A guide to oskar

a.deller@ucl.ac.uk

10th June 2016

Contents

1 Introduction, Acknowledgements, and Excuses.

oskar [1] is a collection of tools written using LabVIEW 2013 [2]. It is designed to assist in coordinating hardware control and to provide a framework for acquired data using HDF5 Hierarchical Data Format (HDF)[3]. Essentially, it is a tool for running experiments. It was designed with a focus on: extensibility (it should be reasonably easy to add new control/ DAQ hardware); on sensibly structuring acquired data; and explicitly pairing data with the metadata (attributes) needed to describe it.

The code was written by Adam Deller at University College London for experiments in laser spectroscopy of positronium. Some features are conceptually based on a program he developed at Swansea University (openseq), which was originally inspired by a similar system invented by Aled Isaac. Many

[1] Tenuous acronym: Orchestrate Sequences. Keep Attribute Record
[2] LabVIEW is a registered trademark of National Instruments.
[3] HDF is a trademark of the HDF group.

of the features of openseq were hashed out with feedback and help from Tim Mortensen. Much of the testing of oskar was done by Ben Cooper and Alberto Munoz Alonso. HDF5 support is provided by the h5labview library developed by Martijn Jasperse.

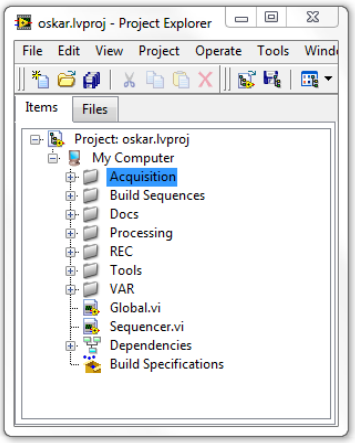
To make use of oskar some experience writing in G (Labview code) is essential. The project is useless without control hardware and data acquisition devices, however, no drivers have been included. You will need to interface any drivers (e.g., using LINX by LabVIEW MakerHub to control an Arduino) with oskar yourself – fortunately, this is a fairly simple process and examples are included. In the data acquisition examples crudely simulated oscilloscopes and cameras are used. Once you have successfully interfaced the drivers you need, sequencer.vi solves the problem of covering large parameter spaces systematically and flexibly. There is very little going on under the hood (see section 3.1 for an overview, or appendix A for a simplistic pseudo-code version), nonetheless you will need a decent grasp of LabVIEW to be able to inspect the code and modify it to suit your needs.

Also, please bear-in-mind that I am not a professional LabVIEW developer and none of the VIs are necessarily examples of good coding practice. Though I've found oskar is generally very stable and easy to use (once you know the basics), be sure to test everything thoroughly for yourself and be vigilant for unexpected behaviour!

2 Installation

oskar requires LabVIEW 2013 (or higher) Full Development Edition (written using 32-bit version on Windows 7 64-bit). The DAQ example fake_camera.vi needs the NI Vision Development Module.

- Install HDF5. Select 32-bit/ 64-bit version to match LabVIEW install. Reboot.
- Download the latest version of the h5labview library and install it using the using JKI VI package manager.
- Download and unzip oskar. Open oskar.lvproj.



Recommended:

- NI VISA (hardware drivers)
- HDFView (for opening and exploring HDF5 files).

There are many freely available libraries for importing hdf5 data for analysis, e.g., h5py for python.

3 The Sequencer and VARs

3.1 Sequencer.vi
- a tool for conducting experimental series

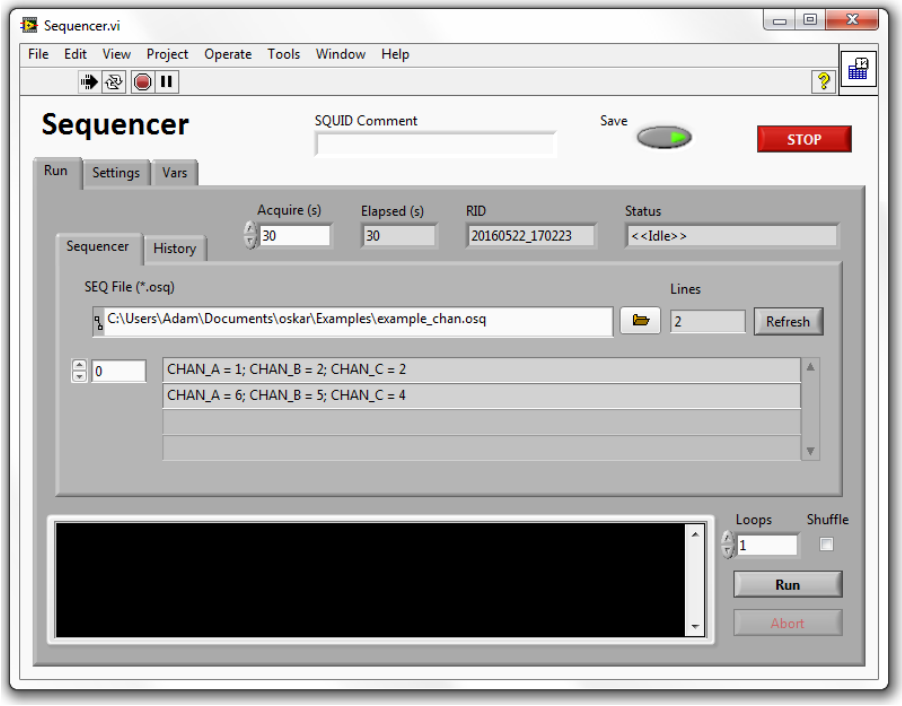
3.1.1 History

The openseq version of sequencer.vi was written to parse configuration files for digital output from an FPGA and tables for an analogue output device, which together were used to control a series of

electrodes for a positron trap[4]. The program acted as a bridge between the simple text files and device drivers, providing a way to automatically run series of experiments where timings and voltages could be systematically varied. Later, a simple system for control of ancillary devices (e.g. a waveform generator) was integrated into the sequencer using VARS. This VARS system was designed to flexibly accommodate any number of additional devices without requiring modification to the sequencer.vi code (see below), and extended its power to control of devices across a network. The sequencer.vi is most powerful when used in combination with modder.vi. The latter was written to automatically generate thousands of sequences for investigating *very* large parameter spaces.

As well as conducting control of hardware, the sequencer is used as a hub for a data acquisition framework based on the HDF5 format. By generating unique run IDs (RID) and hdf5 files for every series of experiments it executes, the sequencer can coordinate programs written for data acquisition (e.g. from cameras/ oscilloscopes) and organise the recording of data files into a structure suitable for batch analysis.

This simplified version of sequencer.vi included in oskar has been written for generically conducting experiments/ data acquisition with control over many variables. No hardware drivers are included, however, the VARs and data acquisition system can be rapidly adapted to control/ coordinate fairly complex experiments.



3.1.2 Overview

The basic premise of sequencer.vi is to read experimental scripts, i.e., sequence files (see below), which describe how parameters should be changed during a measurement series. The program then ensures that the requested conditions (VARS) are met at the right time, and that any data acquisition software is informed of how to save data so that it can be mapped back to these conditions. sequencer.vi is effectively the *conductor* of an experimental measurement and does not perform the actual hardware communication (which is delegated to individual VAR vi's). This modulat approach makes extending control to new instruments relatively painless.

3.1.3 Sequence files

The sequence file (*.osq) provides a means to script a simple experiment. Every *line* contains a single configuration of the experimental variables that one wishes to set, and recorded data is mapped to these using a sequence ID (SQUID) that iterates as each configuration is executed in turn.

The osq files are stored as xml and each *line* is an element of an array called *SEQs*. Each of these elements are clusters that, in the simplest case (see section B for possible extensions), contain only a

[4] J. Clarke *et al.* (2006), *Rev. Sci. Instrum.* **77**, 063302.

string called *Comment* and an array called *VAR Array*. The former does nothing but provide a way to add a comment to each line; the latter is a two column string array: the first column contains every VAR name relevant to a given experiment and the second each corresponding value. This cluster is called a sequence *line*, each of which correspond to one measurement configuration of variables. A series of lines make up the sequence (i.e., a collection of measurement conditions), the execution of which constitutes an experimental RUN. As a simple example, use Editor.vi to open and view the file *./Examples/example_seqs.osq*. This sequence contains just two lines, each with a single VAR called PAUSE, with a value set to 1000 then 2000.

A sequence file is loaded into sequencer.vi by entering its file path into *SEQ File (*.osq)* on the *Run/Sequencer* tab. Loading a file will automatically count the number of *lines* and will indicate the names and values of VARs in the series; this list is updated by pressing *Refresh*. The parameter *Acquire* should be set to the amount of time that data will be recorded for per line before proceeding to the next. *Loops* sets the number of times that the sequencer should loop over the whole series.

When *Run* is pressed (with an osq file successfully loaded) a time-stamp is used to create a RID (%Y%m%d_%H%M%S) that uniquely identifies the measurement series. The sequencer then reads the XML file and proceeds by demanding that the desired experimental conditions of the first line (SQUID = 1) are met, after which it waits for the acquisition period to complete (i.e. *Elapsed* > *Acquire*), then continues to the following line (SQUID = 2), and so forth until all lines have been run for all requested loops. For details of how VARs are met, see section 3.2. If *shuffle* is selected when run is pressed then the lines are randomly reordered at the start of every loop, thus successive loops will be ordered differently.

The 'History' Tab contains a list of all sequence file names and the times which they were run. You can cycle *SEQ File (*.osq)* over previously ran files by hovering over it and scrolling your mouse wheel (press 'Refresh' to reload the sequence).

3.1.4 Saving

The folder './[YYYY]/[MM]/[DD]/[RID]/', wherein a copy of the osq and the '[RID]_raw.h5' file relating to a given run are recorded, is built during initialisation by date_path_h5.vi, using the RID and base(s) specified in the 'Save Base' array (*Settings* tab of sequencer.vi). Specifying more than one base can be used to duplicate data recording. This HDF5 file (or files) is that which data acquisition programs are directed to for recording data; the path is distributed by sequencer.vi using the global variable *H5*.

When a run is started it will raise a dialogue that allows the user to record information regarding the "Author" and "Description". This information is written to attributes of the data file. Details written here often prove very useful during subsequent analysis.

3.2 VARs
- controlling hardware with the sequencer

3.2.1 The principle

The VARs system is used to control hardware with the sequencer. To initialize every line of a sequence file, sequencer.vi becomes a VARs server – effectively broadcasting the message: *I don't care who you are or how you do it but I need each VAR from this list set to these values. Please let me know when it's done!* More explicitly, for each line of a sequence the *VAR Array*, e.g.

VAR array:

Name	*Value*
VAR1	value_1
VAR2	value_2
⋮	
VARn	value_n

is read and for each row of this array two Boolean columns are added: *Received* and *Set* (both initially set to False), as well as the string *Options* (see below). This new array is passed to a global variable and can be viewed on the tab *Vars* in sequencer.vi. For each row the sequencer waits for *Received* to become True, then proceeds to the next row. After all VARs have been received the sequencer waits for all *Set* values to be True (i.e. for every row of the array/ each requested VAR), before continuing to the acquisition stage.

Separate programs with control over the hardware corresponding to each VAR must be written which "listen" for requests from the sequencer. The simple example of *PAUSE* can be found in the VARs folder.

This vi listens for a VAR called PAUSE. Upon receiving the VAR from the sequencer it sets *Received* in the global variable to true, then feeds the VAR value into a wait function (in ms). After waiting for the requested amount of time it updates *Set* to true. A similar example called *HOLD* updates *Set* when the user presses the button on its front panel (the VARs value is irrelevant). Use these examples as a template for making new VAR controls, replacing the code in the centre section with a subVI that can take the VAR value as an input, e.g. to set the current for a power supply/ wavelength of a laser/ frequency of a waveform/ etc. Use an output of the hardware control subVI to ensure the sequencer does not proceed until the desired hardware change has been completed. Note: any VAR hardware control programs should be added to the oskar project in order to have access to the global variables.

Options add flexibility to VAR control and can be used (optionally) by appending the VAR name with an underscore, e.g. VAR_OPTION (for this reason, *only* use underscores in VAR names for specifying options). The sequencer splits the name at the underscore and transmits the name and option separately as individual strings. It's up to the VAR vi to act on the option as appropriate. For example, you could create a VAR control that can set the voltages on a channel of an AO device with VAR name 'VOL'. The option string could be interpreted as which channel to control. For this example

VAR array:

Name	*Value*
VOL_A	5
VOL_B	10

the vi would first set channel A to 5 V then channel B to 10 V. This is generally better than having separate VAR vi's for each channel as it forces the order they are set and prevents interference from simultaneous attempts to communicate with one device. The options format can be used quite efficiently even with many channels, e.g., VAR_1/2/3 could be used to build an array of AO voltages but only when a specific option is received (e.g., VAR_update) will the commands be sent. The level of sophistication in interpreting the options depends entirely on the VAR vi, and not on the sequencer.

To see the VARs system in action try and run */Examples/example_seqs.osq* with sequencer.vi – be sure to have the VAR control *PAUSE* running! To see how options can be used with VARs try */Examples/example_chan.osq* with *CHANNEL.vi*. When testing VARs you may avoid saving by disabling the *Save* option on the sequencer.vi front panel.

3.2.2 The reality

It's important to understand the technical details for how VARs are actually made in order to do something useful with them. Open the template 'PAUSE' and navigate to the block diagram. There are 5 main sections to this code - namely:

1. Create a notifier based on the control 'Var' (in this case: Var = "PAUSE"). This must match the notifier created by the Sequencer.

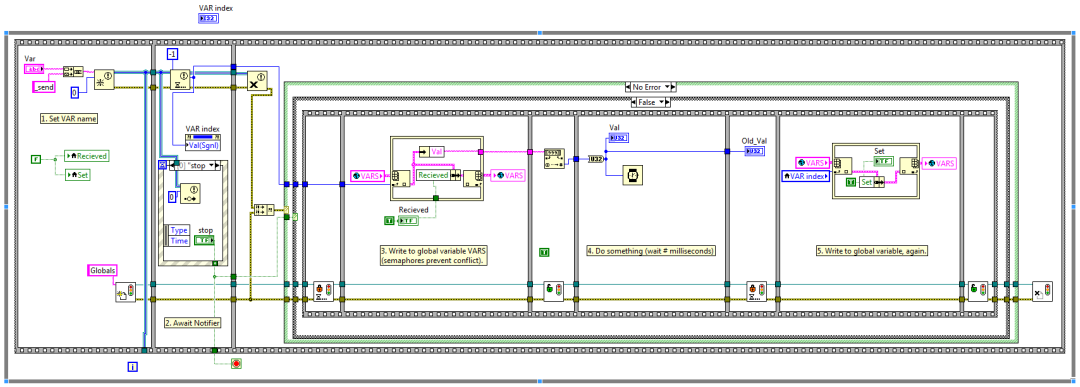
2. Wait for the notifier which, when received, will pass a value indicating the index of the matching VAR in the Global variable 'VARS' (array of clusters).

3. Acquire the semaphore 'Global' to prevent any other VAR vi from writing to the global variables. Use the index obtained in the notifier to select the correct row from VARS. Read the value and write the Boolean 'Received' to True. Release the semaphore.

4. Do something with the VAR value. In this case, convert it to an I32 and feed this to the Wait function. Generically, this is where the vi must somehow interpret the VAR value (passed as a string) then act upon it, for instance, you may pass the value to a subVI that sets the wavelength of a dye laser. Or the delay of a channel in a TTL generator. Or the frequency of a waveform generator. Or the output of a high-voltage supply. Etc., etc., etc.

5. Acquire the semaphore 'Global' then, in the appropriate row of the global variable 'VARS', write the Boolean 'Set' to True. Release the semaphore and wait to receive another notifier.

You can make a copy of PAUSE to use as a template, change the value of VAR (i.e. the name used by the sequencer to communicate with the vi) and write into step 4 the code specific to your own needs. See CHANNEL for a more complicated example that uses options, and which records values set for each channel as a variant. In this way newly received values can be compared to the last value set for that channel, to determine if anything needs to be changed.



4 Writing Sequences

4.1 Editor.vi
- simple sequence writing

The editor provides an easy way to read/ write xml files containing sequences (.osq). Each sequence contains an array, every element of which is a cluster that, in the simplest case, contains a comment field and the VAR_array, e.g.

VAR array:

Name	*Value*
VAR1	2
VAR2	3
VAR3_OPT1	5
VAR3_OPT2	7

Comment:

"a useful comment"

To execute the above sequence line would require 3 VAR vi's to be running and listening for VAR1, VAR2, and VAR3. The VAR3 vi would have to be able to interpret the options OPT1 and OPT2. Every element of the seq must be written by hand, therefore Editor.vi is only really useful for writing very simple seqs, or for editing/ viewing existing .osq files.

4.2 Modder.vi
- automatic sequence building

Modder.vi is designed to automatically write a series of instructions for the sequencer, based on all possible permutations of n different variables each with m_n different values. The program works by performing a string replace on place-holder "TAGs" written into a template sequence, with values from each row of the permutations array.

To write a sequence, first create a template on the *Sequence* tab. The first column should contain the actual names of any VARs which are to be distributed by the sequencer, and the second column should contain the temporary TAG(s) or constant values. The TAG can be any string, so long as it is unique within the XML version of the template sequence. For example, for the VAR "FOO" and TAG "bar", create the template SEQ:

VAR array:

Name	*Value*
FOO	bar

Now navigate to the *Permutations* tab. Fill in a line of the *Build Arrays* list, setting the field *TAG* to: bar. All values you wish to replace bar in the template can be manually entered into the *Values* array, or alternatively you can populate this array automatically using the function *f(x)*. For example, set *f(x)* to x+1; *Steps (m)* to 3; change the *Format* string to integer (%d), then press *Build Array*. Notice that the *Permutations* table now contains 1, 2 and 3 in the first column. Even if you automatically populate the *Values* array, you can manually add/ remove/ randomize elements afterwards. Press *Make*. This will

fill the *Output Seqs* array on the *Sequence* tab, with each element having the template TAG replaced by a value from the permutations array.

The power of modder.vi is evident when using more than one TAG. To see this, create another TAG and watch how the permutations table populates. The *Reorder* buttons lets you randomly order the individual arrays used to make the permutations, or alternatively the final permutation array. This can be useful to scramble any effect of drift with your experiment, which might be misleading if the variables follow a linear series. Keep track of how many permutations exist using the *sizes* indicator; be aware this will increase very quickly as new TAGs are created!

It is not necessary to have a TAG for every VAR as you can write other VARs into the template sequence with their values set to constants. Also, TAGS can be shared between any number of VARs, and can appear as a function if encased between '$math' and '$'. e.g.

VAR array:

Name	*Value*
ALPHA	bar
BETA	$math bar * 2 $

For the example above, *Make* will replace the value for BETA with double that assigned to ALPHA. This can be used to automatically change two variables in a correlated way.

Once a satisfactory output SEQ has been made, it can be saved to an ".osq" file that can be read by the sequencer. Pressing *Save* will also produce an ".omv" file which contains the permutations array and TAG names. This can be used in future to make similar scans with different template sequences, and is loaded using the *Template *.omv file* control. Inspect_omv.vi and Inspect_osq.vi allows you to plot the values saved to an omv/ osq files. This shows the parameter space that your sequence will cover.

Warning:

Avoid very short TAGs (e.g. 'x') as these are unlikely to be unique within the template XML. A simple system that prepends the name of the VAR it will be used with can be helpful, e.g.

VAR array:

Name	*Value*
HV	XHV

This will also make it easier to guess what the TAG was used for when inspecting omv files.

4.3 Inject.vi, Append.vi, and Combine.vi
- adding lines to sequences

Modder.vi offers a very powerful way to build sequences but it is only capable of producing rectangular permutations of the TAGs given it. Sometimes it is convenient to make an occasional measurement with a certain set of values, e.g., to periodically measure a background during a scan. This cannot be done with Modder.vi directly, however, Inject.vi can be used instead to add lines to .osq files at specified positions.

Load a sequence by pressing the file button on the 'Open' path control. This will populate the 'SEQs' array. Next, write the line you want to inject into this array in the 'Injection' field, and fill in the 'Positions' array with those where you wish the lines to go. Now press 'Inject'; scroll through 'SEQs' to check it worked. Save as a new .osq file (or overwrite the original).

To concatenate one sequence onto the back of another use 'Append.vi', which is operated in a similar manner to 'Inject.vi'. Use 'Open' to load files into the left-hand SEQ. Pressing 'Append' copies the lines in the left-side SEQs to the right, or appends them to those already there. Repeat as many times as needed to build your sequence. 'Save' outputs the right-hand SEQ to file. For more advanced options with merging multiple sequences use Combine.vi. This vi takes a list of sequence files and joins together the lines from each, with the option of repeating the individual files n times within the output sequence. With 'shuffle' enabled all of the lines are randomised before the output is saved.

4.4 Filter.vi
- removing lines from sequences

Filter.vi provides another way to adjust the permutations generated by modder.vi, by dropping lines according to simple logic functions.

For instance, if you want to vary the bias applied to two electrodes (E1, E2) you can simply use Modder to create an osq file with two TAGs varied over the desired ranges. This will generate a 2d array of EVERY possible combination of voltages, however, some of the permutations may not be useful, e.g. if E1 one must always be higher than E2 for electrons to reach a detector. In this case, load the full .osq built by Modder.vi into Filter.vi and apply a filter that retains only the useful permutations, e.g., E1 > E2, then save the output as a new sequence file.

5 Data Acquisition

5.1 Integrating Data Acquisition with the Sequencer

The process of marrying experimental control with data acquisition is very simple. There are two methods: i) add an attribute to the HDF5 group for a given squid, which can be used to note simple single-value measurements, e.g., laser power/ lab temperature; and ii) recording of datasets, e.g., oscilloscope waveforms and camera images. In both cases semaphores are used to prevent read/ write conflicts of the single HDF5 data file that each run produces.

5.1.1 SQUID Attributes

Often measurements from pressure gauges or power supplies are useful for verifying if interesting "signals" are real, or just arise from drifting experimental conditions. However, lots of precise repeat measurements of the conditions might be overkill and wasteful, and a record of a single value, or the mean value, for each squid is often sufficient for monitoring. If this is the case, such a measurement can be recorded into the attributes (meta-data) for a squid group using the global variable 'Rec'.

'Rec' is a global variant that is wiped at the start of every squid. At the end of a squid any values written to the variant during the acquisition period (using 'Set variant attribute') will be appended to the attributes with the prefix *REC:*. See the example rec_random.vi, which continuously generates a random number every 5 seconds. Attributes with the same name are overwritten, thus the last value to be written during a given squid will be the one that is ultimately stored. Arbitrarily many measurements can be added to the squid attributes in this way, so long as each has an unique name.

5.1.2 Datasets

Most scientific data will take some form of regularly sized array, such as oscilloscope waveforms, camera images, etc. Each squid might contain many repeat measurements from the same device. In general, to record such data a VI is needed to continuously acquire data, then by reading the following global variables (written to by sequencer.vi):

Save
: Boolean. Is save selected in Sequencer.vi?

Acquire
: Boolean. Is the Sequencer.vi state currently 'acquiring data' (not, e.g., setting VARs)?

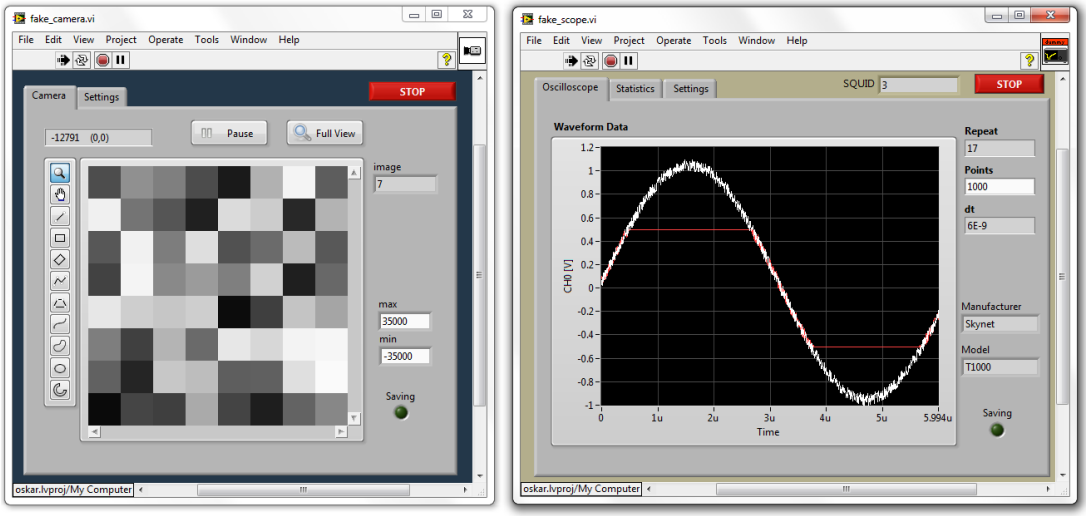
H5
: An array of paths to hdf5 file(s). Multiple entries if sequencer.vi has several lines to *Settings/Save Base* (for backing data up).

SQUID
: Integer (as a string). The iterating number that uniquely identifies the current line of the running sequence.

it's relatively easy to coordinate the saving of data in accordance with the state of the sequencer.

For instance, you likely only wish to record data during the acquisition periods of a running SEQ, therefore in your data-acquisition VI place an appropriately configured "h5_[name].vi" subVI inside a case structure controlled by Save AND Running. See 'h5_scope.vi' for example. This VI packages the relevant meta-data (channel gain, time-base, etc.) into variant attributes and passes both this and the actual data to 'H5_1D_SGL.vi', which then writes the data to the H5 files in a dataset called OSC_# (# is replaced by the channel number), inside of the SQUID subgroup. Something equivalent to "h5_scope.vi" could be made for each data-acquisition device, written to appropriately package the meta-data attributes.

LabVIEW semaphores are used to prevent more than one vi writing to the hdf5 file at the same time. The fake_scope.vi illustrates how this is all achieved. fake_camera.vi is an even simpler example that just uses the generic h5_2D_stacking.vi to save directly.



The format for saving data depends on its shape (e.g. 1D array, 2D image) and data type (SGL, DBL, LONG, etc.). Examples for 1D_SGl, 1D_DBL, 2D_I32, can be found in the ./tools/hdf5 virtual folder. These vi's expect repeat measurements to be made within a squid, for instance, when the 1D_SGL is asked to write data to a pre-existing dataset, it will be appended as a new row. The example "h5_stats.vi" demonstrates how cluster data can be written to a dataset. This might be used when, rather than a time series, a range of values (possibly of differing dtypes) are recorded at once. This DataFrame-like structure is most likely useful if analysis is performed in LabVIEW to determine, e.g. trigger events (time, amplitude, width), or statistical features (mean, max, min, range, FWHM).

Because VAR conditions are stored in each squid's attributes, and as data relating to a squid is stored within its subgroup, it's trivial to map data to the relevant VAR conditions. This assumes of course that data is recorded correctly, however, the individual data acquisition vi's are responsible for ensuring this (not the sequencer) and it's technically possible to get wrong.

Warning:

oskar is designed for experiments where many (>10) repeat measurements are recorded at a rate of 10 Hz to 0.3 Hz, then an experimental parameter is varied and more measurements are similarly made. For lower acquisition rates (e.g., long camera exposure times) you must ensure that data taken near the boundary of two SQUIDS is mapped to the correct one, or safely ignored if ambiguous. This can be done quite easily. For instance, by checking the SQUID global variable has the same value both at the start of an exposure and the end. Alternatively, the global variable EOS (end-of-squid) can be used to synchronise the sequencer to some form of data acquisition or hardware trigger. See 'Tools/skip_squid.vi'. By using EOS with the acquisition time set to be unattainably long, a data acquisition device can effectively control how long the sequencer acquires data for (e.g., until 10 waveforms have been recorded).

5.2 Reviewing Data

If everything is configured correctly a given run will produce a file called [RID]_raw.h5, which is saved to the folder(s) '[Base(s)]/[YYYY]/[MM]/[DD]/[RID]/'. Opening this file using HDFView will reveal a structure akin to:

- RID :
  - 1
    - IMG
    - OSC_0
    - OSC_1
  - 2

↳IMG
⋮

The top level group is the RUN and has attributes, RID, SEQ FILE, AUTHOR, and DESCRIPTION. Each group within the RUN represents a SQUID value. This has attributes DATETIME, START, END, ACQUIRE, and all appropriate VAR settings in the form, VAR:NAME = VALUE. REC data is similarly recorded. Each SQUID group contains all of the datasets that are associated with it (OSC_0, OSC_1 and IMG in the example above), which could represent repeat measurements of 1D/ 2D data from oscilloscope channels or a camera. Datasets can have their own attributes, which can be used to record, e.g., exposure time, amplifier gain, etc. Note: the SQUID group names are strings. This is important if you wish to access a given group using, e.g., h5py.

The very structured format of HDF5 makes batch analysis a breeze. The ability to attach meta-data makes it trivial to associate measurements with experimental conditions and instrument settings. As HDF5 stores data in binary it's an efficient use of HDD space for whatever precision data you require, and it can be read quickly for analysis too. It's even possible to compress data within an hdf5 file, although that has not been implemented here. See h5labview docs for more info.

A Sequencer.vi pseudo-code

The following is simplified pseudo-code illustrating the basic format of what happens when *Run* is pressed in sequencer.vi. The syntax is vaguely based on python. The red numbered lines indicate frames within stacked sequences.

```
[Event: Run Value Change]
[0] pre-initialisation
        global.Run = True
        global.Save = Save
        global.Acquire = False
[1] initilise run
        #Generate RID from timestamp
        RID = time.now(YYYYmmdd_HHMMSS)
        global.RID = RID
        #Build path to data folder
        global.H5 = [‘base\YY\MM\dd\RID’ for base in Save_Base]
        #Read osq file
        seq = read_file(SEQ_File)
[2] run seq
        loop = 1
        squid = 1
        while loop < Loops:
                if Shuffle:
                        #re-order seq lines
                        seq = shuffle_seqs(seq)
                for line in seq:
                        VAR_arr = line.var_array
                        [2.0] initialise SQUID
                        global.SQUID = str(squid)
                        H5[squid].write_attributes(VAR_arr)
                        #create VAR string
                        var_str = var_to_str(VAR_arr)
                        global.SEQLine = var_str
                        print(time.now() + var_str)
                        #get each var name, value and options
                        VARS = get_vars(VAR_arr)
                        [2.1] set VARS
                        #wait if pause is true
                        while Pause and not Abort:
                                wait(200 ms)
                        #distribute VARS
```

```
                global.VARS = VARS
                for var in VARS:
                        #send notifier until recieved
                        while not global.VARS[var].recieved:
                                send_notifier(var.name + "_send")
                                wait(100 ms)
                #wait for all vars to be set
                while not [global.VARS.var.set for var in VARS].all():
                        wait(100 ms)
                [2.2] acquire data
                global.Rec = None
                global.Acquire = True
                start = time.now()
                #wait for acquisition
                while elapsed < acquire:
                        end = time.now()
                        elapsed = end - start
                        wait(200 ms)
                global.Acquire = False
                H5[squid].write_attributes([start, end, elapsed, global.Rec])
                [2.3] Reset
                for var in VARS:
                        var.set = False
                        var.recieved = False
                #increment squid
                squid = squid + 1
        #increment loop
        loop = loop + 1
```

B Extending the Sequencer

It would be pretty difficult to run the experiment for which oskar's predecessor was originally designed using this version of it. This is because that system centred around an FPGA and AO device, with dozens of outputs that would be a nightmare to individually configure using VARS. The original solution was to have custom written controls for each SEQ line to configure these devices, encoded as voltages and DO in an array of clusters.

In principle more controls, such as the above, could be added to each SEQ line cluster, e.g.

VAR array:

Name	*Value*
RW_A	5
RW_F	9.1M

TRAP:

Step	*Time*	*AO*	DO
Accumulate	1	10, 8, 7, 6.5, 50	0, 0, 1, 0, 0, 0
Compress	10m	50, 8, 7, 6.5, 50	0, 1, 1, 0, 0, 0
Eject	100n	10, 8, 7, 6.5, 50	0, 0, 0, 0, 0, 1

Comment:

"a useful comment"

and unbundled from it at the start of each line, then processed to appropriately configure the devices. The quickest way to add an element is using the type-definition Tools/SEQ/Line.ctl; when you 'Save' and 'Apply Changes' these will propagate to the sequence editing vi's (editor, modder, etc).

Adding more elements would require adaptation of the sequencer and read/ write osq VIs. In principle, as long as a string version of the control can be read, modder.vi will be able to write TAG values into the control (e.g. to edit a specific AO/ time-step). Although not an insignificant amount of work, this

would not be too difficult for someone with some reasonable skill with LabVIEW.

Warning:

Unlike creating new VARS, changing the structure of the osq cluster inevitably breaks many of the VIs which read/ write and manipulate these (XML) files. Fortunately, this is not too difficult to fix, if you know what you're doing.

In the interest of minimising dependencies, several useful features have not been implemented in this version of oskar (they would not be too hard to add yourself). These include support for transmitting run information to a MySQL database (LabVIEW Database Connectivity Toolkit and ODBC connector), network VARs (using Datasockets), automatic execution of analysis/ averaging scripts (System Exec.vi).