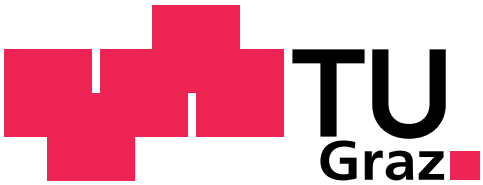
Oliver TERBU

iScope: a biosignal viewer on iOS® devices

Master's thesis



Institute for Knowledge Discovery
Laboratory of Brain-Computer Interfaces
Graz University of Technology
Krenngasse 37, 8010 Graz, Austria
Head: Assoc. Prof. Dipl.-Ing. Dr.techn. Gernot Müller-Putz

Supervisor: Dipl.-Ing. Dr.techn. Clemens Brunner

Evaluator: Assoc. Prof. Dipl.-Ing. Dr.techn. Gernot Müller-Putz

Graz, August 2012

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am 12.8.2012

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, 12th August 2012

Abstract

iScope is a mobile application running on devices with at least iPhone OS (iOS®) 4.0 enabling online visualization of biosignals. During brain-computer interface (BCI) experiments or more generally biosignal measurements, it is challenging to automatically remove noise from biosignals like muscle activity contained in an electroencephalogram (EEG). For this purpose, iScope provides quality monitoring to allow trained persons to take appropriate measures if artifact contamination is observed. Thereby, samples are provided by SignalServer and collected from wireless network using TiA library. The application implements a graphical user interface operated by extended well-known iOS® gestures and the following: managing connections to SignalServer, channel selection, displaying metadata, switching between two types of visualization for time- and spectral domain with different levels of detail, zooming of time dimension, two scaling modes for value dimension, flexibly arranging layout, screen capturing, a custom settings screen and integration into Settings App. iScope has been released as an ad-hoc distribution and can already assist during measurements. Finally, the application performs at an performance level that is able to visualize sampling rates which exploit the device's maximum resolution. In the future, iScope's channel selection feature can be combined with the upcoming selective channel transmission function of SignalServer.

Kurzfassung

iScope ist eine mobile Applikation zur online Visualisierung von Biosignalen für Geräte mit iOS® ab der Version 4.0. Während eines Brain-Computer Interface (BCI) Experimentes oder generell, einer Biosignalmessung ist es schwierig automatisch Störsignale aus den Biosignalen zu entfernen, wie z.B. Muskelaktiviät aus dem Elektroenzephalogramm (EEG). Hier kann iScope zur Überwachung der Qualität der Biosignale genutzt werden. Werden Artefakte im Biosignal entdeckt, kann dadurch geschultes Personal noch rechtzeitig angemessene Maßnahmen ergreifen. SignalServer stellt die Messwerte über drahtloses Netzwerk bereit, welche mit Hilfe der TiA Bibliothek empfangen werden. Eine graphische Benuteroberfläche wurde implementiert, welche mit bekannten, teils modifzierten Fingerbewegungen kontrolliert wird und folgende Funktionalität anbietet: Verwalten der Verbindung zu SignalServer, Selektion von Kanälen, Anzeige von Metadaten, Wechsel zwischen zwei Visualisierungarten von Zeit- und Spektraldomäne in unterschiedlichen Detailstufen, Zoom der Zeitdimension, zwei Skalierungsmodi der Wertedimension, flexible Anordnung der Graphen, Bildschirmaufnahme, einen internen Konfigurationsbildschirm und Integration in Settings App. iScope ist als Ad-hoc Distribution verfügbar und kann bei Messungen bereits eingesetzt werden. Schliesslich ist es auch möglich, Abtastfrequenzen zu visualisieren, welche die maximale Bildschirmauflösung des Gerätes ausnutzen. Im nächsten Schritt soll iScopes Kanalselektion mit der bald verfügbaren selektiven Kanalübetragung von SignalServer gekoppelt werden.

Acknowledgements

First, I would like to thank my supervisor Clemens Brunner and evaluator Gernot Müller-Putz for giving me the opportunity of writing my master thesis at the Laboratory of Brain-Computer Interfaces. I also want to thank all other colleagues at the Institute for Knowledge Discovery as well as Stefanie Lindstaedt for the cooperation with Knowledge Management Institute and Know Center. After all, I want to specially express gratitude to Simone, Renate and Hans.

Contents

Abstract iii

Kurzfassung iv

Acknowledgements v

1. Introduction 1

1.1. Background . . . 1
1.2. Motivation . . . 2
1.3. Aim . . . 3
1.4. Overview . . . 3

2. Related Work 4

2.1. Fourier Analysis/Synthesis . . . 4
2.1.1. Discrete Fourier Transform . . . 4
2.1.2. Short-Time Fourier Transform . . . 6
2.1.3. Fast Fourier Transform . . . 6
2.2. iOS® 4.x . . . 7
2.2.1. Characteristics of iOS® Devices and Apps . . . 7
2.2.2. Technical Specifications of iOS® Devices . . . 9
2.2.3. Floating Point Operations on iOS® Platforms . . . 9
2.3. iOS® SDK Toolchain . . . 11
2.3.1. XCode® 3.2 . . . 11
2.3.2. Interface Builder . . . 12
2.3.3. Instruments® . . . 12
2.4. Related iOS® Frameworks . . . 12
2.4.1. Objective-C® 2.0 . . . 13
2.4.2. Common Design Patterns . . . 13
2.4.3. Foundation Framework . . . 14

2.4.4. UIKit Framework . . . 14
2.4.5. Message UI Framework . . . 19
2.4.6. Accelerate Framework . . . 19
2.4.7. Core Graphics Framework . . . 20
2.5. Other related Tools, Frameworks and Libraries . . . 21
2.5.1. Boost Libraries . . . 21
2.5.2. TinyXML++ Library . . . 22
2.5.3. TiA Library . . . 22
2.5.4. SignalServer . . . 23
2.5.5. Three 20 Library . . . 25
2.5.6. CorePlot Library . . . 25
2.6. Software Licensing . . . 25

3. Implementation Aspects **27**
3.1. Requirements . . . 27
3.1.1. Non-functional Requirements . . . 27
3.1.2. Functional Requirements . . . 27
3.2. General Design Decisions . . . 29
3.3. System Architecture . . . 30
3.3.1. System Overview . . . 30
3.3.2. Automatic Testing . . . 31
3.4. Environment . . . 32
3.4.1. Platform . . . 32
3.4.2. Compiler and Linker . . . 32
3.5. Common Module . . . 33
3.5.1. Managing and Passing Data Samples . . . 34
3.5.2. Generic User Interface Elements . . . 38
3.5.3. Detecting extended Pinch Gestures . . . 39
3.5.4. Spectrum Computation . . . 40
3.5.5. Device Categorization . . . 41
3.6. Signal Server Client Module . . . 41
3.6.1. Building TinyXML++ Library for iOS® . . . 41
3.6.2. Building TiA Library for iOS® . . . 41
3.6.3. Signal Server Client Library . . . 42

3.7. Layout Manager Module . . . 50
3.7.1. Adding Items . . . 52
3.7.2. Applying Layout . . . 52
3.7.3. Changing visible Items . . . 54
3.7.4. (De-)magnifying Items . . . 54
3.7.5. Dragging and Reordering Items . . . 55
3.8. Graphs Module . . . 60
3.8.1. Drawing continuous Content . . . 60
3.8.2. Graph Views . . . 71
3.8.3. View Controllers . . . 77
3.9. iScope - App Module . . . 83
3.9.1. General . . . 83
3.9.2. Application Start and Shutdown . . . 86
3.9.3. Settings App . . . 86
3.9.4. Multi-Language Support . . . 87
3.9.5. Connection Screen . . . 89
3.9.6. Selection Screen . . . 89
3.9.7. Visualization Screen . . . 91
3.9.8. Settings Screen . . . 95
3.9.9. Performance Limits . . . 96

4. Deployment **99**
4.1. Building from Source . . . 99
4.2. Development & Testing . . . 100
4.3. Distribution . . . 100
4.3.1. Ad-Hoc Distribution . . . 100
4.3.2. In-House Distribution . . . 101
4.3.3. App Store℠ . . . 101
4.4. Licensing . . . 101
4.4.1. Licensing Limitations . . . 102

5. Summary and Conclusion **104**

6. Outlook **106**

Bibliography **108**

A. Appendix **116**
A.1. Benchmarking BCiDataBuffer with BCiCommonBenchmarks 116
A.2. Building Boost Libraries for iOS® 118
A.3. Performance Overview of Signal Server Client Module 122
A.4. Legal Disclaimer 125

List of Acronyms

1G	First Generation
2G	Second Generation
3G	Third Generation
4G	Fourth Generation
AGPL	Affero General Public License
API	Application Programming Interface
BCI	Brain-Computer Interface
BSD	Berkeley Software Distribution
CPU	Central Processing Unit
DC	Direct Current
DFT	Discrete Fourier Transform
DSP	Digital Signal Processing
ECG	Electrocardiogram
EEG	Electroencephalogram
EMG	Electromyogram
EOG	Electrooculogram
EPSP	Excitatory Postsynaptic Potential
ERD	Event-related Desynchronization
ERP	Event-related Potential

ERS	Event-related Synchronization
FFT	Fast Fourier Transform
FP7	Seventh Framework Programme
GCC	GNU Compiler Collection
GNU	GNU's Not UNIX
GPL	General Public License
GPLv2	GPL version 2
hBCI	hybrid BCI
HSB	Hue Saturation Brightness
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
iOS	iPhone OS
IPSP	Inhibitory Postsynaptic Potential
ISO	International Organization for Standardization
LLVM	Low Level Virtual Machine
MIT	Massachusetts Institute of Technology
MVC	Model View Controller
RAM	Random-Access Memory
RGB	Red Green Blue
RTTI	Runtime-Time Type Information
SD	Standard Deviation
SDK	Standard Development Kit
SIMD	Single Instruction Multiple Data

SMS	Short Message Service
STFT	Short-Time Fourier Transform
STL	Standard Template Library
SVN	Subversion
TCP	Transmission Control Protocol
TiA	TOBI interface A
TOBI	Tools for Brain-Computer Interaction
ToS	Terms of Service
UCS	Universal Character Set
UDP	User Datagram Protocol
UNIX	UNiplexed Information and Computing System
UTF-16	Universal Multiple-Octet Coded Character Set UCS Transformation Format for 16 Planes of Group 00
VFP	Vector Floating-point Architecture
VoIP	Voice over IP
WYSIWYG	What You See Is What You Get
X11	X Version 11
XML	Extensible Markup Language

1. Introduction

1.1. Background

The non-invasive and portable Electroencephalogram (EEG) is an especially interesting method for measuring brain activity. Thereby, very small electric potentials (μV) have to be recorded which are the sum of Excitatory Postsynaptic Potentials (EPSPs) and Inhibitory Postsynaptic Potentials (IPSPs) of synchronized and similarly spatially adjusted neurons near the surface. Over a time period, amplifiers are used to record potential changes with a number of electrodes on the scalp. [1, 2, 3] Generally, EEG features a very low signal-to-noise-ratio and poor spatial resolution with a measurment accuracy given in cm. On the other hand, good resolution in time (ms) is possible. [4, 5] Typically, the spontaneous EEG shows rhythmic osciallations with various frequencies depending on the constitution of the subject (e.g., alpha waves are present if the subject is relaxed and awake). [6] Furthermore, other specific patterns with lower amplitudes could be detected such as Event-related Potentials (ERPs) that emerge in response to external stimuli or Event-related Desynchronizations (ERDs)/Event-related Synchronizations (ERSs) mainly induced by internal processes (e.g., motor imagery). However, signal processing is required to reveal these phenomena. [7]

Tools for Brain-Computer Interaction (TOBI)[8] is an European Seventh Framework Programme (FP7) project. One aim of this project is to develop prototypes in the area of brain-computer interaction based on biosignals (e.g., EEG) and assistive devices (e.g., buttons and joysticks). [8, 9]

TOBI interface A (TiA) [10] originated from TOBI and intends a standardization of raw biosignals transmission. It specifies a general interface to transmit and receive data from various types of biosignals - such as EEG, Electrocardiogram (ECG), Electromyogram (EMG) or Electrooculogram (EOG) - over seperate control- and data connections by Transmission Control Protocol (TCP) and/or User Datagram Protocol (UDP). The interface is designed as a single-server/multiple-client model whereas the server acquires data from hardware devices (e.g., EEG amplifier) and clients can connect to the server

at runtime to collect data online for further processing during a measurement. In this manner, underlying data acquisition implementation is entirely encapsulated by the server component and clients no longer require knowledge about specific hardware devices. Instead, it is sufficient to communicate through TiA interfaces. [8, 11, 12]

Although TiA can be used for any biosignal processing application, it is specifically intended to unify different Brain-Computer Interface (BCI)- and hybrid BCI (hBCI) systems. [11] Usually, these systems enable one person to control a computer by exploiting electrophysiological phenomena without the necessity of involving peripheral nerves and muscles. BCIs primary extract these phenomena (e.g., ERD/ERS) or features from EEGs. In general, fundamental components of a BCI comprise signal acquisition (e.g., recording EEG), feature extraction (e.g., spatial and/or temporal filtering), feature translation (i.e. features are mapped to device commands) and the output device that gives the user feedback. Feedback educates the user and improves the efficiency and effectiveness of handling the BCI. [13] More flexible systems that incorporate other BCIs or use additional types of biosignals are called hBCIs. [14, 15, 16]

The TiA interface specifically abstracts the stage between signal acquisition and feature extraction. A server implementing the TiA interface might provide data from biosignals to the feature extraction unit for further processing. [11]

1.2. Motivation

During a biosignal measurement, undesired signals or noise (artifacts) are also recorded which interferes with or precludes the detection of desired phenomena. Basically, two categories of artifacts can be distinguished. The first one are non-physiological artifacts which can be avoided by appropriate precautions. For instance, 50/60 Hz noise of the power line or changes of electrodes impedance could affect the EEG. In contrast, the second category addresses physiological artifacts that are often inevitable because they are caused by respiration or ECG (e.g., heart activity), both resulting in rhythmic noise. In the field of BCIs, EOG (e.g., eye blinking) and EMG (e.g., jaw clenching) related artifacts are significantly important because they can mistakenly control the output device instead of intended phenomena. [12, 17, 18]

One possible solution could be implemented by automatically detecting and removing artifacts by dedicated algorithms (e.g., linear filtering). This method has the disadvantage that the EEG would be also suppressed as it is a stochastic signal as well. Nevertheless,

good results are archived for eliminating specific artifacts like ECG as opposed to EMG. It has a larger disturbance impact on the EEG and could not be entirely automatically handled by filtering. For this reason, quality monitoring is important to identify heavy artifact contamination at an early stage of an ongoing measurement, thus preventing rejection and repetition. Often, a mobile monitoring tool is desired because measurements are performed outside of the laboratory where space is limited, no desktop environment could be taken or set up. [12, 17, 18]

1.3. Aim

The aim of this master's thesis is the development of an application (named iScope) that receives biosignal samples via TiA server for subsequent online visualization to monitor respective time- and frequency domain. Then, iScope could be used as a monitoring tool for biosignal measurements. The focus does not concentrate on the exact analysis of biosignals that could rather be done by means of other tools (e.g., SigViewer [19]).

iPhone® OS (iOS®) developed by Apple® was selected as the target platform due to its popularity and intuitive user interface facilitating also less computer skilled people. iPhone® 3GS with iOS® 4.0 is defined as the minimum requirement. However, other iOS® devices (e.g., iPad® 1G, iPod touch® 3G) should be also able to run iScope if at least iOS® 4.0 is installed. Maintenance and further developement have to be also considered as when developing usual desktop applications.

1.4. Overview

Chapter 2 introduces basic knowledge about algorithms, frameworks and concepts required to understand the development and functionality of iScope. Internal design decisions, features of every implemented module and how all modules merge together to build up the final iOS® appplication is covered by Chapter 3. Chapter 4 explains the context and methods of how to deploy iScope as well as demonstrating alternative deployment solutions in respect to legal aspects. Finally, Chapter 5 summarizes and concludes the current development state whereas Chapter 6 provides an outlook about potential future tasks.

2. Related Work

2.1. Fourier Analysis/Synthesis

Basically, the Fourier Analysis/Synthesis deals with decomposition of different kind of signals into corresponding sine and cosine oszillations (Fourier components). This enables the transformation of signals from the time domain to the spectral domain and vice versa. According to the nature of the domains (continuous or discrete), a specific method such as Discrete Fourier Transform (DFT) is applicable. [20]

2.1.1. Discrete Fourier Transform

In practice, signals are not present as continuous functions that are defined for every point in time, rather discretized in the time domain by sampling. Because computers are not able to process continuous data, also the frequency spectrum has to be discrete to enable automatic processing. For this purpose, the DFT addresses both, a discrete time signal and a discrete spectrum. [20]

The complex Fourier components X_m (or bins) are calculated from time samples (x_n) with Formula 2.1. In doing so, N corresponds to the number of discrete values. Inversely, these time samples are recovered from the complex spectrum with Formula 2.2. Thereby, every bin represents information about a specific frequency. Depending on utilizing the magnitude, the squared magnitude or the argument of X_m, different kind of spectra will be computed. In this manner, $|X_m|$ results in the amplitude-, $|X_m|^2$ in the power- and $arg(X_m)$ in the phase spectrum. [20, 21]

The set of bins could be considered as a grid, where every mesh refers to a frequency $f_{m+1} = f_m + \Delta f$ starting with $f_0 = 0$ Hz. Further, the grid size would be $\Delta f = {}^{f_s}/_N$, where f_s is the sampling frequency. Then, the Nyquist frequency ${}^{f_s}/_2$ would be located at the m-th bin where $m = {}^N/_2$. If N is an odd number, the Nyquist frequency will not correspond to a single mesh. [20, 21]

The DFT implies that the time function as well as the spectrum are periodic. Furthermore,

the spectrum of the real DFT is also symmetric. Hence, only $\frac{N}{2} + 1$ unique outputs are generated and required to represent the whole spectrum. In this manner, Direct Current (DC)- and Nyquist components have no phase information. During discretization, the adherence of the sampling theorem is fundamental to ensure the correctness of the spectra. If the sampled time signal includes frequencies greater than the Nyquist frequency, aliasing will appear. [20, 21]

The DFT is applied only on one desirable periodic finite time window of the discrete time signal. Mathematically, a window function is used that equals the multiplication of the time signal with the rectangular function. The window function is defined as 1 within and 0 outside the window. Practically, it is often impossible to select a perfect periodic window because contained frequencies often are not entirely predictable. If the length of the window does not match the period (or a multiply) of all oscillations within the time signal, the results will be a mere approximation. In this case, if the window would be arranged in a series repeatedly, a discontinuity could be observed between the end of one instance and the beginning of the next one. As a consequence, frequencies would be included that do not equal any of the discrete frequencies. Neighbouring bins would approximate these frequencies instead which leads to blured spectra. This phenomenom is also called leakage effect. Other window functions exist to suppress the leakage effect by fading in/out the signal at the boundaries of the window to facilitate a more periodic character. [20, 21]

$$\begin{aligned} X_m &= \sum_{n=0}^{N-1} x_n \cdot e^{\left[\frac{-i2\pi}{N} nm\right]} \\ m, n &= [0, N-1] \end{aligned}$$

Formula 2.1: Discrete Fourier Transform

$$\begin{aligned} x_n &= {}^1\!/_N \cdot \sum_{m=0}^{N-1} X_m \cdot e^{\left[\frac{i2\pi}{N} nm\right]} \\ n, m &= [0, N-1] \end{aligned}$$

Formula 2.2: Inverse Discrete Fourier Transform

2.1.2. Short-Time Fourier Transform

The DFT itself does not reflect the changes of spectra over a period of time. For this particular purpose, the DFT has to be applied in periodic intervals on different time windows of the same length. This process is also known as Short-Time Fourier Transform (STFT). The output of the STFT is a spectogram and often presented as a color-coded graph. The selected windows is either sliding (overlapping) or non-overlapping. In this manner, always a tradeoff between time resolution and granularity of the frequency spectrum exists. If the chosen window length is too large, it will not be possible to track changes over time accurately. Nevertheless, this will lead to increased spectra resolution. In reverse, a too short window length would enhance time resolution but also degrade frequency resolution. [20, 21]

2.1.3. Fast Fourier Transform

A naive implementation of the DFT will have an asymptotic runtime complexity of $\mathcal{O}(N^2)$ that is not adequate for real time applications involving huge time windows. Cooley and Tukey introduced a class of more efficient algorithms known as Fast Fourier Transform (FFT) for computing the DFT with a complexity of $\mathcal{O}(N \cdot \log N)$. These algorithms exploit several characteristics of the DFT such as periodicity and symmetry. Basically, the DFT of N samples is computed recursivly as a result of smaller DFTs until no further splits are possible. This fundamental principle is also called divide and conquer. Many varitions exist that apply different radixes, require specific number of samples and perform either decimation-in-time or decimation-in-frequency. In this context, algorithmns relying on N being a power of two (called power-of-two algorithmn) are especially efficient. Of course, sometimes N cannot be a power of two for specific time windows. In order to put things right, the missing samples have to be padded with zeros. In this process, no new information is generated because the sampling frequeny has not changed at all. Nevertheless, zero padding impacts the resolution of the spectrum by decreasing Δf. It is important to apply explicit window functions before zeros are padded. [20, 21]

However, other algorithm classes (e.g., based on prime factorization) also implement the DFT fast and refer to the term FFT but those are not relevant for this thesis. [20, 21]

2.2. iOS® 4.x

Generally, iOS® is an UNIX®-based operating system that runs on any iPhone®, iPod touch® or iPad®. iOS® is built on a variation of the same March kernel as OS X® and uses similar Berkeley Software Distribution (BSD) interfaces. It also provides individual technologies to support native applications. [22, 23]

The iOS® Standard Development Kit (SDK) contains tools, libraries and header files that are required to develop applications for iOS®. In this thesis, the term app is used as a synonym for applications and iOS® will always refer to iOS® versions 4.x. Figure 2.1 gives an overview over all iOS® technology layers. Furthermore, it shows all those frameworks that are especially important in this thesis. [22, 23, 24, 25]

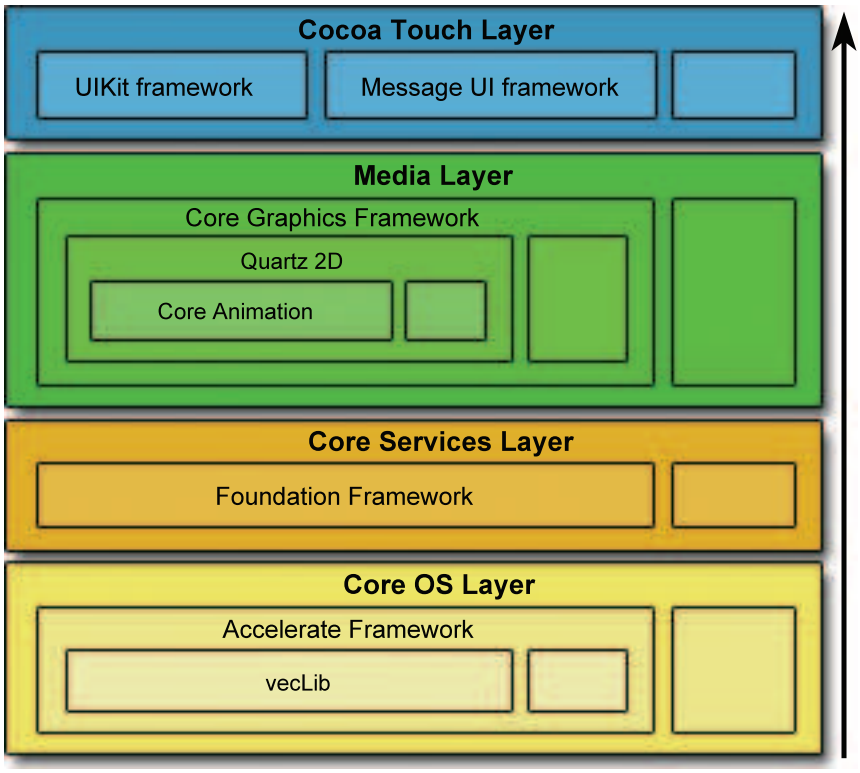


Figure 2.1.: Hierarchic overview of iOS® technology layers referred by this thesis

2.2.1. Characteristics of iOS® Devices and Apps

Although conventional desktop applications and iOS® apps have a lot in common, many aspects differ:

- iOS® applications do not rely on a mouse to manipulate or interact with the graphical user interface. Instead, control elements are operated by touching the screen of the device which creates so called multi-touch sequences (see Section 2.4.4). [23, 24, 25]

- In place of a traditional hardware keyboard, a simulated keyboard automatically becomes visible whenever the user is prompted to enter some text. Compared to a usual desktop, additional hardware is also embedded, depending on the actual device. Amongst others, an accelerometer is integrated that recognizes physical movements to determine the position of the device. [23, 25]

- At the same time, only one application is active and visible to the user. Since iOS® 4, it is possible to run an app in a restricted background mode after the user hits the home button. But this is restrained to a couple of specific tasks and should not be confused with true multitasking which are supposed by modern desktop operating systems. [22, 25]

- Every iOS® app has exactly one window and is not allowed to spawn other ones. The dimensions of the window are statically fitted to the screen size. [25]

- Only a specific part of the filesystem is accessible which was specifically created for the app, called the application's sandbox. Access to other system resources is also restricted, such as low-number network ports. This limitation also indirectly suspends the use of custom dynamic libraries or frameworks. [22, 25]

- Above all other hardware system components (e.g., processor), particularly the physical Random-Access Memory (RAM) is constrained in comparison to state-of-the-art computers. iOS® devices do not perform disk swaps when running out of phsysical memory due to the fact that virtual memory is directly linked to physical memory. Consequently, iOS® apps have to particularly focus on memory awareness. [22, 25, 26]

- Launching, suspending and terminating an app has to take minimum response time. If an app is not able to finish all initializing- or cleanup efforts within five seconds after the user started or closed the app, iOS® will kill the corresponding process immediately. [22, 25]

- When this thesis was written, every available iOS® device offered less screen resolution than a modern computer. Further, the cheapest MacBook® featured

a display of 1280 × 800 pixels, latest iOS® devices only provided a resolution of 1024 × 768 pixels. [25]

- There are also several restrictions and differences in deploying an app but this is covered separately in Chapter 4.

2.2.2. Technical Specifications of iOS® Devices

Apple Inc. does not expose many details about processors which are integrated in their devices. However, by default the official Integrated Development Environment (IDE) of the iOS® SDK provides builts for ARM®v6- and ARM®v7 architectures. Also the iOS® SDK documentation distincts between developing for these two architectures in some sections. Due to the outstanding popularity of Apple Inc.'s smartphones, many inofficial websites also reveal information about assembled hardware components gained through hardware teardowns. Table 2.1, Table 2.2 and Table 2.3 list some technical specifications about various iOS® devices that were taken into account when the practical part of this thesis was developed. [22, 25, 26, 27, 28, 29]

	iPod touch® 1G	iPod touch® 2G	iPod touch® *) 3G	iPod touch® 4G
RAM (MB):	128	128	256	256
Processor:	ARM®11	ARM®11	ARM® Cortex™-A8	ARM Cortex™-A8
Architecture:	ARM®v6	ARM®v6	ARM®v7	ARM®v7
Underclocked to (MHz):	412	533	600	800
Wi-Fi:	802.11 b/g	802.11 b/g	802.11 b/g	802.11 b/g/n
Resolution (pixel):	320 × 480	320 × 480	320 × 480	960 × 640

Table 2.1.: Hardware specifications (iPod touch® devices) - *) only(!) 32GB and 64GB versions; 8GB version shows technical details of iPod touch® devices 2G [28, 29, 30, 31, 32, 33]

2.2.3. Floating Point Operations on iOS® Platforms

iOS® devices according to ARM®v6- or ARM®v7 instruction sets support double- as well as single-precision floating point operations on the hardware. All iOS® ARM®v6 devices utilize the sequential Vector Floating-point Architecture (VFP) at same speed for single-

	iPhone® 1G	iPhone® 3G	iPhone® 3GS	iPhone® 4G
RAM (MB):	128	128	256	512
Processor:	ARM®11	ARM®11	ARM® Cortex™-A8	ARM Cortex™-A8
Architecture:	ARM®v6	ARM®v6	ARM®v7	ARM®v7
Underclocked to (MHz):	412	412	600	800
Wi-Fi:	802.11 b/g	802.11 b/g	802.11 b/g	802.11 b/g/n
Resolution (pixel):	320 × 480	320 × 480	320 × 480	960 × 640

Table 2.2.: Hardware specification (iPhone® devices) [28, 34, 35, 36, 37]

	iPad®	iPad® 2
RAM (MB):	256	512
Processor:	ARM® Cortex™-A8	ARM® Cortex™-A9
Architecture:	ARM®v7	ARM®v7
Underclocked to:	1GHz	1GHz Dual Core
Wi-Fi:	802.11 b/g/n	802.11 b/g/n
Resolution (pixel):	1024 × 768 px	1024 × 768 px

Table 2.3.: Hardware specifications (iPad® devices) [28, 38, 39]

and double-precision floating point arithmetic. In contrast, iOS® devices with an ARM® Cortex™A8/A9 implementing the ARM®v7 architecture integrate a reduced version of the VFP but have an Advanced Single Instruction Multiple Data (SIMD) co-processor called NEON® in addition. Unfortunately, NEON® is only able to process single-precision data. Thus, double-precision arithmetic on ARM®v7 devices is performed almost as fast as on ARM®v6 devices, whereas single-precision operations are substantially performed faster due to assigning NEON® instead of VFP. ARM Limited claims a large possible performance boost ($4 - 8\times$) on for instance simple Digital Signal Processing (DSP) algorithmns, but at least an increase by a factor of two compared to ARM®v6 processors in combination with VFP. Beside framework functions that explictly make use of intrinsic hardware accelerated functions, the compiler also tries to optimize code for specific architectures (e.g., NEON®). [27, 28, 40, 41, 42, 43]

Thumb® instructions are a subset of ARM® instructions and are 16 bit in size instead of 32 bit. This factor reduces code size, saves memory and cache. However, it is recommended to guide the compiler to use Thumb® instructions only when deploying for ARM®v7 enabled devices. Thumb® on ARM®v6 processors omits access to VFP that in turn results in every floating point operation being executed by a significant slower system function. In contrast, Thumb® on ARM®v7 architectures does not have this limitation and is able to interact with VFP as well as with NEON®. [22, 28, 40, 41]

2.3. iOS® SDK Toolchain

2.3.1. XCode® 3.2

XCode® is the native IDE which comes with the official iOS® SDK. It is used to manage iOS® projects (.xcodeproj) for apps or static libraries targeting different architectures. Similar to most other modern IDEs, XCode® provides support for source code editing, building and debugging projects as well as automatic handling of cross-project-references and -dependencies. When building a project, XCode® is able to generate an universal binary containing optimized code for different specified architectures (e.g., ARM®v6 and ARM®v7). XCode® is directly linked with other tools like Interface Builder and Instruments®. [44]

iOS® apps can be started directly from XCode® and run either in iOS® Simulator or on a connected iOS® device. Thereby, iOS® Simulator simulates an execution environment for all iOS® devices. [44]

Three possible compiler options are shipped with iOS® SDK 4.3 to generate code for iOS® Simulator and iOS® devices. GNU Compiler Collection (GCC) 4.2 was the standard compiler until the release of XCode® 4. However, Low Level Virtual Machine (LLVM) [45] GCC 4.2 was already supported in XCode® 3 which uses the LLVM code generator that potentially increases performance. Also the most recent version of XCode® uses that option by default. On the other hand, it is possible to use the Clang 1.7 [46] compiler frontend in combination with the LLVM compiler backend, especially focusing on reduction of compile time and consumed memory. [44, 45, 46, 47]

2.3.2. Interface Builder

Interface Builder is a What You See Is What You Get (WYSIWYG) based design tool to organize user interfaces for iOS® apps. It allows to create, configure and arrange various kind of widget- and control objects. In this manner, it is also possible to connect particular events that are triggered by objects to event handling routines of other objects. The result is stored in the NextStep® Interface Builder format (`.nib`/`.xib`) to enable instancing actual objects at runtime. Basically, every iOS® project has at least one *main nib* file that contains the window- and application object. Further nib files could be also used to manage the appearance of specific elements of the user interface. [44]

2.3.3. Instruments®

Instruments® provides various tools to inspect the runtime behaviour and performance metrics of iOS® apps while running in iOS® Simulator or on real iOS® devices. In doing so, information about memory usage/leaks, Central Processing Unit (CPU) load, display update rates (frames per second) and many more could be gathered. [44]

2.4. Related iOS® Frameworks

"*A framework is a directory that contains a dynamic shared library and the resources [...] needed to support that library*" [23]. Frameworks are linked with an application in the same way as traditional shared libraries. Currently, apps may only link with frameworks of the native iOS® SDK.

This section will not discuss every single aspect of any mentioned framework but will explain selected core paradigms and aspects required to understand the development of the practical part of this thesis.

2.4.1. Objective-C® 2.0

Objective-C® 2.0 was introduced by Apple Inc. as an extension to Objective-C®. Basically, Objective-C® is a SmallTalk-80 derivation layered on top of the C programming language. Hence, Objective-C® code can be combined with plain C. All (object oriented) language features are implemented by the runtime environment which especially focus on dynamic typing and (very) late binding. In this manner, often *messages* are sent to objects where both are resolved at runtime. Amongst others, Objective-C® introduces a feature called *categories* to add new methods to existing classes without modifying these classes directly. Thereby, it is not necessary to have access to the original source code. It is also possible to mix C and C++ but this has several restrictions regarding amongst others, exception handling, class inheritance and -membership. Moreover, most of the iOS® top level frameworks require the use of Objective-C® 2.0. [48, 49]

2.4.2. Common Design Patterns

Several iOS® frameworks make intensive use of design patterns such as Model View Controller (MVC)-, delegate- and target/action pattern. [44, 50]

The delegate pattern involves a complex class that delegates certain active as well as passive tasks. Furthermore, a delegate that is an instance of a class that implements a particular interface. The delegating class uses the interface to communicate with the delegate. Usually the name of the interface is a combination of the term delegate and the name of the corresponding delegating class (e.g., `BCiSignalServerClient` and `BCiSignalServerClientDelegate`). Delegation is a method of customizing the behaviour of a class without creating an explicit subclass of that class. [44, 50]

Classes and methods may feature the target/action pattern to enable the execution of an action on a specific target in order to implement asynchronous program behaviour. In this manner, the action acts as a callback if a concurrent task has been accomplished or a certain event was observed. The action is represented by a *selector* (i.e. function pointer) and the associated target by an Objective-C® object that should recognize and execute the *selector*. [44, 50]

Some of the characteristics and applications of the MVC pattern are discussed in Section 2.4.4.2.

2.4.3. Foundation Framework

The Foundation framework introduces various basic functionality and paradigmas that are not covered by the Objective-C® language by itself like collections, string manipulation, date- and time manipulation, threads, run loops and many more. It also defines the root class `NSObject` which implements fundamental features like reference counting and Runtime-Time Type Information (RTTI). All Objective-C® classes related to the development part of this thesis derive from `NSObject`. Furthermore, the framework distincts between two types of objects that have either mutable or immutable contents. For example, `NSNumber` is a generic wrapper around any number whose objects cannot be changed once they have been created. Every class that offers the capability to modify the contents explicitly contains the term *Mutable* such as `NSMutableArray` as opposed to `NSArray`. In general, every class that shares the NS-prefix is part of the Foundation framework. [22, 23, 49, 51]

2.4.4. UIKit Framework

The UIKit framework constitutes the fundamental functionalities for every iOS® app. It has to take care of several key responsibilities including accomplishing the bulk of tasks in conjunction with initializing and managing the lifecycle of an application, interacting with and constructing the user interface, supporting "multitasking", accelerometer handling and much more. All classes in this thesis sharing the UI-prefix are contained in the UIKit framework. [23, 52]

2.4.4.1. Widgets

An iOS® app has at least one window (or `UIWindow`) and several views (or `UIViews`) in order to display content. In this manner, UIKit provides different kind of standard views like buttons, tables, scroll views and so on that may be altered in different ways (subclassing, changing drawing code, event handling). Every view can also have additional subviews managed by a view hierarchy and is responsible for arranging the layout adequately. Further, views are backed by a `CALayer` that referes to a Core Animation layer which is discussed in Section 2.4.7.1. Every view draws its contents to the backing layer. Directly manipulating the backing layer is more efficient and enables further options especially in respect to animations. [25, 26, 52, 53]

At a glance, UIKit renders widgets when the view is made visible the first time. Then, only a cached snapshot of the contents is drawn whenever the view has to be visualized

again (e.g., moving to a different position, switching visibility). Even if the bounds of the view have changed, the view would not always automatically be redrawn depending on the content mode property. In order to redraw the contents of a view extrinsicly, it is necessary to tell UIKit that the contents of the entire view or parts of it became invalid by calling the view's `setNeedsDisplay:` or `setNeedsDisplayInRect:` method. This might be useful if the data source has been updated. Then UIKit will invoke the drawing code in the *next event loop cycle* which is executed in `drawInRect:` that is implemented by means of the Core Graphics Framework (see Section 2.4.7). [25, 26, 52, 53]

Basically, two ways of applying a proper layout to a view and all of its subviews exist. Sometimes it is sufficient to set the auto-resizing behaviour property of a view that allows very simple automatic layout arrangements if the size of the view has changed. For example, specifying "flexible width" would resize the view by expanding or shrinking the width. This technique is not powerful enough to manage more complex layouts. For this purpose, if the visible area (or bounds) of a view has changed, UIKit will also ask that view to set its layout autonomously by calling `layoutSubviews`. A container view may override this method to set the size and origin of the bounds of the view and all contained subviews manually. The practical part of this thesis uses a mix of both variants and also a slightly different approach. [25, 26, 52, 53]

UIKit also has built-in support to animate view-related properties based on Core Animation like the center, alpha value, transformation matrix or background color. [25, 26, 52, 53, 54]

2.4.4.2. View Controllers

In simplified terms, at least one view controller (or `UIViewController`) is active that manages the current visible view hierarchy. Thereby, in the majority of cases, this view controller is initialized by a nib file. View controllers serve as the controller part in the MVC pattern. In doing so, they are also in charge of linking the application's data, the visual appearance and other controller objects. Additionally, every view controller is responsible to respond to low memory warnings by freeing unnecessary resources, or arranging the user interface if the physical orientation of the device has changed. [25, 26, 55, 52, 53]

2.4.4.3. Application Lifecycle

iOS® devices are intended for new and unconventional application areas that yield a different application lifecycle compared to traditional desktop applications. An app is able to move to five distinct states during the whole lifecycle. If an app has not been launched or was terminated by iOS®, the app is simply in state *not running*. The *active* state signals that the app runs in foreground and is receiving events. An app in *inactive* state still runs in foreground but does not receive any events. This can happen at any time, when an *active* app is transitioning to the *background* state, the user locks the device or if a phone call interrupts the execution of the app. Since iOS® 4.0, applications are enabled to perform limited tasks in *background* after the window of the app already has been dismissed by pressing the home button. An application is in *suspended* state and may be preferable purged by iOS® on low memory warnings after it moved to *background* and stopped executing any code. [22]

Assuming an app is *not running* and the user tapes the associated icon on the home screen, the system launches the corresponding app by calling the app's `main` function. Then UIKit loads the application's *main nib* file and initializes the main event loop by carrying out the `UIApplicationMain` function. While an app is in an appropriate state, particular events (touch events, motion events etc.) from the operating system are delivered to the main event loop and processed on every loop cycle. "*An event loop is simply a run loop: an event-processing loop for scheduling work and coordinating the receipt of events from various input sources attached to the run loop*" [56]. In this manner, various recipients respectivly views, view controllers and other class objects interact with the multi-touch display, accelerometer and others. [22, 56, 57]

The *main nib* file contains the application's main window and amongst others, a serialized concrete object of `UIApplication` and an object of a class conforming to the `UIApplicationDelegate` interface. The latter class is commonly application specific and provides the entry point of the first effectivly called custom code within an iOS® app. Before the main event loop is entered the first time, UIKit initiates this code by calling the proper method of the application delegate in order to perform individual initializing tasks, add specific views to the main window and show the main window including all of its subviews. In the same way, the delegate is also consulted to respond to other transitions of the application state. [22, 52, 56, 57]

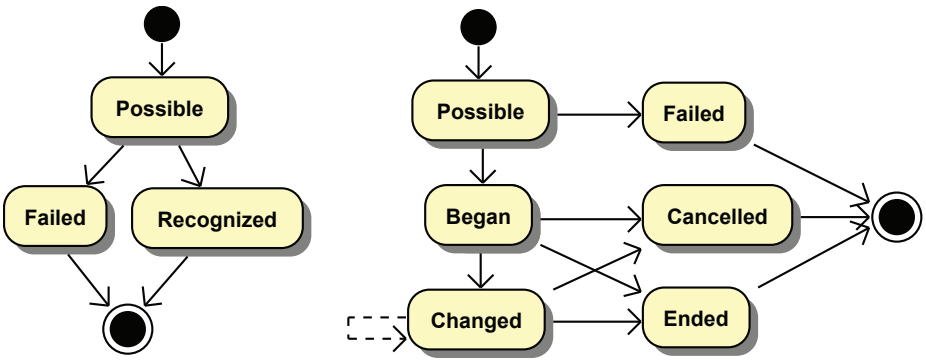
2.4.4.4. Recognizing Multi-Touches and Gestures

Principally, user interaction on iOS® is based on a multi-touch model. Simplified, iOS® keeps track of touches of one up to five fingers on the screen and encapsulates a multi-touch sequence in several event objects according to the spatial and temporal alternation of each involved touch. A multi-touch sequence starts when the first finger touches the screen and ends when the last finger is lifted. In many cases, certain multi-touch sequences or touches are interpreted together as gestures to initiate specific program behaviour, such as zooming by tracing a continuous pinch gesture. Also dragging a widget requires to evaluate the meaning of touches. In practice, smooth gestures should not rely on more than two fingers to guarantee seamless tracking. Event objects are sent to the active application's event queue for further processing within the next event loop cycle. When a touch event object is enqueued, UIKit typically dispatches the event handling to the main window. Every event object contains information about the position of the touch. This enables the main window to detect the top-most view or associated view controller in the view hierarchy that is hit by the touch. Then, this object is responsible of treating the event in the first instance. If the event was not handled, the event is handed over to the next responder in the responder chain along a particular path until the event is finally handled or discarded. These responders are especially views or view controllers. Every responder could analyze these multi-touches through dedicated methods. Some views like `UIScrollView` already evaluate these methods and capture gestures. In this manner, scroll views utilize swipe gestures to scroll inside a content area. In order to detect custom gestures or change the default touch event handling of responders, it is necessary to implement those touch event handling methods individually. [24, 52, 58]

Tracking and recognizing of very common gestures is already encapsulated by a couple of classes extending `UIGestureRecognizer` (see Table 2.4). In this context, it is also significant that gestures are classified as either discrete or continuous. Further, all gesture recognizers implement the target/action pattern. The action is executed if a discrete gesture is finally recognized which occures only once within a multi-touch sequence. In contrast, continuous gestures trigger the action multiple times on each state transition. Depending on discrete- or continuous gesture, the associated gesture recognizer follows one of the well defined state machines in Figure 2.2. [24, 52, 58]

Gesture recognizer	Gesture	Number of touches
UITapGestureRecognizer	Discrete touches	1 to n
UIPinchGestureRecognizer	Continuous pinching	2
UIRotationGestureRecognizer	Continuous circular motion	2
UISwipeGestureRecognizer	Discrete flicking	1 to n
UIPanGestureRecognizer	Continuous dragging	1 to n
UILongPressGestureRecognizer	Long continuous touches	1 to n

Table 2.4.: Gesture recognizers for common gestures



(a) Discrete gesture (b) Continuous gesture

Figure 2.2.: State machines: gesture recognizer

2.4.5. Message UI Framework

The Message UI framework offers view controllers for composing/queuing emails in the native iOS® email outbox. Similar view controllers are provided for writing Short Message Service (SMS). In both cases, the user has not to leave the app. [23]

2.4.6. Accelerate Framework

The Accelerate framework is intended for high performance applications. Since iOS® 4.0, this framework is also available for iOS® devices. It provides hardware accelerated functions including vector operations and complex algorithmns that are automatically optimized for every targeted iOS® architecture. The framework primarly consists of the vecLib which in turn contains a couple of header files for mathematical purposes. Mostly relevant is the vDSP library that is utilized by the practical part of this thesis. It provides implementations of various DSP tools like FFT, several window functions (Blackman-, Hamming-, von Hann windows), decimation and more. All functions are implemented in two ways, using vector instructions (i.e. NEON® etc.) and as scalar code. vDSP decides which version is appropriate depending on the present architecture. [23, 59]

The vDSP FFT Application Programming Interface (API) covers single- and double-precision 1D/2D-FFT algorithms but this thesis copes only with the real single-precision 1D-API that offers a derivation of a power-of-two algorithmn with different radixes. Before processing any data in the frequency domain, an array of twiddle factors has to be calculated that is required for later FFT API calls. Two parameters are expected. The first one is a radix option (radix 2, 3 and 5 are supported) and the second one is a base-two exponent that determines the largest number of elements that subsequential FFTs could process. Hence, FFTs with *different* transformation lengths that are equal or less than $2^{\log_2 n}$ (radix 2), $3 \cdot 2^{\log_2 n}$ (radix 3) and $5 \cdot 2^{\log_2 n}$ (radix 5) may share and reuse the same twiddle factors data structure for economical reasons. Anyhow, it is not recommanded to use a large sized twiddle factors array for FFTs that process a small number of data points to avoid performance loss. The 1D-API is able to calculate the forward as well as the inverse DFT on a given set of real input values. However, the actual results will be slightly different than calculated results by means of Formula 2.1 and Formula 2.2 to increase execution speed. Nevertheless, regaining the correct results only requires unapplying appropriate scaling factors from Formula 2.3 or Formula 2.4. Exploiting the symmetry property of the DFT enables the storage of the output *in-place*, using the same data structure for the output and input, thus requiring no additional

memory. The data structure has two members which represent the imaginary- and real components. Before processing real input data, data has to be packed into a special even-odd representation. In doing so, a given input vector $A = A[0], \ldots, A[n]$ has to be transformed to $A_{EvenOdd} = A[0], A[2], \ldots, A[n-1], A[1], A[3], \ldots, A[n]$ which leads to further optimization. The even elements of the input vector are stored in the real vector attribute and the odd elements in the vector attribute containing the imaginary parts. [59, 60]

$$\begin{aligned} X_m &= DFT(X_m) \cdot 2 \\ m &= [0, N-1] \end{aligned}$$

Formula 2.3: vDSP DFT scaling factor

$$\begin{aligned} x_n &= IDFT(x_n) \cdot N \\ n &= [0, N-1] \end{aligned}$$

Formula 2.4: vDSP inverse DFT scaling factor

2.4.7. Core Graphics Framework

2.4.7.1. Core Animation

Core Animation provides an Objective-C® API for rendering and animating graphical contents. These contents are displayed in Core Animation layers (or `CALayers`) that are organized in a hierarchy similar to UIKit's views (see Section 2.4.4.1). Layers are also using an on demand drawing engine. After the contents have been invalidated using `setNeedsDisplayInRect:` or `setNeedsDisplay`, drawing code is invoked on the next event loop cycle. Different ways exist of injecting custom drawing code. The development part of this thesis only uses overriding the layer's `drawInContext:` method. [54]

In contrast to views, layers are more light-weight and allow extended control over additional properties such as the corner radius of the frame. Furthermore, an implicit animation is applied whenever one of the layer's properties is manipulated. In this case, the property will be gradually interpolated for the configured global animation duration. Implicit animations could also be disabled and used only explicitly. [54]

The iOS® SDK documentation mistakenly claims the existence of a layout manager for arranging multiple layers. Unfortunately, that layout manager is only available in the OS

X® SDK. Hence, more sophisticated layouts have to be applied manually by overriding `layoutSublayers`. [54]

2.4.7.2. Drawing Graphics

Basically, contents of UIKit views and Core Animation layers are drawn by means of the C-based drawing API that Core Graphic provides. Thereby, a coordinate system is applied that does not refer to pixels rather to points in order to achieve an abstraction of the underlying device. This also leads to address coordinates with floating-point values. For example, although the display resolution of the iPhone® 4 amounts twice as much of the resolution of the iPhone® 3GS, both virtual coordinate systems will reflect an equal number of abstract points. The conversion from points to pixels is handled by the system when the graphical contents are rendered to the screen. [25, 26, 52, 53, 61]

The drawing API implements a painter model where a `CGContext` stores information about the destination device. It also allows to configure drawing parameters such as stroke- or fill color of the current drawing operation. In order to draw primitive shapes like lines, circles, rectangles, dedicated methods are already supported. When it comes down to more complex shapes, a path (or `CGMutablePath`) has to be considered. A path is constructed by combining a number of various shapes or path elements. [61]

Core Graphics also offers `CGLayer`, a graphic data structure, that is optimized for offscreen rendering. Thereby, iOS® tries to keep the corresponding contents in the fast video cache as long as possible. In this way, pre-computed graphical contents could be plotted to any destination at different positions very fast and numerous of times. [61]

2.5. Other related Tools, Frameworks and Libraries

2.5.1. Boost Libraries

Boost [62] provides free C++-libraries based on the C++ Standard Library for different purposes and runs on almost any modern operating system. All libraries are licensed under the Boost license. These libraries are either header-only (all functionality is visible to the compiler) or have to be explicitly linked with an application. In order to link these libraries, it is necessary to build them for a particular operating system first. Table 2.5 shows an overview about the most important Boost libraries related to this thesis. [62]

Library	Description	Header-only	Linked library
Boost.System	Error handling	No	Yes
Boost.Asio	Networking	Yes	No
Boost.Date_Time	Date and time operations	Yes	Optional

Table 2.5.: Related boost libraries

2.5.2. TinyXML++ Library

TinyXML++ [63] offers a free C++-interface to TinyXML that is a lightweight Extensible Markup Language (XML) parser. The parser incorporates few source- and header files and can be easily embedded into any application. It uses the Massachusetts Institute of Technology (MIT) license for distribution. [63, 64]

2.5.3. TiA Library

TiA [10] library offers an open source implementation of TiA written in C++ based on TinyXML++ and a couple of Boost libraries (see Table 2.5). From this thesis's point of view, only the client side interface that communicates with a specific server using various commands is interesting. According to the command versions that is supported by the server, these commands may vary a little. The client implementation (dated 17th of March 2011) understands version 0.2 and 1.0 command messages. [10, 65, 66, 67]

The control connection is established via TCP and operates according to a well-defined protocol. A client uses the control connection to send different types of commands to the server. Generally, the client is allowed to request the current server configuration and lists meta information about available biosignals. In addition, initiating/starting/stopping a data transmission request by using either an UDP- or TCP data connection is done by an adequate command. Then, data packets aggregate different types of biosignals that are delivered in specific intervals depending on the meta information of the present signals. Due to the unreliability of UDP, some data packets could get lost. However, in this case, it is possible to calculate potential data loss by tracking data packet sequence numbers at client side. [10, 65, 66, 67]

Every biosignal is described by a set of meta information which consists of the signal type, sampling rate, block size and a list of channel ids. The signal type is limited to a finite number of values (e.g., EEG, EMG, EOG, ECG, heart rate, blood pressure, button). Each data sample does not relate to one signal rather to one specific channel which is identified by the channel id. The source of the signal is sampled with the

sampling rate. Nevertheless, data is acquired in blocks. Hence, sampling rate and block size determine the virtual sampling rate $f_{s\ virtual} = \frac{Sampling\ Rate}{Block\ Size}$. In other words, every $\frac{1}{f_{s\ virtual}}$ seconds, a number of data samples equal to the block size can be acquired at once for a specific channel. If the sampling rate equals zero, data cannot be acquired in periodic cycles, rather aperiodicly and only when the value of the channel has changed (e.g., buttons). [10, 65, 66, 67]

The server configuration dictates the virtual master sampling rate that typically corresponds to at least one signal. All other signals are treated as slaves. Once a data connection has been established, in every master cycle, latest data is acquired in blocks from the master and all slaves (including aperiodic signals) if possible. Then, these data samples are packed into a data packet and sent to connected TCP clients or broadcasted by UDP (see Figure 2.3). [10, 65, 66, 67]

All relevant classes of the library are included in the `tobiss` namespace. The library provides `tobiss::TiAClient`, a client implementation which allows to connect to a TiA server, speaking any command message dialect. After a connection has been established, all mentioned commands may be sent to the server by calling appropriate methods of the API. Amongst others, the client blocks until the server answers the available signal types and meta information. The API also offers a convenient blocking method to receive the next data packet during an active data transmission. [10, 65, 66, 67]

2.5.4. SignalServer

SignalServer is an application implementing the server side of TiA. It is able to acquire raw biosignals from various hardware devices (e.g., g.USBamp, g.Mobilab, g.BSamp, BrainProducts Brainamp series, generic joysticks, LifeTool IntegraMouse, generic mouses) and redistributes obtained data over the network using TiA library. Version 0.2 as well as version 1.0 TiA command messages are supported. Further, the server restricts some master/slave configurations that would lead to aliasing by ensuring the property $f_{s\ virtual\ (master)} >= f_{s\ virtual\ (slave)}$. The practical part of this thesis uses SignalServer to test against TiA server interface. For this purpose, SignalServer provides a configurable sine generator and EEG simulator. Nevertheless, the deployment scenario could also be a real BCI measurment. This document refers to a version of SignalServer that was updated on the 17th of March 2011. [12, 67]

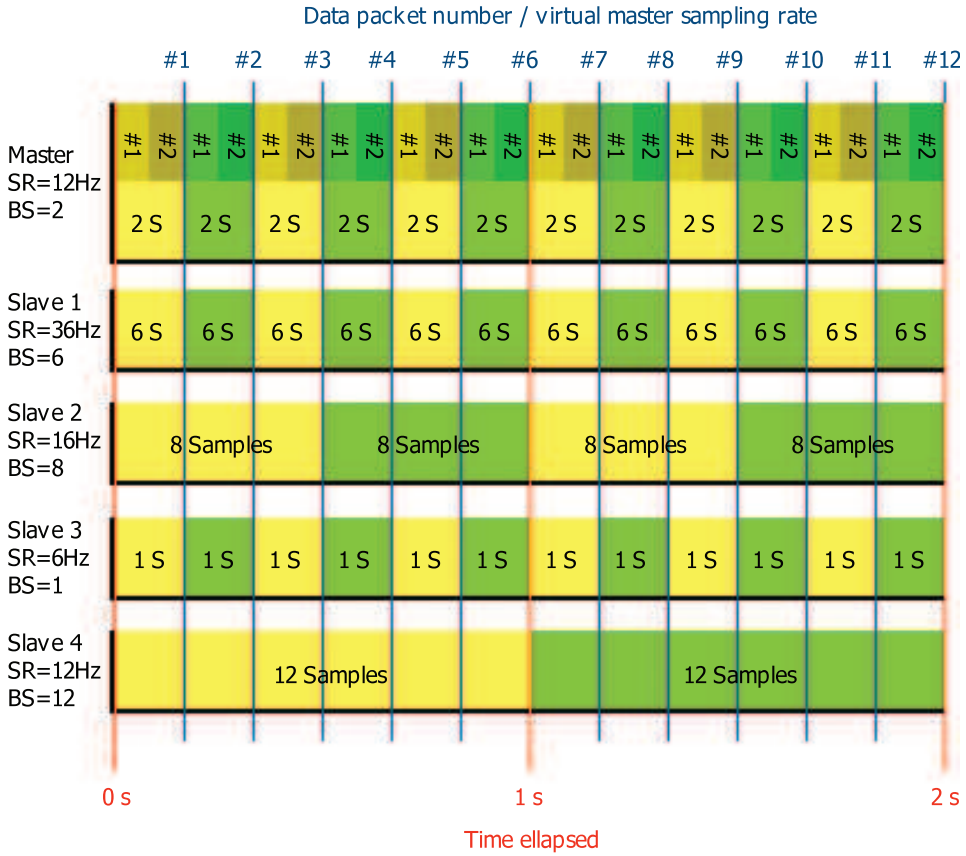


Figure 2.3.: TiA server: master/slave timing diagram - shows 12 data packets; assumes every master and slave has one channel, a sampling rate (SR) and a block size (BS) - #1 with samples of Master, Slave_1 and Slave_3, #2 with samples of Master, Slave_1 and Slave_3, #3 with samples of Master, Slave_1, Slave_2 and Slave_3 ... #12 with samples of Master, Slave_1, Slave_2, Slave_3 and Slave_4

2.5.5. Three 20 Library

Three 20 is an open source and widely used Objective-C® library under the Apache License that provides amongst others, general purpose views and view controllers. One very popular example imitates the behaviour of the native iOS® app launcher application. [68]

2.5.6. CorePlot Library

CorePlot is a popular library available under the new BSD license that provides an Objective-C® 2D visualisation API. It contains implementations of fancy and easy to use graph views (e.g., line chart) and more. [69]

2.6. Software Licensing

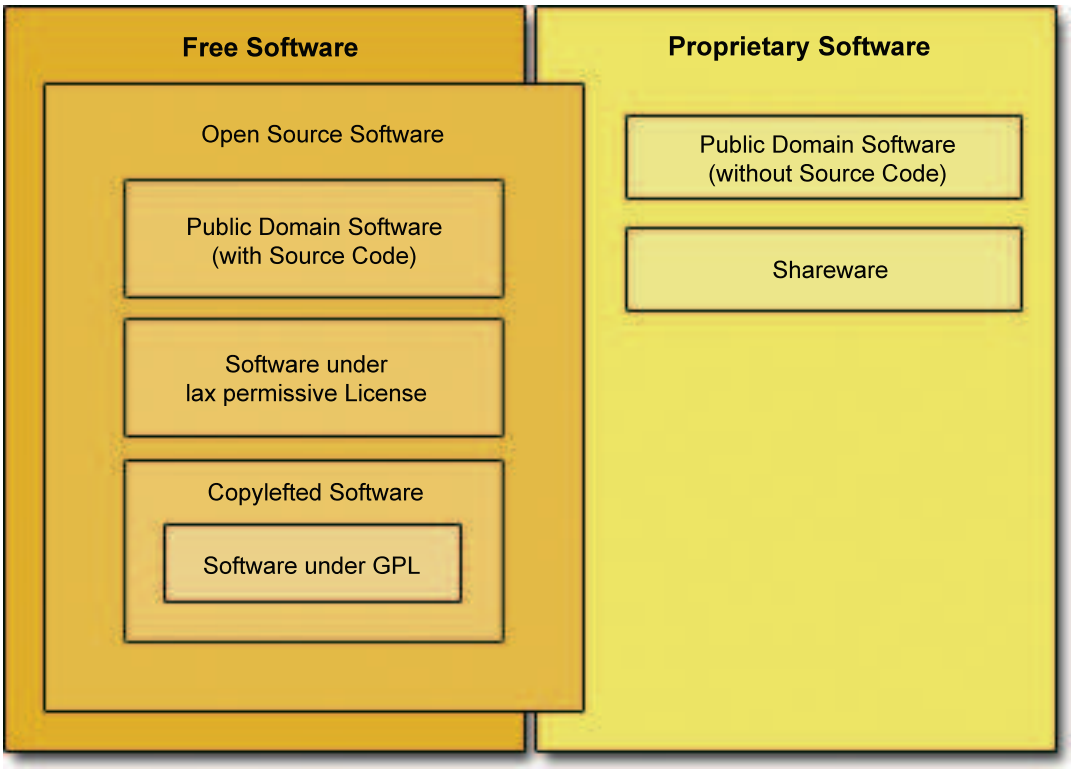


Figure 2.4.: Software license categories [70]

Figure 2.4 gives an overview about different license categories and how they are overlapping each other but those mentioned below are most relevant in this thesis [70].

Free software refers to freedom and not to price. In other words, every recipient of the software has the *"permission to use, copy, and/or distribute, either verbatim or with modifications, either gratis or for a fee"* [70]. Therefore, the availability of the source code is mandatory.

Open source software describes a class of software that is similar but more restrictive and less idealogical driven than free software [70]. Essentially, the software distribution terms have to comply with a couple of criteria. Free redistribution is also required as the disclosure of the source code, the authorization of derived works, the maintenance of the author's integrity, the prohibition of discrimination against persons/groups and fields of endeavor and the direct distribution of the license. Furthermore, the *license* must not be specific to a product, nor restrict other software and has to be technology-neutral [71].

Public domain software is derived from the legal term *not copyrighted*. Copyrights have to be explicitly disclaimed to move a software in the public domain [70].

Software under lax permissive license virtually allows any usage of the source code (e.g., distribution of proprietary binaries). Amongst others, the X11 license (sometimes mistakable mentioned as MIT license) and the two BSD licenses fall in this category [70, 72].

Copylefted software is protected by a copyleft clause. Copyleft is a general concept to ensure that all copies of all versions of the software are licensed under almost equal distribution terms. Consequently, two distinct copylefted licenses are typically mutually exclusive. Software covered by the GNU General Public License (GPL) is just one single case of copylefted software [70].

Proprietary software is the opposite of free software in general. One of the main differences is that the copy right holder of the software may categorical dismiss the distribution of the source code together with the binary. Even in the case of semi-proprietary software, where the source code is also delivered, the original copy right holder will always dictate the terms and conditions under which all subsequent license recipients are allowed to use, modify or redistribute the software [70, 73].

3. Implementation Aspects

3.1. Requirements

3.1.1. Non-functional Requirements

As already introduced in Section 1.3, an application (or app) named iScope has to be developed that acts as a client for SignalServer in order to allow mobile, live quality monitoring during biosignal measurements. Deployment platforms shall be iOS® with version 4.0 and higher. Primarily, iScope has to run on at least iPhone® 3GS due to device availability reasons. Nevertheless, iPod touch® 3G compatibility should be considered. Furthermore, it is a requisite that iScope also runs on successors of iPhone® 3GS as well as on iPad® devices of the first generation and higher. No minimum biosignal data transmission rate was agreed but limitations of iScope have to be tested. Downsampling of received biosignals was explicitly ignored because it will be implemented in SignalServer natively in the near future.

3.1.2. Functional Requirements

First of all, the user shall be able to manually enter the address of a SignalServer or select a SignalServer from a list of previous server sessions and then initiate a connection attempt. If connection has been successfully established, the user is presented an overview of all available biosignal channels hosted by the SignalServer. Then, the user (un)selects an arbitrary number of biosignal channels one after another or (un)selects all channels of a particular type at once. After the selection has been confirmed, live visualization of the selected channels starts. Basically, every channel is visualized by means of a line graph and a spectrogram. The optical appearance of the line graph should be similar to the native iOS® stock market app. The spectrogram utilizes FFT to calculate spectra with a fixed size time window of one second. In contrast, the window function and number of FFT cycles per second should be configurable. No minimum amount of cycles per second was agreed but limits have to be tested. A color coded representation of spectra

shall be used to observe spectra changes over a period of time. Both, line graph as well as spectrogram shall be able to operate in two modes. Data is either visualized at the right edge of the graph and the content area is continuously scrolling, or data is drawn similar to various medical devices at the position of a thin vertical bar that runs from the left to the right edge of the screen cyclically. The number of simultanously visible graphs shall vary on the orientation of the device and the type of device. If the device is held in portrait orientation, more graphs could be placed among each other than in landscape orientation. Furthermore, graphs may be arranged in two columns on iPad® devices whereas only one column is reasonable for iPhone® or iPod touch® devices. Additionally, line graphs and spectrograms shall be placed on two different pages but the user may decide to change the position of graphs in a similar way to the native iOS® app launcher application on the iOS® home screen. This includes reordering of graphs on the same page as well as on different pages. Scrolling between pages and within one page has to stop at integral graph positions in order to prevent partially hidden graphs. It shall be possible to magnify graphs to fit the entire screen size that reveals additional explanation or information (e.g., minimum/maximum values). The observation period is linked to a specific number of seconds. However, the user shall be able to continuously zoom the time window but the zoom should snap in when integral seconds are reached. In addition, the value domain (minimum/maximum represented value) shall be scalable as well. While graphs are visualized, the user shall be able to stop and restart data transmission in order to (un)freeze graphs at any time. The content of the screen shall be captured by a screenshot on demand. Then, the user can decide whether the screenshot is stored on the device or sent by email. iScope shall add a seperate entry in iOS® Settings app to allow configuration of parameters that are applied when the app is started:

- Layout of graphs by specifying the number of visible columns and rows for portrait- and landscape orientation.
- Maximum observation period in seconds.
- General flag to disable spectrograms.
- Window function used by the STFT (rectangular, Hamming, Hann normalized/de-normalized, Blackman, Bartlett).
- Flag that indicates if the output of the FFT is given in dB.
- Maximum amount of FFT cycles per second.

- Flag enabling a simulation mode that allows to demonstrate iScope's capabilities without connecting to a real SignalServer.

On the other hand, iScope shall also feature a dedicated settings screen to configure parameters that are applied immediately if they are changed. These are the visualization mode for all graphs which can either be cyclic or scrolling as already mentioned, or the scaling mode of line graphs and spectrograms separately. Thereby, scaling shall either be fixed by configuring boundaries manually or automatically by detecting maximum- and minimum values online and adjust the visualization accordingly. Finally, every use case shall be controlled by common or extended iOS® gestures (e.g., pinch gesture for scaling or zooming, double tap for magnification, shaking the device for starting/stopping data transmission).

3.2. General Design Decisions

iScope especially sticks to the following performance related principles:

- Locking and synchronization of threads is reduced to a minimum and preferable replaced by other techniques like run loops. For instance, all classes implementing the singleton design pattern have to be initialized explicitly to avoid synchronization overhead [74].
- Blocking the main thread makes the application less responsive and is avoided if it is possible.
- Graphical contents are redrawn only if they have effectivly changed.
- Implicit Core Animation animations are generally disabled.
- According to Section 2.2.3, single-precision data types and functions are used over their double-precision counterparts if the loss of precision is arguable in order to gain a possible performance boost. In this manner, also frequently used results of floating-point operations are cached if a huge amount of redundant computations can be saved.

Various design patterns from Section 2.4.2 are also adopted in all (sub) projects.

C++ classes are wrapped with Objective-C® classes if it was necessary to fully utilize iOS® SDK framework features (e.g., dispatching a selector with an object parameter to

the main run loop).

Finally, all classes implemented in the course of iScope share the BCi-prefix and are derived from `NSObject`. However, class diagrams will omit the `NSObject` inheritence relation due to shortage of space.

3.3. System Architecture

3.3.1. System Overview

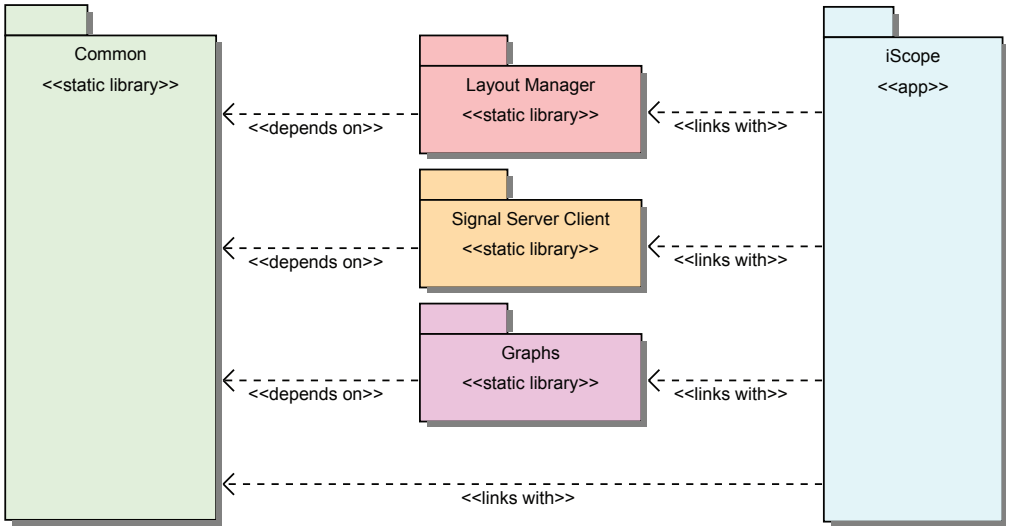


Figure 3.1.: Main system modules overview

Four main modules established as seperate buildable Objective-C static libraries compose the basis for the final app. Static libraries are facilitated due to the lack of iOS® to support individual frameworks. Other external dynamic linked libraries have to be avoided as well. Figure 3.1 illustrates how these modules depend on and interact with each other. Decoupling functionality in independent modules faciliates several aspects:

- Resuability of specific modules in new or existing projects that also improves the chance of discovering possible problems by potentially keeping the codebase permanently alive.
- Flexibility and interchangeability of modules by another version or different implementation of the module.
- Sole development and testing of every module reduces the time to detect problems, failures and bottlnecks which also increases the maintainability.

- Encapsulation of algorithmns and data structures hides the internals of a module and enforces clear and easy to use module interfaces that are less error-prone and raising the testability.

Basically, every module uses functionality from the *Common* module. Although all required header files have to be already present when the module compiles, the *Common* module library is not actually linked with any other produced module library. Doing so would end up in various "duplicate symbols" linker errors when linking different module libraries (e.g., `libbcigraphs.a` and `libbcilayoutmanager.a`) against the same binary. This forces iScope or any other app (e.g., unit test apps) using one of the four main modules to explictly link the *Common* module library with the binary, even though no functionality of that module is referenced directly. The corresponding XCode® projects already take care of that aspect by adding cross-project references for all necessary modules and associating the library files to executable targets. In the same way, every module is managed by an individual XCode® project and may use additional sub modules as well. Table 3.1 shows the mapping between main modules, related XCode® projects and resulting static libraries.

Module	XCode® project (`.xcodeproj`)	Static library (`.a`)
Common	`LibBCiCommon`	`libbcicommon`
Graphs	`LibBCiGraphs`	`libbcigraphs`
Layout Manager	`LibBCiLayoutManager`	`libbcilayoutmanager`
Signal Server Client	`LibBCiSignalServerClient`	`libbcisignalserverclient`
iScope	`BCiScope`	

Table 3.1.: Main system modules: XCode® projects and binaries

3.3.2. Automatic Testing

Every main module is responsible of managing its own unit test target(s). Therby, whitebox- as well as blackbox tests are implemented. For this purpose, GHUnit framework is utilized because it offers more readable/formatted output, a custom GUI-based runner and more test macros than OCUnit/SenTestingKit framework. In conjunction with XCode® 3.2 and below, the main benefit of GHUnit is the ability to debug test methods directly on the device. However, as XCode® 4 has been released and some advantages became obsolete, the test framework should be exchanged in the future. In doing so, GHUnit test classes can be easily integrated in OCUnit test classes. [75, 76]

3.4. Environment

3.4.1. Platform

At the time iScope was initiated, the preferred IDE for iOS® platforms was XCode® 3.2. For this reason, iScope was developed with this particular version of XCode®. Hence, the term XCode® always refers to version 3.2. Nevertheless, all (sub) projects in the context of iScope can be built with XCode® 4 due its downward compatibility with XCode® 3 projects.

iScope will only run on iOS® versions 4 and above resulting from the dependency on the vDSP library which was introduced with iOS® SDK 4.0 (see Section 2.4.6). This limitation is acceptable compared to the availability of a fast signal processing API that is highly optimized for iOS® devices. Unfortunately, iPhone® 1G/2G devices and iPod touch® 1G devices do not support iOS® 4. However, iPhone® 3GS devices which are also the minimum iOS® devices that should be supported is capable of installing iOS® 4.

The proejct is built as an univesal app targeting ARM®v6- as well as ARM®v7 architectures. Furthermore, the app can run on any iOS® device, disregarding if it features a retina display or has the dimensions of a phone or pad.

3.4.2. Compiler and Linker

In general, iScope uses LLVM GCC 4.3 to generate all of its binaries. This is a consequence of Section 2.3.1 that explains several potential advantages.

In order to ensure that Thumb compiler flag is only specified for ARM®v7 architectures, conditional build settings have to be created inside the build settings section in every XCode® project. Avoiding Thumb for ARM®v6 architectures should prevent all negative impacts discussed in Section 2.2.3.

In this context, it is also indispensable to explicitly pass a couple of specific flags to the linker in order to avoid linker errors and abnormal program behaviour:

- *ObjC* causes the linker to load all object files containing Objective-C classes or categories. Enabling this flag is crucial in conjunction with static libraries (e.g., `libbcicommon.a`, `libbcigraphs.a` etc.) that define categories of preexisting classes. The crux is that the Objective-C compiler generates unresolved linker symbols only for each utilized class and not for every single adopted method. Consequently, the linker will not include object files or code of categories in the

executable if associated classes have already been resolved. Although compiling and linking will finish successfully because of the dynamic nature of Objective-C, without setting this flag, the program will very likely be terminated by an "selector not recognized" exception thrown by the runtime system. Specifically, this will happen if a particular method defined in any of those categories was called during program execution.

- *all_load* or *force_load* are addressing a bug in the Objective-C linker that prevents the above mentioned linker flag from loading object files containing categories in static libraries under specific circumstances (compiler versions and target architectures). Accordingly, passing only *ObjC* would also lead to the same runtime exception as if it was never specified. *all_load* simply solves this issue by loading all object files from all static libraries disregarding if these libraries effectivly contain Objective-C code. *force_load* is more flexible by allowing to specify only those static libraries that should be considered by this practice. In this case, the output file will grow by comprising code that will never be used. Alternativly, adding a dummy class in the context of every of these categories would also be effective without having to accept the mentioned downside. Of course, the latter method would even make the need of *ObjC* obsolete, but should be considered as a workaround rather than a real solution. Apple recommends one of the former ways which are also applied to iScope as long as the bug remains.

- *lstdc++* links the binary against the GNU's Not UNIX (GNU) C++ Standard Library (libstdc++). This flag is fundamental since libstdc++ is not automatically linked if only other involved static libraries but the project itself does not cover any Objective-C++ files. Consequently, skipping *lstdc++* would induce many "undefined symbols" linker errors when building iScope.

These individual linker flags are appended to other linker flags parameter of specific (sub) projects or single targets.

3.5. Common Module

This module provides generic functionality used in most modules and sub modules of iScope and depends on no additional library or (sub) module. The corresponding XCode project file is located in `trunk/src/LibBCiCommon`. Table 3.2 shows a list of all contained buildable targets.

Target	Description
`LibBCiCommon`	Static library
`Tests`	Unit Tests
`BCiCommonBenchmarks`	Ring buffer benchmark
`BCiCommonDemo`	Common examples

Table 3.2.: Deployment targets: Common module

3.5.1. Managing and Passing Data Samples

In order to process data generated by different kind of modules transparently, a unified data access interface for data containers is aspired. `BCiReadableDataBuffer` represents this kind of interface. Section 3.5.1.1 will discuss different types of data providing classes used in iScope that implement this interface directly.

In many cases, a generic interface for random data access like `BCiReadableDataBuffer` may be sufficient, but sometimes it is also required to impose the internal structure of traversing a data container. For this purpose, an additional interface according to the iterator design pattern is designed that is explained in Section 3.5.1.2 [74]. Figure 3.2 shows how `BCiReadableDataBuffer`, and `BCiDataIterator` work together and also gives an overview over all classes implementing these interfaces.

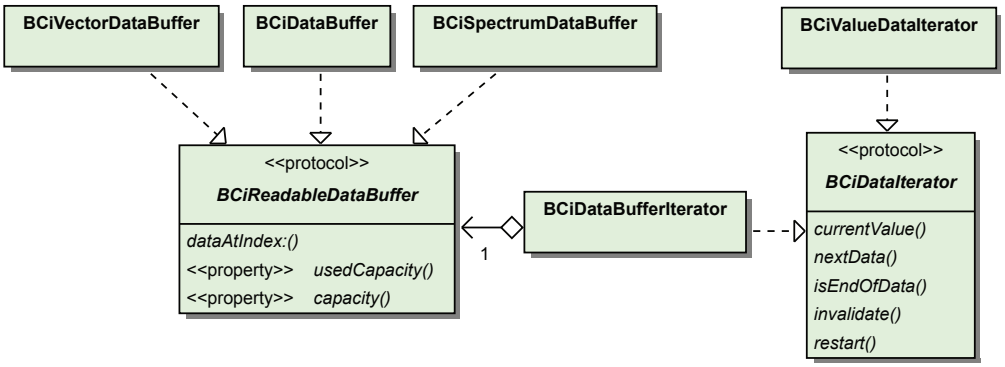


Figure 3.2.: Class diagram: fundamental data containers

3.5.1.1. Readable Data Buffers

Each module might have a data representation that is optimized for a certain use case (e.g., storing power spectra requires a different memory layout than keeping a vector of continuous data samples in the memory). When passing data to another module, explicit and possible expensive data conversion is no longer necessary if the data class already

implements `BCiReadableDataBuffer`. Moreover, changing the underlying data structure of a module will no more effect other modules as well. Amongst other positive effects, this increases the testability by demanding the called API to be tested only against the public interface rather than all implementing classes.

This technique is heavily used to communicate between *Graphs* module (Section 3.8) and *Signal Server Client* module (Section 3.6) but also between other (sub) modules, thus being a fundamental part of decoupling modules from each other and the modular architecture of iScope.

Ring Buffer

When envisioning core tasks of iScope, it is important to have access to a data structure that stores data samples from various signals and ensure the following qualities at the same time:

- Fast insert operations at the end of a sequential data collection.
- Fast random access to any element and traversal of the collection.
- Efficient use of memory constraint to a fixed number of simultanously buffered data samples. At any time, only data within a given interval has to be available while the number of data per interval remains constant.
- Fast single-precision floating-point C-array conversion or representation of arbitrary data sequences to tollfreely utilize vDSP function calls.

All mentioned objectives are met by a ring buffer (also sometimes referred as circular buffer). Inserting data at the beginning or at the end has an asymptotic runtime complexity of $\mathcal{O}(1)$ as well as accessing elements at random. Copying elements to a specific destination induces a complexity of $\mathcal{O}(n)$ that also describes its memory costs, although memory might be allocated only once the ring buffer was initially created. Then, subsequent insert operations would benefit from reusing an existing memory slot. A ring buffer is not optimized for inserting elements at random positions but this is no prerequisite. It is sufficient to use a fixed capacity and override elements prioritized by the first-in, first-out method if the maximium number of elements has been reached. [77]

When looking for a base container class for a ring buffer implementation, Foundation framework provides `NSMutableArray` as the only reasonable native Objective-C collection that affords adding, deleting and accessing data while preserving the original order of

contained data elements. While promoting a comprehensive interface, sufficient to design a ring buffer, would commend the application of `NSMutableArray`, there is also one aspect that disqualifies this approach. `NSMutableArray` is not designed to manage elements of primitive data types like `float`. Instead, in order to enable `NSMutableArray` as a container for basic data samples, each value has to be wrapped by an Objective-C object first. This characteristic has some significant impacts:

- Utilizing vDSP API of the Accelerate framework to process data samples requires expensive C-array conversation.
- Memory overhead of keeping an extra Objective-C wrapper object for every data sample.
- Necessity of implementing an additional wrapper class. Unfortunately, `NSNumber` is immutable which would lead to allocate an object every time a data sample is put into the array.

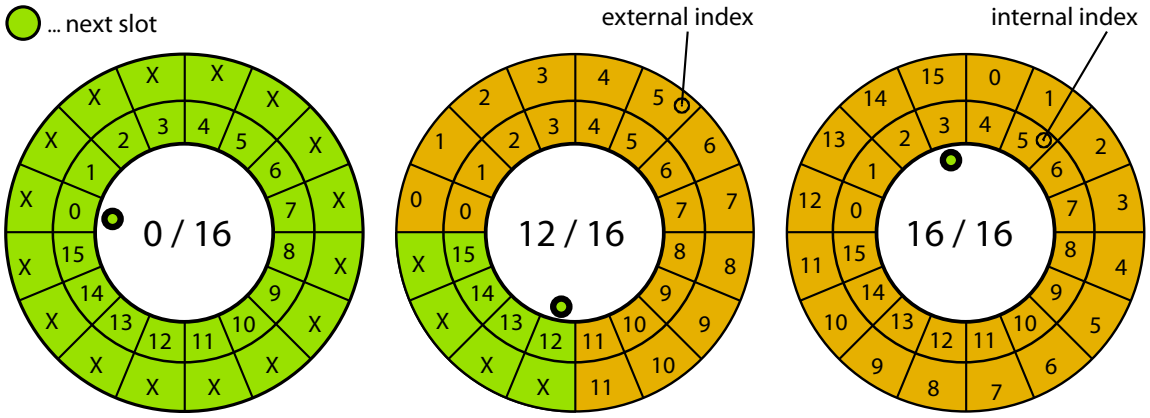
Making use of the `std::deque` class (double ended queue) that is shipped with libstd++ represents another way of constructing a ring buffer without relying on any other external framework or library. In contrast to `NSMutableArray`, `std::deque` does not have the drawback of demanding on a wrapper class for storing primitive data values. Anyway, it has still one major disadvantage in common, the need of copying each element in a C-array before data samples can be passed to one of the vDSP functions.

Of course, there are other C++ libraries offering ring buffer implementations like the popular Boost library (referred as circular buffer). Unfortunately, Boost ring buffer does not already handle *direct* C-array conversion of *arbitrary* buffer index ranges automatically. Consequently, this behaviour has to be done by a seperate wrapper or controller class and will lead to an additional implementation effort anyway. On the other hand, fundamental concepts should not be relient on an external library if possible. These are the reasons why Boost library is not considered for this purpose.

Concluding, `BCiDataBuffer` implements a simple custom ring buffer to fit the requirements and avoid all negative aspects that were discussed above. Internally, a fixed size C-array is used to represent the data and store elements up to a maximum capacity simultaneously. As long as the used capacity does not exceed the maximum capacity, the n-th element of the buffer corresponds to the n-th element of the C-array. Once an element is inserted into an already full buffer, the first element in the buffer will be

overwritten. Consequently, a mapping from the external buffer index to the internal C-array index is necessary whenever an element is accessed. Then, the last element of the buffer differs from the last element of the C-array and corresponds to the first element in the C-array instead. In this way, no further memory has to be allocated on any insert operation. Figure 3.3 demonstrates a buffer in all possible states: empty state, a state that does not require index mapping and a full buffer.

`BCiCommonBenchmarks` compares `BCiDataBuffer` with rudimentary Objective-C(++)



(a) Empty (b) After 12 insert operations (c) After 19 insert operations and full

Figure 3.3.: Ring buffer with a maximum capacity of 16 elements

ring buffer implementations based on `NSMutableArray` and `std::deque` by means of three expected core tasks. See Section A.1 for more information about the tasks and benchmark results. Although all compared techniques have similar asymptotic time complexities for adding, traversing and copying data to a C-array, only the latter task is significantly performed exclusivly faster in the case of `BCiDataBuffer`. This results from the internal C-array representation of the data. Thereby, copying is permitted by performing `memcpy` at worst twice instead of accessing and copying every element of the array individually. `memcpy` is a standard C-function that copies blocks of memory to a destination memory address typically very fast. Performing the traversing task with `NSMutableArray` in combination with fast enumeration and `std::deque` with Standard Template Library (STL) iterator equals the performance of stepping through all elements with `BCiDataBuffer`. `NSMutableArray` seems also to be distinctly behind all other competitors in running the adding task due to the wrapper class overhead.

Spectrum Buffer

`BCiSpectrumDataBuffer` provides an abstraction of vDSP in-place real FFT results which also conforms to `BCiReadableDataBuffer`. Every instance may be configured as returning either amplitudes or power values. Then, subsequent calls of the random access API of `BCiReadableDataBuffer` will compute the associated method implicitly. In addition, another flag specifies whether or not the logarithm should be applied on spectra values. The logarithm is useful to suppress statistical outlier in the biosignal (e.g., as a result of noise) and is calculated with $10 \cdot log_{10}(Magic\ Number + Spectral\ Value)$ where $Magic\ Number$ is a constant and set to 1 in order to prevent passing zero as an argument to the logarithm which would lead to undefined results. However, adding $Magic\ Number$ will not alleviate the desired effect. Due to the special data packing of the vDSP API, the external indices have to be mapped to internal indices every time an element is accessed.

3.5.1.2. Data Iterators

The `BCiDataIterator` interface provides methods for traversing a subset of a data collection including sequential forward data access, testing if the current element is already the last element of the subset, invalidating and restarting the current traversal. Displaying data samples was originally initiated by passing `BCiReadableDataBuffer` conforming objects to different views from view controllers in the *Graphs* module. `BCiDataIterator` replaced this approach by exposing only a subset of all data samples while not changing the underlying data container. This makes the direct index driven data access obsolete and facilitates looser coupling between different modules and/or submodules. In this context, the most valuable benefit is the easy realization of polymorphic traversal.

3.5.2. Generic User Interface Elements

In the course of iScope it was also necessary to create custom user interface elements for generic purposes:

- `BCiTableSectionHeaderView`, a multi-clickable table header view with a color gradient background and a blue arrow button. The background also acts as a button. If the arrow button is tapped, it will rotate by 90°. Every further tap will alternate the sign of the rotation. Taps on both buttons could be customized by the target/action pattern.

- `BCiNavBarTextField`, a textfield especially designed for the use in the navigation bar because native textfields could not be used in this case.
- `BCiSlider`, a slider that will directly reflect its value in a linked textfield.

3.5.3. Detecting extended Pinch Gestures

`UIPinchGestureRecognizer` from Table 2.4 represents the starting point in the course of permitting simple pinch gesture recognition. While up to two fingers are touching an associated hit-test view, the gesture recognizer remains in *Possible* state and tracks all related touches. As finger movement exceeds a threshold, pinching is detected. Then, the gesture recognizer turns from *Possible* state to *Began* state and sends the action to the configured target for the first time. Optionally, but repeatedly if one of the fingers moves while the other finger remains still pressed, the gesture recognizer will change to *Changed* state and sends an additional action. Once the last finger is lifted, *Ended* state is adopted and the action is delivered for the last time. This perfectly fits the needs of iScope but in order to determine if the current pinch was performed in horizontal or vertical direction and inwardly or rather outwardly, `UIPinchGestureRecognizer` has to be extended by the custom `BCiPinchGestureRecognizer` class. On any transition to *Began*- and *Changed* state, `BCiPinchGestureRecognizer` determines the category of the current pinch by calculating the angle (α) between the current coordinates of the two touches. The gesture is classified either as a *horizontal inward pinch*, *horizontal outward pinch*, *vertical inward pinch* or *vertical outward pinch*. If α is less than a given border angle, the pinch is horizontal justified. Values of α that are beyond the border angle will be considered as vertical. If not specified, the border angle is set to 45°. Inward pinches involve touches moving toward each other, as opposed to outward pinches requiring touches that move apart. Figure 3.4 illustrates all possible pinching categories as well as an extra blind area which is the result of the blind angle property that spans an area where no valid categorization is possible. Further, every pinch gesture has a current scale property that reflects the alternation of the most recent touches relative to the original position. This value starts with a value of 1.0. Consequently, during the gesture, more distant original touches will lead to a finer grained control of the scale factor than touches that were not far apart initially. The scale factor is resetted every time the category of the gesture has changed or a valid mapping to one of the categories has become impossible. [52]

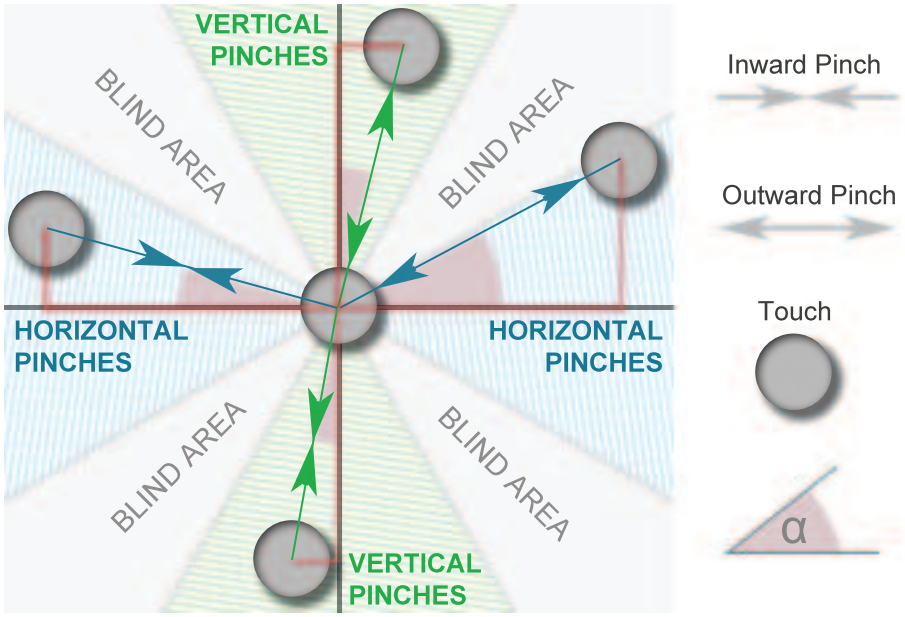


Figure 3.4.: Detecting extended pinch gestures with BCiPinchGestureRecognizer

3.5.4. Spectrum Computation

The spectra of a given real input vector (representing a discrete time signal) are computed with an instance of `BCiSpectrumController` that is based on the vDSP API. Encapsulating the transformation to the spectral domain makes it easy to tune or fully replace the interanlly used FFT implementation or behaviour without changing code in any other source file. As a consequence, the usage of the vDSP API is not visible outside `BCiSpectrumController`. Currently, the FFT is calculated with a real in-place power-of-two algorithmn and the radix 2 option. Because iScope processes signals with different sample rates but calculates the STFT in constant intervals, also twiddle factor arrays with distinct sizes are involved. For the first time an instance of `BCiSpectrum-Controller` is forced to compute a complex spectrum and no suitable array of twiddle factors is present, it will be generated. Then, it remains accessible for *all* instances if an equal sized spectral domain transform is requested.

Basically, the number of samples, an input vector containing these samples and the type of the window function have to be specified in order to compute the spectra. `BCi-SpectrumController` will automatically take care, if the size of the given input vector is not a power of two by padding up to the next power of two with zeros. Rectangular-, Blackman-, Hamming-, von Hann- and Bartlett windows are available and cached for subsequent applications. The spectral results are packed into an instance of `BCiSpectrum-`

DataBuffer which already cut off the redundant half of the output due to the symmetry of the spectrum.

3.5.5. Device Categorization

At certain times, it is significant to have knowledge about the underlying iOS® device. For example, if an algorithmn should be optimized for the computational power or the screen resolution/dimension of a particular device. This is handled by a custom category of UIKit's UIDevice which is BCiUIDevice_Utiliy.

3.6. Signal Server Client Module

The *Signal Server Client* module handles communication with TiA server interface by including parts of TiA library as an independent sub module (see Section 3.6.2). That sub module in turn depends on TinyXML++ library and two Boost libraries (Boost.System, Boost.Date_Time) which requires to have all of these libaries compiled for iOS®. For this purpose, TinyXML++ is built as a static library managed by a seperate XCode® project (see Section 3.6.1). The Boost libraries are built as described in Section A.2.

3.6.1. Building TinyXML++ Library for iOS®

Building TinyXML++ for iOS® is done by LibTicpp.xcodeproj which is an individual XCode® project located in trunk/src/external/LibTicpp. It produces the static library file libticpp.a. A seperate XCode® project is required due to the fact that the shipped Makefile does not consider builds with the iOS® SDK. When building the project, also the TIXML_USE_TICPP preprocessor macro is defined that guids to use the C++ version of the library. LibTicpp is the only XCode® target that finally outputs the static library.

3.6.2. Building TiA Library for iOS®

Basically, the TiA library contains a Qt [78] project file TOBI_SignalServer_tialib.pro that generates Makefiles by using qmake [79] to build the library after all. Unfortunately, the configuration file is not designed to build an iOS® compatible binary. Furthermore, it introduces additional external dependencies (e.g., Lpt tools) that are not required in the case of iScope. The direct conversion from .pro-file to XCode®

project by means of `qmake` will not fix the problem and requires manual adaptation of several project settings anyhow. Hence, `LibTOBISignalServerClient.xcodeproj`, an individual XCode® project is created in `trunk/src/extern/LibTOBISignalServer-Client` that only comprises sources and headers utilized by iScope. In this way, unused server functionality is discarded. The project defines two buildable targets, `LibTOBISignalServerClient` and `TOBISignalServerClientDemo`. The former one produces the static library `libtobisignalserverclient.a` and the latter one builds an app that demonstrates only fundamental capabilities.

When linking against `libtobissignalserverclient.a`, also `libticpp.a`, `libboost_-system.a` and `libboost_date_time.a` have to be linked. Although amongst others, Boost.Asio is referenced by TiA client implementation, there is no need to link an additional binary for a header-only library. In the future, an approach without an explicit XCode® project should be considered similar to treating Boost libraries.

3.6.3. Signal Server Client Library

In fact only the interface of this library is used when iScope internally interacts with *Signal Server Client* module. The library is managed by an XCode® project located in `trunk/src/LibBCiSignalServerClient`. Various buildable targets are maintained by the project which are listed in Table 3.3. The core of the libray makes use of TiA library client implementation that also contains C++ code. Due to the dynamic development character of TiA library (e.g., changes of the communication protocol, client) when iScope was initiated, the library was encapsulated by the adapter design pattern [74]. This was the main reason of designing an individual library. Another main intention was to be able to communicate with TiA server interface while providing model- and controller classes to obtain received data based on interfaces that rely on Objective-C. This approach should facilitate Cocoa Touch and Foundation framework features.

From iScope's point of view, no additional libraries have to be regarded because `libtobisignalserverclient.a` already links with `libticpp.a`, `libboost_system.a` and `libboost_date_time.a`.

The implementation of the *Signal Server Client* library is based on extra properties of SignalServer which are derived from several observations. Once these properties change (or are fixed) in the future, then the library has to be adapted adequately. First of all, the *virtual* sampling rate of signals have to be greater or equal 1 Hz ($f_{s\ virtual} < 1$), otherwise the server would not have been started succesfully. Phase shifting between

Target	Description
`LibBCiSignalServerClient`	Static library
`Tests`	Unit tests
`BCiSignalServerClientDemo`	GUI driven demo client illustrating basic data transmission capabilities

Table 3.3.: Deployment targets: Signal Server Client module

master and slaves never ever occurs with the exception of pseudo phase shifting that results from different block sizes. Data packets are serially numbered starting with 1. The block size and sampling rate are supposed to be static attributes of a signal that do not change during a server session. Latest slave data is fetched in periodic master cycles disregarding if data has been ready in a prior master cycle that potentially leads to undersampling and aliasing. For example, assuming a configuration that includes a master with *Sampling Rate* = 12 Hz, *Block Size* = 2 and a slave with *Sampling Rate* = 9 Hz, *Block Size* = 4. Then, the server calculates the *Ratio* of virtual master sampling rate and virtual slave sampling rate to determine the interval which is used to send slave data blocks. It appears, this interval is always an integral number. This can be observed at the server after starting with the mentioned configuration by noting the printed *Ratio* which amounts 3. The *Ratio* is calculated by rounding up $\frac{f_{s\ virtual\ (master)}}{f_{s\ virtual\ (slave)}} = \frac{6}{2.25} = 2.67 \sim 3$. Consequently, only every third data packet will contain a block of slave data which would not be sufficient. If 6 master data packets are received per second by the client, this results in mere $\frac{f_{s\ virtual\ (master)}}{Ratio} = \frac{6}{3} = 2$ slave data blocks per second instead of correct 2.25 data blocks per second. The sampling rate of the slave would be effectively 8 Hz in place of 9 Hz which causes omitted data samples at the client.

Figure 3.5 gives a rough overview about the structure of the *Signal Server Client* library. Basically, the public API provides two main interfaces, `BCiSignalServerClient` and `BCiSignalDataModel` to communicate with the server and retrieve transmitted data. Both always represent a pair that belongs together where `BCiSignalServerClient` controls the server by sending appropriate commands and `BCiSignalDataModel` reflects the server results (e.g., data, meta information) in response of these commands. In this manner, instances of `BCiChannelBuffer` which contain the actual data samples are retrieved through `BCiSignalDataModel` but cannot be added or removed with the public interface. Even though the library is multi-threaded internally, no explicit and expensive thread synchronization has to be taken into account when accessing data samples through the data model.

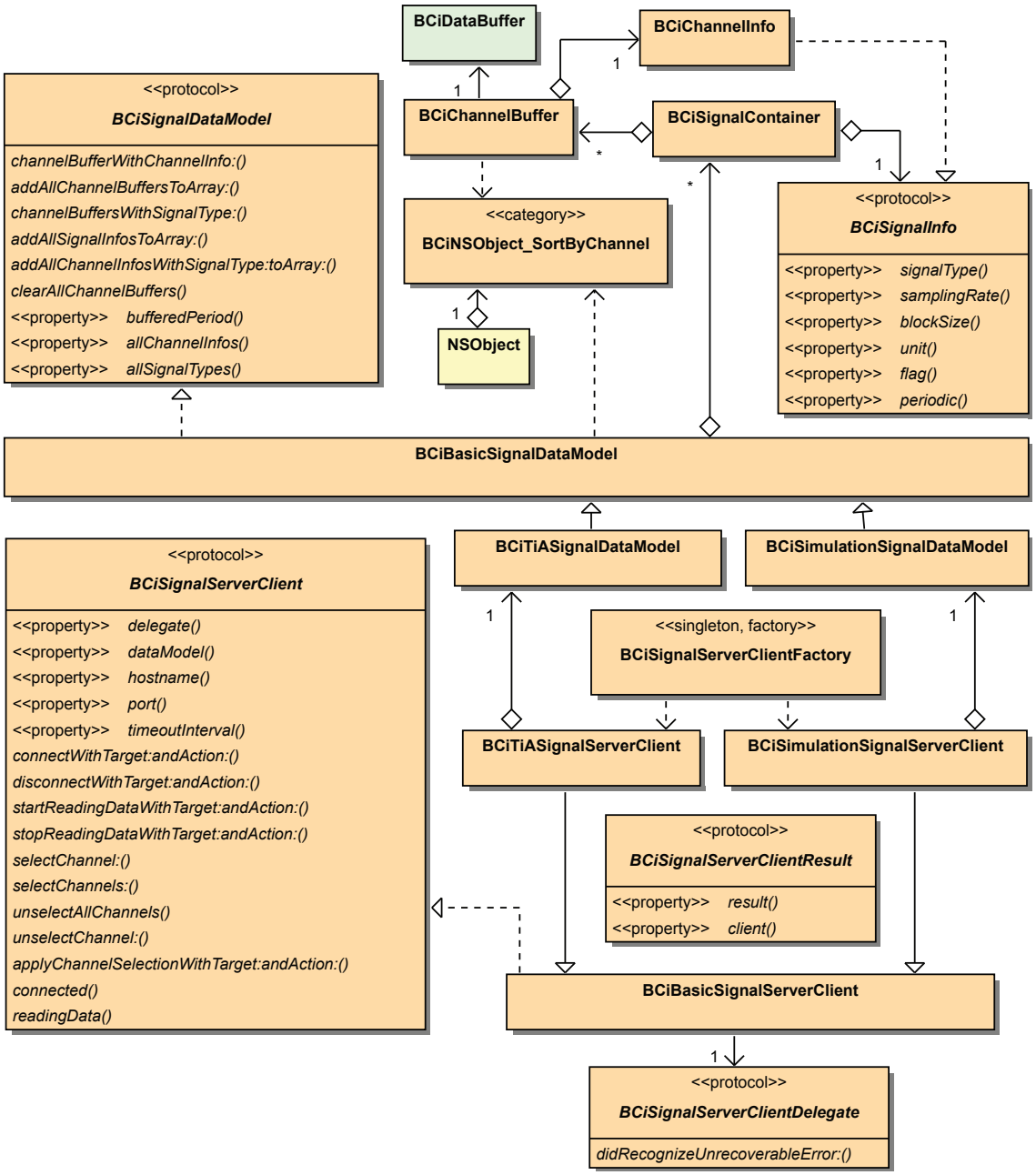


Figure 3.5.: Class diagram: Signal Server Client overview

Additionally, the library implements the factory method design pattern [74] to mask the implementing class of the `BCiSignalServerClient` interface (i.e. TiA client) beyond the scope of the library. In this course, `BCiSignalServerClientFactory` is the only way to instantiate a concrete client. This behaviour allows to exchange the client implementation without effecting any other source file other than the file containing the factory method. For example, in this way a client implementation that does not depend on TiA-, Boost- and TinyXML++ libraries could be easily introduced in the future. Two types of clients can be created, a client for a real SignalServer and a client that only simulates a server. Although two distinct factory methods exist, both clients implement the same `BCiSignalServerClient` interface that makes it simple to write an application which interchanges these two clients. Currently, the factory method referring to a real server transparently returns an instance of `BCiTiASignalServerClient` that utilizes the existing TiA library client. On the other hand, the factory method for server simulation returns an instance of `BCiSimulationSignalServerClient` which emulates a fully-fledged server with a specific configuration.

`BCiChannelBuffer` acquires the already discussed ring buffer `BCiDataBuffer` to store data samples within a defined period. Currently, the last 10 seconds are recorded for every periodic signal channel which may be adjusted by changing the corresponding property of `BCiBasicSignalDataModel`. In constrast, only the current value of an aperiodic signal is revealed. It was a project requirement to use UDP instead of TCP when communicating with the TiA server. When loosing data packets, dedicated sentinel values are added to the underlying ring buffer to preserve the possibility to determine if a specific sample was either received or interpolated. This constant value is defined as FLT_MAX which lies off the expected value range of biosignals. In this case, comparing two float values for equality is feasible because the values were initialized one with another. Alternativly, every data sample could be represented by a custom structure or class with an attribute that signals whether or not the sample was interpolated. However, the sentinel value strategy faciliates performance, has no memory overhead and requires no additional implementation effort. When starting a new data transmission, the data model is cleared and all data samples from prior transmissions are considered as data loss.

Every `BCiSignalServerClient` implements the delegate pattern to report abnormal behaviour. In this case, the current client should be discarded and replaced by a new one.

Some of the methods of the public API involve sending commands to the server. How-

ever, using the client should not block the calling (main) thread until the task specific communication has been finished in order to keep the application responsive. For this reason, all of these methods will return right away after the communication has been initiated and make use of the target/action pattern. Then, the given target is notified about completion by calling the associated action on the main thread. Only one method at once could be handled this way.

Until now, whenever SignalServer transmits data, data samples of every signal and channel are sent to the client, even if only one particular channel is desired. According to [67], it is planned that the client/server protocol will be extended by a mechanism to select those channels that should be included in data transmissions. Utilizing this feature would significantly reduce the data packet size (i.e. the amount of network traffic) and thus, the time to receive and parse all interesting samples. The public API of the *Signal Server Client* library already provides this feature which in turn is used by iScope.

Due to the stated extra properties relating rational virtual sampling rates, handling data loss will only correlate with SignalServer in conjunction with pure integral virtual sampling rates.

3.6.3.1. Real SignalServer Client

Figure 3.6 demonstrates the interaction of the most important classes by the *Real SignalServer Client*.

It is important to explicitly set the UDP receive buffer size of the underlying socket for

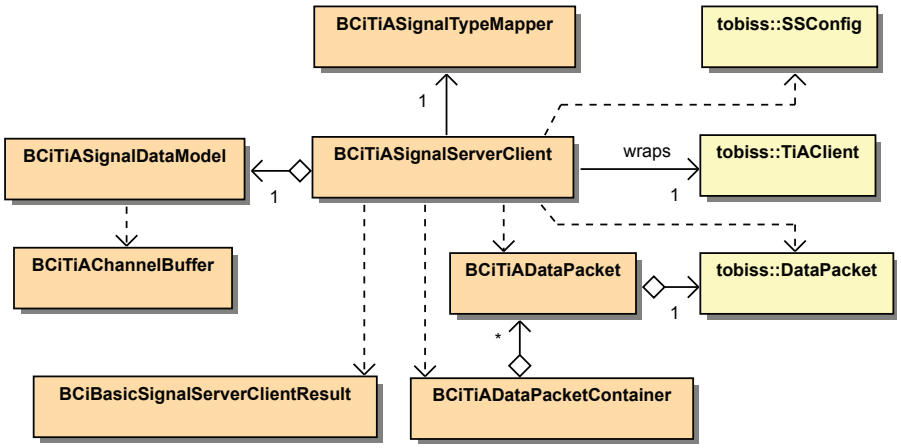


Figure 3.6.: Class diagram: TiA client adapter

Mac OS X® and iOS® to an adequate value (300.000 bytes). The default value of TiA

library client would be to high and causes a program crash or at least unpredictable behaviour.

All blocking TiA library client methods (e.g., connecting, disconnecting) have to be performed in background threads, otherwise the main thread would be locked until the execution of all incorporated methods has finished. This is indispensible because TiA library client does not support a configurable timout. All API calls would not break before an undefined time has expired. A proper synchronization technique is demanded to ensure only one of these methods is performed in a separate thread at the same time. This is done by means of locks and condition variables.

When connecting to a server, primary TiA library client is configured to use 0.2 commands. If the server does not speak the 0.2 dialect, the client fails to connect. Only under these circumstances a connection relying on 1.0 commands is attempted to be established. Of course, this also implies a longer time until the connection is finally set up by having to await the timeout of connecting with 0.2 commands in the first place. At the time, TiA library clients using 0.2 commands are preferred because they seem to be more reliable and stable than their 1.0 counterparts that tend to show odd behaviour.

Starting a data transmission spawns a new long term thread that is assigned to receive new data packets through TiA library API repeatedly until the data transmission is explicitly stopped or the client breaks unintentionally. Synchronizing the reader thread with the main thread when calling the *Signal Server Client* API is also done with appropriate locks and condition variables. This necessity results from upholding the consistency of TiA library client by prohibiting changes of the client state while the client is not finished with an assigned task (e.g., reading a data packet). In this context, also a keep alive timer is implemented that checks in regular intervals if the read data packet loop is still frequently iterated or is blocked for any reason. This occurs if the connection to SignalServer is aborted while TiA library client is already waiting for the next data packet. In this case, the delegate is notified about an unrecoverable error because TiA library client can be neither recycled nor released. Basically, this involves testing if a number of data packets has been read since the last time the timer fired. Further, these checks will have no significant impact on performance due to a reasonable $Timer\ Interval = Master\ Block\ Size + Constant$ (seconds). This process also takes care of thread synchronization.

As anticipated, iOS® is not able to adopt any suggested thread priorities which supersedes improving the performance of the client by tuning the scheduling priority of the reader

thread.

Finding an efficient way to pass data samples received by the reader thread to the data model and access these samples through the data model from the main thread was one of the key challenges when designing `BCiTiASignalServerClient`. Instead of explictly protecting the data model with a mutex whenever adding or accessing data, better performance is attained by performing both operations in the same thread exclusivly. After the reader thread received a data packet, a selector is dispatched to the main event loop that is in charge of extracting data samples from the data packet, handling possible data loss and filling the data model. Finally, the selector is executed in the next main event loop cycle. Due to the fact that only instances of `NSObject` may be passed to selectors, data packets have to be wrapped by an appropriate class which is constituted by `BCiDataPacket`. In order to prevent excessive selector dispatching in conjunction with high virtual master sampling rates, `BCiDataPacketContainers` are dispatched instead that aggregate an amount of `BCiDataPacket` objects. This amount is directly linked to the expected number of data packets per second which limits the number of dispatched selectors to a configurable maximum.

After all, a mechanism to detect and compensate data loss is also implemented in *Signal Server Client* library. Data loss may occur due to overwriting unread data packets in the socket receive buffer if the reader thread cannot handle the virtual master sampling rate. On the other hand, data packets could simply get lost on the way to the client on the network during an UDP data transmission. The data model keeps track of the data packet sequence number of the most recent received data packet and sample per channel. If the difference of the data packet number of an incoming data packet and the most recent data packet number exceeds 1, then packet loss was detected. This might effect one or more channels. Then, all possibly affected channels are marked indicating further handling is required. The effective amount of lost data samples is discovered the next time a sample is added to one of these channels. For this purpose, for every marked channel, the ratio between virtual master sampling rate and virtual sampling rate of the channel (slave) is used to calculate the effective lost amount. In doing so, the current data sequence number of the channel is compared with the former data sequence number (see Formular 3.1). In this process, a optionally phase shifiting would have also been considered.

$$
\begin{aligned}
Ratio &= \frac{f_{s\ virtual\ (master)}}{f_{s\ virtual\ (slave)}} \\
Amount &= \left(\left\lfloor \frac{Data\ Packet\ Number_{incoming}}{Ratio} \right\rfloor - 1\right) \cdot Block\ Size_{slave} - Sequence\ Number_{slave}
\end{aligned}
$$

Formula 3.1: Calculating the amount of lost data samples at the client

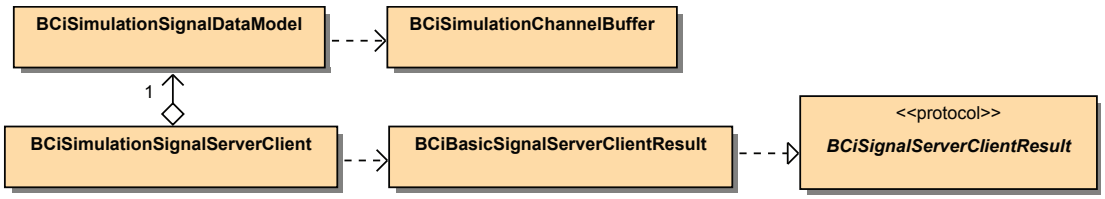


Figure 3.7.: Class diagram: simulated server client

3.6.3.2. Simulated Server Client

Figure 3.7 gives an overview about the *Simulated Server Client* class environment.

At present, periodic EEG signals (see Table 3.4) up to 10 channels and one aperiodic signal of type button with a single channel are generated. In spite of the aperiodic character of the button, it will randomly change its value every 1/3 seconds. In this context, a virtual sampling rate constitutes rather an upper boundary than an effective guideline for the data generation interval of a simulated signal. In fact, the interval is directed by the accuracy of a timer that is configured to fire according to the virtual sampling rate of an associated signal which does not have to correspond to the virtual sampling rate of a master. Additionally, after a specific number ($= Sampling\ Rate \cdot Block\ Size + 1$) of data samples was generated, the following ten data blocks will be discarded and considered as data loss. Unfortunately, the timer does not necessarily achieve the configured frequency depending on the current system load. Moreover, iOS® is no real time system.

Sampling rate	Block size	Virtual sampling rate
32 Hz	1	32 Hz
64 Hz	2	32 Hz
128 Hz	4	32 Hz
512 Hz	8	32 Hz

Table 3.4.: Available simulated periodic signals

Performance Limits

In an ad-hoc Institute of Electrical and Electronics Engineers (IEEE) 802.11n network environment, *Real SignalServer Client* was able to receive data packets at a maximum sampling rate of 700 Hz over UDP and using 0.2 command messages, if only one signal channel was transmitted and data samples were not extracted. Furthermore, if data samples were additionally parsed from data packets, a maximum sampling rate of 650 Hz was observed. Transmitting ten and selecting one up to three channels showed that a sampling rate of 512 Hz could be achieved no matter if data samples were extracted. In every case, an increase of the sampling rate ended up in a loss of data packets of 80% and higher. See Section A.3 for more detailed results.

3.7. Layout Manager Module

Generally, this module defines a common reusable interface (`BCiLayoutManager`) to manage the layout of a set of layout items whereas existing methods of arranging layouts are not sufficient. The layout manager decides to show only a subset of all items at the same time. Thus, an interface to notify about changes of the currently visible items through the delegate pattern (`BCiLayoutManagerDelegate`) is also provided. Items could be instances of any class that conforms to the `BCiLayoutItem` interface. Hence, layout managers do not rely on views directly, rather on custom controller classes that administer dedicated content views. If requested, particular items can be magnified to fit the maximum available screen space. Further, every layout manager also features an edit mode that enables the user to change the layout manually. In the majority of cases, user interaction is handled outside of this module. However, as the layout manager installs itself as the delegate (`BCiLayoutItemDelegate`) of all added items, it is asked to allow and respond to specific user interactions (e.g., dragging, magnification).

Of course, the *Layout Manager* module also hosts a reference implementation of a grid layout manager (or `BCiGridPageLayoutManager`) that fulfills the needs of iScope. In this context, items are placed on a configurable amount of pages. In turn, every page is divided into columns and rows where the number of simultanously visible items per screen remains alterable during program execution. Thereby, the number of possible items per page is virtually unlimited. Navigating through/within pages in order to change the set of visible items is also supported as well as reordering and (de-)magnification of items. In this process, also discreet and familiar animations are applied by means of UIKit to increase usability. All animation durations are configured with seperate constant values

that may be easily modified.

Basically, some aspects of the described behaviour are similar to the well-known native app launcher application of iOS® devices. The Three 20 library already contains an implementation of a view (`TTLauncherView`) adopting that characteristic. Anyhow, it is limited in keeping page dimensions constraint to the screen size which results in an upper bound for the number of items per page and disqualifies this approach. In order to guarantee a virtual inifite number of items per page and items with an adequate size at the same time, a method to switch the visible items on a page like scrolling is required. A table view (`UITableView`) or a similar view would have been reasonable to represent a column on a page because they already have this feature. Nevertheless, this would also complicate the exchange of items in different columns and/or on different pages. Consequently, a new approach is implemented in the course of `BCiGridPageLayoutManager` that is illustrated in Figure 3.8.

The XCode® project file is located in `trunk/src/LibBCiLayoutManager`. It contains

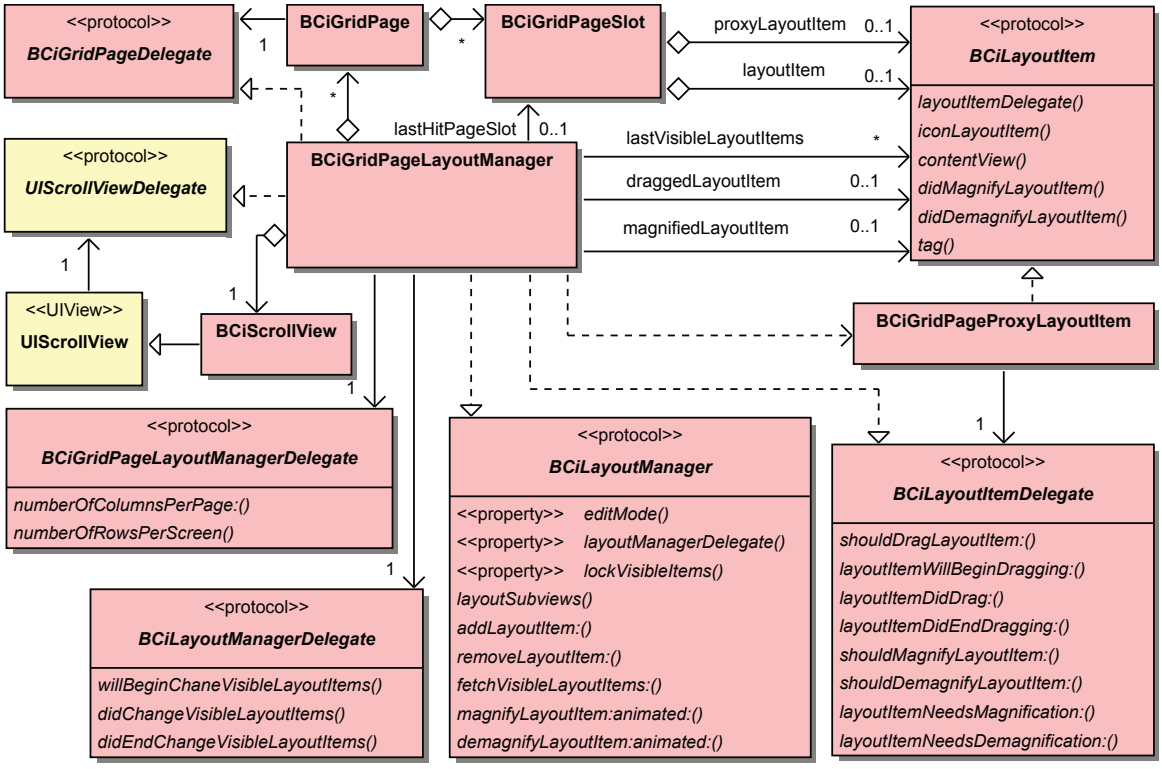


Figure 3.8.: Class diagram: Layout Manager overview

various buildable targets that are listed in Table 3.5.

Target	Description
`LibBCiLayoutManager`	Static library
`Tests`	Unit tests
`BCiLayoutManagerDemo`	GUI driven demo app showing the use of a grid page layout manager

Table 3.5.: Deployment targets: Layout Manager module

3.7.1. Adding Items

When an item is added to a grid layout manager, it is assigned to the next empty slot that belongs to a particular page. If no empty slot was found, new slots will be created. These slots are special cells of a table which is virtually formed by the grid page layout. Only cells with a corresponding slot are actually visible. A slot can be either empty or occupied by a layout item. Anyhow, also empty slots maintain a proxy item that acts as a place holder for a concrete layout item. Proxy items also implement `BCiLayoutItem` and are displayed as empty boxes with a thick round frame. For practical reasons, it is invariant that each page hosts at least one row that could contain occupied as well as empty slots. Further, the layout manager appends new slots line-by-line to ensure that only entire rows are visible. Initially, one row containing empty slots is added to every page. At any time, a totally empty trailing row is detected which does not exist solely on its page, the row is trimmed to avoid a continuing increase of redundant rows. These rows could be a product of explictly removed items or changes of the current layout which also effects the number of items per page.

`BCiLayoutManagerDemo` uses very simple layout items which have different background colors and display a label containing a unique name for each item (see Figure 3.9(a), Figure 3.11(a)).

3.7.2. Applying Layout

In this matter, utilizing Interface Builder would demand the creation of static layouts (i.e. nib files) for every combination of various layout parameters. First of all, this includes the number of rows and columns per page. Furthermore, landscape- and portrait perspectives have to be treated differently. After all, iOS® devices vary in screen resolution that forces to handle the layout separately as well. Therefore, the layout is computed at runtime because a static approach would not be practical.

A slightly modified customized scroll view serves as a container for all items added to a grid layout manager affording a theoretical infinite content area. The scroll view is transparently installed as a subview of a view that is specified at the time the layout manager is created. This view is directly provided by an app and embedded in a layout that is defined outside of this module. Therefore, it also defines the maximum content area for the layout manager and all of its layout items. In this process, also the autoresizing mask of the container view is configured (using "flexible width" and "flexible height") to guarantee that all allocated space is automatically exploited if the bounds of the specified superview change. This might occur if the app switches from portrait- to landscape perspective. At this point, according to Section 2.4.4.1, custom layout code would have been implemented in `layoutSubviews` but scroll views have to be treated in a different way. `layoutSubviews` gets called whenever the bounds of the view have changed. This is also the case, even if the scroll view scrolls content. Consequently, applying the layout in `layoutSubviews` would end in an unacceptable overhead of redundant layout arrangements. Fortunately, also the frame property is automatically adjusted by UIKit that calls `setFrame:` at the time when the autoresizing behaviour is applied. After all, overriding `setFrame:` of a custom scroll view that invokes the layout manager satisfies the requirements.

The grid layout manager implements an on demand layout service that is applied only if the size of the content area has actually changed. In this case, an additional delegate (`BCiGridPageLayoutManagerDelegate`) is asked to answer the maximum number of visible columns and rows per page. Based on the gained parameters the bounding box of every page slot is calculated. After all, the layout of corresponding items is arranged by setting the frames of all associated content views. Then, the content mode of the content view decides how to respond to changes of the bounding box (e.g., redraw contents) but this does not belong to the scope of this module.

The appliance of layouts is also mastered while scroll view is still scrolling or edit mode is activated. For this reason, it is necessary to track and harmonize all incorporated animations with the layout manager.

Changing the content area is implemented in `BCiLayoutManagerDemo` and illustrated in Figure 3.9 and Figure 3.11. In this demo, the delegate of the layout manager returns different values for the number of columns and rows per page depending on the present user interface perspective. In portrait mode, two columns and three rows will be returned by the delegate. On the other hand, the layout manager will obtain two rows and three

columns in landscape mode.

3.7.3. Changing visible Items

Basic scroll views already offer the capability to continuously scroll a content area in all directions by means of gestures. In addition, a paging property can be specified to allow the content being scrolled only by an offset that is equal to the width (horizontal paging) or height (vertical paging) of one page. The visible area of one page is determined by the frame of the scroll view. iScope requires discrete paging in horizontal direction and vertical continuous scrolling within a page. Unfortunately, paging cannot be set for different directions independently. Also switching the property after the direction of the current gesture has been detected would no more effect the current scrolling activity. Hence, paging of the scroll view is deactivated and implemented manually. In this manner, `BCiGridPageLayoutManager` conforms to `UIScrollViewDelegate` and responds to changes of the current offset of the scroll view. After the scroll view stops scrolling and the resulting visible screen area does not show entire pages and items, the offset of the next integral page and item is calculated. In this case, the scroll view starts to scroll again until the offset finally fits.

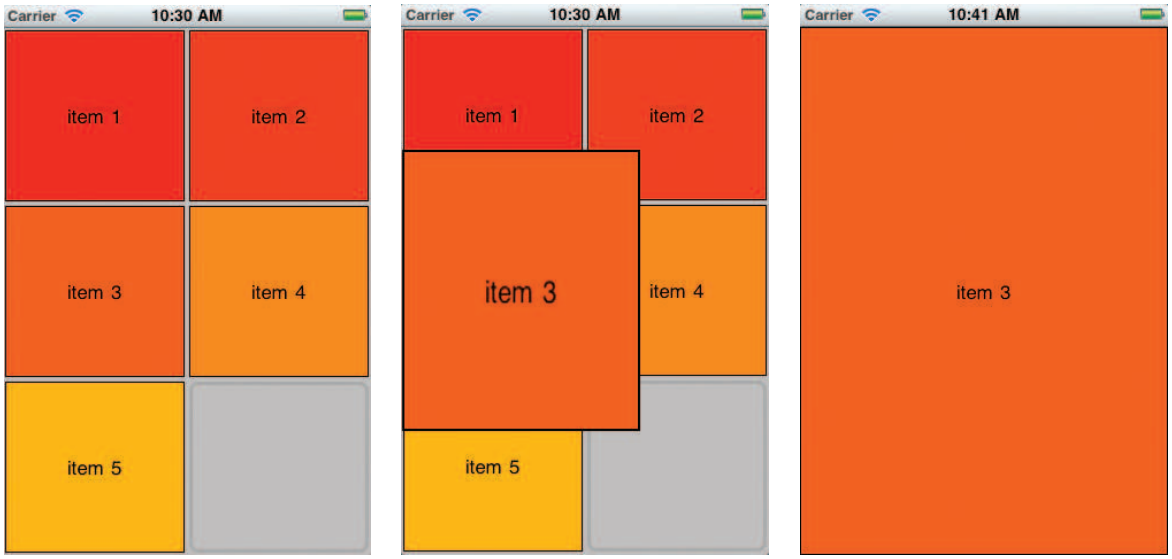
In general, changing visible items is achieved by scrolling or paging as well as (de-)magnification or reordering of items. Pages are turned by horizontal swipe- or pan gestures. Scrolling within a page requires similar but vertical finger movements. Although diagonal scrolling cannot be programmatically avoided, the grid layout manager will adjust the visible content area to show always integral pages and items after all animations have been stopped. It is also mentionable that (de-)magnifying and reordering items could influence the set of visible items as well.

Changing the set of visible items could also occur at any time, even if a prior scrolling activity has not completed requiring further synchronization effort.

3.7.4. (De-)magnifying Items

Implementing the trigger (i.e. user interaction) to start (de-)magnification of items is left to the concrete `BCiLayoutItem` classes. However, the layout manager is asked through the `BCiLayoutItemDelegate` protocol to allow (de-)magnification and is responsible of how to (de-)magnify items. Only if the layout manager is not in edit mode, the content area does not scroll, no item has already been magnified and no pending other actions

exist, magnification will be possible. Then, magnification involves the animated gradient of the scaling property of the item's content view to fit the maximum available space. After the animation has completed, the item exploits the entire bounds of the content area but still has not been redrawn. Then, simultanously the scaling property is resetted and the content view's frame is adjusted. Finally, changing the frame of the item's content view will trigger the behaviour according to the content mode that typically implies a redraw of the represented contents. On the other hand, demagnification requires the presence of a magnified item and is accomplished in reverse. Both aspects are also shown in `BCiLayoutManagerDemo`. Magnification of an item is illustrated in Figure 3.9. *Although an animation has several interpolation stages, the content view would only be drawn once at most.* This method has much better performance than changing the frame at every interpolation stage that would lead to multiple calls of the cost intensive content view's drawing code.



(a) Portrait layout shows five concrete items initially.

(b) After double taping "Item 3", the scaling property is being animated.

(c) The scaling property is resetted and the frame fits the bounds of the content area. Only "Item 3" remains visible.

Figure 3.9.: Screenshot: magnifying layout items

3.7.5. Dragging and Reordering Items

Before any layout manager accepts reordering items or more generally dragging items, it is required to enable the edit mode which is triggered outside of this module. In order

to visualize the edit mode, the transformation matrix of every item's content view is continuously altered by carrying out a wobbling animation. This behaviour is similar to the native app launcher application. In doing so, the frame property will not be effected, *thus no drawing code is called at this point.* Then, if neither scrolling nor other activities are being performed, items are permitted to change their position within the content area. The corresponding gesture is also implemented beyond the scope of this module.

Items notify the layout manager about dragging to a different position has started, has been performed or has already been finished. In these cases, no layout parameters are influenced. Hence, the layout will not change at all and the bounds of all content views remain the same. Nevertheless, the grid layout manager arranges particularly the center property of all affected content views according to the current layout. Altering the center instead of the frame property saves potential expensive drawing code. If dragging has started, the opacity of the dragged item is reduced to 70%. This is reasonable in order to keep all other items in the content area visible, even if the dragged item overlaps another item. Then, whenever the item is dragged again and changes its position, the grid layout manager tests if a specific *drag action* is applicable which will be executed. If no notification has been received for 0.1 seconds and dragging still has not ended, grid layout manager will intrinsicly test if a *drag action* became suitable. Thereby, more than one *drag action* can be performed until dragging has finished. After all, grid layout manager resets the opacity and moves the dragged item according to its page slot.

Currently, the grid layout manager is capable of distinguishing three *drag actions* that are all utilized in the context of reordering items (see Figure 3.11). Nevertheless, it is trivial to add additional *drag actions* to `BCiGridPageLayoutManager` for different purposes in the future. Basically, the grid layout manager keeps track of a couple of information about the last slot that was hit by the center of the dragged item. This includes one of the *hit zones* illustrated in Figure 3.10 and the time that zone was continuously hit so far.

Testing if a *drag action* should be executed is performed in a particular order:

1. *Changing the visible page* occurs if the dragged item has moved to the right or left border of the content area (page) and more than the sixth part of the item's width lays outside of the page. Additionally, the last hit page slot has to be hit for at least 1.5 seconds. Then, the scroll view will change the content offset to show the next or previous page if possible.

2. *Scrolling within a page* is performed if the dragged item overlaps the top- or bottom

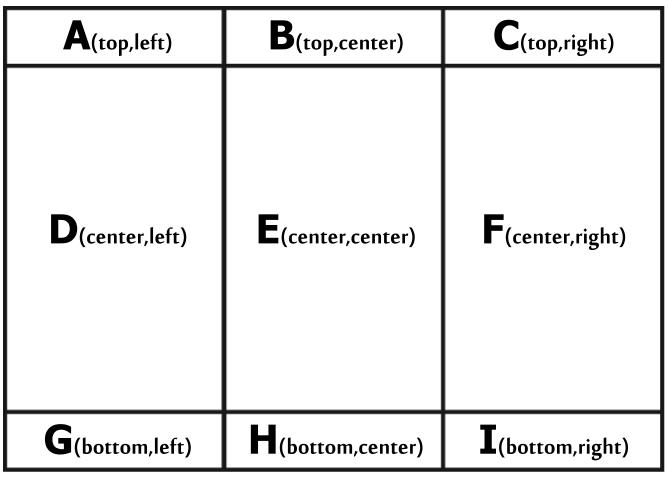


Figure 3.10.: Page slot hit zones - all zones have the same width but height of A,B,C,G,H and I equals the sixth part of the slot's content frame.

bounds of the visible content area by at least the sixth part of the item's height. Then, depending on the affected bound, the visible content area is attempted to be moved by at least half the height of the area either upwards or downwards. If this would lead to slots that are not entirely visible, the offset is rounded to show only entire slots. After the offset was determined, the scroll view begins to scroll. Also this *drag action* requires to hit the same page slot for more than 1.5 seconds continuously.

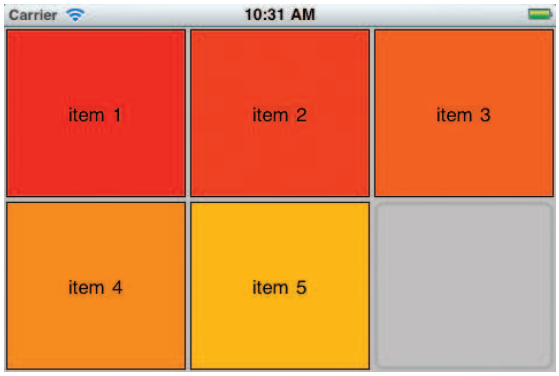
3. *Reordering items* is applied if the last hit page slot was occupied and hit one of the exterior zones, but also if it was empty and the slot at the center was hit. In both cases, the same zone has to be hit by at least 0.7 seconds.

   - If the hit slot is empty, the enclosed proxy item will be replaced with the dragged item. Then, the slot previously associated with the dragged item will contain a proxy item instead.
   - Else if the hit slot is occupied and resides on the same page, all items between the hit slot and the dragged item's slot will shift by one position towards the slot of the dragged item. Shifting is illustrated by animating the center of all item's content views. After all, the dragged item will be assigned to the hit slot.
   - In constrast, if the hit slot is hosted by a different page, all items will shift by one position to the closest empty slot of the hit slot. If no empty slot was found, first, slots for an entire row will be created and appended to the page

of the hit slot. Then, the previously associated slot will be emptied. The rest is similar to the previous case.

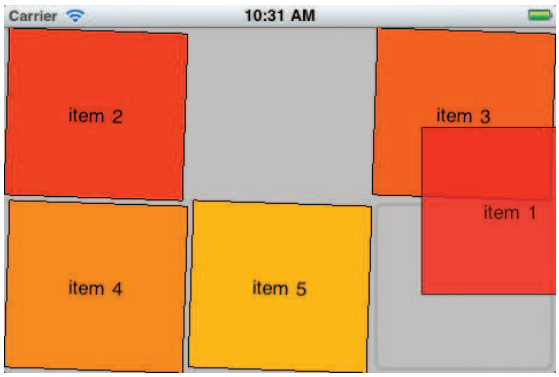
All time limits to trigger specific *drag actions* could be altered by modifying a single particular constant.

In order to save computing time, animations (e.g., wobbling) are applied only on visible items and turned off while the set of visible items is changing (e.g., the content area scrolls).

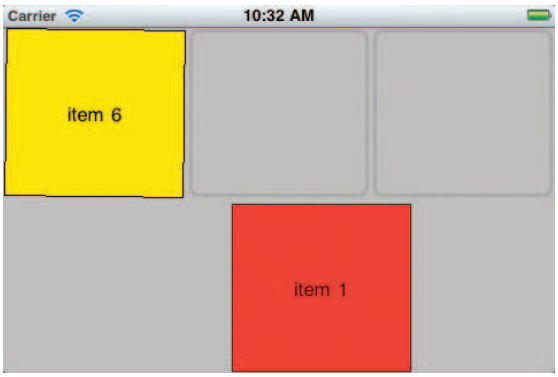


(a) Landscape layout shows five concrete items and one empty slot initially.

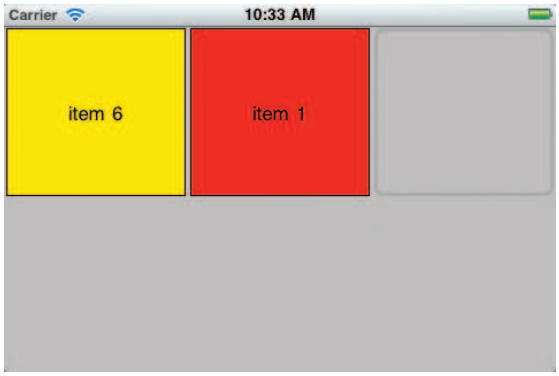
(b) Edit mode has been enabled.



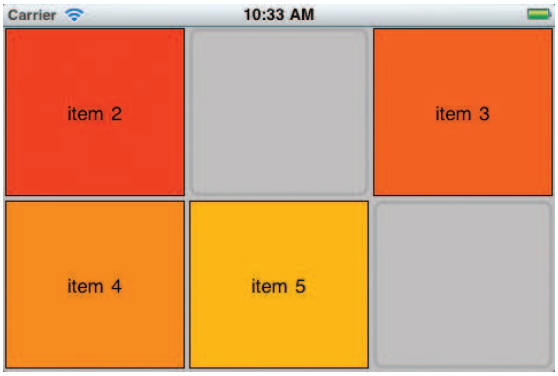
(c) Dragging "Item 1" has started and *reordering items* was performed by moving "Item 1" over "Item 2". Hence, "Item 2" moved to the initial position of "Item 1". *Changing the visible page* could be executed if "Item 1" stays in that position for 1.5 seconds.



(d) After the time has expired, the visible page has been changed. The current page shows only one visible item and two empty slots. The dragged item still has not been released.



(e) *Reordering items* was performed that replaced the dragged item with the proxy item of one empty slot. Subsequent *drag actions* would have been possible but the dragged item has been already released. Finally, the item was arranged according to the slot that was just associated.



(f) The original page now contains two empty slots as a result of *reordering items*. Also the edit mode was disabled again.

Figure 3.11.: Reordering layout items

3.8. Graphs Module

Primarily, the *Graphs* module encapsulates an infrastructure to visualize any live updated set of periodic- or aperiodic data. iScope does not directly use specific views for displaying sampled signals rather special purpose view controllers described in Section 3.8.3.

The XCode® project is located in `trunk/src/LibBCiGraphs` that contains all targets from Table 3.6. These targets could be also built individually.

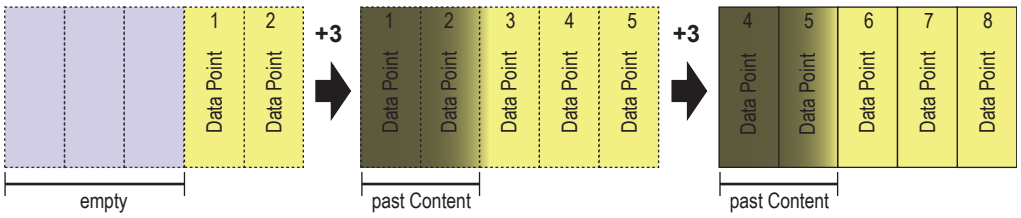
Target	Description
`LibBCiGraphs`	Static library
`Tests`	Unit tests
`BCiGraphsDemo`	GUI driven demo app showing all available graphs visualizing simulated demo data.

Table 3.6.: Deployment targets: Graphs module

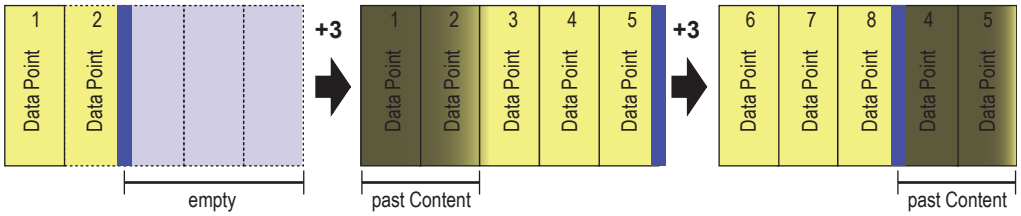
3.8.1. Drawing continuous Content

Most of the visualizations iScope uses to show live graphs enable the tracking of measured (sampled) and processed (spectral) data over time. Consequently, it is essential that every time new data is displayed, also parts of the past content should remain visible. In this manner, every data point is drawn on an equal sized allocated space on a predefined content area, and the amount of added data points determines the spare space that is left to plot previous content. Additionally, it is a project requirement to support two specific content visualization modes. According to medical device monitors (e.g., ECG monitoring), the first mode plots data points at the position of a vertical bar that runs from the left to the right border of the content area over and over as illustrated in Figure 3.12(b). The second mode results in a scrolling graph demonstrated in Figure 3.12(a) similar to the live graph of the native iOS® stock market app. The drawing routine has to meet the following objectives for both content visualization modes:

- Fast drawing of new data points to ensure very high data update rates.
- Fast rendering of a potential *high* number of previously plotted data points at *arbitrary* positions.



(a) Scrolling visualization mode



(b) Vertical bar visualization mode

Figure 3.12.: Continuous content visualization modes - both content areas could exactly plot a maximum of five visible data points.

Almost always, the content will be embedded in a more complex graph except for testing purposes. For this reason, it is more economic to implement the behaviour in an individual light-weight `CALayer`. Customizing a heavy `UIView` would just lead to unnecessary hierarchical view levels in the superview. Figure 3.13 gives an overview about `BCiContinuousContentLayer` that represents a `CALayer`, and is capable of plotting *abstract* data points consisting of one or more values according to both visualization modes. Drawing code is injected by overriding `drawInContext:` as explained in Section 2.4.7.1. Nevertheless, `BCiContinuousContentLayer` will hand a set of data points over to the concrete subclass that knows how to draw these data points starting from a specific *horizontal* position.

A naive approach would redraw the entire visible content whenever data is updated, even though only one data point was added. This leads to bad performance if the total number of visible data points is high, and/or multiple `BCiContinuousContentLayers` operate in parallel. Better performance could be attained by drawing only new data points and reusing already rendered content. Unfortunately, if the layer's drawing routine was invoked and no drawing operations have been applied, only arbitrary data that will not correspond to the previous content would be visible. Of course, using `setNeedsDisplayInRect:` instead of `setNeedsDisplay` would only clear the specified subarea of the layer and conserve the rest but would not permit placing past contents on a different position anyway. Hence, no adequate method exists to avoid loosing previously rendered

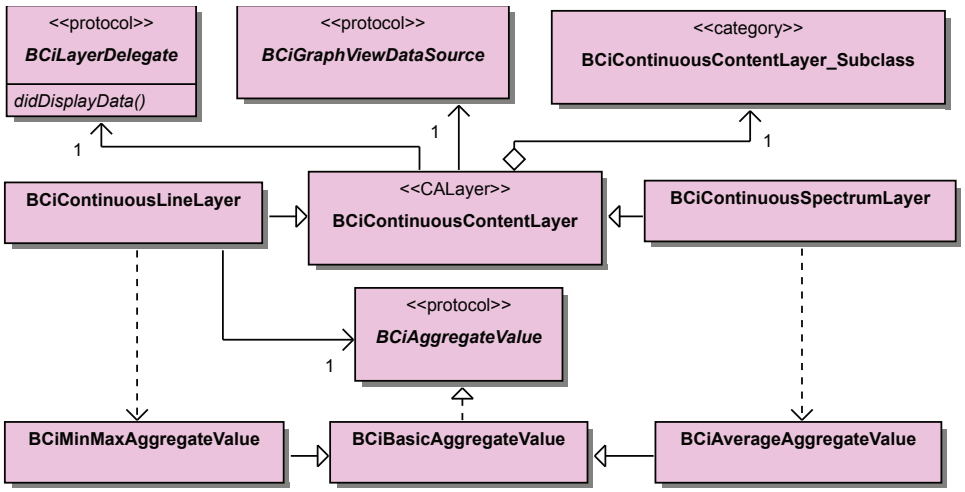


Figure 3.13.: Class diagram: continuous content layer

content which leads to the necessity of explicit caching.

In some cases (e.g., scope), a `CGMutablePath` could be used to store information of rendered data points as path elements or shapes, but several problems disqualify this approach. Above all, this method does not really cache the rendered graphical state and makes it necessary to prompt Core Graphics to render every single shape to the screen anyway, which is still expensive. At a particular point in time, extending the path whenever a data point has been added will probably cause iOS® to run out of memory. For this reason, a solution would be reusing path elements due to the lack of `CGMutablePath` to provide a way to remove elements. Further, at least the scrolling visualization mode involves rendering the past content on a different position that would imply correcting the coordinate offsets of all affected shapes. Both issues would force iterating through the *entire* path and adjusting every element which has no benefit compared to creating a complete new path, except avoiding the heap overhead of instancing new path elements. Additionally, this approach could not be applied if data points involve more complex drawings that do not rely on primitive shapes (e.g., spectra).

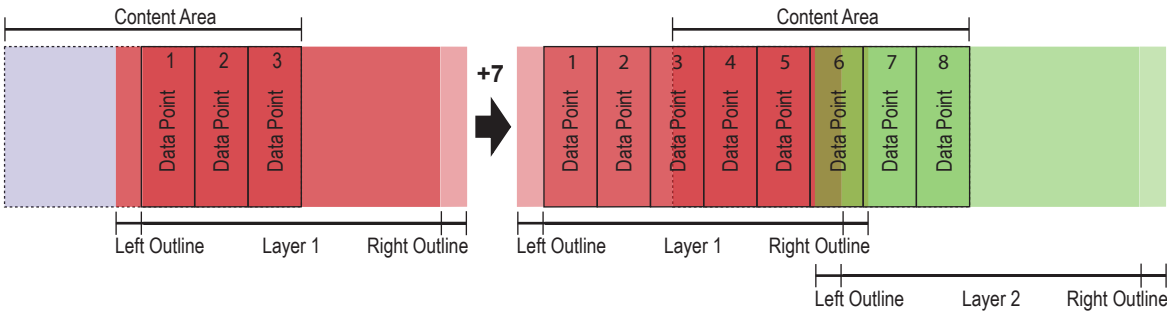
After all, `BCiContinuousContentLayer` implements an approach utilizing two instances of `CGLayer` to truly cache and reuse rendered data points, both matching the dimension of the total content area. The knack is to draw data points in the first layer beginning at the left border until the right border has been crossed. Then, in the same way, drawing is done in the second layer. In this connection, first of all, it is crucial to render the last drawn data point also in the second layer at an appropriate position to ensure a consistent transition to the next layer. Thereby, every concrete `BCiContinuousContentLayer` is

responsible of caching the last drawn data point to enable fast follow-up rendering. If no space is left in the second layer after further data points have been drawn, the first layer will be cleared and recycled. In this manner, drawing on a layer, switching between layers and clearing the previous layer is a reiterative process. Finally, all pre-rendered contents from both layers are drawn on the screen according to the visualization mode. In the case of vertical bar mode, the layers are simply drawn on the top of each other (see Figure 3.14(b)). Thereby, the top layer has to be clipped in order to keep the required contents of the other layer visible. If scrolling mode is designated, the layers are placed side by side as illustrated in Figure 3.14(a). In this regard, it is important to paste each layer at an integral point offset to avoid a visible gap between the two layers which could be the possible result of interpolation. Basically, this is done by rounding the offset to the next integral point in the floating-point coordinate system. This fits perfectly on devices that share a screen scale factor which maps logical points to natural pixels almost $1 : 1$. Although this does not apply to devices with Retina® displays, this method is suitable though. Nevertheless, in the future, an optimization for Retina® displays (and other displays) could round the offset to the next point value that matches an integral pixel based on the screen scale factor.

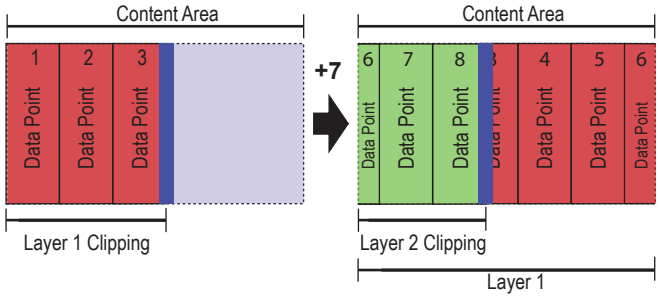
Furthermore, every `BCiContinuousContentLayer` is extended by an optional outline area on the left and right cap. This is necessary if rendered data points do not fit entirely in the designated area and therefore, partially overlap. In this case, the scrolling mode would show incomplete data points at the beginning and ending of every layer if the outline was not set correctly which can be observed in Figure 3.16. If an outline is used in scrolling mode, the outline area of one layer will overlap the other layer.

Similar to common `CALayers`, drawing is triggered by calling `setNeedsDisplay`. Nevertheless, `BCiContinuousContentLayer` overrides this method to have more control over this process. Thereby, a flag indicating data should be redrawn is set and evaluated in the next drawing cycle. Analogously, in order to clear the contents of the layer before any content is drawn, `setNeedsClearData` has to be called in addition. It does not matter, which method was called earlier if both have been applied. In this case, always the contents will be cleared first.

Optionally, rendering can be performed offscreen in a separate background thread. Without the application of `CGLayer` this would not be possible. This feature is adjuvant to keep the app responsive if the expected number of simultanously rendered data points is very high. Then, while rendering contents, all subsequent calls to `setNeedsDisplay` and



(a) Scrolling visualization mode



(b) Vertical bar visualization mode

Figure 3.14.: Offscreen layers in continuous content layer

`setNeedsClearData` will be ignored. After rendering has finished, the drawing routine is scheduled automatically again. In this way, the contents are finally made visible in the next drawing cycle. However, most of the time, this feature is deactivated in iScope.

Regardless of using a background thread, a delegate is informed by the main thread through `BCiLayerDelegate` protocol as soon as data has been displayed again. This allows thread synchronization without locking and a finer grained control over the entire rendering cycle.

Within a drawing cycle, data is provided data set by data set through `BCiGraphViewDataSource`. Every data set is represented as a `BCiDataIterator` facilitating transparency of the actual class that could manage any type of data (e.g., data samples, spectra). This method enables the general implementation of fetching data and continuously drawing data points in the base class, and also leads to less implementation- and testing effort. When executing the drawing routine, the data source is asked multiple times for a data set until no further data set could be gained. Additionally, for each data set, a *number* of *omitted* data points exists. Omitted data points do not comprise real information rather information about the absence of data points. Substitutional, the last drawn real data point will be reused accordingly. If no data point has been drawn yet, the data source

is optionally asked to provide a hint. In order to avoid redrawing a huge number of omitted data points that cannot be displayed in the content area, normalization is applied to draw only the maximum visible amount. In this context, it is important that data point offsets are preserved and show the same results as all omitted data points would have been drawn. This is especially relevant in conjunction with several synchronous layers that represent the same total amount of data points which could be composited differently, respectivly with a varying number of omitted data points. Furthermore, the data source is also responsible of providing meta data for the current drawing cycle which indirectly determines the size of the allocated drawing area per data point. In this way, amongst others, also the minimum- and maximum data point values are specified. It is also mentionable that if background rendering is activated, the data source has to guarantee thread-safetyness during the whole rendering process.

At this stage, no data decimation will be performed. For this reason, in spite of the limited screen resolution of iOS® devices, it should be possible to plot any number of data points to the content area, even though the number is a multiple of the content area width. This is feasible due to the floating-point coordinate system that Core Graphics provides. Nevertheless, aggregation could be enabled to render a reduced amount of data points optimized for the current screen dimensions for performance reasons. In this manner, either data points and/or data point values are combined together that leads to less drawing operations. The concrete `BCiContinuousContentLayer` decides how aggregation is performed based on factors (data point/value aggregate weight) provided by the data source. For further optimization, the rendering quality of a `BCiContinuous-ContentLayer` could also be reduced. In doing so, rendering a huge amount of data points at the same time could also be distinctly accelerated. Although subclasses of `BCiContinuousContentLayer` are suggested to render with less quality if requested, they do not have to adopt this advice. Both techniques are useful if speed (or response time) outranks visual precision.

3.8.1.1. Data Aggregation

Aggregation is used to combine a set of values to generate a reduced amount of corresponding values (aggregates) by means of different aggregate functions. Values are added to an aggregate which in turn applies the aggregate function. Every aggregate has a weight which indicates how many values are accepted until the aggregate is complete. This weight could also be a floating-point number. After the aggregate has been completed, a possible produced overflow will be added as a carry to the next aggregate. Thereby, the

aggregate weight could be switched at any time. Table 3.7 demonstrates an example of the fundamental technique.

Annotation	Current weight	Overflow	Carry	Total weight
	0	0	0	0
Added value	1	0	0	1
Added value	2	0.5	0	2 (complete)
	0	0	0	0
Added carry	0	0	0.5	0.5
Added value	1	0	0.5	1.5 (complete)
		...		

Table 3.7.: Aggregation example: shows two complete aggregates using a weight of 1.5 which is applicable if for example 6 values should be mapped to 4 aggregates. Current weight reflects the amount of added values, whereas total weight sums up all components.

This concept is abstracted by `BCiBasicAggregateValue`. Two aggregate functions are implemented, either calculating the minimum and maximum (`BCiMinMaxAggregateValue`), or the average (`BCiAverageAggregateValue`) of all given values. The first method maps each complete aggregate to two values in a chronological order whereas the second one produces only one unique output.

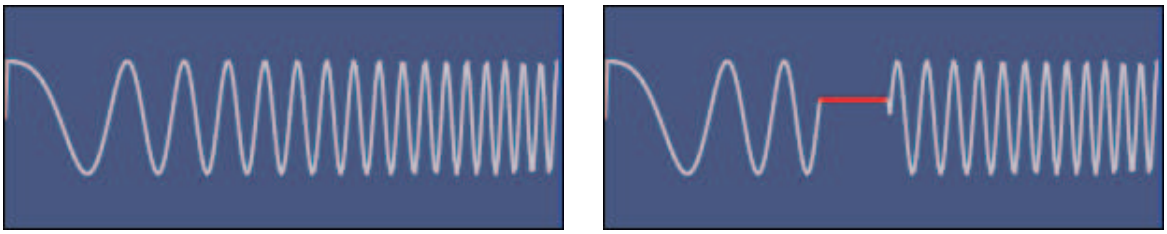
3.8.1.2. Drawing Scope

Before reinventing the wheel, Core Plot was evaluated if it could be utilized to implement the scope by drawing live updated line charts. Although the library provides `CPTXYGraph` that is basically capable of drawing these kind of charts, it is not intended to plot a large amount (≥ 1000) of frequently updated (at 60 FPS) data. Using Core Plot would result in unadequate visualization update rates due to the fact that all data points have to be redrawn on each drawing cycle. [69]

For this purpose, `BCiContinuousLineLayer` is a concrete `BCiContinuousContentLayer` interpreting elements of a given data set as single data points visualized by connected line shapes. These lines are drawn in light-grey color starting from the last drawn data point or the vertical center if no data point has been drawn so far. Then, linear projection is used to map data points to the vertical dimension of the layer where the line ends. Thereby, the layer's top edge refers to the maximum presentable value and the bottom

edge to the minimum value. In this context, it is necessary to keep track of the previously and penultimatly drawn data points. An omitted data point is drawn as a horizontal line from the last data point to the next data point offset at the same vertical position with a different line color (red). Multiple omitted data points are combined into one single horizontal line in order to avoid redundant drawing operations and increase performance. Figure 3.15 illustrates two layers displaying an equal amount of data points which vary in composition.

When drawing any diagonal line shape starting from the left bottom corner of a given



(a) Without omitted data points

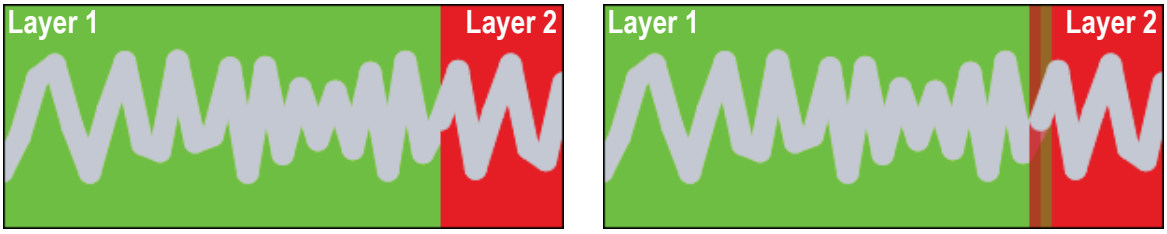
(b) With 30 omitted data points

Figure 3.15.: Screenshot: scope layer example - shows the first second of a linear chirp signal with 256 Hz, starting with 1 Hz and crossing 100 Hz after 3 seconds.

virtual area, some parts of the line will be drawn outside of that area as well. This is due to the fact that not the outer edge but the center of the line is located at the origin of that area. Therefore, it is indispensable to specify an outline that amounts the half of the line thickness (rounded up) which is sufficient to cover the largest area that ever would be drawn outside. In scrolling visualization mode, a missing outline would show incomplete lines on the caps of the internally used `CGLayers` when pasting the layes side by side. Specifically, the part of the line that lies beyond the allocated data point area will be truncated and not visible which could be observed in Figure 3.16.

If `BCiContinuousLineLayer` is asked to use a reduced rendering quality, primarily line shapes are drawn without antialiasing. Further, Core Graphics is assigned to perform every drawing operation targeting one of the internal layers with a lower rendering quality.

`BCiContinuousLineLayer` also provides support for data point aggregation that could dramatically accelerate the presentation of a huge number of data points due to a potential heavy reduction of drawing operations. Thereby, the aggregate weight influences the number of data points which are combined to a lighter aggregate. A min/max-aggregate is



(a) Line is broken at layer caps without an outline (b) Line is continuous with an outline of six points

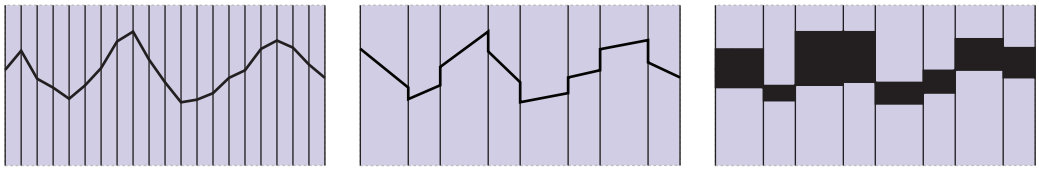
Figure 3.16.: Screenshot: continuous line layer outline - shows one second of a signal with 32 Hz; data points are drawn using 12 point Line shapes

used to preserve information about the entire value range. Complete aggregates are drawn either *disconnected* or *connected* depending on the relation between the aggregate weight, the current width of the allocated data point area and the line thickness. Figure 3.17 illustrates both methods:

- Aggregates are drawn *connected* if the largest possible allocated area for an aggregated data point allows visible distinction between two line shapes or rather aggregates. This is the case if $(Data\ Point\ Width \cdot Aggregate\ Weight) > Line\ Thickness$. Then, every complete aggregate is represented as a horizontally adjusted line, starting from either the maximum- or minimum value of the aggregate and ending at the respective opposite whichever occurred first. Each aggregate is also connected with the ending of the previous aggregate using a simple vertical line. Although this method is still fully implemented, it is no more utilized by iScope because aggregate weights are never specified in the way to meet the criteria above.

- On the other hand, if it would not be possible to distinguish between displayed aggregated data points, further drawing operations can be saved by drawing a *disconnected* line for each aggregate. Mathematically, this appears if $(Data\ Point\ Width \cdot Aggregate\ Weight) <= Line\ Thickness$. Then, every complete aggregate is drawn as a single vertical line. No gaps will be visible because the line shape itself fills the assigned area entirely. In doing so, the thickness of the line corresponds to the number of data points in the completed aggregate. In some cases, the value range of an aggregate does not overlap the range of its antecessor which would result in a vertical, visible gap between both aggregates. However, that gap is filled by extending the line in the direction of the closest ending of the antecessor.

Omitted data points will never be part of an aggregate and always treated regularly. Before any omitted data point is processed, potential incomplete aggregates will be flushed first.



(a) No aggregation (b) Connected aggregates (c) Disconnected aggregates

Figure 3.17.: Drawing aggregates: an aggregate weight of 2.5 is used to map 20 data points to eight aggregates.

Instruments shows that `BCiContinuousLineLayer` reaches up to 60 FPS constantly, irrespective of aggregation or reduced rendering quality is activated. This was tested with one layer that was updated 60 times per second under iOS® 4.3.1 on an iPhone® 3GS compiled with LLVM GCC 4.2. 60 FPS also represents the maximum possible value for any target platform.

3.8.1.3. Drawing Spectra

`BCiContinuousSpectrumLayer` derives from `BCiContinuousContentLayer` in order to display spectra one after another. Every spectrum corresponds to one data point that does not consist of one value but rather a set of values provided by the data source. Values are color-coded by a mapping to the Hue Saturation Brightness (HSB) space. In this manner, only the hue component is adjusted in the range from 0° (red) to 240° (blue), while saturation and brightness remain the same. Thereby, red corresponds to the absolute maximum- and blue to the absolute minimum value. As a consequence, it is guaranteed to create a discrete color spectrum without grey color graduation. Then, the spectrum is vertically divided into uniform sections which are filled with calculated colors. Omitted data points (or spectra) are represented by using a different alpha value. Both types of spectra are illustrated in Figure 3.18. [80]

In order to ensure efficient repeated drawing of the last spectrum, every spectrum is drawn to a `CGLayer` first and after that pasted to one of the two internal buffer layers. If `BCiContinuousSpectrumLayer` is assigned to render the buffered spectrum multiple times, the spectrum layer is still drawn only once but with a different scaling that fits the desired width. This approach saves performance without having any drawbacks in

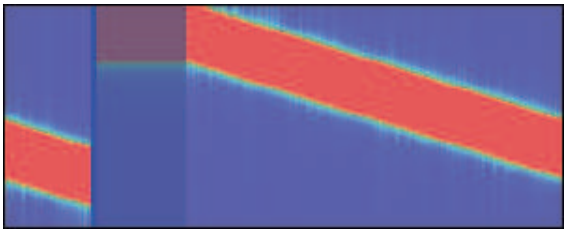


Figure 3.18.: Screenshot: spectogram layer example - shows the first 3.5 seconds of a spectogram based on a linear chirp signal with 256 Hz, starting with 1 Hz and crossing 100 Hz after 3 seconds. Spectra are computed 60 times per second. Every spectrum uses Hamming windows of 1 Second. Therefore, no spectral information is available until the first window could be selected which is indicated by omitted data points.

precision. In general, for the spectrum layer, interpolation in the native Red Green Blue (RGB) color space is deactivated to avoid potential grey colors. Further, the two internal layers use a specific blending mode which causes to override the current contents with the contents of the current spectrum to prevent blurring.

Depending on data value aggregation is enabled, the dimensions of the spectrum layer and how a spectrum is effectivly prepared differs. In contrast, data point aggregation is generally not supported.

- If data value aggregation is disabled, the spectrum layer's height is equal to the number of values per data point (or spectrum) and one in width. Then, every calculated color corresponds to one single point in the spectrum layer. After all, the prepared layer is rendered using an appropriate scaling to match the allocated data point area.

- On the other hand, if data value aggregation is used, the spectrum layer's dimensions exactly match the allocated data point area. In doing so, every value is added to an averaging aggregate first. The aggregate weight is calculated by splitting the content area height into as many uniform sections as values are contained in one data point (or spectrum). Then, for every completed aggregate, the color in the HSB space is computed which is used to finally render the aggregate. In this manner, only a reduced number of actually visible aggregates rather than all data point values are drawn. Additionally, no further scaling is necessary which also improves performance.

3.8.2. Graph Views

In the course of iScope, different graph views (or `BCiGraphViews`) are created for various visualizations. All share a similar interface to notify delegates (`BCiGraphViewDelegate`, `BCiGraphDelegate`) about specific occurences and receive data and/or meta data through designated data sources (`BCiGraphViewDataSource`, `BCiGraphViewMetaDataSource`). Every graph view derives from `UIView` and confirms to the `BCiGraph` interface. Figure 3.19 gives an overview about all implemented graph views.

In the strict sense, these graph views do not visualize data directly, rather manage

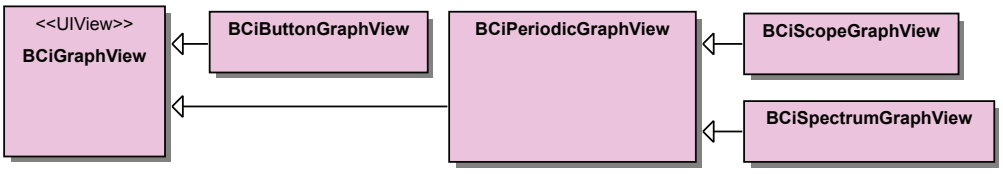


Figure 3.19.: Class Diagram: Graph Views

a couple of specific generic Core Animation sublayers that are assigend for this task. Hence, no drawing routine of `UIView` is used, instead sublayers are forced to redraw their contents by calling `setNeedsDisplay`. Thereby, all involved sublayers with the exception of continuous content layers also implement the `BCiSmartLayer` interface. This will block invoking the drawing routine of a smart layer unless the dimensions of the content area or the represented contents have effectivly changed. In this manner, smart layers are only redrawn if it was really necessary, thus improving performance.

The following *generic* smart layers are implemented:

- `BCiTextLayer` renders a given text in an optimal font size to match the bounds of the layer.
- `BCiYAxisLayer` draws an optionally mirrored vertical axis including labels in an ideal font size.
- In contrast, `BCiXAxisLayer` is capable of rendering a horizontal axis that also shows labels in ideal font sizes.
- `BCiHorizontalGridLayer` visualizes a configurable horizontal grid with solid- and dashed lines.
- On the other hand, `BCiVerticalGridLayer` draws a vertical grid that is highly adaptable, also using solid- and dashed lines.

Every graph view takes care of stacking sublayers in a well-defined order that minimizes drawing effort. Expensive alpha channels are only applied, if transparency increases usability or important information would be hidden. In this case, the bounds of the sublayer are kept as small as possible to avoid unnecessary computations.

Further, all graph views have exactly one background image to create an unified appearance. For this purpose, a stretchable image of no more than 27×31 pixels is used to represent the entire background of arbitrary size. In this manner, also round corners are achieved without setting the corner radius property of the underlying Core Animation layer that would have envolved an alpha channel and drain performance.

Graph views could be switched into detail mode that reveals additional information depending on the concrete view. In doing so, almost always the layout of specific sublayers has to be rearranged. Due to the lack of a suitable layout manager at layer level, every frame has to be set manually.

In general, three types of information are distincted which could be independently updated. These are dynamic contents, either data that is provided by the data source or non-data and simple static content. If the bounds of the graph view have changed, all dynamic non-data- and static content is redrawn. In this case, dynamic data content is only cleared in order to avoid redrawing a potential high number of data points automatically. Nevertheless, the delegate is informed about layout changes which in turn could force to finally redisplay data.

3.8.2.1. Scope Graph View

`BCiScopeGraphView` is a concrete `BCiGraphView` that uses generic smart layers as well as one additional specific smart layer to represent a scope with additional meta information (see Figure 3.20).

According to the detail mode, contents vary slightly. In every case, the view shows at least:

- The title of the graph in the left top corner of the view as a `BCiTextLayer`.
- In the center of the view, a `BCiContinuousLineLayer` represents the actual contents of the graph.
- A `BCiVerticalGridLayer` that uses different line types for seconds, half seconds and tenth seconds which is also linked to the configured observation period.

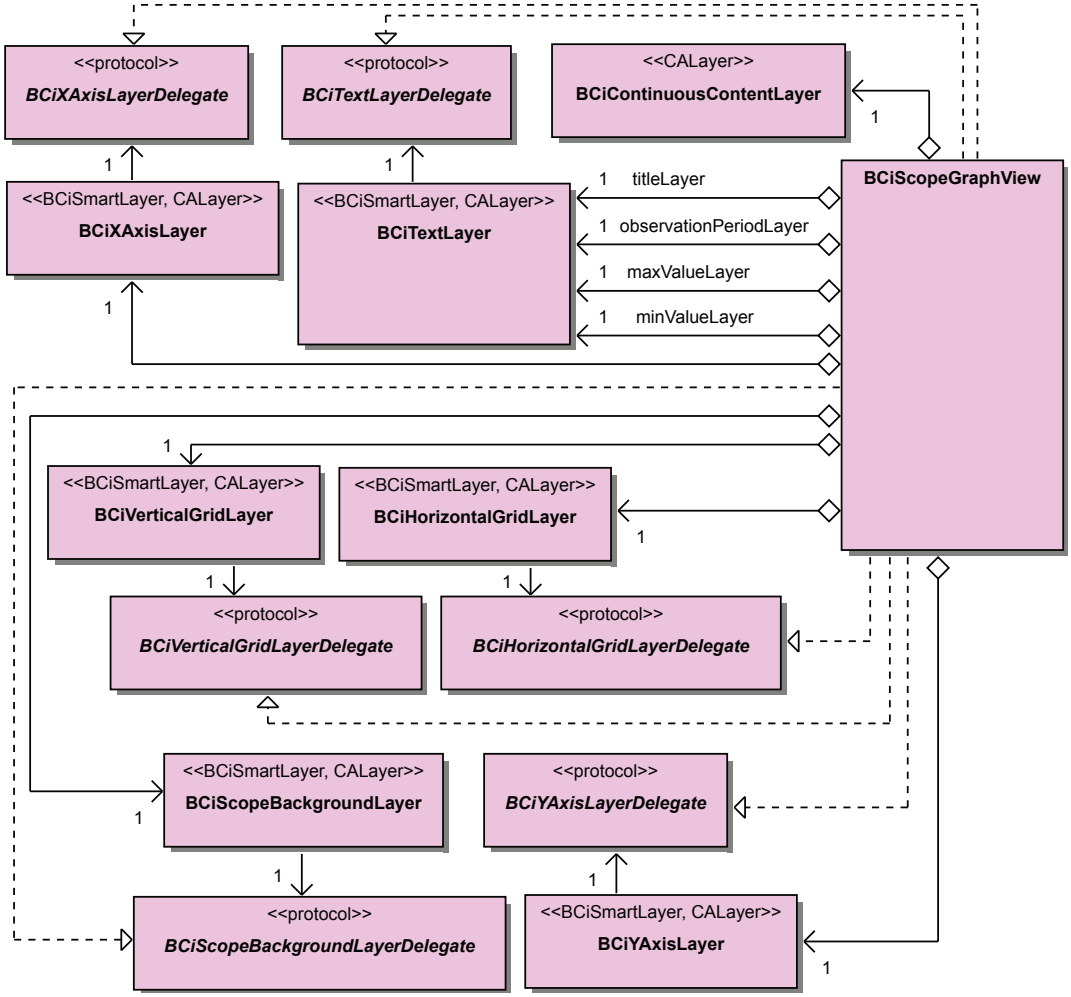


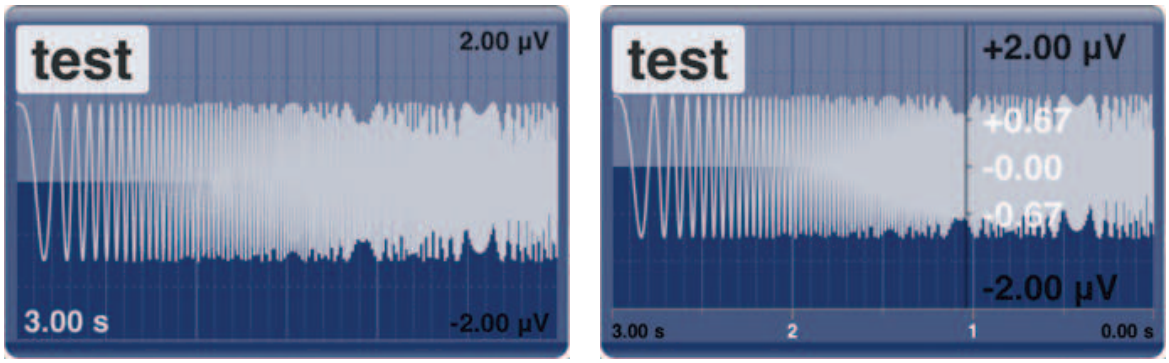
Figure 3.20.: Class diagram: scope graph view

- A centered `BCiHorizontalGridLayer` with five lines.
- In the background, a layer that is vertically splitted into two equal sized areas. Every area is filled with a different color to highlight the center value.

In regular mode, the maximum and minimum value (amplitude) is represented as a `BCiTextLayer` in the top right and bottom right corner. Thereby, the current observation period is displayed in the bottom left corner by means of a `BCiTextLayer`. Figure 3.21(a) shows a recorded sample signal in regular mode.

In contrast, the detail mode shows an additional time axis and omits a seperate label for the observation period as demonstrated in Figure 3.21(b). Additionally, the vertical axis labels every value of the vertical grid.

Only the first two decimal places will be displayed for each label for layout reasons.



(a) Regular (b) Details

Figure 3.21.: Screenshots: scope graph view

3.8.2.2. Spectogram Graph View

`BCiSpectrumGraphView` represents a concrete `BCiGraphView` that displays a spectogram. The internal layer composition is similar to those used in `BCiScopeGraphView`. Nevertheless, it does not dislay any grid and it represents spectra by a `BCiContinuousSpectrumLayer`. It also uses one specific smart layer as illustrated in Figure 3.22.

In regular mode, there is almost no difference compared to `BCiScopeGraphView`, except rather frequencies are shown than amplitudes (see Figure 3.23(a)).

The detail mode shows frequencies instead of amplitudes as well. Additionally, a box displaying the mapping between colors and actual values used to plot the spectra is added at the right edge of the graph which could be observed in Figure 3.23(b).

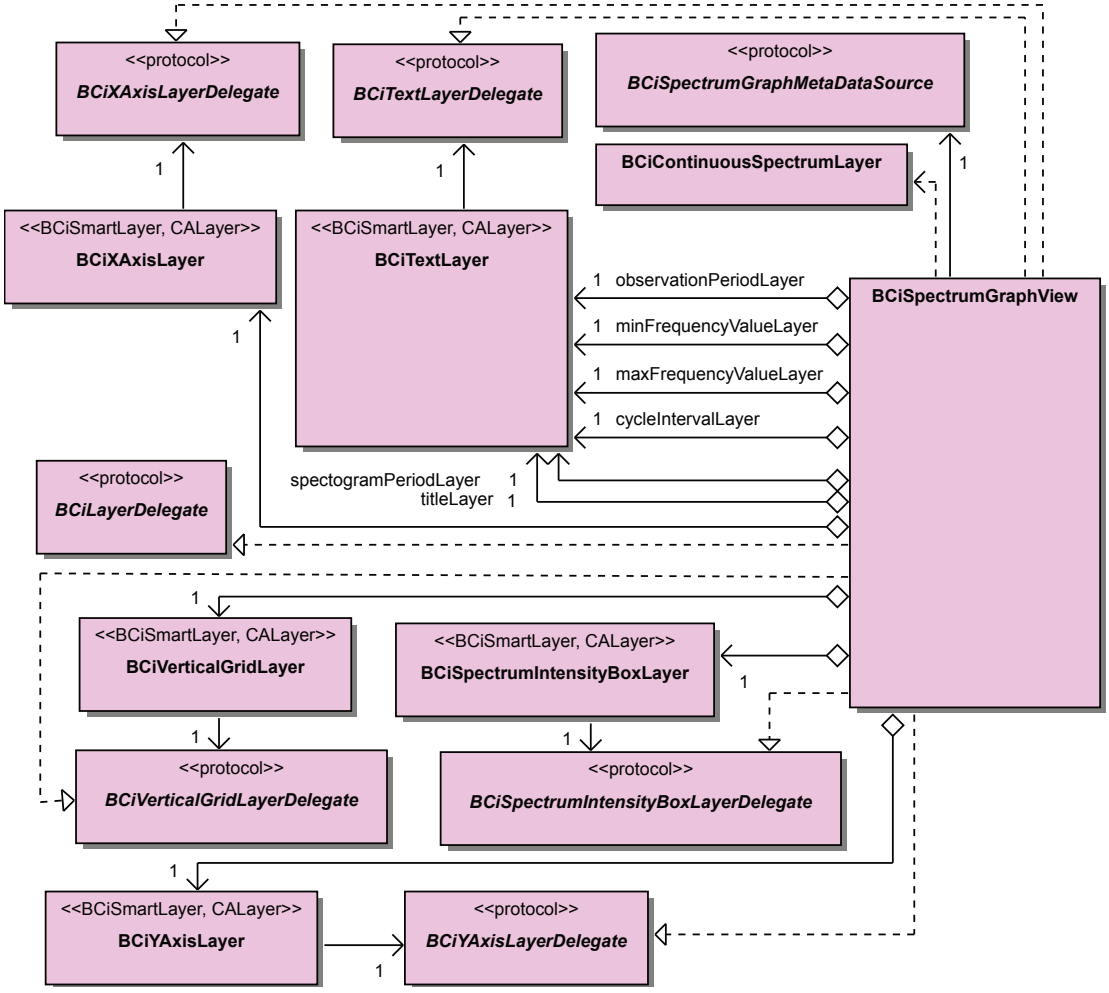
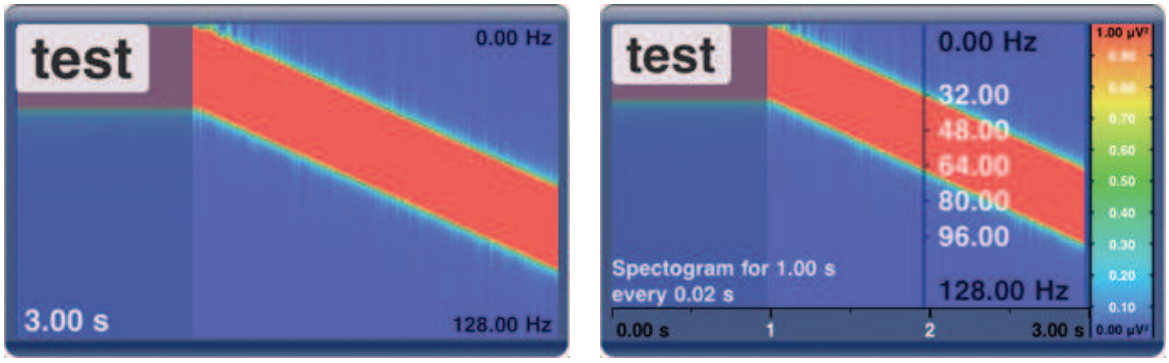


Figure 3.22.: Class diagram: spectrum graph view



(a) Regular (b) Details

Figure 3.23.: Screenshots: spectogram graph view

3.8.2.3. Button Graph View

`BCiButtonGraphView` is a simple `BCiGraphView` reporting the current state of a signal similar to a labeled button. Figure 3.24 demonstrates all involved classes.

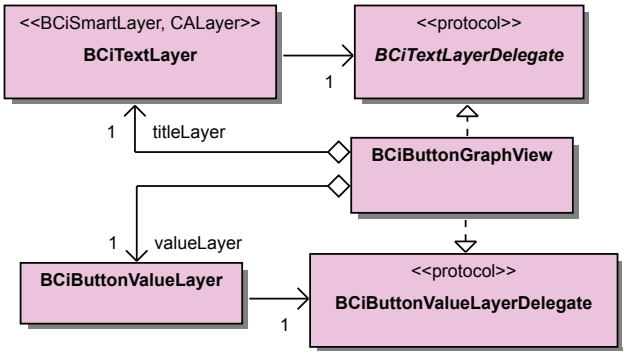


Figure 3.24.: Class diagram: button graph view

Analogously to other graph views, the title of the graph is displayed in the top left corner of the view. Detail mode and regular mode always have the same appearance. Thereby, in the bottom right corner, a rounded box is attached to the view that contains the current value of the signal. The font size is flexible and will always exploit the width of the box. In this manner, only the first three decimal places will be considered as illustrated in Figure 3.25. According to the current state, the background of the box has a different color. The color is dark blue if the current value exceeds a configured threshold, red if the current value is not initialized and grey if the current value is less than the threshold.

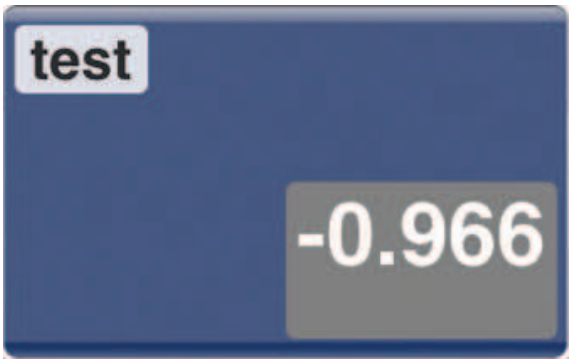


Figure 3.25.: Screenshot: button graph view - current value is less than the configured threshold of 1.0

3.8.3. View Controllers

View controllers of the Graphs module do not substitute UIKit's `UIViewController` and should rather be considered as a logical extension of these. In this section, the term view controller refers to classes implementing the `BCiViewController` interface. They are a building block of a custom *MVC* assemblage. Thereby, every view controller is used to retrieve, manipulate one particular graph view and is in charge of hosting the supplying data model. In turn, a view controller is typically managed by a `UIViewController` and therefore part of another *MVC* construct. Figure 3.26 gives an overview about the public view controller API. Further, view controllers expose its graph views as `BCiGraph` that offers a reduced API and do not reveal the underlying view. This approach facilitates the extendability and exchange of visualizations.

Basically, two main groups of view controllers are distincted, `BCiPeriodicGraphView-Controllers` and `BCiAperiodicGraphViewControllers`. Periodic view controllers are designed to manage visualizations of trends, whereas aperiodic view controllers relinquish the time component. In each case, a designated data source and meta data source is asked through particular interfaces to provide the most recent data. Relying on interfaces rather than concrete classes alleviates testing without having real life data. Figure 3.27 shows the relation between graphs, data sources and view controllers.

The *Graphs* module provides `BCiGraphViewControllerFactory` which accords to the prototype factory- and factory method pattern to instantiate concrete view controllers [74]. Creating view controllers based on registered prototypes makes it easy to add custom view controllers for other purposes or exchange existing ones. At any time, only one instance of the factory exists due to the adoption of the explicit singleton design pattern.

Every view controller derives from `BCiBasicGraphViewController` which implements amongst others, two fundamental functions:

- Fetching data from associated data sources either by omitting or processing every added data point. Data points are omitted by calling `omitData` whereas `updateData` will consider each data point individually. In addition, data points could be optionally marked as lost indicating no information is available for the corresponding time period. For this purpose, the view controller keeps track of the data sequence number gained through the last fetch cycle. Then, only if data has changed in any respect, drawing code could be invoked on the next request. The same also applies to meta data. Omitting every added data point drastically boosts performance due to skipping processing and reducing drawing operations. In this case, graph views

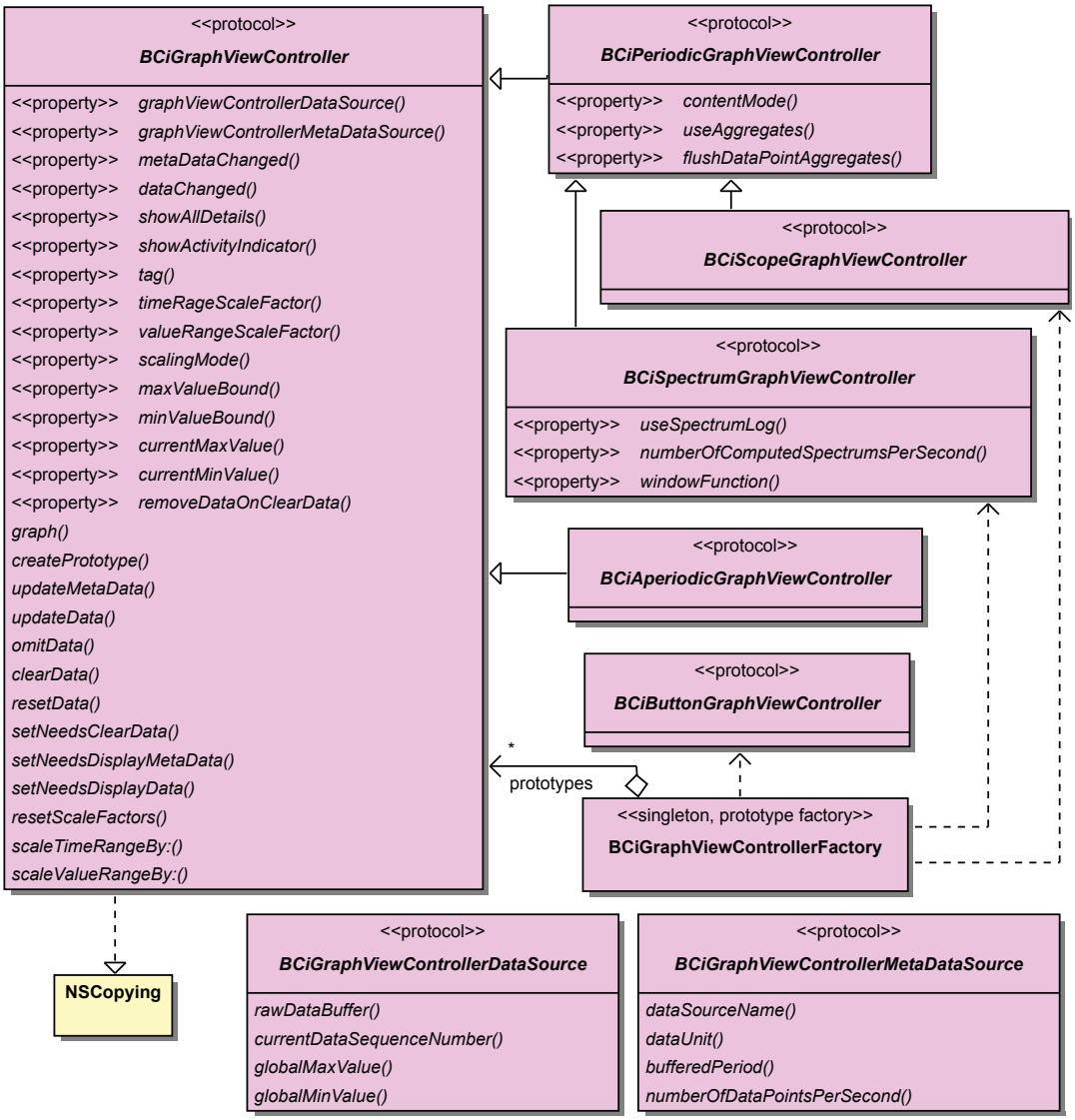


Figure 3.26.: Class diagram: graph view controllers overview

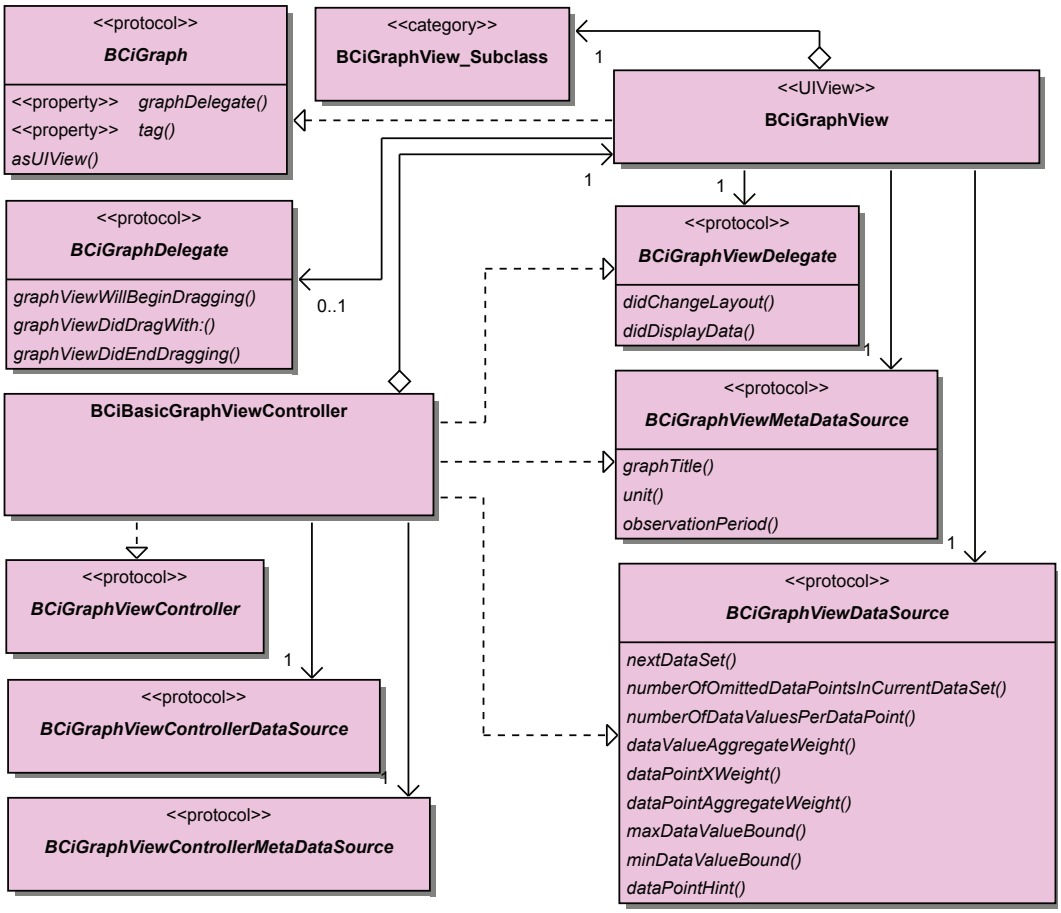


Figure 3.27.: Class diagram: basic graph view controller

will draw omitted data points instead of regular ones. This is useful if the view controller has not been asked to update its contents for a while and the amount of added data points is estimated high. Lost data points will also be treated as omitted in the rendering process.

- Configuring and scaling of maximum and minimum representable data point values. In this manner, scaling factor will snap to the original limits if it was set close enough to exceed or fall below a certain threshold of ± 0.05.

- Locking the internal data model and API calls related to fetch cycle while managed graph view is rendering in the background.

On every fetch cycle, periodic view controllers are asked to store added data points in a seperate data model as illustrated in Figure 3.28. Although multi-level data models could exist in parallel, only one data model is active and used to supply the graph view. Presently, just one data model is hosted that contains data in the highest available resolution. However, switching between data models with different resolutions is already possible and could be utilized in the future. This would transform the work load from rendering to preprocessing (e.g., downsampling) by reducing the amount of data points that have to be effectivly considered for visualization.

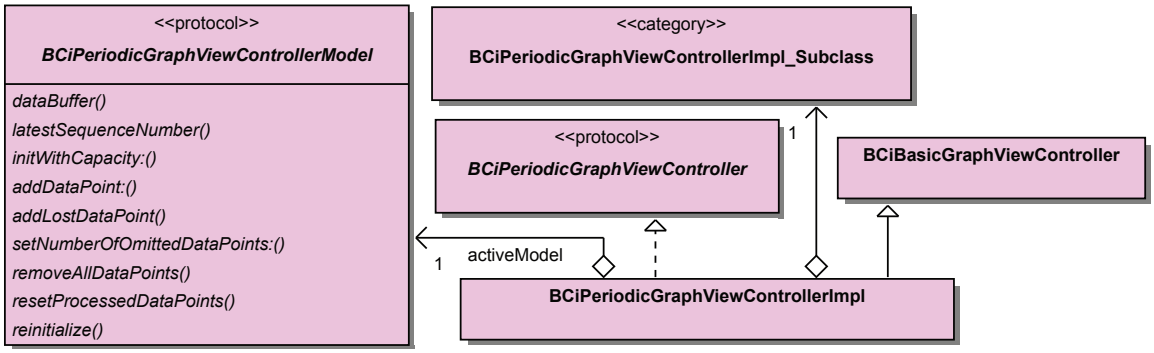


Figure 3.28.: Class diagram: periodic graph view controller

Two periodic view controllers are designed to handle the visualization of scopes (see Figure 3.29) and spectograms (see Figure 3.30). The current implementations are using `BCiScopeGraphView` and `BCiSpectrumGraphView`.

Further, every periodic view controller features the following basic functions:

- Enabling the usage of aggregates. In this case, the view controller decides which form of aggregates is suitable. Also the aggregate weight is determined by the view

controller. In doing so, scope view controllers will use aggregate weights to reduce the number of data points and drawing operations to optimally take advantage of the graph view's content frame. This leads to aggregated data points with a width of one point in the floating-point coordinate system. Even though this approach is not optimized for Retina® displays, it will work on all related devices though. The same applies to spectogram view controllers, although only data point value aggregation is performed. In this manner, one data point comprises the results of one FFT. Then, a number of bins is aggregated that ideally fits the height of the graph view's content frame in order to draw one aggregate with a height of one point in the floating-point coordinate system.

- Setting and scaling the time dimension. Basically, every periodic view controller represents samples of one signal within a specific observation period. Anyhow, the visible time period can be temporarily changed by applying a scale factor. This factor will always result in a time period containing an *integer* number of raw data points. Additionally, the factor snaps to the next full second if the current scaling factor enters a threshold area of ± 0.15 seconds. Although spectogram view controllers will show time periods reflecting only *integer* raw data points, this is not true for spectra which result from comprising multiple raw data points in one spectrum. In this way, scope view controllers as well as spectogram view controllers could be used to observe exactly the same time period of a given signal.

Specifically the spectogram view controller infrastructure makes use of *Common* module's FFT encapsulation. Thereby, time window is not yet configurable and will be always one second of the signal. However, this is determined by a single constant which could be altered without much effort. On the other hand, the number of computed spectra per second and the window function are already fully parametrizeable and specified by the application. Furthermore, lost data points received from the data source will be substituted with their next valid antecessors before the FFT is calculated.

Currently, only one aperiodic view controller exists that attaches a `BCiButtonGraphView` which can be observed in Figure 3.31. It has no explicit data model and also disregards scaling of the time dimension because only the most recent value is utilized.

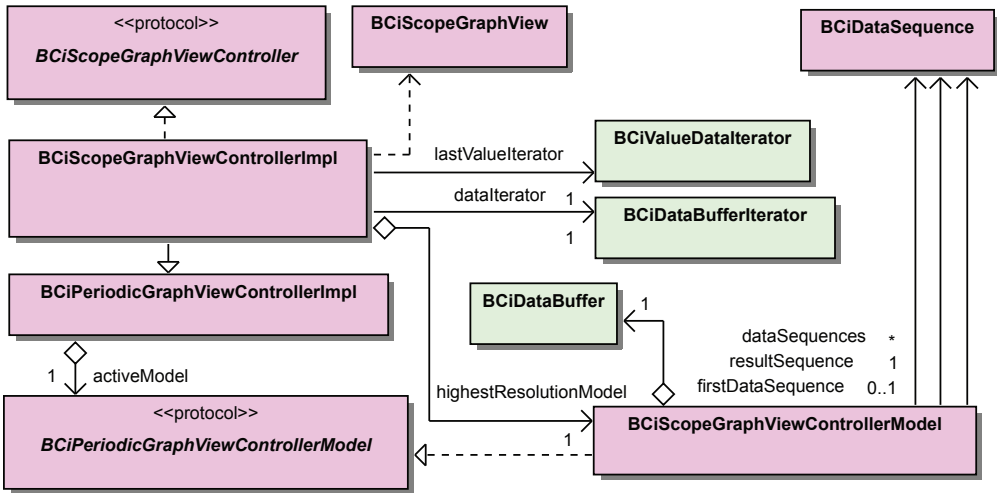


Figure 3.29.: Class diagram: scope graph view controller

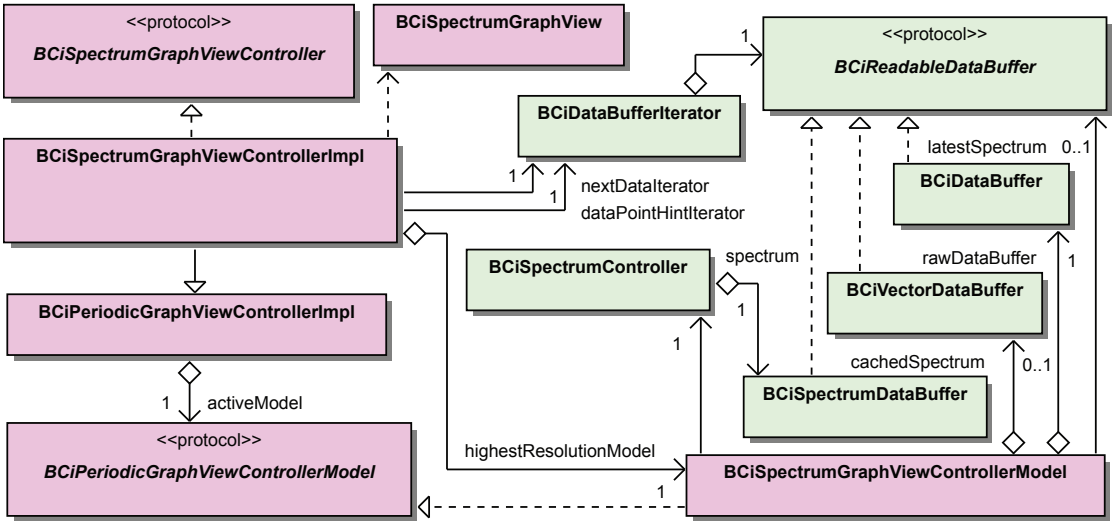


Figure 3.30.: Class diagram: spectrum graph view controller

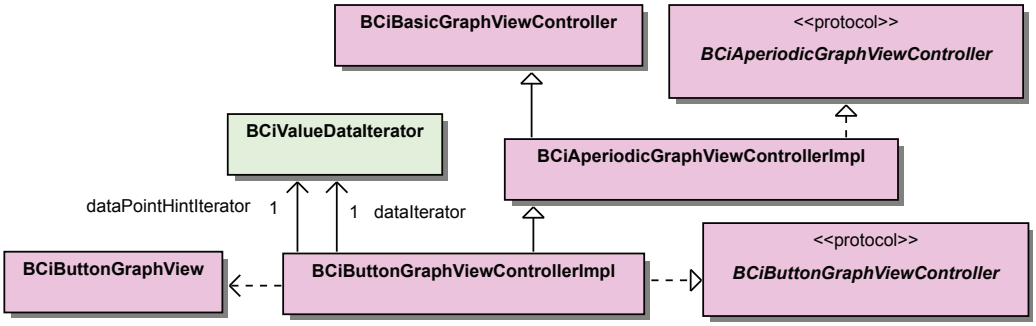


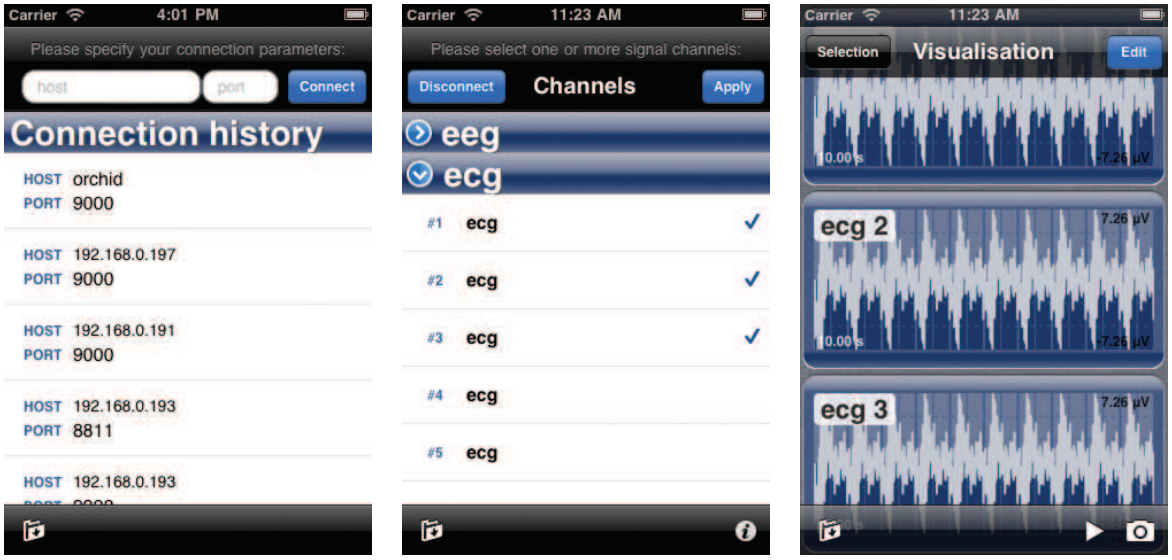
Figure 3.31.: Class diagram: aperiodic graph view controller

3.9. iScope - App Module

This module assembles the executable application that is based on all other module libraries as well as on TinyXML++ and related Boost libraries. Hence, these libraries also have to be statically linked with the final binary. The associated XCode® project file is located in `trunk/src/BCiScope` and contains the target `BCiScope`. In order to work correctly, iScope demands iOS® devices with accelerometer- and wireless LAN capabilities.

3.9.1. General

Basically, iScope involves three main screens which are the connection-, selection- and visualization screen as demonstrated in Figure 3.32. Furthermore, two orthogonal sub-



(a) Connection screen (b) Selection screen (c) Visualization screen

Figure 3.32.: Screenshots: iScope main screens

screens (settings-, signal details screen) will guide the user through particular workflows. Transitioning between each other is done either by using the top navigation bar or the bottom toolbar. The navigation bar allows to reach the previous and next screen relative to the active screen. Thereby, screen transitions will use different UIKit's built-in animations. In order to ensure clear navigation, transitions between main screens use a different animation than transitions to/from subscreens. Figure 3.33 shows a state machine that defines all possible transitions.

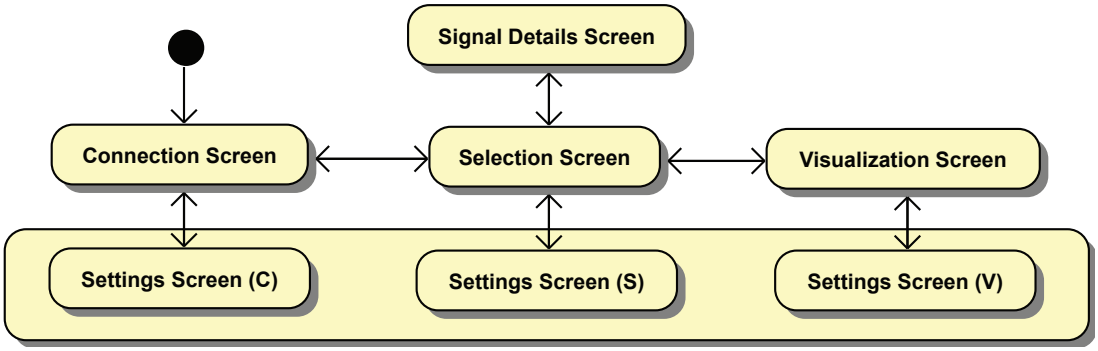


Figure 3.33.: State machine: iScope's active screen - final state is not visible because it can be adopted by every state.

Every screen is represented by a `BCiScreenViewController` which is derived from `UIViewController` and therefore, responsible for all corresponding tasks described in Section 2.4.4.2. Screen view controllers generalize basic functionality like showing/hiding a spinning activity indicator, the presentation of the root view and managing the toolbar. The toolbar is split into two sections. All global buttons are visible on every screen and are left-aligned. Currently, only one global button exists that will open the settings screen. The current screen will be exchanged with the settings screen by flipping around its vertical axis in an animation sequence. In constrast, local (or screen specific) buttons are attached on the right side of the toolbar.

`BCiScreenViewControllerMap` guarantees that the same instance of every screen can be retrieved from any location. Nevertheless, screen view controllers are intiliazed by a nib file that is tailored for the present device. In this manner, some parts of the layout are arranged statically which will not affect the dynamic nature of the internal visualization screen's layout. Although different nib files for iPad® and iPhone®/iPod touch® devices are used, functionality of each screen is implemented in device independent classes. Figure 3.34 shows a global point of view of incorporated classes.

In many respects, the behaviour of iScope is configurable by two types of settings. In general, global settings that are neither frequently changed nor alterable at runtime can be defined in the native iOS® Settings app (see Section 3.9.3). In contrast, local settings can also be changed while iScope is in use and are managed by the settings screen (see Section 3.9.8). `BCiApplicationSettings` encapsulates both types of settings.

Changes of the physical orientation of the device are recognized by means of the accelerometer. In this case, every screen view controller will also arrange its views according to the new orientation (by adopting either the portrait- or landscape perspective).

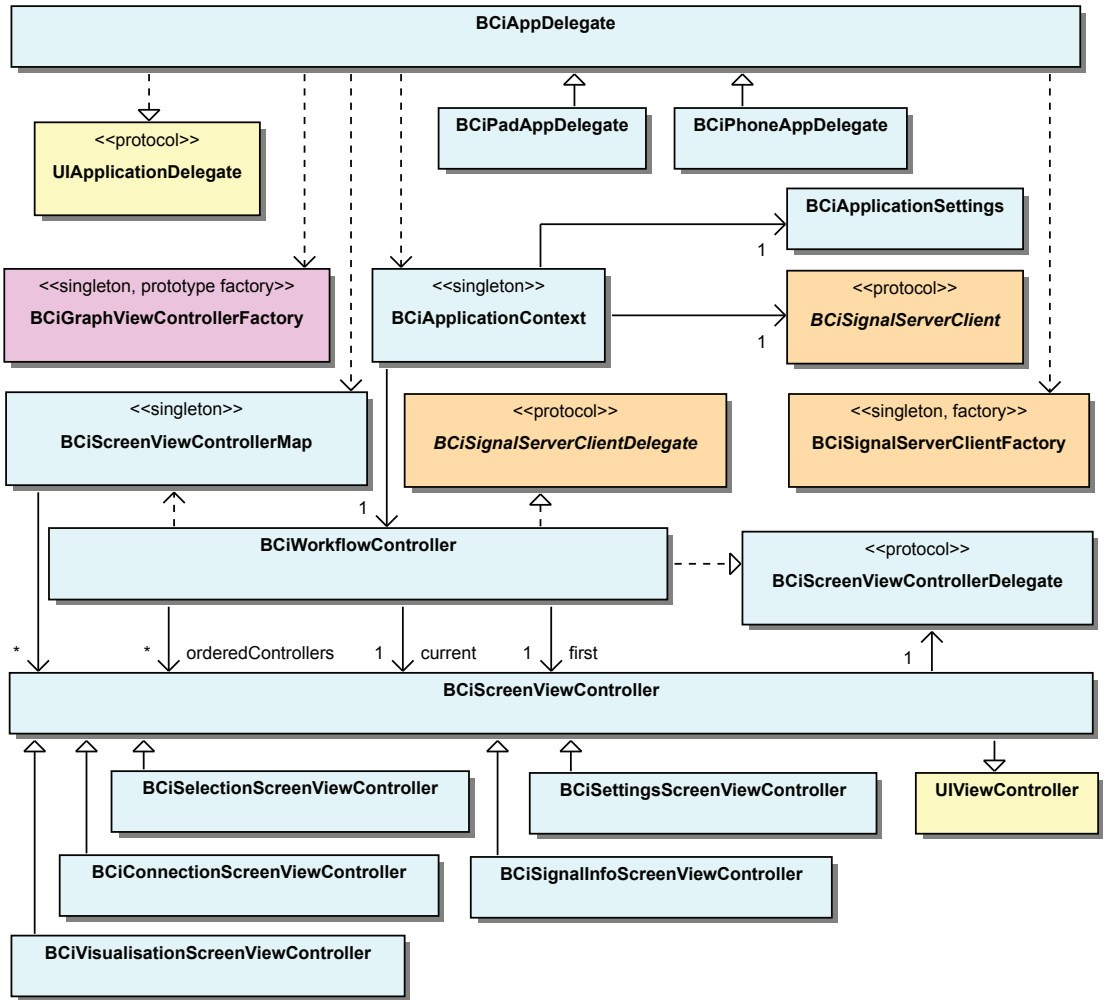


Figure 3.34.: Class diagram: iScope overview

Basically, every screen has access to the singleton instance of the `BCiApplicationContext`, thus the same `BCiSignalServerClient`, `BCiApplicationSettings` and `BCiWorkflowController`. The workflow controller is in charge of managing the order of screens.

3.9.2. Application Start and Shutdown

While iScope is being launched, an image optimized for the current device and orientation is shown. Basically, the image looks like the first screen (i.e. connection screen) without any mutable content (e.g., text). Once the application is ready to use, the image will be replaced by the real connection screen. This approach should suggest the impression that the application launches faster.

iScope responds to changes of the application state. The majority of tasks is done in `BCiAppDelegate`. Further, two device specific application delegates exist that are derived from `BCiAppDelegate`. After iScope has finished launching, they carry out initialization tasks and are responsible for registering unique screen view controller instances in the `BCiScreenViewControllerMap`.

Basically, background execution is not supported, thus hitting the home button will terminate the application. In this case, iScope attempts to regularly close an open signal server connection. Unfortunetaly, there is no guarantee that this task will complete successfully until the application is definitly purged from memory due to the time limitations explained in Section 2.2.1. Pseudo multitasking has no beneficial impact because all non Voice over IP (VoIP) sockets would be still closed after the application has moved to the background state. Consequently, also the socket based control connection to TiA server would have been interrupted.

At any time, if the TiA client becomes corrupted (see Section 3.6.3.1), an explaining pop-up dialog will appear prompting the user to restart the application manually. Otherwise, iScope would keep running, would instance another client and the former client would be memory leaking. It is discouraged by Apple to programmatically quit an application with `exit(0)` or similar.

3.9.3. Settings App

iScope adds a custom entry to the native iOS® Settings app. For this purpose, corresponding setting screens are automatically generated based on interlinked property list

files (`.plist`) located in the app's settings bundle. Basically, every screen is configured by one seperate property list file. XCode® provides a convenient way to edit those files and manage associated entries. Currently, two screens are used for basic settings and settings for more experienced users. Furthermore, iScope specializes its settings bundle for iPhone®-, iPod touch®- and iPad® devices. Amongst others, the possible values for different settings are made device dependend. Figure 3.35 shows a specialized version of both settings screens. All settings are persisted in the user's defaults database which is accessed by `BCiApplicationSettings` to initialize all global settings members.

The following basic settings are defineable:

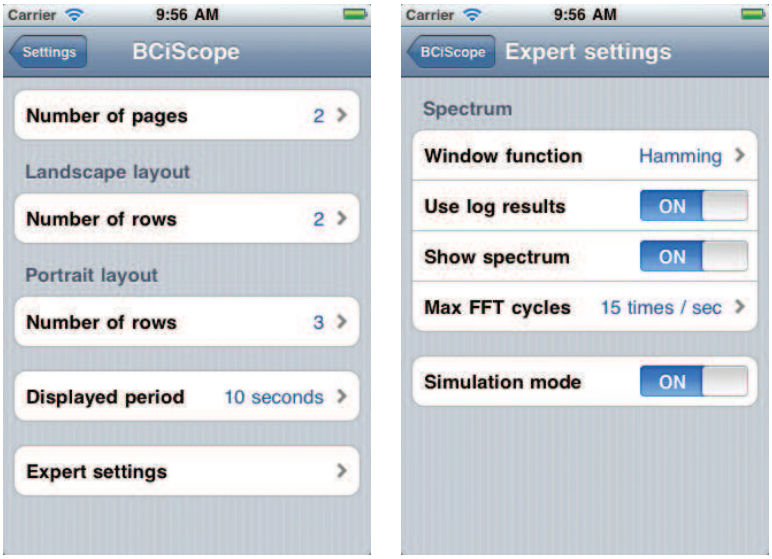
- *Number of pages* reserved for graphs in the visualization screen.
- *Number of rows*, *Number of columns* per interface orientation. These values determine the grid layout of the visualization screen for portrait- and landscape layout. Only iPad® devices support the columns parameter to ensure readability.
- *Displayed period* specifies the maximum recorded time period of any signal.

Furthermore, the settings below are configurable through expert settings screen:

- *Window function* which is applied to the input vector of the FFT that is used to compute the spectogram.
- *Use log results* switches the unit of the spectogram from μV^2 to dB and vice versa.
- *Show spectrum* indicates if spectogram graphs are shown on the visualization screen.
- *Max FFT cycles* per second determines if the spectogram is based on an overlapping STFT. The actual number of cycles depends on the integer result of $f_s / Max\ FFT\ cycles$ which calculates the amount of raw data points after one FFT is computed.
- *Simulation mode* will use *Simulated Server Client* from Section 3.6.3.2 instead of the *Real SignalServer Client* from Section 3.6.3.1.

3.9.4. Multi-Language Support

Basically, every application related file could be internationalized. iOS® apps use language projects (`.lprj`) to maintain translations for specific combinations of language and/or region, both identified by their International Organization for Standardization (ISO)



(a) Basic settings (b) Expert settings

Figure 3.35.: Screenshots: iPhone® Settings app

codes. Thereby, each bundle manages its own language projects. Currently, iScope only has one main- and one settings bundle which provide one language project for English (`eng.lproj`) without specifying any region. However, creating additional language projects requires minimal effort. Always one version of every resource file has to be placed in the developer's native language project. At a glance, whenever a file is accessed, iOS® will automatically try to find the version that best matches the local language- and region settings of the user. If no match was successfull at all, the developer's native version will be used instead.

In this manner, all nib files, string files (`.string`) and all property list files related to the Settings app are internationalized. String files contain translations of specific hard coded text strings which are marked for internationalization. `genstrings`, a tool that is shipped with iOS® SDK can be used to automatically parse the source code and generate these string files in Universal Multiple-Octet Coded Character Set UCS Transformation Format for 16 Planes of Group 00 (UTF-16). Unfortunately, it is not possible to put translation projects in static libraries. In order to provide internationalized versions of all text strings from module libraries, the iScope module has to take care of all corresponding string files by itself.

3.9.5. Connection Screen

After iScope launched, the connection screen as illustrated in Figure 3.32(a) managed by `BCiConnectionScreenViewController` becomes visible. In the center of the screen, a chronological ordered connection history is presented by a scrollable `UITableView` with custom cells based on a nib file. It contains a list of hostname and portnumber pairs used by successful attempts to connect to TiA server(s). Currently, the last 50 pairs are shown but this number could be easily changed by altering a single constant in the source code. Whenever a connection was successfully established, the history is also written to a property list file in the application bundle in order to persist after the termination of iScope.

The navigation bar shows two `BCiNavBarTextFields` for hostname and portnumber as well as a "Connect" button. As the user taps one of the textfields, an onscreen keyboard appears prompting to enter both parameters. In order to hide the keyboard immediately, any other position on the screen has to be tapped again. The textfields also act as a filter for connection history entries. Only those entries will be shown in the current history that match the current input.

Hitting the "Connect" button initiates a connection attempt. In this context, also input parameters are validated. If one of the parameters contain an illegal value, a pop-up dialog will appear displaying a corresponding failure message. If all parameters were valid, `BCiSignalServerClient` will be used to establish a connection and request a list of all available signals in a background thread. In the meantime, a spinning wheel indicating the client is still working appears. All control elements are also disabled to ensure the user cannot initiate further connection attempts or navigate to another subscreen. If the connection was successfully established, the workflow controller will show the selection screen, else a pop-up dialog will open showing an appropriate failure message. In either cases, all control elements are enabled and the activity indicator is hidden again.

3.9.6. Selection Screen

The selection screen is hosted by `BCiSelectionScreenViewController` that primarly consists of a scrollable `UITableView` located at the center. Basically, the table contains multiple sections for all available signal types at TiA server. Thereby, every section is represented by a `BCiTableSectionHeaderView` which allows to customize user interaction. If the user taps the blue arrow button of the section header in original position, all rows of the corresponding section will expand animated. Otherwise, if the section already has

been expanded, the section will collapse also using a smooth animation.

Every row of the table is initialized from an individual nib file that corresponds to one channel. These rows show the channel number on the left and the channel name on the right. If processing of one channel is considered as a problem (e.g., the virtual sampling rate exceeds the estimated maximum of the present device), a red exclamation mark appears on the right side of the row in addition. However, when the selection screen appears, also a pop-up dialog will display a similar but more general statement.

Multiple channels of different signal types can be selected by taping the right cap of a table row. Then, a checkmark on the right cap of the corresponding row will indicate that the channel is effectivly selected. If the user taps a checkmarked row, the associated channel will be unselected again. Similarly, all channels from a section could be selected or deselected by taping the background of the section header and vice versa. Then, the background of the section header changes. It turns light if a further tap will unselect all channels and dark in reverse. If the section was collapsed additionally, it will be expanded first.

Returning to connection screen is done via "Disconnect" button in the navigation bar. This will also try to regularly close the connection to the TiA server asynchronously. On the other hand, "Apply" button in the navigation bar will force the client to collect data of all selected channels. In this context, it is important to bear in mind that also if critical channels are not selected, they will be transmitted, thus have to be read from socket and parsed anyway as explained in Section 3.6.3. However, `BCiSignalServerClient` already provides an appropriate API to select specific channels which is also utilized. Hence, this will also involve client/server communication and asynchronous program behaviour in the future. When the client finished its duties, the workflow controller will show the desired screen. As long as the client is busy, all control elements and the toolbar are disabled. If any error occured, the selection screen remains visible and an adequate message is displayed in a pop-up dialog.

The bottom toolbar contains one local button that brings the signal details screen to front which is managed by `BCiSignalInfoScreenViewController` and illustrated in Figure 3.36. An animation is used that covers the selection screen vertically. Dismissing is done by hitting "Close" button in the top navigation bar. The signal details screen also displays a scrollable `UITableView` in the center. Every row corresponds to one signal. The layout of a row is arranged by a nib file. Thereby, the type (e.g., EEG), sampling rate (e.g., 128 Hz), unit (e.g., μV) and periodicity (either periodic or aperiodic) of the

signal are displayed. If the signal is aperiodic, the sampling rate will be omitted.

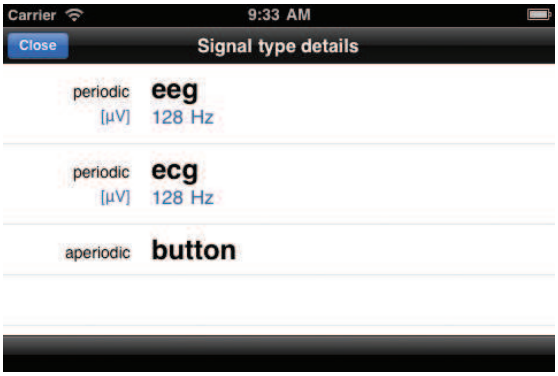


Figure 3.36.: Screenshot: signal details screen

3.9.7. Visualization Screen

Basically, `BCiVisualisationScreenViewController` manages the behaviour and the design of the visualization screen.

While the visualization screen is appearing, the status bar, navigation bar and the toolbar are turning translucent and will finally disappear after further 0.5 seconds. Nevertheless, a single tap will smoothly overlay the contents of the screen with the transculent versions of the bars and vice versa. This approach allows to take advantage of the entire screen without any restrictions on the usage of the navigation- and toolbar.

The content of the visualization screen consists of various graph views according to the set of configured functions and selected channels. Currently, two functions, "scope" and "spectogram", are defined for periodic signals and "button" for aperiodic signals. For every function, a graph view controller prototype is registered in the `BCiGraphView-ControllerFactory`. The controller of the visualization screen will ask the factory to clone prototypes for every channel depending on the periodicity of the signal. In this case, one `BCiScopeGraphViewController` and one `BCiSpectrumGraphViewController` will be created for all periodic signals and one `BCiButtonGraphViewController` will be created for all channels of aperiodic signals. Section 3.8.3 describes all associated visualizations.

After all, these graph view controllers provide the graph views. In turn, the layout of these graph views is arranged by `BCiGridPageLayoutManager`. In this manner, graph views will be uniformly distributed over the pages of the layout manager. The number of pages and graph views per page are specified by the Settings App, respectivly the current

global settings. Using the layout manager will also allow to reorder and (de-)magnify graph views by user interaction as demonstrated in Section 3.7. For this purpose, turning the layout manager into edit mode is done by "Edit" button in the navigation bar or pressing the screen of the device with one finger for at least 1.5 seconds. Then, "Edit" button will be exchanged with "Cancel" button and the layout manager animates the graph views. On the other hand, disabling edit mode is carried out in the same way.

During a fetch cycle, graph view controllers operate over data sources provided by the *Signal Server Client*. In principle, fetch cycles are initiated on a regular basis which occurs whenever a specific update timer fires. The timer interval depends on the highest virtual sampling rate in the current signal selection. If the rate exceeds the device maximum, the maximum is used instead inhibiting a potential performance drain.

The visualization screen implements a lazy update model to increase performance. On every timer tick, only graph view controllers of visible graph views will be asked to perform a fetch cycle, preprocess (e.g., calculate spectra) and visualize their data points afterwards. This is also performed if the layout manager finished changing the set of visible graph views. While the set is changed for any reason, especially in conjunction with scrolling or paging, the update timer will not fire until the layout manager has finished. This is a precaution to avoid a lagging user interface.

Figure 3.37 shows how all modules merge together.

In general, visualization screen supports two modes of operation:

- Data transmission is initiated by *Signal Server Client* and all visible graph views track and visualize received data. While tracking data, the update timer is installed.
- Data is no longer tracked and client has stopped data transmission. Also the update timer will not fire in this state.

Manually toggling these modes is possible either by using a dedicated local toolbar button or physcially shaking the device. A shake is detected if acceleration of the device in any direction exceeds a particular threshold which is determined by a single constant. Accordingly, fine-tuning shake detection is very simple. The toolbar button will change its shape from pause- to play symbol after tracking was activated or deactivated. Tracking data will always be initiated instantly as the visualization screen appears and the selection screen was formerly active.

Due to the lazy update approach, invisible graph view controllers could be inactive for a longer period. Then, handling all accumulated data points would be potentially expensive

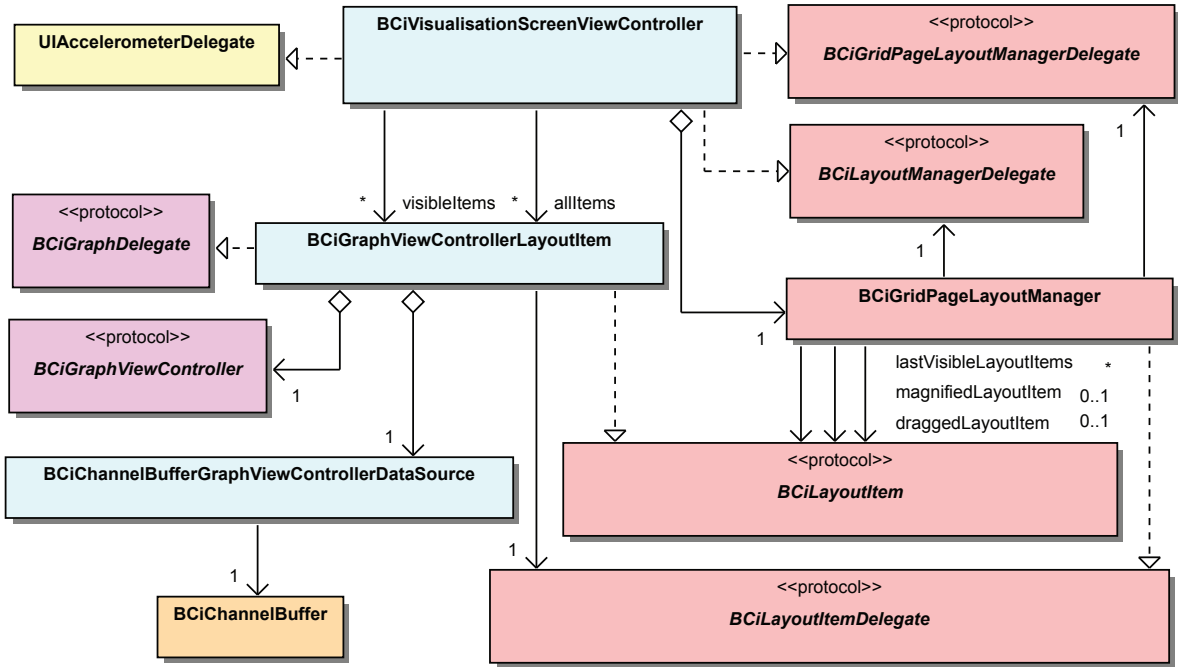


Figure 3.37.: Class diagram: visualization screen

and would slow down the application. This can occur whenever the set of visible graph views changes in conjunction with (de-)magnification of graph views, scrolling within the current page or turning pages and changing the user interface orientation. For this purpose, rather than processing every single data point by calling `updateData`, `omitData` is used to use the proxy representation of omitted data points instead as demonstrated in Section 3.8. This is the standard behaviour if data is currently tracked. In this way, it is impossible to compare two graphs that are not visible on the same area of the screen because scrolling would always induce omitted data points. For this reason, this characteristic is also oppositional in the non-tracking data mode. Hence, potential longer rendering times have to be accepted additionally.

All periodic graph view controllers support seperate scaling of the time- and value dimension. In this manner, both dimensions will snap to particular values (integer seconds, initial scale factors) if certain thresholds are exceeded as mentioned in Section 3.8.3. It is important to recollect, as graph views display only the first two decimal places of the observation period, scaling could lead to unsynchronous graph views in spite of showing equal observation periods. This could happen if the internal accurate observation periods are different.

Basically, scaling is triggered by performing pinch gestures which are tracked by a single

`BCiPinchGestureRecognizer` for the whole screen. Thereby, horizontal pinches will affect the time dimension and vertical pinches the value dimension. Every time the gesture recognizer adopts "Changed" state (see Section 3.5.3), derived scaling factor is applied to all graph view controllers but only visible graph views are forced to refresh visualizations. In this case, tracking data mode causes changes to the value dimension not being shown directly. The visualization is not cleared and the observable effects are delayed until the end of the next regular fetch cycle. In contrast, graph views will reflect changings to the time dimension immediately. In this manner, corresponding graph view controllers clear their associated contents first. Then, they are forced to redraw all previously processed data points by treating them as omitted data points. These two techniques facilitate keeping the user interface responsive by saving dispensable drawing operations.

On the other hand, if non-tracking data mode is activated and regardless which dimension is being scaled, graph view controllers will always clear their contents and redraw every already processed data point. Because a pinch is a continuous gesture and "Changed" state is very likely adopted several times in series, an optimization had to be implemented. In doing so, when the gesture recognizer moves to "Began" state, all visible graph view controllers are assigned to use aggregation and reduce rendering quality. At the end of the pinch which takes place if the gesture recognizer is either in "Cancelled"- or "Ended" state, both actions are rewinded. Furthermore, the contents of the graph views are cleared again. Then, every visible graph view controller uses an individual rendering thread to display the contents of their graph views in full resolution. While rendering, a spinning activity indicator stays in the center of each refreshed graph view. User interaction (pinches, scrolling etc.), all buttons in the toolbar and navigation bar are disabled until every spawned thread has finished.

"Selection" button in the navigation bar allows to return to selection screen. In this case, *Signal Server Client* is assigned to stop data transmission in the background. In the meantime, all toolbar buttons are disabled. If the client successfully stopped transmission, workflow controller will activate selection screen controller again. Otherwise, a pop-up dialog will display a failure message.

Generally, whenever visualization screen becomes visible, current channel selection and application settings are applied. This involves updating the set of graph view controllers maintained by layout manager as well as configuring them by changing the content- and scaling mode.

The toolbar provides a second local button to take screenshots of the current visible screen area. Currently, screenshots are rendered in native resolution but iScope could be extended to use higher resolutions in the future. In this manner, an entire white view fades in by altering the alpha channel from 0 to 1 in an animation sequence which should suggest the flash of a camera. At the end of the sequence, the screenshot is rendered. In the meantime, a spinning wheel is shown and user interaction as well as all control elements are disabled. When the screenshot is ready, a menu appears to ask for the further procedure. The screenshot could be simply discarded, saved to the native Photo app or sent by email. If the email option was selected, a `MFMailComposeViewController` from Section 2.4.5 is activated that shows an email form similar to the iOS® Email app. Then, the screenshot has been already attached and the subject has been entered. Next, the email is scheduled in the email outbox. After all, an appropriate pop-up dialog will indicate the success or failure of the chosen operation.

3.9.8. Settings Screen

The visible appearance of the settings screen is controlled by `BCiSettingsScreenViewController`. It uses a scrollable `UITableView` to organize the layout and structure of all settings. Most of the rows are based on customized built-in table cells but also nib files were created to layout cells containing scaling related settings. "Close" button in the navigation bar is responsible of dismissing the settings screen.

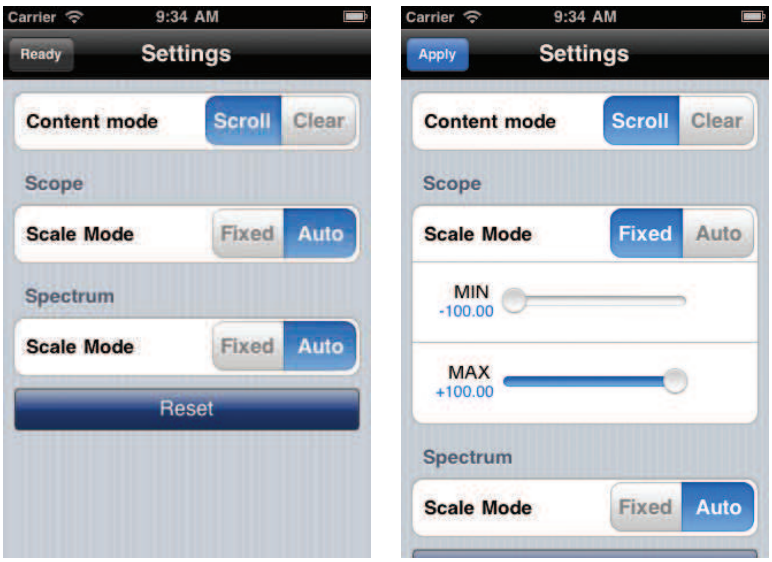
All control elements are initialized with the current `BCiApplicationSettings`. In the first instance, any changes are made to a local copy in order to allow to restore the original settings. If the settings were altered and no longer equal the original settings, "Close" button will turn to a blue "Apply" button which is reversible if this condition will be true again. For this purpose, a "Reset" button is attached to the end of the table. Taping this button will automatically restore the original settings immediately. In contrast, if "Apply" button was hit, settings will be persisted to be still available after the application has been terminated. Furthermore, settings screen will be finally dismissed. For this purpose, also the local settings are synchronized with the user's default database.

The following settings are configurable by settings screen in Figure 3.38:

- *Content mode* determines the visualization mode for all periodic graphs (spectogram- and scope graphs). Thereby, either "Scroll" or "Clear" are possible which correspond to visualizations modes of continuous content layers in Section 3.8.1. In order to

ensure that all periodic graphs are synchronized, only one option for both type of periodic graphs is provided.

- *Scale mode* is individually configurable for both types of periodic graphs. If "Auto" is specified then all associated `BCiGraphViewControllers` will automatically scale values of their graphs. In this manner, represented maximum and minimum will correspond to effectivly measured maximum and minimum. This is useful to fully exploit the height of a scope graph or color range of a spectogram graph. On the other hand, "Fixed" will allow to specify the value limits manually by two `BCiSliders` for each graph type. The Maximum- and minimum slider are bound to each other and will update its ranges if the respective other has been manipulated. This approach prevents illegal and contradicting values like setting a maximum to a lower value than the configured minimum.



(a) Collapsed settings (b) Expanded settings

Figure 3.38.: Screenshots: settings screen

3.9.9. Performance Limits

Technically, as a result of using a floating point coordinate system, any number of data points could be visualized. However, computational power is a restraining factor though. Running iScope with Instruments on an iPhone® 3GS showed that workload is primarly shared between receiving data packets from SignalServer, extracting data samples and

data visualization. The former two tasks are consuming approximately ¼ and the latter task uses about ⅝ of CPU time depending on simultanously transmitted signal channels and displayed graphs per screen - *ten channels were transmitted with a virtual sampling rate of 150 Hz, whereas three scopes were displayed on the same screen.* Consequently, these three tasks have to be well balanced to guarantee iScope is optimally working. This can be already done by configuring SignalServer to use lower virtual sampling rates and applying a layout with less graphs per screen.

Fortunately, screen resolution is also constrained and thus, sampling rates are especially limited by being still able to optically distinct between rendered data points. An iPhone® 3GS displays graphs with content areas that feature a maximum width of 296 or 456 pixels according to the device orientation.

If a scope graph view that records ten seconds is assumed. Then, the maximum representable sampling rate in the content area amounts 30 Hz respectively 45 Hz whereas every data point is at least one pixel in width. In theory, a smaller width is possible but would cause indistinguishable or overlapping data points. When considering two seconds, then 148 Hz and 228 Hz respectively are the upper bounds. Displaying only one second would be less reasonable because the display changes too fast to recognize any trends or artifacts in the signal. Despite the one pixel minimum width, still a signal with ten channels can be transmitted at 250 Hz while three channels are visualized in parallel on one screen. Thereby, iScope operates fluently and the user interface keeps responsive. In this case, even if data is not tracked, as a result of rendering temporary data points in a reduced representation, progressive scaling is still guaranteed for the entire ten seconds or 7500 data points (which are spread over three graphs).

Considering the initial position from above, a maximum of 30 respectively 45 FFT cycles per second can be reasonably visualized in a spectogram graph. Viewed realistically, a width of two pixels has to be used in order to avoid blurring that reduces the maximum effective cycle amount by further 50%. Furthermore, representing a quite smaller time window in the graph is not advisable to ensure keeping track of changes in the spectral domain. Test runs on an iPhone® 3GS revealed exemplary maximum configurations from Table 3.8. Deriving from these results, if a high number of cycles per second is required, only one graph per screen should be configured to optimize performance results.

Generally, performing test runs of iScope demonstrated that the device's maximum resolution could be utilized at an adequate performance level.

Virtual Sampling rate	Transmitted Channels	Graphs per screen	FFT cycles per second
250 Hz	10	1	45
250 Hz	10	3	20
150 Hz	10	3	25

Table 3.8.: Exemplary maximum configurations for spectogram graph - every configuration uses a Hamming window of one second.

4. Deployment

Enrolling iOS® developer program is a prerequisite and cannot be avoided in an official way. There exist other inofficial deployment scenarios but they are not part of this thesis.

An app has to be cryptographically signed with a certificate in order to run on real iOS® devices. The process of how to deploy apps depends on the current project phase which can be either development & testing or distribution. Both phases require having access to iOS® provisioning portal where different administrative tasks are organized (e.g., managing certificates, registering iOS® devices, creating provisioning profiles etc.). In this context, iOS® developers have different roles and responsibilities. Basically, administrative tasks are assigned to *team admins* and *team agents* rather than to simple *team members*.

4.1. Building from Source

At the time, the entire source code of iScope is under source control and located at `https://svn.tugraz.at/svn/iscope/trunk`. After checking out the Subversion (SVN) repository, `BCiScope.xcodeproj` has to be opened with XCode® and the active build configuration has to be specified. *Debug* should be used when iScope is deployed in development & testing phase. For the distribution phase, it is recommanded to clone the *Release* configuration and rename it according to the distribution channel (e.g., *Ad-Hoc Distribution*). In both cases, an appropriate provisioning profile has to be selected for code signing, otherwise the build will not successfully run to completion. *Debug* builds are also fully functional and run on real devices. Finally, either *Device* or *Simulator* has to be selected indicating the target platform for the app.

When the build has started, all required sub projects (i.e. libraries) are generated as well. Neither additional binaries and external libraries have to be installed, nor any further configuration beyond XCode® has to be done. All necessary files are included in specific sub folders of the project directory.

Builds are initiated using the *Build* menu in XCode®. XCode® will create a `build` directory in the `BCiScope` project directory containing a sub directory for the active build environment (combination of active build configuration and target platform - e.g., `Release-iphoneos`). If *Build* option was selected from the *Build* menu, the outcome of the build process is a named folder called the application bundle (`BCiScope.app`) containing all resources and binaries of the application. If *Build and Archive* was chosen, the application bundle is packed into a `.ipa` archive and no provisioning profile (`.mobileprovision`) has to be additionally provided which is more practical for distribution.

4.2. Development & Testing

Every developer or *team member* owns an iOS® development certificate for signing. In the provisioning portal, a dedicated development provisioning profile has to be created which comprises a list of developers allowed to sign a set of apps identified by an *App ID* for particular devices. The app will not run if the development certificate is invalid or the developer, app or device does not match the configuration in the provisioning profile. If all requirements are fulfilled, iScope can be built from source (see Section 4.1) by means of XCode®. Then, iScope will start immediately after XCode® has installed all files (including the provisioning profile) on the physically connected device.

4.3. Distribution

This phase requires a dedicated distribution provisioning profile which is similar to a development provisioning profile. It includes a distribution certificate for digitally signing the resulting binary to run on iOS® devices. The binary itself can be built with XCode® by following steps in Section 4.1.

Distributables are either ad-hoc (e.g., for test users), in-house (for companies) or targeting App Store℠.

4.3.1. Ad-Hoc Distribution

This type of distribution allows apps to be installed on specific devices listed in the provisioning profile. The Standard iOS® developer program has a limitation of up to 100 devices. If no `.ipa` file was generated, the provisioning profile file (`.mobileprovision`) has to be delivered in addition to the application bundle.

The distributer is free in deciding how distribution files are provided to users. In order to install an app ad-hoc, all distribution files have to be dragged into iTunes® and synchronized with the connected iOS® device.

Online services like TestFlight emerged that allow easier, over-the-air distribution of iOS® binaries for beta testing. After creating a free account, `.ipa` files have to be uploaded to the portal and TestFlight takes care of distributing the app to registered users. These users will be notified by email if a new binary is available. Users can directly open the email notification on the iOS® device and install the app without the need of iTunes®. [81]

4.3.2. In-House Distribution

The Enterprise iOS® developer program allows distribution of archived apps without App Store℠ and the necessity of specifying particular devices which are allowed to run these apps. After all, apps are installed with iTunes® similar to ad-hoc distribution.

4.3.3. App Store℠

No devices or users have to be explicitly specified in the distribution provisioning profile. In order to upload an app to the App Store℠, an iTunes® Connect account is required. Next, an entry for iScope has to be created in iTunes® Connect. Then, `BCiScope.app` has to be compressed and uploaded to iTunes® Connect with Application Loader which is shipped with iOS® SDK. Once the app is validated by Apple®, the application is ready for download via the App Store℠.

4.4. Licensing

Basically licensing is an important topic because *"if you don't license your code, it can't be used (legally) by other people"* [82]. Anyway, since iScope only aggregates software components that are licensed under the X Version 11 (X11) license (TinyXML++ library), the Boost Software license (Boost library) and the modified BSD license (TiA library client will supposably use this license in the future), it is also possible to use a proprietary license that omits sharing the source code. Bugs tend to persist shorter in open and frequently updated codebases. For this reason, a free software license or open source software license is contemplable. [83].

4.4.1. Licensing Limitations

Careful considerations need to be taken into account if an open source or free software license is chosen in conjunction with iTunes Store® as distribution channel. When buying and installing an app via iTunes Store®, even without charge, every user has to aggree to Terms of Service (ToS) of iTunes Store® respectivly ToS of App Store℠. GNU Go [84] and Applidium VLC [85], two famous examples, demonstrate why this is an important issue. Both apps were licensed under the GPL version 2 (GPLv2). In either cases, the app was actually removed from iTunes Store® due to the fact that the GPLv2 is legally incompatible with those ToS [86, 87, 88]. More precisly, section 6 of the GPLv2 contains the following condition guarded by a strong copyleft clause:

> *"Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein."* [89]

At the time when Apple® redistributes GPL licensed apps through their App Store℠ to end customers, the ToS will violate this condition. The violation results from limiting the customer's freedom in usage and distribution of the app, granted through *Usage Rules* in their ToS which every customer has to apply to and that is representing this kind of *further restrictions*. [86, 87]

As stated by the free software foundation, creator of the GPL, this problem also applies to all other versions of the GNU GPL and the GNU Affero General Public License (AGPL) as well [86, 89].

Nevertheless, strong belivers in free software could enforce the use of a copylefted license by establishing a dual licensing scheme within their projects. In this manner, the app is licensed under a lax permissive license (e.g., X11 license) in order to avoid legal violations and a license with a copyleft clause (e.g., GNU GPLv3) for the free software and open source community. The former one should not be given away from the project's copyright holder to prevent turning the project into proprietary software. A major disadvantage is that utilizing contributions from the communities for official distribution channels (e.g., App Store℠), legally always involves forcing every developer to formally assign copyrights to the copyright holder of the project. This can lead to an impracticable effort for larger projects with a huge number of already existing contributors. In addition, some developers might mind contributing to two seperate legal entities where the copyright

holder grants himself special rights. Another downside is that dual licensing would not solve the problem entirely, as one of the main intentions of free software, conceding the right to copy, modify and redistribute the software, does not apply to end customers in the same way as it does to the copyright holder of the project. Although they could copy and modify the copylefted code, they still would not be able to distribute a modified copylefted version of the app through App Store℠ [90, 91].

5. Summary and Conclusion

iScope has reached a state considerable as functional and ready for beta testing. For this purpose, ad-hoc distributions have been built, successfully installed and passed common use case tests on real iPhone® and iPad® devices. Typically, beta versions enable all fundamental features but can contain also a couple of minor issues.

Communication with SignalServer is possible in order to start and stop data transmission of different types of biosignals at sufficient sampling rates as described in Section 3.6.3.2. As a valueable result of the module architecture, modifying or exchanging the client part will only have local impacts to *Signal Server Client* module and will not affect the main application or other modules. This is especially important in conjunction with selective channel transmission which is a future feature of SignalServer.

Three types of graphs are implemented to display data of independently selectable channels. Periodic biosignals are represented online in line graphs sample by sample and/or as the result of STFT in spectogram graphs. Also a generic graph was implemented that shows the state of an aperiodic signal such as a button.

Periodic graphs can be visualized in two different modes - data is either drawn at the position of a vertical bar that repeatedly runs from the left to the right edge of the display; or continuously on the very right position of a virtually infinite, scrolling area.

Furthermore, graphs are automatically arranged by a configurable, generic layout manager that respects the current spatial orientation of the device, let the user magnify graphs to show a detailed view highlighting more information and manipulate the ordering of graphs. This approach allows to have a dynamic, configurable number of graphs per screen as well as embedding new types of graphs very easily without touching any line of layout specific code.

Many other parameters used by iScope are also adaptable by means of a settings screen which is directly integrated in the app's workflow or by an entry in the native iOS® Settings app. For instance, every graph is scaleable in two dimensions (value- and time dimension) with either of two scale modes disregarding data is simultanously transmitted.

For this purpose, an auto scaling feature could be used to fit values into their ideal boundaries. Otherwise, the value range could be manually fixed using a nice user interface resembling the look and feel of iOS® and its core applications.

In this manner, graphs can be reordered in a similar way to apps in the native app launcher application. After all, just a few common and individual gestures involving one up to two fingers are enough to handle every use case.

Finally, as a consequence of selective measures such as lazy update of invisible graphs, reducing the rendering quality of temporary visible graph content and exploiting fast graphic cache, it was feasible to achieve aimed performance requirements as mentioned in Section 3.9.9.

6. Outlook

In the next step, beta testing of the ad-hoc distribution should be done in real world scenarios (i.e. BCI measurement with non developer user and a human subject) and fixing all possibly discovered issues to be ready to potentially release the binary on App Store℠.

The current version of iScope is not already tailored for retina displays but would run on those devices anyhow. View controllers of the *Graphs* Module have to optimize the aggregate weight parameter(s) for retina displays (or similar potential future display enhancements) as described in Section 3.8.1.

The entire application is managed by XCode® 3.2 projects. However, iScope can be built with XCode® 4 in compatibility mode. Nevertheless, the application should be ported to native XCode® 4 projects which should be a straight forward process.

In the future, issues regarding too high sampling rates to reasonably represent on constrained iOS® device screens will be eliminated by a downsampling module integrated into SignalServer. Additionally, performance will be increased by making use of selective channel transmission which is also a future feature of SignalServer. iScope's selection screen and *Signal Server Client* module already respect this feature which will reduce the current communication- and parsing load.

There is already a feature request that lies beyond the scope of this thesis. Spectogram graphs or more precisely their view controllers should support the possibility to scroll and scale the range of displayed frequencies. This poses the problem of how to control these use cases by user interaction. The swipe gesture is already assigned to scrolling between different pages respectively graphs as well as the pinch gesture that is used to scale time- and value dimension depending on the orientation of the pinch. A two-finger swipe would be a reasonable workaround for the first use case. However, this will not work for scaling as pinching with more fingers would be cumbersome. Consequently, at least for that use case, a new gesture or workflow extension has to be considered to determine which dimension shall be scaled. Another approach might allow these new use

cases only when a graph is magnified whereas scrolling between graphs and pages is not possible by design and both gestures are unassigned.

Bibliography

[1] S. Deutsch and A. Deutsch. *Understanding the nervous system: an engineering perspective*. IEEE Press, New York, 1993.

[2] A. Zani and A. Proverbio. *The cognitive electrophysiology of mind and brain*. Academic Press, 2002.

[3] W. Tatum, A. Husain, S. Benbadis, and P. Kaplan. *Handbook of EEG interpretation*. Demos Medical Pub, New York, 2008.

[4] R. Srinivasan. Methods to improve the spatial resolution of EEG. *International Journal of Bioelectromagnetism*, 1(1):102–111, 1999.

[5] M. Slater, A. Schlögl, and G. Pfurtscheller. Presence research and eeg. In *Proceedings of PRESENCE 2002: The 5th Annual International Workshop on EEG Presence*, 2002.

[6] E. Niedermeyer. Alpha rhythms as physiological and abnormal phenomena. *International Journal of Psychophysiology*, 26(1-3):31–49, 1997.

[7] G. Pfurtscheller and F. H. Lopes da Silva. Event-related EEG/MEG synchronization and desynchronization: basic principles. *Clinical Neurophysiology*, 110(11):1842–1857, November 1999.

[8] Welcome to TOBI — TOBI : Tools for brain-compuer interaction. `http://www.tobi-project.org`. [online; accessed 19-Mar-2012].

[9] European commission: Cordis: FP7 : Home. `http://cordis.europa.eu/fp7`, 2012. [online; accessed 20-Mar-2012].

[10] tools4bci ... TiA. `http://tools4bci.sourceforge.net/tia.html`, 2012. [online; accessed 24-June-2012].

[11] C. Breitwieser, I. Daly, C. Neuper, and G. R. Müller-Putz. Proposing a standardized protocol for raw biosignal transmission. *Biomedical Engineering, IEEE Transactions on*, 59(3):852 –859, march 2012.

[12] tools4BCI. `http://tools4bci.sourceforge.net`. [online; accessed 19-Mar-2012].

[13] J. R. Wolpaw, N. Birbaumer, D. J. McFarland, G. Pfurtscheller, and T. M. Vaughan. Brain-computer interfaces for cmmunication and control. *Clinical Neurophysiology*, 113(6):767 – 791, 2002.

[14] A. Kreilinger, V. Kaiser, C. Breitwieser, J. Williamson, C. Neuper, and G. R. Müller-Putz. Switching between manual control and brain-computer interface using long term and short term quality measures. *Frontiers in Neuroscience*, 5(00147), 2012.

[15] G. Pfurtscheller, B. Z. Allison, G. Bauernfeind, C. Brunner, T. S. Escalante, R. Scherer, T. O. Zander, G. R. Müller-Putz, C. Neuper, and N. Birbaumer. The hybrid BCI. *Frontiers in Neuroscience*, (00003), 2010.

[16] G. R. Müller-Putz, C. Breitwieser, F. Cincotti, R. Leeb, M. Schreuder, L. Leotta, M. Tavella, L. Bianchi, A. Kreilinger, A. Ramsay, M. Rohm, M. Sagebaum, L. Tonin, C. Neuper, and J. D. R. Millan. Tools for brain-computer interaction: a general concept for a hybrid BCI (hBCI). *Frontiers in Neuroinformatics*, 5(00030), 2011.

[17] M. Fatourechi, A. Bashashati, R. K. Ward, and G. E. Birch. EMG and EOG artifacts in brain computer interface systems: a survey. *Clinical Neurophysiology*, 118(3):480 – 494, 2007.

[18] N. A. Chadwick, D. A. McMeekin, and T. Tan. Classifying eye and head movement artifacts in EEG signals. In *Digital ecosystems and technologies conference (DEST), 2011 proceedings of the 5th IEEE international conference on*, pages 285–291, 31 2011-june 3 2011.

[19] SigViewer. `http://sigviewer.sourceforge.net`. [online; accessed 19-Mar-2012].

[20] A. V. Oppenheim and R. W. Schafer. *Discrete-time signal processing*. Prentice Hall, London, 0003. aufl. edition, 2010.

[21] R. G. Lyons. *Understanding digital signal processing*. Addison Wesley Pub. Co, Reading, Mass, 1997.

[22] iOS Application Programming Guide. http://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/iPhoneAppProgrammingGuide.pdf. [online; accessed 24-May-2011].

[23] iOS Technology Overview. http://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/iPhoneOSTechOverview.pdf. [online; accessed 10-April-2011].

[24] Event Handling Guide for iOS. http://developer.apple.com/library/ios/documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/EventHandlingiPhoneOS.pdf. [online; accessed 24-May-2011].

[25] D. Mark and J. LaMarche. *Beginning iPhone 4 development - exploring the iOS SDK*. Apress Distributed by Springer-Verlag, 2011.

[26] D. Mark and J. LaMarche. *Beginning iPhone development - exploring the iOS SDK*. Apress Distributed by Springer-Verlag, 2009.

[27] Instruction Set Architectures - ARM. http://www.arm.com/products/processors/technologies/instruction-set-architectures.php. [online; accessed 4-June-2011].

[28] P. Lebeaupin. A few things iOS developer ought to know about the ARM architecture - Wandering Coder. http://wanderingcoder.net/2010/07/19/ought-arm/. [online; accessed 3-June-2011].

[29] What processor do the iPod, iPod mini, iPod nano, iPod touch, and iPod shuffle models use? @ EveryiPod.com. http://www.everymac.com/systems/apple/ipod/ipod-faq/ipod-processor-type-portal-player-samsung.html. [online; accessed 3-June-2011].

[30] iPod Touch 1st Generation Teardown - iFixit. http://www.ifixit.com/Teardown/iPod-Touch-1st-Generation-Teardown/596. [online; accessed 3-June-2011].

[31] iPod Touch 2nd Generation Teardown - iFixit. http://www.ifixit.com/Teardown/iPod-Touch-2nd-Generation-Teardown/586. [online; accessed 3-June-2011].

[32] iPod Touch 3rd Generation Teardown - iFixit. http://www.ifixit.com/Teardown/iPod-Touch-3rd-Generation-Teardown/1158. [online; accessed 3-June-2011].

[33] iPod Touch 4th Generation Teardown - iFixit. http://www.ifixit.com/Teardown/iPod-Touch-4th-Generation-Teardown/3562. [online; accessed 3-June-2011].

[34] Apple iPhone Specifications — TheUnlockr. http://theunlockr.com/2011/05/22/apple-iphone-specifications/. [online; accessed 3-June-2011].

[35] Apple iPhone 3G Specifications — TheUnlockr. http://theunlockr.com/2011/05/22/apple-iphone-3g-specifications/. [online; accessed 3-June-2011].

[36] Apple iPhone 3GS Specifications — TheUnlockr. http://theunlockr.com/2011/05/22/apple-iphone-3gs-specifications/. [online; accessed 3-June-2011].

[37] Apple iPhone 4 Specifications — TheUnlockr. http://theunlockr.com/2011/05/22/apple-iphone-4-specifications/. [online; accessed 3-June-2011].

[38] Apple iPad 3G - Complete Specifications - The Phone Database. http://www.thephonedatabase.com/Apple_iPad_3G_339_Cell_Phone. [online; accessed 3-June-2011].

[39] Apple iPad 2 3G - Complete Specifications - The Phone Database. http://www.thephonedatabase.com/Apple_iPad_2_3G_443_Cell_Phone. [online; accessed 3-June-2011].

[40] NEON - ARM. http://www.arm.com/products/processors/technologies/neon.php. [online; accessed 2-June-2011].

[41] Floating Point - ARM. http://www.arm.com/products/processors/technologies/vector-floating-point.php. [online; accessed 2-June-2011].

[42] Cortex-A8 Technical Reference Manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/DDI0344K_cortex_a8_r3p2_trm.pdf. [online; accessed 3-June-2011].

[43] P. Lebeaupin. Introduction to NEON on the iPhone - Wandering Coder. http://wanderingcoder.net/2010/06/02/intro-neon/. [online; accessed 3-June-2011].

[44] iOS Development Guide. http://developer.apple.com/library/ios/documentation/Xcode/Conceptual/iphone_development/iOS_Development_Guide.pdf. [online; accessed 10-April-2011].

[45] The LLVM Compiler Infrastructure Project. http://llvm.org. [online; accessed 12-Sep-2011].

[46] "clang" C language family frontend for LLVM. http://clang.llvm.org. [online; accessed 12-Sep-2011].

[47] LLVM Compiler Overview. http://developer.apple.com/library/ios/#documentation/CompilerTools/Conceptual/LLVMCompilerOverview/_index.html. [online; accessed 12-Sep-2011].

[48] The Objective-C Programming Language. http://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf. [online; accessed 12-Sep-2011].

[49] S. Kochan. *Programming in Objective-C 2.0*. Addison Wesley Professional, Upper Saddle River, NJ, 2009.

[50] Cocoa Fundamentals Guide. http://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaFundamentals.pdf. [online; accessed 23-June-2012].

[51] Foundation Framework Reference. http://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/ObjC_classic/FoundationObjC.pdf. [online; accessed 6-June-2011].

[52] UIKit Framework Reference. https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIKit_Framework/UIKit_Framework.pdf. [online; accessed 31-May-2011].

[53] View Programming Guide for iOS. http://developer.apple.com/library/ios/featuredarticles/ViewControllerPGforiPhoneOS/ViewControllerPGforiPhoneOS.pdf. [online; accessed 12-Sep-2011].

[54] Core Animation Programming Guide. http://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/CoreAnimation_guide/CoreAnimation_guide.pdf. [online; accessed 22-June-2012].

[55] E. Freeman. *Head first design patterns*. O'Reilly, 2004.

[56] Cocoa Application Competencies for iOS: Main event loop. http://developer.apple.com/library/ios/documentation/General/Conceptual/Devpedia-CocoaApp/MainEventLoop.html. [online; accessed 26-May-2011].

[57] Threading Programming Guide. https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/Multithreading/Multithreading.pdf. [online; accessed 26-May-2011].

[58] E. Sadun. *The iPhone developer's cookbook - building applications with the iPhone SDK*. Pearson Education, 2008.

[59] vDSP Programming Guide. http://developer.apple.com/library/ios/documentation/Performance/Conceptual/vDSP_Programming_Guide/vDSP_Programming_Guide.pdf. [online; accessed 26-May-2011].

[60] vDSP Reference. https://developer.apple.com/library/ios/documentation/Accelerate/Reference/vDSPRef/vDSPRef.pdf. [online; accessed 6-June-2011].

[61] Quartz 2D Programming Guide. http://developer.apple.com/library/ios/documentation/GraphicsImaging/Conceptual/drawingwithquartz2d/drawingwithquartz2d.pdf. [online; accessed 22-June-2012].

[62] Boost C++ Library. http://www.boost.org, 2011. [online; accessed 23-June-2011].

[63] ticpp - TinyXML++. http://code.google.com/p/ticpp, 2011. [online; accessed 27-June-2011].

[64] TinyXML Main Page. http://www.grinninglizard.com/tinyxml, 2011. [online; accessed 27-June-2011].

[65] C. Breitwieser and C. Eibel. TiA – documentation of TOBI interface a. *ArXiv e-prints*, March 2011.

[66] C. Breitwieser, C. Eibel, A. Schuller, and M. Rowies. TOBI interface a. http://bci.tugraz.at/TOBI/TiA_public_doc, 2011. [online; accessed 27-June-2011].

[67] C. Breitwieser. Signal Server - User Manual. http://xp-dev.com/sc/browse/84250/, 2011. [online; accessed 27-June-2011].

[68] Three 20. http://new.three20.info, 2011. [online; accessed 27-June-2011].

[69] core-plot - Cocoa plotting framework for Mac OS X and iOS. http://code.google.com/p/core-plot, 2011. [online; accessed 27-June-2011].

[70] Categories of free and nonfree software - GNU Project - Free Software Foundation. http://www.gnu.org/philosophy/categories.html. [online; accessed 10-April-2011].

[71] The Open Source Definition — Open Source Initiative. http://www.opensource.org/docs/osd. [online; accessed 10-April-2011].

[72] Various Licenses and Comments about Them - GNU Project - Free Software Foundation. http://www.gnu.org/licenses/license-list.html. [online; accessed 20-April-2011].

[73] Selling free software - GNU Project - Free Software Foundation. http://www.gnu.org/philosophy/selling.html. [online; accessed 28-April-2011].

[74] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[75] gabriel/gh-unit - GitHub. https://github.com/gabriel/gh-unit, 2011. [online; accessed 19-May-2011].

[76] Sen:te - OCUnit. http://www.sente.ch/software/ocunit, 2011. [online; accessed 19-May-2011].

[77] J. Gaspar. Templated Circular Buffer Container - Boost 1.44.0. http://www.boost.org/doc/libs/1_44_0/libs/circular_buffer/doc/circular_buffer.html, 2011. [online; accessed 19-May-2011].

[78] Qt - A cross platform application and UI framework. http://qt.nokia.com, 2012. [online; accessed 30-June-2012].

[79] Qt 4.7: qmake Manual. http://doc.qt.nokia.com/4.7/qmake-manual.html, 2012. [online; accessed 30-June-2012].

[80] R. Gonzalez. *Digital image processing*. Prentice Hall, Upper Saddle River, N.J, 2008.

[81] TestFlight - iOS beta testing on the fly. https://testflightapp.com/. [online; accessed 21-June-2011].

[82] V. Lindberg. *Intellectual property and open source*. O'Reilly, 2008.

[83] D. Cooper, C. DiBona, and M. Stone. *Open sources 2.0*. O'Reilly, 2005.

[84] GNU Go - GNU Project - Free Software Foundation (FSF). http://www.gnu.org/software/gnugo. [online; accessed 1-July-2012].

[85] Applidium - VLC. http://applidium.com/en/applications/vlc. [online; accessed 1-July-2012].

[86] B. Smith. More about the App Store GPL Enforcement – Free Software Foundation. http://www.fsf.org/blogs/licensing/more-about-the-app-store-gpl-enforcement, 2010. [online; accessed 10-April-2011].

[87] iTUNES STORE - TERMS AND CONDITIONS. http://www.apple.com/legal/itunes/us/terms.html. [online; accessed 11-April-2011].

[88] Vaughan-Nichols, S. J. No GPL Apps for Apple's App Store — ZDNet. http://www.zdnet.com/blog/open-source/no-gpl-apps-for-apples-app-store/8046, 2011. [online; accessed 13-April-2011].

[89] A. M. S. Laurent. *Understanding open source & free software licensing*. O'Reilly, 2004.

[90] K. Fogel. *Producing open source software: how to run a successful free software project*. O'Reilly, 2005.

[91] S. Walli. Open Minded: Solving the Apple App Store Incompatibility with the GPL. http://www.networkworld.com/community/blog/solving-apple-app-store-incompatibility-gpl, 2011. [online; accessed 10-April-2011].

A. Appendix

A.1. Benchmarking BCiDataBuffer with BCiCommonBenchmarks

Table A.1, Table A.2 and Table A.3 show the results of `BCiCommonBenchmarks` as described in Section 3.5.1.1. Every instance of a task was executed 1000 times to average the final results. The benchmark was compiled with LLVM GCC 4.2 and ran on a real iPhone 3GS with iOS® 4.3.1.

- *Adding task* carries out adding sample values to the ring buffer twice as much the buffer size. In this regard, new values are inserted into `NSMutableArray` either at the tail (forward version) or at the front (backward version). The forward version tests `NSNumber` as well as a custom mutable class to wrap values. In the course of applying a custom wrapper, added objects are reused as the maximum number of elements has been reached to avoid unnecessary heap allocations. If the buffer was already full, `NSMutableArray` and `std::deque` have to remove an element from the buffer before inserting another one.

- *Traversing task* steps through and actually accesses all elements of the buffer. `std::deque` accesses the elements either with the *STL iterator class* or with an appropriate random access method. `NSMutableArray` is benchmarked with *fast enumerations* and also a proper random access method.

- *Copying task* copies the whole buffer to a C-array of equal length. This involves iterating through all elements of the buffer if `NSMutableArray` or `std::deque` are used.

Adding				
Buffer size	BCiDataBuffer	std::deque	NSMutableArray (forward)	NSMutableArray (backward)
10^2	0.082 ms	−12.50%	+1287.50% [1)] +2312.50% [2)]	+1362.50% [1)]
10^3	0.069 ms	−34.78%	+1576.81% [1)] +2707.25% [2)]	+1550.73% [1)]
10^4	0.649 ms	−8.01%	+1650.38% [1)] +2869.34% [2)]	+1630.82% [1)]
10^5	6.027 ms	+13.04%	+1792.43% [1)] +3063.88% [2)]	+1825.17% [1)]

Table A.1.: Ring buffer benchmark results: adding task - [1)] with NSNumber; [2)] with custom wrapper class

Traversing				
Buffer size	BCiDataBuffer	std::deque	NSMutableArray (forward) [*)]	NSMutableArray (backward) [*)]
10^2	0.007 ms	−14.29% [1)] +114.29% [2)]	+0.00% [3)] +14.29% [2)]	+28.57% [2)]
10^3	0.046 ms	+26.09% [1)] +243.48% [2)]	+19.57% [3)] +52.17% [2)]	+54.35% [2)]
10^4	0.464 ms	+22.63% [1)] +242.67% [2)]	+15.73% [3)] +48.92% [2)]	+65.09% [2)]
10^5	4.504 ms	+25.56% [1)] +256.88% [2)]	+21.20% [3)] +58.80% [2)]	+57.60% [2)]

Table A.2.: Ring buffer benchmark results: traversing task - [1)] with STL iterator; [2)] with random access method; [3)] with fast enumeration; [*)] with NSNumber wrapper

Copying				
Buffer size	BCiDataBuffer	std::deque	NSMutableArray (forward) *)	NSMutableArray (backward) *)
10^2	0.001 ms	+500.00%	+800.00%	+700.00%
10^3	0.002 ms	+2750.00%	+3600.00%	+3550.00%
10^4	0.009 ms	+6333.33%	+7966.67%	+8733.33%
10^5	0.113 ms	+5038.94%	+6404.42%	+6434.51%

Table A.3.: Ring buffer benchmark results: copying task - *) with NSNumber wrapper

A.2. Building Boost Libraries for iOS®

No individual XCode project is required to build the Boost libraries for iOS®. Instead, Boost (version 1.44.0) uses Jam (similar to `Make`) as a build tool to compile each libary separately. These are specifically Boost.System and Boost.Date_Time in respect to iScope. All other required functionality can be obtained by simply including corresponding Boost header files. Jam faciliates customization of all tools that are applied during the build process (i.e. specifying iOS® SDK compilers). First of all, in order to inject a compiler of the iOS® SDK to Jam, it is necessary to edit `user-config.jam` which has to be explicitly created. It should be located either in the home directory (`$HOME`) or the Boost build directory (`$BOOST_BUILD_PATH`). Basically, this file is read on startup of Boost.Build and allows to define additional compilers and other tools that should be considered. Every library has to be built thrice, targeting i386 (for iOS® Simulator), ARM®v6 (for older iOS® devices) and ARM®v7 (for newer iOS® devices). Thereby, the ARM®v6 build omits the Thumb compiler flag. After bootstrapping Boost (see the Boost documentation) and entering the Boost directory, every build is done in two steps: editing `user-config.jam`, then invoking Jam. It is required to rename the output file after each build step. Listing A.2 shows the Jam configuration targeting ARM®v6, Listing A.1 for ARM®v7 and Listing A.3 for iOS® Simulator.

```
using darwin : 4.2.1~iphone
 : <PathToSDK>/iPhoneOS.platform/[...]/llvm-gcc-4.2
   -arch armv7 -mthumb
   -fvisibility=hidden -fvisibility-inlines-hidden
 : <striper>
 : <architecture>arm <target-os>iphone
 ;
```

Listing A.1: Boost user-config.jam - ARM®v7

```
using darwin : 4.2.1~iphone
 : <PathToSDK>/iPhoneOS.platform/[...]/llvm-gcc-4.2
   -arch armv6 -fvisibility=hidden -fvisibility-inlines-hidden
 : <striper>
 : <architecture>arm <target-os>iphone
 ;
```

Listing A.2: Boost user-config.jam - ARM®v6

```
using darwin : 4.2.1~iphonesim
 : <PathToSDK>/iPhoneSimulator.platform/[...]/llvm-gcc-4.2
   -arch i386 -fvisibility=hidden -fvisibility-inlines-hidden
 : <striper>
 : <architecture>x86 <target-os>iphone
 ;
```

Listing A.3: Boost user-config.jam - iOS® Simulator

Finally, the libraries can be created by calling Jam as demonstrated in Listing A.4 for ARM®v6 and ARM®v7 devices. Creating iOS® Simulator libraries make use of Jam as shown in Listing A.5. Thereby, `<iOSVersion>` specifies the target iOS® version. Unless the corresponding iOS® SDK has been installed, building will not work.

```
./bjam toolset=darwin architecture=arm target-os=iphone
       macosx-version=iphone-<iOSVersion> define=_LITTLE_ENDIAN
       link=static install
```

Listing A.4: Boost Jam invocation targeting iOS® devices

```
./bjam toolset=darwin architecture=x86 target-os=iphone
       macosx-version=iphonesim-<iOSVersion> link=static install
```

Listing A.5: Boost Jam invocation targeting iOS® Simulator

The ARM®v6 and iOS® Simulator builds will result in libraries (`libboost_date_time.a`, `libboost_system.a`) containing code for one single architecture. In contrast, the ARM®v7 build produces fat libraries targeting ARM®v7- and ARM®v6 architectures. Unfortunately, these ARM®v6 contents are generated using the Thumb flag because the ARM®v7 options are used. However, `lipo`, a command line tool available under Mac OS X® which creates or operates on universal (multi-architecture) files (e.g., libraries), is able to bypass this issue. The tool can be used to produce fat libraries for Boost.System and Boost.Date_Time by merging the libraries of the different builds together (see Listing A.6). After all, the library files contain all three target architecture types (ARM®v6, ARM®v7 and i386) and correct corresponding contents.

```
<...>$ ls -1
libboost_date_time.a
libboost_date_time_arm6_build.a
libboost_date_time_arm7.a
libboost_date_time_arm7_build.a
libboost_date_time_simulator_build.a
libboost_system.a
libboost_system_arm6_build.a
libboost_system_arm7.a
libboost_system_arm7_build.a
libboost_system_simulator_build.a
<...>$ lipo -info libboost_system_arm6_build.a
input file libboost_system_arm6_build.a is not a fat file
```

```
Non-fat file: libboost_system_arm6_build.a is architecture:
armv6
<...>$ lipo -info libboost_system_simulator_build.a
input file libboost_system_simulator_build.a is not a fat file
Non-fat file: libboost_system_simulator_build.a is architecture:
i386
<...>$ lipo -info libboost_system_arm7_build.a
Architectures in the fat file: libboost_system_arm7_build.a are:
armv7 armv6
<...>$ lipo -thin armv7 libboost_system_arm7_build.a
           -output libboost_system_arm7.a
<...>$ lipo -info libboost_system_arm7.a
input file libboost_system_arm7.a is not a fat file
<...>$ lipo -create libboost_system_arm6_build.a
                   libboost_system_arm7.a
                   libboost_system_simulator_build.a
           -output libboost_system.a
<...>$ lipo -info libboost_system.a
Architectures in the fat file: libboost_system.a are:
armv6 armv7 i386
<...>$ lipo -thin armv7 libboost_date_time_arm7_build.a
           -output libboost_date_time_arm7.a
<...>$ lipo -create libboost_date_time_arm6_build.a
                   libboost_date_time_arm7.a
                   libboost_date_time_simulator_build.a
           -output libboost_date_time.a
```

Listing A.6: Creating fat (universal) Boost libraries - (formatted output)

Fat library- and header files for Boost.System and Boost.Date_Time are located in `trunk/extern/lib`. However, these libraries could be rebuilt with a different compiler (flags) or targeting other (additional) architectures and therefore exchanged in the future.

A.3. Performance Overview of Signal Server Client Module

Table A.4, Table A.5, Table A.6, Table A.7, Table A.8 and Table A.9 list detailed performance results for the tests described in Section 3.6.3.2.

Sampling rate [Hz]	Mean loss [Packets]	SD	Absolute loss [%]
32	0.028	0.314	2.723
64	0.030	0.461	2.903
128	0.023	0.573	2.276
256	0.037	1.000	3.528
512	0.024	1.102	2.316
600	0.025	1.225	2.408
650	0.001	0.024	0.057
700	0.025	1.351	2.400
800*)	7.128	125.967	87.695

Table A.4.: Packet loss: 1 channel - calculated packet loss based on up to 100000 successfully received data packets, except for *) where data transmission was aborted after ten minutes due to a dramatically increase of data loss. data packets were only read from network socket but not processed.

Sampling rate [Hz]	Mean loss [Packets]	SD	Absolute loss [%]
32	0.032	0.340	3.105
64	0.030	0.460	2.925
128	0.013	0.415	1.241
256	0.024	0.793	2.309
512	0.019	1.003	1.902
600	0.032	1.404	3.102
650	0.020	1.121	1.935
700	0.022	1.244	2.121
800 *)	1.806	46.679	64.359

Table A.5.: Packet loss: 1 channel - calculated packet loss based on up to 100000 successfully received data packets, except for *) where data tansmission was aborted after ten minutes due to a dramatically increase of data loss. data packets were read from socket, contents were parsed and added to data model.

Sampling rate [Hz]	Mean loss [Packets]	SD	Absolute loss [%]
32	0.031	0.318	2.965
64	0.024	0.463	2.299
128	0.023	0.568	2.281
256	0.001	0.018	0.034
512	0.029	1.225	2.775
600	0.017	1.009	1.639
650 *)	25.639	461.439	96.245

Table A.6.: Packet loss: 10 channels, 1 selected - calculated packet loss based on up to 100000 successfully received data packets containing ten channels, except for *) where data transmission was aborted after ten minutes to a dramatically increase of data loss. data packets of one channel were only read from network socket but not processed.

Sampling rate [Hz]	Mean loss [Packets]	SD	Absolute loss [%]
32	0.034	0.356	3.325
64	0.024	0.414	2.377
128	0.030	0.631	2.896
256	0.020	0.715	1.923
512	0.020	1.007	1.923
600	0.017	0.999	1.632
650 *)	16.580	391.668	94.311

Table A.7.: Packet loss: 10 channels, 1 selected - calculated packet loss based on up to 100000 successfully received data packets cotanining ten channels, except for *) where data transmission was aborted after ten minutes due to a dramatically increase of data loss. data packets of one channel were read from socket, contents were parsed and added to data model.

Sampling Rate [Hz]	Mean Loss [Packets]	SD	Absolute Loss [%]
32	0.025	0.299	2.395
64	0.019	0.357	1.828
128	0.020	0.527	1.971
256	0.021	0.724	2.020
512	0.020	1.036	1.994
600	0.016	0.990	1.606
650 *)	8.757	375.606	89.749

Table A.8.: Packet loss: 10 channels, 3 selected - calculated packet loss based on up to 100000 successfully received data packets containing three channels, except for *) where data transmission was aborted after ten minutes due to a dramatically increase of data loss. data packets were only read from network socket but not processed.

Sampling rate [Hz]	Mean loss [Packets]	SD	Absolute loss [%]
32	0.019	0.247	1.857
64	0.019	0.367	1.829
128	0.018	0.481	1.738
256	0.018	0.696	1.802
512	0.001	0.025	0.060
600	0.017	1.004	1.648
650 *)	10.036	206.420	90.938

Table A.9.: Packet loss: 10 channels, 3 selected - calculated packet loss based on up to 100000 successfully received data packets containing three channels, except for *) where data transmission was aborted after ten Minutes due to a dramatically increase of data loss. data packets of one channel were read from socket, contents were parsed and added to data model.

A.4. Legal Disclaimer

ARM®, Cortex™, NEON®, Thumb® are trademarks or registered trademarks of ARM Limited. IOS® is a registered trademark of Cisco in the U.S. and other countries and is used under license by Apple Inc. Apple®, MacBook®, Mac®, Mac OS®, OS X®, Objective-C®, Instruments®, iPad®, iPhone®, iPod touch®, iTunes®, iTunes Store®, App Store℠, Cocoa®, Cocoa Touch®, Retina®, Quartz®, XCode® are registered trademarks or service marks by Apple Inc. in the U.S. and other countries. UNIX® is a registered trademark of The Open Group in the U.S. and other countries. NextStep® is a registered trademark of NeXT Software Inc. in the U.S. and other countries.