VELA Hardware Controllers:

A Guide

**Duncan Scott, Alex Brynes**

**and Matthew Toplis**

**March 2016**

**Abstract**

A set of c++ / Python controllers has been written for various pieces of hardware on VELA. The aim is to provide a self-consistent framework with a single-point of entry that will make it easier to monitor/control machine parameters, to provide automated sequencing, data acquisition and analysis to develop ‘*any conceivable high level* *application*.’ The functions provided to the end-user (client-side) interface are presented. For a more comprehensive guide to the philosophy behind the development of these controllers, and more information on how they are structured, see [1]. The c++ version of functions is presented here with the type of each parameter. Each hardware controller is also available to import into Python 2.7 with the same functionality. This report should be considered a ‘*living document*’ and it is hoped it will be continually updated and revised as improvements and more hardware is included within the project.

# Introduction

This document will give an overview of the c++ version of the *VELA* *Hardware Controllers* API. For an introduction to the API please see [1].

# Set-up

The project source currently lives at:

[\\fed.cclrc.ac.uk\org\NLab\ASTeC\Projects\VELA\Software\c++](file:///\\fed.cclrc.ac.uk\org\NLab\ASTeC\Projects\VELA\Software\c++)

It is planned to move to a repository at a later date

# Usage

Currently the source code should be included with your project and compiled with it. To include control of hardware type *X* all that should be required is toinstantiate a “*vela*X*Controller”* object and then call methods on that object. The following will give an overview of each type of hardware controller and public member functions. In general the aim is to make the API *self-documenting* by using simple human readable function and variable names that should directly correlate to the *usual* names for things on VELA . In its current state the code is not, well-used, bug-tested or checked for internal consistency. If you spot any errors, omissions or inconsistencies please report them. The folder tree structure, starting from the c++ root folder is shown in Figure 1.

# ENUMS

Enumerated types are used throughout the API to define states. At a later date these types may all go in the same namespace, but currently they are spread between a ‘global’ namespace VELA\_ENUM in ‘*velaStructs.h*’ and individual hardware namespaces, e.g. ‘*velaXStructs.h*’ where *X* is the item of hardware. Functions that return enum states should always be available in two flavours, those that return an integer of the value and those that return a string of the value.

# Configuration Files

Generic and “offline” information about hardware is contained in the config files. They define a lot of the information required to use this API, for example object names, data types, which EPICS process variables (PVs) are monitored etc. Config files use keywords that are defined in configDefinitions.h in the Utilities folder. It is hoped that helper functions, such as getObjectNames() will be provided for all hardware types to allow full use of the API without too much reference to the config file.

Figure : Tree structure for the c++ API

# Generic Functions for all Controllers

Due to the class inheritance structure there are functions that every controller type has access to.

## Messages

Two types of messages can be enabled, messages and debug-messages. They send text via std::cout to the command line.

|  |  |  |  |
| --- | --- | --- | --- |
| **void** | debugMessagesOn | () |  |
| **void** | debugMessagesOff | () |
| **void** | messagesOn | () |
| **void** | messagesOff | () |
| **void** | silence | () | Turn all messages off |
| **void** | verbose | () | Turn all messages on |

## Generic Interlocks

Not every EPICS PV has an interlock but for those that do (as defined in the config files), each controller will be able to display the state of each generic interlock. Some specific hardware types, such as the RF modulator have additional specific interlock functions. Each interlock and state is represented by enums defined in the VELA\_ENUM namespace.

|  |  |
| --- | --- |
| **VELA\_ENUM::ILOCK\_NUMBER** | ILOCK\_1, ILOCK\_2, ILOCK\_3, ILOCK\_4, ILOCK\_5, ILOCK\_6, ILOCK\_7, ILOCK\_8, ILOCK\_9, ILOCK\_10, ILOCK\_11, ILOCK\_12, ILOCK\_13, ILOCK\_14, ILOCK\_15, ILOCK\_16, ILOCK\_17, ILOCK\_18, ILOCK\_19, ILOCK\_20, ILOCK\_21, ILOCK\_22, ILOCK\_23, ILOCK\_24, ILOCK\_ERR |
| **VELA\_ENUM::ILOCK\_** | ILOCK\_BAD, ILOCK\_GOOD, ILOCK\_ERROR |

|  |  |  |
| --- | --- | --- |
| **std::map**< VELA\_ENUM::ILOCK\_NUMBER, VELA\_ENUM::ILOCK\_STATE > | getILockStates | **(const std::string** & name**)** |
| **std::map<** VELA\_ENUM::ILOCK\_NUMBER, std::string **>** | getILockStatesStr | **(const std::string** & name**)** |

* Returns a map of the set of interlocks and their corresponding state for object name. (Not all objects will have interlocks).

# VELA BPMs

The BPM controller currently provides functions to:

* Set and read the attenuations and delays on each BPM.
* Get the X/Y values from EPICS.
* Monitor the raw data for a given number of shots.
* Calculate the X and Y values, and the charge data, for each of these shots **(** see [ 2 ] for these algorithms **)**.
* Track charge and change SA1/SA2 appropriately **(** see [ 3 ] for this algorithm **)**.

The config file defines, for each BPM:

* BPM PV roots and object names
  + “BPM01”, … “BPM06”, “BA1-BPM01”, “BA1-BPM02” to be given as arguments in the functions below
* PVs for SA1/2, RA1/2, SD1/2, RD1/2, X, Y and DATA:B2V.VALA
* Calibrated SA1/2 and calibration voltages and charge
  + Required for recalibration procedure, see below
* Pickup mechanical zero offset, and x/y offsets
  + required for orbit procedures

## Names

|  |  |  |
| --- | --- | --- |
| **std::vector<** **std::string** **>** | getBPMNames | () |

* Returns a list of the names of BPMs given in the config file, to be used as arguments in other BPM functions.

## Data Monitoring

When the BPM controller is instantiated, the user has access to the “Getters” (see below), so that X, Y, Q, etc., can be returned. Only when using the monitorDataForNShots function will the controller fill up vectors of BPM data. After the monitoring is finished, the controller does not return to continuous monitoring (although this may be implemented in future). The controller will always be calculating X, Y and Q on a shot-to-shot basis.

|  |  |  |  |
| --- | --- | --- | --- |
| **void** | monitorDataForNShots | (size\_t N, const std::string & name) | Monitors raw BPM voltages on BPM name for a N shots |

## Data Acquisition

The following functions access the data saved after a call to monitorDataForNShots:

|  |  |  |  |
| --- | --- | --- | --- |
| **std::vector<** **double** **>** | getXVec | (const std::string & name) | Returns a vector of the shot-to-shot X / Y position, or charge, Q, for BPM name calculated from the raw data acquired in monitorDataForNShots |
| **std::vector<** **double** **>** | getYVec | (const std::string & name) |
| **std::vector<** **double** **>** | getQVec | (const std::string & name) |
| **std::vector<** **std::vector<** **double** **>>** | getBPMRawData | (const std::string & name) | Returns the raw data |
| **std::vector<** **double** **>** | getTimeStamps | (const std::string & name) | Returns the timestamps of the raw data |
| **std::vector<** **std::string** **>** | getStrTimeStamps | (const std::string & name) |

This function determines the correct attenuation setting for the charge seen at one of the VELA charge monitors. If this function is always running in the background (say, every 100 shots or so), then the attenuation settings for all BPMs in the injector hall should have the correct attenuation. This function requires a value for the charge, which can be obtained using the VELA Scope Controller. This **qScope** value (e.g. from the WCM) can then be passed into this function to recalibrate a given BPM.

|  |  |  |
| --- | --- | --- |
| **void** | reCalAtt | (const std::string & bpmName, double qScope) |

## General Getters and Setters

These functions are used to get and set different BPM settings.

|  |  |  |  |
| --- | --- | --- | --- |
| **void** | setSA1 | (const std::string name, long val) | Sets attenuation for name, to val, in the range 0 – 20. |
| **void** | setSA2 | (const std::string name, long val) |
| **long** | getRA1 | (const std::string name) | Returns read attenuation value for name |
| **long** | getRA2 | (const std::string name) |
| **void** | setSD1 | (const std::string name, long val) | Set delay for name to val, in the range 0 – 511. |
| **void** | setSD2 | (const std::string name, long val) |
| **long** | getRD1 | (const std::string name) | Returns read delay parameters for name |
| **long** | getRD2 | (const std::string name) |
| **double** | getXFromPV | (const std::string name) | Returns beam position from dedicated EPICS PV |
| **double** | getYFromPV | (const std::string name) |
| **double** | getX | (const std::string name) | Returns beam position using the algorithms in [ 2 ]. The X and Y PVs for each BPM neglect to take account of certain offsets, etc. which this calculation does not. |
| **double** | getY | (const std::string name) |
| **double** | getQ | (const std::string name) | Returns the charge as measured by name, calibration factors defined in the config file. {REF} |

# VELA Q-Scope

The Q-Scope is the oscilloscope typically used to measure the bunch charge from the wall current monitor, Faraday Cups and ICTs. There are currently two Python scripts which can be used to log measurements from the LeCroy scope on velascope02 to EPICS **(**both located at velascope02\Desktop\ and \\fed.cclrc.ac.uk\Org\NLab\ASTeC\Projects\VELA\Software\Python\LeCroy**)** – the first, **scopePVals.py** monitors the measured P values as per the measurement set up on the scope; the second, **scopeTraces.py**, returns the zoomed-in traces from each of the four scope channels. Both of these scripts should not be run simultaneously, as this causes measurements to be missed. In order to take more accurate measurements, **scopeTraces.py** should be running at all times on velascope02. These scripts are prone to crashing if the trace goes off the range of the scope, so if you are not getting any sensible charge readings, try restarting the script from the command line. The scope is currently set up so that the regions of interest on channels C1, C2 and C3 correspond to the WCM, ICT1 and ED-FCUP. This setup is saved as Setup 2 on the scope **(**30-Oct-2015 13:03:42**)**. As we try out more setups, these will be saved in a database, and eventually, we hope to be able to control the scope remotely.

The velaScopeController allows the user to:

* Get the P1 – 4 values for each channel **(** if scopePVals.py is running **)**.
* Monitor the P1 – 4 numbers for a number of shots **(** if scopePVals.py is running **)**.
* Monitor the **(**zoomed-in**)** raw traces from each channel for a given number of shots.
* Use the data generated from the trace monitor to:
  + Find the minimum / maximum of each trace.
  + Calculate the area under the trace.
  + Determine the average noise.
  + Get all the raw trace data for each shot.
  + Get the timestamps **(**as both strings and doubles**)** for each shot.
* Get the charge for a given diagnostic **(** provided that it is specified in the config file **)**.

There are 2 config files, one of which corresponds to the P1 – 4 monitors **(**using **scopePVals.py**, which should not really be used**)**, and one which corresponds to the trace monitors **(**using **scopeTraces.py)**. These specify:

* EPICS PVs for P1 – 4 / TR1 – 4.
* The timebase of the traces.
* Which PV corresponds to which diagnostic **(**so if this is changed, the config file should be altered accordingly and the DLL rebuilt**)**.
* The size of the trace array.

## Local Structs

|  |  |
| --- | --- |
| velaScopeStructs:: SCOPE\_PV\_TYPE | P1, P2, P3, P4, TR1, TR2, TR3, TR4 |

The scope’s P values depend on how each channel of the scope is set up. There are a variety of options regarding what these P values are – they could be the minimum / maximum value of a trace, the area under a trace, etc. Using Setup 2, P1 corresponds to the WCM, P2 corresponds to ICT-1, and P3 corresponds to the ED-FCUP. Currently, the config file for the scope controller defines the channel that each diagnostic device is connected to, so each time the scope setup is changed, the scope controller will have to be rebuilt. This will hopefully be modified soon.

## Names

|  |  |  |
| --- | --- | --- |
| **std::vector<** **std::string** **>** | getScopeNames | **(** **)** |

* Returns a list of all arguments for the scopes – ‘WVFxx’ corresponds to traces, and ‘SCOPxx’ corresponds to the P values. At the minute, there is only one scope used on VELA, with the EPICS name EBT-INJ-SCOPE-01. When CLARA is operational, the number of scopes in use may be increased. So, in order to use the functions below, the user will need to use either **WVF01** or **SCOP01** as the argument corresponding to the scope name, depending on whether traces (waveforms) or P values are being monitored.

## Data Monitoring and Acquisition

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **void** | monitorTracesForNShots | | **(** **size\_t** N **)** | When scopeTraces.py is running fills 4N vectors of raw scope traces, N for each channel. That is, 4 times N vectors. |
| **void** | | monitorNumsForNShots | **(** **size\_t** N **)** | **(**When **scopePVals.py** is running**)** – fills 4 vectors of scope P values, each vector of size N. |

|  |  |  |
| --- | --- | --- |
| **std::vector<std::vector<double>>** | getScopeTrace | **(conststd::string scope, velaScopeStructs::SCOPE\_PV\_TYPE &pvType** **)** |

* Returns all the data generated by monitorTracesForNShots.

|  |  |  |
| --- | --- | --- |
| **std::vector<** **double** **>** | getMinOfTraces | **(** **const std::string** scope , velaScopeStructs::SCOPE\_PV\_TYPE & pvType **)** |
| **std::vector<** **double** **>** | getMaxOfTraces | **(** **const std::string** scope , velaScopeStructs::SCOPE\_PV\_TYPE & pvType **)** |
| **std::vector<** **double** **>** | getAreaUnderTraces | **(** **const std::string** scope , velaScopeStructs::SCOPE\_PV\_TYPE & pvType **)** |

* Uses the raw data generated in monitorTracesForNShots to find the min/max value for the given wave format each shot. There is an example **(** in the link in the Abstract **)** of how to use velaScopeStructs::SCOPE\_PV\_TYPE as an argument.
* The area under the trace is currently calculated using the trapezium rule.

|  |  |  |
| --- | --- | --- |
| **std::vector<** **double >** | getAvgNoise | **(** **const std::string** scope , velaScopeStructs::SCOPE\_PV\_TYPE & pvType **)** |

* NOTE: This function should probably be used with caution – it has been prototyped, but needs further testing. The charge as seen by the ICT is given by the area under the ICT scope trace. As the ICTs are quite noisy **(**by which I mean that the base level of the trace is subject to shot-to-shot fluctuations**)**, it is necessary to subtract the noise from each shot. For now, this function calculates the average noise from a region of the trace which is not expected to see any beam, for each shot.

## Getters

Before **monitorTracesForNShots / monitorNumsForNShots** is called, the controller will not fill up a vector of values, but once these functions are called, vectors of trace / P value data will be filled up. Each of these getters returns the last element of the vector corresponding to the measurement of the charge at a given diagnostic. The functions with names corresponding to diagnostic devices (WCM, ICT etc.) depend on the trace data. They do not require the monitor functions to be called, though.

For example, **getWCMQ** will return the first minimum of the last trace in a vector of traces corresponding to the WCM (multiplied by a calibration factor).

|  |  |  |  |
| --- | --- | --- | --- |
| **double** | getScopeP1 | **(** **const std::string** scope**)** | If scopePVals.py is running, this will return the P values measured on the scope. |
| **double** | getScopeP2 | **(** **const std::string** scope**)** |
| **double** | getScopeP3 | **(** **const std::string** scope**)** |
| **double** | getScopeP4 | **(** **const std::string** scope**)** |
| **double** | getWCMQ | **()** | The config file defines a “diagType” parameter, and so the scope controller can determine which EPICS PV corresponds to which scope channel. This function returns the last value of the **getMinOfTraces** function for the PV corresponding to the WCM, in pC. The functions corresponding to the ICTs use the results from **getAreaUnderTraces** and **getAvgNoise** to return a charge in pC. The diagnostic corresponding to each channel is defined in the config file. |
| **double** | getICT1Q | **()** |
| **double** | getICT2Q | **()** |
| **double** | getFCUPQ | **()** |
| **double** | getEDFCUPQ | **()** |

# VELA Vacuum Valves

The *velaVacValves* controller enables the user to:

* Open / close each vacuum valve
* Determine if a valve is open / closed.
* Get the state of a vacuum valve.
* Get a list of the vacuum valve names defined in the config file.

The generic functions specified above can also be used to determine the state of the interlocks on a given vacuum valve, in the case of it tripping off. This generic controller will not tell you which interlock has caused the valve to close, however.

The config file defines:

* EPICS PVs for each vacuum valve.
* A “State” PV, and “ON”/”OFF” PVs used for opening/closing valves.
* The number of vacuum valve interlocks.

|  |  |
| --- | --- |
| VELA\_ENUM::STATE | VALVE\_CLOSED, VALVE\_OPEN, VALVE\_TIMING, VALVE\_ERROR |

## Names

|  |  |  |  |
| --- | --- | --- | --- |
| **std::vector<** **std::string** **>** | getVacValveNames | **(** **)** | Returns the set of vacuum valve names specified in the config file. |

## Open and Close Valves

|  |  |  |  |
| --- | --- | --- | --- |
| **void** | openValve | **(** **const std::string** valveName **)** | opens/closes valveName |
| **void** | closeValve | **(** **const std::string** valveName **)** |
| **void** | openValve1 | **()** | Open or close valve defined in function name |
| **void** | openValve2 | **()** |
| **void** | openValve3 | **()** |
| **void** | openValve4 | **()** |  |
| **void** | openValve5 | **()** |
| **void** | openValve6 | **()** |
| **void** | openValve7 | **()** |
| **void** | closeValve1 | **()** |
| **void** | closeValve2 | **()** |
| **void** | closeValve3 | **()** |
| **void** | closeValve4 | **()** |
| **void** | closeValve5 | **()** |
| **void** | closeValve6 | **()** |
| **void** | closeValve7 | **()** |
| **bool** | openAndWaitValve | **(** **const std::string** valveName, **const time\_t** waitTime **)** | sets the valveName to the state requested, and returns true if this state is achieved within waitTime, false otherwise. |
| **bool** | closeAndWait | **(** **const std::string** valveName, **const time\_t** waitTime **)** | sets the valveName to the state requested, and returns true if this state is achieved within waitTime, false otherwise. |
| **bool** | openAndWaitValve1 | **()** |
| **bool** | openAndWaitValve2 | **()** |  |
| **bool** | openAndWaitValve3 | **()** |  |
| **bool** | openAndWaitValve4 | **()** |  |
| **bool** | openAndWaitValve5 | **()** |  |
| **bool** | openAndWaitValve6 | **()** |  |
| **bool** | openAndWaitValve7 | **()** |  |
|  |  |  |  |

## Valve State Interrogators

|  |  |  |  |
| --- | --- | --- | --- |
| **bool** | isOpen | **(** **const std::string** valveName **)** | returns true if the given valve is in the open **(**closed**)** state |
| **bool** | isClosed | **(** **const std::string** valveName **)** |
| VELA\_ENUM::STATE | getVacValveState | **(** **const std::string** valveName **)** | Returns the state of the requested valve |

# VELA Shutters

The *velaShutterController* enables the user to:

* Open / close each laser shutter,
* Determine if a shutter is open / closed.
* Get the state of a shutter.
* Get a list of the shutter names defined in the config file.

The generic functions specified above can also be used to determine the state of the interlocks on a given shutter.

The config file defines:

* EPICS PVs for each shutter.
* A state PV, ‘*Sta’*, and ‘*On*’ and ‘*Off*’ PVs used for opening/closing shutters.
* The number of shutter interlocks.

## Shutter States ENUM

The states of a shutter are defined with the following enum

|  |  |
| --- | --- |
| VELA\_ENUM:: SHUTTER\_STATE | SHUTTER\_CLOSED, SHUTTER\_OPEN, SHUTTER\_TIMING, SHUTTER\_ERROR |

## Shutter States

|  |  |  |  |
| --- | --- | --- | --- |
| VELA\_ENUM::SHUTTER\_STATE | getShutterState | **(** **const std::string** **&** name **)** | Returns the state of name |
| **std::string** | getShutterStateStr | **(** **const std::string** **&** name **)** |
| **bool** | isOpen | **(** **const std::string** **&** name **)** | Returns true if the name is open **(**closed**)** |
| **bool** | isClosed | **(** **const std::string** **&** name **)** |

## Names

|  |  |  |  |
| --- | --- | --- | --- |
| **std::vector<** **std::string** **>** | getShutterNames | **()** | Returns the set of shutter names specified in the config file. |

## Open and Close Shutters

|  |  |  |  |
| --- | --- | --- | --- |
| **void** | open | **(** **const std::string** **&** name **)** | Opens or closes the shutter name, or that defined in the function name |
| **void** | close | **(** **const std::string** **&** name **)** |
| **void** | openShutter1 | **()** |
| **void** | openShutter2 | **()** |
| **void** | closeShutter1 | **()** |
| **void** | closeShutter2 | **()** |
| **void** | openAndWait | **(** **const std::string** **&** name**,** **const time\_t** waitTime = 2**)** |  |
| **void** | closeAndWait | **(** **const std::string** **&** name**,** **const time\_t** waitTime = 2**)** |  |
| **bool** | openAndWaitShutter1 | **( const time\_t** waitTime = 2 **)** |  |
| **bool** | openAndWaitShutter1 | **( const time\_t** waitTime = 2 **)** |  |
| **bool** | closeAndWaitShutter1 | **( const time\_t** waitTime = 2 **)** |  |
| **bool** | closeAndWaitShutter2 | **( const time\_t** waitTime = 2 **)** |  |

# VELA RF Controller

The *velaRFGunrController* enables the user to:

* Cycle through the main operational modes of the Modulator,
* Set and Read the amplitude and Phase
* Control and monitor RF power readings
* Get a list of the shutter names defined in the config file.

The RF Object is composed of the following pieces of virtual hardware, defined in the rf\_gun config files.

1 Modulator

2 RF Power Objects, “CAVITY” and ”KLYSTRON”

1 LLRF Object

## Modulator State and ENUM

The states of the RF modulator are defined in a local enum

|  |  |
| --- | --- |
| velaRFStructs::MOD\_STATE | ERROR1, UNKNOWN1, OFF, off\_Request, HV\_Intrlock, Standby\_Request, Standby, HV\_Off\_Requ, Trigger\_Interl, HV\_Request, HV\_On, Trig\_Off\_Req, Trig\_Request, Trig |

|  |  |  |  |
| --- | --- | --- | --- |
| velaRFStructs::MOD\_STATE | getModState | **( )** | Returns the current modulator state |
| **std::string** | getModStateStr | **( )** |

## Modulator Interlock States & ENUM

The modulator has many **(**many**)** different interlock states that are handled by a bespoke function instead of the generic interlock functions. **(**All the interlock states are not well mapped and as more states become known this enum may be updated.**)**

|  |  |
| --- | --- |
| velaRFStructs::MOD\_EXILOCK1 | BAD, GOOD, ERROR, UNKNOWN |

|  |  |  |  |
| --- | --- | --- | --- |
| velaRFStructs::MOD\_STATE | getModiLock | **( )** | Returns the current modulator interlock state, defined in the velaRFStructs:: MOD\_EXILOCK1 enum. |
| **std::string** | getModiLockStr | **( )** |

## Cycling Through Modulator States

During beam time there are 4 modes the modulator is cycled through, these can be set individually.

|  |  |  |  |
| --- | --- | --- | --- |
| **bool** | switchOnRF | **(** **const std::string** **&** name **)** | Attempts to set the modulator from the to “Trig “ from the current state |
| **bool** | switchOffRF | **(** **const std::string** **&** name **)** | Attempts to set the modulator to “OFF “ from the current state **(**should never fail?**)** |
| **void** | setModToOff | **()** | Attempts to set the modulator from the current state to the defined in the function name. |
| **void** | setModToStandby | **()** |
| **void** | setModToHVOn | **()** |
| **void** | setModToTrig | **()** |
| **void** | modReset | **()** | Attempts to reset the modulator interlocks to “GOOD” |
| **void** | modResetAndWait | **(const size\_t** waitTime **)** | Attempts to reset the modulator interlocks to “GOOD” for waitTime seconds, returns values from call to isModILockStateGood**()** |

## Modulator State Interrogation

|  |  |  |
| --- | --- | --- |
| **bool** | isModILockStateGood | **(** **const std::string** **&** name **)** |
| **bool** | isModILockStateNotGood | **(** **const std::string** **&** name **)** |
| **bool** | isModWarmedUp | **()** |
| **bool** | isModNotWarmedUp | **()** |
| **bool** | isModInHVOn | **()** |
| **bool** | isModInTrig | **()** |
| **bool** | isModInOff | **()** |
| **bool** | isModInStandby | **()** |

## LLRF OBJECT FUNCTIONS

These are used to set and read the amplitude (amp) and phase (Phi).

|  |  |  |  |
| --- | --- | --- | --- |
| **void** | setAmp | **( long** val **)** |  |
| **void** | setPhi | **( long** val **)** |  |
| **double** | getAmpR | **()** | The AMP RD Value in EPICS |
| **long** | getAmpW | **()** | The Amp WR Value in EPICS |
| **double** | getPhi | **()** |  |

## RF POWER OBJECT FUNCTIONS

These are used to read back and monitor the RF power readings. For the VELA gun there are two types of RF power objects, defined in the config file. **(**Typically the “KLYSTRON” and the “CAVITY.”**)**. These objects read the forward and reverse powers, as traces and at the curor position. As well as getters and setters there are functions to set up monitoring of the traces.

## Names

|  |  |  |  |
| --- | --- | --- | --- |
| **std::vector<** **std::string** **>** | getRFPowerNames | **()** | Returns the names of the RF power objects specified in the config file. |

## Monitoring Functions

|  |  |  |  |
| --- | --- | --- | --- |
| **void** | monitorTracesForNShots | **()** | Starts recording of the next N shots of RF power traces |
| **void** | monitorTracesForNShotsAndWait | **(const size\_t** N**)** | The function does not return until N traces have been captured |
| **bool** | isMonitoringTraces | **()** |  |
| **bool** | isNotMonitoringTraces | **()** |  |
| **bool** | killTraceMonitors | **()** | Stop all active RF trace monitoring |

## Getters and Setters

|  |  |  |  |
| --- | --- | --- | --- |
| **double** | getFwd | **(** **const std::string** **&** n **)** | Returns, or sets the item from power object n |
| **double** | getRev | **(** **const std::string** & n**)** |
| **double** | getRatio | **(** **const std::string** & n**)** |
| **double** | getFwdCursor | **(** **const std::string** & n **)** |
| **double** | setFwdCursor | **(** **const std::string** **&** **n,** **const double** v **)** |
| **double** | getRevCursor | **(const std::string** & n ) |
| **double** | setRevCursor | ( const std::string **& n**, const double v ) |
| **double** | setAllCursors | **(const double** **val** **)** |
| **std::vector<std::vector<double>>** | getRevT | () | Returns any data recorded after a call to  monitorTraces has been made |
| **std::vector<std::vector<double>>** | getFwdT | () |
| **std::vector<double>** | getFwdTStamp | () |
| **std::vector<std::string>** | getFwdTStampStr | () |
| **std::vector<double>** | getRevTStamp | () |
| **std::vector<std::string>** | getRevTStampStr | () |

# VELA MAGNET Controller

This controller enables users to control the magnets and enables the user to:

* Switch ON and OFF magnets
* Set and Read magnet currents
* Degauss magnets
* Interrogate magnet objects to find properties.

## Magnet Controller: Magnet Names

These functions return lists of magnet names, defined in the config file. As well as all names they can be returned by type.

|  |  |  |  |
| --- | --- | --- | --- |
| **std::vector<** **std::string** **>** | getMagnetNames | **(** **)** | Returns all magnet names from the config file |
| **std::vector<** **std::string** **>** | getQuadNames | **(** **)** |  |
| **std::vector<** **std::string** **>** | getHCorNames | **(** **)** |  |
| **std::vector<** **std::string** **>** | getVCorNames | **(** **)** |  |
| **std::vector<** **std::string** **>** | getDipNames | **(** **)** |  |
| **std::vector<** **std::string** **>** | getSolNames | **(** **)** |  |

## Magnet Controller: Magnet State Interrogators

These **Bool**ean interrogator functions are used to check properties, and states of magnet object “NAME.” Offline properties are defined in the config file.

|  |  |  |  |
| --- | --- | --- | --- |
| **bool** | isAQuad | **(** **const std::string** **&** name **)** | Returns true if name is of the type identified in the function name. Magnet types are defined in the config file |
| **bool** | isABSol | **(** **const std::string** **&** name**)** |
| **bool** | isAHCor | **(** **const std::string** **&** name**)** |
| **bool** | isAVCor | **(** **const std::string** **&** name**)** |
| **bool** | isADip | **(** **const std::string** **&** name**)** |
| **bool** | isASol | **(** **const std::string** **&** name**)** |
| **bool** | isACor | **(** **const std::string** **&** name**)** |
| **bool** | isNR | **(** **const std::string** **&** name**)** | Returns true if the polarity switching type of the magnet matches that in the function name. Magnet Polarity switching types are defined in the config file |
| **bool** | isBipolar | **(** **const std::string** **&** name**)** |
| **bool** | isNRGanged | **(** **const std::string** **&** name**)** |
| **bool** | isNRorNRGanged | **(** **const std::string** **&** name**)** |
| **void** | showMagRevType | **()** | Prints all magnet reverse types to stdout |
| **bool** | isON | **(** **const std::string** **&** name**)** | Status of magnet PSU |
| **bool** | isOFF | **(** **const std::string** **&** name**)** |
| **bool** | isON\_psuN | **(** **const std::string** **&** name**)** | Status of magnet N-R polarity flipping PSU |
| **bool** | isOFF\_psuN | **(** **const std::string** **&** name**)** |
| **bool** | isON\_psuR | **(** **const std::string** **&** name**)** |
| **bool** | isOFF\_psuR | **(** **const std::string** **&** name**)** |
| **bool** | isDegaussing | **(** **const std::string** **&** name**)** |  |
| **bool** | isNotDegaussing | **(** **const std::string** **&** name**)** |  |

## Getting and Setting Set (SI and Read (RI) Magnet Currents

There are multiple versions of getting and setting the currents. These handle setting multiple magnets simultaneously as well as functions that wait for a magnet to reach a requested current **(**to within a tolerance**)** before returning.

|  |  |  |  |
| --- | --- | --- | --- |
| **double** | getSI | **(** **const std::string** **&** name**)** |  |
| **std::vector<** **double** **>** | getSI | **(** **const std::vector<** **std::string** **>** **&** name**)** |  |
| **double** | getRI | **(** **const std::string** **&** name**)** |  |
| **std::vector<** **double >** | getRI | **(const std::vector<** **std::string** **>** **&** name**)** |  |

## Setting SI

There are 4 different version of the setSI function. The first two attempt to set SI for magName to value. They return True if EPICS accepted the command. That does not mean that the current in the magne tactually gets to the value requested. The second two functions will wait timeOUT seconds for the magnet SRI values to reach the SI value, to within tolerances. These functions return True if the current is set, or when setting SI for multiple magnets, a vector of magnet names that successfully reached the requested SI value.

|  |  |  |
| --- | --- | --- |
| **bool** | setSI | **(** **const std::string** **&** name**,** **const double** val **)** |
| **bool** | setSI | **(** **const std::vector<** **std::string** **>** **&** name**,** **const std::vector<** **double** **>&** val**)** |
| **bool** | setSI | **(** **const std::string** **&** name**,** **const double** val**,** **const double** tolerances**,** **const size\_t** timeOUT **)** |
| **std::vector<std::string>** | setSI | **(** **const std::vector<** **std::string** **>** **&** name**,** **const std::vector<** **double** **>** **&** val**,**  **const std::vector<** **double** **>** **&** tolerances**,** **const size\_t** timeOUT **)** |

The RI tolerance is defined in the config file, but can be overwritten using:

|  |  |  |
| --- | --- | --- |
| **void** | setRITolerance | **(** **const std::string** **&** name**,** **const double** val**)** |

## Switching On and Off Power Supplies

|  |  |  |  |
| --- | --- | --- | --- |
| **bool** | switchONpsu | **(** **const std::string** **&** name**)** | These functions return true if the command was successfully sent to EPICS, not if the PSU switched on **(**off**)**. |
| **bool** | switchONpsu | **(** **const std::vector<** **std::string** **>&** name**)** |
| **bool** | switchOFFpsu | **(** **const std::string** **&** name**)** |
| **bool** | switchOFFpsu | **(** **const std::vector<** **std::string** **>&** name**)** |

## Degaussing

Degaussing always happens in a separate thread. This means that a magnet can be degaussed and still accessed from the main thread, **(**meaning it’s currents can be changed.**)** **(*I think we need to include a guard against this so you can’t change SI from the public setSI functions? Currently I have prevented users doing that at a higher (the GUI) level, but I guess this is actually evil and so should be avoided.*)**

|  |  |  |  |
| --- | --- | --- | --- |
| **size\_t** | degauss | **(** **const std::string** **&** name**,** **bool** resetToZero **=** **true** **)** | These functions return the number of magnets |
| **size\_t** | degauss | **(** **const std::vector<** **std::string** **>** **&** name**,** **bool** resetToZero **=** **true** **)** |
| **size\_t** | degauss | **(** **bool** resetToZero **=** **true** **)** |

## Magnet States

The relevant states of all magnets can be contained in a magnetStateStruct ., defined in the velaMagStructs namespace. These can be used to quickly get all magnet states, or apply all states to all magnets. They are used when using DBURTS.

|  |  |  |
| --- | --- | --- |
| velaMagStructs::magnetStateStruct | getCurrentMagnetState | **()** |
| **void** | applyMagnetStateStruct | **(const** velaMagStructs::magnetStateStruct & ms **)** |

## DBURTS

For complete portability fileName should give the full path to where data is to be saved / read from.

|  |  |  |
| --- | --- | --- |
| velaMagStructs::magnetStateStruct | getDBURT | **(const std::string** **&** fileName**)** |
| velaMagStructs::magnetStateStruct | getDBURTCorOnly | **(const std::string** **&** fileName**)** |
| velaMagStructs::magnetStateStruct | getDBURTQuadOnly | **(const std::string** **&** fileName**)** |
| **void** | applyDBURT | **(const std::string** **&** fileName**)** |
| **void** | applyDBURTCorOnly | **(const std::string** **&** fileName**)** |
| **void** | applyDBURTQuadOnly | **(const std::string** **&** fileName**)** |

DBURTS can be written either from data contained in a passed in magnetStateStruct, or directly using the internal data to the magnetController. The functions returns whether the file was written successfully or not.

|  |  |  |
| --- | --- | --- |
| **bool** | writeDBURT | **(** **const std::string** **&** fileName **=** ""**,** **const std::string** **&** comments **=** "" **)** |
| **bool** | writeDBURT | **(** **const** velaMagStructs::magnetStateStruct **&** ms**,** **const std::string** **&** fileName **=** ""**,**  **const std::string** **&** comments **=** "" **)** |

## Magnet Object Raw Data Access

For c++ development we allow access to the raw magnet object structs **(**defined in velaMagStructs**)** via reference or pointer

|  |  |  |
| --- | --- | --- |
| **const** velaMagStructs::magnetObject & | getMagObjConstRef | **(** **const std::string** **&** name **)** |
| **const** velaMagStructs::magnetObject \* | getMagObjConstPtr | **(** **const std::string** **&** name **)** |

# VELA CAMERA Controller

This controller enables user to start and stop cameras and acquire images. Image data processing, following techniques devloped in [ref] is still to be added to this controller

## Start and Stop Camera

|  |  |  |  |
| --- | --- | --- | --- |
| **bool** | start | **(const std::string** **&** cam **)** | Start **(**stop**)** a camera acquiring data. The return value is if the command was correctly sent to EPICS, not if the camera has actually started **(**stopped**)** |
| **bool** | stop | **(const std::string** **&** cam **)** |
| **bool** | start | **(const std::vector<** **std::string** **>** **&** cam **)** |
| **bool** | stop | **(const std::vector<** **std::string** **>** **&** cam **)** |
| **bool** | startAndWait | **(const std::string** **&** cam**,** **size\_t** timeout **)** | Wait timeout secs for camera to start. Returns success or list of cameras that started |
| **std::vector<std::string>** | startAndWait | **(const std::vector<std::string>** **&**cam**,**  **size\_t** timeout**)** |
| **bool** | stopAll | **()** |  |

## Acquire / Monitor Camera Data

Acquisition of camera data is done with the following functions: If N**>**0 then N shots of data will be acquired, each available after acquisiiton. If N=-1 **(**default**)** then each successive shot of data overwrites the previous data. **(**This functionality will be used with online data processing.**)**

|  |  |  |
| --- | --- | --- |
| **bool** | startCamDataMonitor | **(const std::string** **&** cam**,** **size\_t** N **= -1)** |
| **std::vector<** **std::vector<**velaCamStructs::camDataType**>>** | getRawData | **(const std::string** **&** name**)** |
| **std::vector<** **double** | getRawDataStamp | **(const std::string** **&** name**)** |
| **std::vector<** **std::string** **>** | getRawDataStampStr | **(const std::string** **&** name**)** |

## Camera State Interrogators

Interrogate camera objects to find on / off or monitoring status.

|  |  |  |  |
| --- | --- | --- | --- |
| **bool** | isON | **( const std::string** **&** cam **)** | Interrogate camera objects to find on / off or monitoring status |
| **bool** | isOFF | **( const std::string** **&** cam **)** |
| **bool** | isMonitoring | **(** **const std::string** **&** cam **)** |
| **bool** | isNotMonitoring | **(** **const std::string** **&** cam **)** |

# VELA Screen Controller

# The *velaScreenController* enables the user to;

# Move all YAG screens in and out

* Move Screens 1/2/3 to each slit/hole position
* Stop a screen while it is moving
* Get the state of each screen (Position in mm for YAGs 1/2/3 and In or Out for the other screens)

The generic functions specified above can also be used to determine the state of the interlocks on a given screen. YAGs 1/2/3 have separate interlocks to monitor, defined in the PV\_MONITORS in the config file.

**Screen States ENUM**

The states of a screen are defined with the following enum:

|  |  |
| --- | --- |
| VELA\_ENUM:: SCREEN\_STATE | SCREEN\_IN, SCREEN\_OUT, SCREEN\_ERROR, SCREEN\_UNKNOWN, SCREEN\_H\_MIRROR, SCREEN\_50U\_SLIT, SCREEN\_H\_25U\_SLIT, SCREEN\_H\_63MM\_HOLE, SCREEN\_H\_10MM\_HOLE, SCREEN\_H\_YAG, SCREEN\_V\_YAG, SCREEN\_V\_SLIT, SCREEN\_MOVING |

The SCREEN\_IN and SCREEN\_OUT only apply to screens where the only options are in or out (all screens except YAGs 1/2/3) the rest of the states are for YAGs 1/2/3 (except SCREEN\_ERROR and SCREEN\_UKNOWN which apply to all screens).

**Screen States**

|  |  |  |  |
| --- | --- | --- | --- |
| VELA\_ENUM::SCREEN\_STATE | getScreenState | **(** **const std::string** **&** name **)** | Returns the state of name, when passed only a screen name returns whether the YAG screen is IN or OUT |
| **std::string** | getScreenStateStr | **(** **const std::string** **&** name **)** |
| **VELA\_ENUM::SCREEN\_STATE** | getScreenState | **(** **const std::string** **&** name, **const std::string** **&** V\_H **)** | Returns the vertical or horizontal state (V\_H) of name, when passed either “VERTICAL” or “HORIZONTAL” as V\_H assumes it is either YAG 1/2/3 and so can have more possible states that just IN or OUT |
| **std::string** | getScreenStateStr | **(** **const std::string** **&** name, **const std::string** **&** V\_H **)** |
| **bool** | IsIn | **(** **const std::string** **&** name **)** | Returns true if name is IN, used for all screens except YAGs 1/2/3 |
| **bool** | IsOut | **(** **const std::string** **&** name **)** | Returns true if name is OUT, used for all screens except YAGs 1/2/3 |
| **double** | getScreenPosition | **(** **const std::string** **&** name **)** | Returns the current positon of the screen in mm (used for YAGs 1/2/3) |
| **bool** | horizontal\_disabled\_check | **(const std::string** **&** name) | Returns true if the interlock disabling horizontal movement for name is true (used for YAGs 1/2/3) |
| **bool** | vertical\_disabled\_check | **(const std::string** **&** name) | Returns true if the interlock disabling vertical movement for name is true (used for YAGs 1/2/3) |

**Names**

|  |  |  |  |
| --- | --- | --- | --- |
| **std::vector<** **std::string** **>** | getScreenNames | **()** | Returns the set of screen names specified in the config file. |

**Move Screens**

|  |  |  |  |
| --- | --- | --- | --- |
| **void** | Screen\_In | **(** **const std::string** **&** name **)** | Moves name IN or OUT, when called with YAG 1/2/3 as name moves the screen to the position of the YAG |
| **void** | Screen\_Out | **(** **const std::string** **&** name **)** |
| **void** | Screen\_Move | **(const std::string** **&** name, **const std::string** **&** position**)** | Moves name to one of the positions defined in the config file (position). Only used for YAGs 1/2/3 |
| **void** | Screen\_Stop | **(const std::string** **&** name**)** | Stops name at its current position, only used for YAGs 1/2/3 |
| **void** | All\_Out | **()** | Moves all screens to the OUT position |
| **void** | controller\_move\_to | **(const std::string** **&** name, **const std::string** **&** V\_H, **const double &** position) | Move name to position (given as a number in mm) in either the vertical or horizontal (V\_H), this is only used for YAGs 1/2/3 |

**Object Info**

|  |  |  |  |
| --- | --- | --- | --- |
| **void** | get\_info | **(const std::string** **&** name) | Prints all info about name object, such as object name, PV root, state, horizontal/vertical position (for YAGs 1/2/3), interlock states (YAGs 1/2/3) |
| **void** | get\_config\_values | **(const std::string** **&** name) | Prints all info defined in the config files for name screen object i.e. the positions that name can be sent to, number of interlocks etc. |

# DEPRECATED

Quantum Boxes

The quantum boxes may well be replaced after the winter shutdown, but just in case they remain, the quantum box controller allows the user to:

* Arm / disarm each quantum box **(**either RF-DLY or TIM-DLY**)**, in 2 separate ways **(**see below**)**.
* See if a quantum box is armed or disarmed.
* Get the state of a quantum box.
* Get a list of the quantum box names.

The config file defines:

* EPICS PVs for each quantum box.
* We only really want to be able to return the state of a quantum box, and set its state to “armed” or “disarmed”, and this is all that is specified here.

**void** **armQBox(** **const std::string** **qBox** **)**

* **(**similarly for disarmQBox**)** – sets the given quantum box state to Armed or Disarmed.

**bool** **isArmed(** **const std::string** **qBox** **)**

* **(**similarly for isDisarmed**)** – returns True if the quantum box is in the state requested, false otherwise.

**bool** **armAndWait(** **const std::string qBox**, **const time\_t** **waitTime** **)**

* **(**similarly for **disarmAndWait)** – sets the quantum box to the state requested, and waits to see if this state is achieved in the given time.

**std::vector<** **std::string** **>** **getQBoxNames()**

* Returns a list of all the quantum box names defined in the config file.

VELA\_ENUM::QBOX\_STATE getQBoxState**(** **const std::string** qBox **)**

* Returns the state of the requested quantum box.

# References

[ 1 ]

[\\fed.cclrc.ac.uk\org\NLab\ASTeC\Projects\VELA\documentation\notes\Hardware\_Controllers\_A\_Mid-Level\_System\_for\_Controlling\_Hardware\_and\_Accessing\_Data\_on\_VELA-CLARA.docx](file:///\\fed.cclrc.ac.uk\org\NLab\ASTeC\Projects\VELA\documentation\notes\Hardware_Controllers_A_Mid-Level_System_for_Controlling_Hardware_and_Accessing_Data_on_VELA-CLARA.docx)

[ 2 ] <http://projects.astec.ac.uk/EBTFManual/index.php/Diagnostics:BPMs>

[ 3 ] [\\FED.cclrc.ac.uk\Org\NLab\ASTeC\Projects\VELA\Software\c++\Applications\bpmReCalibrate](file:///\\FED.cclrc.ac.uk\Org\NLab\ASTeC\Projects\VELA\Software\c++\Applications\bpmReCalibrate)