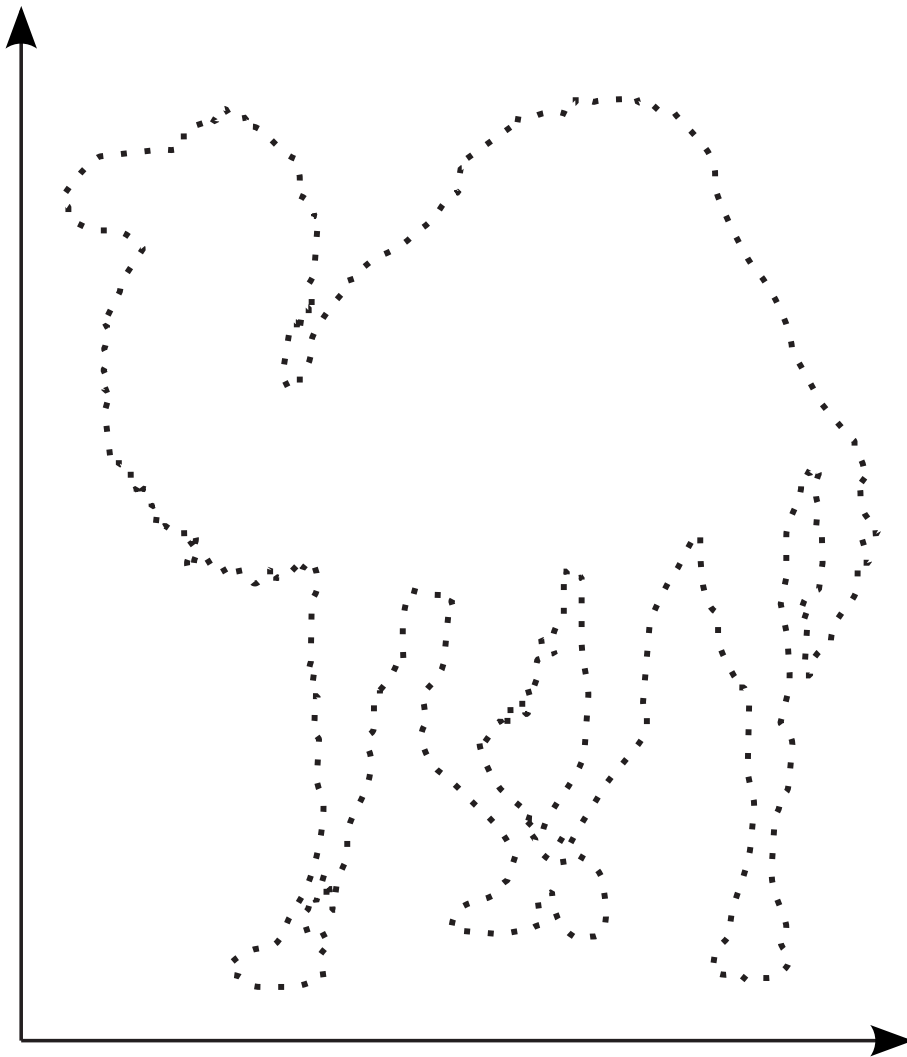

Lab::Measurement documentation



Contents

1	The Lab::Measurement package	5
1.1	Change log for the Lab::Measurement distribution	5
1.2	Lab::Measurement::Installation	7
1.3	Lab::VISA::Tutorial	11
1.3.1	Introduction	11
1.3.2	Measurement automation basics	11
1.3.3	Architecture	12
1.3.4	Using pure VISA calls	13
1.3.5	Using the Lab::Instrument class	15
1.3.6	Using Lab::Instrument::xxx virtual instruments	17
1.3.7	Using Lab::Tools	18
1.3.8	References	23
1.4	Implementing a current/voltage source driver	25
1.5	Example scripts	29
1.5.1	yoko-goto.pl	29
1.5.2	query_id.pl	31
1.5.3	srs_read.pl	33
1.5.4	gatesweep.pl	35
1.5.5	biasfield.pl	37
1.6	Utility scripts	39
1.6.1	plotter.pl	39
1.6.2	metainfo.pl	41
1.6.3	make_filelist.pl	43
1.6.4	make_overview.pl	45
1.7	High-level tool classes	47
1.7.1	Lab::Data::Writer	47
1.7.2	Lab::Measurement	49
1.7.3	Lab::Data::Meta	53
1.7.4	Lab::Data::XMLtree	57
1.7.5	Lab::Data::Plotter	61
1.8	Instrument control classes	63
1.8.1	Lab::Instrument	63
1.8.2	Multimeters	67
1.8.3	Voltage sources	77
1.8.4	Lock-in amplifiers	91
1.8.5	RF generators	93

Contents

1.8.6	Superconducting magnet power supplies	95
1.8.7	Temperature control devices	99
1.8.8	Cryostat handling devices	105
1.9	Connection classes	107
1.9.1	Lab::Connection	107
1.9.2	Lab::Connection::DEBUG	111
1.9.3	Lab::Connection::GPIB	113
1.9.4	Lab::Connection::LinuxGPIB	115
1.9.5	Lab::Connection::MODBUS_RS232	117
1.9.6	Lab::Connection::VISA	119
1.9.7	Lab::Connection::VISA_GPIB	121
1.9.8	Lab::Connection::IsoBus	123
1.10	Bus classes	125
1.10.1	Lab::Bus	125
1.10.2	Lab::Bus::DEBUG	127
1.10.3	Lab::Bus::LinuxGPIB	129
1.10.4	Lab::Bus::RS232	133
1.10.5	Lab::Bus::MODBUS_RS232	135
1.10.6	Lab::Bus::VISA	137
1.10.7	Lab::Bus::IsoBus	139
2	The Lab::VISA package	141
2.1	Lab::VISA::Installation	141
2.2	Lab::VISA	147

1 The Lab::Measurement package

1.1 Change log for the Lab::Measurement distribution

Initial release of Lab::Measurement

This section gives an overview of the most important points when you port a measurement script from the old Lab::Instrument and Lab::Tools distribution to Lab::Measurement.

Lab::Instrument classes

- The abbreviated way of specifying a GPIB board and address in the constructor is not supported anymore. Instead of the old

```
my $hp=new Lab::Measurement::HP34401A($board, $address);
```

you now have to explicitly provide

```
my $hp=new Lab::Measurement::HP34401A({
    connection_type => 'LinuxGPIB',
    gpib_board      => $board,
    gpib_address    => $address,
});
```

- The configuration parameters "gpib_board" and "gpib_address" are now for consistency spelled all in lowercase. Your script will fail if you use the uppercase "GPIB" variant.
- Every device now needs a configuration parameter "connection_type" (see above).
- In general, functions that read out device values are all prefixed with "get_" now, instead of "read_".
- SR830 functions like get_range and get_tc do not return strings anymore, but values in SI base units

Lab::Measurement class

- The default file suffixes have been changed from "DATA" and "META" to "dat" and "meta".

1 *The Lab::Measurement package*

- You can not abort the scripts using Lab::Measurement with "CTRL-C" anymore. Instead, just press "q", and the script will cleanly terminate at the next measurement point. The background for this is that some device drivers cannot handle an interruption, leading to undefined hardware behaviour.

COPYRIGHT AND LICENCE

(c) 2011 [Andreas K. Httel](#)

1.2 Lab::Measurement::Installation

Installation guide for Lab::Measurement

Introduction

Since Lab::Measurement does not contain any device driver code itself, its installation is pretty straightforward. However, before you can actually use it, you will have to install a driver binding back-end, such as Lab::VISA or Linux-GPIB, plus its dependencies. Please see the documentation of these packages for more details.

Installation on Windows XP with ActiveState Perl

Install Perl.

- Tested with ActivePerl from <http://www.activestate.com/Products/activeperl/index.mhtml>
- Make sure to include Perl Package Manager.
- Make sure to activate the check box to include perl directory in PATH variable.

Install gnuplot (not mandatory)

- Download from http://sourceforge.net/project/showfiles.php?group_id=2055 (gp425win32.zip)
- Extract and put it somewhere
- Add directory containing `pgnuplot.exe` to path: My Computer => Properties => Advanced => Environment Variables

Install the dependencies of our perl modules. Depending on how familiar you are with the perl infrastructure, the easiest might be to use PPM, the Perl Package Manager included with ActivePerl.

Lab::Measurement needs

```
XML::Generator (PPM would write it as XML-Generator)
XML::DOM
XML::Twig
YAML
```

Install Lab::Measurement

1 *The Lab::Measurement package*

- Unzip/copy sources
- Run the following commands in the source directory

```
perl Build.PL
perl Build
perl Build install
```

Have fun!

Installation on Windows XP with Strawberry Perl

Strawberry Perl is a Perl distribution for Windows that most closely mimics a Perl installation under Linux. It comes with gcc compiler, dmake and the other relevant tools included.

Lab::Measurement should in principle install out of the box with just the command

```
cpan Lab::Measurement
```

executed on the commandline.

Installation on Linux

As a Linux user you will probably be able to figure out most things yourself.

Install the dependencies

Best you'll use your distribution package management. You need

```
XML::Generator
XML::DOM
XML::Twig
YAML
... and GnuPlot
```

Install Lab::Measurement

- Unzip/copy sources
- Run the following commands in the source directory

```
perl Build.PL
perl Build
perl Build install
```

Have fun!

COPYRIGHT AND LICENCE

(c) 2010,2011 Daniel Schröder, Andreas Hüttel, Daniela Taubert, and others.

1 *The Lab::Measurement package*

1.3 **Lab::VISA::Tutorial**

Tutorial on using *Lab::VISA* and related packages

This document needs a complete rewrite...

1.3.1 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.

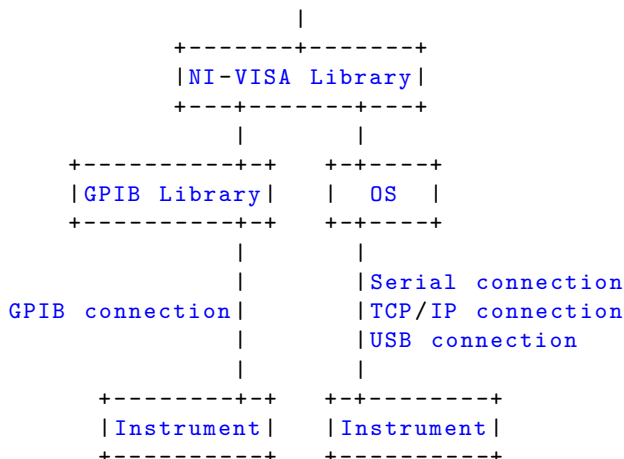
This tutorial will explain how to write measurement scripts that build on any of these stages. However, this tutorial does not intend to teach the Perl language itself. Some introduction into VISA and GPIB terminology is given, but then some familiarity also with these concepts is assumed. Not much is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

1.3.2 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 standardised interface to communicate with



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.

1.3.4 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) {

```

1 The Lab::Measurement package

```
    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,
my $result,
    # indicates if the operation was successful
    # 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.

1.3.5 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

```

1 The Lab::Measurement package

```
# 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:CENTOFF',          #disconnect channels

# Set up the Instrument
    ':FUNCPATT',              #set mode to Pulse/Pattern
    ':PER20ns',              #set period to 20 ns

# Set up Channel 1
    ':FUNC:MODE1PPULSE',     #set pattern mode to Pulse
    ':WIDT15ns',            #set width to 5 ns
    ':VOLT1:AMPL2.000V',     #set ampl to 2 V
    ':VOLT1:OFFSET1.5V',     #set offset to 1.5 V
    ':OUTP1:POSON',          #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 acquisition tasks.

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

1.3.6 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 destroyed 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__

```

1 The `Lab::Measurement` package

This example shows the usage of a dedicated virtual instrument class, namely `Lab::Instrument::HP34`, 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__
```

1.3.7 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]

1 The *Lab::Measurement* package

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. Therefore 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 acquired. 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; Lftung 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',

```

1 The Lab::Measurement package

```

        '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

1.3.8 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/>

1 *The Lab::Measurement package*

1.4 Implementing a current/voltage source driver

This document is ment as a guideline to and a help with the implementation of drivers for current and voltage sources. Since the complexity of the `Lab::Instrument` and `Lab::Instrument::Source` classes increases, it becomes more and more cumbersome to carefully read the (sometimes outdated) class documentation to keep track of correct interfaces, i.e., required methods and return values, to provide source device drivers.

The config hash

Let us start with what comes first, the config hash. It is used to provide default values for parameters that control the higher level functionality, namely `gate_protect` and to define the device parameters that should be stored internally (e.g. the range or the current output mode). At the moment when this documentation is written, an example for a correct config hash can be found in the class definition of the YokogawaGS200:

```
our %fields = (
  supported_connections => [ 'VISA_GPIB', 'GPIB', 'VISA', 'DEBUG'
    ],

  # default settings for the supported connections
  connection_settings => {
    gpib_board => 0,
    gpib_address => 22,
  },

  device_settings => {

    gate_protect           => 1,
    gp_equal_level         => 1e-5,
    gp_max_units_per_second => 0.05,
    gp_max_units_per_step  => 0.005,
    gp_max_step_per_second => 10,

    stepsize               => 0.01, # default stepsize for
      sweep without gate protect

    max_sweep_time=>3600,
    min_sweep_time=>0.1,
  },

  # If class does not provide set_$var for those, AUTOLOAD will
  take care.
  device_cache => {
    function               => "VOLT", # 'VOLT' -
      voltage, 'CURR' - current
    range                  => undef,
    level                  => undef,
    output                 => undef,
  },
}
```

1 The Lab::Measurement package

```
        device_cache_order => ['function', 'range'],
    );
```

Let me introduce the objects in this hash. `connection_settings` is more or less self-explanatory and should be overwritten by the user anyway.

The device_settings hash

The `device_settings` hash contains, in the case of a source driver, all the settings that are important to use the `gate_protect` feature of the `Lab::Instrument::Source` class. The values given are a careful choice, the user who wants to use gate protect will redefine them anyway. For a new driver, the hash can just be copy/pasted.

The device_cache hash

The `device_cache` hash contains all device parameters, i.e., parameters that can be set and read to and from the device, which should be stored on the software side. It is your decision what variables you add to the list, but make sure you

1. implement getter and setter for all these variables except the Current/Voltage level.
2. use `undef` as default if it is likely that this parameter is given on init. If it is not given, it will be read from the device.

The device_cache_order array

If the order of initializing parameters on the device is important, you should specify the order in this array.

The getter methods

The default for the getter should be to return the cached variable, i.e. the variable which is stored on the computer. If the option

```
    from_device => 1
```

is given, the variable should be read from the device.

The setter methods

should always set both on device and in the software cache. You can also use a

```
    error_check=>1
```

in the `$self-write>` command, then a possible error which appears on the device will automatically be set. Read also the section on error checking.

Default values

Best is to use `undef`.

Methods that **MUST** be provided by the device class

Please make sure you implement the following:

1. `get/set` for each variable in the `device_cache` with one exception: `set_level`. The setter should return the set value.
2. The sub `_set_level($target)` which will be called from `Lab::Instrument::Source` to use gate protect. Implement instead of `set_level()`.
3. A function `get_status()`.

The status sub

The sub `get_status` should read out the status byte of the device and create a hash with a descriptive flag and the state of the corresponding bit. The error bit should have the key **"ERROR"**.

Methods that should be implemented

It is convenient to implement the following functions if possible:

1. A sub `_sweep_to_level($target,$time)`.
2. A sub `get_error()`.

The sweep function

`_sweep_to_level($target,$time)` is given a target level `$target` and a sweep time `$time`. If the device supports this functionality, it should be implemented here. It should return `$target`.

`get_error()`

should read out the device's error stack. It should return ONE error at once in a single array with

```
[$errorcode,$errormessage]
```

The error checking framework

It is possible to wrap every `write($cmd)` call by an error checking routine. This can be invoked by providing the option `error_check`. For example:

```
$self->write($cmd, 'error_check' => 1)
```

After sending the command in `$cmd` to the device, the framework will use `get_status()` to read out the **ERROR** status bit. If it is set, `get_error()` will be used to fetch the error from the device.

General remarks on device driver development

Allow pass-through of commands in set & get

The advanced user should be given the possibility to do dirty workarounds when using the driver. To do this, he can provide options in the `write()` call, that are interpreted on connection level. This should in general also be possible when using `set_level` or any command that involves a `write()` call.

1.5 Example scripts

1.5.1 yoko-goto.pl

Sweeps a Yokogawa 7651 dc voltage source to a value given on the command line.

Usage example

```
$ perl yoko_goto.pl 12 0.8
```

Sweeps the Yokogawa 7651 dc voltage source with GPIB address 12 (on GPIB adaptor 0) to 0.8V, using a maximum step size of 5mV and at most 10 steps per second.

Author / Copyright

```
(c) Andreas K. Httel 2011
```

1 *The Lab::Measurement package*

1.5.2 query_id.pl

Queries and prints the instrument ID of a GPIB instrument; the GPIB address is the only command line parameter.

Usage example

```
$ perl query_id.pl 3
```

Author / Copyright

(c) Andreas K. Httel 2011

1 *The Lab::Measurement package*

1.5.3 srs_read.pl

Reads out reference amplitude, reference frequency, and current r and phi values of a Stanford Research SR830 lock-in amplifier. The only command line parameter is the GPIB address.

Usage example

```
$ perl srs_read.pl 8
```

Author / Copyright

```
(c) Andreas K. Httel 2011
```

1 *The Lab::Measurement package*

1.5.4 gatesweep.pl

Script to record a trace $I(V_g)$, i.e. dc current as function of gate voltage, in a Coulomb blockade measurement.

Measurement setup

Script: configuration section

Script: metadata section

Script: actual measurement loop

Author / Copyright

(c) Daniel Schmid, Markus Gaass, David Kalok, Andreas K. Httel 2011

1 *The Lab::Measurement package*

1.5.5 biasfield.pl

Script to record $I_{dc}(B, V_{bias})$ and the lock-in output $I_{ac}(B, V_{bias})$ in a Coulomb blockade measurement.

Author / Copyright

(c) Daniel Schmid, Markus Gaass, David Kalok, Andreas K. Httel 2011
Andreas K. Httel 2012

1 *The Lab::Measurement package*

1.6 Utility scripts

1.6.1 `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.

1 *The Lab::Measurement package*

1.6.2 metainfo.pl

Show info from meta file.

SYNOPSIS

metainfo.pl [OPTIONS] METAFILE

DESCRIPTION

This is a commandline tool to...

OPTIONS AND ARGUMENTS

The file METAFILE contains meta information about one dataset. This information is printed.

1 *The Lab::Measurement package*

1.6.3 `make_filelist.pl`

Generate a list of all plots defined in all metafiles of the current directory

SYNOPSIS

```
huettel@pc55508 ~ $ make_filelist.pl
```

DESCRIPTION

This is a commandline tool to quickly generate a list of all plots defined in the current directory. It generates a file `filelist.txt` suitable as input of `make_overview.pl`.

1 *The Lab::Measurement package*

1.6.4 make_overview.pl

Generate a LaTeX overview file with plots of all measurements in a directory

SYNOPSIS

```
huettel@pc55508 ~ $ make_overview.pl
```

Evaluates `filelist.txt` in the current directory, reads the specified metafiles, generates the specified plots and a LaTeX file `overview.tex`.

SYNTAX of filelist.txt

```
% Chapter 1 title
%% Section 1.1 title
Plotname      MYMEASUREMENT.META
Plotname      MYMEASUREMENT2.META
% Chapter 2 title
Plotname      ANOTHERMEASUREMENT.META
```

Pretty simple, huh? The only important thing is - the separator between the plot name and the file name has to be a TAB.

1 *The Lab::Measurement package*

1.7 High-level tool classes

1.7.1 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  => "#",
```

Example usage is like this:

```
$writer->configure({
    output_col_sep      => ":",
    output_comment_char => "//",
});
```

1 The Lab::Measurement package

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.

1.7.2 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",
    }
  ],

```

1 The Lab::Measurement package

```
{
  '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.

1 *The Lab::Measurement package*

1.7.3 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',
  {
```

1 The Lab::Measurement package

```

        original_filename => ['SCALAR'], # nur von GPplus-Import
            unterst tzt
        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'], # unn tz, aber von
            GPplus-Import unterst tzt
        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();
```

1 *The Lab::Measurement package*

1.7.4 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])

1 *The Lab::Measurement package*

1.7.5 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 acquired.

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

jpg

fulllabels

last_live

METHODS

available_plots

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

plot

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

1 *The Lab::Measurement package*

start_live_plot

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

update_live_plot

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

stop_live_plot

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

1.8 Instrument control classes

1.8.1 Lab::Instrument

Instrument base class

SYNOPSIS

Lab::Instrument is meant to be used as a base class for inheriting instruments. For very simple applications it can also be used directly, like

```
$generic_instrument = new Lab::Instrument ( connection_type =>
    VISA_GPIB, gpib_address => 14 );
my $idn = $generic_instrument->query('*IDN?');
```

Every inheriting class constructor should start as follows:

```
sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self = $class->SUPER::new(@_);
    $self->${\"__PACKAGE__\"::'_construct'}(\"__PACKAGE__\"); # check for
        supported connections, initialize fields etc.
    ...
}
```

Beware that only the first set of parameters specific to an individual GPIB board or any other bus hardware gets used. Settings for EOI assertion for example.

If you know what you're doing or you have an exotic scenario you can use the connection parameter "ignore_twins => 1" to force the creation of a new bus object, but this is discouraged - it will kill bus management and you might run into hardware/resource sharing issues.

DESCRIPTION

Lab::Instrument is the base class for Instruments. It doesn't do much by itself, but is meant to be inherited in specific instrument drivers. It provides general `read`, `write` and `query` methods and basic connection handling (internal, `_set_connection`, `_check_connection`).

CONSTRUCTOR

new This blesses \$self (don't do it yourself in an inheriting class!), initializes the basic "fields" to be accessed via AUTOLOAD and puts the configuration hash in \$self->config to be accessed in methods and inherited classes.

Arguments: just the configuration hash (or even-sized list) passed along from a child class constructor.

METHODS

write

```
$instrument->write($command <, {optional hashref/hash}> );
```

Sends the command `$command` to the instrument. An option hash can be supplied as second or also as only argument. Generally, all options are passed to the connection/bus, so additional named options may be supported based on the connection and bus and can be passed as a hashref or hash. See *Lab::Connection*.

Optional named parameters for hash: `error_check => 1/0` Invoke `$instrument->check_errors` after write. Default off.

read

```
$result=$instrument->read({ read_length => <max length>, brutal => <1/0>};
```

Reads a result of `ReadLength` from the instrument and returns it. Returns an exception on error.

If the parameter `brutal` is set, a timeout in the connection will not result in an Exception thrown, but will return the data obtained until the timeout without further comment. Be aware that this data is also contained in the the timeout exception object (see *Lab::Exception*).

Generally, all options are passed to the connection/bus, so additional named options may be supported based on the connection and bus and can be passed as a hashref or hash. See *Lab::Connection*.

query

```
$result=$instrument->query({ command => $command,
                             wait_query => $wait_query,
                             read_length => $read_length});
```

Sends the command `$command` to the instrument and reads a result from the instrument and returns it. The length of the read buffer is set to `read_length` or to the default set in the connection.

Waits for `wait_query` microseconds before trying to read the answer.

Generally, all options are passed to the connection/bus, so additional named options may be supported based on the connection and bus and can be passed as a hashref or hash. See *Lab::Connection*.

WriteConfig this is NOT YET IMPLEMENTED in this base class so far

```
$instrument->WriteConfig( 'TRIGGER' => { 'SOURCE' => 'CHANNEL1',
                                         'EDGE' => 'RISE' },
                        'ACQUIRE' => 'HRES',
                        'MEASURE' => { 'VRISE' => 'ON' } );
```

Builds up the commands and sends them to the instrument. To get the correct format a command rules hash has to be set up by the driver package

e.g. for SCPI commands `$instrument->{'CommandRules'} = { 'preCommand' => ':', 'inCommand' => ':', 'betweenCmdAndData' => ' ', 'postData' => " " # empty entries can be skipped };`

get_error

```
($errcode, $errmsg) = $instrument->get_error();
```

Method stub to be overwritten. Implementations read one error (and message, if available) from the device.

get_status

```
$status = $instrument->get_status();
if( $instrument->get_status('ERROR') ) {...}
```

Method stub to be overwritten. This returns the status reported by the device (e.g. the status byte retrieved via serial poll from GPIB devices). When implementing, use only information which can be retrieved very fast from the device, as this may be used often.

Without parameters, has to return a hashref with named status bits, e.g.

```
$status => { ERROR => 1, DATA => 0, READY => 1 }
```

If present, the first argument is interpreted as a key and the corresponding value of the hash above is returned directly.

The 'ERROR'-key has to be implemented in every device driver!

check_errors

```
$instrument->check_errors($last_command);

# try
eval { $instrument->check_errors($last_command) };
# catch
if ( my $e = Exception::Class->caught('Lab::Exception::
DeviceError') ) {
    warn "Errors from device!";
    @errors = $e->error_list();
    @devtype = $e->device_class();
    $command = $e->command();
}
}
```

Uses `get_error()` to check the device for occurred errors. Reads all present errors and throws a `Lab::Exception::DeviceError`. The list of errors, the device class and the last issued command(s) (if the script provided them) are enclosed.

1 *The Lab::Measurement package*

1.8.2 Multimeters

Lab::Instrument::Multimeter

Generic digital multimeter interface

DESCRIPTION The Lab::Instrument::Multimeter class implements a generic interface to digital all-purpose multimeters. It is intended to be inherited by other classes, not to be called directly, and provides a set of generic functions. The class

CONSTRUCTOR

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

METHODS

get_value

```
$value=$hp->get_value();
```

Read out the current measurement value, for whatever type of measurement the multimeter is currently configured.

id

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

Returns the instruments ID string.

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);
```

Display a message on the front panel.

display_clear

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

Clear the message displayed on the front panel.

1 *The Lab::Measurement package*

Lab::Instrument::HP34401A

HP/Agilent 34401A digital multimeter

SYNOPSIS

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

my $Agi = new Lab::Instrument::HP34401A({
    connection => new Lab::Connection::GPIB(
        gpib_board => 0,
        gpib_address => 14,
    ),
});
```

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 (not ported yet).

CONSTRUCTOR

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

METHODS**autozero**

```
$hp->autozero($setting);
```

`$setting` can be 1/'ON', 0/'OFF' or 'ONCE'.

When set to "ON", the device takes a zero reading after every measurement. "ONCE" perform one zero reading and disables the automatic zero reading. "OFF" does... you get it.

configure_voltage_dc

```
$hp->configure_voltage_dc($range, $integration_time);
```

Configures all the details of the device's DC voltage measurement function.

`$range` is a positive numeric value (the largest expected value to be measured) or one of 'MIN', 'MAX', 'AUTO'. It specifies the largest value to be measured. You can set any value, but the HP/Agilent 34401A effectively uses one of the values 0.1, 1, 10, 100 and 1000V.

`$integration_time` is the integration time in seconds. This implicitly sets the provided resolution.

pl_freq Parameter: pl_freq

```
$hp->pl_freq($new_freq);  
$npl_freq = $hp->pl_freq();
```

Get/set the power line frequency at your location (50 Hz for most countries, which is the default). This is the basis of the integration time setting (which is internally specified as a count of power line cycles, or PLCs). The integration time will be set incorrectly if this parameter is set incorrectly.

display_text

```
$Agi->display_text($text);  
print $Agi->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. Inherited from *Lab::Instrument::Multimeter*

display_on

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

Turn the front-panel display on. Inherited from *Lab::Instrument::Multimeter*

display_off

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

Turn the front-panel display off. Inherited from *Lab::Instrument::Multimeter*

display_clear

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

Clear the message displayed on the front panel. Inherited from *Lab::Instrument::Multimeter*

id

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

Returns the instrument ID string. Inherited from *Lab::Instrument::Multimeter*

get_value Inherited from *Lab::Instrument::Multimeter*

get_resistance

```
$resistance=$Agi->get_resistance($range,$resolution);
```

Preset and measure resistance with specified range and resolution.

get_voltage_dc

```
$datum=$Agi->get_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.

get_voltage_ac

```
$datum=$Agi->get_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.

get_current_dc

```
$datum=$Agi->get_current_dc($range,$resolution);
```

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

get_current_ac

```
$datum=$Agi->get_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.

beep**get_error****reset****config_voltage**

```
$inttime=$Agi->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.

1 The Lab::Measurement package

get_with_trigger_voltage_dc

```
@array = $Agi->get_with_trigger_voltage_dc()
```

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

scroll_message

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

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

beep

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

Issue a single beep immediately.

get_error

```
($err_num, $err_msg)=$Agi->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

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

Reset the multimeter to its power-on configuration.

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->get_voltage_dc();

```

DESCRIPTION The Lab::Instrument::HP3458A class implements an interface to the Agilent / HP 3458A digital multimeter.

CONSTRUCTOR

```

my $hp=new(%parameters);

```

METHODS**pl_freq Parameter: pl_freq**

```

$hp->pl_freq($new_freq);
$npl_freq = $hp->pl_freq();

```

Get/set the power line frequency at your location (50 Hz for most countries, which is the default). This is the basis of the integration time setting (which is internally specified as a count of power line cycles, or PLCs). The integration time will be set incorrectly if this parameter is set incorrectly.

get_voltage_dc

```

$voltage=$hp->get_voltage_dc();

```

Make a dc voltage measurement. This also enables autoranging. For finer control, use `configure_voltage_dc()` and `triggered_read`.

`__head2 triggered_read`

```

@values = $hp->triggered_read();
$value = $hp->triggered_read();

```

Trigger and read value(s) using the current device setup. This expects and digests a list of values in ASCII format, as set up by `configure_voltage_dc()`.

triggered_read_raw

```
$result = $hp->triggered_read_raw( read_until_length => $length
);
```

Trigger and read using the current device setup. This won't do any parsing and just return the answer from the device. If `$read_until_length` (integer) is specified, it will try to continuously read until it has gathered this amount of bytes.

configure_voltage_dc

```
$hp->configure_voltage_dc($range, $integration_time);
```

Configure range and integration time for the following DCV measurements.

`$range` is a voltage or one of "AUTO", "MIN" or "MAX". `$integration_time` is given in seconds or one of "DEFAULT", "MIN" or "MAX".

configure_voltage_dc_trigger

```
$hp->configure_voltage_dc_trigger($range, $integration_time,
    $count, $delay);
```

Configures range, integration time, sample count and delay (between samples) for triggered readings.

`$range`, `$integration_time`: see `configure_voltage_dc()`. `$count` is the sample count per trigger (integer). `$delay` is the delay between the samples in seconds.

configure_voltage_dc_trigger_highspeed

```
$hp->configure_voltage_dc_trigger_highspeed($range,
    $integration_time, $count, $delay);
```

Same as `configure_voltage_dc_trigger`, but configures the device for maximum measurement speed. Values are transferred in SINT format and can be fetched and decoded using `triggered_read_raw()` and `decode_SINT()`. This mode allows measurements of up to about 100 kSamples/second.

`$range`: see `configure_voltage_dc()`. `$integration_time`: integration time in seconds. The default is 1.4e-6. `$count` is the sample count per trigger (integer). `$delay` is the delay between the samples in seconds.

set_display_state Parameter: `display_state`

```
$hp->set_display_state(1/'on'/0/'off');
```

Turn the front-panel display on/off (1/0)

set_display_text Parameter: `display_text`

```
$hp->set_display_text($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.

check_errors

```
$instrument->check_errors($last_command);

# try
eval { $instrument->check_errors($last_command) };
# catch
if ( my $e = Exception::Class->caught('Lab::Exception::
    DeviceError') ) {
    warn "Errors from device!";
    @errors = $e->error_list();
    @devtype = $e->device_class();
    $command = $e->command();
}
else {
    $e = Exception::Class->caught();
    ref $e ? $e->rethrow; die $e;
}
}
```

Uses `get_error()` to check the device for occurred errors. Reads all present error and throws a `Lab::Exception::DeviceError`. The list of errors, the device class and the last issued command(s) (if the script provided them) are enclosed.

set_nlpc

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

Sets the integration time in units of power line cycles.

reset

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

Reset the multimeter to its power-on configuration. Same as `preset('NORM')`.

preset

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

\$config can be 'FAST' / 0 'NORM' / 1 'DIG' / 2

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

selftest

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

Starts the internal self-test routine.

autocalibration

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

Starts the internal autocalibration. Warning... this procedure takes 11 minutes with the 'ALL' mode!

\$mode can be 'ALL' / 0 'DCV' / 1 'AC' / 2 'OHMS' / 4 each meaning the obvious.

decode_SINT

```
@values = $hp->decode_SINT( $SINT_data, <$iscale> );
```

Takes a data blob with SINT values and decodes them into a numeric list. The used \$iscale parameter is read from the device by default if omitted. Make sure the device still has the same settings as used to obtain \$SINT_data, or iscale will be off which leads to invalid data decoding.

1.8.3 Voltage sources

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 that implement real voltage sources (e.g. the *Lab::Instrument::Yokogawa7651* class).

The class provides a unified user interface for those 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.

There's no direct user application of this class.

CONSTRUCTOR

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

This constructor will only be used by instrument drivers that inherit this class, not by the user. It accepts an additional configuration hash as parameter 'default_device_settings'. The first hash contains the parameters used by default for this device and its subchannels, if any. The second hash can be used to override options for this instance while still using the defaults for derived objects. If `\%config` is missing, `\%default_config` is used.

The instrument driver (e.g. *Lab::Instrument::Yokogawa7651*) has e.g. a constructor like this:

```
$yoko=new Lab::Instrument::Yokogawa7651({
    connection_type => 'LinuxGPIB',
    gpib_board      => $board,
    gpib_address    => $address,

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

METHODS

configure

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

1 The Lab::Measurement package

Supported configure options:

In general, all parameters which can be changed by access methods of the class/object can be used. In fact this is what happens, and the config hash given to `configure()` is just a shorthand for this. The following are equivalent:

```
$source->set_gate_protect(1);
$source->set_gp_max_units_per_second(0.1);
...

$source->configure({ gate_protect=>1, gp_max_units_per_second=>0.1,
...})
```

Options in detail:

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_units_per_second`, `gp_max_units_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_units_per_second

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

gp_max_units_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_units

The smallest allowed output voltage.

gp_max_units

The largest allowed output voltage.

gp_equal_level

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

set_level

```
$new_volt=$self->set_level($srclvl);
```

Sets the output to \$srclvl (in Volts or Ampere). If the configure option `gate_protect` is set to a true value, the safety mechanism takes into account the `gp_max_units_per_step`, `gp_max_units_per_second` etc. settings, by employing the `sweep_to_level` method.

Returns for `fast_set` off the actually set output source level. This can be different from \$srclvl, due to the `gp_max_units`, `gp_min_units` settings. For `fast_set` on, `set_level` returns always \$level.

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

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

__set_level(\$targetlvl) Function Stub. Has to be overwritten by device driver.

The function should set the source level to \$targetlvl on the device. Should return the newly set source level.

get_level() Function Stub. Has to be overwritten by device driver.

The function should return the source level from the device cache. If called with the option `from_device => 1`, the value should be fetched from the device. Should return the current source level.

get_range() Function Stub. Has to be overwritten by device driver.

The function should return the source range from the device cache. If called with the option `from_device => 1`, the value should be fetched from the device. Should return the current source range.

set_range() Function Stub. Has to be overwritten by device driver.

The function should set the source range on the device. Should return the newly set source range.

sweep_to_level

```
$new_volt=$self->sweep_to_level($srclvl);
$new_volt=$self->sweep_to_level($srclvl,$channel);
```

1 The Lab::Measurement package

This method sweeps the output source level to the desired value and only returns then. If the specific Instrument implemented `__sweep_to_level`, this version is preferred.

Returns the actually set output source level. This can be different from `$srclvl`, due to the `gp_max_units`, `gp_min_units` settings.

get_level

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

Returns the source level currently set.

create_subsource

```
$bigc2 = $bigsource->create_subsource( channel=>2,  
gp_max_units_per_second=>0.01 );
```

Returns a new instrument object with its default channel set to channel `$channel_nr` of the parent multi-channel source. The `device_settings` given to the parent at instantiation (or the `default_device_settings` if present) will be used as default **values**, which can be overwritten by parameters to `create_subsource()`.

Lab::Instrument::DummySource

Dummy voltage source

DESCRIPTION The Lab::Instrument::DummySource class implements a dummy voltage source that does nothing but can be used for testing purposes.

Only developers will ever make use of this class.

1 *The Lab::Measurement package*

Lab::Instrument::Yokogawa7651

Yokogawa 7651 DC source

SYNOPSIS

```

use Lab::Instrument::Yokogawa7651;

my $gate14=new Lab::Instrument::Yokogawa7651(
    connection_type => 'LinuxGPIB',
    gpib_address => 22,
    gate_protect => 1,
    level => 0.5,
);
$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(%configuration_HASH) HASH is a list of tuples given in the format

```

key => value,

```

please supply at least the configuration for the connection: `connection_type => "LinuxGPIB"` `gpib_address =>`

you might also want to have gate protect from the start (the default values are given):

```

    gate_protect => 1,

    gp_equal_level          => 1e-5,
    gp_max_units_per_second => 0.05,
    gp_max_units_per_step   => 0.005,
    gp_max_step_per_second  => 10,
    gp_max_units_per_second => 0.05,
    gp_max_units_per_step   => 0.005,

    max_sweep_time=>3600,
    min_sweep_time=>0.1,

```

If you want to use the sweep function without using gate protect, you should specify

```

    stepsize=>0.01

```

Additionally there is support to set parameters for the device "on init": If those values are not specified, defaults are supplied by the driver.

1 The Lab::Measurement package

```
function                                     => Voltage, # specify "
    Voltage" or "Current" mode, string is case
    insensitive
range                                         => undef,
level                                        => undef,
output                                       => undef,
```

If those values are not specified, the current device configuration is left unaltered.

METHODS

set_voltage(\$voltage) Sets the output voltage to \$voltage. Returns the newly set voltage.

get_voltage() Returns the currently set \$voltage. The value is read from the driver cache by default. Provide the option
device_cache => 1
to read directly from the device.

set_current(\$current) Sets the output current to \$current. Returns the newly set current.

get_current() Returns the currently set \$current. The value is read from the driver cache by default. Provide the option
device_cache => 1
to read directly from the device.

set_range(\$range) Set the output range for the device. \$range should be either in decimal or scientific notation. Returns the newly set range.

get_info() Returns the information provided by the instrument's 'OS' command, in the form of an array with one entry per line. For display, use `join(',', $yoko->get_info());` or similar.

set_output(\$onoff) Sets the output switch to "1" (on) or "0" (off). Returns the new output state;

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

Range	Accuracy 90d +-(% of setting + V)	Accuracy 1yr +-(% of setting + V)	Temperature Coefficient +-(% of setting + V)/ 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 Resistance	Output Noise DC to 10Hz	Output Noise DC to 10kHz (typical data)
10mV	-	approx. 20hm	3 Vp -p	30 Vp -p
100mV	-	approx. 20hm	5 Vp -p	30 Vp -p
1V	+/-120mA	less than 2m0hm	15 Vp -p	60 Vp -p
10V	+/-120mA	less than 2m0hm	50 Vp -p	100 Vp -p
30V	+/-120mA	less than 2m0hm	150 Vp -p	200 Vp -p

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

DC current

Range	Maximum Output	Resolution	Stability (24 h) +-(% of setting + A)	Stability (90 days) +-(% of setting + A)
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	1 A	0.004 + 3	0.016 + 5

Range	Accuracy (90 days) +-(% of setting + A)	Accuracy (1 year) +-(% of setting + A)	Temperature Coefficient +-(% of setting + A)/ 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 Resistance	Output Noise DC to 10Hz	Output Noise DC to 10kHz (typical data)
1mA	+ -30 V	more than 100M0hm	0.02 Ap -p	0.1 Ap -p
10mA	+ -30 V	more than 100M0hm	0.2 Ap -p	0.3 Ap -p
100mA	+ -30 V	more than 10M0hm	2 Ap -p	3 Ap -p

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

Lab::Instrument::YokogawaGS200

Yokogawa GS200 DC source

SYNOPSIS

```

use Lab::Instrument::YokogawaGS200;

my $gate14=new Lab::Instrument::YokogawaGS200(
    connection_type => 'LinuxGPIB',
    gpib_address => 22,
    function => 'VOLT',
    level => 0.4,
);
$gate14->set_voltage(0.745);
print $gate14->get_voltage();

```

DESCRIPTION The `Lab::Instrument::YokogawaGS200` class implements an interface to the discontinued voltage and current source GS200 by Yokogawa. This class derives from `Lab::Instrument::Source` and provides all functionality described there.

CONSTRUCTORS

new(%configuration_HASH) HASH is a list of tuples given in the format

key => value,

please supply at least the configuration for the connection: `connection_type => "LinuxGPIB"` `gpib_address =>`

you might also want to have gate protect from the start (the default values are given):

```

    gate_protect => 1,

    gp_equal_level           => 1e-5,
    gp_max_units_per_second => 0.05,
    gp_max_units_per_step   => 0.005,
    gp_max_step_per_second  => 10,
    gp_max_units_per_second => 0.05,
    gp_max_units_per_step   => 0.005,

    max_sweep_time=>3600,
    min_sweep_time=>0.1,

```

Additionally there is support to set parameters for the device "on init":

```

function           => undef, # 'VOLT' -
    voltage, 'CURR' - current
range              => undef,
level              => undef,
output            => undef,

```

If those values are not specified, they are read from the device.

METHODS

sweep_to_voltage(\$voltage,\$time) Sweeps to \$voltage in \$time seconds. For this function to work, the source has to be in output mode.

Returns the newly set voltage. This function is also called internally to set the voltage when gate protect is used.

program_run(\$program) Runs a program stored on the YokogawaGS200. If no program name is given, the currently loaded program is executed.

program_pause Pauses the currently running program.

program_continue Continues the paused program.

set_voltage(\$voltage) Sets the output voltage. The driver checks whether you stay inside the currently selected range. Returns the newly set voltage.

set_voltage_auto(\$voltage) Sets the output voltage. The range is chosen automatically. Does not work with gate protect on. Returns the newly set voltage.

set_current(\$current) See set_voltage

set_current_auto(\$current) See set_current_auto

set_range(\$range)

```
Fixed voltage mode
10E-3      10mV
100E-3     100mV
1E+0       1V
10E+0      10V
30E+0      30V
```

```
Fixed current mode
1E-3              1mA
10E-3             10mA
100E-3            100mA
200E-3            200mA
```

Please use the format on the left for the range command.

set_function(\$function) Sets the source function. The Yokogawa supports the values

"CURR" for current mode and "VOLT" for voltage mode.

Returns the newly set source function.

set_voltage_limit(\$limit) Sets a voltage limit to protect the device. Returns the new voltage limit.

set_current_limit(\$limit) See set_voltage_limit.

output_on() Sets the output switch to on and returns the new value of the output status.

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

get_error() Queries the error code from the device. This is a very useful thing to do when you are working remote and the source is not responding.

get_voltage()

get_current()

get_output()

get_range()

1 *The Lab::Measurement package*

1.8.4 Lock-in amplifiers

Lab::Instrument::SR830

Stanford Research SR830 Lock-In Amplifier

SYNOPSIS

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

my $sr=new Lab::Instrument::SR830(
    connection_type=>'LinuxGPIB',
    gpib_address=>12,
);

($x,$y) = $sr->get_xy();
($r,$phi) = $sr->get_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

get_xy

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

Reads channels x and y simultaneously; returns an array.

get_rphi

```
($r,$phi)= $sr830->get_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 number in Volt.

1 The Lab::Measurement package

get_sens

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

Returns the value of the sensitivity, as number in Volt.

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 number in seconds.

get_tc

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

Returns the time constant, as number in seconds.

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 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.

1.8.5 RF generators

Lab::Instrument::HP83732A

HP 83732A Series Synthesized Signal Generator

SYNOPSIS

DESCRIPTION

CONSTRUCTOR

METHODS

1 *The Lab::Measurement package*

1.8.6 Superconducting magnet power supplies

Lab::Instrument::MagnetSupply

Base class for magnet power supply instruments

(c) 2010 David Borowsky, Andreas K. Httel
2011 Andreas K. Httel

1 *The Lab::Measurement package*

Lab::Instrument::IPS12010

Oxford Instruments IPS 120-10 superconducting magnet supply

(c) 2010, 2011 [Andreas K. Httel](#)

1 *The Lab::Measurement package*

1.8.7 Temperature control devices

Lab::Instrument::TemperatureControl

Base class for temperature control instruments

(c) 2011 [Andreas K. Httel](#)

1 *The Lab::Measurement package*

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=...
```

1 The Lab::Measurement package

Lab::Instrument::ITC503

Oxford Instruments ITC503 Intelligent Temperature Control

SYNOPSIS

```
use Lab::Instrument::ITC503;

my $itc=new Lab::Instrument::ITC503(
    isobus_address=>3,
);
```

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

1 *The Lab::Measurement package*

1.8.8 Cryostat handling devices

Lab::Instrument::ILM210

Oxford Instruments ILM Intelligent Level Meter

SYNOPSIS

```
use Lab::Instrument::ILM210;

my $ilm=new Lab::Instrument::ILM210(
    connection_type=>'IsoBus',
    base_connection=>...,
    isobus_address=>5,
);
```

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

CONSTRUCTOR

```
my $ilm=new Lab::Instrument::ILM210(
    connection_type=>'IsoBus',
    base_connection=> $iso,
    isobus_address=> $addr,
);
```

Instantiates a new ILM210 object, attached to the GPIB or RS232 connection (of type Lab::Connection) \$iso, with IsoBus address \$addr.

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.

1 *The Lab::Measurement package*

1.9 Connection classes

1.9.1 Lab::Connection

Connection base class

SYNOPSIS

This is the base class for all connections. Every inheriting classes constructors should start as follows:

```
sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self = $class->SUPER::new(@_);
    $self->_construct(__PACKAGE__); #initialize fields etc.
    ...
}
```

DESCRIPTION

`Lab::Connection` is the base class for all connections and implements a generic set of access methods. It doesn't do anything on its own.

A connection in general is an object which is created by an instrument and provides it with a generic set of methods to talk to its hardware counterpart. For example `Lab::Instrument::HP34401A` can work with any connection of the type GPIB, that is, connections derived from `Lab::Connection::GPIB`.

That would be, for example `Lab::Connection::LinuxGPIB` `Lab::Connection::VISA_GPIB`

Towards the instrument, these look the same, but they work with different drivers/backends.

CONSTRUCTOR

new Generally called in child class constructor:

```
my $self = $class->SUPER::new(@_);
```

Return blessed \$self, with @_ accessible through `$self->Config()`.

METHODS

Clear Try to clear the connection, if the bus supports it.

Read

```
my $result = $connection->Read();
my $result = $connection->Read( timeout => 30 );

configuration hash options:
brutal => <1/0> # suppress timeout errors if set to 1
read_length => <int> # how many bytes/characters to read
...see bus documentation
```

Reads a string from the connected device. In this basic form, its merely a wrapper to the method `connection_read()` of the used bus. You can give a configuration hash, which options are passed on to the bus. This hash is also meant for options to Read itself, if need be.

Write

```
$connection->Write( command => '*CLS' );

configuration hash options:
command => <command string>
...more (see bus documentation)
```

Write a command string to the connected device. In this basic form, its merely a wrapper to the method `connection_write()` of the used bus. You need to supply a configuration hash, with at least the key 'command' set. This hash is also meant for options to Read itself, if need be.

Query

```
my $result = $connection->Query( command => '*IDN?' );

configuration hash options:
command => <command string>
wait_query => <wait time between read and write in seconds> #
    overwrites the connection default
brutal => <1/0> # suppress timeout errors if set to true
read_length => <int> # how many bytes/characters to read
...more (see bus documentation)
```

Write a command string to the connected device, and immediately read the response.

You need to supply a configuration hash with at least the 'command' key set. The `wait_query` key sets the time to wait between read and write in usecs. The hash is also passed along to the used bus methods.

BrutalRead The same as read with the 'brutal' option set to 1.

BrutalQuery The same as Query with the 'brutal' option set to 1.

LongQuery The same as Query with 'read_length' set to 10240.

config Provides unified access to the fields in initial @_ to all the child classes. E.g.

```
$GPIB_Address=$instrument->Config(gpib_address);
```

Without arguments, returns a reference to the complete \$self->Config aka @_ of the constructor.

```
$Config = $connection->Config();  
$GPIB_Address = $connection->Config()->{'gpib_address'};
```

1 *The Lab::Measurement package*

1.9.2 Lab::Connection::DEBUG

Debug connection

DESCRIPTION

Connection to the DEBUG bus.

1 *The Lab::Measurement package*

1.9.3 Lab::Connection::GPIB

Base class for GPIB connections

SYNOPSIS

This is the base class for all connections providing a GPIB interface. Every inheriting class constructor should start as follows:

```

sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self = $class->SUPER::new(@_);
    $self->_construct(__PACKAGE__); #initialize fields etc.
    ...
}

```

DESCRIPTION

Lab::Connection::GPIB is the base class for all connections providing a GPIB interface. It is not usable on its own. It inherits from *Lab::Connection*.

Its main use so far is to define the data fields common to all GPIB interfaces.

CONSTRUCTOR

new Generally called in child class constructor:

```
my $self = $class->SUPER::new(@_);
```

Return blessed \$self, with @_ accessible through \$self->Config().

METHODS

This just calls back on the methods inherited from Lab::Connection.

If you inherit this class in your own connection however, you have to provide the following methods. Take a look at e.g. Lab::Connection::VISA_GPIB and at the basic implementations in *Lab::Connection* (they may even suffice).

Write() Takes a config hash, has to at least pass the key 'command' correctly to the underlying bus.

Read() Takes a config hash, reads back a message from the device.

Clear() Clears the instrument.

1 *The Lab::Measurement package*

config Provides unified access to the fields in initial @_ to all the child classes. E.g.

```
$GPIB_PAddress=$instrument->Config(GPIB_PAddress);
```

Without arguments, returns a reference to the complete \$self->Config aka @_ of the constructor.

```
$Config = $connection->Config();  
$GPIB_PAddress = $connection->Config()->{'GPIB_PAddress'};
```

1.9.4 Lab::Connection::LinuxGPIB

Connection class which uses LinuxGPIB (libgpib0) as a backend.

SYNOPSIS

This is not called directly. To make a GPIB supporting instrument use Lab::Connection::LinuxGPIB, set the `connection_type` parameter accordingly:

```
$instrument = new HP34401A( connection_type => 'LinuxGPIB', gpib_board =>
0, gpib_address => 14 )
```

DESCRIPTION

Lab::Connection::LinuxGPIB provides a GPIB-type connection with the bus *Lab::Bus::LinuxGPIB*, using Linux GPIB (aka libgpib0 in debian) as backend.

It inherits from *Lab::Connection::GPIB* and subsequently from *Lab::Connection*.

For *Lab::Bus::LinuxGPIB*, the generic methods of *Lab::Connection* suffice, so only a few defaults are set: `wait_status=>0, # usec`; `wait_query=>10, # usec`; `read_length=>1000, # bytes`

CONSTRUCTOR

new

```
my $connection = new Lab::Connection::LinuxGPIB(
    gpib_board => 0,
    gpib_address => $address,
    gpib_saddress => $secondary_address
)
```

METHODS

This just falls back on the methods inherited from *Lab::Connection*.

config Provides unified access to the fields in initial `@_` to all the child classes. E.g.

```
$GPIB_Address=$instrument->Config(gpib_address);
```

Without arguments, returns a reference to the complete `$self->Config` aka `@_` of the constructor.

```
$Config = $connection->Config();
$GPIB_Address = $connection->Config()->{'gpib_address'};
```

1 *The Lab::Measurement package*

1.9.5 Lab::Connection::MODBUS_RS232

Connection class for Lab::Bus::MODBUS_RS232

1 *The Lab::Measurement package*

1.9.6 Lab::Connection::VISA

VISA-type connection class which uses *Lab::Bus::VISA* and thus NI VISA (*Lab::VISA*) as a backend.

SYNOPSIS

This is not called directly. To make a VISA supporting instrument use *Lab::Connection::VISA*, set the `connection_type` parameter accordingly:

```
$instrument = new HP34401A( connection_type => 'VISA', resource_name =>
'GPIB0::14::INSTR', )
```

DESCRIPTION

Lab::Connection::VISA provides a VISA-type connection with *Lab::Bus::VISA* using NI VISA (*Lab::VISA*) as backend.

It inherits from *Lab::Connection*.

CONSTRUCTOR

new

```
my $connection = new Lab::Connection::VISA(
    connection_type => 'VISA',
    resource_name => 'GPIB0::14::INSTR',
)
```

METHODS

This just falls back on the methods inherited from *Lab::Connection*.

config Provides unified access to the fields in initial `@_` to all the child classes. E.g.

```
$GPIB_Address=$instrument->Config(gpib_address);
```

Without arguments, returns a reference to the complete `$self->Config` aka `@_` of the constructor.

```
$Config = $connection->Config();
$GPIB_Address = $connection->Config()->{'gpib_address'};
```

1 *The Lab::Measurement package*

1.9.7 Lab::Connection::VISA_GPIB

GPIB-type connection class which uses *Lab::Bus::VISA* and thus NI VISA (*Lab::VISA*) as a backend.

SYNOPSIS

This class is not called directly. To make a GPIB supporting instrument use *Lab::Connection::VISA_GPIB*, set the *connection_type* parameter accordingly:

```
$instrument = new HP34401A(
    connection_type => 'VISA_GPIB',
    gpib_board => 0,
    gpib_address => 14
)
```

DESCRIPTION

Lab::Connection::VISA_GPIB provides a GPIB-type connection with *Lab::Bus::VISA* using NI VISA (*Lab::VISA*) as backend.

It inherits from *Lab::Connection::GPIB* and subsequently from *Lab::Connection*.

The main feature is to assemble the standard gpib connection options *gpib_board* *gpib_address* *gpib_saddress* into a valid NI VISA resource name (see *Lab::Connection::VISA* for more details).

CONSTRUCTOR

new

```
my $connection = new Lab::Connection::VISA_GPIB(
    gpib_board => 0,
    gpib_address => $address,
    gpib_saddress => $secondary_address
)
```

METHODS

This just falls back on the methods inherited from *Lab::Connection*.

config Provides unified access to the fields in initial *@_* to all the child classes. E.g.

```
$GPIB_Address=$instrument->Config(gpib_address);
```

Without arguments, returns a reference to the complete *\$self->Config* aka *@_* of the constructor.

```
$Config = $connection->Config();
$GPIB_Address = $connection->Config()->{'gpib_address'};
```

1 *The Lab::Measurement package*

1.9.8 Lab::Connection::IsoBus

IsoBus connection class which uses *Lab::Bus::IsoBus* as a backend.

SYNOPSIS

This is not called directly. To make an Isobus instrument use *Lab::Connection::IsoBus*, set the `connection_type` parameter accordingly:

```
$instrument = new ILM210( connection_type => 'IsoBus', isobus_address => 3,
)
```

DESCRIPTION

Lab::Connection::IsoBus provides a connection with *Lab::Bus::IsoBus*, transparently handled via a pre-existing bus and connection object (e.g. serial or GPIB).

It inherits from *Lab::Connection*.

CONSTRUCTOR

new

```
my $connection = new Lab::Connection::IsoBus(
    connection_type => 'IsoBus',
    isobus_address => 3,
)
```

METHODS

This just falls back on the methods inherited from *Lab::Connection*.

config Provides unified access to the fields in initial `@_` to all the child classes. E.g.

```
$IsoBus_Address=$instrument->Config(isobus_address);
```

Without arguments, returns a reference to the complete `$self->Config` aka `@_` of the constructor.

```
$Config = $connection->Config();
$IsoBus_Address = $connection->Config()->{'isobus_address'};
```

1 *The Lab::Measurement package*

1.10 Bus classes

1.10.1 Lab::Bus

Bus base class

SYNOPSIS

This is a base class for inheriting bus types.

DESCRIPTION

`Lab::Bus` is a base class for individual buses. It does not do anything on its own. For more detailed information on the use of bus objects, take a look on a child class, e.g. `Lab::Bus::LinuxGPIB`.

`Lab::Bus::BusList` contains a hash with references to all the active buses in your program. They are put there by the constructor of the individual bus `Lab::Bus::new()` and have two levels: Package name and a unique bus ID (GPIB board index offers itself for GPIB). This is to transparently (to the use interface) reuse bus objects, as there may only be one bus object for every (hardware) bus. `weaken()` is used on every reference stored in this hash, so it doesn't prevent object destruction when the last "real" reference is lost. Yes, this breaks object orientation a little, but it comes so handy!

```
our %Lab::Bus::BusList = [
    $Package => {
        $UniqueID => $Object,
    }
    'Lab::Bus::GPIB' => {
        '0' => $Object,           "0" is the gpib board index
    }
]
```

Place your twin searching code in `$self->search_twin()`. Make sure it evaluates `$self->IgnoreTwin()`. Look at `Lab::Bus::LinuxGPIB`.

CONSTRUCTOR

new Generally called in child class constructor:

```
my $self = $class->SUPER::new(@_);
```

Return blessed `$self`, with `@_` accessible through `$self->Config()`.

METHODS

Config Provides unified access to the fields in initial `@_` to all the child classes.

1 *The Lab::Measurement package*

1.10.2 Lab::Bus::DEBUG

Interactive debug bus

DESCRIPTION

This will be an interactive debug bus, which prints out the commands sent by the measurement script, and lets you manually enter the instrument responses.

Unfinished, needs testing.

1 *The Lab::Measurement package*

1.10.3 Lab::Bus::LinuxGPIB

LinuxGPIB bus

SYNOPSIS

This is the GPIB bus class for the GPIB library `linux-gpib` (aka `libgpib0` in the debian world).

```
my $GPIB = new Lab::Bus::LinuxGPIB({ gpib_board => 0 });
```

or implicit through instrument and connection creation:

```
my $instrument = new Lab::Instrument::HP34401A({
    connection_type => 'LinuxGPIB',
    gpib_board => 0,
    gpib_address=>14,
})
```

DESCRIPTION

See <http://linux-gpib.sourceforge.net/> for details on the LinuxGPIB package. The package provides both kernel drivers and Perl bindings. Obviously, this will work for Linux systems only. On Windows, please use `Lab::Bus::VISA`. The interfaces are (errr, will be) identical.

Note: you don't need to explicitly handle bus objects. The Instruments will create them themselves, and existing bus will be automagically reused.

In GPIB, instantiating two bus with identical parameter "gpib_board" will logically lead to the reuse of the first one. To override this, use the parameter "ignore_twins" at your own risk.

CONSTRUCTOR

new

```
my $bus = Lab::Bus::GPIB({
    gpib_board => $board_num
});
```

Return blessed \$self, with @_ accessible through \$self->config().

gpib_board: Index of board to use. Can be omitted, 0 is the default.

Thrown Exceptions

Lab::Bus::GPIB throws

1 The Lab::Measurement package

```
Lab::Exception::GPIBError
  fields:
  'ibsta', the raw ibsta status byte received from linux-gpib
  'ibsta_hash', the ibsta bit values in a named hash ( 'DCAS' => $val,
    'DTAS' => $val, ... ).
    Use Lab::Bus::GPIB::VerboseIbstatus() to get a nice
    string representation

Lab::Exception::GPIBTimeout
  fields:
  'Data', this is meant to contain the data that (maybe) has been read
    /obtained/generated despite and up to the timeout.
  ... and all the fields of Lab::Exception::GPIBError
```

METHODS

connection_new

```
$GPIB->connection_new({ gpib_address => $paddr });
```

Creates a new connection ("instrument handle") for this bus. The argument is a hash, whose contents depend on the bus type. For GPIB at least 'gpib_address' is needed.

The handle is usually stored in an instrument object and given to connection_read, connection_write etc. to identify and handle the calling instrument:

```
$InstrumentHandle = $GPIB->connection_new({ gpib_address => 13 });
$result = $GPIB->connection_read($self->InstrumentHandle(), { options
  });
```

See Lab::Instrument::Read().

TODO: this is probably not correct anymore

connection_write

```
$GPIB->connection_write( $InstrumentHandle, { Cmd => $Command } );
```

Sends \$Command to the instrument specified by the handle.

connection_read

```
$GPIB->connection_read( $InstrumentHandle, { Cmd => $Command,
  ReadLength => $readlength, Brutal => 0/1 } );
```

Sends \$Command to the instrument specified by the handle. Reads back a maximum of \$readlength bytes. If a timeout or an error occurs, Lab::Exception::GPIBError or Lab::Exception::GPIBTimeout are thrown, respectively. The Timeout object carries the data received up to the timeout event, accessible through \$Exception->Data().

Setting Brutal to a true value will result in timeouts being ignored, and the gathered data returned without error.

timeout

```
$GPIB->timeout( $connection_handle , $timeout );
```

Sets the timeout in seconds for GPIB operations on the device/connection specified by `$connection_handle`.

config Provides unified access to the fields in initial `@_` to all the child classes. E.g.

```
$GPIB_Address=$instrument->config(gpib_address);
```

Without arguments, returns a reference to the complete `$self->config` aka `@_` of the constructor.

```
$config = $bus->config();
$GPIB_PAddress = $bus->config()->{'gpib_address'};
```

1 *The Lab::Measurement package*

1.10.4 Lab::Bus::RS232

RS232 or Virtual Comm port bus

SYNOPSIS

```
my $bus = Lab::Bus::RS232({
    port => '/dev/ttyACM0'
});
```

Return blessed \$self, with @_ accessible through \$self->config().

port: Device name to use (e.g. COM1 under Windows or /dev/ttyUSB1 under Linux)

TODO: check this!!!

DESCRIPTION

This is a bus for Lab::Measurement to communicate via RS232 or Virtual Comm port e.g. for FTDI devices.

CONSTRUCTOR

new All parameters are used as by Device::SerialPort. port is needed in every case. An additional parameter **reuse** is available if two instruments use the same port. This is mainly implemented for USBprologix gateway. **reuse** can be a SerialPort object or a Lab::Instrument... package. Default value for timeout is 500ms and can be set by the parameter "timeout". Other options: handshake, baudrate, databits, stopbits and parity

METHODS

Used by Lab::Instrument. Not for direct use!!!

Read Reads data.

Write Sent data to instrument

Handle Give instrument object handle

1 *The Lab::Measurement package*

1.10.5 Lab::Bus::MODBUS_RS232

RS232/RS485 MODBUS RTU protocol bus

SYNOPSIS

```
use Lab::Bus::MODBUS_RS232;
my $h = Lab::Bus::MODBUS_RS232->new({
    Interface => 'RS232',
    Port => 'COM1|/dev/ttyUSB1'
    slave_address => '1'
});
```

COM1 is the Windows notation, /dev/ttyUSB1 the Linux equivalent. Use as needed.

DESCRIPTION

This is an interface package for Lab::Measurement to communicate via RS232/RS485 with a MODBUS RTU enabled device. It uses Lab::Bus::RS232 (RS485 can be done using a RS232<->RS485 converter for now). It's main use is to calculate the checksums needed by MODBUS RTU.

Refer to your device for the correct port configuration.

As of yet, this driver does NOT fully implement all MODBUS RTU functions. Only the function codes 3 and 6 are provided.

CONSTRUCTOR

new All parameters are used as by Device::SerialPort respectively Lab::Bus::RS232. 'port' is needed in every case. Default value for timeout is 500ms and can be set by the parameter "Timeout". Other options you probably have to set: Handshake, Baudrate, Databits, Stopbits and Parity.

METHODS

Used by Lab::Connection. Not for direct use!!!

connectionRead Reads data. Arguments: function (0x01,0x02,0x03,0x04 - "Read Coils", "Read Discrete Inputs", "Read Holding Registers", "Read Input Registers") slave_address (0xFF) mem_address (0xFFFF, Address of first word) mem_count (0xFFFF, Count of words to read)

connectionWrite Send data to instrument. Arguments:

```
function (0x05,0x06,0x0F,0x10 - "Write_Single_Coil", "Write_Single_
Register", "Write_Multiple_Coils", "Write_Multiple_Registers")
```

1 *The Lab::Measurement package*

Currently only 0x06 is implemented.

```
slave_address (0xFF)
```

```
mem_address ( 0xFFFF, Address of word )
```

```
Value ( 0xFFFF, value to write to mem_address )
```

1.10.6 Lab::Bus::VISA

National Instruments VISA bus

SYNOPSIS

This is the VISA bus class for the NI VISA library.

```
my $visa = new Lab::Bus::VISA();
```

or implicit through instrument creation:

```
my $instrument = new Lab::Instrument::HP34401A({
    BusType => 'VISA',
});
```

DESCRIPTION

soon

CONSTRUCTOR

new

```
my $bus = Lab::Bus::VISA({
});
```

Return blessed \$self, with @_ accessible through \$self->config().

Options: none

Thrown Exceptions

Lab::Bus::VISA throws

```
Lab::Exception::VISAError
fields:
'status', the raw ibsta status byte received from linux-gpib
```

```
Lab::Exception::VISATimeout
fields:
'data', this is meant to contain the data that (maybe) has been read
/obtained/generated despite and up to the timeout.
... and all the fields of Lab::Exception::GPIBError
```

METHODS

connection_new

```
$visa->connection_new({ resource_name => "GPIB0::14::INSTR" });
```

Creates a new instrument handle for this bus.

The handle is usually stored in an instrument object and given to `connection_read`, `connection_write` etc. to identify and handle the calling instrument:

```
$InstrumentHandle = $visa->connection_new({ resource_name => "GPIB0  
::14::INSTR" });  
$result = $visa->connection_read($self->InstrumentHandle(), { options  
});
```

See `Lab::Instrument::Read()`.

connection_write

```
$visa->connection_write( $InstrumentHandle, { command => $command,  
wait_status => $wait_status } );
```

Sends `$command` to the instrument specified by the handle, and waits `$wait_status` microseconds before evaluating the status.

connection_read

```
$visa->connection_read( $InstrumentHandle, { command => $command,  
read_length => $read_length, brutal => 0/1 } );
```

Sends `$Command` to the instrument specified by the handle. Reads back a maximum of `$readlength` bytes. If a timeout or an error occurs, `Lab::Exception::VISAError` or `Lab::Exception::VISATimeout` are thrown, respectively. The `Timeout` object carries the data received up to the timeout event, accessible through `$Exception->Data()`.

Setting `Brutal` to a true value will result in timeouts being ignored, and the gathered data returned without error.

connection_query

```
$visa->connection_query( $InstrumentHandle, { command => $command,  
read_length => $read_length, wait_status => $wait_status,  
wait_query => $wait_query, brutal => 0/1 } );
```

Performs an `connection_write` followed by an `connection_read`, each given the supplied parameters. Waits `$wait_query` microseconds between Write and Read.

1.10.7 Lab::Bus::IsoBus

Oxford Instruments IsoBus bus

SYNOPSIS

soon

1 *The Lab::Measurement package*

2 The Lab::VISA package

2.1 Lab::VISA::Installation

Installation guide for Lab::VISA

Introduction

Lab::VISA has been tested to work on Linux and Windows, both with ActiveState Perl and the Microsoft VC++ Compiler, and Strawberry Perl with the included gcc compiler.

Installation on Windows XP with ActiveState Perl

Work with administrator account during installation.

Install VISA (and GPIB drivers) if necessary.

- Download current VISA release from NI (tested with 4.4.1, 4.5.0)
- Run installer
- Check location of visa32.lib (eg. C:\Programme\IVI Foundation\VISA\WinNT\lib\msc\visa32.lib) and remember for later.

Install Microsoft Visual C++ from <http://www.microsoft.com/express/Downloads/>

From now on run all commandline programs during installation only from the "Visual Studio Command line", which can be found in the Start menu.

Install Perl.

- Tested with ActivePerl from <http://www.activestate.com/Products/activeperl/index.mhtml>
- Make sure to include Perl Package Manager.
- Make sure to activate the check box to include perl directory in PATH variable.

Install gnuplot (not mandatory)

- Download from http://sourceforge.net/project/showfiles.php?group_id=2055 (gp425win32.zip)
- Extract and put it somewhere
- Add directory containing `pgnuplot.exe` to path: My Computer => Properties => Advanced => Environment Variables

Install dependencies of our perl modules. Depending on how familiar you are with the perl infrastructure, the easiest might be to use PPM, the Perl Package Manager included with ActivePerl.

Install Lab::VISA

- Unzip/copy sources
- In file `Makefile.PL` adapt the LIBS and INC settings according to your installation (the location looked up above). This is what worked for me:

```
'LIBS' => [ '-IC:\\Programme\\IVI_Foundation\\VISA\\WinNT\\lib\\msc\\visa32.lib" ' ]
'INC'   => '-IC:\\Programme\\IVI_Foundation\\VISA\\WinNT\\include" '
```

You can find the LIBS folder by checking the registry key "InstDir" in folder "HKEY_LOCAL_MACHINE\\SOFTWARE\\National Instruments\\NI-VISA for Windows 95/NT".

- Run the following commands in the source directory

```
perl Makefile.PL
nmake
nmake install
```

Have fun!

Installation on Windows XP with Strawberry Perl

Strawberry Perl is a Perl distribution for Windows that most closely mimics a Perl installation under Linux. It comes with gcc compiler, dmake and the other relevant tools included.

Lab::VISA should in principle install out of the box with just the command

```
cpan Lab::VISA
```

executed on the commandline. Unfortunately there is a bug in ExtUtils::MakeMaker (see here) that prevents this. Two possible workarounds are explained below.

Have Windows and Strawberry Perl installed

Install NI-VISA

Download 361mb file `visa462full.exe` from NI's website

Install only 'Run Time Support' (I chose all items below that; it's not much)

Locate msc version of `visa32.lib` and `visa.h` and adjust `Makefile.PL`. This is what worked for me:

```
'LIBS' => q("-IC:/Programme/IVI□Foundation/VISA/WinNT/lib/msc/
           visa32.lib"),
'INC'   => q("-IC:/Programme/IVI□Foundation/VISA/WinNT/include")
,
```

Work around the bug (known to be present in ExtUtils-MakeMaker-6.56)

Option 1: Patch ExtUtils::MakeMaker

Apply the change described at <https://rt.cpan.org/Ticket/Display.html?id=49026> to the file `Kid.pm` of your installation of ExtUtils::MakeMaker

```
perl Makefile.PL
```

Option 2: Edit generated Makefile

If you don't like to modify the installed version of ExtUtils::MakeMaker, you can edit the generated Makefile. These changes will be lost after executing `perl Makefile.PL` again though. This option is recommended if you just want to install Lab::VISA.

```
perl Makefile.PL
```

In the generated file `Makefile`:

Find the two lines containing the words `EXTRALIBS` and `LDLOADLIBS`. Add the `"C:\path\to\visa32.lib"` to each of these lines. On my system they read:

```
EXTRALIBS = "C:\Programme\IVI_Foundation\VISA\WinNT\lib\  
msc\visa32.lib" C:\strawberry\c\lib\libmoldname.a ...  
LDLOADLIBS = "C:\Programme\IVI_Foundation\VISA\WinNT\lib\  
msc\visa32.lib" C:\strawberry\c\lib\libmoldname.a ...
```

```
dmake
```

```
dmake install
```

Installation on Linux

As a Linux user you will probably be able to figure out things yourself. Here is a rough outline:

Before you start, you must have the VISA library by National Instrument installed. If you plan to use GPIB connections (which is very likely), you must also have the necessary drivers (NI-488.2) for your GPIB adapter card installed. Refer to National Instruments' very good documentation for additional information:

<http://digital.ni.com/softlib.nsf/webcategories/85256410006C055586256BBB002C0E91?opendoc>

In file `Makefile.PL` adapt the `LIBS` and `INC` settings according to your installation. This is what worked for me:

```
'LIBS'      => ['-lvisa'],  
'INC'       => '-I/usr/local/vxipnp/linux/include/',
```

Then do the usual

```
perl Makefile.PL
make
make install
```

Testing the installation

Here is a quick test program that you can run with `perl -Mlib test.pl`:

```
#!/usr/bin/perl

use Lab::VISA;

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

printf "status: %x (%s)\n", $status, (($status == $Lab::VISA::
    VI_SUCCESS) ? "success" : "no success");
print "sesn: $sesn\n";

my ($status, $findList, $retcnt, $instrDesc) = Lab::VISA::viFindRsrc
    ($sesn, "ASRL1::INSTR");

printf "status: %x (%s)\n", $status, (($status == $Lab::VISA::
    VI_SUCCESS) ? "success" : "no success");
print "findList: $findList\n";
print "retcnt: $retcnt\n";
print "instrDesc: $instrDesc\n";

__END__
```

COPYRIGHT AND LICENCE

(c) 2010,2011 Daniel Schröer, Andreas K. Hüttel, Daniela Taubert, and others. 2012
Andreas K. Hüttel

2 *The Lab::VISA package*

2.2 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.

2 *The Lab::VISA package*