MxDC Developer Documentation

Release 4.0.185-g20fd45f

March 13, 2012

CONTENTS

Part I
CONTENTS

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.

API DOCUMENTATION

This section is automatically generated from the code.

2.1 Devices

2.1.1 Base Device

class bcm.device.base.BaseDevice
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
  ```

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

`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.

`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")
```

class bcm.device.base.HealthManager(***kwargs*)
> 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.

> 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.

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

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

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

2.1.2 Automounter

class bcm.device.automounter.BasicAutomounter
> Basic Automounter object.

> An automounter object contains a number of :class:'AutomounterContainer's.

> **Signals:**

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

- *message*: messages from the Automounter

- *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.

`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)

`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
```

`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.

`mount`(*port*, *wash=False*, *wait=False*)

> Mount the sample at the specified port, optionally washing the sample in the process.

> **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)

`probe`()

> Command the automounter the probe for containers and port states.

`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.

2.1.3 Counters

class `bcm.device.counter.Counter`(*pv_name*, *zero=0.0*)

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

> `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.

2.1.4 Detector

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

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

> `get_origin`()

> > Obtain the detector origin/beam position in pixels.

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

> `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.

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.

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.

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.

wait(*state='idle'*)

Wait until the detector reaches a given state.

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

2.1.5 Diffractometer

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

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

`stop()`
Stop both child devices.

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

2.1.6 Goniometer

class `bcm.device.goniometer.Goniometer`(*name, blname, mnt_cmd, minibeam*)

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

`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

`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.

`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.

class `bcm.device.goniometer.MD2Goniometer`(*name*)

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

`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

`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.

`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.

`stop()`

    Stop and abort the current scan if any.

class `bcm.device.goniometer.BackLight(`*name*`)`

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

2.1.7 Multi-Channel Analyzer

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

    mcaRecord based single element fluorescence detector object.

class `bcm.device.mca.VortexMCA(`*name*, *channels=2048*`)`

    EPICS based 4-element Vortex ME4 detector object.

2.1.8 Monochromator

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

```
stop()

wait()
```

2.1.9 Motors

class bcm.device.motor.MotorBase(*name*)
>   Base class for motors.

exception bcm.device.motor.MotorError
>   Base class for errors in the motor module.

2.1.10 Video

exception bcm.device.video.VideoError
>   Base class for errors in the video module.

2.1.11 Miscellaneous

class bcm.device.misc.MotorShutter(*motor*)
>   Used for CMCF1 cryojet Motor
>
>   is_open()
>   >   Convenience function for open state

class bcm.device.cryojet.Cryojet(*cname, lname, nname=''*)
>   EPICS Based cryoject device object at the CLS.
>
>   resume_flow()
>   >   Restores the flow rate to the previously saved setting.
>
>   stop_flow()
>   >   Stop the flow of the cold nitrogen stream. The current setting for flow rate is
>   >   saved.

class bcm.device.cryojet.CryojetNozzle(*name*)
>   A specialized in-out actuator for pneumatic cryoject nozzles at the CLS.

class bcm.device.diagnostics.DeviceDiag(*device, descr=None*)
>   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.

class bcm.device.diagnostics.DiagnosticBase(*descr*)
>   Base class for diagnostics.
>
>   get_status()
>   >   Check the state of the diagnostic.

Returns:

int. One of the following: 0 - DIAG_STATUS_GOOD 1 - DIAG_STATUS_WARN 2 - DIAG_STATUS_BAD 3 - DIAG_STATUS_UNKNOWN

class bcm.device.diagnostics.ServiceDiag(*service, descr=None*)
    A diagnostic object for generic services which emits an error when it is disconnected or disabled.

class bcm.device.diagnostics.ShutterStateDiag(*device, descr=None*)
    A diagnostic object for shutters which emits a warning when the shutter is closed and an error if it is inactive.

2.2 Engines

2.2.1 Diffraction Engine

2.2.2 Spectroscopy Engine

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

2.2.3 Centering Engine

Created on Jan 26, 2010

@author: michel

2.2.4 Scanning Engine

class bcm.engine.scanning.AbsScan(*mtr, start_pos, end_pos, steps, cntr, t, i0=None*)
    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*)
    An Absolute scan of two motors.

class bcm.engine.scanning.CntScan(*mtr, start_pos, end_pos, cntr, t, i0=None*)
    A Continuous scan of a single motor.

class bcm.engine.scanning.GridScan(*mtr1, start_pos1, end_pos1, mtr2, start_pos2, end_pos2, steps, cntr, t, i0=None*)
    A absolute scan of two motors in a grid.

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

A relative 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)

A relative scan of a two motors.

exception bcm.engine.scanning.ScanError
Scan Error.

2.2.5 Fitting Engine

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

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)

2.2.6 Scripting Engine

exception bcm.engine.scripting.ScriptError
Exceptioins for Scripting Engine.

2.2.7 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*)
Pitch Optimizer for 08B1-1

Created on Jan 26, 2010

@author: michel

2.3 Beamline

2.3.1 Interfaces

class bcm.beamline.interfaces.IBeamline(*name, bases=(), attrs=None, __doc__=None, __module__=None*)

A beamline object.

2.3.2 MX Beamline

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

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.

MXBeamline.setup()
    Setup and register the beamline devices from configuration files.

2.3.3 Remote Client

class bcm.beamline.remote.BeamlineClient

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

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