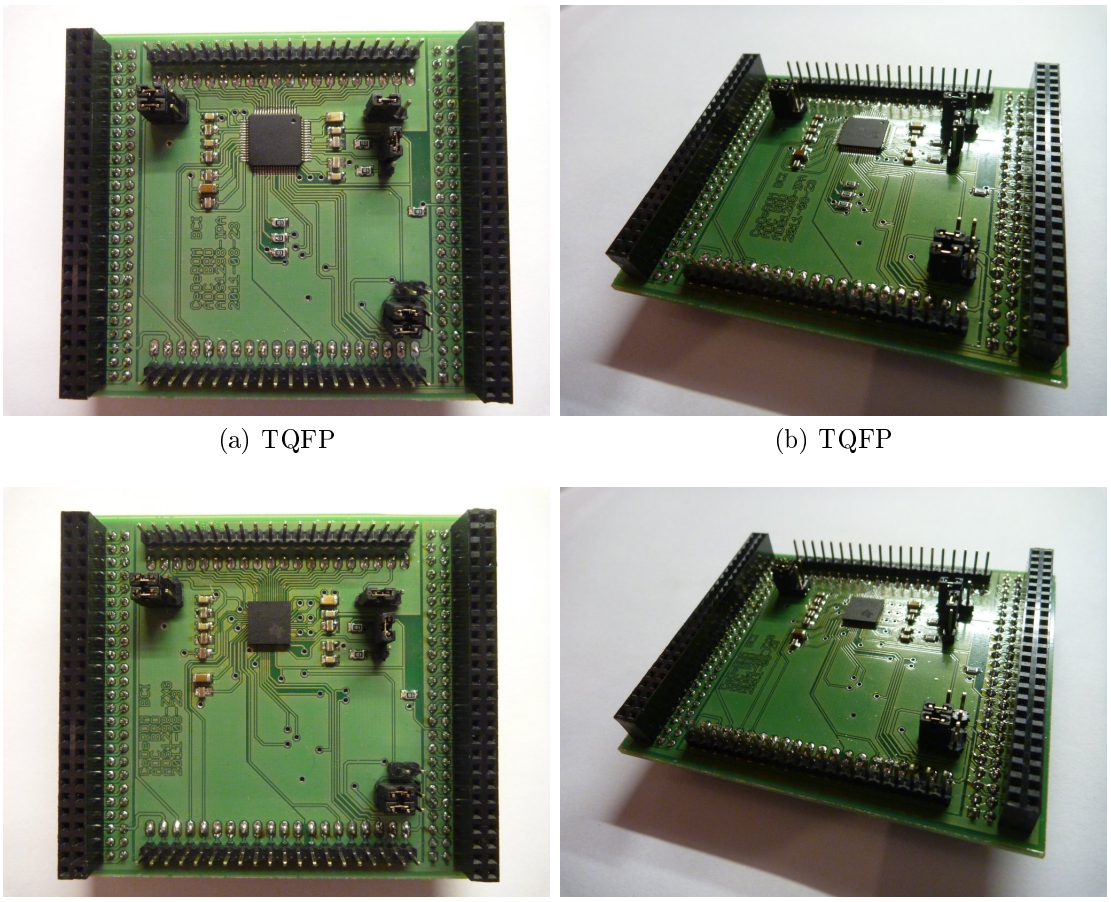
for biomedical applications!

I have also created from scratch advanced library component of ADS1298(IPA/ZXG) with footprint for Eagle CAD [62] and shared it with user community using central components public repository [64] so it is possible to draw a schematic and design a PCB with this device that was not possible before. The schematic symbol presents abilities and internal structure of ADS1298 in a very clear way.

2.2.7 EXP_BRD: ADS1298 Electrodes

This module is dedicated to `ADS1298 ADC_BRD` (section 2.2.6) module providing sockets for analog EEG inputs and also input/output section for user interaction based on ADS1298's four GPIO lines. All of four GPIO lines are buffered and can act as inputs and/or outputs – by default all lines are pulled high (as they cannot float). When line is set as input pressing button will force low state, when line is set as output low state will activate the buffered LED, so all lines are „active–low" both for input and output, otherwise pulled high by a resitor. It is possible to mount buzzer instead of LED to generate an audio tone. Buttons can be used for testing user interaction such as reflex or stimuli response. LEDs and buzzers can be used to generate stimuli, or clean time marker (i.e. audio–video binary pulse sequence) on video recording for better synchronization and easier result analysis.

Figure 2.18 presents schematics of ADS1298 EEG Electrodes Expansion Board, while Figure 2.19 presents the PCB layout and component placement. Photography of assembled circuit board is presented on Figure 2.20.



(a) TQFP (b) TQFP

(c) BGA (d) BGA

Figure 2.17: CeDeROM BCI Analog–To–Digital Conversion Board based on BGA ADS1209–ZXG, PCB design.

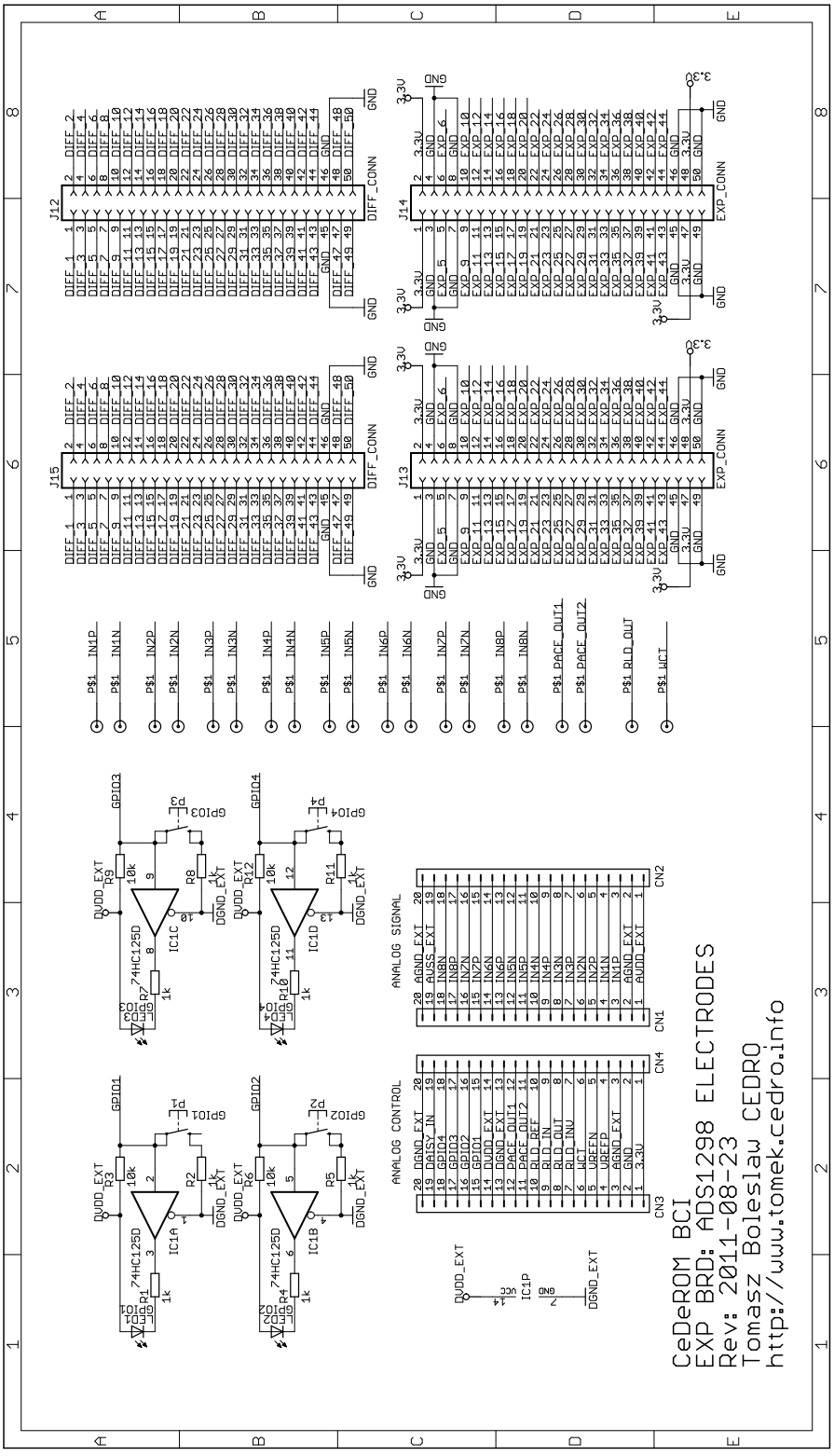
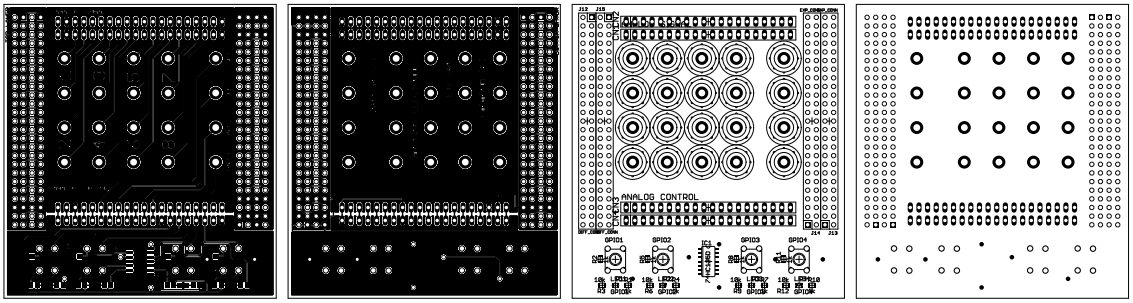


Figure 2.18: CeDeROM BCI EEG Electrodes Board for ADS1298 Schematics.



(a) Top Layer (b) Bottom Layer (mirror) (c) Top Components (d) Bottom Components (mirror)

Figure 2.19: CeDeROM BCI ADS1298 EEG Electrodes Board PCB design.

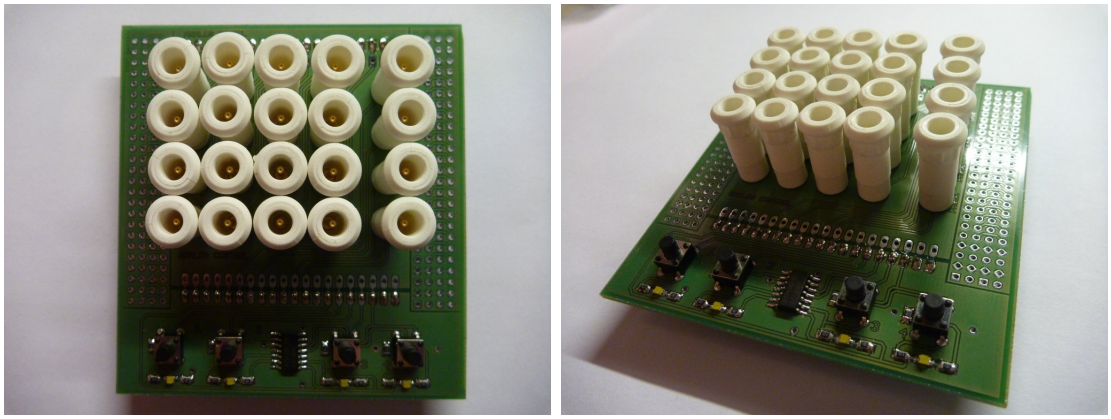


Figure 2.20: CeDeROM BCI ADS1298 EEG Electrodes Board, assembled.

I have also created from scratch a library of Multi–Contact [55] EEG connectors for Eagle CAD [62] and shared it with user community using central components public repository [64] so it is possible to draw a schematic and design a PCB with this device that was not possible before. Connectors are mounted orthogonally to the board surface through–hole style. Electrodes are not shielded, so there are two electrodes per channel for differential measurement. Additional output electrodes are groupped together for DRL, WCT and PACE functions if necessary. Board design assumes obligatory microswitch mount as they also create connections with pair of their pins, so if it is necessary to mount external button it should be soldered to existing button, or new bridged connection should be made instead. Board layout is very simple and it can be implemented even with one layer PCB.

2.2.8 ADC_BRD: ADS1278

Although ADS1298–based module (chapter 2.2.6) is very flexible biological signal acquisition device it might be necessary to measure some other kind of signals or use totaly

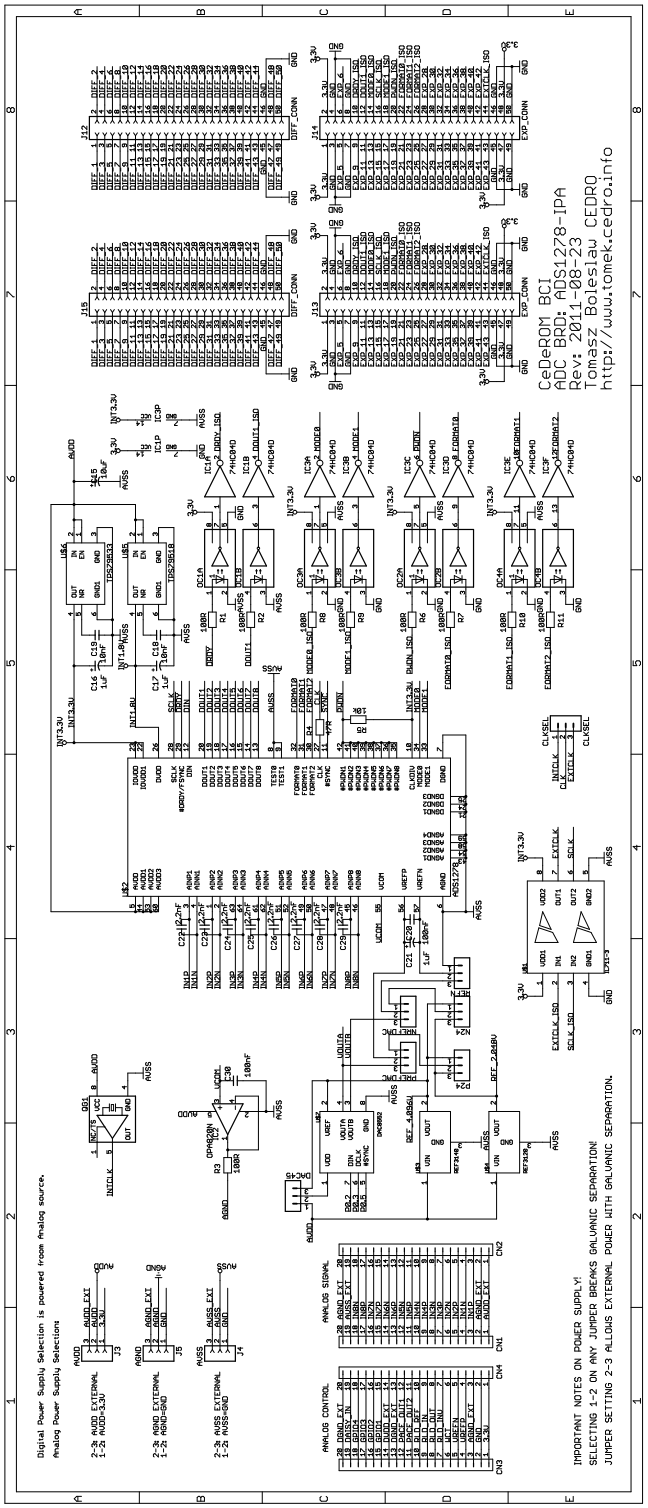
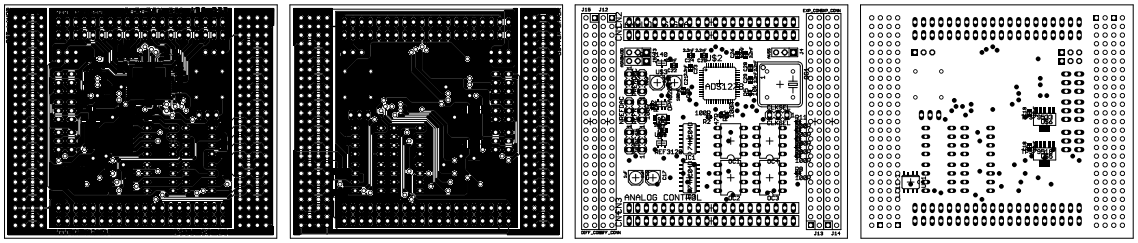


Figure 2.21: CeDeROM BCI General Purpose Analog–To–Digital Conversion Board: ADS1278-IPA (TQFP footprint).



(a) Top Layer (b) Bottom Layer (mirror) (c) Top Components (d) Bottom Components (mirror)

Figure 2.22: CeDeROM BCI General Purpose Analog–To–Digital Conversion Board based on TQFP ADS1278–IPA, PCB design.

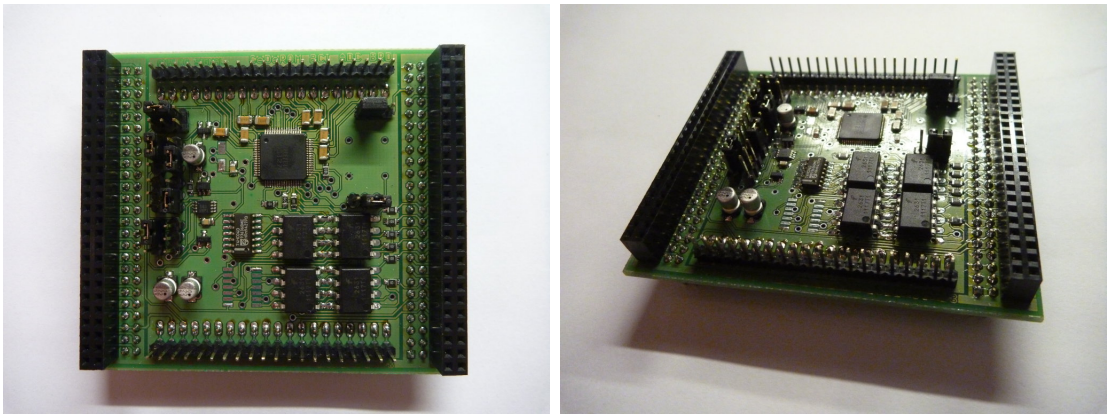


Figure 2.23: CeDeROM BCI General Purpose Analog–To–Digital Conversion Board based on TQFP ADS1278–IPA, assembled boards photos.

different analog front–end that ADS198 could not fit. This is why separate module, a `ADC_BRD: ADS1278`, was created using ADS1278 [81] integrated circuit from BurrBrown / Texas Instruments [76].

ADS1278 does not use SPI for communications (but can act as SPI–like transmitter–only) so it requires more control and signal lines, which all of them are also galvanically separated with optocouplers. Power supply needs to be supplied from `PWR_BRD: Isolated` because module has its own ultra–low–noise precission 3.3V and 1.8V voltage regulators. There are three built–in reference voltage sources for ADC – two precission refernce voltage source providing 2.048V / 4.096V and the double 16–bit Ultra–Low–Glitch DAC (Digital–To–Analog Converter) based on DAC8552 [82] device that can provide any given voltage from available range if properly programmed. Converter use inputs of analog connector, so it can be connected to another `AMP_BRD` with dedicated analog front–end of required design.

Schematics of the module is presented on Figure 2.21 and the PCB layout and components placement is presented on Figure 2.22. Figure 2.23 presents a photo of assembled circuit board.

This board has fairly complex layout because of number of applied optocouplers on data and control lines. This is also a test board for separating elements, their characteristics and the controller input/output port performance, because ADS1278 in current configuration produce serial bitstream for all eight 24–bit DACs at every conversion cycle with no buffering, which turned out to be problematic for simple microcontroller port to serve (even one channel per conversion) making it perfect candidate to perform stress–testing. The use of FPGA seems necessary in this case as previous experiments with ARM7 device (LPC2148) showed it did not have enough computing power and input/output port efficiency to serve all 24–bit and 8 channels transmission from ADS1278 device.

.

2.2.9 EXP BRD: Atari Joystick

It is important to research and demonstrate possibilities of interaction with external multimedia and entertainment devices such as gaming platforms as they create market that is still expanding into new areas of everyday life. It is also important to have fun of designing and share that joy with others, just as Atari Incorporated [91] and many other companies did many years ago when we, as young kids, discovered computer world and videogames long before we could understand how they work.

8–bit Atari Personal Computer System was my first computer ever (well after PONG videogame also invented by Atari, cloned by polish company ELWRO). It was a gift from my amazing parents back then around 1989 (I was 7 years old) when data was stored on the magnetic cassete tapes noone even knew what it could be used for because computers were known only from movies for an ordinary people. Months of learning BASIC, first programs, virtual encounters just like in the legendary TRON movie, no other life problems, fresh and warm meals full of proteins, all this made my computer and geneally childhood memories often coming back to those good days. Although there are many modern computer systems and videogame consoles, I have decided to create dedicated demonstration platform on 8–bit Atari machine running incredible game `Yoomp` [92] written in 2008 by polish group of enthusiasts that shared with me a cartridge containing the game (thank you guys!). I will be probably the first person ever to create Brain Computer Interface device for 8–bit Atari. The good news for other Retro Computer lovers is that joystick port was a standard so they might use my design on their machines as well bringing memories back to life again, maybe even showing their kids how computers looked twenty years ago.

The joystick port is a standard `DB-9M` port, but the plug should be longer and narrow to fit into casing. Old computers used 5V logic, so the joystick port had one reference voltage output, one ground output, and five inputs for each direction (up, down, left, right, fire). There were also two analog inputs for X and Y axis, but standard joysticks used simple on/off scheme to send signals. Joystick therefore is a simple module that separates galvanicaly our BCI system from the target computer with optocouplers producing on/off digital signals and making use of biomedical equpiment safe.

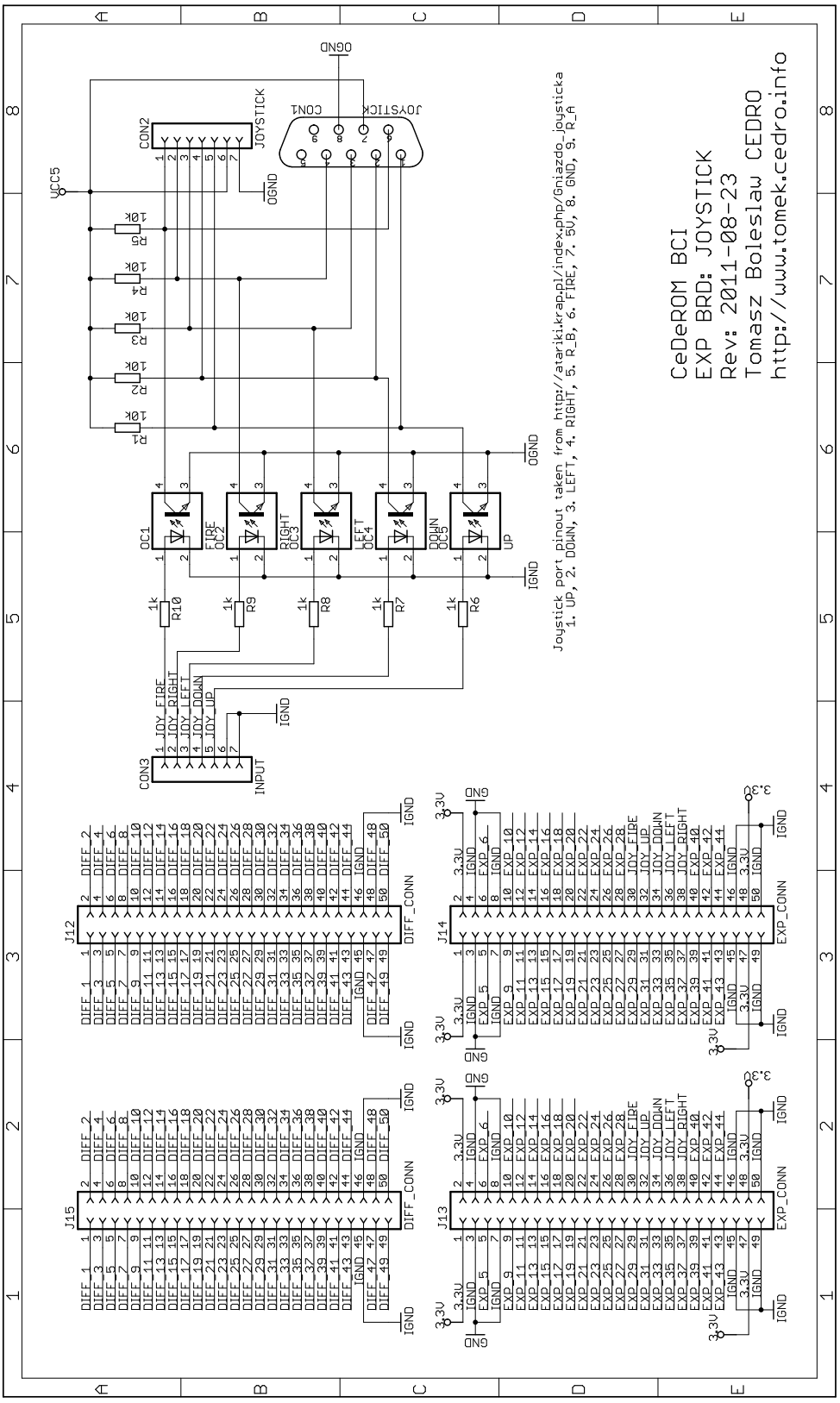
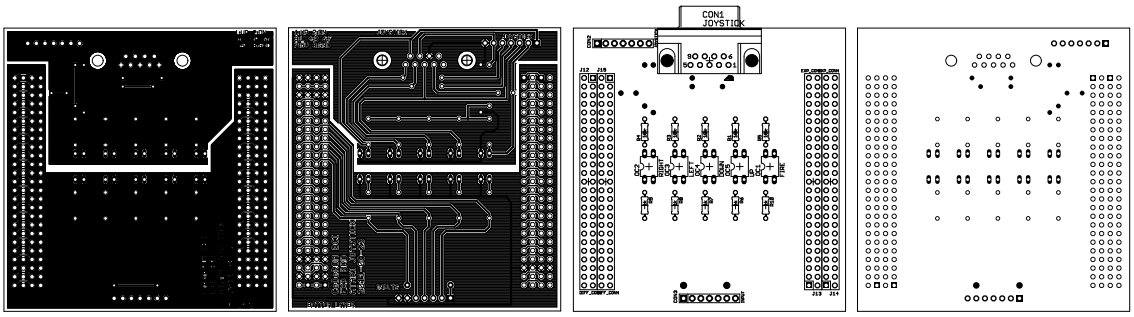


Figure 2.24: CeDeROM BCI Expansion Board: Atari Joystick Schematics.



(a) Top PCB (b) Bottom PCB (mirror) (c) Top Components (d) Bottom Components (mirror)

Figure 2.25: CeDeROM BCI Expansion Board: Atari Joystick PCB design.

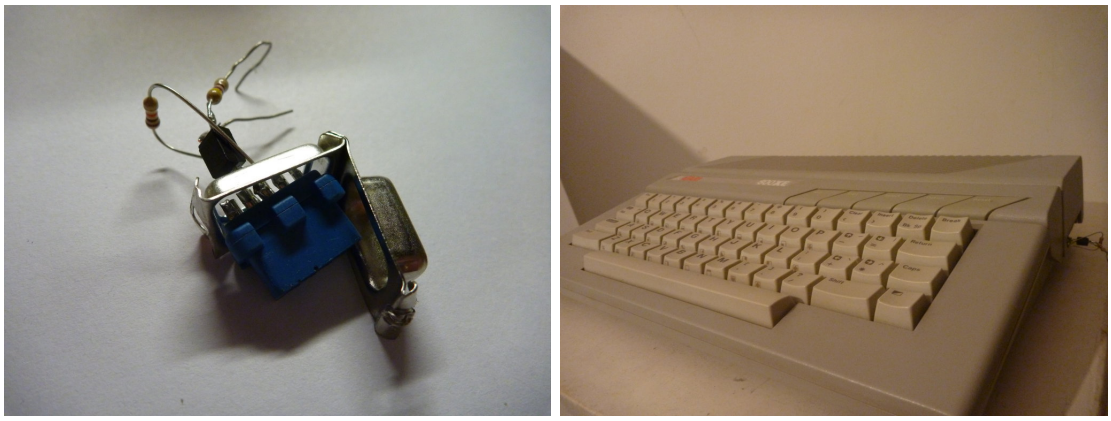


Figure 2.26: First steps of joystick interface design and testing on my precious Atari.

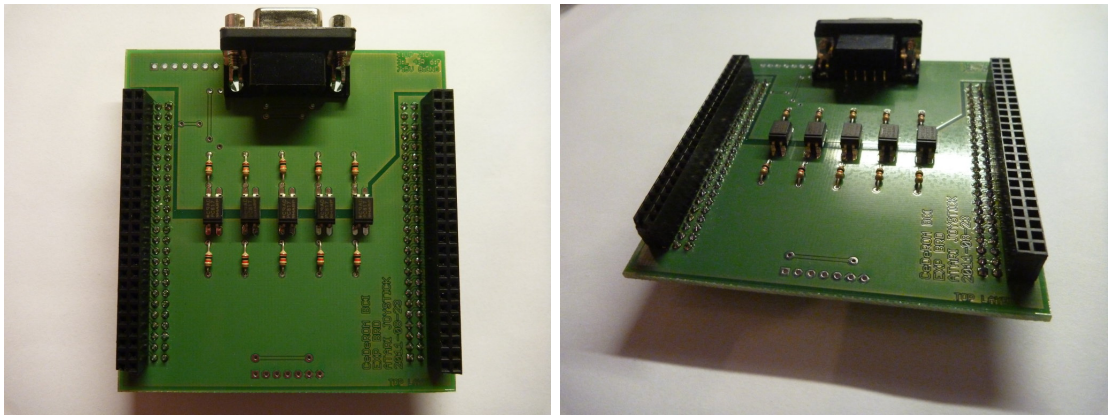


Figure 2.27: CeDeROM BCI Joystick Expansion Board, assembled.

Figure 2.24 presents joystick board schematics, Figure 2.25 presents PCB layouts and the component placementi, Figure 2.27 presents assembled circuit board. The first steps of the test circuit and the DB–9 plug modification trick are presented on Figure 2.26.

Joystick module can be used with both Mobile ARM module and stationary FPGA platform. The FPGA module in default configuration produces PONG videogame with VGA output, so joystick signals left–right are directly related with a control signals driving the movement of PONG player pad up–down. User can choose if he/she wants to play PONG on FPGA or Yoomp on Atari. It is possible to connect this module to ARM board and exchange information with Personal Computer running some additional software that will drive the joystick according to some user defined algorithm. When the algotithm is ready it can be processed and uploaded into FPGA device to work in a real time.

2.3 Example Usecases

This chapter presents example practical usecases that can be implemented with my modular research system. This chapter shows the biggest advantage of my solution and the approach to implement it – versatile and rapid system reconfiguration that can lead to a commercial product implemented using low lost (often free of charge) Open–Source applications that are developed and created along with this project, also making use in other projects unrelated with this research. The agile and smart approach with open attitude can bring more benefit to the resulting system. Its not perfect, its still evolving, its ready for new feature requests. Existing commercial closed source solutions also have their own problems and limitations but it is impossible to fix them or extend as required.

2.3.1 Standalone FPGA Application

As mentioned in chapter 2.2.1 FPGA equipment is a perfect solution for research and prototyping of various BCI applications that require realtime and/or computational power exceeding capabilities of an ordinary microprocessor devices but easily applicable in programable logical devices. Device can be used to verify dedicated mixed–signal applications designed and written on external computers from scratch or using dedicated Matlab Signal Processing Toolboxes. The possibilities of creating dedicated hardware accelerated designs seems endless, but also inexpensive functional prototyping of standalone applications before manufacturing is very important. After successful verification design can be syntesized for selected silicon technology process and send for ASIC (Application Specific Integrated Circuit [74]) manufacturing.

2.3.2 Standalone BCI–PONG Videogame

Example standalone application of CeDeROM BCI FPGA device contains PONG videogame with VGA display that is controlled with brainwave activity of the person playing the game. Simple EEG acquisition can done with use of `ADC_BRD: ADS1298` module (section 2.2.6) that transfer bistream into FPGA with hardware FFT implementation (or more

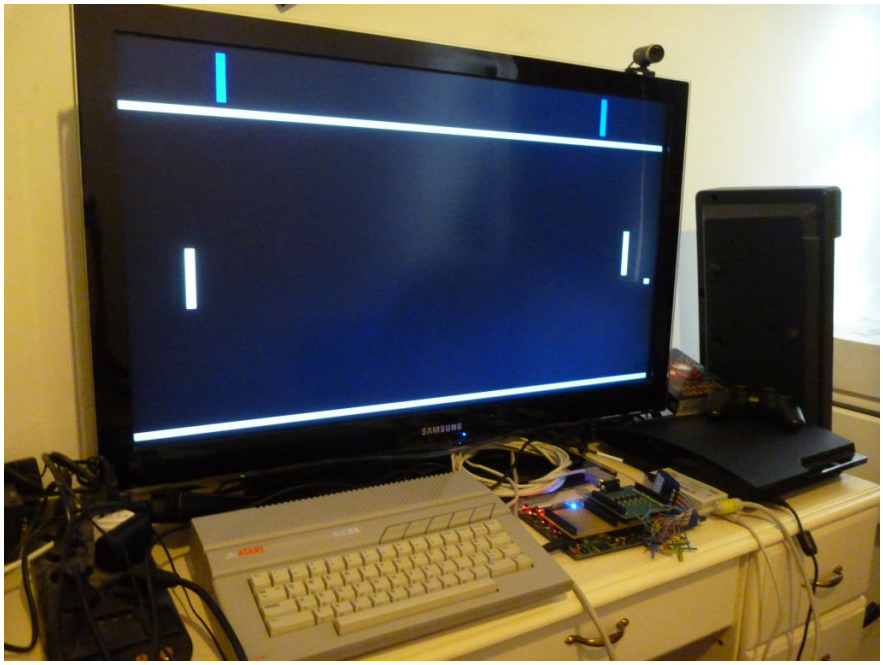


Figure 2.28: CeDeROM BCI FPGA acting as standalone PONG videogame.

sophisticated decomposition) that drives the videogame controls based on signal changes from selected range of frequencies. *Neurofeedback Alpha Training Technique* [71] [72] is a popular method of controlling brainwave activity in range of 8..12Hz so called *Alpha Waves* [73] also used in commercial and open–source biofeedback and bci computer applications such as BCI2000 [85], OpenViBE [86], BrainBay [87], so it probably can be used to control such videogames action.

It is possible to expand capabilities of such system by implementing different videogames and more sophisticated algorithms for brain activity pattern detection. Hardware can remain unchanged, only the logic will change, so it can become a firmware or lets say „game pack" that can be sold separately. This is also great news for scientific research as many different groups can use relatively simple and inexpensive common equipment to test and veriry their results such as acquisition method, filtering, decission making, based on different easily exchangable firmwares.

2.3.3 Universal Joystick Controller

With use of `EXP_BRD: Atari Joystick` (section 2.2.9) it is also possible to drive external systems such as videogame consoles, automation systems, etc. This example use additional hardware module to control external computer, an 8–bit Atari with *Yoomp* videogame loaded from cartridge memory after powerup.

Similar implementation can use built–in Ethernel controller to send data over Internet network to next room or another side of the planet. With dedicated hardware module system can act as HID (Human Interface Device [75]) to drive mouse cursor or write letters

Figure 2.29: CeDeROM BCI FPGA acting as (Atari) videogame controller.

on „mind–keyboard", using dedicated signal processing and classification algorithms.

Yoomp computer game inspired us so much in the Cybernetic Research Student Group that we have written dedicated computer game module for BCI2000 [85] system for research purposes on standard Personal Computer with all supported BCI hardware devices.

Because FPGA gives opportunity to include whole computer system in one chip, in near future I will try to fit whole 8–bit Atari machine inside single FPGA device, so the solution also becomes standalone.

2.3.4 Modern OpenEEG Replacement

The ARM–based module can provide unprecedented performance and parameters being still compatible with existing OpenEEG solutions. As reported on OpenEEG project mailing list there are users for who actual design is not enough and too outdated to use on modern computers. Implementing Virtual COM Port over USB and P2/P3 simple protocols will make it compatible with existing OpenEEG platform.

Therefore all parts of this project can serve as modern replacement for open BCI / Neurofeedback research platform. Final solution can be closed into one small device sold for small price also serving as funding source for my future research. Free and/or Open–Source applications that can work with OpenEEG include biofeedback and bci computer applications such as BCI2000 [85], OpenViBE [86], BrainBay [87] (Figure 2.31).

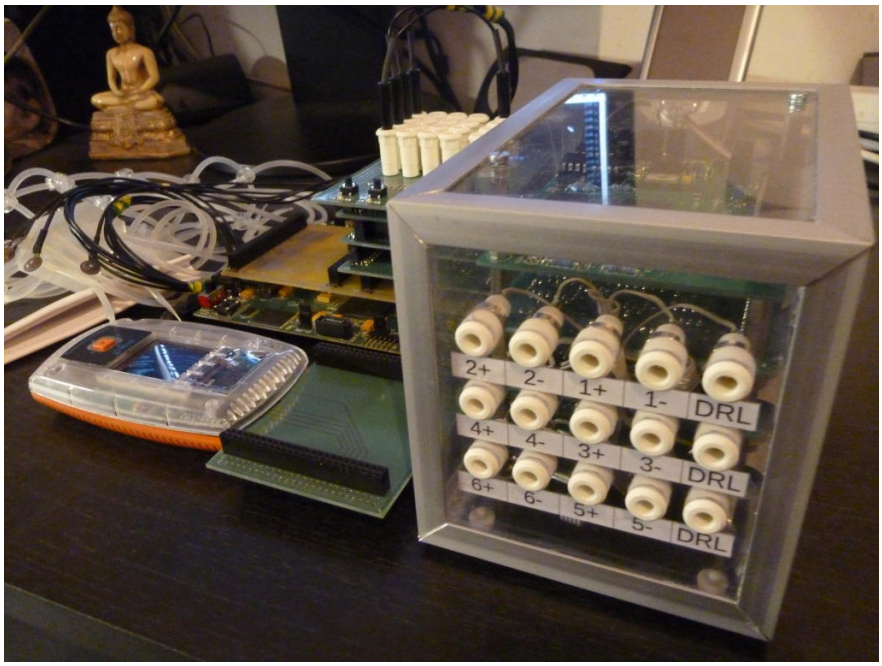
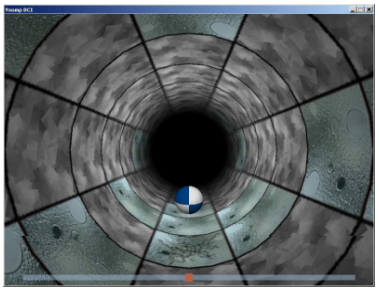
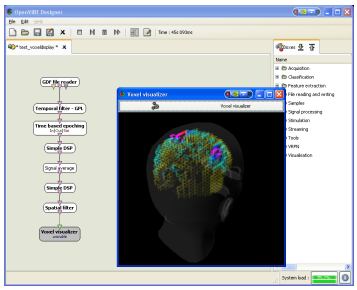


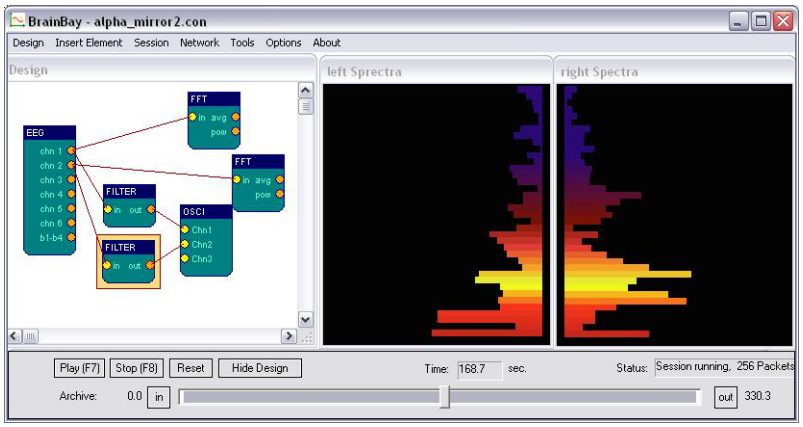
Figure 2.30: CeDeROM BCI ARM (left) replacement for OpenEEG (right).



(a) BCI2000



(b) OpenVibe



(c) BrainBay

Figure 2.31: Free applications to work with OpenEEG–like devices.

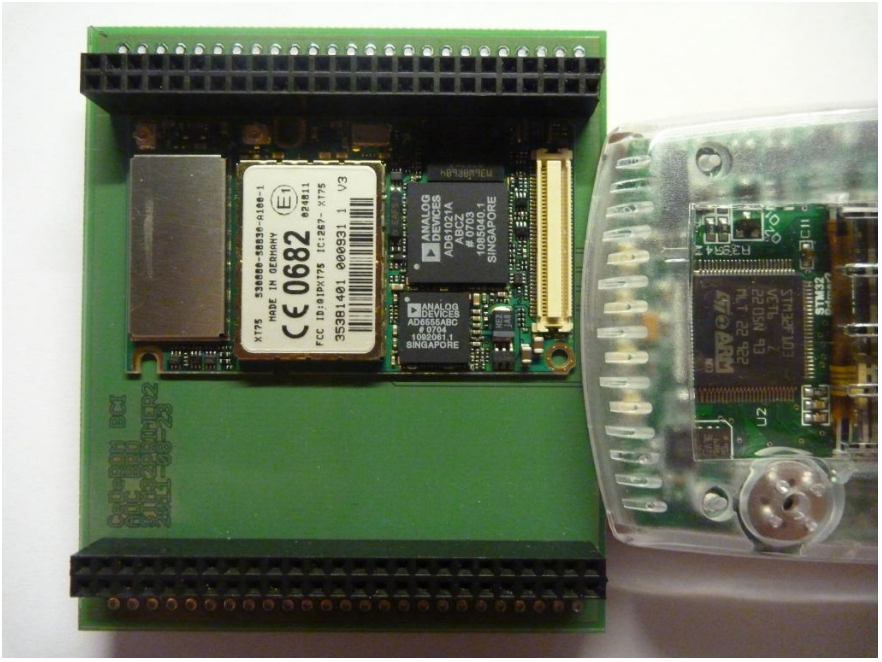


Figure 2.32: GSM/GPS module ready for use with CeDeROM BCI.

2.3.5 Mobile Holter

ARM–based module use Stm32Primer2 development board equipped with ARM–Cortex–M3 CPU, USB port, graphic display and uSD card slot for result storage. This makes it perfect candidate for mobile holter–like applications for constant monitoring of patient EKG, EEG, respiratory and other functions.

Optical Disk attached to the document contains lot of examples on FreeRTOS, USB, VCOMPort, uSD, FAT32, DSP and many more implemented on Stm32Primer development board. They can be treated just as software modules that can be matched with necessary hardware modules of CeDeROM BCI to create final solution.

Stm32Primer2 can be replaced or equipped with additional communication GSM/GPS module that will allow monitoring of patient location and alarm in case of emergency. The one presented on Figure 2.32, a Siemens XT75 GSM/GPS module, also features ARM core with Java VM, so it can even act as standalone CPU BOARD with no need to use Stm32Primer2 device...

2.3.6 Integrated Solutions

Because ARM is a CPU design, not a silicon itself, it is possible to create and verify new, dedicated design with my platform, to be then miniaturized, manufactured as ASIC (Application Specific Integrated Circuit), or become a part of bigger system.

Chapter 3

Summary and Conclusions

Experimental system presented in this document was designed to be a versatile reconfigurable modular platform for various research projects aimed at measurement and processing of bioelectrical signals, such as EEG, EKG, EMG, specifically a Brain Computer Interface project conducted by Cybernetic Research Student Group, that I am member and founder, providing low cost hardware and open–source software modules for various implementation tasks. This manuscript contains history and documentation of my work on the project. It is divided into two main parts – *Know How* and *Solution Approach.*

Know How (section 1) is an introductionary guide for people with no experience in electronics, biomedical enginering and IT in general, or people that consider taking part in such project but don't know yet what field of research would suit them best. Some of my own discoveries such as writing device drivers in Matlab (section 1.10), Serial Wire Debug implementation (section 1.11), or design of Brain Computer Interface Open Protocol (section 1.13) are documented in this part along with other information and skills necessary to make such system exist and work from the „logic behind the scenes" point of view. The secondary purpose of this documentation is to show how much knowledge and work is necessary in order to create even simplest device of this kind. Please keep in mind that this system is a work of a one man, so it would not be possible to create everything from scratch in such short time, but rebranding of existing commercial solutions would not bring any innovative solution. This is why I have decided to follow the harderst possible paths of creating new utilities for use in my projects, especially those which did not exist before, keeping in mind they will be free of charge and freely customizable in future. Such investment not only allows me to create better, smarter and cheaper solutions, but also share my results with other people if necessary. For instance implementation of the LibSWD even with external support took most of my time (unfortunately far more than planned, leaving no time for other tasks) but this will be the first in the world implementation of Serial Wire Debug Open Framework allowing everyone to program ARM Cortex devices. Ofcourse I could keep that secret, but I use efficient and extremely flexible Open–Source Software solutions in my everyday life, so it would be not fair only to take results of the others people work and give nothing in return. The financial gain is also possible in this model – it can be obtained by innovative implementation and useful work made by such solutions. Looters and parasites are everywhere, luckily they cannot

steal something that is already free. I am sure this attitude will help creating my own devices, but also better world that I live in for almost 30 past years of this life, giving other people opportunity to learn and create high quality things that I may find useful one day.

Solution Approach (section 2) presents physical electronic modules resembling the system with detailed description and schematics. System consists of control boards based on STM32 ARM–Cortex M3 (section 2.2.2) for mobile applications and FPGA board based on Xilinx Spartan 3A–DSP (section 2.2.1) for real time DSP, safe power supply with galvanic separation (section 2.2.5), general purpose hex 24–bit SigmaDelta ADC based on ADS1278 IC from BurrBrown / Texas Instruments (section 2.2.8) already tested with NXP's LPC2148 microcontroller, integrated biological signal acquisition frontend with 24–bit SigmaDelta ADC and SPI interface based on ADS1298 IC from BurrBrown / Texas Instruments (section 2.2.6), standard EEG electrodes connector board (section 2.2.7), user interaction with LED and push–buttons (integrated on electrodes board), and finally the demonstration expansion board with computer joystic interface (section 2.2.9) to control external hardware.

There are few possible commercial applications presented in *Example Usecases* (section 2.3) that can be implemented with both software and hardware components designed and identified during this research. Standalone FPGA applications can include mind driven PONG videogame with VGA output (section 2.3.2) or standalone videogame controller (section 2.3.3), but most of all it can serve as efficient realtime DSP or ASIC functional verification platform. ARM Cortex based example devices include modern OpenEEG [84] replacement (section 2.3.4) and mobile (Tele) EEG Holter (section 2.3.5). There are many free and open–source software examples and IP–Cores all over the Internet that makes it possible to implement such applications almost free of charge.

Not all tasks were successfully finished to fulfill requirements of a working commercial product, but considering limited time and resources, a good start for such platform has been made with clear direction of further development. Many methods discovered and presented in this document can be used in other areas of science and engineering, serving as good start point for interdisciplinary team work. Neural interfacing is a very hard and wide field of research, even highly experienced scientific groups around the world with great deal of commercial support still have problems to „control environment by thoughts". I hope my work proved that such system can be built from scratch using low–cost components, open–source software, skills, commitment and enough patience. Still there is more to accomplish that already has been done. I also belive that one day my hobby will turn into profitable business of useful, peaceful and helpful solutions.

Bibliography

[1] *Tomasz Bolesław CEDRO homepage* ,
`http://www.tomek.cedro.info`

[2] *Cybernetic Research Student Group, Warsaw University of Technology,*
`http://cyber.ise.pw.edu.pl`

[3] *Research Group on Biocybernetic Aparatus, Institute of Electronic Systems, Faculty of Electronics and Information Technologies, Warsaw University of Technology, Poland* ,
`http://www.ise.pw.edu.pl/index.php?id=138`

[4] *Nuclear and Medical Electronic Division, Institute of Radioelectronic Systems, Faculty of Electronics and Information Technologies, Warsaw University of Technology, Poland* ,
`http://www.ire.pw.edu.pl/zejim/`

[5] *Institute of Electronic Systems, Faculty of Electronics and Information Technologies, Warsaw University of Technology, Poland* ,
`http://www.ise.pw.edu.pl/`

[6] *Faculty of Electronics and Information Technologies, Warsaw University of Technology, Poland* ,
`http://www.elka.pw.edu.pl`

[7] *Warsaw University of Technology, Poland* ,
`http://www.pw.edu.pl`

[8] *Institute of Electrical and Electronics Engineers* ,
`http://www.ieee.org/`

[9] *Design and Development of Medical Electronic Instrumentation: A Practical Perspective of the Design, Construction, and Test of Medical Devices. David Prutchi (Author), Michael Norris (Author). Publisher: Wiley-Interscience; 1 edition (November 22, 2004). ISBN-10: 0471676233. ISBN-13: 978-0471676232.* ,
`http://home.comcast.net/~prutchi/`

[10] *Improving Common-Mode Rejection Using the Right-Leg Drive Amplifier, Texas Instruments* ,
`http://www.ti.com/litv/pdf/sbaa188`

[11] *High Speed Data Conversion, Texas Instruments* ,
`http://www.ti.com/litv/pdf/sbaa045`

[12] *Analog-to-Digital Converter Grounding Practices Affect System Performance, Texas Instruments* ,
`http://www.ti.com/litv/pdf/sbaa052`

[13] *Interleaving Analog-to-Digital Converters, Texas Instruments* ,
`http://www.ti.com/litv/pdf/sbaa049`

[14] *Thermal Noise Analysis in ECG Applications, Texas Instruments* ,
`http://www.ti.com/litv/pdf/sbaa185`

[15] *Principles of Data Acquisition and Conversion, Texas Instruments* ,
`http://www.ti.com/litv/pdf/sbaa051`

[16] *Analog Front-End Design for ECG Systems Using Delta-Sigma ADCs (Rev. A), Texas Instruments* ,
`http://www.ti.com/litv/pdf/sbaa160a`

[17] *A Glossary of Analog-to-Digital Specifications and Performance Characteristics (Rev. A), Texas Instruments* ,
`http://www.ti.com/litv/pdf/sbaa147a`

[18] *What Designers Should Know About Data Converter Drift, Texas Instruments* ,
`http://www.ti.com/litv/pdf/sbaa046`

[19] *Power Management for Precision Analog, Texas Instruments* ,
`http://www.ti.com/litv/pdf/slvt170`

[20] *Complete Analog Front End for ECG/EEG [WMV], Texas Instruments* ,
`http://www.ti.com/litv/wmv/sbac102`

[21] *Respiration Rate Measurement Using Impedance Pneumography, Texas Instruments* ,
`http://www.ti.com/litv/pdf/sbaa181`

[22] *Wikipedia, The Free Encyclopedia*,
`http://www.wikipedia.org`

[23] *The Free Software Foundation* ,
`http://www.fsf.org`

[24] *The GNU Operating System* ,
`http://www.gnu.org/`

[25] *GNU ARM Toolchain project* ,
`http://www.gnuarm.com`

[26] *Eclipse Integrated Desktop Environment (IDE)* ,
`http://www.eclipse.org`

[27] *YAGARTO: Yet another GNU ARM toolchain* ,
`http://www.yagarto.de/`

[28] *LibUSB project* ,
`http://libusb.sourceforge.net`

[29] *The MathWorks and the MatLab Software* ,
`http://www.mathworks.com`

[30] *Octave project, a GNU MatLab clone* ,
`http://www.gnu.org/software/octave`

[31] *SciLab project, yet another free MatLab clone* ,
`http://www.scilab.org`

[32] *Universal Serial Bus (USB) Specification* ,
`http://www.usb.org`

[33] *„USB in a NutShell" introductionary book to USB standard"* ,
`http://www.beyondlogic.org/usbnutshell`

[34] *The FreeBSD Operating System* ,
`http://www.freebsd.org`

[35] *The BSD License* ,
`http://www.opensource.org/licenses/bsd-license.php`

[36] *Open On–Chip–Debugger* ,
`http://openocd.sf.net`

[37] *Universal JTAG utility* ,
`http://urjtag.sf.net`

[38] *LibSWD, a Serial Wire Debug Open Library* ,
`http://libswd.sf.net`

[39] *Stm32Circle, home of Stm32Primer Development Kits* ,
`http://stm32circle.com/`

[40] *Raisonance, State-of-the Art Tools for the Microelectronics Industry* ,
`http://www.raisonance.com/`

[41] *My scratchpad on first in the world open source implementation of SWD for ARM–Cortex devices* ,
`http://stm32primer2swd.sf.net`

[42] *ARM Corporation* ,
`http://www.arm.com`

[43] *Serial Wire Debug* ,
`http://www.arm.com/products/system-ip/debug-trace/coresight-soc-components/serial-wire-debug.php`

[44] *ARM Debug Interface version 5 Specification*,
`http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0031a/index.html`

[45] *ARM–Cortex–M1 Technical Reference Manual* ,
`http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0413c/index.html`

[46] *ARM Information Center* ,
`http://infocenter.arm.com/`

[47] *KrisTech Company* ,
`http://www.kristech.pl/`

[48] *Intel Corportation* ,
`http://www.intel.com`

[49] *SAMSUNG Semiconductor* ,
`http://www.samsungsemi.com/`

[50] *Qualcomm Wireless Technology & Innovation* ,
`http://www.qualcomm.com/`

[51] *Texas Instruments Incorporated* ,
`http://www.ti.com/`

[52] *STMicroelectronics* ,
`http://www.st.com/`

[53] *Future Technology Devices International Ltd.* ,
`http://www.ftdichip.com/`

[54] *Samtec Incorporated* ,
`http://www.samtec.com/`

[55] *Multi-Contact AG - Advanced Contact Technology* ,
`http://www.multi-contact.com/`

[56] *Xilinx Incorporated, leading FPGA designer and manufacturer.* ,
`http://www.xilinx.com/`

[57] *Xilinx ISE Design Suite* ,
`http://www.xilinx.com/products/design-tools/ise-design-suite/`

[58] *Xilinx User Community Forums* ,
`http://forums.xilinx.com/`

[59] *ELF/ABI problem solution* ,
`http://forums.freebsd.org/showthread.php?t=17776`

[60] *XC3SProg, Open–Source Xilinx FPGA programming utility* ,
`http://xc3sprog.sourceforge.net/`

[61] *Xilinx Forum thread on unsupported SPI Flash memory on official development board* ,
`http://forums.xilinx.com/t5/Xilinx-Boards-and-Kits/program-upload-on-spartan-3a-dsp-1800-board/m-p/60194`

[62] *Eagle CAD Software, inexpensive and multiplatform Easily Applicable Graphical Layout Editor* ,
`http://www.cadsoft.de/` , `http://www.cadsoftusa.com/`

[63] *Eagle CAD licensing page* ,
`https://www.cadsoft.de/buy-eagle/`

[64] *Eagle CAD free–of–charge components and scripts repository* ,
`http://www.cadsoftusa.com/downloads/`

[65] *PCB Industry Standard Gerber Format* ,
`http://en.wikipedia.org/wiki/Gerber_format`

[66] *PCB Industry Standard Excellon Format* ,
`http://en.wikipedia.org/wiki/Excellon`

[67] *Serial Vector Format description* ,
`http://en.wikipedia.org/wiki/Serial_Vector_Format`

[68] *XILINX JTAG tools on Linux without proprietary kernel modules* ,
`http://rmdir.de/~michael/xilinx/`

[69] *XP Power, global power supply components for electronics industry* ,
`http://www.xppower.com/`

[70] *Murata Manufacturing Co. Ltd., global components for electronic industry* ,
http://www.murata.com/

[71] *Neurofeedback reference* ,
http://en.wikipedia.org/wiki/Neurofeedback

[72] *Hardt, J.V.; Kamiya, J. (1976). "Conflicting results in EEG alpha feedback studies". Applied Psychophysiology and Biofeedback 1 (1): 63–75. Retrieved 2007-12-05* ,
http://www.springerlink.com/index/Q6XU641775678664.pdf

[73] *Alpha Waves reference* ,
http://en.wikipedia.org/wiki/Alpha_wave

[74] *Application Specific Integrated Circuit reference* ,
http://en.wikipedia.org/wiki/Application-specific_integrated_circuit

[75] *Human Interface Device reference* ,
http://en.wikipedia.org/wiki/Human_interface_device

[76] *Texas Instruments Incorporated* ,
http://www.ti.com/

[77] *ADS1298, a complete integrated biological signal acquisition IC with 8 24–bit SigmaDelta ADC* ,
http://focus.ti.com/docs/prod/folders/print/ads1298.html

[78] *ADS1298–IPA TQFP Package reference* ,
http://www.ti.com/litv/pdf/mtqf006a

[79] *ADS1278–ZXG, BGA Package reference* ,
http://www.ti.com/litv/pdf/mpbg581

[80] *Getting Started With the ADS1298ECGFE-PDK [WMV]* ,
http://www.ti.com/litv/wmv/sbac111

[81] *ADS1278, octal differential input 24–bit SigmaDelta ADC* ,
http://focus.ti.com/docs/prod/folders/print/ads1278.html

[82] *DAC8552, 16-Bit, Dual-Channel, Ultralow Glitch, Voltage Output, Digital to Analog Converter* ,
http://focus.ti.com/docs/prod/folders/print/dac8552.html

[83] *Brain Implant – Science of Fiction?* ,
http://en.wikipedia.org/wiki/Brain_implant

[84] *The OpenEEG Project* ,
http://openeeg.sourceforge.net/

[85] *BCI2000 Software Suite* ,
`http://bci2000.org/`

[86] *OpenViBE, Software for Brain Computer Interfaces and Real Time Neurosciences* ,
`http://openvibe.inria.fr/`

[87] *BrainBay, an OpenSource Biosignal project* ,
`http://shifz.org/brainbay/`

[88] *NeuroSky, Brain Wave Sensors for Every Body* ,
`http://www.neurosky.com/`

[89] *Emotiv, Brain Computer Interface Technology* ,
`http://www.emotiv.com`

[90] *gTec Medical Engineering* ,
`http://www.gtec.at/`

[91] *Atari* ,
`http://www.atari.com/`

[92] *Yoomp! An 8-bit Atari Game* ,
`http://yoomp.atari.pl/`

[93] *Kontakt Chemie Industries* ,
`http://www.kontaktchemie.dk/`

[94] *LPKF Laser & Electronics AG* ,
`http://www.lpkf.de/`