Control of a multifunction Arm Prosthesis Model

Jørn Bersvendsen

Master of Science in Engineering Cybernetics
Submission date: June 2011
Supervisor: Øyvind Stavdahl, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Control of a multifunction arm prosthesis model

Jørn Bersvendsen

Problem description

The Department of Engineering Cybernetics has an arm/hand with seven motorized degrees of freedom. The model is to be used as a demonstrator and eye-catcher in exhibitions, stands etc. The joints are driven by RC model servos, controlled by a LEGO NXT controller, currently without software.

In this assignment you are to create a control system based on the available hardware, the department's EMG sensor systems and a relevant control algorithm for multifunctional prostheses.

1. Give a short presentation of different signal features used in pattern recognition for myoelectric prosthesis control. Emphasize aspects such as computational complexity, real time aspects and classification accuracy.

2. Describe different relevant software platforms for a potential control system for the given model, and make a justified choice.

3. Establish a functional specification in agreement with the supervisors, and perform a structured software design based on the functional specification.

4. Implement and test the system as far as possible within the allotted time.

Abstract

In this thesis a working control system for a 7 degrees of freedom hand prosthesis model controlled by electromyographic and accelerometer signals has been developed. The complete system consists of a wireless EMG and accelerometer measurement system, two National Instruments data acquisition modules, a desktop computer, a Lego Mindstorms NXT brick and a hand model with 7 motorized degrees of freedom.

The controller is based on pattern recognition and signal classification. Several different EMG features for this purpose are presented and implemented. Two different linear classifiers were used and their performance studied.

The LabVIEW software platform was used for both the computer and the NXT. The developed software has a modular design, facilitating future development and extension. Its design and implementation are presented and discussed.

Preface

This thesis is submitted in fulfilment of the degree of Master of Science at the Norwegian University of Science and Technology, Department of Engineering Cybernetics.

Working with this thesis has been an intriguing, informative and ever so challenging experience, the results of which I feel righteously proud. I hope and believe I have accomplished something of value to the department, that will help interest others in the field of prosthetics and medical cybernetics, and that students and researchers might find useful in the future.

I would like to take this opportunity to thank *Øyvind Stavdahl* for being my enthusiastic and excellent supervisor, as well as *Anders Fougner* for being my insightful and very helpful co-supervisor.

Contents

1 Introduction

1.1 Reader's guide

This thesis is divided into 8 main chapters. The first chapter after this introduction, chapter 2, presents some background information on myoelectric signals, myoelectric control and pattern recognition and classification. Chapter 3 presents the functional specification of the system and its parts, and chapter 4 discusses different possible software platforms for the control system implementation. Chapter 5 shows the design and implementation of the software that realizes the specified controller. The results are presented in chapter 6 before reaching a conclusion in chapter 7. Finally, chapter 8 suggests some future work to properly utilize the results of this thesis.

There are two appendices, appendix A and B, that are to be considered as part of this thesis work. These present a practical developer's guide to the system hardware and software respectively. Upon request from the supervisors, they are written such that they may be printed and used separately from this document, and are made appendices for this reason.

Part of the problem is to analyse aspects of features such as calculation time and classification accuracy. Because an quantitative discussion on this requires actual data, part of this discussion was moved to the results chapter as the developed system was used for measurement and feature extraction.

Throughout this thesis I will assume that the reader has a good understanding of mathematics, computer science and software design. I will also assume that the reader is familiar with the LabVIEW and MATLAB platforms. A quick overview of LabVIEW can be found in appendix C which readers unfamiliar with LabVIEW are strongly encouraged to read.

1.2 Work methodology

The scope of this problem is very broad. It covers computer science concepts from byte-level communication protocols to high-level abstraction, object oriented programming and integration of different development environments, all with a heavy mathematical back-end for a medical and cybernetic purpose.

The first step was to research the field of myographic control, specifically pattern recognition methods and different EMG features used for this purpose. Then an overview of different software platforms was gathered, most importantly for the NXT.

Software development is an iterative process. An idea that looks good on paper may be cumbersome to use in practice, and some flaws or sub-optimalities are not evident before having extensive experience with the software. Therefore, the functional specification was worked out during the course of development, in collaboration with the supervisors.

Design and implementation started on the NXT, covering servo control and then communication. After that, communication on the host was implemented. Hardware connections, measurement, training, features and classifiers followed in order, with the graphical user interface being developed in parallel. A significant effort was put into implementation, resulting in a system consisting of over 150 LabVIEW VIs and several MATLAB functions. In order to facilitate further development, much time was spent researching the best practice implementation methods by LabVIEW engineers. The National Instruments developer zone[1] and the LabVIEW community[2] were frequently used and of great help.

As with any project, some bumps in the road were experienced, specifically with the hand model. Some time had to be spent on error searching, tweaking and fixing the servos on the model. At the time of writing the problem description, it was thought that the EMG measurement system could provide real-time signals over USB. When this proved not to be the case, analog measurements were introduced, increasing the system layout complexity.

In spite of this, there was still enough time left to test the system after development. A thorough analysis of the controller (i.e. classifier and features) was performed after the system had been successfully implemented.

[1]http://zone.ni.com
[2]http://www.ni.com/labview/community/

1.3 Thesis context

1.3.1 Hardware

EMG and acceleration measurement system

The EMG and accelerometer measurement system used for this thesis work was the *Trigno* system by Delsys. This system consists of 16 separate units, each containing an EMG sensor and a triaxial accelerometer. These are wirelessly connected to a base station where the 16 EMG and the 48 inertial measurements are outputted as analog signals.

All the analog signals are in the ± 5 V range. The raw EMG signals are amplified by a factor of 909 to fill this dynamic range. The accelerometer range can be selected as either ± 1.5 g or ± 6 g.

The base station also offers a USB interface to work with the Delsys EMGworks software. However, since this software does not support exporting of the signals in real-time, analog sampling was used for this thesis work.

Analog sampling

The analog signals were sampled by two National Instrument data acquisition units; a NI DAQPad-6016 and a NI PCI-6025E. Together these units allow for analog to digital conversion of 32 channels with a sampling rate of up to 200 kHz at 16-bit resolution. The signals were wired to allow sampling of all signals from eight electrode units (8 EMG + 24 accelerometer signals).

NXT

The LEGO mindstorm NXT 2.0 module is a robotics controller. It is designed to be robust, easy to use, develop for and program.

The core processor is a 32-bit Atmel ARM processor with 64 KB RAM and a clock frequency of 48 Hz. Several I/O ports are available, supporting analog and digital communication. A full-speed USB port and a bluoetooth module is used for communication with computers and other NXT modules.

A standard NXT module comes with a Lego firmware, but a JTAG interface is available enabling low-level programming of the processor. On the NXT module used during this thesis a JTAG connector has been soldered on. In addition, two wires are connected to the batteries, making the NXT able to function as an external power source.

The NXT module is shown in figure 1.1 and a overview of its specifications is shown in table 1.1.

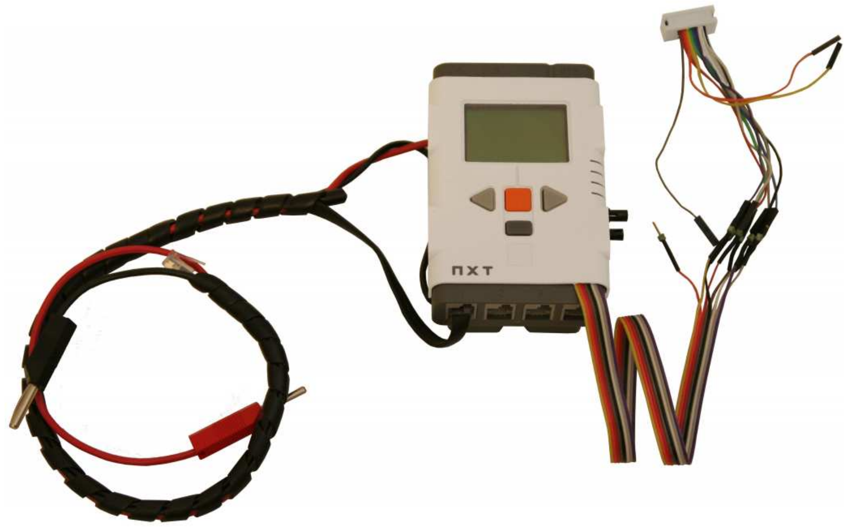


Figure 1.1: Lego Mindstorm NXT 2.0 module with modifications. Image: (Håkonsen 2010)

Table 1.1: NXT 2.0 module specification. For more details refer to the hardware developer kit datasheet.

Main processor	Atmel 32-bit ARM processor, AT91SAM7S256 • 256 KB FLASH • 64 KB RAM • 48 MHz
Co-processor	Atmel 8-bit AVR processor, Atmega48 • 4 KB FLASH • 512 byte RAM • 8 MHz
Bluetooth	CSR BlueCore 4 v2.0 +EDR System
USB 2.0	Full speed port (12 MB/s)
4 input ports	6-wire interface supporting both digital and analog interface
3 output ports	6-wire interface supporting input from encoders

The prosthesis model

The hand prosthesis model that was used in this thesis work was developed by Kristian Håkonsen in his master's thesis (Håkonsen 2010). The model is shown in figure 1.2. It has seven motorized degrees of freedom, listed in table 1.2.

Table 1.2: Model functions and motorized degrees of freedom (DOF).

Part	Movement	DOF number
Forearm	Supination Pronation	1
Wrist	Flexion Extension	2
	Ulnar deviation Radial deviation	3
Thumb	Flexion Extension	4
	Abduction Adduction	5
Index finger	Flexion Extension	6
Middle, ring and little finger	Flexion Extension	7

All the motorized degrees of freedom are controlled by a electronic servo motors. These motors are connected to a NXTServo-v2 servo controller which communicates with the NXT over I^2C. This servo controller allows speed and position control of up to eight servos.

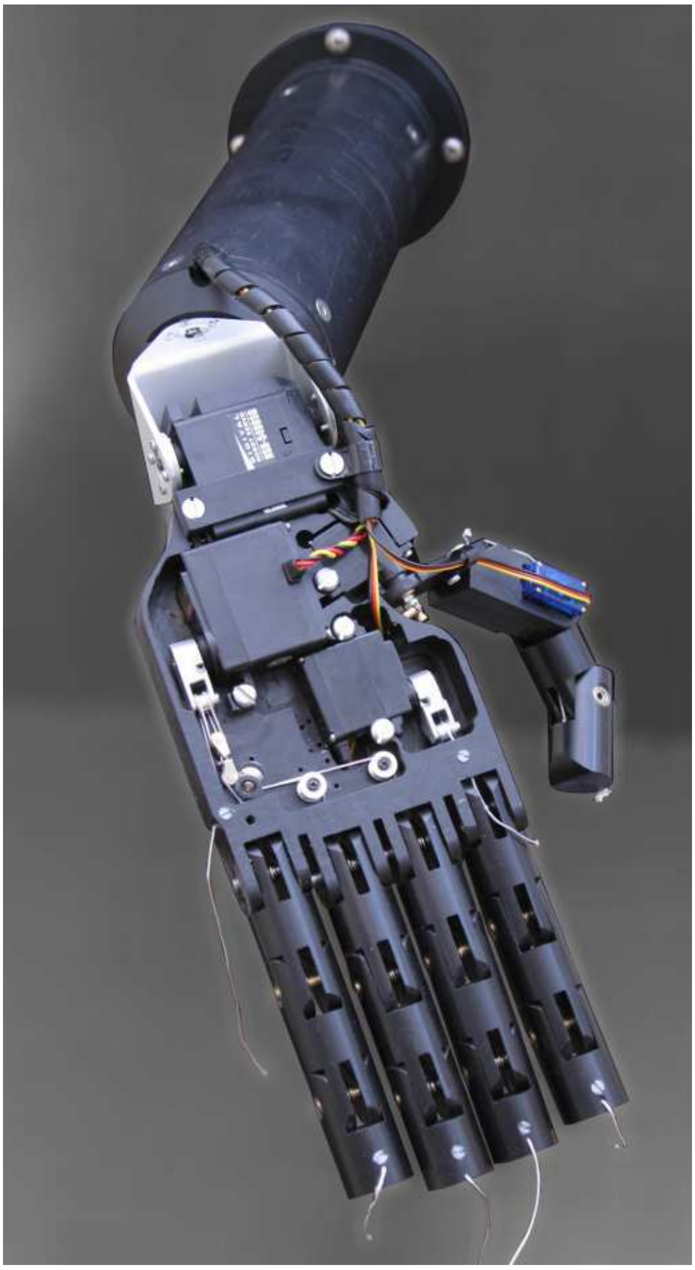


Figure 1.2: Handle model. Picture from (Håkonsen 2010).

2 Background

2.1 The myoelectric signal

This section will give an introduction to electromyography. Section 2.1 is reprinted from the author's previous work in (Bersvendsen 2010).

The electrical activity produced by a contracting muscle is referred to as a *myoelectric* signal. The physiological origin and nature of the myoelectric signal are described in a number of books covering biomedical engineering, such as (Muzumdar 2004).

The goal of this chapter is to present the reader with a rough overview of the processes involved in a muscle contraction. The focus will be on how these processes relate to the myoelectric signal measured by electrodes on the surface of the skin.

Although there are different muscle types in the body, this chapter will exclusively deal with the *skeletal* muscles; those under voluntary control effecting the movements of limbs.

2.1.1 Anatomy and origins

Skeletal muscles are generally connected through *tendons* to two different skeletal bones, such that their contraction produce a movement of the two bones relative to each other. Note that a muscle may not actively stretch, it has only the ability to actively contract.

Roughly speaking, a skeletal muscle consists of several *fascicles* that are bundles of *muscle fibres*, organized as a "bundle of spaghetti". The muscle fibres can be as long as the muscle itself, and each fibre is connected to (or *innervated* from) a single *axon* at a single point about halfway down the fibre.

The axon is a cell that is part of the central nervous system (*CNS*) and its cell body lies in the spinal cord. Each fibre is innervated by only one axon, but an axon innervates anything from 1 to 1000+ fibres, depending on whether the muscle is used for forceful or fine-tuned movements. The collection of an axon and all the fibres it innervates is called a *motor unit*.

When the axon of a motor unit receives activity from the CNS, it leads an electrical impulse towards its connected fibres. When this impulse meets a fibre it is transported along the fibre in both directions away from the innervation point, making the fibre contract as the impulse moves. When an axon creates and leads the electrical impulse, we say that it *fires*. This causes the fibres it innervates to perform a short contraction called a *twitch*.

Note that when an axon fires, all fibres that it innervates twitches. One cannot voluntarily control single fibres, only single axons. Once the axon fires, the innervated fibres will produce the same contraction every time[1]. This means that the produced force of the fibres can not be modulated by the strength of the impulse, only by the frequency of which they are sent, referred to as the *firing rate*.

A single skeletal muscle contains muscle fibres of different properties. Some generate a lot of force when contracting, but fatigue easily. Others generate less force, but can twitch repeatedly over long periods of time. The fibres are generally arranged such that different axons innervate fibres with similar force-fatigue properties.

Another way in which the CNS can modulate the produced muscle force is the *recruitment* of motor units; the order in which the motor units are excited. It comes as no surprise that the less forceful motor units are recruited first, and if they do not deliver the required force the more powerful ones are recruited.

A final important note is the way in which the motor units twitch with respect to each other. They do this *asynchronously*; different axons fire at different times. Since all fibres have a maximum rate at which they can twitch, synchronously firing could not lead to smooth contractions.

2.1.2 Surface EMG measurement

The process of electrically measuring myoelectric signals is called *electromyography* or *EMG*. One may measure the signal either inside the muscle or at the surface of the skin, in which case we call it *surface-EMG* or *SEMG*. SEMG has the obvious benefit of being non-invasive and is much more practical for prosthetic users.

When the electric pulse travels along a muscle fibre, as discussed in the earlier section, it has a spatial spread along the fibre. If one measures the voltage difference between two points along the fibre, the signal will look something like figure 2.1

[1]When not considering time-varieties such as fibre fatigue.

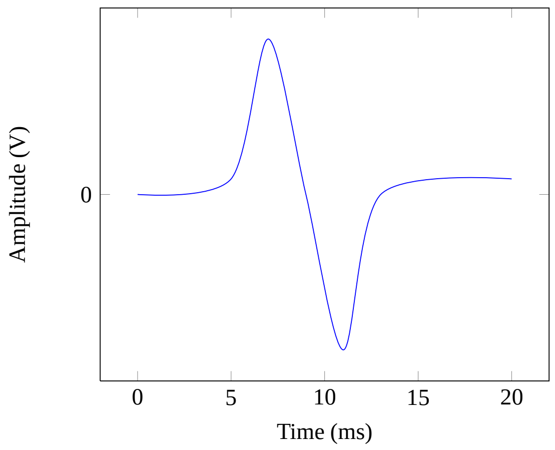


Figure 2.1: Qualitative measured action potential along a muscle fibre. Note that the amplitude significantly depend on the measuring condition (subject and device), but is usually in the order of a few mV.

To accomplish such a measurement on the surface of the skin, one often uses a differential amplifier between two electrodes. This require a total of three contact points to the skin, two measure points and one ground point.

SEMG measurement poses several difficulties. For one, the measured signals are very small (usually some mV in amplitude) and subject to a common-mode due to electrical interference from external power sources (i.e. the power grid). This common-mode is present everywhere on the skin, and can easily be as large as 10-15 V, which is significantly larger than the signal of interest. To overcome this, the common-mode rejection ratio (CMRR) of the differential amplifier needs to be very large.

2.1.3 Nature of the surface EMG signal

The signal shown in figure 2.1 is that of a single fibre. When measuring the activity over a large part of the muscle, the measured signal is actually a sum of several of these signals originating from different motor units at different locations within the muscle. Since the motor units fire asynchronously, as discussed in section 2.1.1, the individual motor unit signals are shifted in time, seemingly at random. When the force exerted by the muscle increases, both the firing rate of the motor units and the number of motor units that are recruited generally increase.

Figure 2.2 shows an actual SEMG measurement[2] of the biceps at rest and during sub-maximal isometric[3] contraction (note that the amplitude of the signals are scaled so they do not reflect the amplitude of the raw SEMG signal). One can see that the signal looks like white noise with variance related to the muscle activity. This may be a practical model, but it is important to note that the signal is *not* noise, it just resembles it.

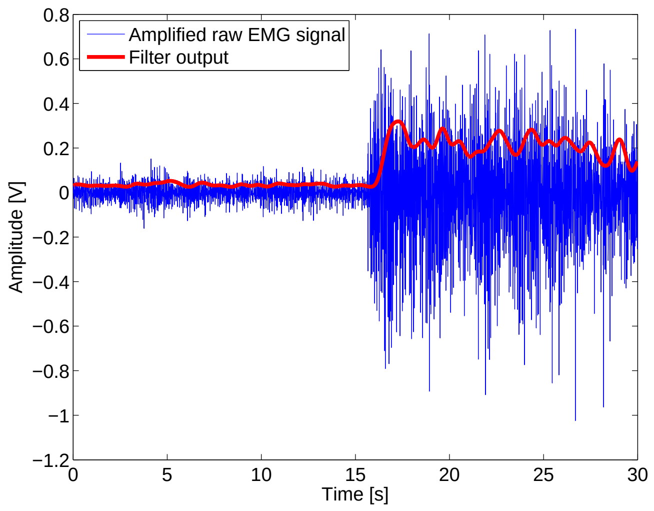


Figure 2.2: *Blue:* Measured SEMG signal (scaled) of biceps at rest and during sub-maximal isometric contraction. *Red:* Rectified and low-pass filtered and scaled SEMG signal.

However, a frequently used measure of the muscle activity is the “noisiness” of the signal. In its simplest form an EMG signal processor consists of a rectifier and a low pass filter. The red line in figure 2.2 shows the result of this operation.

2.2 Myoelectric control systems

Many different myoelectric control systems for prosthetics have been developed. In the simplest case *threshold control* may be used to give an on/off control of a function (e.g. open/close hand). *Proportional control* can give the user control of the speed or force of one or more motors by relating it to some value increasing with muscle activation.

[2]The measurement is from an experiment performed by the author at the University of Twente.
[3]Where the muscle generates force without changing length.

In this thesis *pattern recognition control* will be used, allowing a user to control several functions (i.e. motor movements) in an on/off fashion. Figure 2.3 shows an overview of the main parts in a complete pattern recognition control system. This thesis will treat the software part of this diagram.

The origin and measurement of the SEMG signal was discussed in section 2.1 and feature extraction and classification aspects are discussed in section 2.3.

The low-level feedback consists of sensory information from the prosthesis (e.g. touch sensors, position or speed of motors) which is used by the controller to create correct actuations. One of the major drawbacks of prosthetic control based on EMG is the lack of feedback (proprioception) which healthy persons receive from the muscles that are missing on an amputee. This can be acquired through high-level feedback such as vision or tactile stimulation (e.g. vibration).

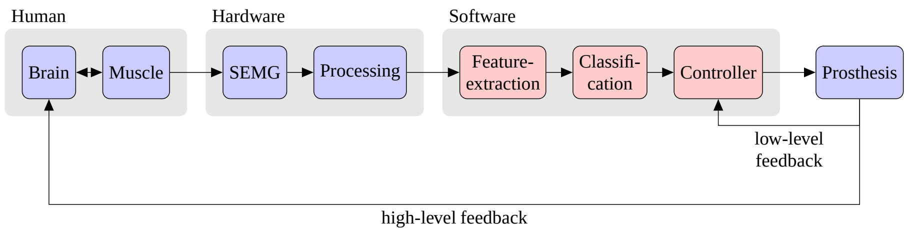


Figure 2.3: Pattern recognition control system overview. Based on (Oskoei & Hu 2007).

2.3 Classification

2.3.1 Problem overview

This section will present the signal classification problem and some solutions. The section is based on the works of Theodoridis & Koutroumbas (2008).

The problem of classification is to study some *object* and labeling it with a *class* of a pre-defined set of classes. In the context of this thesis, the classes are motion classes performed by a subject and the object is a set of measured accelerometer and EMG signals. A motion class may be the specific movement of a single joint (e.g. wrist flexion) or a grip involving several joints (e.g. clenching the fist).

When a certain motion is performed several times, the resulting measurements will never be exactly the same. This is not simply due to noises and uncertainties in the measurement equipment, but the SEMG signal is non-deterministic in nature. However, the muscles will be activated in roughly the same way every time, so the measured signals will have roughly the same properties.

There are several aspects of the measured SEMG signal that may be studied for classification purposes. One may for instance look at the power of the signal, its frequency contents or the number of times the signal crosses zero per second. Such aspects are called *features* of the signal.

Based on a set of measurements, several different features may be calculated, some resulting in a single number and some resulting in several numbers. All these feature values can then be stacked into a vector, which is called a *feature vector*.

A general classifier works by comparing this feature vector to existing feature vectors that are already classified. To label a feature vector with a class is called *classifying*. The existing feature vectors that the classifier uses is called the *training data* of the classifier.

2.3.2 Solution outline

Given a problem consisting of three classes described by two scalar features. The feature-vector now consists of two elements and is a point on the $\mathbb{R}^2$ plane. One solution of the classifier problem is to divide this plane into three mutually exclusive regions. In this case a given feature vector $\mathbf{x}$ lies in one (and only one) region, and the resulting class is then said to be the one associated with this region[4].

An example of such a feature-space division is shown in figure 2.4. Note that in this case the class divisions are straight lines. These specific type of classifiers are called *linear* classifiers, and are the ones used throughout this thesis.

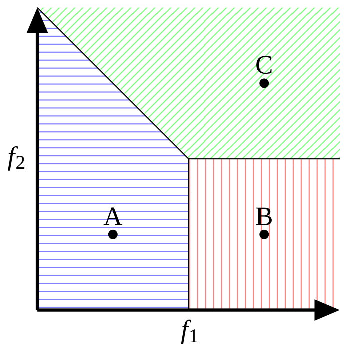


Figure 2.4: Example of a classification scheme with three classes (A, B and C) and two feature values (f_1 and f_2). The feature space ($\mathbb{R}^2$ plane) is divided into three mutually exclusive regions, one for each class.

This concept generalizes to any number of classes and any number of features. In the general case there are M classes and l feature values, in which case the $\mathbb{R}^l$ feature-space

[4]What is done on the border may be arbitrary.

is divided into M mutually exclusive regions.

Because there are many ways of creating these regions, even when limited to linear constraints, many different linear classifiers have been developed.

2.3.3 Least squares method (LSM)

The least squares method is a computationally simple method that does not assume anything on the training data set (e.g. class separability).

For each class an output is defined that is a linear combination of the feature vector elements. This linear combination is chosen such as to minimize an error in the least squares sense.

Given a training set of M classes and N feature vectors of length l with class association. For each class i a weight vector $\mathbf{w}_i$ is chosen such that it minimizes the cost function

$$J(\mathbf{w}) = \sum_{k=1}^{N} (y_k - \mathbf{x}_k^\top \mathbf{w})^2 \tag{2.1}$$

where y is the desired output and $\mathbf{x}$ is the feature vector. This is chosen as a positive number (i.e. 1 in this thesis) for y_k if $\mathbf{x}_k$ belongs to class i, and a non-positive (i.e. 0 in this thesis) if not.

(2.1) can be solved as a standard least squares problem, and the optimal $\mathbf{w}_i$ is given by

$$\mathbf{w}_i = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y} \tag{2.2}$$

where

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^\top \\ \mathbf{x}_2^\top \\ \vdots \\ \mathbf{x}_N^\top \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1l} \\ x_{21} & x_{22} & \cdots & x_{1l} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \cdots & x_{Nl} \end{bmatrix}, \mathbf{y} = \begin{bmatrix} y_1^\top \\ y_2^\top \\ \vdots \\ y_N^\top \end{bmatrix} \tag{2.3}$$

A subtle "trick" can be done in order for the output to be a linear combination of the feature vector elements *and* a constant. This is done by adding a 1 as the last element in the feature vector, giving a new vector

$$\bar{\mathbf{x}} = \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} \tag{2.4}$$

which gives the new $\mathbf{X}$ matrix

$$\mathbf{X} = \begin{bmatrix} \bar{\mathbf{x}}_1^\top \\ \bar{\mathbf{x}}_2^\top \\ \vdots \\ \bar{\mathbf{x}}_N^\top \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1^\top & 1 \\ \mathbf{x}_2^\top & 1 \\ \vdots & 1 \\ \mathbf{x}_N^\top & 1 \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1l} & 1 \\ x_{21} & x_{22} & \cdots & x_{1l} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{N1} & x_{N2} & \cdots & x_{Nl} & 1 \end{bmatrix} \tag{2.5}$$

We now have M optimal linear combinations $\mathbf{w}_i$ of $\bar{\mathbf{x}}$. These have the property that they come closer to 1 as $\bar{\mathbf{x}}$ approaches the training feature vectors of class i and come closer to 0 as $\bar{\mathbf{x}}$ approaches the training feature vectors of all other classes. The classification algorithm can then simply calculate all the linear combinations and choose the class i for which $\bar{\mathbf{x}}^\top \mathbf{w}_i$ is maximized.

Note that $(\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top$ in (2.2) is called the *pseudoinverse* of $\mathbf{X}$ which can be evaluated efficiently in software. Another simplification is that all the linear combinations may be evaluated at once by

$$\mathbf{G}\bar{\mathbf{x}} \tag{2.6}$$

where

$$\mathbf{G} = \begin{bmatrix} \mathbf{w}_1^\top \\ \mathbf{w}_2^\top \\ \vdots \\ \mathbf{w}_M^\top \end{bmatrix} \tag{2.7}$$

Similarly, all the $\mathbf{w}$ vectors may be calculated at the same time by

$$\mathbf{G} = \begin{bmatrix} \mathbf{w}_1 & \mathbf{w}_2 & \cdots \mathbf{w}_M \end{bmatrix} = (\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top \mathbf{Y} \tag{2.8}$$

where

$$\mathbf{Y} = \begin{bmatrix} \mathbf{y}_1 & \mathbf{y}_2 & \cdots & \mathbf{y}_M \end{bmatrix} = \begin{bmatrix} y_{11} & y_{21} & \cdots & y_{M1} \\ y_{12} & y_{22} & \cdots & y_{M2} \\ \vdots & \vdots & \ddots & \vdots \\ y_{1N} & y_{2N} & \cdots & y_{MN} \end{bmatrix} \tag{2.9}$$

To demonstrate the computational elegance of this method, the following MATLAB snippet trains an LMS classifier (i.e. calculates the $\mathbf{G}$ matrix)

```
G = X\Y;
```

and classification is done by

```
[¬, class] = max(G*[x; 1]);
```

Figure 2.5 illustrates how this works in practice. Two classes (red and blue) are given by a set of feature vectors with length $l = 1$ (i.e. scalars). Training the LMS classifier produces two (one for each class) linear combinations of the feature vector (a scalar) plus a constant, resulting in a line in $\mathbb{R}^2$. These linear functions are plotted, as well as the maximum value (gray). One can see that the feature is classified as red if it is below a value of about 1.5 and blue otherwise. If $l = 2$ the feature vectors are distributed in the $\mathbb{R}^2$ plane and the linear combinations would be planes in $\mathbb{R}^3$, and so on.

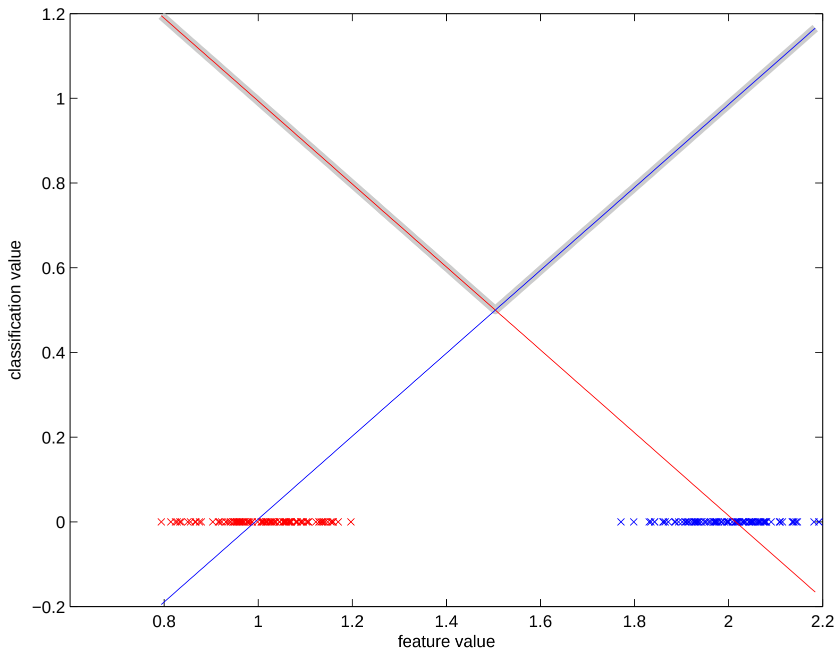


Figure 2.5: Example of LMS classification with two classes and scalar features. The gray line shows the maximum of the classification values.

2.3.4 Linear discriminant analysis (LDA)

The *LDA* classifier creates the linear class separations by studying the statistical properties (i.e. mean and covariance) of several feature vectors for each class acquired during training.

An existing MATLAB implementation of this classifier as presented in (Englehart, Hudgins, Parker & Stevenson 1999) was supplied by the supervisors.

2.4 Signal features

2.4.1 Feature expressions

Section 2.4.1 will present the most commonly used features in EMG prosthesis control based on (Zardoshti-Kermani, Wheeler, Badie & Hashemi 1995, Boostani & Moradi 2003, Fougner 2007, Bach 2009, Englehart et al. 1999, Parker, Englehart & Hudgins 2006). All features are calculated on a time-series of a single EMG channel. This time-series of N samples is called a *window of length N*.

Average absolute value (AAV)

This is one of the most intuitive and commonly used signal feature, and is simply given by

$$\text{AAV} = \frac{1}{N}\sum_{i=1}^{N}|x_i| \tag{2.10}$$

This feature is easy to calculate, has linear time complexity (AAV $\in O(n)$), and can be easily implemented analogously in hardware. However, it is significantly dependent on the quality of the electrode-skin interface, which may be changed over time (e.g. by sweat).

Variance (VAR)

The unbiased variance estimation given by

$$\text{VAR} = \frac{1}{N-1}\sum_{i=1}^{N}x_i^2 \tag{2.11}$$

is another commonly used feature. Note that (2.11) assumes an expected value of zero, which is the case for EMG signals. This value is related to the power of the signal[5],

[5] The signal power P is the “energy per sample”, given by

$$P = \frac{E}{N} = \frac{1}{N}\sum_{i=1}^{N}(x_i-\mu)^2$$

but has the same problems as AAV. Easy to calculate and has linear time complexity (VAR $\in O(n)$).

Willison amplitude (WAMP)

The Willison amplitude, introduced in (Willison 1963), counts the number of times the difference between two consecutive samples exceed a pre-defined threshold value. This can be expressed mathematically as

$$\text{WAMP} = \sum_{i=1}^{N-1} f\left(|x_i - x_{i+1}\right) \tag{2.12}$$

where

$$f(x) = \begin{cases} 1 & \text{if } x > \text{threshold} \\ 0 & \text{otherwise} \end{cases} \tag{2.13}$$

Willison himself used a threshold value of 100 μV, but other values have been used by others. Easy to calculate and has linear time complexity (WAMP $\in O(n)$).

Zero crossing (ZC)

The zero-crossings feature, as the name suggests, counts the number of times the signal crosses zero, and is given by

$$\text{ZC} = \sum_{i=1}^{N-1} \text{sgn}\left(-x_i x_{i+1}\right) \tag{2.14}$$

where

$$\text{sgn}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.15}$$

Instead of using $x > 0$ in (2.15) one can use $x =$ threshold where the threshold is some positive value. Easy to calculate and has linear time complexity (ZC $\in O(n)$).

---

which gives

$$\sigma^2 = P - \mu^2$$

where μ is the average value and σ is the biased variance estimate

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N} x_i^2$$

Number of turns (NT)

Counts the number of turning points of the signal; the number of times the signal slope changes sign. This can be calculated by

$$\text{NT} = \sum_{i=1}^{N-2} \text{sgn}\left(-(x_{i+1} - x_i)(x_{i+2} - x_{i+1})\right) \tag{2.16}$$

where

$$\text{sgn}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.17}$$

Easy to calculate and has linear time complexity (NT $\in O(n)$).

Average amplitude change (AAC)

Calculates the average absolute amplitude change between two consecutive samples, given by

$$\text{AAC} = \frac{1}{N-1} \sum_{i=1}^{N-1} |x_{i+1} - x_i| \tag{2.18}$$

Easy to calculate and has linear time complexity (AAC $\in O(n)$).

Myopulse percentage rate (MYOP)

Calculates the fraction of samples exceeding a given threshold, given by

$$\text{MYOP} = \frac{1}{N} \sum_{i=1}^{N} \text{tresh}(x_i) \tag{2.19}$$

where

$$\text{tresh}(x) = \begin{cases} 1 & \text{if } x > \text{threshold} \\ 0 & \text{otherwise} \end{cases} \tag{2.20}$$

Easy to calculate and has linear time complexity (MYOP $\in O(n)$).

Histogram (HIST)

Creates *n bins* of amplitude values, ranging from the minimum to the maximum value. Then counts the the number of samples in each bin, resulting in *n* feature values. Boostani & Moradi (2003) used nine bins of unknown levels. Easy to calculate and has linear time complexity (HIST $\in O(n)$).

Auto-regressive coefficients (AR)

This feature fits a n'th order AR filter[6] to the window. A 4'th order model is widely used, as increasing past this does not generally infer an increase in classifier performance. Quadratic time complexity (AR $\in O(n^2)$) using Levinson-Durbin recursion (which is used by the internal `levinson` function in MATLAB).

Cepstral coefficients (CC)

Cepstral coefficients are widely used in speech and music recognition applications, and may also be used with EMG signals. Fougner (2007) defines this feature as finding the Fourier transform of the logarithm magnitude spectrum. **?** on the other hand, define this to be the *inverse* Fourier transform of the logarithm magnitude spectrum, but notes that different techniques are used in practice. The following expression is given in (Kang, Cheng, Lai, Shiu & Kuo 1996, Boostani & Moradi 2003) and used in papers such as (Pattichis & Elia 1999).

$$c_1 = a_1 \quad c_n = -\sum_{k=1}^{n} \left(1 - \tfrac{k}{n}\right) a_k c_{n-k} - a_n \tag{2.21}$$

were c_i is the i'th cepstrum coefficient and a_i is the i'th AR coefficient.

With the definition in Xavier & Rodet (2003) the DFT and IDFT algorithms dominate the time complexity. Using the FFT and IFFT gives CC $\in O(n \log n)$.

Wavelet coefficients

The Fourier transform contains information on a signal located in frequency, but looses time domain information. Because myographic signals contain transitory characteristics, this is not necessarily an optimal feature.

The *wavelet transform* is a mathematical transformation similar to a windowed Fourier transform that have proved to give good results for EMG classification. With the wavelet transformation, analysis can be done on time-localized subsets of the data without the limiting precision of a windowed Fourier transform.

2.4.2 Correlation between features

With pattern recognition methods where several features are used, it is important that the features give information on different aspects of the signal (i.e. that they are as uncorrelated as possible). Otherwise one is basically wasting computation time, as an increased number of features in this case does not increase the available information.

[6]
$$y_i = c + \sum_{i=1}^{n} x_i + \varepsilon_i$$
where ε is a white noise with zero mean and some variance.

Some of the features mentioned in section 2.4.1 are intuitively correlated, such as AAV and VAR. As shown in (2.10) AAV is essentially the sum of the absolute sample values, whereas VAR is essentially the sum of the absolute sample values squared, as shown in (2.11). Obviously, $|x|$ is significantly correlated with x^2, so one would expect AAV and VAR not to complement each other very well.

Another example is AAV and AAC defined in (2.10) and (2.18) respectively. An EMG signal generally looks like white noise around zero, with amplitude increasing with muscle activation. Obviously, when the amplitude of such a signal increases, the amplitude change between consecutive samples will increase as well.

On the other hand, ZC and NT are independent on the amplitudes, which intuitively suggests that they are less correlated with AAV and AAC.

A quantitative study of the correlation between different features was done using the developed software for measurement and feature extraction. These results are presented in chapter 6.4.4. Note that in order to represent the numbers in a two-dimensional array only scalar features were calculated.

2.4.3 Calculation time

All of the features presented in section 2.4.1, with the exception of the wavelet coefficients, were implemented in MATLAB. In order to investigate the calculation time a simple test was carried out. Vectors of random numbers with different lengths were used as input for each feature, and the execution time was recorded. For each input vector and feature, the calculation was done 100 times and the resulting time was averaged over these trials. The vector lengths were multiples of $N = 1000$ samples. The results are shown in figure 2.6.

Note that the actual times used are not really important, as they are obviously exclusive to the computer on which the test ran. Furthermore, as MATLAB is such a high-level language, the results will not necessarily be the same for a low-level implementation (e.g. written in C). Note however that the times are in the millisecond scale, which means that there will be no problem in calculating them fast enough on the computer used during development.

More important is the time usage relative to the other features, and the time usage relative to different data set size for the same feature. The results show that the AR and cepstral coefficients are the most computationally demanding.

A similar study was done in (Bach 2009) which included wavelet features. His approach (e.g. input data size) is not documented, but using the MATLAB wavelet toolbox he concluded that calculation of wavelet features took about 35 to 650 times longer than that of AAV. However, these features were not implemented in this thesis.

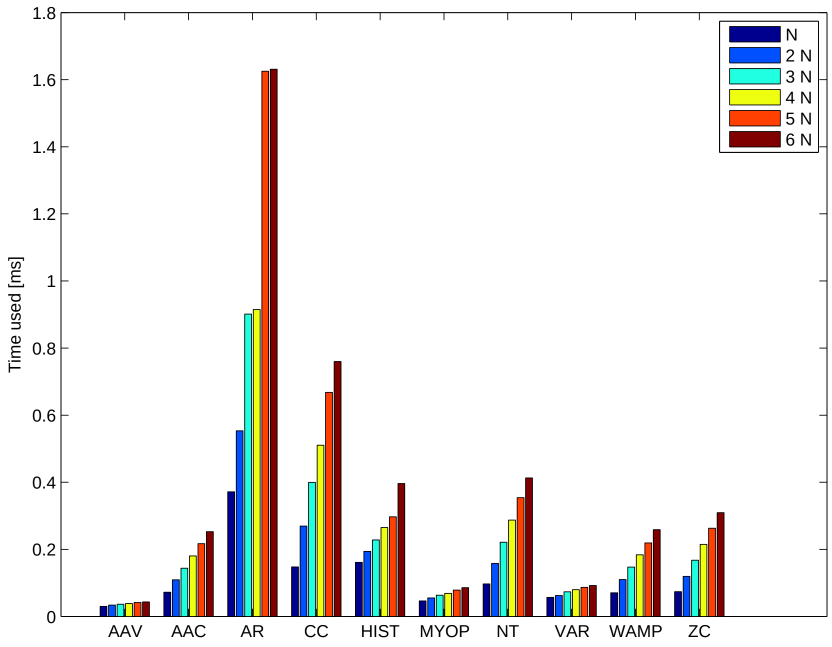


Figure 2.6: Calculation time of implemented features with different signal lengths. The base length is $N = 1000$.

3 Function specification

3.1 Main usage overview

The complete system is meant to be used as a demonstrator on exhibitions, stands or in classroom settings. As a minimum the prosthesis must be controllable by a single person to demonstrate its functionality to an audience. If possible however, spectators should be able to use the system, with as little training and configuration time as possible.

Where a regular prosthesis must be robust enough to work day and day again with little or no need for reconfiguration, this system may need re-training for each event it is to be used for.

Since the software is primarily to be used by a few people with good knowledge of the system, and due to the time constraints of the thesis work, error handling is not a high-priority goal. It is much more important that the system is functional in normal cases, than that unexpected events should be treated correctly. The software will not protect the user from himself; if the user makes an error in configuration (e.g. sets the sample rate too high), the system may display an error and simply require restart.

3.2 Contextual goals

Given that the system will include real-time acquisition of EMG and accelerometer signals it is obvious that this may be a tool for researches and students to test prosthesis control schemes in practice. A secondary goal is therefore to make the system easily extendable and to achieve a well-documented and modular software design.

As this is a system primarily meant for one specific purpose and set of hardware, the limits on both hardware and software follow the availability for the department where this thesis is written. In practice, this means that any software or hardware that is readily available for students and employees at the department can be a requirement to run the system. Furthermore, the department will benefit from a software platform written in programming languages that students and employees are generally used to.

3.3 Specification overview

The system contains two modules, the *host* running on a computer and the low-level controller running on the NXT. When working together, the host acquires measurements and calculates set-points for the NXT. These set-points are then communicated and the NXT controls the servos.

In addition, these nodes should have the ability of working independently. For the host, this means that one should be able to run the measurements and calculations, but instead of communicating the set-points they are somehow shown to the user instead. For the NXT, this means that instead of accepting set-points from the host, the set-points are read from a file on the NXT to perform a pre-defined movement.

There are currently 7 servos used, but the servo controller is capable of 8 connections. Adding an eight servo should therefore require a minimal change in code.

Furthermore, the NXT has several additional, unused ports that may be used in the future. This requires a flexible communication protocol between the nodes, making it less cumbersome to add additional information exchange. As these ports may be used for sensors and not merely actuators, the NXT should be able to send messages back to the host, both as a response (e.g. the host requests information) and by its on initiative (e.g. an NXT event occurs). This requires the host and the NXT to have both send and receive capabilities.

3.4 Host

The host software is where the analog signals are gathered and the majority of the calculations are done. It acquires the measurements and performs the high-level aspects of the control scheme, communicating set-points to the NXT.

3.4.1 Measurement

The software should be able to acquire and handle measurements from any combination of the 16 EMG and accelerometer sources, even though the current hardware setup limits this to 8.

There is no need for individually selecting separate accelerometer components (X, Y or Z). However, one should be able to acquire only the EMG signal, only the accelerometer signals, none or all signals for each electrode unit.

The sample rate, window size and window update rate should be configurable.

3.4.2 Training

During the training phase a number of motion classes are shown to the user in sequence. The user mimics the motion depicted on the screen and the selected sources are measured and stored in memory.

The user may choose any number of classes to train, and class descriptions may be added, removed or edited from outside the software source code.

For each class there is a preparation and a sample phase. The duration of these phases should be configurable, and the complete training sequence may be repeated a defined number of times. The order in which the classes appear to the user may be sequential or randomized according to the configuration.

3.4.3 Feature extraction and classifier training

Once the training is complete a feature vector is calculated for each measured window. The feature calculations should be easily edited and new ones added from outside of the software source code. The user may decide which of the available features to extract. Once the feature extraction is complete, a classifier should be trained based on a set of feature vectors with class relationship.

3.4.4 Demonstrator

Once the classifier training is complete, the user may run the system. In this mode a window is measured from the configured sources, and a feature vector is calculated. This feature vector is inputted to the classifier, which outputs the current motion class. This motion class is then displayed to the user.

Each class may have a set of speed servo set-points associated with it. If the NXT is connected, these set-points are sent to the NXT which controls the servos.

3.4.5 Save/load

At any time one should be able to save the current state of the application to a file. The state includes all configuration, active measurement recordings and calculations.

3.5 NXT

The function of the NXT unit is to control the servo motors on the prosthesis model. It will have two modes. In the *controller* mode it accepts motor set-points from the host and realizes these set-points on the sevos. The set-points should be set in a unit intuitively related to the angle of the motorized joints. In addition, the host should be able to set the raw control set-points of the servos (i.e. the pulse-width of the PWM signals). Both position and speed set-points should be supported.

Furthermore, the position and speed of the servos should be readable over the same communication channel, again in a unit intuitively related to the angle of the motorized joints.

In the *demo* mode the software makes the model move in a predefined way. The purpose of this mode is to demonstrate the possibilities of the model and to serve as an eye-catcher. The model hand may for example wave to or point at an audience.

3.5.1 Goals

The primary goal of the NXT software is that it ought to be robust enough to operate for long periods of time without reprogramming or even resetting the NXT. Once programmed, one should be able to utilize every aspect of the model hardware over the communication channel.

4 Software platforms overview

4.1 NXT

4.1.1 NXT-G (LabVIEW)

The NXT comes with a limited, “LEGO-like” version of the LabVIEW programming language G, called NXT-G. In addition, LabVIEW comes with a NXT add-on that allows one to create LabVIEW programs that run on the NXT with regular LabVIEW code with limited block support. All of the NXT features are supported in NXT-G.

The communication between a NXT-G program running on the NXT and a regular LabVIEW program running on a computer is very straight-forward and robust. Both wired USB and wireless bluetooth is supported. However, NXT-G infers significant overhead in execution as it is a very high-level programming language, resulting in a performance loss. A significant overhead is also inferred in the NXT-host communication, gaining robustness with the cost of communication speed.

As this is the way the NXT was meant to be programmed from the manufacturer, good support and a lot of example code is available.

4.1.2 Next Byte Codes (NBC) and Not eXactly C (NXC)

Next Byte codes (NBC) is a simple, open-source, assembly-like language that can be used to program the NXT. Not eXactly C[1] is a high-level open-source language, similar to C, built on the Next Byte Codes (NBC) compiler.

The NXC syntax is very similar to ANSI C, but there are constraints on the number of functions and variables specific to the NXT.

4.1.3 MATLAB and Simulink

Both MATLAB and Simulink have toolboxes that can generate low-level code that can run on the NXT. Most NXT features are supported, including LEGO sensors, low-level I^2C and serial communication. Available freely online[2].

4.1.4 leJOS NXJ

leJOS[3] is a tiny, open-source Java Virtual Machine that is ported to the NXT. In addition, leJOS NXJ is a set of tools and Java APIs to help code for and program the NXT. leJOS currently supports the following features

- Object oriented programming
- Preemptive threads
- Multi-dimensional arrays
- Recursion
- Synchronization
- Exceptions
- Java types
- Most of the `java.lang`, `java.util` and `java.io` classes
- A well-documented robotics API

leJOS comes bundled with an API for the full Oracle Java VM that simplifies host-NXT communication on the host side. A developer plugin for the popular Java IDE[4] Eclipse[5] is also available. The project was started in 2006 and is still in active development. The newest release (leJOS NXJ 0.9) was released May 16th, 2011.

[1]http://bricxcc.sourceforge.net/nbc/
[2]http://www.mathworks.com/academia/lego-mindstorms-nxt-software/legomindstorms-matlab.html
[3]http://lejos.sourceforge.net/nxj.php
[4]Integrated developer environment
[5]http://eclipse.org

4.1.5 C/C++

The ARM7 chip could be programmed with machine code compiled from assembly, C or C++. Several such compilers exists, both free (e.g. the Imagecraft ICCARM Embedded Development Suite) and commercial (e.g. the IAR Embedded Workbench). The main advantage of such a compiler is that it provides complete native access to the hardware with absolutely no overhead. Specialized libraries for the specific hardware platform exist, such as LibNXT[6]. One would expect this to significantly increase the utilization of the available processing power on the NXT. However, one would not be able to utilize the existing LabVIEW libraries that are specifically designed for the NXT.

As a result, the development time for a program with equal robustness would (from experience) be higher when opting for the C/C++ software platform then a high-level language.

4.1.6 Other platforms

There are interpreters and compilers for many different languages available, including Lua[7], Ruby[8], Ada[9], Python[10], C#[11] and even Haskell[12]. These were not considered in great detail as they seem to have a small user group compared to the other platforms discussed. In addition, these languages are not commonly used by students or employees at the department.

4.2 Host

As the host can run any operating system, the options here are practically endless. The only limit is that the National Instruments DAQ modules must be supported, but this is generally the case on most major platforms. However, most choices of NXT software platform has a logical host counterpart to facilitate communication between the nodes.

4.3 Choice

The decision was taken to use LabVIEW for both the host and the NXT. Considering that National Instruments is the main line of hardware used, this seems to be the most natural choice.

[6]http://code.google.com/p/libnxt/
[7]http://hempeldesigngroup.com/lego/pbLua/
[8]http://ruby-nxt.rubyforge.org/
[9]http://libre.adacore.com/libre/tools/mindstorms/
[10]http://code.google.com/p/nxt-python/
[11]http://nxtnet.codeplex.com/ and http://www.mindsqualls.net/
[12]http://hackage.haskell.org/package/NXT

In addition, LabVIEW offers a tight integration with MATLAB which is a great tool for performing the control system calculations. The graphical user interface in LabVIEW is extremely easy to create compared to regular programming languages. The DAQmx drivers and NXT communication API are specifically designed for LabVIEW which results in a very robust platform. Finally, both LabVIEW and MATLAB are available for students and employees at the department.

The main shortcoming of this choice is performance, mainly experienced on the NXT. Should it be experienced, however, that the LabVIEW platform introduces such high overheads that the communication and execution speed is reduced to unacceptable levels, this choice must be reevaluated.

Since LabVIEW offers integration with native code and the .NET framework, a possible solution, in the event that the NXT software is unacceptably slow, is to re-program the NXT and write a communication protocol in native code. This would allow the host software to still run in LabVIEW.

5 Software design and implementation

5.1 System overview

Figure 5.1 shows a schematic diagram of the different parts of the system and their relationship. The analog signals of the Trigno EMG and accelerometer measurement system are measured by two NI DAQ modules. Note that the Trigno system needs a USB connection to the host in order to activate the analog outputs. The host communicates with the NXT over either bluetooth (BT) or USB which again communicates over I^2C with the model.

This section will present the design and implementation of the host and NXT software (red boxes in figure 5.1). The focus will be on giving a top-level view and understanding of the system, but source code of some key features will also be shown.

5.2 Code terms

The following terms and code structures will be used throughout this chapter. These are considered “best practice” approaches by National Instruments themselves, and are commonly used in academic LabVIEW applications such as (Elliott, Vijayakumar, Zink & Hansen 2007, Hosek, Prykäri, Alarousu & Myllylä 2009). For more information please refer to the provided links to the LabVIEW developer zone articles.

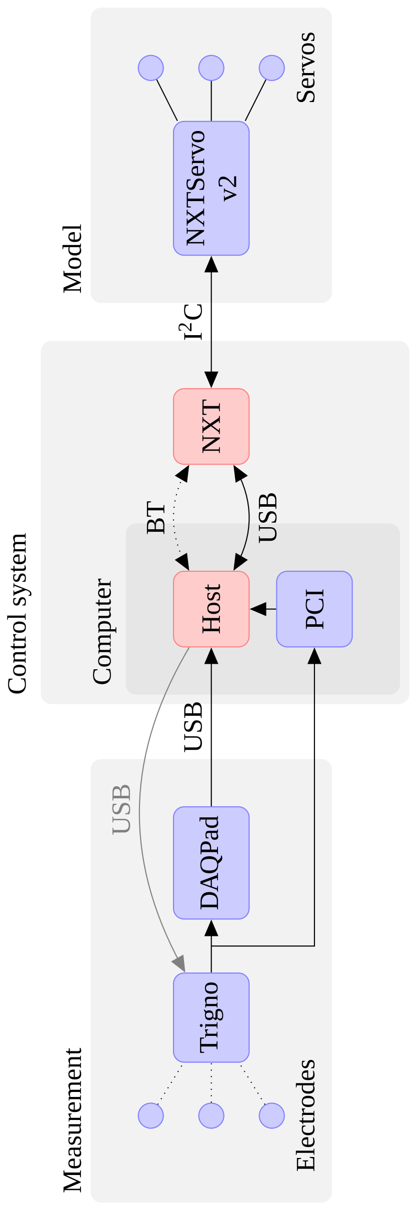


Figure 5.1: System overview.

Standard error in/out functionality[1] Requires a VI to have an error input and output. If the error input contains error information, this is passed on to the error output. In this case the VI may choose not to complete its tasks and instead return immediately. If the error input contains no error, the error output will contain information on an error that occurred inside the VI, if any.

Functional global variable[2] A local variable encapsulated in a VI that allows different access methods and is not reentrable. Most commonly both read and write access is given, but some VIs may include an initialize option or may lack the write option. Can be implemented using a local variable or a while loop with a shift register. In practice, more than one local variable may be used inside the VI.

Dynamic event registration[3] Events may be registered dynamically using a control reference. In this way events can be handled in a sub-VI instead of the VI containing the control. Once registered, some events may be disabled or re-enabled.

Producer/consumer architecture[4] The producer/consumer architecture consists of one or more producer-loops and one or more consumer-loops. All loops have access to a synchronization variable, such as a queue. The producers produce some value (e.g. a measurement) and adds it to the queue. The consumers wait for something to be added to the queue, read the element and process it (e.g. shows it on a graph).

5.3 Design pattern implementations

This section will demonstrate the implementation style of LabVIEW design patterns that are frequently used in the developed software. Each design pattern will be demonstrated with an example. These examples are fictional and simplistic, but the core concepts of the design patterns nevertheless remain the same.

Some examples may seem so easy to implement in ordinary ways as to make the proposed implementation seem bloated. Note that in practice these design patterns are used in much larger pieces of software, such as the developed system, where abstraction and modularization are critical concepts.

5.3.1 Functional global variable

A functional global variable is one or more local variables encapsulated in a VI that allows different access methods and is not reentrable. The different access method are specialized for the needs of the specific variable.

[1]http://zone.ni.com/reference/en-XX/help/371361G-01/lvconcepts/using_standard_error_in/ and http://zone.ni.com/reference/en-XX/help/371361G-01/lvconcepts/using_standard_error_out/

[2]http://labviewwiki.org/Functional_global_variable

[3]http://zone.ni.com/reference/en-XX/help/371361G-01/lvconcepts/using_events_in_labview/

[4]http://zone.ni.com/devzone/cda/tut/p/id/3023

Consider a variable containing a single struct with two elements, a string and a number. We want to be able to read from and write to this variable. In addition we have a special action called initialize, which may perform some initialization action. Finally, we want to be able to set the numeric element of the variable without changing the string value. Standard error-out functionality should be used.

Figure 5.2 shows the front panel of this functional global variable. Besides the error wires it contains three inputs and one output. The `Action` input is an enum control specifying the available actions or access methods. The `Value in` and `Numeric in` controls are used to set the value when `Action` is `Write` and `Set numeric` respectively.

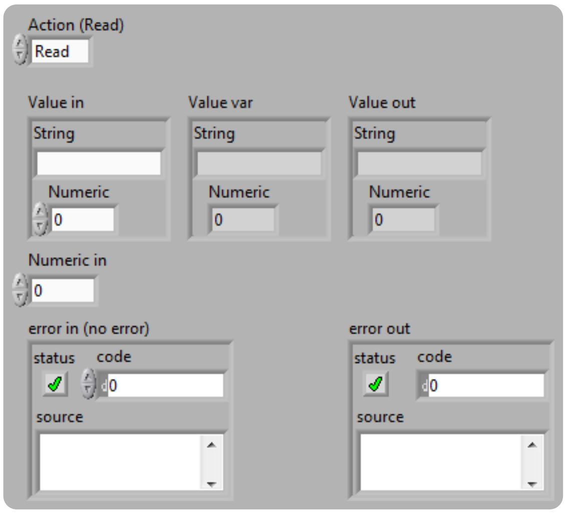


Figure 5.2: Functional global variable example front panel.

Figure 5.3 shows the standard error-out functionality, as well as the general block diagram layout. A switch case structure is used to perform an action based on the `Action` input value. All cases are shown in figure 5.4.

Note that the output is always given, even in the write action cases. This is part of the design pattern and is useful for signal routing to specify execution order. In addition, the VI may change the input value before writing and outputting it (e.g. by limiting the numeric to a certain range in this example).

5.3.2 Dynamic event registration

Events may be registered dynamically using a control reference. In this way events originating in a control can be handled in a sub-VI instead of the VI containing the

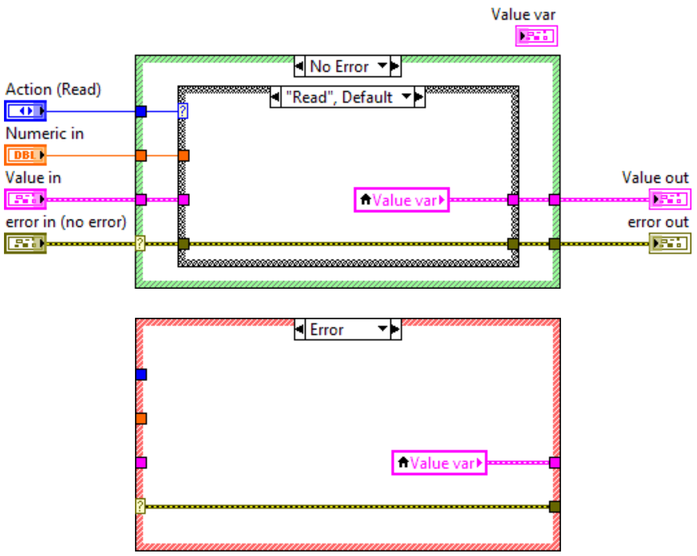


Figure 5.3: Functional global variable example. Block diagram showing the standard error out functionality.

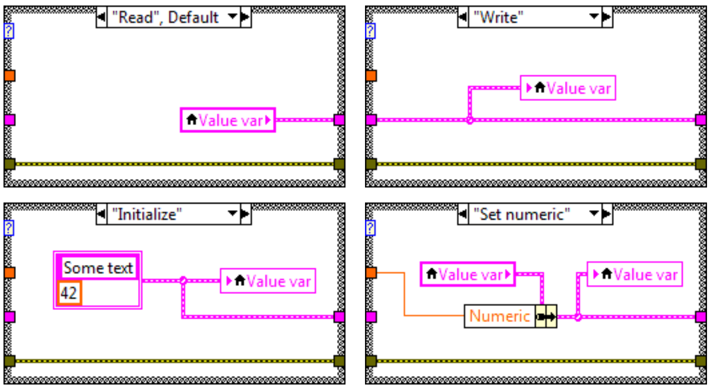


Figure 5.4: Functional global variable example. Block diagrams for each action case.

control.

Consider a simple application containing two elements, a numeric control and a boolean indicator. The indicator is to display whether or not the value of the numeric control exceeds a value of 5. Once started, the application should run until aborted by LabVIEW (i.e. no dedicated stop button on the front panel).

Figure 5.5 shows the front panel and an ordinary (i.e. statically registered) even structure that performs this task. Note that this event structure may not be placed in any other VI but the one that is running the front panel, as the control and indicator are used directly and the event structure refers to an event originating in this VI.

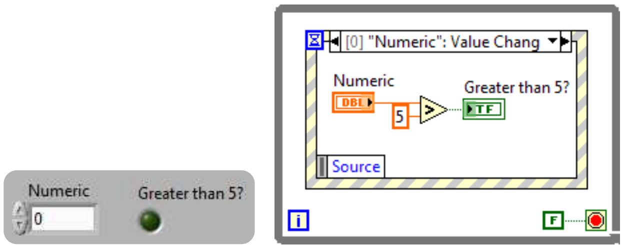


Figure 5.5: Dynamic event registration example. Front panel and block diagram using ordinary, *statically* registered events.

Inferring dynamic event registration means that the event structure may be contained by another VI (called a *handler* for cases like this). In this case references to the original front panel objects are used to read and write the control values. Figure 5.6 shows the implementation of the handler using dynamic event registration. The new block diagram of the main VI is shown in figure 5.7.

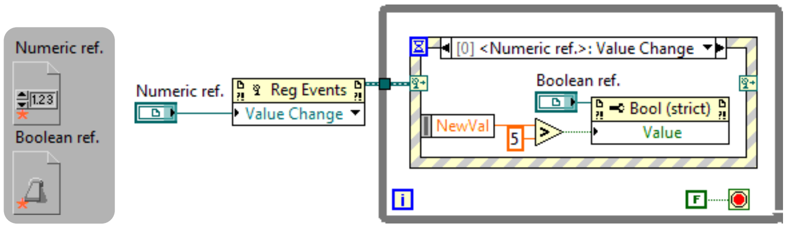


Figure 5.6: Dynamic event registration example. Front panel and block diagram of the dynamic handler.

Note that the handler does not know or care which VI calls it, meaning that this VI may now be reused by any other piece of software.

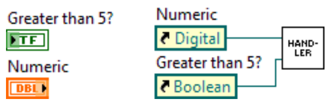


Figure 5.7: Dynamic event registration example. New main VI block diagram using the dynamic handler.

5.3.3 Producer/consumer architecture

The producer/consumer architecture consists of one or more producer-loops and one or more consumer-loops. These loops both have access to a buffer. The producers produce some value (e.g. a measurement) and adds it to the buffer. The consumers wait for something to be added to the buffer, read the element and process it (e.g. shows it on a graph).

LabVIEW offers several synchronization primitives that may be used as a buffer, out of which the *queue* was chosen. Each *node* in this architecture can be a *producer*; a node that enqueues elements on the buffer, or a *consumer*; a node that dequeues elements from the buffer. In the general case there may be more than one buffer. In this case a node may be a *consumer-producer* or even a *producer-producer* or *consumer-consumer*, however the latter two cases are not used in the developed software.

In order to terminate the producer and consumer loops, a second queue is used. This is called the *stop queue*[5]. Enqueueing any element in this queue signals that all loops are to terminate. This is done by either the main application (e.g. when the user presses a stop button) or the nodes themselves (e.g. if an error occours inside a loop).

Figure 5.8 shows an overview of the relationships of these different nodes where two buffers are used.

Consider the case where we have two loops, one producer and one consumer. The producer produces numbers one by one through some process that the consumer is unaware of. The consumer on the other hand uses those numbers for some purpose that he producer is unaware of.

For this example we will assume that the producer simply produces random numbers, and the consumer adds these to a waveform chart. In addition, should anything go wrong during execution of any loop[6] both loops are to terminate. Finally, the front panel should have a stop button to stop both loops and exit the application.

Figure 5.9 shows the block diagram of an implementation using the producer/consumer architecture that satisfied the specification. Note that the producer and consumer loops would benefit from being separate sub-VIs. This would require the use of a reference for the chart in the consumer loop, as discussed in section 5.3.2. Note also that a

[5]This is a special case of a *command queue* which may contain information on an action that should be performed by each loop (e.g. start, stop, initialize, restart etc.).

[6]in this case, nothing really *can* go wrong, but for the sake of demonstration.

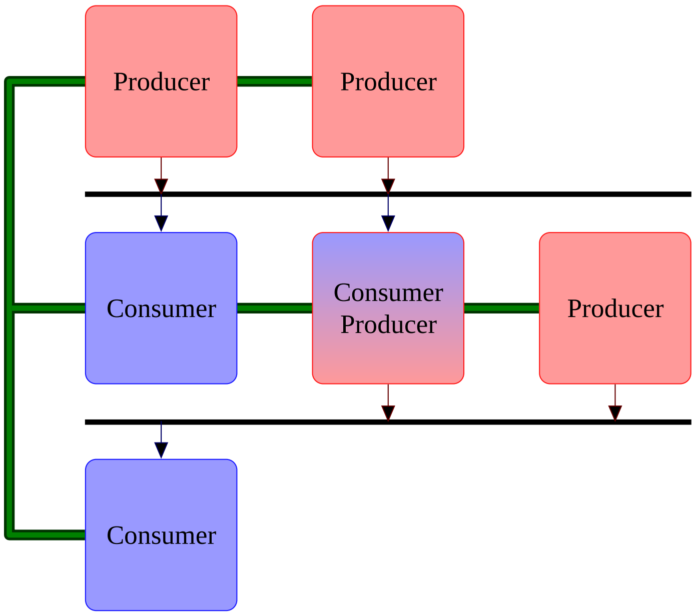


Figure 5.8: Producer consumer architecture overview. The black, horizontal lines represent buffers (i.e. queues) and the green lines represent the stop queue.

lossy enqueue is used in this implementation. This means that if the number of elements in the queue is equal to the buffer size[7], the oldest element in the queue is discarded. This ensures that deadlocks can not occur.

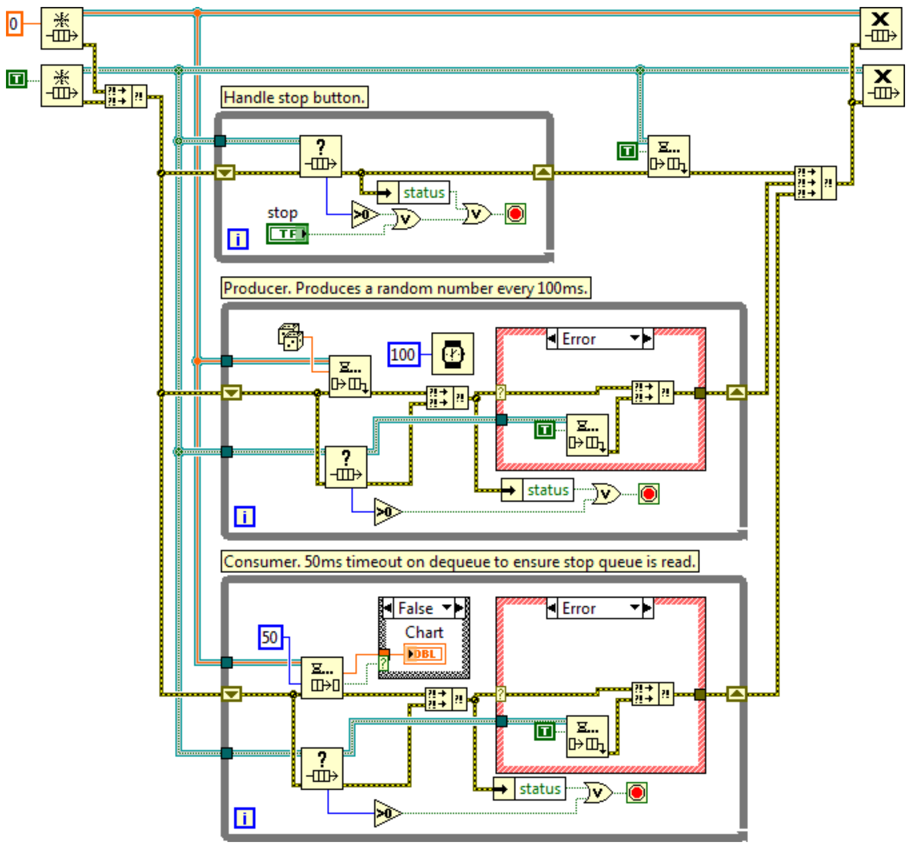

0
Handle stop button.
status
stop
Producer. Produces a random number every 100ms.
100
Error
status
Consumer. 50ms timeout on dequeue to ensure stop queue is read.
50
False
Chart
DBL
Error
status

Figure 5.9: Producer/consumer example block diagram.

[7]Specified upon queue creation. In this example the buffer size is unlimited as no size is specified.

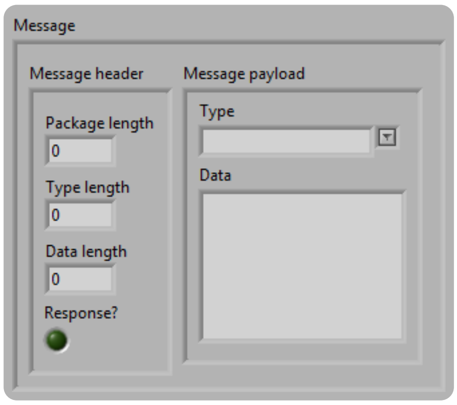
5.4 Host-NXT communication

5.4.1 Overview and terms

The communication between the host and the NXT revolves around the central type definition, *message*. It consists of two parts, a *header* and a *payload*.

The header contains information on the message structure and a flag telling whether the sender awaits a reply or not. This can be calculated based on the payload, with the exception of the response flag which needs to be set separately. The payload consists of a variable-length message type identifier string and a type-dependent data field.

The act of turning a message instance to a string (i.e. a byte array) for sending is called *flattening* the message. Similarly, the act of turning a received string into a message instance is called *un-flattening*. Figure 5.10 shows the message type definition and flattened, binary representation.



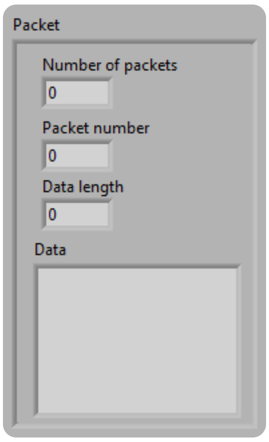
Content	Representation
package length	unsigned byte
type length	unsigned byte
data length	unsigned byte
response?	unsigned byte
message type	string
data	string

Figure 5.10: Message type and flattened structure.

Since the native LabVIEW flatten/un-flatten functions work differently on a PC and the NXT, a custom flattening/un-flattening scheme was implemented.

Because the native LabVIEW NXT communication API supports a maximum of 58 bytes per transmission and a message may be of variable length and larger than this limit, the *packet* type is introduced. The packet type consists of three bytes of structure information (analogous to the message header) and a variable-length data field. A flattened message can be *split* into one or more packets, and the receiver can *join* these to create a flattened message.

Similar to messages, a packet is turned to a string by flattening the packet and the packet type is remade by un-flattening. Figure 5.11 shows the packet type definition and flattened representation.


Packet

Number of packets

0

Packet number

0

Data length

0

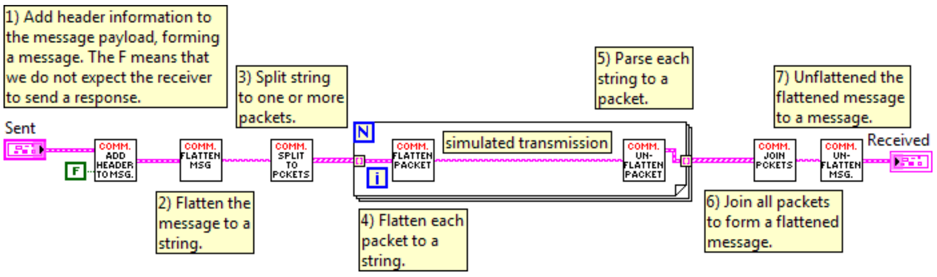
Data

Content	Representation
number of packets	unsigned byte
packet number	unsigned byte
data length	unsigned byte
data	string

Figure 5.11: Packet type and flattened structure.

Implementation and usage

Figure 5.12 shows an example of how the different message type definitions relate to each other in the source code. This example shows how a message payload is turned to a set of flattened strings on the sender side, and how these are turned to a message instance on the receiver side. Note that all the VIs used here are shared between the host and the NXT.


1) Add header information to the message payload, forming a message. The F means that we do not expect the receiver to send a response.

2) Flatten the message to a string.

3) Split string to one or more packets.

4) Flatten each packet to a string.

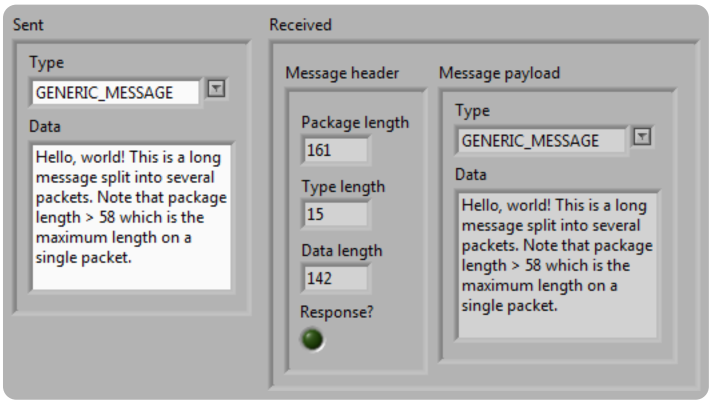
5) Parse each string to a packet.

6) Join all packets to form a flattened message.

7) Unflattened the flattened message to a message.

Figure 5.12: Message/packet example block diagram.

The front panel of this example after it has been run is shown in figure 5.13.



Hello, world! This is a long message split into several packets. Note that package length > 58 which is the maximum length on a single packet.

Hello, world! This is a long message split into several packets. Note that package length > 58 which is the maximum length on a single packet.

Figure 5.13: Message/packet example front panel.

5.4.2 Message payload types

This section will discuss the different message payload types used in the developed software. All the types discussed in this section refer to the *message payload* field of the message type definition in figure 5.10.

Each payload type has its own type definition. The act of going from such an instance to a message payload is called *composing* a message. In the other direction, the act of going from a message payload to an instance of a given message payload type is called *parsing* the message payload.

The following payload types are implemented. The message type field is the same as the payload type name unless otherwise specified.

GET_SERVO_STATE Sent from the host to the NXT. Instructs the NXT to read the position and speed of each servo and transmit a `SERVO_STATE` message back to the host. Shown in figure 5.14.

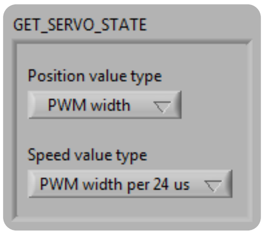
SERVO_STATE If sent from the host to the NXT with the `SET_SERVO_STATE` type field, it instructs the NXT to realize the given servo position and speed set-points. May also be sent from the NXT to the host with the `GET_SERVO_STATE_RESP` type field as a response to a `GET_SERVO_STATE` message. Shown in figure 5.15.

SET_SERVO_POS Sent from the host to the NXT. Instructs the NXT to realize the given servo position set-points. The speed is calculated such that the servos reach their set-points in the same time as the time difference since the last message was received. Shown in figure 5.16.

SET_SERVO_SPEED Sent from the host to the NXT. Instructs the NXT to realize the given servo speed set-points. Based on whether the `increment?` flag is set, the position set-points are chosen to the their maximum or minimum values respectively. Shown in figure 5.17.

SET_SERVO_SPEED_FAST Sent from the host to the NXT. This is a special version of `SET_SERVO_SPEED` that is specifically designed for the 7 servos currently in use. The NXT will also use a servo I/O function specialized for 7 servos. This optimization significantly increases the available servo update rate at the cost of modularity. Shown in figure 5.18.

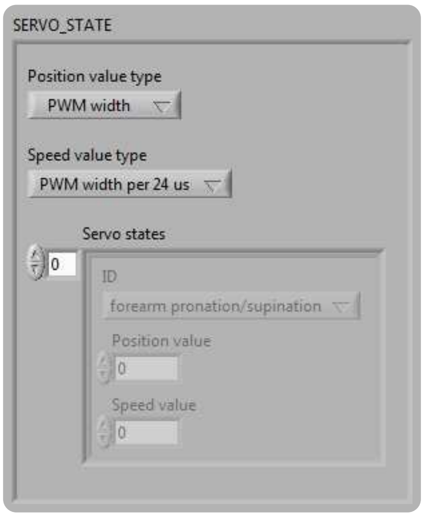
The following figures show the data type and flattened representation of all the message types listed above. The position and speed value types are explained in section 5.5.1. Note that a word here is an integer using two bytes.



GET_SERVO_STATE
Position value type: PWM width
Speed value type: PWM width per 24 us

Content	Representation
position type	unsigned byte
speed type	unsigned byte

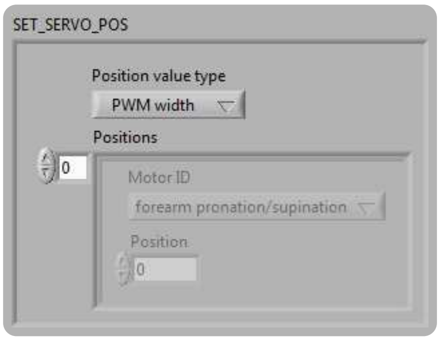
Figure 5.14: `GET_SERVO_STATE` message type and flattened representation.



SERVO_STATE
Position value type: PWM width
Speed value type: PWM width per 24 us
Servo states: 0
ID: forearm pronation/supination
Position value: 0
Speed value: 0

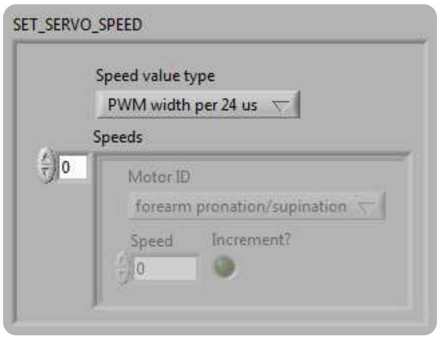
Content	Representation
position type	unsigned byte
speed type	unsigned byte
number of entries	unsigned byte
motor id	unsigned byte
position	unsigned word
speed	unsigned byte
⋮	⋮
motor id	unsigned byte
position	unsigned word
speed	unsigned byte

Figure 5.15: `SERVO_STATE` message type and flattened representation.



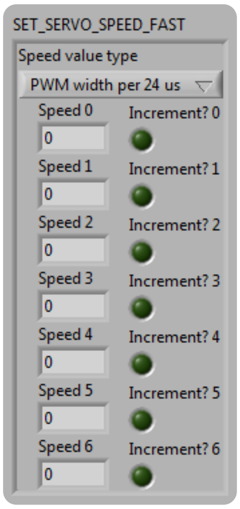
Content	Representation
value type	unsigned byte
number of entries	unsigned byte
motor id	unsigned byte
position	unsigned word
⋮	⋮
motor id	unsigned byte
position	unsigned word

Figure 5.16: `SET_SERVO_POS` message type and flattened representation.



Content	Representation
value type	unsigned byte
number of entries	unsigned byte
motor id	unsigned byte
speed	unsigned byte
increment?	unsigned byte
⋮	⋮
motor id	unsigned byte
speed	unsigned byte
increment?	unsigned byte

Figure 5.17: `SET_SERVO_SPEED` message type and flattened representation.



Content	Representation
value type	unsigned byte
speed motor 0	unsigned byte
⋮	⋮
speed motor 6	unsigned byte
increment? 0	unsigned byte
⋮	⋮
increment? 6	unsigned byte

Figure 5.18: SET_SERVO_SPEED_FAST message type and flattened representation.

Implementation and usage

Figure 5.19 shows an example of how the composers and parsers relate to each other in the source code. Each message type has its own composer and parser. The composers are generally located on the host and the parsers on the NXT. The exception is the SERVO_STATE message type for which the composer and parser are shared.

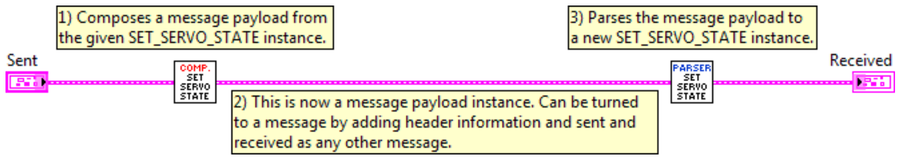

1) Composes a message payload from the given SET_SERVO_STATE instance.
3) Parses the message payload to a new SET_SERVO_STATE instance.
Sent
Received
COMP. SET SERVO STATE
PARSER SET SERVO STATE
2) This is now a message payload instance. Can be turned to a message by adding header information and sent and received as any other message.

Figure 5.19: Compose/parse example block diagram.

An overview of all composers and parsers is shown in figure 5.20. The two on the far right are polymorphic composers and parsers, added purely for convenience.

5.4.3 Send and receive algorithms

Both the host and the NXT have send and read capabilities between each other. The LabVIEW offers a basic API for this purpose, and this is exactly the same for communication over USB and bluetooth. USB communication was used throughout the thesis

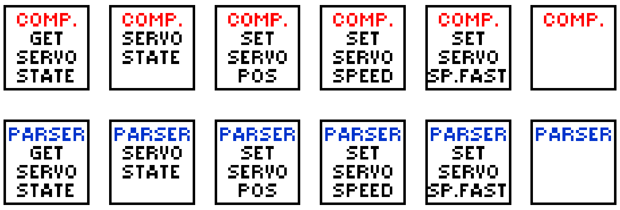


Figure 5.20: Compose and parser VIs. All composers return and all parsers accept a message payload.

work, but bluetooth could be used in the future with absolutely no code change. This API is designed to be robust and “fool proof”. If a message is transmitted, it is guaranteed not to contain any errors when received. A minimum amount of communication initialization is needed. If the USB cable is disconnected during transmission, the program can continue as if nothing happened when it is connected again. However, successful transmission require the receiver to call the read API at the same time as the sender sends a message, and the high-level VIs for this are designed without a timeout.

A communication module was built on this API for both the host and the NXT. This module offers methods to send and receive messages in the types specified in section 5.4.1. For each method a timeout is given to prevent deadlocks in the case that connection is lost during transmission.

There are two send and receive methods implemented, *regular* and *broadcast*. The regular method was designed and implemented first and is designed to be robust and modular. Using this method a message can be of any length and will be transmitted in several smaller packets, where each packet is acknowledged by the receiver.

After this was implemented, the overhead delay in sending a single byte message was found to be significantly higher than expected. To optimize the servo update rates the newer and faster method, *broadcast*, was implemented. This method operates on the same message type, but a maximum of 58 bytes can be transmitted. The message is still acknowledged by the receiver. Due to lack of time, the broadcast method is only implemented as a receiver on the NXT and sender on the host.

Regular method

Figure 5.21 shows a state diagram of the send algorithm. The algorithm starts in `trans_req` where it sends the string `trans_req` until it is acknowledged. It then loops in the `trans_pkg` state until all packets are sent, and waits for an `ACK` string for each packet.

If at any time something goes wrong (i.e. nothing or something unexpected is received) the function either falls back to the `trans_req` state or exits, depending on whether the time limit of the function has been reached.

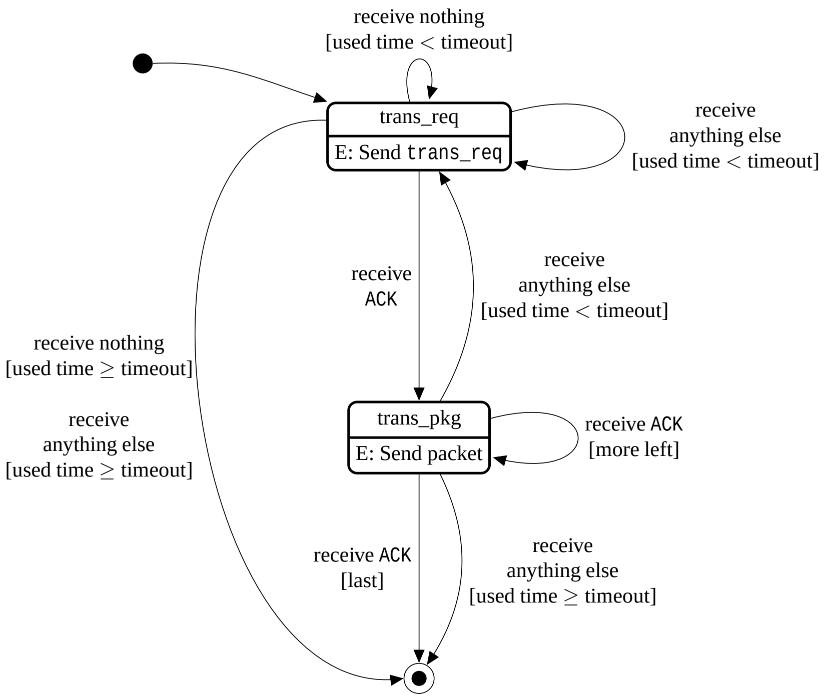


Figure 5.21: Send message state diagram. *E:* denotes an action performed when entering the state.

Figure 5.22 shows a state diagram of the receive algorithm. The algorithm starts in `rec_req` where it waits for the string `trans_req` to be received. It then loops in the `rec_pkg` state until all packets are received, and transmits an `ACK` string upon receiving a packet.

If at any time something goes wrong (i.e. nothing or something unexpected is received) the function either falls back to the `rec_req` state or exits, depending on whether the time limit of the function has been reached.

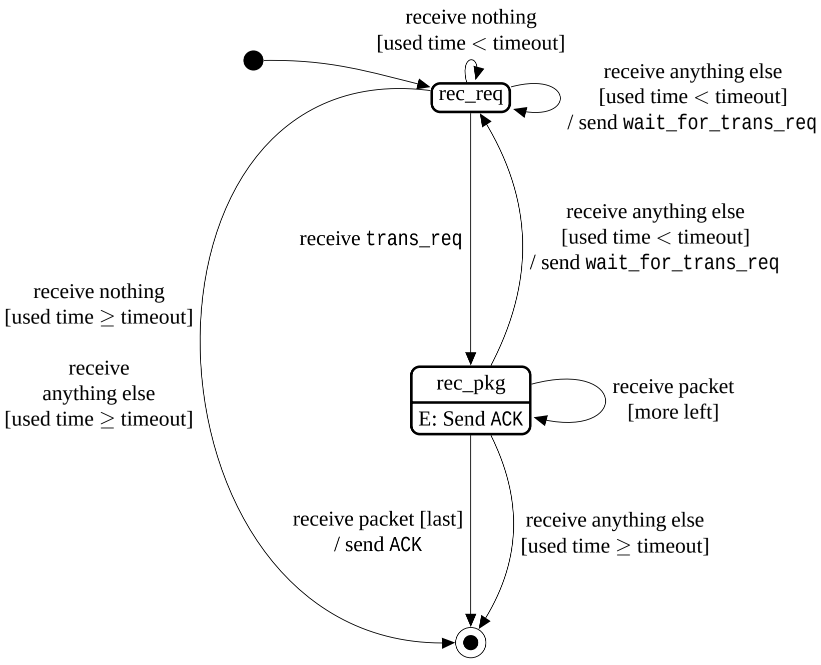

receive nothing [used time < timeout]
rec_req
receive anything else [used time < timeout] / send wait_for_trans_req
receive trans_req
receive anything else [used time < timeout] / send wait_for_trans_req
receive nothing [used time ≥ timeout]
receive anything else [used time ≥ timeout]
rec_pkg
E: Send ACK
receive packet [more left]
receive packet [last] / send ACK
receive anything else [used time ≥ timeout]

Figure 5.22: Receive message state diagram. *E:* denotes an action performed when entering the state.

Note that the timeouts are only checked when something unexpected occurs. This means that, if there are no packet losses, once the packet transmissions are started, they will complete successfully even if the functions take longer to complete than the specified timeout.

These functions are implementer separately for the host and NXT, as they use different low-level communication APIs. However, they both follow the state diagrams discussed here.

Broadcast method

The broadcast VIs have the same signatures as the regular method VIs. They differ in that only 58 bytes may be transmitted. It is made for optimization and should be used whenever one knows at compile-time that a message may never exceed this length.

The sender in this case simply re-sends the message at regular intervals until it times out or receives an acknowledgment. Similarly, the receiver reads until it times out or receives a message where it sends an acknowledgment and returns.

Implementation and usage

The VIs in figure 5.23 are responsible for communication. All send and receive messages operate on the message type definition shown in figure 5.10.

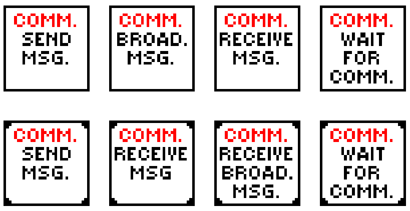


Figure 5.23: Send and receive VIs.

5.5 NXT software

5.5.1 Servo controller

Low-level API

The NXTServo-v2 controller comes with LabVIEW support available online[8]. This is simply a wrapper for the NXT I^{2}C API with some useful register mappings.

This code was used as provided by the manufacturer, but upon inspection several suboptimalities were found. For one, both the servo speed and position values are written even if only one of these values were changed, and the servo voltage is always read.

Considering that there seems to be a several millisecond overhead/delay in calling the I^{2}C API, this can become a bottleneck when handling fast update rates. To overcome this a separate writing mode, called the *fast* mode, was implemented in addition to the supplied API. The fast mode is used with the `SET_SERVO_SPEED_FAST` message type

[8]http://www.mindsensors.com/index.php?module=pagemaster
&PAGE_user_op=view_page&PAGE_id=93

and differs from the manufacturer's API in that it assumes exactly 7 servos and is able to write the position and speed of all servos in only two write operations.

Each servo is enumerated in `motor_id.ctl` which is shared between and used throughout the host and the computer software. The servo numbers follow the definition in table 1.2.

NXTServo-v2 controller

The NXTServo-v2 controller operates on PWM width in μs for position set-points and PWM width change in μs per 24 μs for speed set-points. If a position set-point P is written and the associated speed register S is greater than zero, the controller increases or decreases the PWM width by S each 24 μs until the position P is reached. If $S = 0$ then P is written at once and the servo moves as fast as possible.

The servos accept pulse widths between 500 μs and 2500 μs, although the construction of the hand limits the movement of the servos much more than this. The position set-points are given in a 16-bit unsigned word, and the speed set-points are given in a single byte.

When the servo controller receives power it initializes by writing 1500 μs to all servos. This is a standard neutral position for the servos, but as it is not the natural position of the hand, a separate initialization is done when the NXT application starts.

Set-point value types

The host should not have to worry about the pulse widths on the servos, so there is a need for an additional, more useful unit type for both position and speed. The NXT therefore operates with the *position value type* and *speed value type* type definitions. Conversions between the different types are done by the VIs shown in figure 5.24.

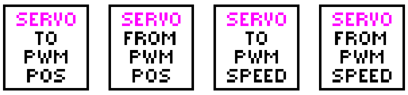


Figure 5.24: NXT servo value type conversion VIs.

As the maximum and minimum servo PWM widths are NXT-specific information, all unit types are available when sending set-points between the host and the NXT. Note that in section 5.4.1 all message types are shown to accept an unsigned word for position set-points and an unsigned byte for speed set-points. Floating-point numbers with a known range (i.e. -1 to 1 or 0 to 1) are transformed to an unsigned byte/word by linear interpolation (e.g. floating point numbers between 0 and 1 are mapped to unsigned bytes between 0 and 255) before sending. This is done by the VIs shown in figure 5.24.

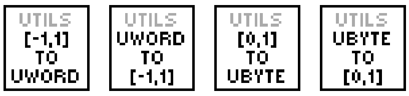


Figure 5.25: Integer/double interpolation utilities.

The only currently implemented and used position type, in addition to the raw PWM width, is the *Unit [-1...0...1]* position type. This type represents the position with a floating point number from -1 to 1, where 0 is a neutral state and -1 and 1 are the extreme positions. For servos that operate in only one direction from the neutral position the range from 0 to 1 is used. For values in between a linear interpolation is performed to find the corresponding pulse width.

The only currently implemented and used speed type, in addition to the raw PWM width change, is the *Unit [0...1]* speed type. This type represents the speed as a fraction of the maximum speed value.

Servo limits

The servo limits are configured in `servo_get_limits.vi`. For each servo the -1, 0 and 1 position points are defined, which makes up the maximum, minimum and neutral value. Note that some servos rotate clock-wise and others counter-clockwise, meaning that the -1 value may be higher or lower than the 0 and 1 values. Some servos operate in one direction only, and this is represented by having the same value for the -1 and 0 or 1 and 0 values. In addition, the maximum speed is defined for each servo.

Integration with other NXT software modules

All the servo I/O operations are encapsulated in the functional global variable `var_servo_state.nxt.vi`. This VI remembers the state of each servo and allows other VIs to read and write the state of all servos or a single one. It also supports the *fast* mode discussed earlier. Figure 5.26 shows the most important VIs for this purpose.

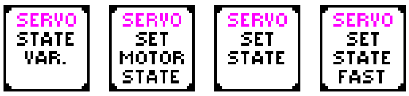


Figure 5.26: NXT servo VIs.

5.5.2 Controller mode

The controller mode combines the communication and servo I/O module parts to form the perform the main functionality of the NXT. The NXT starts my initializing the servos to their neutral state, before entering an eternal loop. In this loop a message is received and acted upon. No receive timeout is specified, so the program will run until aborted from outside (i.e. the user presses the back button on the NXT).

Figure 5.27 shows the top-level block diagram with explanations. Note that the broadcast communication mode is used in this block diagram. Regular mode can be used by replacing the VI labeled `2)` with `comm_receive_message.nxt.vi`.

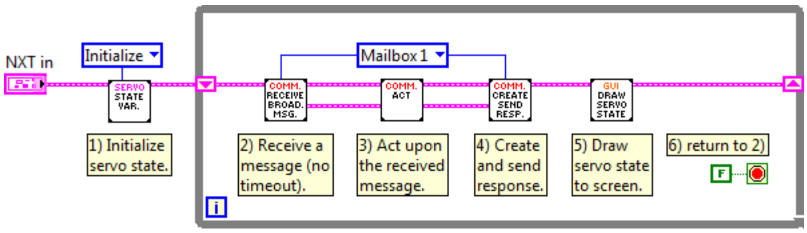

NXT in
Initialize
SERVO STATE VAR.
1) Initialize servo state.
Mailbox 1
COMM. RECEIVE BROAD. MSG.
2) Receive a message (no timeout).
COMM. ACT
3) Act upon the received message.
COMM. CREATE SEND RESP.
4) Create and send response.
GUI DRAW SERVO STATE
5) Draw servo state to screen.
6) return to 2)
F
i

Figure 5.27: Controller mode top-level block diagram.

5.5.3 Demo mode

When the NXT enters the demo mode, it starts by reading a specified demo file. It first reads the very first bytes of the file that constitutes a mode identifier string. This mode identifier may be either `POS` or `SPEED`, and determines the further structure of the file as well as the operation of the NXT. Table 5.1 shows the structure of these files.

The NXT then operates by reading a set of bytes which is converted to a set of set-points based on the position/speed value type. These set-points are then actuated and the NXT continues reading the file. The delta time flag of the file specifies the *minimum* time interval for the onset of each set-point set. Given the non-deterministic execution time, there is no guarantee that the actually used time will not exceed this limit.

When the set-points are positions, the speed is calculated such that the servos reach their destinations in the same time as the time since the last set-points were read.

Note that the `SPEED` mode was not fully implemented. The file format is supported and speed-set points are supported by the servo module, so this is exclusively due to time constraints.

Table 5.1: Demo file structures.

(a) Position mode

Content	Representation
mode length (3)	unsigned byte
mode (`POS`)	string
value type	unsigned byte
delta time [ms]	unsigned byte
number of servos (n)	unsigned byte
setpoint 0	unsigned word
⋮	⋮
setpoint *n*	unsigned word
setpoint 0	unsigned word
⋮	⋮
setpoint *n*	unsigned word
⋮	⋮

(b) Speed mode

Content	Representation
mode length (5)	unsigned byte
mode (`SPEED`)	string
value type	unsigned byte
delta time [ms]	unsigned byte
number of servos (n)	unsigned byte
setpoint 0	unsigned byte
⋮	⋮
setpoint *n*	unsigned byte
setpoint 0	unsigned byte
⋮	⋮
setpoint *n*	unsigned byte
⋮	⋮

5.6 Host software

5.6.1 Architectural overview

The main application consists of several *sub-applications*. A sub-application is the top-level module type in the application architecture. Each sub-application is a VI whose front panel is never shown and that never returns (until the application is aborted). All GUI controls that a sub-application needs are given as references when the VI call is made. A sub-application communicates with the rest of the application through functional global variable VIs. The complete application consists of the following sub-applications; Measurement, Training, Calculation and Demo.

The other main module type is the *GUI handler*. These operate similarly to sub-applications, except that they do not communicate with any other part of the system. GUI handlers operate exclusively on the graphical user interface (e.g. enables or disables choices based on a configuration selection). They may read from but not write to global variables. The general structure of a GUI handler is an event structure with a dynamic event terminal surrounded by a never-ending while loop.

An illustration of the modularization of the main application is shown in figure 5.28.

The main application works by first initializing itself and then calling every sub-application and GUI handler in parallel. In this way, all sub-applications contain their

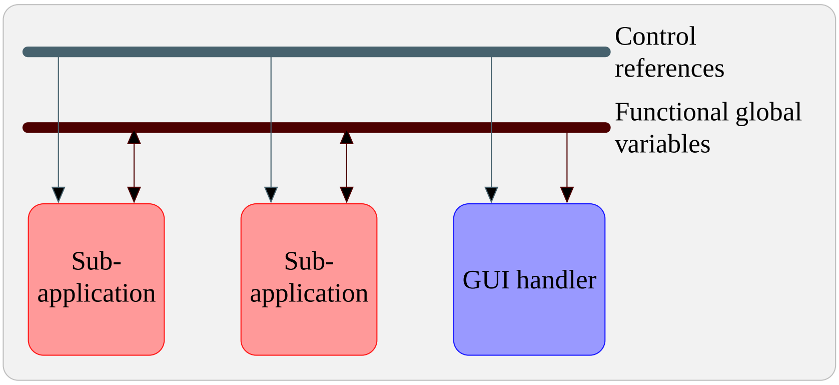


Figure 5.28: Application architecture overview. Note that there may be several sub-applications and GUI handlers.

own internal state. The only loop running in the main application after this is a loop containing the boolean controls configured with latching mechanisms. The LabVIEW architecture demands that these be read only one place in the code (i.e. to avoid race conditions), and this must be in the VI that contains the controls (i.e. the main application VI).

5.6.2 Measurements

The EMG and accelerometer measurements are acquired by two National Instruments data acquisition modules. These both support the NI DAQmx measurement drivers which are integrated with LabVIEW.

Before a measurement is started, a measurement *task* is created. To this task all the analog input channels are registered, with a name, unique input pin identifier and a maximum and minimum value.

For each channel the *non-referenced single-ended (NRSE)* measurement mode is used. This means that each channel is measured with respect to a single-node analog input sense (AISENSE) that is independent on the measurement system ground.

After all channels are registered to the task, a timer is registered as well. The following two sample modes are used.

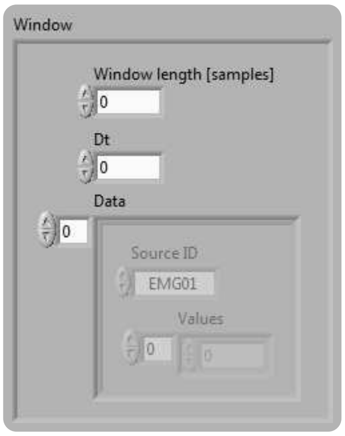
Finite samples Acquires a given number of samples. Uses hardware timing to ensure that exactly the given number of samples are returned. Infers a delay of about 10 ms for each read operation. Read operations are blocking calls.

Continuous samples Starts acquiring samples and saves them to a hardware buffer. Once the task is started, each consecutive call to get the samples will return the buffered samples. Runs until the task is stopped. Ensures that no samples are lost, as long as the buffer does not overflow. Read operations are non-blocking and may return zero samples.

To ensure that no data is lost in between read operations, the continuous sample mode is used when the system is running. For the training sequence, however, the finite samples mode is used, as only one read operation per class is needed and this ensures acquisition of an exact number of samples.

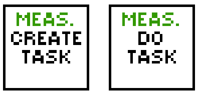
Implementation and usage

The output of the DAQmx call is an array of waveforms[9]. However, the rest of the application operate on a custom type definition *window*. This type definition is shown in figure 5.29, as well as the VIs used for measurement. The block diagram of an example using these VIs is shown in figure 5.30.



(a) *Window* type definition.

(b) Window/waveform VIs.



(c) Measurement VIs.

Figure 5.29: Measurement type definitions and VIs.

5.6.3 Feature calculations

Since features are calculated by mathematical expressions and algorithms, a textual language is much better suited for this job than LabVIEW. Because of the mathematical

[9]A waveform is a standard LabVIEW data structure containing an array of floating point numbers, a delta time flag and some metadata.

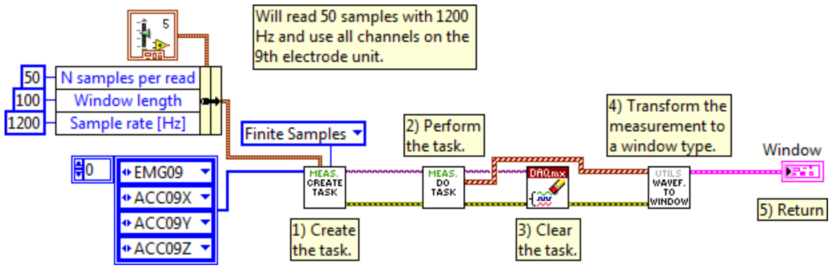


Figure 5.30: Measurement example block diagram.

aspects of feature calculations, MATLAB was the language of choice for feature implementation. As mentioned, LabVIEW offers a tight integration with MATLAB, and reference MATLAB code for many features are readily available from the works of (Håkonsen 2010).

For future use it is important that features may be added, changed or removed without editing the software code. To achieve this the application is configured with the path to a directory containing a set of MATLAB `m`-files. Every `m`-file in this directory is then assumed to be a MATLAB function with a known function signature.

All features accept two arguments and return a single result value. The first argument, `data`, is a matrix where each column is a time series from a separate source. The second argument, `config`, is a structure containing the sample rate and the ID of each source (see section 5.6.5). All features must return a vector of real numbers with at least one element.

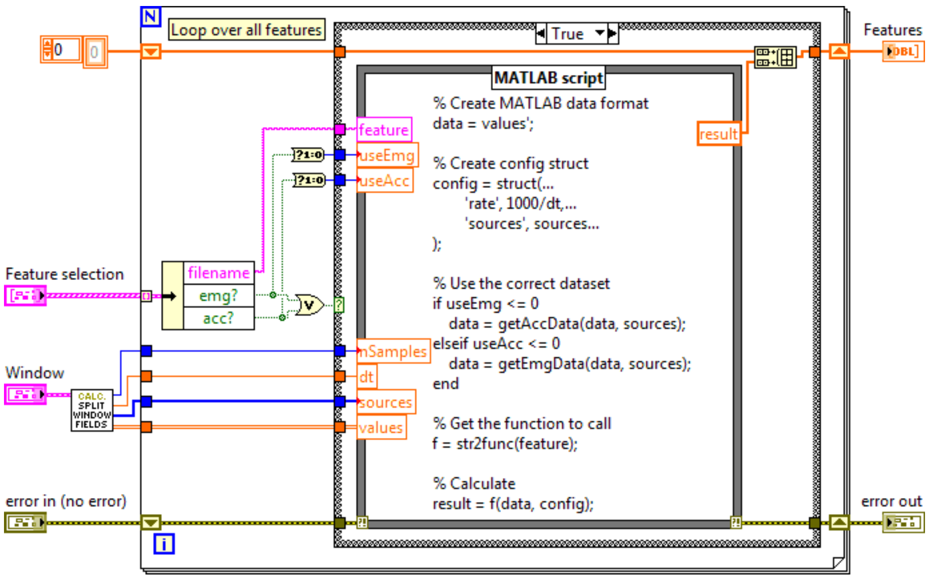
When the feature directory is added to the MATLAB search path, which is done programmatically, a function can be called based on its name. This is done by the following code

```
% Get the function to call
f = str2func(feature);

% Calculate
result = f(data, config);
```

where `feature` is the feature function name (file name excluding extension).

Because some features are specific to either EMG or accelerometer signals, the user may select what the input of each feature should be. This can be either EMG only, accelerometer only, both or none. If no inputs are chosen for a feature, this feature is simply not calculated.

The application can then loop over this selection, create the correct sub-set of input signals for each feature, and concatenate all outputs to create the final feature vector. This is performed by `calc_extract_feature_vector.vi` and the implementation is shown in figure 5.31.



```
% Create MATLAB data format
data = values';

% Create config struct
config = struct(...
    'rate', 1000/dt,...
    'sources', sources...
);

% Use the correct dataset
if useEmg <= 0
   data = getAccData(data, sources);
elseif useAcc <= 0
   data = getEmgData(data, sources);
end

% Get the function to call
f = str2func(feature);

% Calculate
result = f(data, config);
```

Figure 5.31: `calc_extract_feature_vector.vi` block diagram.

The following features were implemented in MATLAB and are available in the application; AAC, AAV, AR, CC, HIST, MYOP, NT, VAR, WAMP and ZC.

5.6.4 Classifier training and classification

To enhance the future usefulness of the software, the software architecture is specified to handle several different types of classifiers. The easiest and most straightforward way of doing this is to make each classifier have their own set of VIs, and use a switch-case on some classifier type identifier whenever the system needs to access these VIs. The obvious drawback of this method is that whenever a new classifier is introduced a new case must be added everywhere it is used in the code.

Another option is to use MATLAB scripts in the same way as is done for features. However, classifiers can be a much more complex than features. Classifiers may for instance be adaptive, have a state and require memory in between executions, whereas features are stateless.

What is needed is a structure that allows the software to operate on a definition of what a classifier should be able to do (i.e. be trained and perform a classification), without knowing anything about how it is implemented. This structure must be able to store it's own data structure (e.g. a matrix for the LDA and LMS classifiers). This is a textbook example of where the *object oriented* design pattern is most useful.

Object oriented programming (OOP) is, in National Instrument's own words, a new and advanced addition to LabVIEW. Although there are significant differences between OOP in LabVIEW and in a text-based language such as Java or C++, the most fundamental aspects remain the same. These include

- Encapsulation of a data structure.
- Member functions, both static and non-static.
- Class hierarchy, including overriding functions.
- Member function visibility, including public (i.e. visible to all other functions), private (i.e. visible to the class only) and protected (i.e. visible to the class and all sub-classes only).
- Casting to a more specific (i.e. towards the base class) or to a more general (i.e. towards the child class) class version.

LabVIEW classes are contained in a `.lvclass` file, and can be instantiated from a file path. This makes it ideal for use in this specific case, as the application only needs to know where the class is located on disk to use it. Thus, when a new classifier is added and to be used, the application is simply given this new path instead of the old one, and no further code change is needed[10].

Implemented classifiers

Figure 5.32 shows a class diagram of the two implemented classifiers. The base class, `Classifier`, has two member fields. These store the configuration present at the time of training. Only those that are critical to the execution (i.e. the feature and class selection) are stored. Thus, the classifier is not concerned about other configurations (e.g. window length, sampling rate etc.) even if changes here may produce sub-optimal classification. Both of these fields are available for reading trough public getter functions. However, since the fields are private, they may not be written to by any other class than `Classifier`, which writes these in the `train` function. To ensure that these are written correctly, all sub-classes are required to unconditionally call the base implementation in the `train` function. This condition is supported by LabVIEW OOP, and any sub-class that fail to adhere to this demand will give a compile error.

The internal data structure for both the `Classifier LDA` and `Classifier LMS` is a matrix of real numbers, called `G`. This field is private and is only used in the respective `classify` overrides.

[10]In fact, a classifier could be created outside the context of the application project. However, this is unpractical simply because the input and output values of the member functions are typedefs contained in the project.

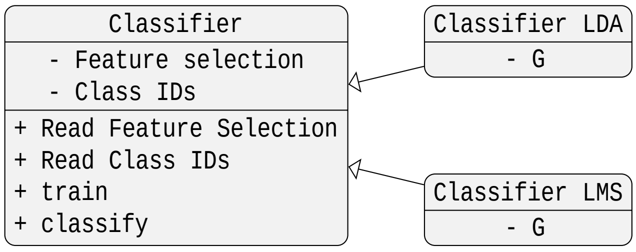


Figure 5.32: Class diagram of the classifier classes.

Both classifiers are implemented using a MATLAB back-end. In the LDA classifier an existing MATLAB implementation was used, whereas a new implementation was done for the LMS classifier. As both are linear classifiers, classification boils down to a matrix multiplication, and both classifiers use the following code for classification, contained in the `doClassify` function.

```
% Actual classification
[val, i] = max(G*[x 1]');

% Get classId
id = classIds(i);
```

The LMS classifier training boils down to a pseudo-inverse matrix multiplication. Training is performed by the `trainLMS` method containing the following code.

```
% Create the training matrices
classIds = unique(target);
nVal = size(data,2);
Y = zeros(nClass,nVal);
for iClass=1:nClass;
    class = classIds(iClass);
    Y(iClass,:) = (target == class);
end
X = [data' ones(nVal,1)];

% Actual training
G = (X\Y')';
```

Implementation and usage

A class member function VI looks just like a regular VI, but it has an instance of the object as an input and output. The `train` and `classify` function VIs for the LMS classifier are shown in figures 5.33 and 5.34 respectively. Note that the only difference between these VIs and those of the LDA is the type of the object input and output, and that the `trainLMS` function is replaced with `trainLDA`.

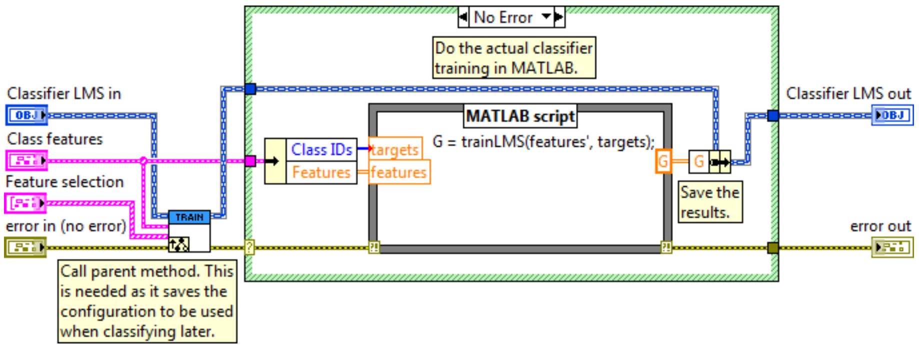

No Error
Do the actual classifier training in MATLAB.
Classifier LMS in
Class features
Feature selection
error in (no error)
TRAIN
Class IDs
Features
targets
features
MATLAB script
G = trainLMS(features', targets);
G
G
Save the results.
Classifier LMS out
error out
Call parent method. This is needed as it saves the configuration to be used when classifying later.

Figure 5.33: LMS `train` function VI.

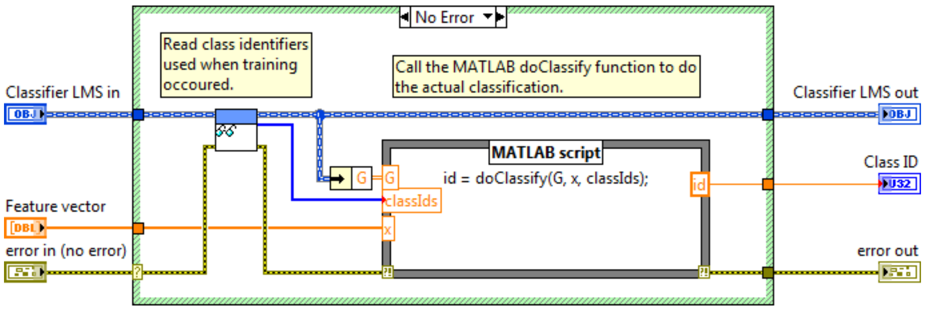

No Error
Read class identifiers used when training occoured.
Call the MATLAB doClassify function to do the actual classification.
Classifier LMS in
Feature vector
error in (no error)
MATLAB script
id = doClassify(G, x, classIds);
G
G
classIds
x
id
Classifier LMS out
Class ID
error out

Figure 5.34: LMS `classify` function VI.

5.6.5 Application configuration files

Signal routing

All channel names are enumerated in `source_id.ctl`. The EMG channels are labeled `EMG01`, `EMG02`, ..., `EMG16`. The accelerometer channels are labeled `ACC01X`, `ACC01Y`, `ACC01Z`, ..., `ACC16X`, `ACC16Y`, `ACC16Z`. The numbers refer to the numbers on the Trigno electrode units.

The channel routing is located in `./Computer/Config/sources.txt`. Each row in this file assigns an enumerated channel name to a LabVIEW task channel in the following format; `<channel id>=<task channel>`.

As an example, the following lines assigns the EMG channel of the 9'th and 10'th electrode units to the first and second analog input pins of the module with id `Dev4`:

```
EMG09=Dev4/ai0
EMG10=Dev4/ai1
```

Classes

All classes have an unique string identifier, a label and an illustration file name. The available classes are located in `./Computer/Config/classes.txt`. Each row in this file adds a new class available for training in the following format; `<id> <label> <image>`. Note that the white-space characters are tabular characters. The image field is the absolute path to a `jpg` file.

As an example, the following lines creates the "wrist flexion" and "wrist extension" classes:

```
WRIST_FLEXION       Wrist flexion       <some path>\wrist flexion.jpg
WRIST_EXTENSION     Wrist extension     <some path>\wrist extension.jpg
```

Servo set-points

Each class can have a set of servo set-points associated with it. These set-points are located in `./Computer/Config/servo_setpoints.txt`. Each row in this file associates a class with a set of servo set-points in the following format; `<id> <setpoints>`. Note that the white-space character is a tabular character. The `setpoints` field consists of 7 floating point numbers separated by white-space. The i'th number in this list gives the set-point of the i'th servo, enumerated in `motor_id.ctl` discussed in section 5.5.1. The set-points are given in the "unit" speed type, as discussed in the same section. The class identifiers must match those defined in the class-file discussed above.

As an example, the following lines defines the wrist flexion and wrist extension set-points.

```
WRIST_FLEXION        0   0  1  0  0  0  0
WRIST_EXTENSION      0   0 -1  0  0  0  0
```

5.6.6 Sub-application: Measurement

The measurement sub-application is a very simple one. Its function is to measure the selected signals with the selected source configuration, and display the results to the user. Figure 5.35 shows a state diagram of the sub-application.

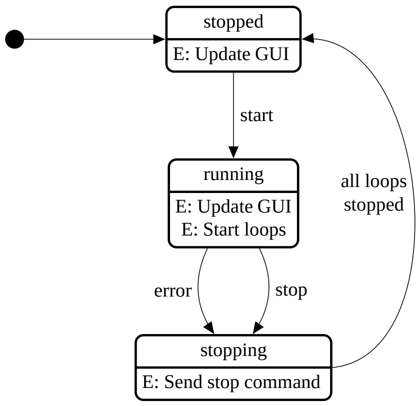


Figure 5.35: Measurement sub-application state diagram.

The sub-application is implemented as a producer-consumer architecture. When the measurement is running a producer communicates with the DAQmx drivers to acquire a set of measurements that are then added to a buffer. From this buffer a consumer pulls the measurements and displays them in a graph.

In addition, a GUI handler monitors the buffer and displays its state (i.e. the number of elements waiting to be consumed). This is useful as it illustrates whether there is enough processing power to perform all tasks without data loss or delay.

5.6.7 Sub-application: Training

The training sub-application handles all the training-related aspects of the application. When a training sequence is started with a given configuration, a class sequence is created based on the selected classes.

For each class a producer-consumer relationship is established with a measurement and a GUI loop. When the measurement is done the loops exits and the data is added to a training data set. Once all classes are trained, the data set is saved to a functional global variable, so that it may be used by other sub-applications.

The training may be paused or aborted at any time. In the paused state the user may return to a previous class and continue the training from there, overwriting existing data.

If the target functional global variable already contains training data, the user may either overwrite with, append with or discard the new data. Upon pressing the abort button, the user is prompted with a warning message, and may return to the training if the abort command was a mistake on his or her part.

Figure 5.36 shows a state diagram of the sub-application.

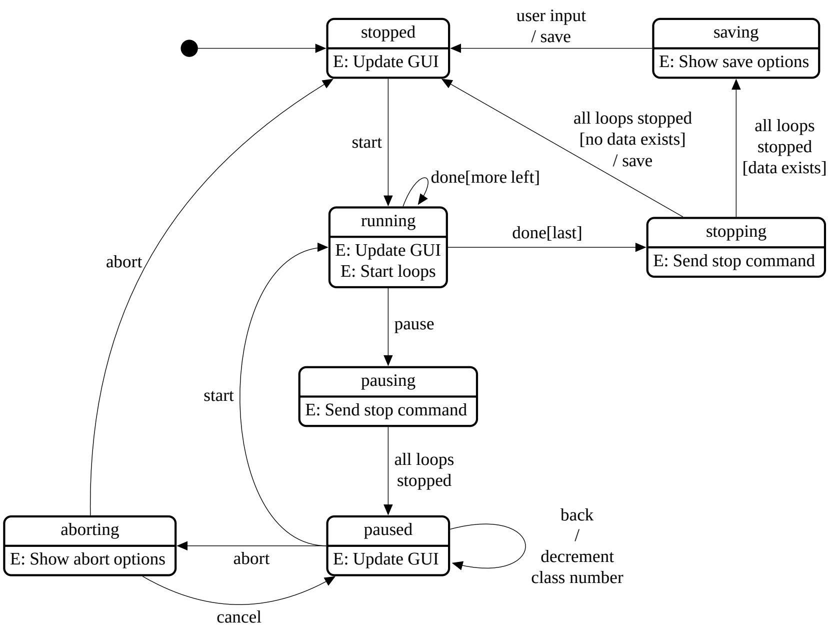


Figure 5.36: Training sub-application state diagram.

5.6.8 Sub-application: Classifier training

The classifier training sub-application calculates a classifier based on the available training data and the current configuration. Since this is done in a single step, it is the simplest sub-application, coding-wise, as no producer-consumer relationship is needed.

Figure 5.37 shows the block diagram of this sub-application VI. Note the dynamic event registration and global functional variable usage for both the training data and current classifier.

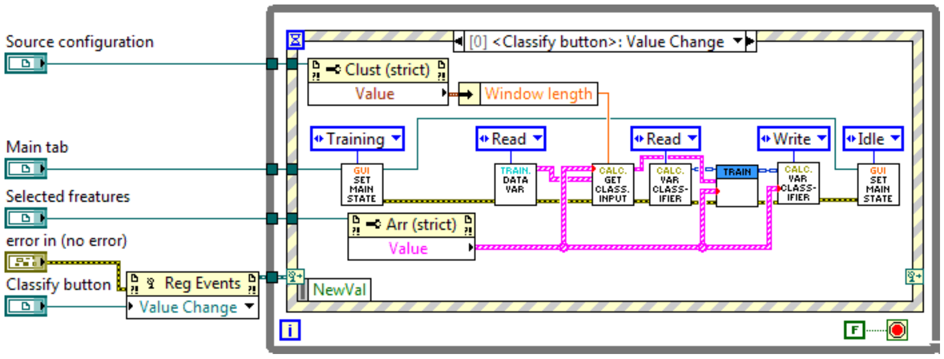


Figure 5.37: Classifier training sub-application block diagram.

5.6.9 Sub-application: Demo

The demo sub-application brings the measurement modules, the trained classifier and the NXT together to function as the demonstrator that is the original purpose of the application. Measured signals are classified and the appropriate servo set-points are communicated to the NXT, which actuates these on the servos.

Figure 5.38 shows an overview of the communication flow between the different modules and loops during normal execution. The signals are explained in table 5.2.

Again a producer-consumer architecture is used. The measurement loop produces a measurement by contacting the DAQ hardware, and adds these measurements to a buffer. The calculation loop consumes the measurement from this buffer and produces a class identifier which is added to a second buffer. Lastly, the communication loop consumes a classification from this buffer and communicates the appropriate servo set-points to the NXT.

The NXT simply awaits a communication, and upon receiving, actuates the received set-points on the servos.

The status of each buffer is displayed in the same way as in the measurement sub-application.

Figure 5.39 shows the block diagram of `demo_run.vi`. This function initializes and runs the producer-consumer architecture for the demo sub-application. Note that in addition there is a GUI controller handling the stop-button, but this is outside the scope of this function. The communication loop is only created if the user enables it, meaning that the demonstration must be restarted in order for changes in this setting to take effect. This is done because the communication loop tries to initialize the communication, resulting in an error if the NXT is not connected.

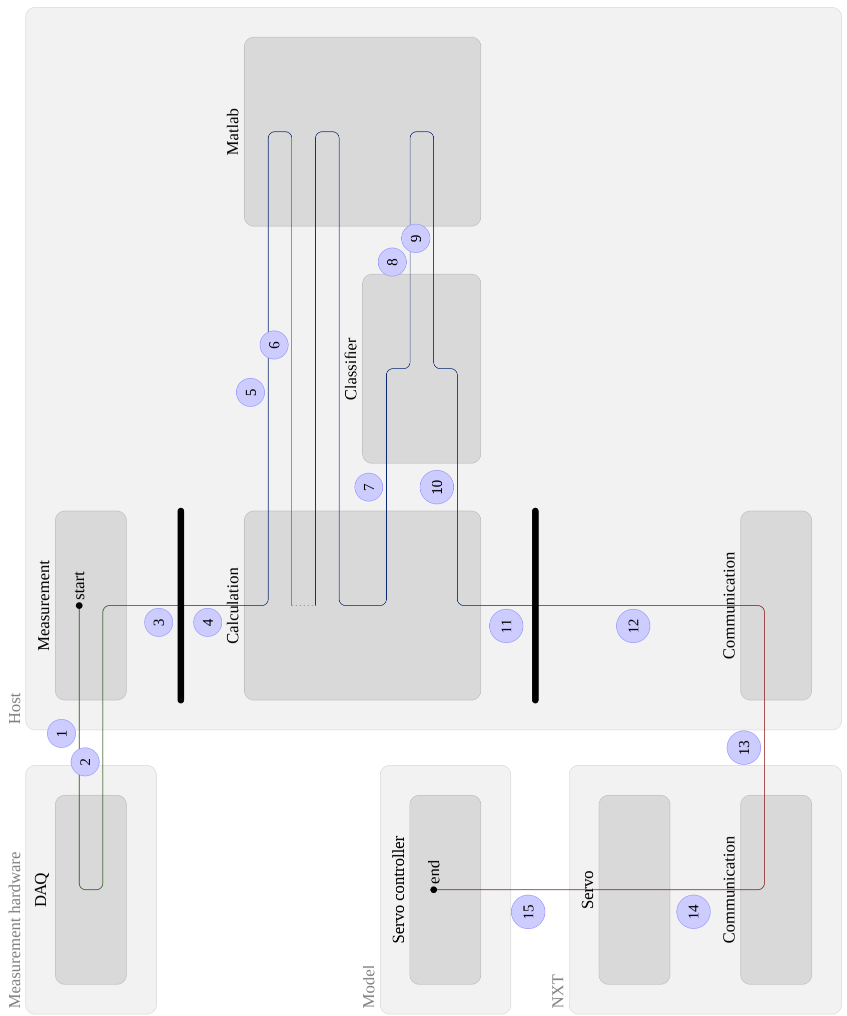


Figure 5.38: Standard operation information flow.

Table 5.2: Explanation of labels in figure 5.38

Label	Explanation
1	The measurement loop makes a call to the DAQmx drivers that communicate with the DAQ hardware.
2	The call returns a set of measured time series.
3	The measured window is added to the measurement buffer.
4	The calculation loop polls the measurement from the buffer.
5	A MATLAB script is called for each feature.
6	Each MATLAB script returns an array of feature values.
7	A feature vector is handled to the classifier instance.
8	The classifier may call a MATLAB script to perform the calculations.
9	The MATLAB script returns a class identifier.
10	The classifier returns a class identifier.
11	The class identifier is added to the communication buffer.
12	The communication loop polls the class identification from the buffer.
13	A set of servo set-points corresponding to the class identifier are communicated to the NXT.
14	The commuication loop on the NXT receives and parses the set-points and gives them to the servo module.
15	The servo module communicates with the servo controller over I^2C to actuate the set-points.

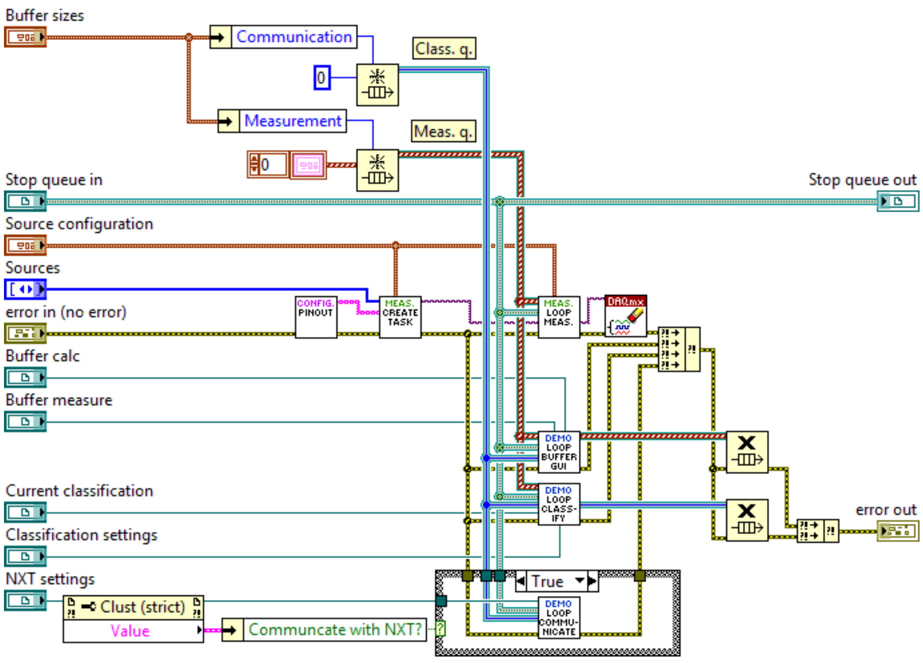


Figure 5.39: Demo sub-application core top-level block diagram.

6 Results

6.1 NXT software

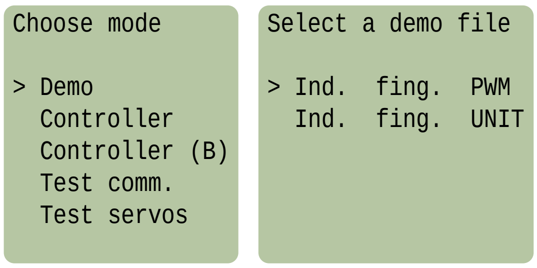
All the NXT functionality is contained in a single application. Upon entering this application a simple menu is displayed, operated by the arrow and enter buttons on the NXT. There are two main modes, “demo” and “controller”. The controller mode has a second parameter which is the send/receive method the application will use. In addition, a servo test mode is available.

When selecting the demo mode, a new menu is shown where the user may select which demo file to run[1]. Screenshots of these menus are shown in figure 6.1.

When the controller mode is entered the servos are initialized to their neutral position, and the NXT starts listening to communication from the host. After receiving a message and updating the servos, the servo state is displayed on the screen as `<pos> / <speed>` with values given in PWM width. Screenshots of these menus are shown in figure 6.2.

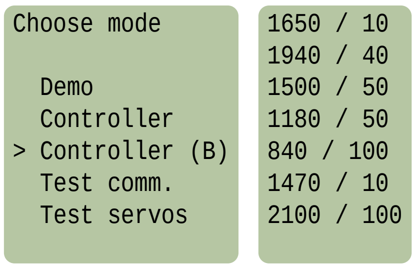
The servo test application is useful when changes to the model has been done. It allows manual position control of each servo. After selecting a servo the position can be increased or decreased by pressing the arrow buttons. Pressing the enter button returns to the servo menu. Screenshots of these menus are shown in figure 6.3.

[1] Note that this menu must be changed in code as the NXT does not support file listings programmatically.



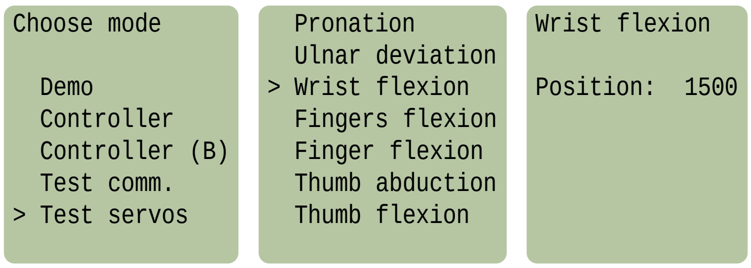
```
Choose mode             Select a demo file

> Demo                  > Ind. fing. PWM
  Controller              Ind. fing. UNIT
  Controller (B)
  Test comm.
  Test servos
```

Figure 6.1: NXT menu, “demo” application.



```
Choose mode             1650 / 10
                        1940 / 40
  Demo                  1500 / 50
  Controller            1180 / 50
> Controller (B)        840 / 100
  Test comm.            1470 / 10
  Test servos           2100 / 100
```

Figure 6.2: NXT menu, “controller” application.



```
Choose mode               Pronation          Wrist flexion
                          Ulnar deviation
  Demo                  > Wrist flexion      Position:  1500
  Controller              Fingers flexion
  Controller (B)          Finger flexion
  Test comm.              Thumb abduction
> Test servos             Thumb flexion
```

Figure 6.3: NXT menu, “test servos” application.

6.2 Host software

The host application is designed to allow the user to efficiently configure, train and run the controller. It is easy to use and feels robust and bug-free. Screenshots of the different aspects of the application user interface are shown in figures 6.4 through 6.11.

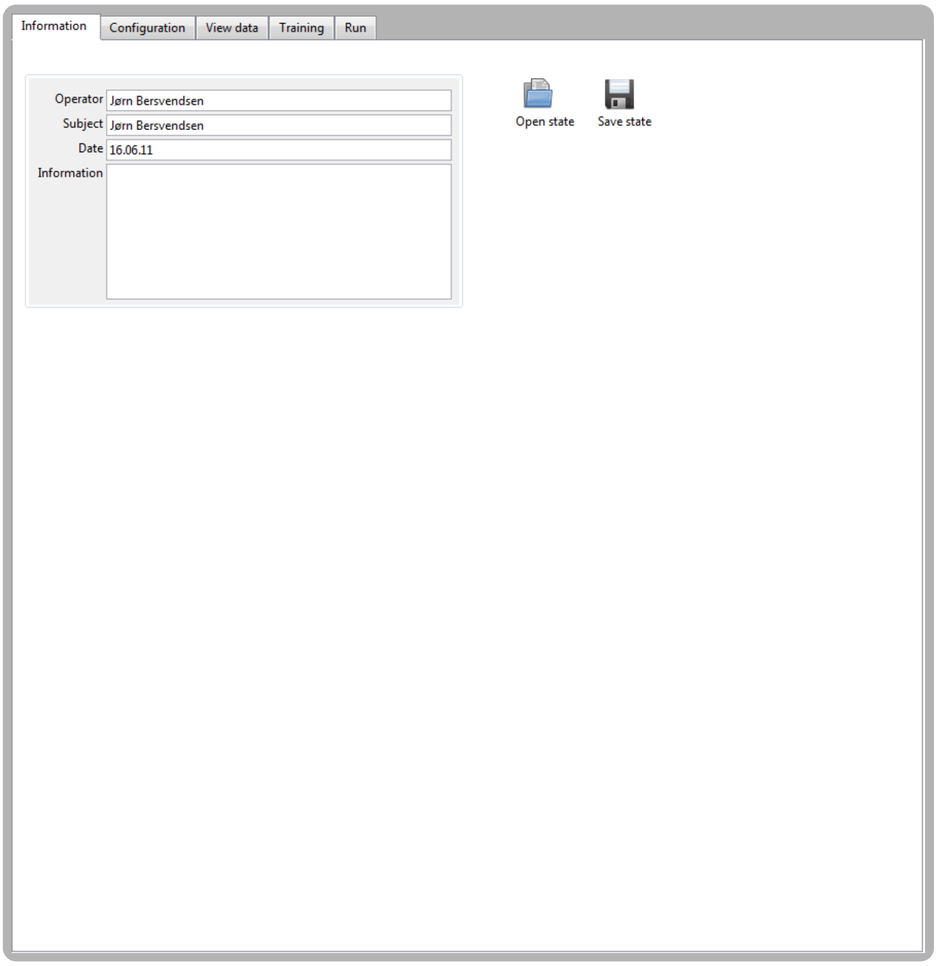
Every aspect of the functional specification presented in chapter 3 is covered, as well as some additional ones. For instance, the training data may be exported to a MATLAB `mat` file. This enables the application to be used exclusively for its training module, and all processing to be done off-line.

Another additional feature is the option of analyzing features after completing a training sequence. When this option is chosen, several classifiers are trained based on the selected features, and displayed to the user. Even though the results are only plotted in $\mathbb{R}^2$ (e.g. as in figure 6.12) when the true feature space is much larger, this still gives an indication on what features are likely to perform well on the current data set.

The classifier may be configured with two options. If the “strict transitions?” flag is checked, the only class transitions allowed are those entering or leaving the “no movement” class. Although it makes the controller slower to operate, this will in some cases reduce the number of errorous classifications.

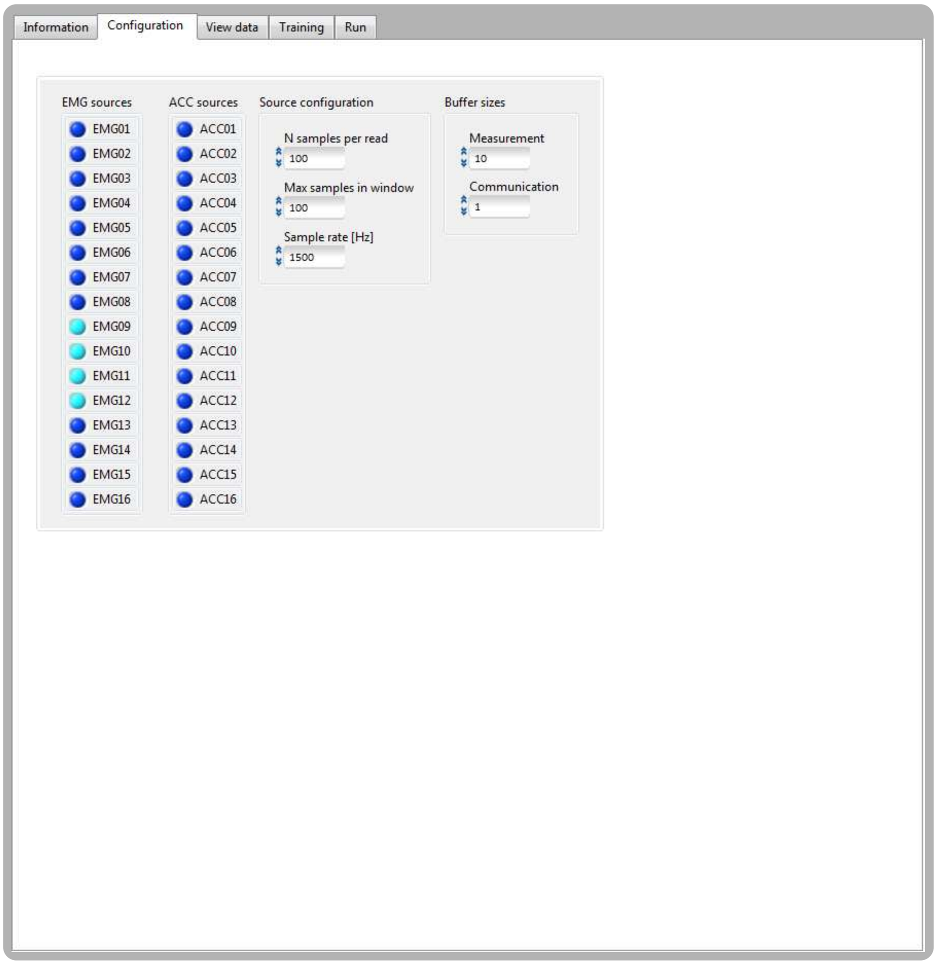
The other classifier option is the “classification hysteresis” number. If this number is set to N, the controller must receive $N+1$ consecutive classifications of class c in order to change the current class to c. This slows the controller down, but has proved to be essential for the model to move smoothly, and is very effective at removing unwanted, rapid class changes.

When running the demonstrator the user may select whether or not to communicate with the NXT. In addition, the speed of the NXT servos may be set by a slide bar. The value of this bar (in the range $[0,1]$) is multiplied to the configured servo speed set-points. If the speed is set too high, the user must concentrate too hard to achieve fine tuning control of the model. If set too low however, the model becomes unacceptable slow. This slide bar enables the user to change the speed at run-time which has proved to be a valuable option.



Operator	Jørn Bersvendsen
Subject	Jørn Bersvendsen
Date	16.06.11
Information	

Figure 6.4: Information tab. Allows the user to save or load from an application state file. The input fields are used to add information to a saved state and are all optional.



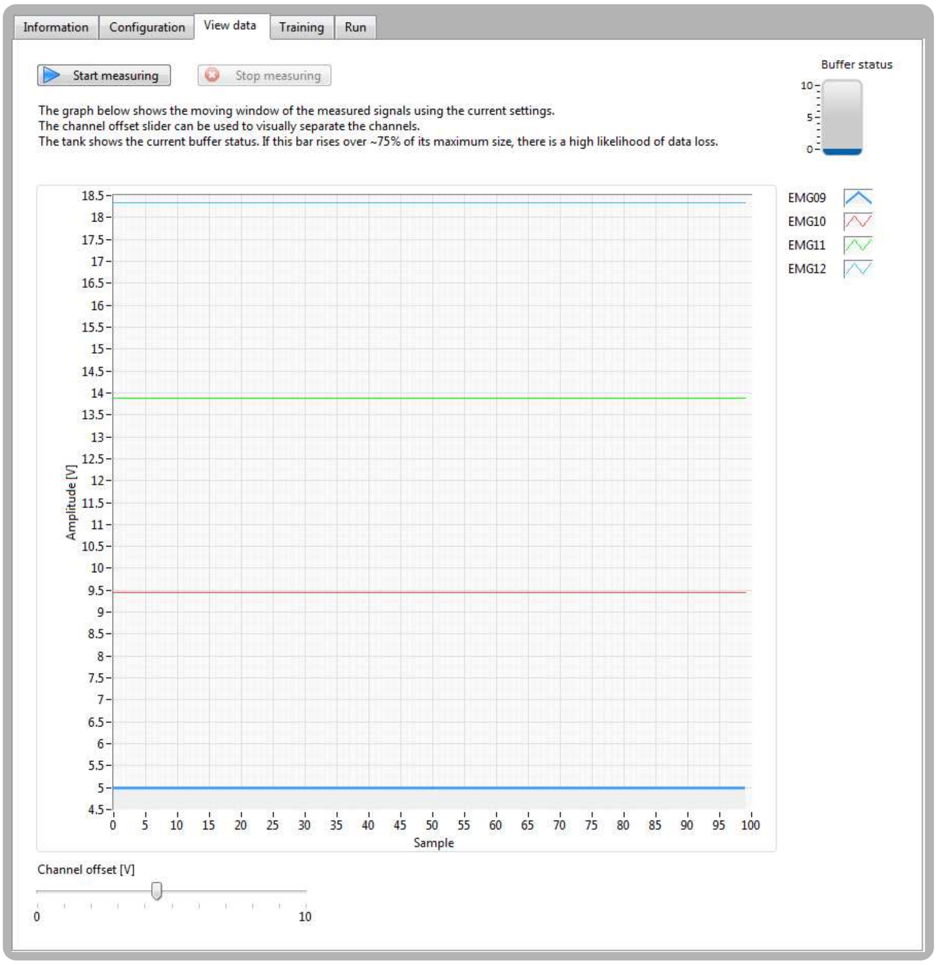
EMG sources: EMG01, EMG02, EMG03, EMG04, EMG05, EMG06, EMG07, EMG08, EMG09, EMG10, EMG11, EMG12, EMG13, EMG14, EMG15, EMG16

ACC sources: ACC01, ACC02, ACC03, ACC04, ACC05, ACC06, ACC07, ACC08, ACC09, ACC10, ACC11, ACC12, ACC13, ACC14, ACC15, ACC16

Source configuration
N samples per read: 100
Max samples in window: 100
Sample rate [Hz]: 1500

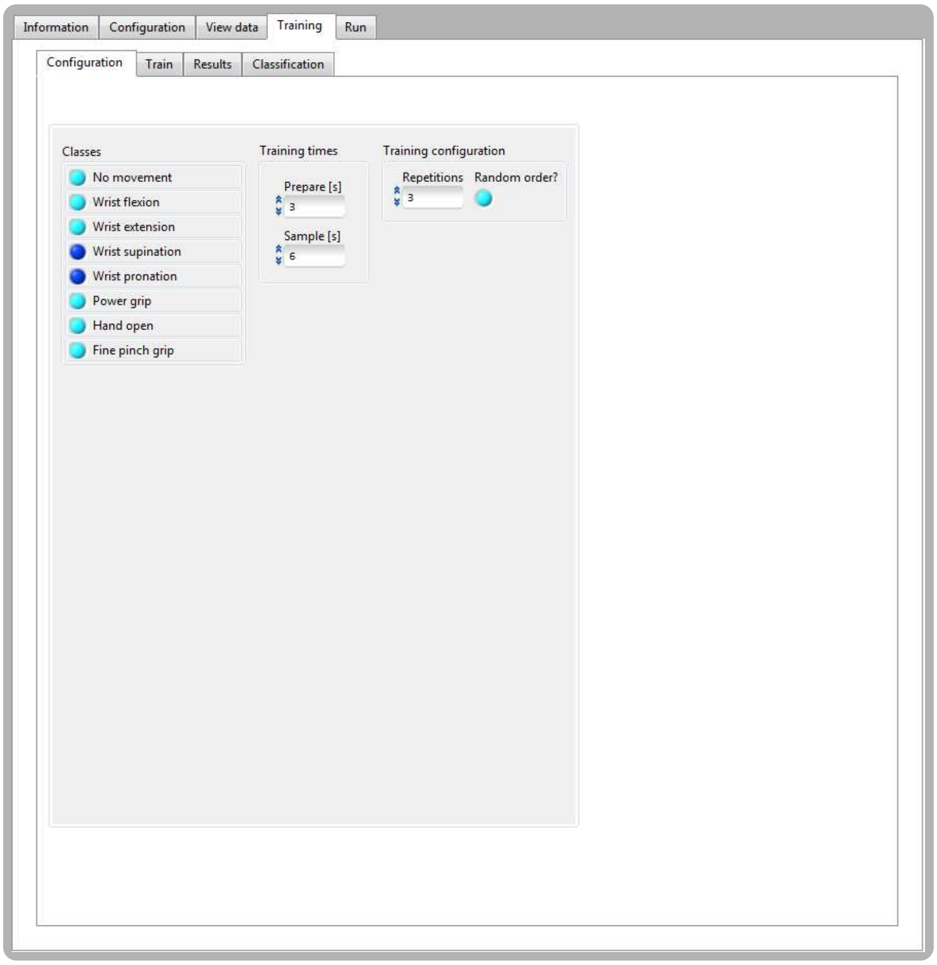
Buffer sizes
Measurement: 10
Communication: 1

Figure 6.5: Configuration tab. Allows the user to configure and select which sources to use.



The graph below shows the moving window of the measured signals using the current settings.
The channel offset slider can be used to visually separate the channels.
The tank shows the current buffer status. If this bar rises over ~75% of its maximum size, there is a high likelihood of data loss.

Figure 6.6: View data tab. Measures and displays data based on the current configuration.



Classes
- ● No movement
- ● Wrist flexion
- ● Wrist extension
- ● Wrist supination
- ● Wrist pronation
- ● Power grip
- ● Hand open
- ● Fine pinch grip

Training times
Prepare [s]: 3
Sample [s]: 6

Training configuration
Repetitions: 3
Random order? ●

Figure 6.7: Training configuration tab. Allows the user to configure a training sequence.

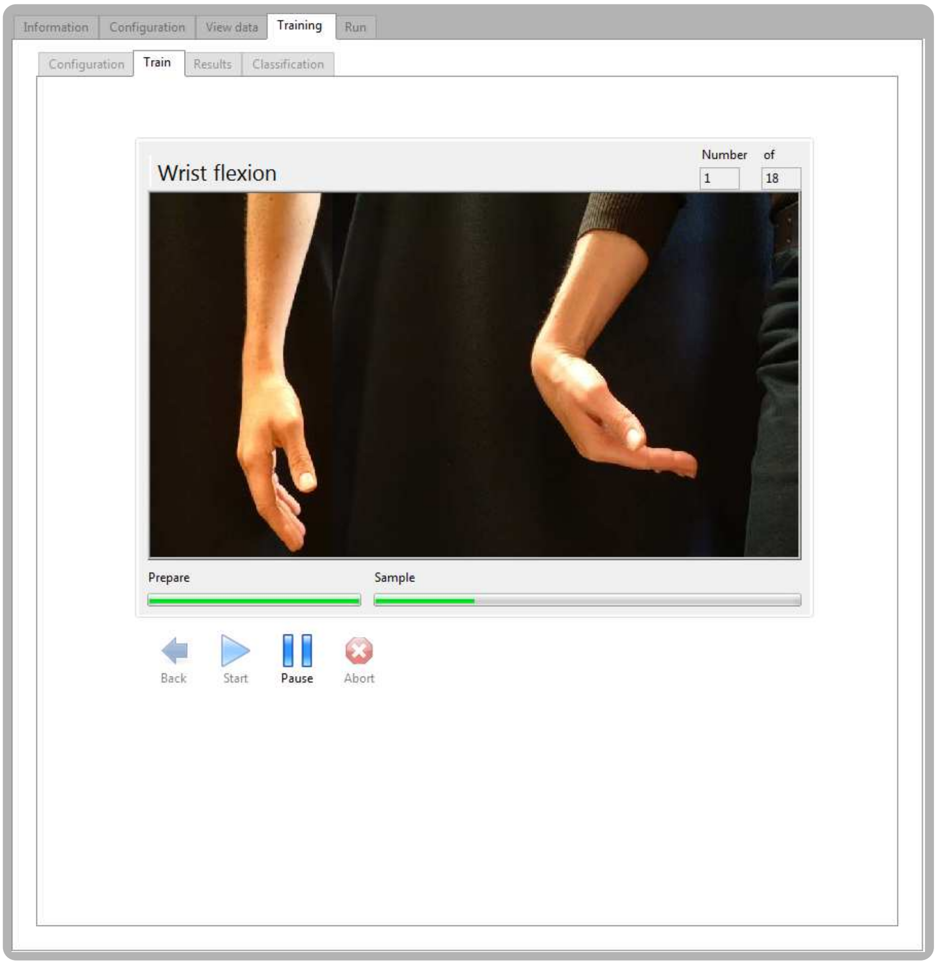


Figure 6.8: Training running tab. The training sequence can be paused, restarted and aborted.

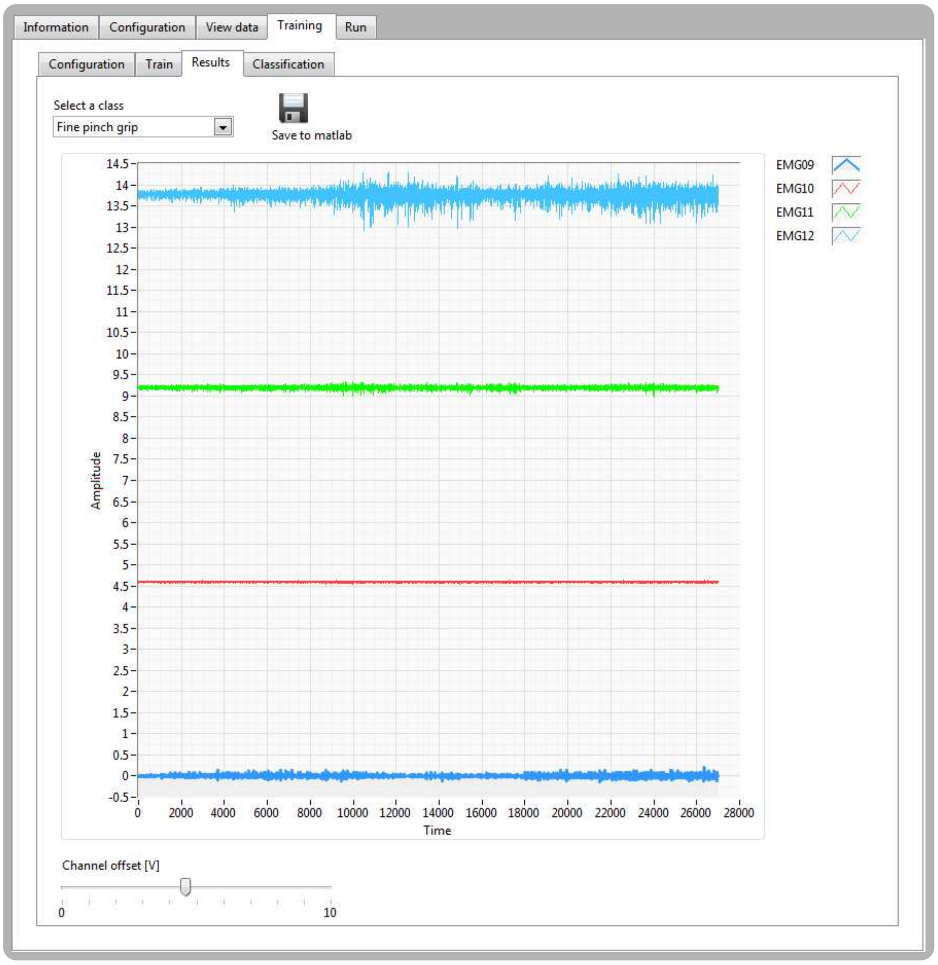
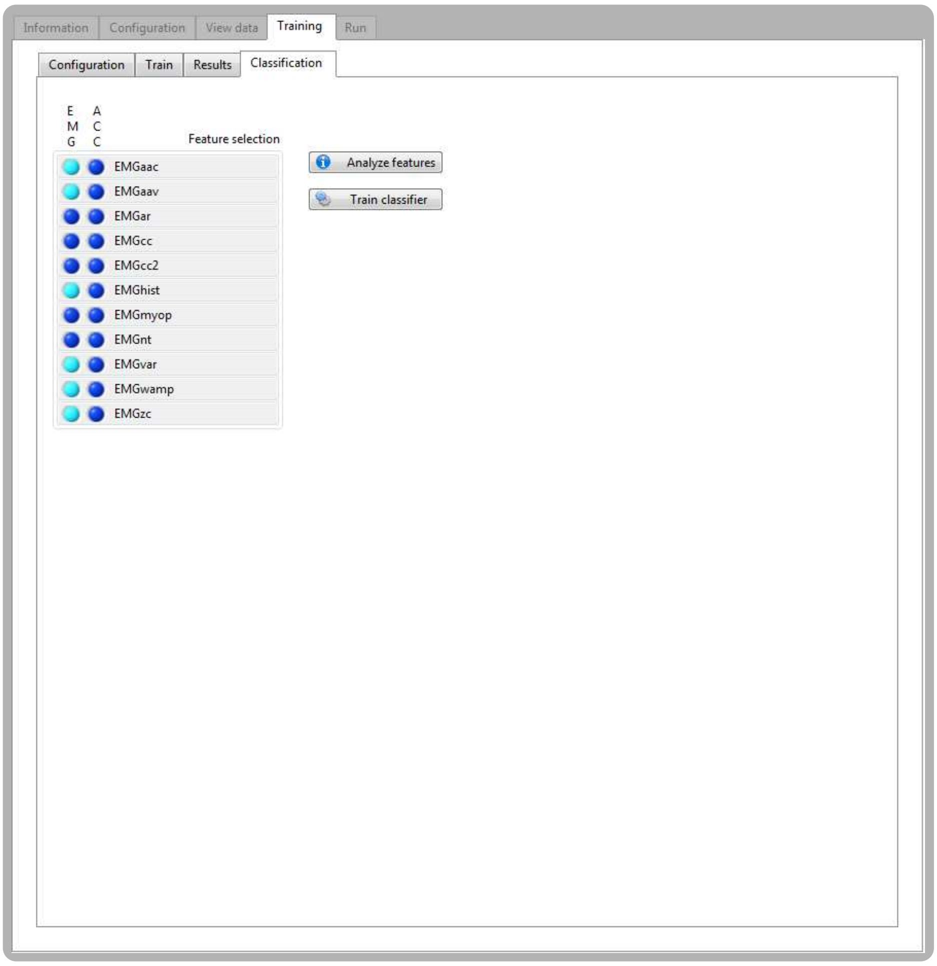


Figure 6.9: Training results tab. Upon completing a training sequence the user may inspect the signals visually. These may also be exported to MATLAB for offline processing.



EMG	ACC	Feature selection
●	●	EMGaac
●	●	EMGaav
●	●	EMGar
●	●	EMGcc
●	●	EMGcc2
●	●	EMGhist
●	●	EMGmyop
●	●	EMGnt
●	●	EMGvar
●	●	EMGwamp
●	●	EMGzc

Figure 6.10: Classifier training tab. The user may select the input of each feature. The “analyze features” option displays the classification as shown in figures 6.12 and 6.13.

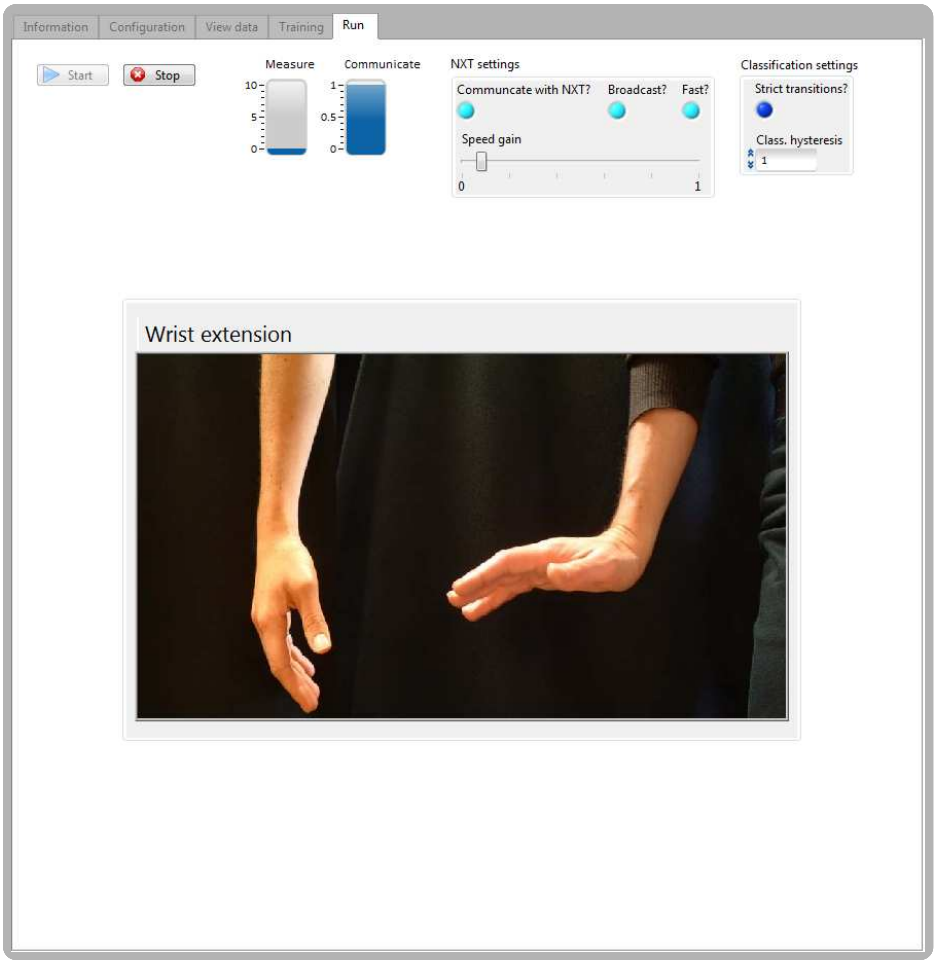


Figure 6.11: Demonstrator tab. The user may configure the classification and the host-NXT communication.

6.3 Controller

With four electrodes, two on the wrist flexors and two on the wrist extendors, one can control the following classes with almost no error in classification; no movement, wrist flexion, wrist extension, power grip, hand open, underarm pronation and supination. For this to work best the training set ought to be at least two repetitions of at least two seconds of sampling per class. Note that with only one test subject (i.e. the author) one cannot conclude that these results will reproduce for other users.

Using at least a single classification hysteresis (i.e. two similar, consecutive classifications must occur in order to change class) proved to be vital to avoid oscillations and rapid class change in between motion classes. Requiring more than two consecutive classifications turned out to infer such high delays (with a 10 Hz window update rate) as to make the system very hard to use.

6.4 Classifier performance

There are several aspects of the classifier that can be studied in the context of this thesis. The following two terms are used throughout section 6.4. The *separability* of a classifier is the percentage of feature vectors in the training data that are classified correctly by the classifier. However, this not the whole story, as the controller may make classification errors even with a training set separability of 100 %, simply because the EMG signals are non-deterministic. The *performance* of a classifier is the percentage of feature vectors from a test data set that are classified correctly.

The strategy for calculating these were to separate the training data into two parts. The first is used to train a classifier and calculate its separability, and the second is used to calculate the performance.

The separability and performance of LDA and LMS classifiers using different, two-dimensional feature spaces are discussed in section 6.4.1.

Note that when visualizing and analyzing the classifiers it is necessary to reduce the feature-space to comprehensible sizes. In practice, the feature space can easily be $\mathbb{R}^7 0$ using various features and eight electrodes.

6.4.1 LDA and LMS classifier performance

Figures 6.12 and 6.13 show the LMS and LDA classifier schemes and separability for different features on the same data set. Two electrodes were used, one on the wrist flexors and one on the wrist extendors. All the calculated features are scalar, so the feature space is always $\mathbb{R}^2$, making visualization easy.

By visual inspection, the LDA classifier seems to perform consistently better than the LMS classifier. However, the LDA training time is consistently higher than for the LMS classifier. Tests on different training data set sizes revealed that the LDA training takes close to twice the time compared to the LMS classifier. However, the actual calculation time for the MATLAB scripts are done in a matter of milliseconds on a standard desktop computer, so for the purpose of this thesis this does not matter[2].

Figures 6.14 and 6.15 show the classifier schemes and performance when only half the training data set of section 6.4.1 is used for classifier training. Note that most separability values are now reduced compared to figures 6.12 and 6.13.

6.4.2 LDA and LMS classifier separability with multiple features

So far only two scalar features were used at one time. This results in an easily plotted, two-dimensional feature space, but the performance is not always acceptable. This is specially true when the number of classes increase. To investigate what impact the number of features have on classifier separability and performance, the following experiment was carried out.

For a given data set, N different feature values were calculated. Then, for each combination of n feature values out of N possible, LMS and LDA classifiers were trained and their separabilities calculated. This was repeated for increasing n, and for each n the LSM and LDA classifiers with highest separability were kept and defined as *optimal*. The performance of these classifiers were then calculated.

The dataset was recorded using the same electrode configuration as in section 6.4.1, but two additional classes were introduced. The available features were AAV, ZC, NT, MYOP, WAMP and AR. Since the AR feature has four values (i.e. coefficients) and the others are scalar, 9 feature values were available for each EMG signal. Two electrodes were used, resulting in a maximum feature space of $\mathbb{R}^{18}$.

Figure 6.16(a) shows the results for $n = 2$, and figures 6.16(b) and 6.16(c) show the results for all n. The figure clearly demonstrates that the LDA performs significantly better than LMS for small feature spaces, but the difference decrease as the feature space grows. Another interesting observation is that the curves are not strictly increasing. In other words, there are some features that *are harmful* for the classification.

[2] The main application uses a few seconds on training, even though the actual MATLAB script executes in milliseconds. This may be due to overheads in transferring large amounts of data between LabVIEW and MATLAB, but as it poses no real problem this was not investigated further.

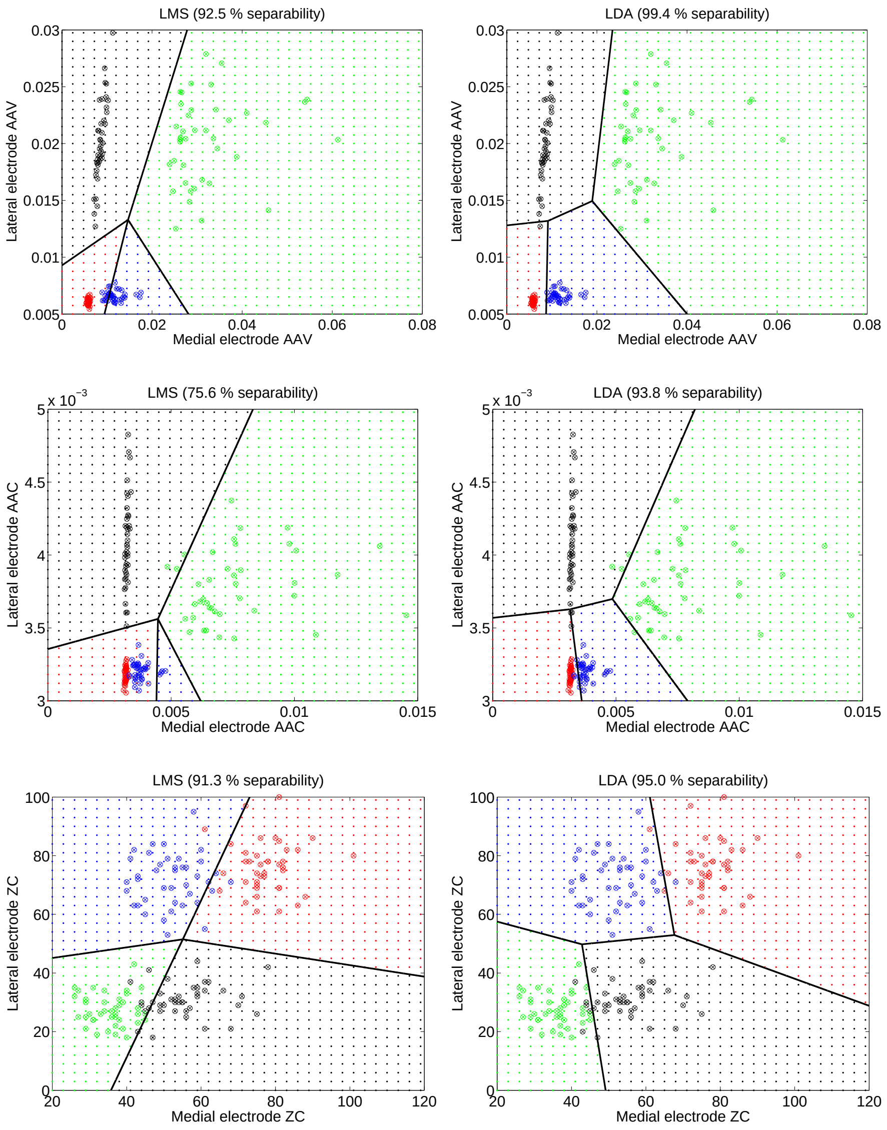


Figure 6.12: Comparison between LMS and LDA classifier performance in the $\mathbb{R}^2$ feature space. *Red:* no movement, *Blue:* wrist flexion, *Black:* wrist extension, *Green:* power grip.

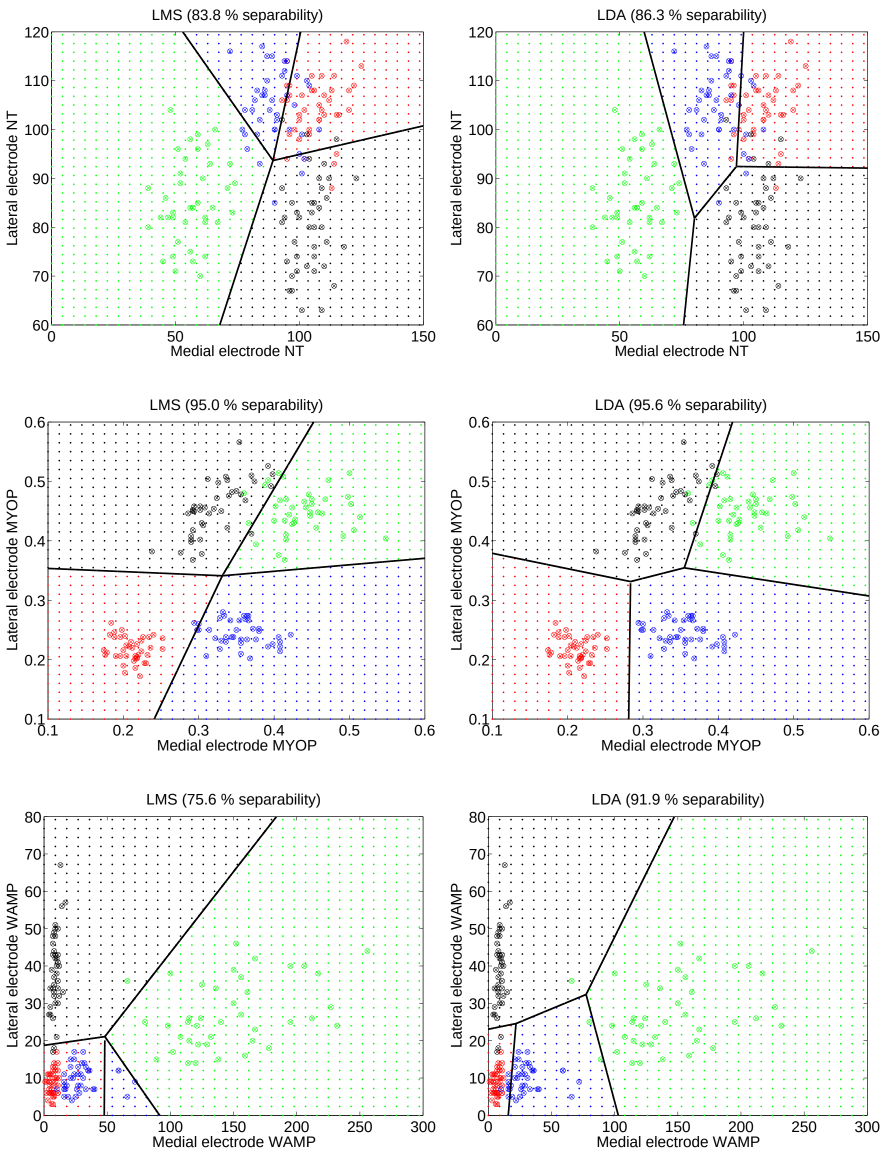


Figure 6.13: Comparison between LMS and LDA classifier performance in the $\mathbb{R}^2$ feature space. *Red:* no movement, *Blue:* wrist flexion, *Black:* wrist extension, *Green:* power grip.

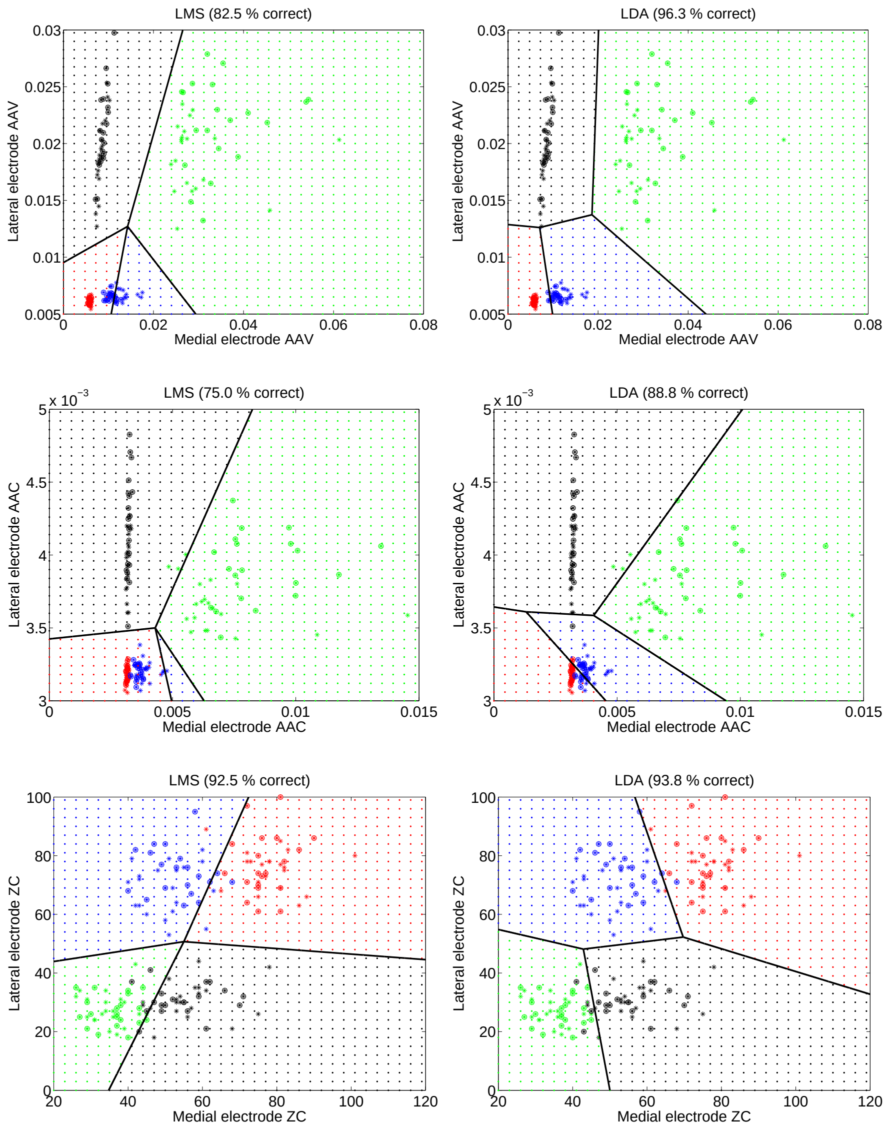


Figure 6.14: Comparison between LMS and LDA classifier performance in the $\mathbb{R}^2$ feature space. *Red:* no movement, *Blue:* wrist flexion, *Black:* wrist extension, *Green:* power grip.

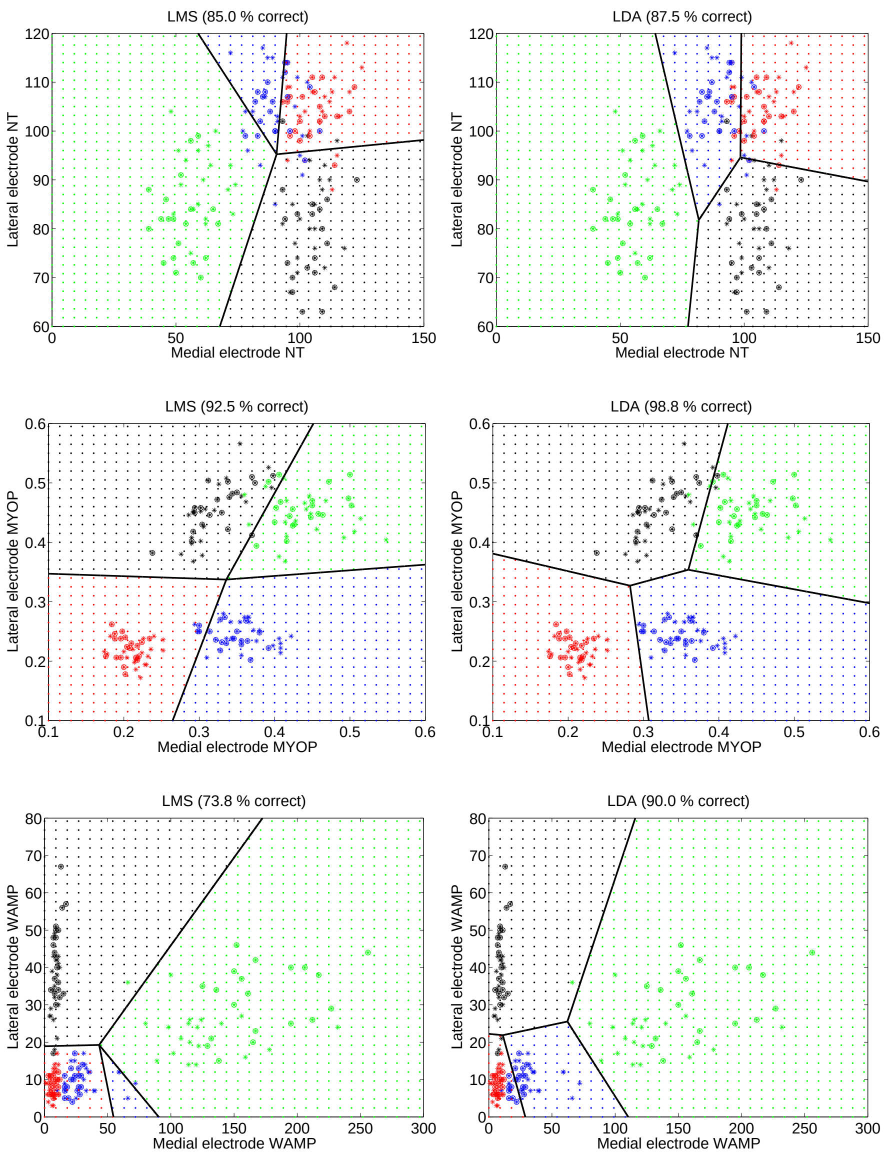
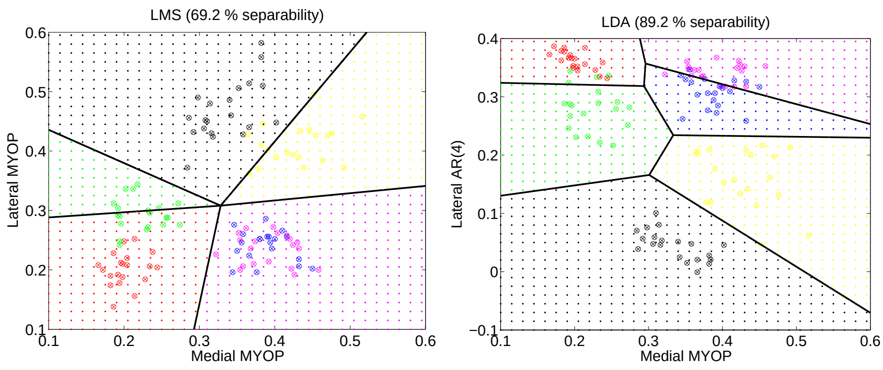
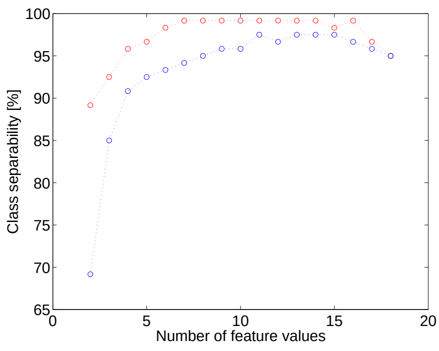


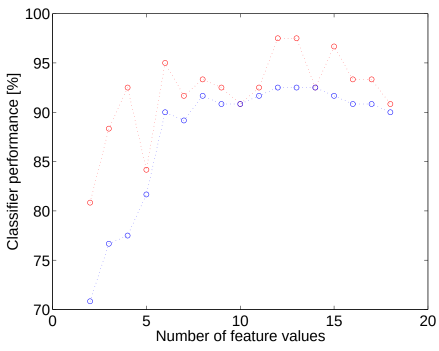
Figure 6.15: Comparison between LMS and LDA classifier performance in the $\mathbb{R}^2$ feature space. *Red:* no movement, *Blue:* wrist flexion, *Black:* wrist extension, *Green:* power grip.



(a) Separability with feature space $\mathbb{R}^2$. *Red:* no movement, *Blue:* wrist flexion, *Black:* wrist extension, *Green:* power grip, *Magenta:* underarm pronation, *Yellow:* underarm supination.



(b) Separability with increasing feature space. *Red:* LDA, *Blue:* LMS.



(c) Performance with increasing feature space. *Red:* LDA, *Blue:* LMS.

Figure 6.16: Comparison between optimal LMS and LDA classifiers separability and performance for different feature space sizes.

6.4.3 Feature/classification combinations

Because the LDA and LMS classifiers work in different ways, a feature set resulting in good separability for a LDA classifier does not necessarily do so for a LMS classifier.

To study this, the following experiment was done. For each combination of 4 feature values of the lateral and 4 of the medial electrode, a LDA and a LMS classifier was trained and its separability was calculated. Note that the feature space is exactly the same size for each classifier ($\mathbb{R}^8$). The same dataset as presented in section 6.4.2 was used.

Figure 6.17 shows the class separability for all classifiers. The values are sorted such that combination 1 has the highest separability, and the last combination has the lowest (this is done separately for the LDA and LMS classifiers). This figure clearly demonstrate the importance of using correct features, as the difference in separability from the best to the worst combination can easily be 25 %. Note that there are several combinations resulting in the same separability.

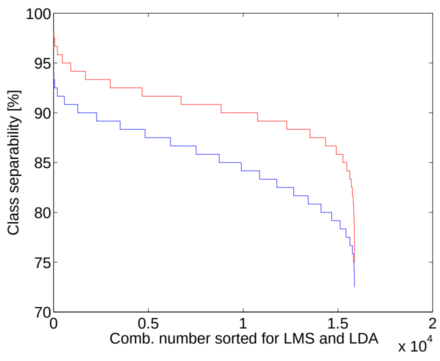


Figure 6.17: LMS and LDA performance with different feature combinations. The x axis shows the combination number, ranging from the best to the worst separability for each classifier. *Red:* LDA, *Blue:* LMS.

Figure 6.18 shows the separability of the 50 best LMS and LDA combinations. One can see that the LDA generally outperforms the LMS, but for some feature value combinations, the LMS has the highest separability. Feature selection is therefore dependent on the classifier in use.

6.4.4 Correlation between features

Having independent features is important in order to utilize the available processing power of the platform where calculation is done, as using heavily correlated features does not increase the classification quality. For this thesis work processing power was

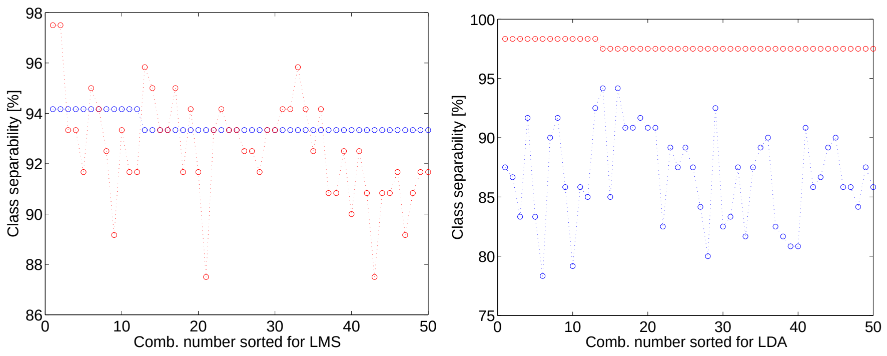


Figure 6.18: LMS and LDA separability with different feature combinations. The x axis shows the combination number, ranging from the best to the worst separability LMS (a) and LDA (b). *Red:* LDA, *Blue:* LMS.

superfluous as a desktop computer was used, but for usage on an actual prosthesis this is not the case.

The correlation between different feature values was calculated for the same dataset as used in section 6.4.1. Pearson's correlation coefficient given by

$$\rho_{X,Y} = \frac{\mathrm{cov}(X,Y)}{\sigma_X \sigma_Y} \tag{6.1}$$

was estimated by

$$\hat{\rho}_{X,Y} = \frac{\sum_{i=1}^{N}(X_i - \overline{X})(Y_i - \overline{Y})}{\sqrt{\sum_{i=1}^{N}(X_i - \overline{X})^2}\sqrt{\sum_{i=1}^{N}(Y_i - \overline{Y})^2}} \tag{6.2}$$

where $\overline{X}$ represents the average value of all X samples. The results are shown in tables 6.1 and 6.2.

AAV is generally highly correlated with AAC and VAR, whereas ZC, NT and MYOP are less correlated with other features. These results are in agreement with the conclusions suggested in chapter 2.4.2.

One can also see that whereas AAV and VAR are always highly correlated, AAV and AAC are highly correlated when the muscle is active (i.e. flexion and power grip for the medial side, extension and power grip for the lateral side). This seems to be the case between AAV and WAMP as well.

Table 6.1: Feature correlation, electrode on medial side of underarm.

(a) No movement

	AAV	AAC	MYOP	NT	VAR	WAMP	ZC
AAV	1	0.52	0.87	0.21	0.82	0.0077	0.44
AAC		1	0.45	0.35	0.43	0.062	0.38
MYOP			1	0.049	0.58	0.029	0.28
NT				1	0.37	0.25	0.062
VAR					1	0.1	0.25
WAMP						1	0.12
ZC							1

(b) Wrist flexion

	AAV	AAC	MYOP	NT	VAR	WAMP	ZC
AAV	1	0.91	0.59	0.38	0.97	0.89	0.46
AAC		1	0.36	0.42	0.93	0.96	0.22
MYOP			1	0.12	0.43	0.33	0.58
NT				1	0.43	0.41	0.015
VAR					1	0.92	0.34
WAMP						1	0.24
ZC							1

(c) Wrist extension

	AAV	AAC	MYOP	NT	VAR	WAMP	ZC
AAV	1	0.59	0.81	0.086	0.96	0.44	0.65
AAC		1	0.37	0.032	0.61	0.57	0.27
MYOP			1	0.076	0.65	0.26	0.65
NT				1	0.12	0.22	0.04
VAR					1	0.43	0.56
WAMP						1	0.3
ZC							1

(d) Power grip

	AAV	AAC	MYOP	NT	VAR	WAMP	ZC
AAV	1	0.96	0.01	0.69	0.96	0.86	0.0027
AAC		1	0.089	0.76	0.91	0.94	0.18
MYOP			1	0.052	0.092	0.085	0.45
NT				1	0.58	0.84	0.28
VAR					1	0.75	0.0034
WAMP						1	0.24
ZC							1

Table 6.2: Feature correlation, electrode on lateral side of underarm.

(a) No movement

	AAV	AAC	MYOP	NT	VAR	WAMP	ZC
AAV	1	0.19	0.85	0.016	0.81	0.24	0.47
AAC		1	0.17	0.33	0.24	0.17	0.26
MYOP			1	0.054	0.63	0.093	0.34
NT				1	0.019	0.33	0.00025
VAR					1	0.24	0.11
WAMP						1	0.097
ZC							1

(b) Wrist flexion

	AAV	AAC	MYOP	NT	VAR	WAMP	ZC
AAV	1	0.63	0.84	0.12	0.92	0.37	0.62
AAC		1	0.52	0.54	0.58	0.096	0.38
MYOP			1	0.21	0.65	0.36	0.55
NT				1	0.016	0.085	0.073
VAR					1	0.42	0.48
WAMP						1	0.29
ZC							1

(c) Wrist extension

	AAV	AAC	MYOP	NT	VAR	WAMP	ZC
AAV	1	0.88	0.59	0.59	0.98	0.85	0.5
AAC		1	0.37	0.71	0.86	0.94	0.32
MYOP			1	0.089	0.54	0.43	0.55
NT				1	0.58	0.75	0.15
VAR					1	0.82	0.45
WAMP						1	0.28
ZC							1

(d) Power grip

	AAV	AAC	MYOP	NT	VAR	WAMP	ZC
AAV	1	0.84	0.67	0.4	0.96	0.78	0.53
AAC		1	0.48	0.54	0.84	0.94	0.26
MYOP			1	0.16	0.63	0.47	0.5
NT				1	0.48	0.51	0.0039
VAR					1	0.78	0.39
WAMP						1	0.23
ZC							1

7 Conclusion

Through this thesis a working control system for a 7 degrees of freedom hand prosthesis model controlled by electromyographic and accelerometer signals has been developed. It has been an ambitious project with a very satisfying results, of which the author is truly proud.

7.1 Controller

The controller is training based, meaning that each user gets a unique controller. Every algorithm used can fit to any number of electrodes and degrees of freedom. This training is done in a minute or two, and configuration is done from a simple graphical user interface.

Using four electrodes and a training sequence of a couple of minutes, the author was able to control the hand using 7 motion classes. The level of precision is limited by the time delay from a movement is performed til the hand starts moving. This time delay was definitely noticeable, but the achieved precision was certainly good enough for demonstration purposes.

Some classes were harder to consistently classify correctly than others, such as pinch grip. However, when using eight electrodes instead of four, this may be easier to achieve.

The performance of the controller was thoroughly analyzed, using different classifiers and features. This gives insight into how different classifier and feature combinations perform for myoelectric prosthesis control.

7.2 Host software

In addition to implementing a controller and fulfilling the functional specification, almost every aspect of the host software is designed with future extension in mind. New motion classes may easily be added, as well as new classifiers and features.

The MATLAB back-end enables developers to change features and classifiers in a powerful, high-level language, and without knowledge of the LabVIEW code.

It is the author's own conclusion that the developed software is not merely a controller as assigned in the thesis problem description, it is a *software framework* designed to be easily extended in the future. Much effort has been put in to making the software modular, and to make as many aspects as possible configurable as opposed to hard-coded. Each of the $\tilde{1}50$ VIs in the developed software has a few descriptive lines of text in its documentation available in the LabVIEW editor, and some are more heavily documented. The structure of the LabVIEW project, file names, file hierarchy and the VI icon practices are very strict and well structured.

The LabVIEW platform has proved to be a great choice of software platform for the host and the application has a robust and bug-free feeling to it.

7.3 NXT software

The NXT was programmed in NXT-G using the LabVIEW NXT toolkit. Because the overheads in the native communication APIs (both USB, bluetooth and I^2C) were much higher than expected, the NXT has become the bottleneck of the system, limiting the servo set-points update rate. However, with some effort this update rate has been increased to acceptable levels, although the delay from a class is shown on the screen to the model starts moving is noticeable. On the other hand, the LabVIEW NXT toolkit makes it easy to create robust code, and the NXT software feels bug-free and is easy to use.

8 Suggested future work

8.1 Controller changes

The analysis of the classifier performance demonstrates the importance of *feature selection*. In an optimal case, the classifier would always use the features that give optimal performance for the given subject at the given time. For the analysis in this thesis, a simple brute-force approach was used, which takes too much time to be used in practice. However, many resources are available on feature selection, including Theodoridis & Koutroumbas (2008). Implementing such an algorithm would increase the demonstrator performance.

8.2 Software changes

8.2.1 Error handling

As mentioned earlier, the software does not perform any error-handling, which is an addition that would benefit the user. Error-wires are used throughout the source code to facilitate error-handling implementation in the future, but errors are most commonly displayed and cleaned. An overview of possible errors and corresponding actions should be developed, before changing the appropriate VIs to adhere to this specification.

8.2.2 Code “tweaks”

Through my experience developing with the designed software framework, some subtle possible improvements were found that were not implemented. This is strictly due to time constraints, as none of these aspects are hard to implement.

- Make the base classifier object store more configurations (i.e. source configuration). As it is now, the user may train a classifier then change which electrodes to use. If the user then tries to run the demonstrator, an error may occur because the new source configuration is applied to an unsupported classifier.

  In order to fix this, the classifier should store the source configuration used during training. This is done for the feature selection and training class selection, so it could easily be done the same way for source configuration.

- Combine broadcast and regular communication mode VIs. As it is now, the receiver must know in advance what communication mode the sender uses. This is not particularly troublesome, but at the same time not really elegant. A possible solution is to have each transmission contain a single byte (e.g. the first one), identifying the communication mode (e.g. 1 for broadcast and 0 for regular mode).

- When demonstration is running on the host, the measurement buffer status bar does not contain as much information as it could. A better indicator of the system performance would be the difference between the number of measured samples actually read and the number specified in the configuration. If this number is high, it means that classification and feature calculation takes too much time.

  This would be very easy to fix, simply change the input of the buffer graph.

- The broadcast send/receive method is only implemented as sender on the host and receiver on the NXT. Implementing this in the other direction would be useful for when the NXT needs to communicate back to the host.

- The speed type used now is a number between 0 and 1. However, to specify the direction an additional flag is needed when sending. This could be simplified by using the same -1 to 1 uword as for position, where the sign determines the direction. The only thing to note is that the precision available when using an uword is much larger than the NXTServo-v2 allows.

8.3 Changes in the system layout

Through my experience with the system, having two nodes (host and NXT) feels rather superfluous. As it is now the host performs all the work with the exception of the I^2C communication with the servo controller. For a future project I would suggest removing one of these nodes.

Removing the NXT is the most natural approach, as a computer is needed anyway (i.e. for the data acquisition and training). This could be done by replacing the NXT with a simple circuit board with a serial interface, which is supported by LabVIEW. This board would not need any logic, it could be a byte-to-byte transfer from UART to I^{2}C. Another option would be to replace the NXTServo-v2 controller with another servo controller supporting a USB or USART interface.

One could also move in the other direction, i.e. from the computer towards the NXT. Note that a computer would still be needed to acquire the signals, but implementing the control scheme on the NXT will take the system in the direction of an actual prosthesis implementation. For this purpose the LabVIEW NXT toolkit will certainly be inadequate, but a low-level C implementation could perform quite well given the high clock speed of the ARM7.

Bibliography

Bach, P. F. (2009), Myoelectric signal features for upper limb prostheses, Master's thesis, Norwegian University of Science and Technology.

Bersvendsen, J. (2010), Prosthesis simulator based on pidstop. Course work at the Norwegian University of Science and Technology.

Boostani, R. & Moradi, M. H. (2003), 'Evaluation of the forearm emg signal features for the control of a prosthetic hand', *Physiological Measurement* **24**(2), 309.

Chan, F., Yang, Y.-S., Lam, F., Zhang, Y.-T. & Parker, P. (2000), 'Fuzzy emg classification for prosthesis control', *Rehabilitation Engineering, IEEE Transactions on* **8**(3), 305 –311.

Elliott, C., Vijayakumar, V., Zink, W. & Hansen, R. (2007), 'National instruments labview: A programming environment for laboratory automation and measurement', *Journal of the Association for Laboratory Automation* **12**(1), 17 – 24.

Englehart, K., Hudgin, B. & Parker, P. (2001), 'A wavelet-based continuous classification scheme for multifunction myoelectric control', *Biomedical Engineering, IEEE Transactions on* **48**(3), 302 – 311.

Englehart, K., Hudgins, B., Parker, P. A. & Stevenson, M. (1999), 'Classification of the myoelectric signal using time-frequency based representations', *Medical Engineering and Physics* **21**(6-7), 431 – 438.

Fougner, A. L. (2007), Proportional myoelectric control of a multifunctional upper-limb prosthesis, Master's thesis, Norwegian University of Science and Technology.

Håkonsen, K. S. (2010), Utvikling og evaluering av en flerfunksjonell armprotesemodell, Master's thesis, Norwegian University of Science and Technology.

Hosek, P., Prykäri, T., Alarousu, E. & Myllylä, R. (2009), 'Application of labview: Complex software controlling of system for optical coherence tomography', *Journal of the Association for Laboratory Automation* **14**(2), 59 – 68.

Huang, H.-P., Liu, Y.-H. & Wong, C.-S. (2003), Automatic emg feature evaluation for controlling a prosthetic hand using supervised feature mining method: an intelligent approach, *in* 'Robotics and Automation, 2003. Proceedings. ICRA '03. IEEE International Conference on', Vol. 1, pp. 220 – 225 vol.1.

Kang, W. J., Cheng, C. K., Lai, J. S., Shiu, J. R. & Kuo, T. S. (1996), 'A comparative analysis of various emg pattern recognition methods', *Medical Engineering and Physics* **18**(5), 390 – 395.

Kehtarnavaz, N. (2007), Labview graphical programming environment, *in* 'Digital Signal Processing System Design', 2 edn, Academic Press, Burlington, pp. 5 – 56.

Mallat, S. (2009), *A wavelet tour of signal processing: The sparse way*, 3 edn, Academic Press.

Muzumdar, A. (2004), *Powered upper limb prostheses: control, implementation and clinical application*, Springer-Verlag.

Oskoei, M. A. & Hu, H. (2007), 'Myoelectric control systems–a survey', *Biomedical Signal Processing and Control* **2**(4), 275 – 294.

Parker, P., Englehart, K. & Hudgins, B. (2006), 'Myoelectric signal processing for control of powered limb prostheses', *Journal of Electromyography and Kinesiology* **16**(6), 541 – 548. Special Section (pp. 541-610): 2006 ISEK Congress.

Pattichis, C. S. & Elia, A. G. (1999), 'Autoregressive and cepstral analyses of motor unit action potentials', *Medical Engineering and Physics* **21**(6-7), 405 – 419.

Theodoridis, S. & Koutroumbas, K. (2008), *Pattern Recognition*, 4 edn, Elsevier.

Vignolo, L. D., Rufiner, H. L., Milone, D. H. & Goddard, J. C. (2011), 'Evolutionary cepstral coefficients', *Applied Soft Computing* **11**(4), 3419 – 3428.

Willison, R. G. (1963), A method of measuring motor unit activity in human muscle, *in* 'Proceedings of the Physiological Society', pp. 35–36.

Xavier, W. H. & Rodet, X. (2003), Discrete cepstrum coefficients as perceptual features, *in* 'Proceedings of the International Computer Music Conference'.

Zardoshti-Kermani, M., Wheeler, B., Badie, K. & Hashemi, R. (1995), 'Emg feature evaluation for movement control of upper extremity prostheses', *Rehabilitation Engineering, IEEE Transactions on* **3**(4), 324 –333.

List of figures

A
Developer's guide: Hardware

This short guide will serve as a quick reference to the hardware aspects of the system.

Servo warning
The thumb abduction/adduction servo has been disconnected on the model, because connecting it results in a short-circuit.

A.1 Power supply

Power warning
When external power is used for the servo motors, this power source *must be disconnected* whenever the NXT is turned off or disconnected from the servo controller. If this is not done there will be noise on the PWM lines resulting in chaotic movement of the hand.

The NXT may be used as a power supply for the servos. However, since the batteries are drained so quickly, using an external power source is recommended.

The power source should be at 8.0 V and support a maximum current of about 2.0 A. The normal power drain is much less than this, but when running several motors at once, the current peaks about this value.

A.2 Trigno/host connections

This section describes all connections of the analog signals as used by the developed software “out of the box”. Note that the signal routing may be changed in software as described in the software developer’s guide.

Tables A.1 and A.2 show the connection schemes of the analog signals from the Trigno system to the DAQPad-6016 and PCI-6025E respectively. All pin numbers and names refer to the manufacturer’s datasheets.

The wire colors refer to those on the cable provided by Delsys with the Trigno system. Note that the wires in this cable are twisted in groups of two. The twisted cables always have alternating colors (e.g. the gray/tan wire is twisted with the tan/gray wire).

Note that the accelerometer signals exhibit a fixed 48 ms delay from the time a sensor detects an event to the time the analog signal is reproduced.

A.3 NXTServo-v2 controller

The NXTServo-v2 controller operates on PWM width in μs for position set-points and PWM width change in μs per 24 μs for speed set-points. If a position set-point P is written and the associated speed register S is greater than zero, the controller increases or decreases the PWM width by S each 24 μs until the position P is reached. If $S = 0$ then P is written at once and the servo moves as fast as possible.

The servos accepts pulse widths between 500 μs and 2500 μs, although the construction of the hand limits the movement of the servos much more than this. The position set-points are given in a 16-bit unsigned word, and the speed set-points are given in a single byte.

When the servo controller receives power it initializes by writing 1500 μs to all servos. This is a standard neutral position, but as this is not a natural position of the hand, a separate initialization is done when the NXT application starts.

Table A.1: Trigno to DAQPad-6016 connections. By DAQPad input name and DAQPad input pin.

DAQPad inputs		Trigno outputs		Wire colors	
Name	Pin	Name	Pin	Main	Stripe
AI 0	1	EMG 9	51	Gray	Tan
AI 1	4	EMG 10	49	Blue	Tan
AI 2	7	EMG 11	47	Yellow	Tan
AI 3	10	EMG 12	45	Pink	Tan
AI 4	17	AX 9	50	Violet	Tan
AI 5	20	AX 10	48	Green	Tan
AI 6	23	AX 11	46	Orange	Tan
AI 7	26	AX 12	44	Brown	Tan
AI 8	2	AZ 9	17	Tan	Gray
AI 9	5	AZ 10	15	Tan	Blue
AI 10	8	AZ 11	13	Tan	Yellow
AI 11	11	AZ 12	11	Tan	Pink
AI 12	18	AY 9	16	Tan	Violet
AI 13	21	AY 10	14	Tan	Green
AI 14	24	AY 11	12	Tan	Orange
AI 15	27	AY 12	10	Tan	Brown
AI SENSE	13	GND	22	Brown	Blue

DAQpad inputs		Trigno outputs		Wire colors	
Pin	Name	Pin	Name	Main	Stripe
1	AI 0	51	EMG 9	Gray	Tan
2	AI 8	17	AZ 9	Tan	Gray
4	AI 1	49	EMG 10	Blue	Tan
5	AI 9	15	AZ 10	Tan	Blue
7	AI 2	47	EMG 11	Yellow	Tan
8	AI 10	13	AZ 11	Tan	Yellow
10	AI 3	45	EMG 12	Pink	Tan
11	AI 11	11	AZ 12	Tan	Pink
13	AI SENSE	22	GND	Brown	Blue
17	AI 4	50	AX 9	Violet	Tan
18	AI 12	16	AY 9	Tan	Violet
20	AI 5	48	AX 10	Green	Tan
21	AI 13	14	AY 10	Tan	Green
23	AI 6	46	AX 11	Orange	Tan
24	AI 14	12	AY 11	Tan	Orange
26	AI 7	44	AX 12	Brown	Tan
27	AI 15	10	AY 12	Tan	Brown

Table A.2: Trigno to PCI-6025E connections. By PCI input name and PCI input pin.

PCI inputs		Trigno outputs		Wire colors	
Name	Pin	Name	Pin	Main	Stripe
ACH0	3	EMG 13	42	Violet	White
ACH1	5	AZ 13	8	White	Violet
ACH2	7	EMG 14	40	Green	White
ACH3	9	AZ 14	6	White	Green
ACH4	11	EMG 15	38	Orange	White
ACH5	13	AZ 15	4	White	Orange
ACH6	15	EMG 16	36	Brown	White
ACH7	17	AZ 16	2	White	Brown
ACH8	4	AX 13	41	Blue	White
ACH9	6	AY 13	7	White	Blue
ACH10	8	AX 14	39	Yellow	White
ACH11	10	AY 14	5	White	Yellow
ACH12	12	AX 15	37	Pink	White
ACH13	14	AY 15	3	White	Pink
ACH14	16	AX 16	35	Tan	White
ACH15	18	AY 16	1	White	Tan
AISENSE	19	GND	43	Gray	White

PCI inputs		Trigno outputs		Wire colors	
Pin	Name	Pin	Name	Main	Stripe
3	ACH0	42	EMG 13	Violet	White
4	ACH8	41	AX 13	Blue	White
5	ACH1	8	AZ 13	White	Violet
6	ACH9	7	AY 13	White	Blue
7	ACH2	40	EMG 14	Green	White
8	ACH10	39	AX 14	Yellow	White
9	ACH3	6	AZ 14	White	Green
10	ACH11	5	AY 14	White	Yellow
11	ACH4	38	EMG 15	Orange	White
12	ACH12	37	AX 15	Pink	White
13	ACH5	4	AZ 15	White	Orange
14	ACH13	3	AY 15	White	Pink
15	ACH6	36	EMG 16	Brown	White
16	ACH14	35	AX 16	Tan	White
17	ACH7	2	AZ 16	White	Brown
18	ACH15	1	AY 16	White	Tan
19	AISENSE	43	GND	Gray	White

B
Developer's guide: Software

This is short guide will serve as a reference for developers changing software configurations or extending the software. This is a practical guide, for more information please refer to the main thesis.

B.1 Installation

For the development a 32-bit version of Windows 7 was used. The following software needs to be installed.

LabVIEW The version used during development was LabVIEW 2010 Professional Development System.

LabVIEW NXT toolkit This is freely available from the NI Developer Zone[1] if not included with the LabVIEW installation.

VI package manager This is an application that manages additional packages for LabVIEW. Specifically it is needed to easily download and install certain Open-G packages. Available freely (the community edition) online[2].

[1]http://zone.ni.com/devzone/cda/tut/p/id/4435
[2]http://jki.net/vipm

LabVIEW packages Using the VI package manager, install the following packages.

- `UI Control Suite: System Controls 2.0`, contains controls used by the host application.
- `nirsc_html_help_common`
- `oglib_appcontrol`
- `oglib_array`
- `oglib_error`
- `oglib_file`
- `oglib_lvdata`
- `oglib_lvzip`
- `oglib_md5`
- `oglib_string`
- `oglib_dynamicpalette`

All packages, except the first one, are dependencies of the `matio` package written in open-g, that will allow LabVIEW to export MATLAB files.

LabVIEW `matio` package Download the `matio` package freely available online[3]. The downloaded file can be opened in the VI package manager and installed from there. The version used during development was 0.1-8.

MATLAB The version used during development was MATLAB R2010b (7.11.0).

B.2 LabVIEW project structure and practices

All LabVIEW code for both the host and NXT software is contained in a single Lab-VIEW project. All source files are located in one of the following three folders.

Computer Contains code that is used exclusively on the host.

NXT Contains code that is used exclusively on the NXT.

Shared Contains code that is used on both the host and the NXT.

Within the NXT folder each VI is categorized further. VIs that are written exclusively for the NXT that does not make sense to run anywhere else (e.g. code that handles the NXT display or buttons) get the file extension `.nxt.vi` as opposed to the regular `.vi`.

[3]http://sourceforge.net/projects/matio-labview/

B.2.1 VI icon practices

All VIs have an icon that follows a strict rule. Each icon consists of a white background with a black frame, and up to four lines of text. For the normal VIs the first line is a color-coded identifier of the VI type (e.g. `comm.` for communication, `train.` for training), and the rest of the lines make a descriptive text.

The exceptions to this rule are the main executable VIs that have their names written in the center of the icon, and the NXT-specific VIs that have a slightly different background frame. Examples of all these icon types are shown in figure B.1.

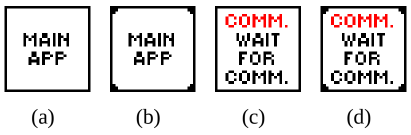


Figure B.1: Examples of VI icons. Main executable VIs for host (a) and NXT (b). Normal VIs for host (c) and NXT (d).

B.3 Configuration

Configuration files warning
All configuration files must end with a line-break in order for LabVIEW to parse them correctly. Windows (CRLF), UNIX (LF) and Mac (CR) line breaks are supported.

B.3.1 Application configuration

The functional global variable `var_config_app.vi`[4] contains the application configuration. This configuration can be changed by writing a new default value to the `Application Configuration variable` control. Below is a description of each field.

Config file pinout Absolute path to the analog channel pinout file. Value used during development is `./Computer/Config/sources.txt`.

Config file training classes Absolute path to the training classes file. Value used during development is `./Computer/Config/classes.txt`.

[4]Located in `./Computer/VI/Configuration/`

Config MATLAB script path This path will be added to the MATLAB work path. Note that sub-folders are *not* added. Value used during development is `./Computer/Matlab`.

Config classifier class path Absolute path to the classifier `.lvclass` class file. Value used during development is `./Computer/Classes/Classifier LDA/ Classifier LDA.lvclass`.

Config features path Absolute path of the folder containing the feature `m`-files. This path will be added to the MATLAB work path. Note that sub-folders are *not* added. Each file is assumed to be a proper feature function. Value used during development is `./Computer/Matlab/Features`.

Config file servo set points Absolute path to the set-points file. Value used during development is `./Computer/Config/servo_setpoints.txt`.

B.3.2 Analog channel pinout file

All channel names are enumerated in `source_id.ctl`. The EMG channels are labeled `EMG01`, `EMG02`, ..., `EMG16`. The accelerometer channels are labeled `ACC01X`, `ACC01Y`, `ACC01Z`, ..., `ACC16X`, `ACC16Y`, `ACC16Z`. The numbers refer to the numbers on the Trigno electrode units.

The channel routing is located in `./Computer/Config/sources.txt`. Each row in this file assigns an enumerated channel name to a LabVIEW task channel in the following format; `<channel id>=<task channel>`.

The following file defines the connection routing as used during development.

```
EMG09=Dev4/ai0
ACC09X=Dev4/ai4
ACC09Y=Dev4/ai12
ACC09Z=Dev4/ai8
EMG10=Dev4/ai1
ACC10X=Dev4/ai5
ACC10Y=Dev4/ai13
ACC10Z=Dev4/ai9
EMG11=Dev4/ai2
ACC11X=Dev4/ai6
ACC11Y=Dev4/ai14
ACC11Z=Dev4/ai10
EMG12=Dev4/ai3
ACC12X=Dev4/ai7
ACC12Y=Dev4/ai15
ACC12Z=Dev4/ai11
EMG13=Dev5/ai0
ACC13X=Dev5/ai8
ACC13Y=Dev5/ai9
ACC13Z=Dev5/ai1
EMG14=Dev5/ai2
```

```
ACC14X=Dev5/ai10
ACC14Y=Dev5/ai11
ACC14Z=Dev5/ai3
EMG15=Dev5/ai4
ACC15X=Dev5/ai12
ACC15Y=Dev5/ai13
ACC15Z=Dev5/ai5
EMG16=Dev5/ai6
ACC16X=Dev5/ai14
ACC16Y=Dev5/ai15
ACC16Z=Dev5/ai7
```

Note that `Dev4` refers to the DAQpad, and `Dev5` to the PCI card. This is specific to the computer on which the application runs, and can be changed with the "Measurement & Automation Explorer" application that comes bundled with LabVIEW. These are located under `My System/Devices and Interfaces` and can be changed by renaming the device in question.

B.3.3 Training classes file

All classes have a unique string identifier, a label and a illustration file name. The available classes are located in `./Computer/Config/classes.txt`. Each row in this file adds a new class available for training in the following format; `<id> <label> <image>`. Note that the white-space characters are tabular characters. The image field is the absolute path to a `jpg` file.

The following file defines the classes used during development.

```
NO_MOVEMENT         No movement         .\Computer\Images\nomov.jpg
WRIST_FLEXION       Wrist flexion       .\Computer\Images\wflex.jpg
WRIST_EXTENSION     Wrist extension     .\Computer\Images\wext.jpg
WRIST_SUPINATION    Wrist supination    .\Computer\Images\wsup.jpg
WRIST_PRONATION     Wrist pronation     .\Computer\Images\wpron.jpg
POWER_GRIP          Power grip          .\Computer\Images\pgrip.jpg
HAND_OPEN           Hand open           .\Computer\Images\open.jpg
FINE_PINCH_GRIP     Fine pinch grip     .\Computer\Images\pinch.jpg
```

Training classes warning
The `NO_MOVEMENT` class is special in that it is used explicitly when the "strict transitions" controller mode is enabled. This class is therefore assumed to exist *and be the first in the list.*

B.3.4 Servo set-points file

Each class can have a set of servo set-points associated with it. These set-points are located in `./Computer/Config/servo_setpoints.txt`. Each row in this file associates a class with a set of servo set-points in the following format; `<id> <setpoints>`. Note that the white-space character is a tabular character. The `setpoints` field consists of 7 floating point numbers separated by white-space. The i'th number in this list gives the set-point of the i'th servo, enumerated in `motor_id.ctl`. The set-points are given in the "unit" position type, as discussed in section 5.5.1. The class identifiers must match those defined in the class-file discussed above.

The following file defines the set-points used during development.

```
NO_MOVEMENT         0    0   0   0   0   0   0
WRIST_FLEXION       0    0   1   0   0   0   0
WRIST_EXTENSION     0    0  -1   0   0   0   0
WRIST_SUPINATION   -1    0   0   0   0   0   0
WRIST_PRONATION     1    0   0   0   0   0   0
POWER_GRIP          0    0   0   1   1   0   1
HAND_OPEN           0    0   0  -1  -1   0  -1
FINE_PINCH_GRIP     0    0   0   0   1   0   1
```

Note that if there are no servo set-points defined for a class a zero-vector is used.

B.4 Features

All features are MATLAB `m`-files in the configured feature folder. Each file is assumed to contain a function accepting two arguments and returning a *row*-vector. The following code snippet demonstrates how the AAV feature is implemented and represents the method signature that is common for all feature files.

```
% EMGaav    Calculate average absolute value
%   EMGaav(DATA, CONFIG) calculates the AAV of DATA
%       DATA:       A nSamples x nSources matrix where each column is a
%                   time series from a separate source (i.e. channel).
%       CONFIG:     A struct containing the fields
%                       rate:       Sample rate in Hz.
%                       sources:    A list of source identifiers. The ith
%                                   element corresponds to the ith column
%                                   in DATA.
function AAV = EMGaav(data, config)
    AAV = mean(abs(data));
end
```

B.5 Required MATLAB back-end

The classifier and feature MATLAB scripts rely on some MATLAB functions that need to be in the search path when the system runs. The following MATLAB scripts must be placed in the path configured int the application configuration discussed in section B.3.1.

`doClassify.m` Used by both the LDA and the LMS `classify` functions.

`getAccData.m` Used by the feature extraction LabVIEW function.

`getEmgData.m` Used by the feature extraction LabVIEW function.

`trainLDA.m` Used by the LDA classifier.

`trainLMS.m` Used by the LMS classifier.

In addition, there are some methods that are not needed by the software as is, but might be useful for future classifier and feature implementation.

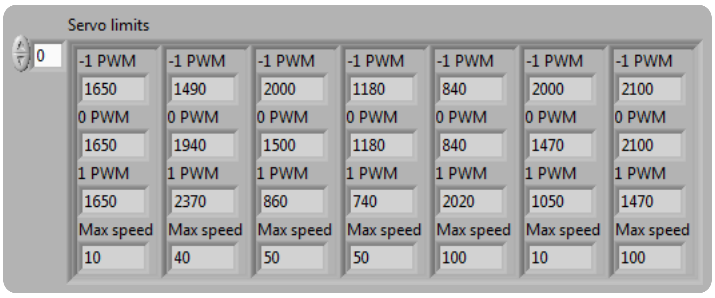
`id2label.m` Creates a source label (e.g. `EMG01`) from a source id.

`isEmg.m` Returns whether the supplied source id is an EMG signal.

`isAcc.m` Returns whether the supplied source id is an accelerometer signal.

B.6 Servo limits (NXT)

The servo limits are configured in `servo_get_limits.vi`[5]. The default value of the indicator gives the servo limits. For each servo three position points are given (-1, 0 and 1 points) in PWM width in *μ*s. In addition, the maximum speed is given in PWM width *μ*s per 24 *μ*s. Element *i* in this array refers to servo *i* as defined in `motor_id.ctl`. Figure B.2 shows the values used during development.


Servo limits

0

-1 PWM	1650	1490	2000	1180	840	2000	2100
0 PWM	1650	1940	1500	1180	840	1470	2100
1 PWM	1650	2370	860	740	2020	1050	1470
Max speed	10	40	50	50	100	10	100

Figure B.2: `servo_get_limits.vi` default output values as used during development.

[5]Located in `./NXT/VI/Servo/`

B.7 NXT demo files

The NXT demo files are downloaded using the NXT terminal available from `Tools - NXT tools - NXT terminal`. Because the current NXT firmware does not support directory listings, the list of available demo files must be changed in code. This is done in `application_demo.nxt.vi`.

To generate a demo file with the structure shown in figure 5.1 the following MATLAB script can be used.

```
% GENERATE DEMO FILE
% generate_file_pos(FILENAME, CONFIG, DATA) writes a demo-file to the
% given file that may be run on the NXT.
%
%     FILENAME:   The name of the file to write. Will be overwritten if
%                 already existing.
%     CONFIG:     A cluster containing the following fields:
%                    type:     POS or SPEED.
%                    datatype:    Setpoints type
%                                     0=PWM, 1=[-1,1] for POS
%                                     0=PWM per 24 us for SPEED
%                    dt:       Time to sleep (in ms) between the onset of
%                              each setpoint.
%     DATA:       An nSetpoints*nServos matrix containing the set-points.

% Jorn Bersvendsen, jornb87@gmail.com
% 24. May 2011

function generate_file(filename, config, data)
    % Open file reference
    file = fopen(filename, 'w+');

    % Error handling
    if file == -1
        disp([  'Error: Could not open/create file ',...
                filename,...
                ' for writing.']);
        return;
    end

    % Accept data in both column and row order
    s = size(data);
    if s(1) < s(2)
        data = data';
        s = size(data);
    end
    nPoints = s(1);
    nMotors = s(2);

    % Write type (POS/SPEED)
    fwrite(file, length(config.type), 'uint8');
    fwrite(file, config.type, 'char*1');

    % Write config
    fwrite(file, config.datatype, 'uint8');      % setpoint type
    fwrite(file, config.dt, 'uint8');            % delta time
    fwrite(file, s(2), 'uint8');                 % number of servos;

    % Write data
    for i=1:nPoints
        for j=1:nMotors
            if strcmp('SPEED', config.type)
                % SPEED type
                fwrite(file, data(i, j), 'uint8');
            else
                % POS type
                d = data(i, j);

                % Check for [-1, 1] unit type
                if config.datatype == 1
                    d = floor((max(-1, min(1, d)) + 1) * 32767);
                end

                fwrite(file, d, 'uint16');
            end
        end
    end

    % Write carrige return
    fprintf(file, '\r');

    % Close file
    fclose(file);
end
```

B.8 Trigno analog outputs

In order to activate the analog outputs the Trigno base station must be connected to a computer by USB. Running the “Trigno Analog Output” application provided with the Trigno system activates the analog outputs.

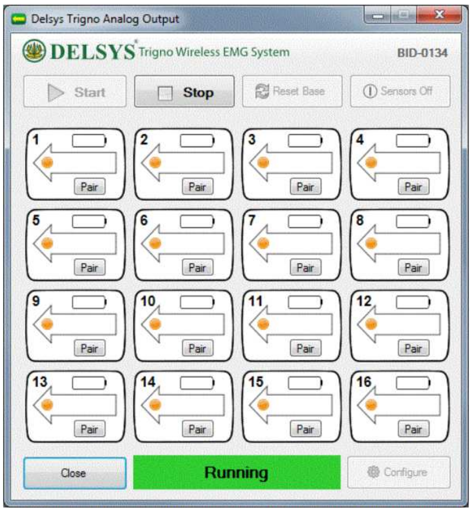


Figure B.3: Screenshot of the Trigno analog output application.

C
Quick guide to LabVIEW

This chapter will give an overview of the LabVIEW programming language and development environment. It is based on (Elliott et al. 2007, Hosek et al. 2009, Kehtarnavaz 2007).

LabVIEW (*La*boratory *V*irtual *I*nstrumentation *E*ngineering *W*orkspace) is a graphical programming language. The source code for a program is not written but drawn, similar to a flowchart diagram.

The main unit of execution, similar to a function/method in regular programming languages such as C, is the VI (*V*irtual *I*nstrument). A VI has three parts; a front panel, a block diagram and a connector pane.

The *front panel* is analogous to a method declaration in C and defines a method by its inputs and outputs. However, the front panel is also a graphical user interface in that all (i.e. *almost* all) inputs and outputs are intractable. The intractable inputs are called *controls* and outputs are called *indicators*.

The *block diagram* is analogous to the method source code in C. Every control and indicator is represented as an object on both the front panel and the block diagram. However, many different front panel objects are represented the same way on the block diagram. A boolean control on the block diagram may be a switch or a button on the front panel. Similarly, slides, knobs, dials an gauges are all represented as numeric controls on the block diagram.

Since LabVIEW is a graphical programming languages, the VI is called using an icon on the block diagram instead of a function name in the source code. The *connector* *pane* defines how the VI looks and is used on the block diagram.

Examples of different block diagram structures with explanations are shown in figure C.1.

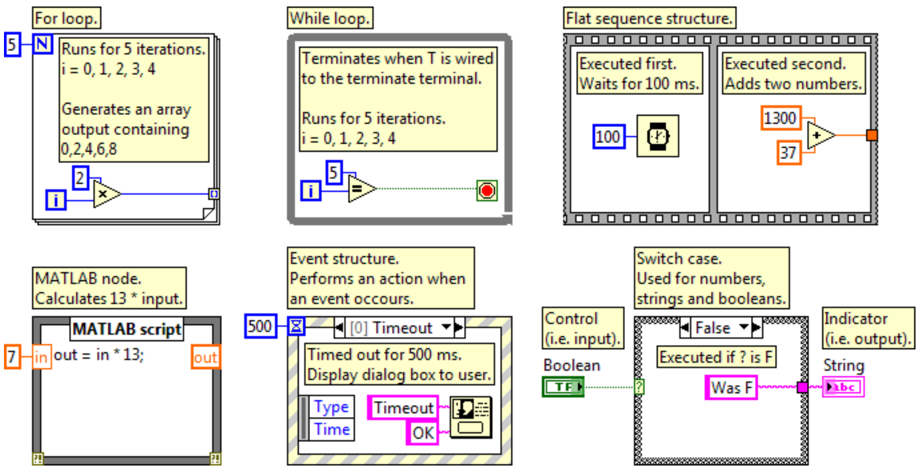

For loop.
5
N
Runs for 5 iterations. i = 0, 1, 2, 3, 4
Generates an array output containing 0,2,4,6,8
2
i
While loop.
Terminates when T is wired to the terminate terminal.
Runs for 5 iterations. i = 0, 1, 2, 3, 4
5
i
Flat sequence structure.
Executed first. Waits for 100 ms.
100
Executed second. Adds two numbers.
1300
37
MATLAB node. Calculates 13 * input.
MATLAB script
7
in
out = in * 13;
out
Event structure. Performs an action when an event occours.
500
[0] Timeout
Timed out for 500 ms. Display dialog box to user.
Type
Time
Timeout
OK
Switch case. Used for numbers, strings and booleans.
Control (i.e. input).
Boolean
False
Executed if ? is F
Was F
Indicator (i.e. output).
String

Figure C.1: Commonly used LabVIEW block diagram structures.

Note that there is one control (`Boolean`) and one indicator (`String`) in this code, both on the switch case in the lower right corner. Their representation on the front panel is shown in figure C.2.

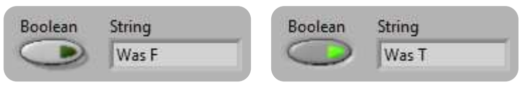


Figure C.2: Front panel of the block diagram in figure C.1 after execution with different boolean control values.