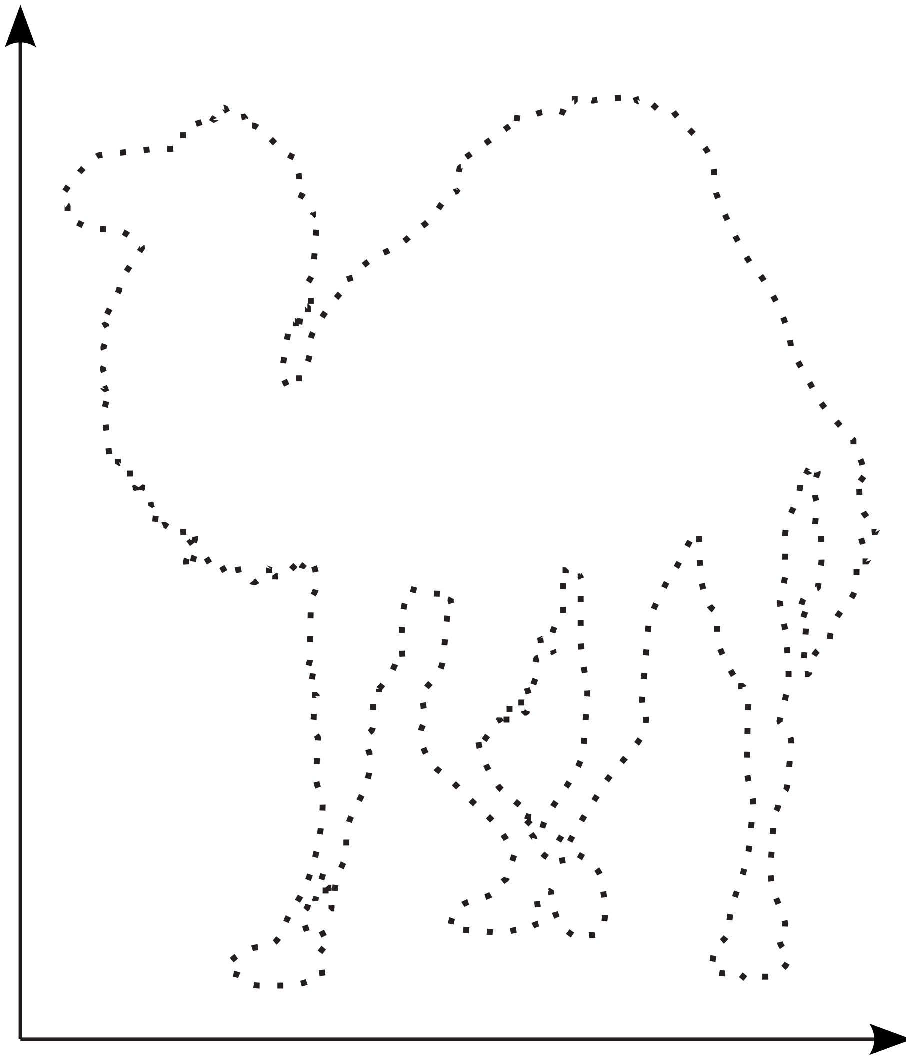
Daniel Schröer, Andreas K. Hüttel, and others

Lab::VISA Documentation



February 22, 2012 (Revision: 595)

Contents

1 Using Lab::VISA **5**

1.1 Lab::VISA::Tutorial . . . 5

1.1.1 Introduction . . . 5

1.1.2 Measurement automation basics . . . 5

1.1.3 Architecture . . . 6

1.1.4 Using pure VISA calls . . . 7

1.1.5 Using the Lab::Instrument class . . . 10

1.1.6 Using Lab::Instrument::xxx virtual instruments . . . 11

1.1.7 Using Lab::Tools . . . 13

1.1.8 References . . . 17

2 The Lab::VISA package **19**

2.1 Lab::VISA . . . 19

3 The Lab::Instrument package **23**

3.1 Lab::Instrument . . . 23

3.2 Voltage Sources . . . 27

3.2.1 Lab::Instrument::Source . . . 27

3.2.2 Lab::Instrument::Yokogawa7651 . . . 31

3.2.3 Lab::Instrument::KnickS252 . . . 35

3.2.4 Lab::Instrument::IOtech488 . . . 37

3.2.5 Lab::Instrument::SRS_SIM928 . . . 39

3.2.6 Lab::Instrument::Dummysource . . . 41

3.3 Multimeters . . . 43

3.3.1 Lab::Instrument::HP34401A . . . 43

3.3.2 Lab::Instrument::HP34411A . . . 47

3.3.3 Lab::Instrument::HP34420A . . . 51

3.3.4 Lab::Instrument::HP34970A . . . 53

3.3.5 Lab::Instrument::HP3458A . . . 55

3.4 Amplifiers . . . 57

3.4.1 Lab::Instrument::SR830 . . . 57

3.5 High Frequency . . . 59

3.5.1 Lab::Instrument::SR780 . . . 59

3.5.2 Lab::Instrument::SR620 . . . 63

3.5.3 Lab::Instrument::HP8360 . . . 65

3.5.4 Lab::Instrument::Agilent81134A . . . 67

3.6 Cryostat Handling . . . 69
3.6.1 Lab::Instrument::IsoBus . . . 69
3.6.2 Lab::Instrument::ILM . . . 71
3.6.3 Lab::Instrument::ITC . . . 73
3.6.4 Lab::Instrument::Lakeshore336 . . . 75
3.6.5 Lab::Instrument::Lakeshore370 . . . 77
3.6.6 Lab::Instrument::TRMC2 . . . 79

4 The Lab::Tools package 81
4.1 Lab::Data::Writer . . . 81
4.2 Lab::Measurement . . . 83
4.3 Lab::Data::Meta . . . 87
4.4 Lab::Data::XMLtree . . . 91
4.5 Lab::Data::Plotter . . . 95
4.6 plotter.pl . . . 97

1 Using Lab::VISA

1.1 Lab::VISA::Tutorial

Tutorial on using Lab::VISA and related packages

1.1.1 Introduction

Lab::VISA and its related packages allow to perform test and measurement tasks with Perl scripts. It provides an interface to National Instruments' NI-VISA library, making the standard VISA calls available from within Perl programs. Dedicated instrument driver classes relieve the user from taking care for internal details and make measurements as easy as

```
$voltage=$multimeter->read_voltage().
```

The Lab::... software is divided into three parts. They are built on top of each other and provide increasing comfort. Your measurement scripts can be based on each of these stages.

The lowest level is *Lab::VISA*. It makes the NI-VISA library accessible from perl and therefor allows to make any standard VISA call.

The modules in the *Lab::Instrument* package make communication with instruments easier by silently handling the protocol involved.

Package *Lab::Tools* is the highest abstraction layer. These modules provide support for writing good measurement scripts. They offer means of saving data and related meta information to disk, plotting data etc.

This tutorial will explain how to write measurement scripts that build on any of these stages. However, this tutorial does not intend to teach the Perl language itself. Some introduction into VISA and GPIB terminology is given, but then some familarity also with these concepts is assumed. Not much is required. If you feel the need for more information on Perl or VISA/GPIB, please see the References section[1-6].

1.1.2 Measurement automation basics

This section provides a very brief introduction to the terms VISA and GPIB. For a more detailed explanation of the VISA and GPIB standards, the involved communication principles and the available commands for your specific instruments, please refer to the literature[1-3]. Usage of the higher level modules from the *Lab::Instrument* package requires almost no knowledge about VISA and GPIB at all.

VISA

Traditionally, test and measurement instruments can be connected and controlled via various standards and protocols. VISA, the Virtual Instrument Software Architecture[1,2], is an effort to provide a single standarised interface to communicate with instruments via several protocols. It was developed by the VXIplug&play Systems Alliance[4] and is currently maintained by the IVI foundation[5]. VISA can control VXI, GPIB, serial, or computer-based instruments and makes the appropriate driver calls depending on the type of instrument used. Hence, VISA is located in the application layer. The National Instruments NI-VISA library is one implementation of the VISA standard.

In one word: VISA tries to make it unimportant, how an instrument is connected physically.

GPIB

GPIB (IEEE488)[3] is a lower lying standard invented by Hewlett-Packard. It describes a way of connecting instruments. The standard is divided into the physical layer IEEE488.1 that defines cables and signals and the command layer IEEE488.2 that describes a syntax for messages between communicating instruments. SCPI (Standard Commands for Programmable Instruments) is an extension of IEEE488.2 and refines the available commands further, with the goal of obtaining a language that is independent of the exact model of the instruments in use. This could be very useful, as, in theory, it would allow you to exchange one instrument in your setup with a similar one from another manufacturer, without having to change your measurement software. In practise however, not many instruments support this standard, and even if, small differences make things a pain. As described below, the Lab::Instrument package follows another route to achieve interchangeability by providing common interfaces for similar instruments at a much higher level (e.g. the *Lab::Instrument::Source* interface).

In one word: GPIB tries to make communication with various instruments more similar.

1.1.3 Architecture

A schematic view of the various software layers between your perl measurement script and the instrument hardware looks like this:

```
+-------------------------+ +---------------+ +------------------+
|Lab::Instrument::HP34401A| |L::I::KnickS252| |L::I::Yokogawa7651|
+-----------------------+-+ +----+----------+ +----+-------------+
                        |        |                 |
                        |     +--+-----------------+--+
                        |     |Lab::Instrument::Source|
                        |     +---+-------------------+
                        |         |
                     +--+---------+--+
                     |Lab::Instrument|
                     +-------+-------+
                             |
                        +----+----+
                        |Lab::VISA|
                        +----+----+
                             |
                     +-------+-------+
                     |NI-VISA Library|
                     +---+-------+---+
                         |       |
              +----------+-+   +-+-----+
              |GPIB Library|   |  OS   |
              +----------+-+   +-+-----+
                         |       |
                         |       |Serial connection
          GPIB connection|       |TCP/IP connection
                         |       |USB connection
                         |       |
                +--------+-+   +-+---------+
                |Instrument|   |Instrument|
                +----------+   +----------+
```

The *Lab::VISA* module provides a perl binding for National Instruments' NI-VISA library. It makes the standard VISA calls available from within Perl programs.

The *Lab::Instrument* module builds on top of Lab::VISA and simplifies the routine tasks of opening VISA resources, sending and receiving messages.

The instrument classes like *Lab::Instrument::KnickS252* are specialized modules for certain instruments. Most other measurement software packages would call this a virtual instruments or an instrument drivers. Each such class provides methods that are specific for one instrument. The Lab::Instrument::IPS120_10 class for example class is dedicated to a certain magnet power supply and therefore provides methods like `set_target_field`. Similar instruments (e.g. various voltage sources) however share common interfaces (e.g. *Lab::Instrument::Source*) to make interchangeability of similar instruments possible.

1.1.4 Using pure VISA calls

First we will see how to use the plain VISA interface and communicate with an instrument with standard VISA `viRead` and `viWrite` calls. It will show that this method is

rather laborious. Later we will learn how `Lab::Instrument` makes life easier.

All the examples in the following sections can be found in the Tutorials directory of the *Lab::VISA* package.

```
#!/usr/bin/perl

# example1.pl

use strict;
use Lab::VISA;

# Initialize VISA system and
# Open default resource manager
my ($status,$default_rm)=Lab::VISA::viOpenDefaultRM();
if ($status != $Lab::VISA::VI_SUCCESS) {
    die "Cannot open resource manager: $status";
}

# Open one resource (an instrument)
my $gpib=24;            # we want to open the instrument
my $board=0;            # with GPIB address 24
                        # connected to GPIB board 0 in our computer
my $resource_name=sprintf("GPIB%u::%u::INSTR",$board,$gpib);

($status, my $instr)=Lab::VISA::viOpen(
    $default_rm,        # the resource manager session
    $resource_name,     # a string describing the
    $Lab::VISA::VI_NULL,# access mode (no special mode)
    $Lab::VISA::VI_NULL # time out for open (no time out)
);
if ($status != $Lab::VISA::VI_SUCCESS) {
    die "Cannot open instrument $resource_name. status: $status";
}

# We set a time out for communication with this instrument
$status=Lab::VISA::viSetAttribute(
    $instr,             # the session identifier
    $Lab::VISA::VI_ATTR_TMO_VALUE,  # which attribute to modify
    3000                # the new value
);
if ($status != $Lab::VISA::VI_SUCCESS) {
    die "Error while setting timeout value: $status";
}

# Clear the instrument
my $status=Lab::VISA::viClear($instr);
if ($status != $Lab::VISA::VI_SUCCESS) {
    die "Error while clearing instrument: $status";
}

# Now we are going to send one command and read the result.
```

```
# We send the simple SCPI command "*IDN?" which asks the instrument
# to identify itself. Of course the instrument must support this
# command, in order to make this example work.
my $cmd="*IDN?";
($status, my $write_cnt)=Lab::VISA::viWrite(
    $instr,              # the session identifier
    $cmd,                # the command to send
    length($cmd)         # the length of the command in bytes
);
if ($status != $Lab::VISA::VI_SUCCESS) {
    die "Error␣while␣writing:␣$status";
}


# Now we will read the instruments reply
($status,                # indicates if the operation was successful
 my $result,             # the answer string
 my $read_cnt)=          # the length of the answer in bytes
Lab::VISA::viRead(
    $instr,              # the session identifier
    300                  # read 300 bytes
);
if ($status != $Lab::VISA::VI_SUCCESS) {
    die "Error␣while␣reading:␣$status";
}
# The result string will be 300 bytes long, but only $read_cnt
# bytes are part of the answer. We cut away the rest.
$result=substr($result,0,$read_cnt);

print "$result\n";

# As good citizens we'll cleanup now.
# Close the instrument
$status=Lab::VISA::viClose($instr);
# And the resource manager
$status=Lab::VISA::viClose($default_rm);

__END__
```

First we have to open a resource manager. This manager can then provide us with a handle to one instrument. We try to open an instrument that is connected via GPIB to the GPIB board 0 in our computer and uses the GPIB address 24. This is specified by the resource name. We then ask the instrument for its identification string, read the answer and print it. We do so by sending the `*IDN?` command. This is a standard SCPI command, that all instruments that support the SCPI language, will understand. Agilent instruments do so for example.

We see that there is a lot of protocol overhead involved, that makes this very simple example a bit lengthy and ugly. These things should be factored out. The Lab::Instrument class can do all this dirty work for us, as we will learn in the next section.

1.1.5 Using the Lab::Instrument class

The *Lab::Instrument* class can do for us the routine work of connecting to certain instrument.

```
#!/usr/bin/perl

# example2.pl

use strict;
use Lab::Instrument;

my $gpib=24;              # we want to open the instrument
my $board=0;              # with GPIB address 24
                          # connected to GPIB board 0 in our computer

# Create an instrument object
my $instr=new Lab::Instrument($board,$gpib);

my $cmd="*IDN?";

# Query the instrument
# Query is a combined Write and Read
my $result=$instr->Query($cmd);

print $result;

__END__
```

This program achieves exactly the same as `example1.pl`, but with only two lines of code: one to open the instrument, one to query it. We don't have to care about resource managers and string lengths and cleaning up. Lab::Instrument does it for us. Now that's already quite nice, eh?

Let's see another example. This time we will send a great bunch of commands to an Agilent 81134A pulse generator, to set it up for pulse mode.

```
#!/usr/bin/perl

# example3.pl

use strict;
use Lab::Instrument;

# Open instrument
# We use the other form of the constructor here.
my $instr=new Lab::Instrument({
    GPIB_board      => 0,
    GPIB_address    => 10
});
```

```
# Send a bunch of commands to configure instrument
for ((
# Protect the DUT
     ':OUTP:CENT␣OFF',         #disconnect channels

# Set up the Instrument
     ':FUNC␣PATT',             #set mode to Pulse/Pattern
     ':PER␣20␣ns',             #set period to 20 ns

# Set up Channel 1
     ':FUNC:MODE1␣PULSE',      #set pattern mode to Pulse
     ':WIDT1␣5␣ns',            #set width to 5 ns
     ':VOLT1:AMPL␣2.000␣V',    #set ampl to 2 V
     ':VOLT1:OFFSET␣1.5␣V',    #set offset to 1.5 V
     ':OUTP1:POS␣ON',          #enable output channel 1

# Generate the Signals
     ':OUTP:CENT␣ON',          #reconnect the channels
     ':OUTP0:SOUR␣PER',        #use trigger mode Pulse
     ':OUTP0␣ON',              #enable trigger output
)) {
     $self->{vi}->Write($_);
}

__END__
```

This example shows that Perl's great list handling makes it the ideal language for instrument control and data aquisition tasks.

By only using *Lab::Instrument* you should already be able to do about everything that can be done the instruments in your lab.

1.1.6 Using Lab::Instrument::xxx virtual instruments

Many common tasks, like reading a voltage from a digital multimeter, require that a series of GPIB commands is sent to an instrument. These commands are different for similar instruments from different manufacturers.

The virtual instrument classes in the *Lab::Instrument* package attempt to hide these details from the user by providing high level methods like `set_voltage($voltage)` and `get_voltage()`.

Additionally they provide an optional safety mechanism for voltage sources. This is used to protect sensitive samples which could be destoyed by sudden voltage changes. See the documentation of the *Lab::Instrument::Source* module for details.

```
#!/usr/bin/perl

# example4.pl

use strict;
use Lab::Instrument::HP34401A;
```

```
my $gpib=24;            # we want to open the instrument
my $board=0;            # with GPIB address 24
                        # connected to GPIB board 0 in our computer

# Create an instrument object
my $hp=new Lab::Instrument::HP34401A($board,$gpib);

# Use the id method to query the instruments ID string
my $result=$hp->id();

print $result;

__END__
```

This example show the usage of a dedicated virtual instrument class, namely *Lab::Instrument::HP34401A*, the driver for a Hewlett-Packard/Agilent 34401A digital multimeter. An instance of this class is created that is connected to one certain instrument. We use the `id()` method that returns the instrument's id string again.

Next we see an example on how to use the safety mechanism of *Lab::Instrument::Source* that is inherited by voltage sources like *Lab::Instrument::Yokogawa7651*.

```
#!/usr/bin/perl

# example5.pl

use strict;
use Lab::Instrument::Yokogawa7651;

unless (@ARGV > 0) {
    print "Usage: $0 GPIB-address [goto_voltage]\n";
    exit;
}

my ($gpib,$goto)=@ARGV;

my $source=new Lab::Instrument::Yokogawa7651(0,$gpib);

if (defined $goto) {
    $source->sweep_to_voltage($goto);
} else {
    print $source->get_voltage();
}

__END__
```

1.1.7 Using Lab::Tools

With the tools introduced so far you should be able to easily write short individual scripts for your measurement tasks. These scripts will probably serve as well as all other home grown solutions using LabView or whatever. The modules in the `Lab::Tools` package now provide additional tools to write better measurement scripts.

One main goal is to provide means to keep additional information stored along with the raw measured data. Additional information means all the notes that you would usually write down in your laboratory book, like date and time, settings of additional instruments, the environment temperature, the color of the shirt you were wearing while recording the data and everything else that might be of importance for a later interpretation of the data. In my experience, having to write these things in a book by hand is tedious and error-prone. It's the kind of job that computers were made for.

Another goal is to free the laborant from having to repeat himself all the time when the data is used for analysis or presentation. Let us assume that, for example, you are measuring a very small current with the help of a current amplifier. This current amplifier will output a voltage that is proportional to the original current, so in fact you will be measuring a voltage that can be converted to the original current by multiplying it with a certain factor. The last paragraph has shown that the *Lab::Tools* modules will help you to keep track of this additional information *current amplifier constant of proportionality*. But as long as the precise formula for this transformation is not stored together with the data, you will still find yourself repeatedly typing in the same expressions, whenever you work with the data. This is where the *axis* concept comes into play. Already at the time you are preparing your measurement script, you define an *axis* named *current* that stores the expression to calculate the current from the voltage. From there you work with the current-axis and will never have to care about the conversion again. And of course you can define many different axes. Read on!

The Meta data

The general concept is that a *dataset* is composed out of *data* and *metadata*, i.e. additional information about the *data*. This *metadata* is maintained by the *Lab::Data::Meta* class and is usually stored in a file `dataset_filename.META`, while the *data* is saved in `dataset_filename.DATA`.

The *meta* file is stored in YAML or XML format and contains a number of elements which are defined in *Lab::Data::Meta*. The most important ones are *column*, *block*, *axis* and *plot*. For the following discussion of these fields, let's assume a *data* file that looks like this:

```
0.01   2.0   3
0.01   2.1   3.4
0.01   2.2   2.9

0.02   2.0   1.7
0.02   2.1   2.4
0.02   2.2   2.2
```

This dataset shows an example where one quantity (third column) is measured in dependence of two others (first and second column). The data was recorded in two traces, where one input value is kept constant (1st column) and then for every setting of the other input value (2nd column) a datapoint is taken (3rd column). Then the the first input value is increased and the next trace is recorded.

column The above example measurement has three columns. You will want to store additional information for each of these columns: What is being set or measured, what is the unit of the stored value etc. This information is stored in the *column* records of the *meta* file. More details on the available fields is given in the *Lab::Data::Meta* manpage.

block The example data above was aquired in two traces or scans or sweeps, which are separated by an empty line in the *data* file now. *Lab::Data::Meta* adopts the Gnuplot[7] terminology and calls these *blocks*. Along with every block, additional information like the time the trace was started can be saved. Most of this is done automatically. See the *Lab::Data::Meta* and *Lab::Measurement* manpages.

axis Usually you will not want to work with the raw data as it is stored in the columns of the file. For example you could want to plot the sum of two columns. Also you might want to display the data using another unit. Therefor you can define a new axis, that is defined as the sum of these two columns times any factor (which can be saved as a constant, see below) for the right unit. The expression `amp * ($C1 + 10 * $C2)` defines an axis as the sum of two columns multiplied with a constant `amp`. Additionally, axes have labels, ranges and such.

plot With the plot element, default views on the data be defined. These views can then be plotted with a single command, using the *Lab::Data::Plotter* module and the script `plotter.pl`. Because all the necessary information is stored in the *meta* file, these plots will automatically contain the right axes, ranges, labels, units and any other information you wish! Plots can already be defined at the time the measurement script is written, and can also be added later. If you use the *Lab::Measurement* module, you can display any of these plots live, while the data is being aquired. Since this entire system can run on Linux, you can X-forward this graph to your remote desktop at the beach. Imagine the possibilities.

constant This section of meta data can be used to store additional values that are important for the later interpretation of the raw data. Examples for such values could be amplification factors, voltage dividers etc. Constants have names that can be used in expressions of `axis` definitions.

The Lab::Measurement class

The *Lab::Measurement* class makes it easy to write a measurement script that takes advantage of the meta data system introduced above...

Examples

```
#!/usr/bin/perl

# example6.pl

# Eine Spannungsquelle fahren, Leitfähigkeit (ohne Lock-In) messen

use strict;
use Lab::Instrument::KnickS252;
use Lab::Instrument::HP34401A;
use Time::HiRes qw/usleep/;
use Lab::Measurement;

##################################

my $start_voltage   =-0.05;
my $end_voltage     =-0.25;
my $step            =-1e-3;

my $knick_gpib      =4;
my $hp_gpib         =24;

my $v_sd            =-300e-3/1000;
my $amp             =1e-9;    # Ithaco amplification

my $R_Kontakt       =1089;

my $sample          ="S5c (81059)";
my $title           ="QPC links unten";
my $comment         =<<COMMENT;
Strom von 12 nach 14; V_{SD,DC}=$v_sd V; Lüftung an; Ca. 25mK.
Ithaco: Amplification $amp, Supression 10e-10 off, Rise Time 0.3ms.
Fahre Ghf4 (Yoko04)
COMMENT

##################################

my $knick=new Lab::Instrument::KnickS252({
    'GPIB_board'    => 0,
    'GPIB_address'  => $knick_gpib,
    'gate_protect'  => 1,

    'gp_max_volt_per_second' => 0.002,
});

my $hp=new Lab::Instrument::HP34401A(0,$hp_gpib);
```

```
my $measurement=new Lab::Measurement(
    sample          => $sample,
    title           => $title,
    filename_base   => 'qpctest',
    description     => $comment,

    live_plot       => 'QPC␣current',

    constants       => [
        {
            'name'          => 'G0',
            'value'         => '7.748091733e-5',
        },
        {
            'name'          => 'RKontakt',
            'value'         => $R_Kontakt,
        },
        {
            'name'          => 'V_SD',
            'value'         => $v_sd,
        },
        {
            'name'          => 'AMP',
            'value'         => $amp,
        },
    ],
    columns         => [
        {
            'unit'          => 'V',
            'label'         => 'Gate␣voltage',
            'description'   => 'Applied␣to␣gates␣via␣low␣path␣filter
                .',
        },
        {
            'unit'          => 'V',
            'label'         => 'Amplifier␣output',
            'description'   => "Voltage␣output␣by␣current␣amplifier␣
                set␣to␣$amp.",
        }
    ],
    axes            => [
        {
            'unit'          => 'V',
            'expression'    => '$C0',
            'label'         => 'V_{Gate}',
            'min'           => ($start_voltage < $end_voltage)
                               ? $start_voltage
                               : $end_voltage,
            'max'           => ($start_voltage < $end_voltage)
                               ? $end_voltage
                               : $start_voltage,
            'description'   => 'Gate␣voltage',
        },
        {
```

```
                'unit'            => 'A',
                'expression'      => "abs(\$C1)*AMP",
                'label'           => 'I_{QPC}',
                'description'     => 'QPC current',
            },
            {
                'unit'            => '2e^2/h',
                'expression'      => "(1/(V_SD/(-\$C2*AMP)-RKontakt))/G0",
                'label'           => "G_{QPC}",
                'description'     => "QPC conductance",
                'min'             => -0.1,
                'max'             => 7
            },

        ],
        plots             => {
            'QPC current'     => {
                'type'            => 'line',
                'xaxis'           => 0,
                'yaxis'           => 1,
                'grid'            => 'xtics ytics',
            },
            'QPC conductance'=> {
                'type'            => 'line',
                'xaxis'           => 0,
                'yaxis'           => 3,
                'grid'            => 'ytics',
            }
        },
);

$measurement->start_block();

my $stepsign=$step/abs($step);
for (my $volt=$start_voltage;
     $stepsign*$volt<=$stepsign*$end_voltage;
     $volt+=$step) {
    $knick->set_voltage($volt);
    usleep(500000);
    my $meas=$hp->read_voltage_dc(10,0.0001);
    $measurement->log_line($volt,$meas);
}

my $meta=$measurement->finish_measurement();
```

TODO: more examples

1.1.8 References

[1] NI-VISA User Manual (http://www.ni.com/pdf/manuals/370423a.pdf)

[2] NI-VISA Programmer Manual (http://www.ni.com/pdf/manuals/370132c.pdf)

[3] NI 488.2 User Manual (http://www.ni.com/pdf/manuals/370428c.pdf)

[4] http://www.vxipnp.org/

[5] http://www.ivifoundation.org/

[6] http://perldoc.perl.org/

[7] http://www.gnuplot.info/

2 The Lab::VISA package

2.1 Lab::VISA

Perl interface to National Instruments VISA library

SYNOPSIS

```
use Lab::VISA;
```

DESCRIPTION

This library offers a Perl interface to National Instruments' NI-VISA library.

With this library you can easily control the instruments in your lab (multimeters, voltage sources, magnet sources, pulse generators etc.) with Perl. You can perform complicated measurement jobs with just some Perl loops.

It comes even better: The general Lab::Instrument class reduces the communication overhead to minimal `read`, `write` and `query` methods. And on top of this, there are specialized instrument classes (virtual instruments) such as Lab::Instrument::HP34401A, that offer even more high level comfort with methods as `read_voltage`. Everything is prepared so that you can just start the measurement.

The Lab::Tools package offers classes to simplify the task to log data to files and to maintain this data.

This manpage describes the perl syntax of the API. Each function is explained with some sentences which I cited from [1]. See this manual for further documentation on the library.

Installation instructions can be found in *Lab::VISA::Installation*.

A general tutorial for using Lab::VISA and assorted packages is located in *Lab::VISA::Tutorial*.

[1] NI-VISA Programmer Reference Manual. Part Number 370132C-01.

FUNCTIONS

viClear

```
$status=Lab::VISA::viClear($vi);
```

The `viClear()` operation performs an IEEE 488.1-style clear of the device.

viClose

```
$status=Lab::VISA::viClose($object);
```

The `viClose()` operation closes a session, event, or a find list. In this process all the data structures that had been allocated for the specified vi are freed. Calling `viClose()` on a VISA Resource Manager session will also close all I/O sessions associated with that resource manager session.

viFindNext

```
($status, $instrDesc)=
    Lab::VISA::viFindNext($findList);
```

The `viFindNext()` operation returns the next device found in the list created by `viFindRsrc()`. The list is referenced by the handle that was returned by `viFindRsrc()`.

viFindRsrc

```
($status, $findList, $retcnt, $instrDesc)=
    Lab::VISA::viFindRsrc($sesn, $expr);
```

The `viFindRsrc()` operation matches the value specified in the `expr` parameter with the resources available for a particular interface.

On successful completion, this function returns the first resource found (`instrDesc`) and returns a count (`retcnt`) to indicate if there were more resources found for the designated interface. This function also returns, in the `findList` parameter, a handle to a find list. This handle points to the list of resources and it must be used as an input to `viFindNext()`. When this handle is no longer needed, it should be passed to `viClose()`.

The search criteria specified in the `expr` parameter has two parts: a regular expression over a resource string and an optional logical expression over attribute values. The regular expression is matched against the resource strings of resources known to the VISA Resource Manager. If the resource string matches the regular expression, the attribute values of the resource are then matched against the expression over attribute values. If the match is successful, the resource has met the search criteria and gets added to the list of resources found.

All resource strings returned by `viFindRsrc()` will always be recognized by `viOpen()`. However, `viFindRsrc()` will not necessarily return all strings that you can pass to `viParseRsrc()` or `viOpen()`. This is especially true for network and `TCPIP` resources.

viGetAttribute

```
($status, $attrState)=
    Lab::VISA::viGetAttribute($object, $attribute);
```

The `viGetAttribute()` operation is used to retrieve the state of an attribute for the specified session, event, or find list.

viOpen

```
($status, $vi)=
    Lab::VISA::viOpen($sesn, $rsrcName, $accessMode, $openTimeout);
```

The `viOpen()` operation opens a session to the specified resource. It returns a session identifier that can be used to call any other operations of that resource. The address string passed to `viOpen()` must uniquely identify a resource.

For the parameter accessMode, the value `VI_EXCLUSIVE_LOCK` (1) is used to acquire an exclusive lock immediately upon opening a session; if a lock cannot be acquired, the session is closed and an error is returned. The value `VI_LOAD_CONFIG` (4) is used to configure attributes to values specified by some external configuration utility. Multiple access modes can be used simultaneously by specifying a bit-wise OR of the values other than `VI_NULL`. NI-VISA currently supports `VI_LOAD_CONFIG` only on Serial INSTR sessions.

viOpenDefaultRM

```
($status, $sesn)=
    Lab::VISA::viOpenDefaultRM();
```

The `viOpenDefaultRM()` function must be called before any VISA operations can be invoked. The first call to this function initializes the VISA system, including the Default Resource Manager resource, and also returns a session to that resource. Subsequent calls to this function return unique sessions to the same Default Resource Manager resource.

When a Resource Manager session is passed to `viClose()`, not only is that session closed, but also all find lists and device sessions (which that Resource Manager session was used to create) are closed.

viRead

```
($status, $buf, $retCount)=
    Lab::VISA::viRead($vi, $count);
```

The `viRead()` operation synchronously transfers data. The data read is to be stored in the buffer represented by buf. This operation returns only when the transfer terminates. Only one synchronous read operation can occur at any one time.

viSetAttribute

```
$status=Lab::VISA::viSetAttribute($vi, $attribute, $attrState);
```

The `viSetAttribute()` operation is used to modify the state of an attribute for the specified object.

viWrite

```
($status, $retCount)=
    Lab::VISA::viWrite($vi, $buf, $count);
```

The `viWrite()` operation synchronously transfers data. The data to be written is in the buffer represented by buf. This operation returns only when the transfer terminates. Only one synchronous write operation can occur at any one time.

3 The Lab::Instrument package

3.1 Lab::Instrument

General VISA based instrument

SYNOPSIS

```
use Lab::Instrument;

my $hp22 =  new Lab::Instrument(0,22); # GPIB board 0, address 22
print $hp22->Query('*IDN?');
```

DESCRIPTION

`Lab::Instrument` offers an abstract interface to an instrument, that is connected via GPIB, serial bus, USB, ethernet, or Oxford Instruments IsoBus. It provides general `Read`, `Write` and `Query` methods, and more.

It can be used either directly by the programmer to work with an instrument that doesn't have its own perl class (like Lab::Instrument::HP34401A). Or it can be used by such a specialized perl instrument class (like `Lab::Instrument::HP34401A`) to delegate the actual visa work. (All the instruments in the default package do so.)

CONSTRUCTOR

new

```
$instrument  = new Lab::Instrument($board,$addr);

$instrument2 = new Lab::Instrument({
    GPIB_board   => $board,
    GPIB_address => $addr
});

$instrument3 = new Lab::Instrument($resourcename);

$instrument4 = new Lab::Instrument($isobus,$addr);
```

Creates a new instrument object and open the instrument with GPIB address `$addr` connected to the GPIB board `$board` (usually 0). Alternatively, the VISA resource name `$resourcename` can be specified as string for serial or USB devices. All instrument classes that internally use the `Lab::Instrument` module (that's all instruments in the default distribution) can use these three forms of the constructor.

Lastly, an IsoBus device can be instantiated by providing the IsoBus instrument `$isobus` (of type `Lab::Instrument::IsoBus`) as first parameter and the numeric IsoBus address of the device `$addr` as second parameter.

METHODS

Write

```
$write_count=$instrument->Write($command);
```

Sends the command `$command` to the instrument.

Read

```
$result=$instrument->Read($length);
```

Reads a result of maximum length `$length` from the instrument and returns it. Dies with a message if an error occurs.

BrutalRead

```
$result=$instrument->BrutalRead($length);
```

Same as Read, but this command ignores all error conditions.

Query

```
$result=$instrument->Query($command, $wait_query, $wait_status);
```

Sends the command `$command` to the instrument and reads a result from the instrument and returns it. The length of the read buffer is haphazardly set to 300 bytes.

Optional second and third arguments specify waiting times, i.e. how long the instrument needs to process the query and provide a result (`$wait_query`) and how long the instrument needs to react on a command at all and set the status line (`$wait_status`). Both parameters are set to 10us if not specified in the command line.

LongQuery

```
$result=$instrument->LongQuery($command, $wait_query, $wait_status);
```

Same as Query, but with a read buffer size of 10240 bytes. If you need to read even more data, you will have to use separate `Write` and `Read` calls.

BrutalQuery

```
$result=$instrument->BrutalQuery($command);
```

Same as Query (i.e. buffer size 300 bytes), but with a lot less finesse. :) Sends command, asks for a value and returns whatever is returned. All error conditions including timeouts are blatantly ignored.

Clear

```
$instrument->Clear();
```

Sends a clear command to the instrument.

Handle

```
$instr_handle=$instrument->Handle();
```

Returns the VISA handle. You can use this handle with the *Lab::VISA* module.

3.2 Voltage Sources

3.2.1 Lab::Instrument::Source

Base class for voltage source instruments

SYNOPSIS

DESCRIPTION

This class implements a general voltage source, if necessary with several channels. It is meant to be inherited by instrument classes (virtual instruments) that implement real voltage sources (e.g. the Lab::Instrument::Yokogawa7651 class).

The class provides a unified user interface for those virtual voltage sources to support the exchangeability of instruments.

Additionally, this class provides a safety mechanism called `gate_protect` to protect delicate samples. It includes automatic limitations of sweep rates, voltage step sizes, minimal and maximal voltages.

The only user application of this class is to define a voltage source object which represents a single channel of a multi-channel voltage source. Otherwise, you will always have to instantiate classes derived from Lab::Instrument::Source.

CONSTRUCTOR

```
$self=new Lab::Instrument::Source($multisource, $channel);
$self=new Lab::Instrument::Source($multisource, $channel, \%config);
```

This constructor can be used to create a source object which represents channel `$channel` of the multi-channel voltage source `$multisource`. The default configuration of this source is the configuration of `$multisource`; it can be partially or entirely overridden with an additional `\%config` hash.

```
$self=new Lab::Instrument::Source(\%default_config,\%config);
```

This constructor will only be used by instrument drivers that inherit this class, not by the user.

The instrument driver (e.g. Lab::Instrument::KnickS252) has a constructor like this:

```
$knick=new Lab::Instrument::KnickS252({
  GPIB_board       => $board,
  GPIB_address     => $address,

  gate_protect     => $gp,
  [...]
});
```

METHODS

configure

```
$self->configure(\%config);
```

Supported configure options:

fast_set

This parameter controls the return value of the set_voltage function and can be set to 0 (off, default) or 1 (on). For fast_set off, set_voltage first requests the hardware to set the voltage, and then reads out the actually set voltage via get_voltage. The resulting number is returned. For fast_set on, set_voltage requests the hardware to set the voltage and returns without double-check the requested value. This, albeit less secure, may speed up measurements a lot.

gate_protect

Whether to use the automatic sweep speed limitation. Can be set to 0 (off) or 1 (on). If it is turned on, the output voltage will not be changed faster than allowed by the `gp_max_volt_per_second`, `gp_max_volt_per_step` and `gp_max_step_per_second` values. These three parameters overdefine the allowed speed. Only two parameters are necessary. If all three are set, the smallest allowed sweep rate is chosen.

Additionally the maximal and minimal output voltages are limited.

This mechanism is useful to protect sensible samples that are destroyed by abrupt voltage changes. One example is gate electrodes on semiconductor electronics samples, hence the name.

gp_max_volt_per_second

How much the output voltage is allowed to change per second.

gp_max_volt_per_step

How much the output voltage is allowed to change per step.

gp_max_step_per_second

How many steps are allowed per second.

gp_min_volt

The smallest allowed output voltage.

gp_max_volt

The largest allowed output voltage.

qp_equal_level

Voltages with a difference less than this value are considered equal.

set_voltage

```
$new_volt=$self->set_voltage($voltage);
```

Sets the output to `$voltage` (in Volts). If the configure option `gate_protect` is set to a true value, the safety mechanism takes into account the `gp_max_volt_per_step`, `gp_max_volt_per_second` etc. settings, by employing the `sweep_to_voltage` method.

Returns for `fast_set` off the actually set output voltage. This can be different from `$voltage`, due to the `gp_max_volt`, `gp_min_volt` settings. For `fast_set` on, `set_voltage` returns always `$voltage`.

For a multi-channel device, add the channel number as a parameter:

```
$new_volt=$self->set_voltage($voltage,$channel);
```

step_to_voltage

```
$new_volt=$self->step_to_voltage($voltage);
$new_volt=$self->step_to_voltage($voltage,$channel);
```

Makes one safe step in direction to `$voltage`. The output voltage is not changed by more than `gp_max_volt_per_step`. Before the voltage is changed, the methods waits if not enough times has passed since the last voltage change. For step voltage and waiting time calculation, the larger of `gp_max_volt_per_second` or `gp_max_step_per_second` is ignored (see code).

Returns the actually set output voltage. This can be different from `$voltage`, due to the `gp_max_volt`, `gp_min_volt` settings.

sweep_to_voltage

```
$new_volt=$self->sweep_to_voltage($voltage);
$new_volt=$self->sweep_to_voltage($voltage,$channel);
```

This method sweeps the output voltage to the desired value and only returns then. Uses the step_to_voltage method internally, so all discussions of config options from there apply too.

Returns the actually set output voltage. This can be different from `$voltage`, due to the `gp_max_volt`, `gp_min_volt` settings.

get_voltage

```
$new_volt=$self->get_voltage();
$new_volt=$self->get_voltage($channel);
```

Returns the voltage currently set.

3.2.2 Lab::Instrument::Yokogawa7651

Yokogawa 7651 DC source

SYNOPSIS

```
use Lab::Instrument::Yokogawa7651;

my $gate14=new Lab::Instrument::Yokogawa7651(0,11);
$gate14->set_range(5);
$gate14->set_voltage(0.745);
print $gate14->get_voltage();
```

DESCRIPTION

The Lab::Instrument::Yokogawa7651 class implements an interface to the discontinued voltage and current source 7651 by Yokogawa. This class derives from *Lab::Instrument::Source* and provides all functionality described there.

CONSTRUCTORS

new($gpib_board,$gpib_addr)

METHODS

set_voltage($voltage)

get_voltage()

set_range($range)

```
Fixed voltage mode
2    10mV
3    100mV
4    1V
5    10V
6    30V

Fixed current mode
4    1mA
5    10mA
6    100mA
```

get_info() Returns the information provided by the instrument's 'OS' command.

output_on() Sets the output switch to on.

output_off() Sets the output switch to off. The instrument outputs no voltage or current then, no matter what voltage you set.

get_output() Returns the status of the output switch (0 or 1).

initialize()

set_voltage_limit($limit)

set_current_limit($limit)

get_status() Returns a hash with the following keys:

```
CAL_switch
memory_card
calibration_mode
output
unstable
error
execution
setting
```

The value for each key is either 0 or 1, indicating the status of the instrument.

INSTRUMENT SPECIFICATIONS

DC voltage The stability (24h) is the value at 23 +- 1°C. The stability (90days), accuracy (90days) and accuracy (1year) are values at 23 +- 5°C. The temperature coefficient is the value at 5 to 18°C and 28 to 40°C.

```
Range  Maximum      Resolution  Stability 24h    Stability 90d
       Output                   +-(% of setting +-(% of setting
                                +muV)            +muV)
--------------------------------------------------------------
10mV   +-12.0000mV  100nV       0.002 + 3        0.014 + 4
100mV  +-120.000mV  1muV        0.003 + 3        0.014 + 5
1V     +-1.20000V   10muV       0.001 + 10       0.008 + 50
10V    +-12.0000V   100muV      0.001 + 20       0.008 + 100
30V    +-32.000V    1mV         0.001 + 50       0.008 + 200

Range  Accuracy 90d     Accuracy 1yr     Temperature
       +-(% of setting +-(% of setting Coefficient
       +muV)            +muV)            +-(% of setting
                                         +muV)/°C
------------------------------------------------------
10mV   0.018 + 4        0.025 + 5        0.0018 + 0.7
100mV  0.018 + 10       0.025 + 10       0.0018 + 0.7
1V     0.01 + 100       0.016 + 120      0.0009 + 7
```

```
10V     0.01 + 200       0.016 + 240      0.0008 + 10
30V     0.01 + 500       0.016 + 600      0.0008 + 30

Range    Maximum Output                Output Noise
         Output  Resistance            DC to 10Hz  DC to 10kHz
                                       (typical data)
----------------------------------------------------------
10mV     -       approx. 2Ohm          3muVp-p      30muVp-p
100mV    -       approx. 2Ohm          5muVp-p      30muVp-p
1V       +-120mA less than 2mOhm       15muVp-p     60muVp-p
10V      +-120mA less than 2mOhm       50muVp-p     100muVp-p
30V      +-120mA less than 2mOhm       150muVp-p    200muVp-p
```

Common mode rejection: 120dB or more (DC, 50/60Hz). (However, it is 100dB or more in the 30V range.)

DC current

```
Range    Maximum      Resolution  Stability (24 h)    Stability (90 days)
         Output                   +-(% of setting     +-(% of setting
                                  + muA)              + muA)
-----------------------------------------------------------------------
1mA      +-1.20000mA  10nA        0.0015 + 0.03       0.016 + 0.1
10mA     +-12.0000mA  100nA       0.0015 + 0.3        0.016 + 0.5
100mA    +-120.000mA  1muA        0.004  + 3          0.016 + 5

Range    Accuracy (90 days)  Accuracy (1 year)   Temperature
         +-(% of setting     +-(% of setting     Coefficient
         + muA)              + muA)              +-(% of setting
                                                 + muA)/°C
-----    ---------------------------------------------------
1mA      0.02 + 0.1          0.03 + 0.1          0.0015 + 0.01
10mA     0.02 + 0.5          0.03 + 0.5          0.0015 + 0.1
100mA    0.02 + 5            0.03 + 5            0.002  + 1

Range   Maximum     Output                    Output Noise
        Output      Resistance           DC to 10Hz  DC to 10kHz
                                                     (typical data)
-------------------------------------------------------------------
1mA     +-30 V      more than 100MOhm    0.02muAp-p  0.1muAp-p
10mA    +-30 V      more than 100MOhm    0.2muAp-p   0.3muAp-p
100mA   +-30 V      more than 10MOhm     2muAp-p     3muAp-p
```

Common mode rejection: 100nA/V or more (DC, 50/60Hz).

3.2.3 Lab::Instrument::KnickS252

Knick S 252 DC source

SYNOPSIS

```
use Lab::Instrument::KnickS252;

my $gate14=new Lab::Instrument::KnickS252(0,11);
$gate14->set_range(5);
$gate14->set_voltage(0.745);
print $gate14->get_voltage();
```

DESCRIPTION

The Lab::Instrument::KnickS252 class implements an interface to the Knick S 252 dc calibrator. This class derives from *Lab::Instrument::Source* and provides all functionality described there.

CONSTRUCTOR

```
$knick=new Lab::Instrument::KnickS252($gpib_board,$gpib_addr);
# Or any other type of construction supported by Lab::Instrument.
```

METHODS

set_voltage

```
$knick->set_voltage($voltage);
```

get_voltage

```
$voltage=$knick->get_voltage();
```

set_range

```
$knick->set_range($range);
# $range is 5 or 20
#  5  is 5V
# 20  is 20V
```

get_range

```
$range=$knick->get_range();
# $range is 5 or 20
#  5  is 5V
# 20  is 20V
```

3.2.4 Lab::Instrument::IOtech488

IOtech DAC488HR four channel voltage source

SYNOPSIS

```
use Lab::Instrument::IOtech488;

my $gates=new Lab::Instrument::IOtech488({
    GPIB_board   => 0,
    GPIB_address => 11,
});
$gates->set_range(1,6);  # 2 volt unipolar for channel 1

$gates->set_voltage(1,0.745);

print $gates->get_voltage(1);

my $plunger=new Lab::Instrument::Source($gates, 3);

$plunger->set_voltage(-0.5);
```

DESCRIPTION

The Lab::Instrument::IOtech488 class implements an interface to the IOtech DAC488HR four-channel voltage source. This class derives from *Lab::Instrument::Source* and provides all functionality described there.

CONSTRUCTORS

new({})

```
my $gates=new Lab::Instrument::IOtech488({
    GPIB_board   => 0,
    GPIB_address => 11,
});
```

METHODS

set_voltage($voltage,$channel)

get_voltage($channel)

set_range($range,$channel)

```
# Ranges
# 1 -  1 volt bipolar
# 2 -  2 volt bipolar
# 3 -  5 volt bipolar
# 4 - 10 volt bipolar
# 5 -  1 volt unipolar
# 6 -  2 volt unipolar
# 7 -  5 volt unipolar
# 8 - 10 volt unipolar
```

A change of range will set the output to zero!!!

get_info() Returns the information provided by the instrument's 'U9' command.

reset()

3.2.5 Lab::Instrument::SRS_SIM928

SRS SIM928 voltage source module for SIM900 mainframe

SYNOPSIS

```
use Lab::Instrument::SRS_SIM928;

my $gates=new Lab::Instrument::SRS_SIM928(0,11);
$gates->set_voltage(0.745,1);
print $gate14->get_voltage(1);

my $plunger=new Lab::Instrument::Source($gates, 3);

$plunger->set_voltage(-0.5);
```

DESCRIPTION

The Lab::Instrument::SRS_SIM928 class implements an interface to the SIM928 voltage source modules. This class derives from *Lab::Instrument::Source* and provides all functionality described there.

CONSTRUCTORS

new($gpib_board,$gpib_addr)

METHODS

set_voltage($voltage,$channel)

get_voltage($channel)

get_battery_status($channel) Provides information on the battery in the module `$channel`.

clear($channel) Clears the error status.

reset($channel) Resets the module. Voltage is set to zero and output is turned OFF.

id() Returns the information provided by the instrument's '*IDN?' command.

3.2.6 Lab::Instrument::Dummysource

Dummy voltage source

DESCRIPTION

The Lab::Instrument::Dummysource class implements a dummy voltage source that does nothing but fullfill testing purposes.

Only developers will ever make use of this class.

3.3 Multimeters

3.3.1 Lab::Instrument::HP34401A

HP/Agilent 34401A digital multimeter

SYNOPSIS

```
use Lab::Instrument::HP34401A;

my $hp=new Lab::Instrument::HP34401A(0,22);
print $hp->read_voltage_dc(10,0.00001);
```

DESCRIPTION

The Lab::Instrument::HP34401A class implements an interface to the 34401A digital multimeter by Agilent (formerly HP). This module can also be used to address the newer 34410A and 34411A multimeters, but doesn't include new functions. Use the Lab::Instrument::HP34411A class for full functionality.

CONSTRUCTOR

```
my $hp=new(\%options);
```

METHODS

read_voltage_dc

```
$datum=$hp->read_voltage_dc($range,$resolution);
```

Preset and make a dc voltage measurement with the specified range and resolution.

$range

Range is given in terms of volts and can be `[0.1|1|10|100|1000|MIN|MAX|DEF]`. `DEF` is default.

$resolution

Resolution is given in terms of `$range` or `[MIN|MAX|DEF]`. `$resolution=0.0001` means 4 1/2 digits for example. The best resolution is 100nV: `$range=0.1`; `$resolution=0.000001`.

read_voltage_ac

```
$datum=$hp->read_voltage_ac($range,$resolution);
```

Preset and make an ac voltage measurement with the specified range and resolution. For ac measurements, resolution is actually fixed at 6 1/2 digits. The resolution parameter only affects the front-panel display.

read_current_dc

```
$datum=$hp->read_current_dc($range,$resolution);
```

Preset and make a dc current measurement with the specified range and resolution.

read_current_ac

```
$datum=$hp->read_current_ac($range,$resolution);
```

Preset and make an ac current measurement with the specified range and resolution. For ac measurements, resolution is actually fixed at 6 1/2 digits. The resolution parameter only affects the front-panel display.

read_resistance

```
$datum=$hp->read_resistance($range,$resolution);
```

Preset and measure resistance with specified range and resolution.

config_voltage

```
$inttime=$hp->config_voltage($digits,$range,$count);
```

Configures device for measurement with specified number of digits (4 to 6), voltage range and number of data points. Afterwards, data can be taken by triggering the multimeter, resulting in faster measurements than using read_voltage_xx. Returns string with integration time resulting from number of digits.

read_with_trigger_voltage_dc

```
@array = $hp->read_with_trigger_voltage_dc()
```

Take data points as configured with config_voltage(). returns an array.

display_on

```
$hp->display_on();
```

Turn the front-panel display on.

display_off

```
$hp->display_off();
```

Turn the front-panel display off.

display_text

```
$hp->display_text($text);
print $hp->display_text();
```

Display a message on the front panel. The multimeter will display up to 12 characters in a message; any additional characters are truncated. Without parameter the displayed message is returned.

display_clear

```
$hp->display_clear();
```

Clear the message displayed on the front panel.

scroll_message

```
$hp->scroll_message($message);
```

Scrolls the message `$message` on the display of the HP.

beep

```
$hp->beep();
```

Issue a single beep immediately.

get_error

```
($err_num,$err_msg)=$hp->get_error();
```

Query the multimeter's error queue. Up to 20 errors can be stored in the queue. Errors are retrieved in first-in-first out (FIFO) order.

reset

```
$hp->reset();
```

Reset the multimeter to its power-on configuration.

id

```
$id=$hp->id();
```

Returns the instruments ID string.

3.3.2 Lab::Instrument::HP34411A

HP/Agilent 34410A or 34411A digital multimeter

SYNOPSIS

```
use Lab::Instrument::HP34411A;

my $hp=new Lab::Instrument::HP34411A(0,22);
print $hp->read_voltage_dc(10,0.00001);
```

DESCRIPTION

The Lab::Instrument::HP34411A class implements an interface to the 34410A and 34411A digital multimeters by Agilent (formerly HP). Note that the module Lab::Instrument::HP34401A still works for those newer multimeter models.

CONSTRUCTOR

```
my $hp=new(\%options);
```

METHODS

read_voltage_dc

```
$datum=$hp->read_voltage_dc($range,$resolution);
```

Preset and make a dc voltage measurement with the specified range and resolution.

$range

Range is given in terms of volts and can be `[0.1|1|10|100|1000|MIN|MAX|DEF]`. `DEF` is default.

$resolution

Resolution is given in terms of `$range` or `[MIN|MAX|DEF]`. `$resolution=0.0001` means 4 1/2 digits for example. The best resolution is 100nV: `$range=0.1`; `$resolution=0.000001`.

read_voltage_ac

```
$datum=$hp->read_voltage_ac($range,$resolution);
```

Preset and make an ac voltage measurement with the specified range and resolution. For ac measurements, resolution is actually fixed at 6½ digits. The resolution parameter only affects the front-panel display.

read_current_dc

```
$datum=$hp->read_current_dc($range,$resolution);
```

Preset and make a dc current measurement with the specified range and resolution.

read_current_ac

```
$datum=$hp->read_current_ac($range,$resolution);
```

Preset and make an ac current measurement with the specified range and resolution. For ac measurements, resolution is actually fixed at 6½ digits. The resolution parameter only affects the front-panel display.

read_resistance

```
$datum=$hp->read_resistance($range,$resolution);
```

Preset and measure resistance with specified range and resolution.

config_voltage

```
$hp->config_voltage($inttime,$range,$count);
```

Configures device for measurement with specified integration time, voltage range and number of data points (up to 1 million). Afterwards, data can be taken by triggering the multimeter, resulting in faster measurements than using read_voltage_dc, especially when using $count >> 1.

config_voltage_plc

```
$hp->config_voltage_plc($plc,$range,$count);
```

Same as config_voltage, but here the number of power line cycles ($plc) is given instead of an integration time.

read_with_trigger_voltage_dc

```
@array = $hp->read_with_trigger_voltage_dc()
```

Take data points as configured with config_voltage(). Returns an array.

display_on

```
$hp->display_on();
```

Turn the front-panel display on.

display_off

```
$hp->display_off();
```

Turn the front-panel display off.

display_text

```
$hp->display_text($text);
print $hp->display_text();
```

Display a message on the front panel. The multimeter will display up to 12 characters in a message; any additional characters are truncated. Without parameter the displayed message is returned.

display_clear

```
$hp->display_clear();
```

Clear the message displayed on the front panel.

scroll_message

```
$hp->scroll_message($message);
```

Scrolls the message `$message` on the display of the HP.

beep

```
$hp->beep();
```

Issue a single beep immediately.

get_error

```
($err_num,$err_msg)=$hp->get_error();
```

Query the multimeter's error queue. Up to 20 errors can be stored in the queue. Errors are retrieved in first-in-first out (FIFO) order.

reset

```
$hp->reset();
```

Reset the multimeter to its power-on configuration.

id

```
$id=$hp->id();
```

Returns the instruments ID string.

3.3.3 Lab::Instrument::HP34420A

HP/Agilent 34420A nanovolt/microohm meter

3.3.4 Lab::Instrument::HP34970A

HP/Agilent 34970A Data Acquisition Switch Unit

SYNOPSIS

DESCRIPTION

CONSTRUCTOR

```
my $hp=new(\%options);
```

METHODS

read_voltage_dc

```
$datum=$hp->read_voltage_dc($range,$resolution,@scan_list);
```

Preset and make a dc voltage measurement with the specified range and resolution.

$range

Range is given in terms of volts and can be [0.1|1|10|100|1000|MIN|MAX|DEF]. DEF is default.

$resolution

Resolution is given in terms of $range or [MIN|MAX|DEF]. $resolution=0.0001 means 4 1/2 digits for example. The best resolution is 100nV: $range=0.1; $resolution=0.000001.

@scan_list

A list of channels to scan. See the instrument manual.

conf_monitor

```
$hp->conf_monitor(@channels);
```

read_monitor

```
@channels=$hp->read_monitor();
```

display_on

```
$hp->display_on();
```

Turn the front-panel display on.

display_off

```
$hp->display_off();
```

Turn the front-panel display off.

display_text

```
$hp->display_text($text);
print $hp->display_text();
```

Display a message on the front panel. The multimeter will display up to 12 characters in a message; any additional characters are truncated. Without parameter the displayed message is returned.

display_clear

```
$hp->display_clear();
```

Clear the message displayed on the front panel.

beep

```
$hp->beep();
```

Issue a single beep immediately.

get_error

```
($err_num,$err_msg)=$hp->get_error();
```

Query the multimeter's error queue. Up to 20 errors can be stored in the queue. Errors are retrieved in first-in-first out (FIFO) order.

reset

```
$hp->reset();
```

Reset the multimeter to its power-on configuration.

scroll_message

```
$hp->scroll_message();
```

3.3.5 Lab::Instrument::HP3458A

Agilent 3458A Multimeter

SYNOPSIS

```
use Lab::Instrument::HP3458A;

my $dmm=new Lab::Instrument::HP3458A({
    GPIB_board    => 0,
    GPIB_address => 11,
});
print $dmm->read_voltage_dc();
```

DESCRIPTION

The Lab::Instrument::HP3458A class implements an interface to the Agilent / HP 3458A digital multimeter. The Agilent 3458A Multimeter, recognized the world over as the standard in high performance DMMs, provides both speed and accuracy in the R&D lab, on the production test floor, and in the calibration lab.

CONSTRUCTOR

```
my $hp=new(\%options);
```

METHODS

read_voltage_dc

```
$datum=$hp->read_voltage_dc();
```

Make a dc voltage measurement.

display_on

```
$hp->display_on();
```

Turn the front-panel display on.

display_off

```
$hp->display_off();
```

Turn the front-panel display off.

display_text

```
$hp->display_text($text);
print $hp->display_text();
```

Display a message on the front panel. The multimeter will display up to 12 characters in a message; any additional characters are truncated.

display_clear

```
$hp->display_clear();
```

Clear the message displayed on the front panel.

beep

```
$hp->beep();
```

Issue a single beep immediately.

get_error

```
($err_num,$err_msg)=$hp->get_error();
```

Query the multimeter's error queue. Up to 20 errors can be stored in the queue. Errors are retrieved in first-in-first out (FIFO) order.

set_nlpc

```
$hp->set_nlcp($number);
```

Sets the integration time in power line cycles.

reset

```
$hp->reset();
```

Reset the multimeter to its power-on configuration.

preset

```
$hp->preset($config);
```

Choose one of several configuration presets (0: fast, 1: norm, 2: DIG).

selftest

```
$hp->selftest();
```

Starts the internal self-test routine.

autocalibration

```
$hp->autocalibration();
```

Starts the internal autocalibration. Warning... this procedure takes 11 minutes!

3.4 Amplifiers

3.4.1 Lab::Instrument::SR830

Stanford Research SR830 Lock-In Amplifier

SYNOPSIS

```
use Lab::Instrument::SR830;

my $sr830=new Lab::Instrument::SR830(0,10);

($x,$y) = $sr780->read_xy();
($r,$phi) = $sr780->read_rphi();
```

DESCRIPTION

The Lab::Instrument::SR830 class implements an interface to the Stanford Research SR830 Lock-In Amplifier.

CONSTRUCTOR

```
$sr830=new Lab::Instrument::SR830($board,$gpib);
```

METHODS

read_xy

```
($x,$y)= $sr830->read_xy();
```

Reads channels x and y simultaneously; returns an array.

read_rphi

```
($r,$phi)= $sr830->read_rphi();
```

Reads amplitude and phase simultaneously; returns an array.

set_sens

```
$string=$sr830->set_sens(1E-7);
```

Sets sensitivity (value given in V); possible values are: 2 nV, 5 nV, 10 nV, 20 nV, 50 nV, 100 nV, ..., 100 mV, 200 mV, 500 mV, 1V If the argument is not in this list, the next higher value will be chosen.

Returns the value of the sensitivity that was actually set as string.

get_sens

```
$sens = $sr830->get_sens();
```

Returns sensitivity (as string, e.g. "50 nV").

set_tc

```
$string=$sr830->set_tc(1E-3);
```

Sets time constant (value given in seconds); possible values are: 10 us, 30us, 100 us, 300 us, ..., 10000 s, 30000 s If the argument is not in this list, the next higher value will be chosen.

Returns the value of the time constant that was actually set as string.

get_tc

```
$tc = $sr830->get_tc();
```

Returns the time constant (as string, e.g. "3 ms").

set_frequency

```
$sr830->set_frequency(334);
```

Sets reference frequency; value given in Hz. Values between 0.001 Hz and 102 kHz can be set.

get_frequency

```
$freq=$sr830->get_frequency();
```

Returns reference frequency (value given in Hz).

set_amplitude

```
$sr830->set_amplitude(0.005);
```

Sets output amplitude to the value given (in V); values between 4 mV and 5 V are possible.

get_amplitude

```
$ampl=$sr830->get_amplitude();
```

Returns amplitude of the sine output in V.

id

```
$id=$sr830->id();
```

Returns the instruments ID string.

3.5 High Frequency

3.5.1 Lab::Instrument::SR780

Stanford Research SR780 Network Signal Analyzer

SYNOPSIS

```
use Lab::Instrument::SR780;

my $sr780=new Lab::Instrument::SR780(0,10);

$sr780->send_commands(
    #INPUT CH1
    'ISRC 0',#        Source          Analog
    'I1MD 1',#        Mode            A-B
    'I1GD 1',#        Ground          Ground
    'I1CP 1',#        Coupling        AC
    'I1RG 1',#        Range           Auto
    'I1AF 1',#        AA Filter       On
    'I1AW 0',#        A-Wt Filter     Off
    'I1AR 0',#        Auto Range      Normal
    'IAOM 1',#        Auto Offset     On
    'EU1M 0',#        EU              Off
    'EU1L 1',#        EU Label        m/s
    'EU1V 1',#        EU/Volt         1 EU/V
    'EU1U EU',#       User Label      EU

    #Measure Display A
    'DISP 0,1',#      Display         Live
    'DFMT 0',#        Display         Single
    'ACTD 0',#        Active Display  0
    'MGRP 0,0',#      Measurement     FFT ch1
    'MEAS 0,0',#      Measurement     FFT1
    'VIEW 0,0',#      View            Log Mag
    'UNIT 0,1',#      Units           Vrms
    'PSDU 0,1',#      PSD             On
    'FBAS 0,1',#      Base Freq       102.4 kHz
    'FSPN 0,102400',#Span              102.4 kHz
    'FSTR 0,0',#      Start Freq      0 Hz
    'FLIN 0,3',#      Lines           800
    'FWIN 0,0',#      Window          Uniform
    'FWFL 0,100',#    Force           100%
    'FWTC 0,50',#     Expo            50%

    #Average Display A
    'FAVG 0,1',#      Average         On
    'FAVM 0,1',#      Mode            RMS
    'FAVT 0,1',#      Type            Exponential
    'FAVN 0,1000',#   Number          1000
    'FOVL 0,100',#    Time Incr       100.00%
    'FREJ 0,0',#      Reject          Off
    'PAVO 0,0',#      Preview         Off
    'PAVT 0,2',#      Prv Time        2 s
```

```
        #Display  Display  A
        'XAXS␣0,1',#     X  Axis          Log
        'GRID␣0,1',#     Grid             On
        'GDIV␣0,1',#     Grid  Div        10
        'TRRC␣0,0',#     Xdcr  Convert    Acceleration
        'DBMR␣50',#      dBm  Ref         50
        'PHSL␣0,0',#     Phase  Suppress  0.0000e+000
        'DDXW␣0,0.5',#   d/dx  Window     0.5
    ));

    $sr780->start_measurement();

    while (1000 != $sr780->average_status('A')) { sleep(1) }

    $sr780->pause_measurement();

    my @data=$sr780->read_data_pairs('A');
    for (@data) {
        print (join "\t",@$_),"\n";
    }

    $sr780->play(3);
```

DESCRIPTION

The Lab::Instrument::SR780 class implements an interface to the Stanford Research SR780 Network Signal Analyzer.

CONSTRUCTOR

```
$sr780=new Lab::Instrument::SR780($board,$gpib);
```

METHODS

start_measurement

```
$sr780->start_measurement();
```

Starts the measurement. Any average in progress is reset and started over. If the measurement is paused, starts the measurement over. This method is the same as pressing the [Start/Reset] key.

pause_measurement

```
$sr780->pause_measurement();
```

If the measurement is already in progress, pauses the measurement. If the measurement is paused, it has no effect. This method is similar to pressing the [Pause/Cont] key.

continue_measurement

```
$sr780->continue_measurement();
```

If the measurement is paused, continues the measurement. If the measurement is running, it has no effect. This method is similar to pressing the [Pause/Cont] key.

read_display_data

```
@data=$sr780->read_display_data($display);
```

Returns a list of data from the given display (`$display` is either 'A' or 'B').

read_data_pairs

```
@pairs=$sr780->read_data_pairs($display);
```

Reads the data of display `$display`. The data is returned as a list of references to arrays with a frequency,data pair.

average_status

```
$num_avg=$sr780->average_status($display);
```

Returns the number of already completed averages for display `$display`.

send_commands

```
$sr780->send_commands(@commands);
```

Sends a list of commands to the instrument. Useful for mass configuration.

id

```
$id=$sr780->id();
```

Returns the instruments ID string.

play

```
$sr780->play($sound);
```

Plays the predefinded sound `$sound` (between 0 and 6).

tone

```
$sr780->tone($tone,$duration);
```

Plays the note `$tone` (between 0 and 66) on the internal speaker. The tone is played for the time `$duration`, which is in units of 5ms.

play_song

```
$sr780->play_song();
```

Plays the song *Alle meine Entchen* on the internal speaker of the SR780. While this might not be especially useful for your measurements, it will guaranty you *ubergeek* status among the colleagues in your lab.

3.5.2 Lab::Instrument::SR620

Stanford Research 620 Frequency Counter

3.5.3 Lab::Instrument::HP8360

HP 8360 B-Series Swept Signal Generator

SYNOPSIS

DESCRIPTION

CONSTRUCTOR

METHODS

3.5.4 Lab::Instrument::Agilent81134A

Agilent 81134A pulse generator

SYNOPSIS

```
use Lab::Instrument::Agilent81134A;

my $a=new Lab::Instrument::Agilent81134A(0,22);
```

DESCRIPTION

The Lab::Instrument::Agilent81134A class will provide an interface to the Agilent 81134A pulse generator. Right now, only two small example methods are provided.

CONSTRUCTOR

```
my $a=new(\%options);
```

METHODS

To be written.

3.6 Cryostat Handling

3.6.1 Lab::Instrument::IsoBus

Oxford Instruments IsoBus device

SYNOPSIS

```
use Lab::Instrument::IsoBus;

my $isobus=new Lab::Instrument::ILM(0,1);
my $ilm=new Lab::Instrument::ILM($isobus,$addr);
```

DESCRIPTION

The Lab::Instrument::IsoBus class implements an interface to the Oxford Instruments IsoBus. The IsoBus is treated as a VISA device, and can thus be attached directly at a serial port or at a GPIB gateway device. The corresponding VISA resource has to be specified at initialization.

Later, IsoBus devices attached to this IsoBus can be instantiated by passing the IsoBus as first constructor argument.

CONSTRUCTOR

```
my $isobus=new Lab::Instrument::IsoBus($gpibadapter,$gpibaddr);
```

Instantiates a new IsoBus object. All argument variants valid for the `Lab::Instrument` constructor can be used. This way, an IsoBus can be attached to a GPIB gateway device or directly to a serial port.

METHODS

IsoBus_Write

```
$write_cnt=$isobus->IsoBus_Write($addr,$command);
```

Sends `$command` to the device attached to this IsoBus with IsoBus address `$addr`. The number of bytes actually written is returned.

IsoBus_Read

```
$result=$isobus->IsoBus_Read($addr,$length);
```

Reads at most `$length` bytes from the device attached to this IsoBus with IsoBus address `$addr`. The resulting string is returned.

IsoBus_valid

```
$is_an_isobus=$isobus->IsoBus_valid();
```

Returns 1.

3.6.2 Lab::Instrument::ILM

Oxford Instruments ILM Intelligent level meter

SYNOPSIS

```
use Lab::Instrument::ILM;

my $ilm=new Lab::Instrument::ILM($isobus,3);
print $ilm->get_level();
```

DESCRIPTION

The Lab::Instrument::ILM class implements an interface to the Oxford Instruments ILM helium level meter (tested with the ILM210).

CONSTRUCTOR

```
my $ilm=new Lab::Instrument::ILM($isobus,$addr);
```

Instantiates a new ILM object, for example attached to the IsoBus device (of type `Lab::Instrument::IsoBus`) `$IsoBus`, with IsoBus address `$addr`. All constructor forms of `Lab::Instrument` are available.

METHODS

get_level

```
$perc=$ilm->get_level();
$perc=$ilm->get_level(1);
```

Reads out the current helium level in percent. Note that this command does NOT trigger a measurement, but only reads out the last value measured by the ILM. This means that in slow mode values may remain constant for several minutes.

As optional parameter a channel number can be provided. This defaults to 1.

3.6.3 Lab::Instrument::ITC

Oxford Instruments ITC Intelligent Temperature Control

SYNOPSIS

```
use Lab::Instrument::ITC;

my $irc=new Lab::Instrument::ILM($gpibadaptor,3);
```

DESCRIPTION

The Lab::Instrument::ITC class implements an interface to the Oxford Instruments ITC intelligent temperature controller (tested with the ITC503). This driver is still very much work in progress and also lacks documentation.

3.6.4 Lab::Instrument::Lakeshore336

Lakeshore 336 Temperature controller
UNGETESTET

SYNOPSIS

```
use Lab::Instrument::Lakeshore336;

my $lake=new Lab::Instrument::Lakeshore336(0,10);

$temp = $lake->read_t();
$r = $lake->read_r();
```

DESCRIPTION

The Lab::Instrument::Lakeshore336 class implements an interface to the Lakeshore 336 AC Resistance Bridge.

CONSTRUCTOR

```
$lake=new Lab::Instrument::Lakeshore370($board,$gpib);
```

METHODS

read_t

```
$t = $lake->read_t();
```

Reads temperature in Kelvin (only possible if temperature curve is available, otherwise returns zero).

read_r

```
$r = $lake->read_r();
```

Reads resistance in ohms.

id

```
$id=$sr780->id();
```

Returns the instruments ID string.

3.6.5 Lab::Instrument::Lakeshore370

Lakeshore 370 AC Resistance Bridge

SYNOPSIS

```
use Lab::Instrument::Lakeshore370;

my $lake=new Lab::Instrument::Lakeshore370(0,10);

$temp = $lake->read_t();
$r = $lake->read_r();
```

DESCRIPTION

The Lab::Instrument::Lakeshore370 class implements an interface to the Lakeshore 370 AC Resistance Bridge.

CONSTRUCTOR

```
$lake=new Lab::Instrument::Lakeshore370($board,$gpib);
```

METHODS

read_t

```
$t = $lake->read_t();
```

Reads temperature in Kelvin (only possible if temperature curve is available, otherwise returns zero).

read_r

```
$r = $lake->read_r();
```

Reads resistance in ohms.

set_channel

```
$lake->set_channel(4);
```

Sets channel to scan (with autoscan = off); returns channel the bridge was set to.

id

```
$id=$lake->id();
```

Returns the instruments ID string.

3.6.6 Lab::Instrument::TRMC2

ABB TRMC2 temperature controller

SYNOPSIS

```
use Lab::Instrument::TRMC2;
```

DESCRIPTION

The Lab::Instrument::ILM class implements an interface to the ABB TRMC2 temperature controller. The driver works, but documentation is lacking.

CONSTRUCTOR

```
my $trmc=...
```

4 The Lab::Tools package

4.1 Lab::Data::Writer

Write data to disk

SYNOPSIS

```
use Lab::Data::Writer;

my $writer=new Lab::Data::Writer($filename,$config);

$writer->log_comment("This is my test log");

my $num=$writer->log_start_block();
$writer->log_line(1,2,3);
```

DESCRIPTION

This module can be used to log data to a file, comfortably.

CONSTRUCTOR

new

```
$writer=new Lab::Data::Writer($filename,$config);
```

See *configure* below for available configuration options.

METHODS

configure

```
$writer->configure(\%config);
```

Available options and default values are

```
output_data_ext      => "dat",
output_meta_ext      => "meta",

output_col_sep       => "\t",
output_line_sep      => "\n",
output_block_sep     => "\n",
output_comment_char  => "# ",
```

get_filename

```
($filename,$filepath)=$writer->get_filename()
```

log_comment

```
$writer->log_comment($comment);
```

Writes a comment to the file.

log_line

```
$writer->log_line(@data);
```

Writes a line of data to the file.

log_start_block

```
$num=$writer->log_start_block();
```

Starts a new data block.

import_gpplus(%opts)

Imports GPplus TSK-files. Valid parameters are

```
filename  => 'path/to/one/of/the/tsk-files',
newname   => 'path/to/new/directory/newname',
archive   => '[copy|move]'
```

The path `path/to/new/directory/` must exist, while `newname` shall not exist there.

4.2 Lab::Measurement

Log, describe and plot data on the fly

SYNOPSIS

```
use Lab::Measurement;

my $measurement=new Lab::Measurement(
    sample              => $sample,
    title               => $title,
    filename_base       => 'qpc_pinch_off',
    description         => $comment,

    live_plot           => 'QPC current',

    columns             => [
        {
            'unit'              => 'V',
            'label'             => 'Gate voltage',
            'description'       => 'Applied to gates via low path filter
               .',
        },
        {
            'unit'              => 'V',
            'label'             => 'Amplifier output',
            'description'       => "Voltage output by current amplifier 
               set to $amp.",
        }
    ],
    axes                => [
        {
            'unit'              => 'V',
            'expression'        => '$C0',
            'label'             => 'Gate voltage',
            'min'               => ($start_voltage < $end_voltage) ?
               $start_voltage : $end_voltage,
            'max'               => ($start_voltage < $end_voltage) ?
               $end_voltage : $start_voltage,
            'description'       => 'Applied to gates via low path filter
               .',
        },
        {
            'unit'              => 'A',
            'expression'        => "abs(\$C1)*$amp",
            'label'             => 'QPC current',
            'description'       => 'Current through QPC',
        },
        {
            'unit'              => '2e^2/h',
            'expression'        => "(\$A1/$v_sd)/$g0)",
            'label'             => "Total conductance",
        },
```

```
            {
                'unit'              => '2e^2/h',
                'expression'        => "(1/(1/abs(\$C1)-1/$U_Kontakt)) * (
                   $amp/($v_sd*$g0))",
                'label'             => "QPC conductance",
                'min'               => -0.1,
                'max'               => 5
            },

        ],
        plots               => {
            'QPC current'      => {
                'type'              => 'line',
                'xaxis'             => 0,
                'yaxis'             => 1,
                'grid'              => 'xtics ytics',
            },
            'QPC conductance'=> {
                'type'              => 'line',
                'xaxis'             => 0,
                'yaxis'             => 3,
                'grid'              => 'ytics',
            }
        },
);

$measurement->start_block();

my $stepsign=$step/abs($step);
for (my $volt=$start_voltage;$stepsign*$volt<=$stepsign*$end_voltage;
    $volt+=$step) {
    $knick->set_voltage($volt);
    usleep(500000);
    my $meas=$hp->read_voltage_dc(10,0.0001);
    $measurement->log_line($volt,$meas);
}

my $meta=$measurement->finish_measurement();
```

DESCRIPTION

This module simplifies the task of running a measurement, writing the data to disk and keeping track of necessary meta information that usually later you don't find in your lab book anymore.

If your measurements don't come out nice, it's not because you were using the wrong software.

CONSTRUCTORS

new

```
$measurement=new Lab::Measurement(%config);
```

where %config can contain

```
sample        => '',  # see Meta
title         => '',  # single line
filename      => '',
filename_base => '',  # for auto_naming
description   => '',  # multi line

columns       => [],
axes          => [],
plots         => [],  # See Meta

live_plot     => '',  # Name of plot that is to be plotted live
live_refresh  => '',
live_latest   => '',

writer_config => {},  # Configuration options for Lab::Data::Writer
```

METHODS

start_block

```
$block_num=$measurement->start_block($label);
```

log_line

```
$measurement->log_line(@data);
```

finish_measurement

```
$meta=$measurement->finish_measurement();
```

now_string

```
$now=$measurement->now_string();
```

log($datum,$column,$description)

magic log. deprecated.

4.3 Lab::Data::Meta

Meta data for datasets

SYNOPSIS

```
use Lab::Data::Meta;

my $meta2=new Lab::Data::Meta({
    dataset_title   => "testtest",
    column          => [
        {label   =>'hallo'},
        {label   =>'selber hallo',
         unit    =>'mV'},
    ],
    axis            => [
        {
            unit        => 's',
            description => 'the time',
        },
        {
            unit        => 'eV',
            description => 'kinetic energy',
        },
    ],
});
```

DESCRIPTION

This module maintains meta information on a dataset. It's build on top of Lab::Data::XMLtree.

CONSTRUCTOR

new

```
$meta=new Lab::Data::Meta(\%metainfo);
```

Currently, `Lab::Data::Meta` supports the following bits of meta information:

```
data_complete                => ['SCALAR'],  # boolean

dataset_title                => ['SCALAR'],
dataset_description          => ['SCALAR'],  # multiline
sample                       => ['SCALAR'],
data_file                    => ['SCALAR'],  # relativ zur
    descriptiondatei
```

```
    block                   => [
        'ARRAY',
        'id',
        {
            original_filename   => ['SCALAR'],  # nur von GPplus-Import
                unterstützt
            timestamp           => ['SCALAR'],  # Format %Y/%m/%d-%H:%M
                :%S
            description         => ['SCALAR'],
            label               => ['SCALAR'],
        }
    ],
    column                  => [
        'ARRAY',
        'id',
        {
            unit                => ['SCALAR'],
            label               => ['SCALAR'],  # evtl. weg
            description         => ['SCALAR'],  # evtl. weg
            min                 => ['SCALAR'],  # unnütz, aber von
                GPplus-Import unterstützt
            max                 => ['SCALAR'],  # dito
        }
    ],
    axis                    => [
        'ARRAY',
        'id',
        {
            label               => ['SCALAR'],
            unit                => ['SCALAR'],
            expression          => ['SCALAR'],
            min                 => ['SCALAR'],
            max                 => ['SCALAR'],
            description         => ['SCALAR'],  # evtl. weg
        }
    ],
    plot                    => [
        'HASH',
        'name',
        {
            type                => ['SCALAR'],  # line, pm3d
            xaxis               => ['SCALAR'],
            xformat             => ['SCALAR'],
            yaxis               => ['SCALAR'],
            yformat             => ['SCALAR'],
            zaxis               => ['SCALAR'],
            zformat             => ['SCALAR'],
            cbaxis              => ['SCALAR'],
            cbformat            => ['SCALAR'],
            logscale            => ['SCALAR'],  # z.b: 'x' oder 'yzxcb'
            time                => ['SCALAR'],  # ??? (was: wie oben (
                anders als in GnuPlot) (Achsen müssen %s-Format haben))
            grid                => ['SCALAR'],  # z.B. 'ytics' oder '
                xtics ytics'
```

```
            palette             => ['SCALAR'],
            label               => [
                'ARRAY',
                'id',
                {
                    text        => ['SCALAR'],
                    x           => ['SCALAR'],
                    y           => ['SCALAR'],
                }
            ],
        }
    ],
    constant                => [
        'ARRAY',
        'id',
        {
            name                => ['SCALAR'],
            value               => ['SCALAR'],
        }
    ],
```

new_from_file

```
$meta=new_from_file Lab::Data::Meta($filename);
```

METHODS

save

```
$meta->save($filename);
```

get_abs_path

```
my $path=get_abs_path();
```

4.4 Lab::Data::XMLtree

Handle and store XML and perl data structures with precise declaration.

SYNOPSIS

```
use Lab::Data::XMLtree;

my $data_declaration = {
    info                     => [#              type B
        'SCALAR',
        {
            basename     => ['PSCALAR'],#       type A
            title        => ['SCALAR'],#        type A
            place        => ['SCALAR']#         type A
        }
    ],
    column                   => [#              type K
        'ARRAY',
        'id',
        {
            # PSCALAR means that this element will not
            # be saved. Does not work for YAML yet.
            min          => ['PSCALAR'],#       type A
            max          => ['PSCALAR'],#       type A
            description  => ['SCALAR']#         type A
        }
    ],
    axis                     => [#              type F
        'HASH',
        'label',
        {
            unit         => ['SCALAR'],#        type A
            logscale     => ['SCALAR'],#        type A
            description  => ['SCALAR']#         type A
        }
    ]
};
#create Lab::Data::XMLtree object from file
$data=Lab::Data::XMLtree->read_xml($data_declaration,'filename.xml')
    ;

#the autoloader
# get
print $data->info_title;
# get with $id
print $data->column_description($id);
# set with $key and $value
$data->axis_description($label,'descriptiontext');

#save data as YAML
$data->save_yaml('filename.yaml');
```

DESCRIPTION

`Lab::Data::XMLtree` will take you to similar spots as XML::Simple does, but in a bigger bus and with fewer wild animals.

That's not a bad thing. You get more control of the data transformation processes and you get some extra functionality.

DATA DECLARATION

Lab::Data::XMLtree uses a data declaration, that describes, what the perl data structure looks like, and how this data structure is converted to XML.

CONSTRUCTORS

new($declaration,[$data])

Create a new Lab::Data::XMLtree. $data must be hashref and should match the declaration. Returns Lab::XMLtree object.

The first two elements define the folding behaviour.

SCALAR|PSCALAR

Element occurs zero or one time. No folding necessary.

Examples:

```
$data->{dataset_title}='content';
```

ARRAY|PARRAY

Element occurs zero or more times. Folding will be done using an array reference. If $id is given, this XML element will be used as an id.

Example:

```
$data->{column}->[4]->{label}='testlabel';
```

HASH|PHASH

Element occurs zero or more times. Folding will be done using a hash reference. If $key is given, this XML element will be used as a key.

Example:

```
$data->{axis}->{gate voltage}->{unit}="mV";
```

read_xml($declaration,$filename)

Opens a XML file $filename. Returns Lab::Data::XMLtree object.

read_yaml($declaration,$filename)

Opens a YAML file $filename. Returns Lab::Data::XMLtree object.

METHODS

merge_tree($tree)

Merge another Lab::Data::XMLtree into this one. Other tree must not necessarily be blessed.

save_xml($filename)

Saves the tree as XML to $filename.

save_yaml($filename)

Saves the tree as YAML to $filename. PSCALAR etc. don't work yet.

to_string()

Returns a stringified version of the object. (Using Data::Dumper.)

autoload

Get/set anything you want. Accounts the data declaration.

PRIVATE FUNCTIONS

_load_xml($declaration,$filename)

_merge_node_lists($declaration,$destination_perlnode_list,$source_perlnode_list)

_parse_domnode_list($domnode_list,$defnode_list)

_write_node_list($generator,$defnode_list,$perlnode_list)

_getset_node_list_from_string($perlnode_list,$defnode_list,$nodes_string)

_get_defnode_type($defnode)

_magic_keys($defnode_list,$perlnode_list,$node_name,[@types])

_magic_get_perlnode($defnode_list,$perlnode_list,$node_name,$key,[@types])

_magic_set_perlnode($defnode_list,$perlnode_list,$node_name,$key,$value,[@types])

4.5 Lab::Data::Plotter

Plot data with Gnuplot

SYNOPSIS

```
use Lab::Data::Plotter;

my $plotter=new Lab::Data::Plotter($metafile);

my %plots=$plotter->available_plots();
my @names=keys %plots;

$plotter->plot($names[0]);
```

DESCRIPTION

This module can plot data with GnuPlot. It plots data from `.DATA` files and takes into account the data information in the corresponding `.META` file.

The module also offers the possibility to plot data live, while it is being aquired.

CONSTRUCTOR

new

```
$plotter=new Lab::Data::Plotter($meta,\%options);
```

Creates a Plotter object. `$meta` is either an object of type Lab::Data::Meta or a filename that points to a `.META` file.

Available options are

dump

eps

fulllabels

last_live

METHODS

available_plots

```
my %plots=$plotter->available_plots();
```

plot

```
$plotter->plot($plot);
```

start_live_plot

```
$plotter->start_live_plot($plot);
```

update_live_plot

```
$plotter->update_live_plot();
```

stop_live_plot

```
$plotter->stop_live_plot();
```

4.6 plotter.pl

Plot data with GnuPlot

SYNOPSIS

plotter.pl [OPTIONS] METAFILE

DESCRIPTION

This is a commandline tool to plot data that has been recorded using the Lab::Measurement module.

OPTIONS AND ARGUMENTS

The file `METAFILE` contains the meta information for the data that is to be plotted. The name OR number of the plot that you want to draw must be supplied with the `-plot` option, unless you use the `-list_plots` option, that lists all available plots defined in the `METAFILE`.

–help|-?

Print short usage information.

–man

Show manpage.

–listplots

List available plots defined in `METAFILE`.

–plot=name –plot=number

Show the plot with name `name` or number `number`. Numbers are given by the `-list_plots` option.

–dump=filename

Do not plot now, but dump a gnuplot file `filename` instead.

–eps=filename

Don't plot on screen, but create eps file `filename`.

–fulllabels

Also show axis descriptions in plot.