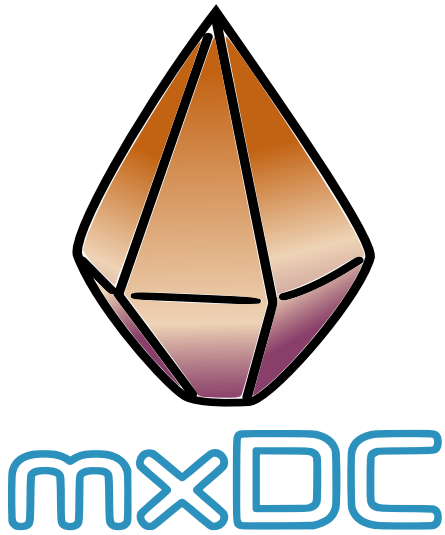

MxDC Developer Documentation

Release 4.1.6-gedc482b

March 16, 2012

CONTENTS

Part I
CONTENTS

CHAPTER

ONE

INTRODUCTION

MxDC (Macromolecular Crystallography Data Collector) is a modular software package layered above the low-level beamline instrument control system. The first layer, also called the beamline control module (BCM), provides a functional abstraction of all hardware devices usually found at MX beamlines, while the graphical userface provides an integrated user-friendly interface for interactive data acquisition.

This is the documentation for MxDC and the BCM. It documents the application programming interfaces for developers wishing extend or customize the software. Some information about dependences and installation procedures is also provided.

This document is targeted at beamline administrators or software developers rather than users of MxDC. If you are a user looking for user documentation, please consult the MxDC user guide

1.1 Prerequisites

MxDC and the BCM have been designed and tested to work on Linux systems. Although it would probably work after considerable effort on other Unix-like system or even on Windows, we do not recommend Non-Linux systems.

MxDC and the BCM needs at least **Python 2.4** to run. It does not yet run under Python 3+. It is a pure python package which requires no compilation. Additionally, several python modules are required, without which MxDC and the BCM will not run or function correctly.

Required modules:

- `matplotlib` >= 0.99
- `numpy` >= 1.2
- `scipy` >= 0.6
- `python-ctypes` (for python < 2.5 only)

- `python-simplejson` (for python < 2.5 only)
- `python-imaging` >= 1.1.6
- `python-zope-interface` >= 3.3
- `Twisted` >= 8.2
- `avahi-tools` (provides python avahi module needed by mdns module)
- `dbus-python`, usually installed by default on most recent distributions
- Others: (`pygtk` usually included with distribution)

Note: If using Twisted >= 9.0, `pyasn1`, and `python-crypto` are also required

---

General tips:

- To avoid dependency nightmares, always prefer packages already available for your distribution. Only build where necessary.
- For installation at the CMCF beamlines, dependencies not available as part of the distribution have been built and are available in `/cmcf_apps/deps`. Select rpm folder matching your architechture (x86 or x86_64) and install the packages.
- On Scientific Linux/RHEL/CENTOS 5+, most of the packages are available from distribution repositories. Make sure you have EPEL and RPMFusion repositories enabled from the start. That way, doing yum install <package-name> should get you most of the above packages "for free".
- A few of the packages may need to be built. It is recommended to build rpm's and and then install those, so you can easily uninstall them or update them, and it is easy to tell what versions are installed. Therefore, it is not recommended to do:

  ```
  python setup.py install
  ```

  Instead, do:

  ```
  python setup.py bdist_rpm
  ```

  which should generate an rpm package within the `dist` subdirectory. You can then install the rpm package normally.

Other Programs/Libraries:

- CBFlib: this library must be installed to allow MxDC to recognize CBF diffraction image files. If it is not available, only MarCCD and SMV formatted image files will be supported.
- XREC is used for automated crystal centering and should be available in the execution path. Without it, the automatec "crystal" and "loop" centering will fail.

- CHOOCH is required for MAD and SAD scans.
- MxLIVE and Autoprocess servers are automatically discovered at run-time but can be specified in the configuration file manually if not part of the local network. Without these, automated data processing and uploading of datasets or processing reports will not work.

1.2 Installation

1. Uncompress the tarball archive into a target directory
2. Edit `deploy/bcm.csh` and `deploy/bcm.csh` to suit the installation
3. Create a configuration file in `etc/xxx.py` to match your beamline hardware. It is better to copy an existing one and modify it.
4. Make sure `bcm.csh` is sourced on login.

1.3 Usage

To run the programs:

- `mxdc` will launch the application.
- `sim-mxdc` will lauch the application with simulated devices.
- `sampleviewer` will launch the hutch sample video screen only.
- `hutchviewer` will launch the hutch control screen only.
- `blconsole` will lauch a beamline python console for interactive scanning and manipulation etc.

CHAPTER

TWO

API DOCUMENTATION

This section is automatically generated from the code.

2.1 Interfaces

class bcm.device.interfaces.IDevice(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

A generic device interface.

class bcm.device.interfaces.IAutomounter(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: bcm.device.interfaces.IDevice

A sample automounter device interface.

class bcm.device.interfaces.ICollimator(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: bcm.device.interfaces.IDevice

An X-ray beam collimator device object.

class bcm.device.interfaces.ICounter(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: bcm.device.interfaces.IDevice

An integrating counter object.

class bcm.device.interfaces.IGoniometer(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: bcm.device.interfaces.IDevice

A goniometer device object.

class bcm.device.interfaces.IImagingDetector(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: bcm.device.interfaces.IDevice

An imaging detector device for aquiring image frames.

2.2 Devices

2.2.1 Base Device

class bcm.device.base.BaseDevice

Bases: gobject._gobject.GObject

A generic device object class. All devices should be derived from this class. Objects of this class are instances of *gobject.GObject*.

Attributes:

- *pending_devs*: a list of inactive child devices.
- *health_manager*: A HealthManager object.
- *state_info*: A dict containing state information.
- *name*: the name of the device

Signals:

- *active*: emitted when the state of the device changes from inactive to active and vice-versa. Passes a single boolean value.
- *busy*: emitted to notify listeners of a change in the busy state. A single boolean parameter is passed along with the signal. *True* means busy, *False* means not busy.
- *health*: signals an device sanity/error condition. Passes two parameters, an integer error code and a string description. The integer error codes are:
  - 0 : No error
  - 1 : MINOR, no impact to device functionality.
  - 2 : MARGINAL, no immediate impact. Attention may soon be needed.
  - 4 : SERIOUS, functionality impacted but recovery is possible.
  - 8 : CRITICAL, functionality broken, recovery is not possible.
  - 16 : DISABLED, device has been manually disabled.

- *message*: signal for sending messages from the device to any listeners. Messages are passed as a single string parameter.

Dynamic Attributes:

- *<signal>_state*: for each signal defined in derived classes, the last transmitted data corresponding to the current state can be obtained through the dynamic variable with the suffix "_state" added to the signal name. For example:

```
>>> obj.active_state
>>> True
```

Dynamic Methods:

- *is_<signal>()*: for each boolean signal defined in derived clases, the current state can be obtained by adding the prefix "**is_**" to the signal name. For example:

```
>>> obj.is_active()
>>> True
```

`get_state()`
Obtain a copy of the device state.

Returns: A dict mapping state keys to their correponding values. Entries contain at least the following entries:

- *active* : Boolean
- *busy*: Boolean
- *health*: tuple(int, string)
- *message*: string

`set_state(**kwargs)`
Set the state of the device and emit the corresponding signal.

Kwargs: Keyworded arguments follow the same conventions as the state dictionary and correspond to any signals defined for the device. Signals must be previously defined as supported signals of the device.

For example:

```
mydevice.set_state(active=True, busy=False,
                   health=(1, 'error','too hot'),
                   message="the device is overheating")
```

`add_pv(*args, **kwargs)`
Add a process variable (PV) to the device.

Create a new process variable, add it to the device. Keyworded arguments should be the same as those expected for instantiating a `bcm.protocol.ca.PV` object.

Returns: A reference to the created `bcm.protocol.ca.PV` object.

`add_devices(*devices)`
Add one or more devices as children of this device.

class `bcm.device.base.HealthManager(**kwargs)`
Bases: `object`

Manages the health states. The object enables registration and removal of error states and consistent reporting of health based on all currently active health issues.

Takes key worded arguments. The keyword name is the context, and the value is an error string to be returned instead of the context name with all health information for the given context.

`register_messages(**kwargs)`
Update or add entries to the context message register

`add(severity, context, msg=None)`
Adds an error state of the given context as a string and a severity value as a integer. If a message is given, it will be stored and used instead of the context name. Only one message per context type is allowed. Use a different context if you want different messages.

`remove(context)`
Remove all errors of the given context as a string.

`get_health()`
Generate an error code and string based on all the currently registered errors within the health registry.

2.2.2 Automounter

class `bcm.device.automounter.BasicAutomounter`
Bases: `bcm.device.base.BaseDevice`

Basic Automounter object.

An automounter object contains a number of AutomounterContainers.

Signals:

- *state*: transmits changes in automounter states. The signal data is a dictionary with three entries {'busy': <boolean>,
- *enabled*: <boolean>, 'needs':[<str>,<str>,...]}.

- *mounted*: emitted when a sample is mounted or dismounted. The data transmitted is a tuple of the form (<port no.>, <barcode>) when mounting and *None* when dismounting.
- ***progress*: notifies listeners of automounter progress. Data is a** tuple of the form (<% complete>, <description>).

`abort()`
Abort all actions.

`probe()`
Command the automounter the probe for containers and port states.

`mount(`*port*, *wash=False*, *wait=False*`)`
Mount the sample at the specified port, optionally washing the sample in the process. Does nothing if the requested port is already mounted. Dismounts the mounted sample prior to mounting if a another sample is already mounted.

Args:

- *port* (str): Address to mount.

Kwargs:

- *wash* (bool): Whether to wash or not (default is False)
- *wait* (bool): Run asynchronously or block (default is async, False)

Returns: bool. True if the requested sample is successfully mounted, and False otherwise.

`dismount(`*port=None*, *wait=False*`)`
Dismount the sample into the specified port if provided, otherwise dismount it to the original port from which it was mounted.

Kwargs:

- *port* (str): Destination address to dismount to, default is original port
- *wait* (bool): Run asynchronously or block (default is async, False)

`wait(`*start=True*, *stop=True*, *timeout=240*`)`
Wait for the automounter

Kwargs:

- *start* (bool): Wait for automounter to start.
- *stop* (bool): Wait for automounter to stop.
- *timeout* (int): Maximum time to wait

Returns: True if the wait was successful. False if the wait timed-out.

`is_mountable(`*port*`)`
Check if a sample location can be mounted safely. Does not guarantee that the port is actually mountable, only that the the port was marked as mountable during the last probe operation.

Args: *port* (str): The sample location to check.

Returns: True or False.

`is_mounted(`*port=None*`)`
Check if any sample or a specific sample location is currently mounted.

Kwargs: *port* (str): The sample location to check.

Returns: True if a port is specified and is mounted or if no port is specified and any sample is mounted. False otherwise.

`get_port_state(`*port*`)`
Obtain the detailed state of the specified sample location.

Args: *port* (str): The sample location

Returns: int. one of:

```
0 -- PORT_EMPTY
1 -- PORT_GOOD
2 -- PORT_UNKNOWN
3 -- PORT_MOUNTED
4 -- PORT_JAMMED
5 -- PORT_NONE
```

2.2.3 Counters

class `bcm.device.counter.Counter(`*pv_name*, *zero=0.0*`)`
Bases: `bcm.device.base.BaseDevice`

EPICS based Counter Device objects. Enables counting and averaging of process variables over given time periods.

Args: pv_name (str): Process Variable name.

Kwargs: zero (float): Zero offset value. Defaults to 0.0;

Returns float. The average process variable value during the count time.

`count(`*t*`)`
Count for the specified amount of time and return the numeric value corresponding to the average count. This method blocks for the specified time.

Args: t (float): averaging time in seconds.

Returns float. The average process variable value during the count time.

2.2.4 Detector

class `bcm.device.detector.MXCCDImager(`*name*, *size*, *resolution*, *detector_type='MX300'*`)`

Bases: `bcm.device.base.BaseDevice`

MX Detector object for EPICS based Rayonix CCD detectors at the CLS.

Args: *name* (str): Root name of the EPICS record Process Variables. *size* (int): The size in pixels of the detector. Assumes square detectors. *resolueion* (float): The pixel size in

Kwargs: *detector_type* (str): The type of detector. e.g. "MX300" for Rayonix MX CCD 300.

`initialize(`*wait=True*`)`
Initialize the detector and take background images if necessary. This method does not do anything if the device is already initialized.

Kwargs: *wait* (bool): If true, the call will block until initialization is complete.

`start(`*first=False*`)`
Start acquiring.

Kwargs: *first* (bool): Specifies whether this is the first of a series of acquisitions. This is used to customize the behaviour for the first.

`stop()`
Stop and Abort the current acquisition.

`save(`*wait=False*`)`
Save the current buffers according to the current parameters.

Kwargs: *wait* (bool): If true, the call will block until the save operation is complete.

`get_origin()`
Obtain the detector origin/beam position in pixels.

Returns: tuple(x, y) corresponding to the beam-x and beam-y coordinates.

`wait(`*state='idle'*`)`
Wait until the detector reaches a given state.

Kwargs: *state* (str): The state to wait for. Default 'idle'.

`set_parameters(`*data*`)`
Set the detector parameters for the image header and file names.

Args: *data* (dict): A dictionary of key value pairs for the parameters. supported parameters are:

- *filename* (str), Output file name of the image.
- *directory* (str), Directory name to store image.
- *beam_x* (int), Detector X-origin in pixels.
- *beam_y* (int), Detector Y-origin in pixels.
- *distance* (float), Detector distance in mm.
- *exposure_time* , Exposure time in seconds.
- *axis* (str), Spindle rotation axis.
- *wavelength* (float), Wavelength of radiation in Angstroms.
- *delta_angle* (float), Delta oscillation angle in deg.
- *frame_number* (int), Frame number.
- *name* (str), File name prefix for the image.
- *start_angle* (float), Starting spindle position of image in deg.
- *energy* (float), Wavelength of radiation in KeV.
- *comments* (str), File comments.

2.2.5 Diffractometer

class `bcm.device.diffractometer.Diffractometer`(*distance*, *two_theta*, *name='Diffractometer'*)

Bases: `bcm.device.base.BaseDevice`

A Container device object which groups a detector distance and detector swing-out.

Args: *distance* (class::*interfaces.IMotor* provider): device which controls the detector distance. *two_theta* (class::*interfaces.IMotor* provider): device which controls the detector swing-out angle.

Kwargs: *name* (str): The name of the device group.

`wait()`
Wait for both child devices to finish moving.

`stop()`
Stop both child devices.

2.2.6 Goniometer

class bcm.device.goniometer.BackLight(*name*)
Bases: bcm.device.misc.BasicShutter

A specialized in-out actuator for pneumatic OAV backlight at the CLS.

class bcm.device.goniometer.Goniometer(*name*, *blname*, *mnt_cmd*, *minibeam*)
Bases: bcm.device.goniometer.GoniometerBase

EPICS based Parker-type Goniometer at the CLS 08ID-1.

Args: *name* (str): PV name of goniometer EPICS record. *blname* (str): PV name for Backlight PV. *mnt_cmd* (str): PV name for toggling mount mode. *minibeam* (str): PV name for minibeam motor.

configure(***kwargs*)
Configure the goniometer to perform an oscillation scan.

Kwargs: *time* (float): Exposure time in seconds *delta* (float): Delta oscillation range in deg *angle* (float): Starting angle of oscillation in deg

set_mode(*mode*, *wait=False*)
Set the mode of the goniometer environment.

Args:

***mode* (str) one of:**

- "CENTERING" : Prepare for centering
- "MOUNTING" : Prepare for mounting/dismounting samples
- "COLLECT" : Prepare for data collection
- "BEAM" : Inspect the beam
- "SCANNING" : Prepare for scanning and fluorescence measurements

Kwargs: *wait* (bool): if True, block until the mode is completely changed.

scan(*wait=True*)
Perform an oscillation scan according to the currently set parameters

Kwargs: *wait* (bool): if True, wait until the scan is complete otherwise run asynchronously.

class bcm.device.goniometer.MD2Goniometer(*name*)
Bases: bcm.device.goniometer.GoniometerBase

EPICS based MD2-type Goniometer at the CLS 08B1-1.

Args: *name* (str): Root PV name of the goniometer EPICS record.

`configure(**kwargs)`
Configure the goniometer to perform an oscillation scan.

Kwargs: *time* (float): Exposure time in seconds *delta* (float): Delta oscillation range in deg *angle* (float): Starting angle of oscillation in deg

`set_mode(mode, wait=False)`
Set the mode of the goniometer environment.

Args:

***mode* (str) one of:**

- "CENTERING" : Prepare for centering
- "MOUNTING" : Prepare for mounting/dismounting samples
- "COLLECT" : Prepare for data collection
- "BEAM" : Inspect the beam
- "SCANNING" : Prepare for scanning and fluorescence measurements

Kwargs: *wait* (bool): if True, block until the mode is completely changed.

`scan(wait=True)`
Perform an oscillation scan according to the currently set parameters

Kwargs: *wait* (bool): if True, wait until the scan is complete otherwise run asynchronously.

`stop()`
Stop and abort the current scan if any.

2.2.7 Multi-Channel Analyzer

class `bcm.device.mca.XFlashMCA(name, nozzle=None, channels=4096)`
Bases: `bcm.device.mca.BasicMCA`

mcaRecord based single element fluorescence detector object.

class `bcm.device.mca.VortexMCA(name, channels=2048)`
Bases: `bcm.device.mca.BasicMCA`

EPICS based 4-element Vortex ME4 detector object.

2.2.8 Monochromator

class bcm.device.monochromator.Monochromator(*simple_energy*, *energy*, *mostab=None*)

Bases: bcm.device.base.BaseDevice

wait()

stop()

2.2.9 Motors

exception bcm.device.motor.MotorError

Bases: exceptions.Exception

Base class for errors in the motor module.

class bcm.device.motor.MotorBase(*name*)

Bases: bcm.device.base.BaseDevice

Base class for motors.

Signals:

- *changed* (float): Emitted everytime the position of the motor changes. Data contains the current position of the motor.
- *timed-change* (tuple(float, float)): Emitted everytime the motor changes. Data is a 2-tuple with the current position and the timestamp of the last change.

class bcm.device.motor.Motor(*pv_name*, *motor_type*)

Bases: bcm.device.motor.MotorBase

Motor object for EPICS based motor records.

Args:

- *pv_name* (str): Root PV name for the EPICS record.
- *motor_type* (str): Type of EPICS motor record. Accepted values are:

```
"vme" - CLS VME58 and MaxV motor record without encoder support.
"vmeenc" - CLS VME58 and MaxV motor record with encoder support.
"cls" - OLD CLS motor record.
"pseudo" - CLS PseutoMotor record.
```

get_position()

Obtain the current position of the motor in device units.

Returns: float.

`configure(**kwargs)`

Configure the motor. Currently only allows changing the calibration flag and reseting the position for "vme" and "vmeenc" motors only.

Kwargs:

- *calib* (int): 0 removes the calibration flag, anything else sets it.
- *reset* (float): position to recalibrate the current motor to.

`move_to(pos, wait=False, force=False)`

Request the motor to move to an absolute position. By default, the command will not be sent to the motor if its current position is the same as the requested position within its preset precision. In addition the command will not be sent if the motor health severity is not zero (GOOD).

Args:

- *pos* (float): Target position to move to.

Kwargs:

- *wait* (bool): Whether to wait for move to complete or return immediately.
- *force* (bool): Force a command to be sent to the motor even if the target is the same as the current position.

`move_by(val, wait=False, force=False)`

Similar to `move_to()`, except request the motor to move by a relative amount.

Args:

- *val* (float): amount to move position by.

Kwargs:

- *wait* (bool): Whether to wait for move to complete or return immediately.
- *force* (bool): Force a command to be sent to the motor even if the target is the same as the current position.

`stop()`

Stop the motor from moving.

`wait(start=True, stop=True)`

Wait for the motor busy state to change.

Kwargs:

- *start* (bool): Wait for the motor to start moving.
- *stop* (bool): Wait for the motor to stop moving.

`get_settings()`
Obtain the motor settings for saving/restore purposes.

Returns: A dict.

class `bcm.device.motor.VMEMotor(`*name*`)`
Bases: `bcm.device.motor.Motor`

Convenience class for "vme" type motors.

Args:

- *name* (str): Root PV name of the motor record.

class `bcm.device.motor.ENCMotor(`*name*`)`
Bases: `bcm.device.motor.Motor`

Convenience class for "vmeenc" type motors.

Args:

- *name* (str): Root PV name of the motor record.

class `bcm.device.motor.CLSMotor(`*name*`)`
Bases: `bcm.device.motor.Motor`

Convenience class for "cls" type motors.

Args:

- *name* (str): Root PV name of the motor record.

class `bcm.device.motor.PseudoMotor(`*name*`)`
Bases: `bcm.device.motor.Motor`

Convenience class for "pseudo" type motors.

Args:

- *name* (str): Root PV name of the motor record.

class `bcm.device.motor.BraggEnergyMotor(`*name*, *enc=None*, *motor_type='vme'*`)`
Bases: `bcm.device.motor.Motor`

A specialized energy motor for using just the monochromator bragg angle.

Args:

- *name* (str): Root PV name of motor record.

Kwargs:

- *enc* (str): PV name for an optional encoder feedback value from which to read the energy value.
- *motor_type* (str): Type of EPICS motor record. Accepted values are:

```
"vme" - CLS VME58 and MaxV motor record without encoder support.
"vmeenc" - CLS VME58 and MaxV motor record with encoder support.
"cls" - OLD CLS motor record.
"pseudo" - CLS PseutoMotor record.
```

class bcm.device.motor.FixedLine2Motor(*x*, *y*, *slope*, *intercept*, *linked=False*)

Bases: bcm.device.motor.MotorBase

A specialized fixed offset pseudo-motor for moving two motors along a straight line.

Args:

- *x* (MotorBase): x-axis motor.
- *y* (MotorBase): y-axis motor.
- *slope* (float): slope of the line.
- *intercept* (float): y-intercept of the line.

Kwargs:

- *linked* (bool): Whether the two motors are linked. Two motors are linked if they can not be moved at the same time.

get_position()

Obtain the position of the *x* motor only.

class bcm.device.motor.RelVerticalMotor(*y1*, *y2*, *omega*, *offset=0.0*)

Bases: bcm.device.motor.MotorBase

A specialized pseudo-motor for moving an x-y stage attached to a rotating axis vertically. Such as a centering table attached to a goniometer. The current position is always zero and all moves are relative.

Args:

- *y1* (MotorBase): The first motor which is moves vertically when the angle of the

  axis is at zero.
- *y2* (MotorBase): The second motor which is moves horizontally when the angle of the axis is at zero.
- *omega* (MotorBase): The motor for the rotation axis.

Kwargs:

- *offset* (float): An angle correction to apply to the rotation axis position to make *y1* vertical.

2.2.10 Video

class bcm.device.video.VideoSrc(*name='Basic Camera'*, *maxfps=5.0*)
Bases: bcm.device.base.BaseDevice

Base class for all Video Sources. Maintains a list of listenners (sinks) and updates each one when the video frame changes.

Kwargs: *name* (str): Camera name. *maxfps* (flaot): Maximum frame rate.

add_sink(*sink*)
Add a video sink.

Args: *sink* (bcm.device.interfaces.IVideoSink provider).

del_sink(*sink*)
Remove a video sink.

Args: *sink* (bcm.device.interfaces.IVideoSink provider).

start()
Start producing video frames.

stop()
Start producing video frames.

get_frame()
Obtain the most recent video frame.

Returns: A Image.Image (Python Imaging Library) image object.

2.2.11 Miscellaneous

class bcm.device.misc.MotorShutter(*motor*)
Bases: bcm.device.base.BaseDevice

Used for CMCF1 cryojet Motor

is_open()
Convenience function for open state

class bcm.device.misc.DiskSpaceMonitor(*descr*, *path*, *threshold=0.9*, *freq=5.0*)
Bases: bcm.device.base.BaseDevice

An object which periodically monitors a given path for available space.

Args: *descr* (str): A description. *path* (str): Path to monitor.

Kwargs:

***threshold* (float): Warn if fraction of used space goes above this** value. Default (0.9).

freq (float): Frequency in minutes to check disk usage. Default (5)

class bcm.device.cryojet.CryojetNozzle(*name*)
Bases: bcm.device.misc.BasicShutter

A specialized in-out actuator for pneumatic cryoject nozzles at the CLS.

Args: name (str): Process variable root name.

class bcm.device.cryojet.Cryojet(*cname*, *lname*, *nname=''*)
Bases: bcm.device.base.BaseDevice

EPICS Based cryoject device object at the CLS.

Args: cname (str): Root name for EPICS cryojet record. lname (str): root name for EPICS cryo-level controller record.

Kwargs: nname (str): Root name of the EPICS cryoject nozzle record.

stop_flow()
Stop the flow of the cold nitrogen stream. The current setting for flow rate is saved.

resume_flow()
Restores the flow rate to the previously saved setting.

class bcm.device.diagnostics.DiagnosticBase(*descr*)
Bases: gobject._gobject.GObject

Base class for diagnostics.

get_status()
Check the state of the diagnostic.

Returns: tuple(int, str). The string is a status description while the value of the integer corresponds to one of:

0 - DIAG_STATUS_GOOD 1 - DIAG_STATUS_WARN 2 - DIAG_STATUS_BAD 3 - DIAG_STATUS_UNKNOWN 4 - DIAG_STATUS_DISABLED

class bcm.device.diagnostics.DeviceDiag(*device*, *descr=None*)
Bases: bcm.device.diagnostics.DiagnosticBase

A diagnostic object for generic devices which emits a warning when the device health is not good and an error when it is disconnected or disabled.

Args: *device* (a class::*device.base.BaseDevice* object) the device to monitor.

Kwargs: *descr* (str): Short description of the diagnostic.

class bcm.device.diagnostics.ServiceDiag(*service*, *descr=None*)
Bases: bcm.device.diagnostics.DiagnosticBase

A diagnostic object for generic services which emits an error when it is disconnected or disabled.

Args: *service* (a class::*service.base.BaseService* object) the service to monitor.

Kwargs: *descr* (str): Short description of the diagnostic.

2.3 Engines

2.3.1 Diffraction Engine

2.3.2 Spectroscopy Engine

This module defines classes aid interfaces for X-Ray fluorescence.

2.3.3 Centering Engine

Created on Jan 26, 2010

@author: michel

2.3.4 Scanning Engine

exception `bcm.engine.scanning.ScanError`
Bases: `exceptions.Exception`

Scan Error.

class `bcm.engine.scanning.AbsScan`(*mtr*, *start_pos*, *end_pos*, *steps*, *cntr*, *t*, *i0=None*)
Bases: `bcm.engine.scanning.BasicScan`

An absolute scan of a single motor.

class `bcm.engine.scanning.AbsScan2`(*mtr1*, *start_pos1*, *end_pos1*, *mtr2*, *start_pos2*, *end_pos2*, *steps*, *cntr*, *t*, *i0=None*)
Bases: `bcm.engine.scanning.BasicScan`

An Absolute scan of two motors.

class `bcm.engine.scanning.RelScan`(*mtr*, *start_offset*, *end_offset*, *steps*, *cntr*, *t*, *i0=None*)
Bases: `bcm.engine.scanning.AbsScan`

A relative scan of a single motor.

class bcm.engine.scanning.CntScan(*mtr*, *start_pos*, *end_pos*, *cntr*, *t*, *i0=None*)
Bases: bcm.engine.scanning.BasicScan

A Continuous scan of a single motor.

class bcm.engine.scanning.RelScan2(*mtr1*, *start_offset1*, *end_offset1*, *mtr2*, *start_offset2*, *end_offset2*, *steps*, *cntr*, *t*, *i0=None*)
Bases: bcm.engine.scanning.AbsScan2

A relative scan of a two motors.

class bcm.engine.scanning.GridScan(*mtr1*, *start_pos1*, *end_pos1*, *mtr2*, *start_pos2*, *end_pos2*, *steps*, *cntr*, *t*, *i0=None*)
Bases: bcm.engine.scanning.BasicScan

A absolute scan of two motors in a grid.

2.3.5 Fitting Engine

bcm.engine.fitting.peak_fit(*x*, *y*, *target='gaussian'*)

Returns the coefficients for the target function 0 = H - Height of peak 1 = L - FWHM of Voigt profile (Wertheim et al) 2 = P - Position of peak 3 = n - lorentzian fraction, gaussian offset

Success (boolean)

bcm.engine.fitting.histogram_fit(*x*, *y*)
calcfwhm(x,y) - with input x,y vector this function calculates fwhm and returns (fwhm,xpeak,ymax, fwhm_x_left, fwhm_x_right) x - input independent variable y - input dependent variable return ymax, fwhm, xpeak, x_hpeak[0], x_hpeak[1], cema cema is center of mass

success boolean

2.3.6 Scripting Engine

exception bcm.engine.scripting.ScriptError
Bases: exceptions.Exception

Exceptioins for Scripting Engine.

2.3.7 Miscelaneous Engines

Created on Oct 25, 2010

@author: michel This module defines classes for Optimizers.

class bcm.engine.optimizer.PitchOptimizer(*name*, *pitch_func*, *min_count=0.0*)
Bases: bcm.device.base.BaseDevice

Pitch Optimizer for 08B1-1

Created on Jan 26, 2010

@author: michel

2.4 Beamline

2.4.1 Interfaces

class bcm.beamline.interfaces.IBeamline(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)
Bases: zope.interface.Interface

A beamline object.

2.4.2 MX Beamline

class bcm.beamline.mx.MXBeamline(*console=False*)
Bases: object

MX Beamline(Macromolecular Crystallography Beamline) objects

Initializes a MXBeamline object from a python configuration file. The configuration file is loaded as a python module and follows the following conventions:

- Must be named the same as BCM_BEAMLINE environment variable followed by .py and placed in the directory defined by BCM_CONFIG_PATH. For example if the BCM_BEAMLINE is '08B1', the module should be '08B1.py'
- Optionally will also load a local module defined in the file $(BCM_BEAMLINE)_local.py e.g '08B1_local.py for the above example.
- **Global Variables:** BEAMLINE_NAME = Any string preferably without spaces BEAMLINE_TYPE = Only the string 'MX' for now BEAMLINE_ENERGY_RANGE = A tuple of 2 floats for low and hi energy limits BEAMLINE_GONIO_POSITION = Goniometer orientation according to XREC (i.e 1,2,3 etc) DEFAULT_EXPOSURE = A float for the default exposure time DEFAULT_ATTENUATION = A float for attenuation in % DEFAULT_BEAMSTOP = Default beam-stop position SAFE_BEAMSTOP = Safe Beam-stop position during mounting XRF_BEAMSTOP = Beam-stop position for XRF scans LIMS_API_KEY = A string MISC_SETTINGS = A dictionary containing any other key value pairs

will be available as beamline.config['misc']

DEVICES = A dictionary mapping device names to device objects. See SIM.py for a standard set of names.

CONSOLE_DEVICES = Same as above but only available in the console in addition to the above

SERVICES = A dictionary mapping service names to service client objects
BEAMLINE_SHUTTERS = A sequence of shutter device names for all shutters

required to allow beam to the end-station in the order in which they have to be opened.

Kwargs: console (bool): Whether the beamline is being used within a console or not. Used internally to register CONSOLE_DEVICES if True. Default is False.

`MXBeamline.setup()`

Setup and register the beamline devices from configuration files.

2.4.3 Remote Client

class `bcm.beamline.remote.BeamlineClient`

Bases: `gobject._gobject.GObject`

`BeamlineClient.setup(`**args*, ***kwargs*`)`

CHAPTER

THREE

ADDITIONAL INFORMATION

If you can't find the information you're looking for, have a look at the index of try to find it using the search function:

- *genindex*
- *search*

INDEX

E

F

G

H

I

M

P

R

S

V

W

X