# MxDC Developer Documentation

*Release 4.0.171-g929974f*

March 01, 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 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 as a parameter. *True* represents a change from inactive to active, and *False* represents the opposite.
- *busy*: emitted to notify listeners of a change in the busy state of the device. 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 the following:
  - 0 : No error
  - 1 : MINOR, no impact to device functionality. No attention needed.
  - 2 : MARGINAL, no immediate impact to device functionality but may impact future functionality. Attention may soon be needed.
  - 4 : SERIOUS, functionality impacted but recovery is possible. Attention needed.
  - 8 : CRITICAL, functionality broken, recovery is not possible. Attention needed.
  - 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.

`add_devices(`**args*`)`
: 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

`is_active()`
: Convenience function for checking if the device is active. Returns a boolean.

`is_busy()`
: Convenience function for checking if the device is busy. Returns a boolean.

`set_state(`*`**kwargs`*`)`

Set the state of the device.

Set the state of the device and emit the appropriate signal based on the specified keyworded arguments. Keyworded arguments follow the same conventions as the state dictionary and correspond to any signals defined for the device. For example:

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

Signals will be emitted for the specified keyworded arguments, which be defined as supported signals of the device.

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

An object which manages the health state of a device.

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

Automounter Device objects

An automounter object is an event driven `GObject` which contains a number of Automounter Containers.

**Each Automounter device obeys the following interface:** signals:

***state*:** a signal which transmits changes in automounter states. The The signal data is a dictionary with three entries as follows: {'busy': <boolean>, 'enabled': <boolean>, 'needs':[<str1>,<str2>,...]}

***message*:** a signal which emits messages from the Automounter

***mounted*:** a signal 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. Transmitted data is a tuple of the form (<% complete>, <description>)

methods:

**probe(probe_string)** command the automounter the probe for containers and port states as specified by the DCSS compatible probe string provided.

**mount(port, wash=False)** mount the sample at the specified port, optionally washing the sample in the process.

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

**class** `bcm.device.automounter.AutomounterContainer`(*location*, *status_str=None*)
An event driven object for representing an automounter container.

**Signals:** *changed*: Emits the *changed* GObject signal when the state of the container changes.

`configure`(*status_str=None*)
This method sets up the container type and state from a status string. If no status string is provided, it resets the container to the 'unknown' state.

**class** `bcm.device.automounter.BasicAutomounter`
Basic Automounter objects. Contains a number of Automounter Containers.

**Signals:**

***state*:** a signal which transmits changes in automounter states. The The signal data is a string

***mounted*:** a signal emitted when a sample is mounted. The data transmitted is a tuple of the form (<port no.>, <barcode>) when mounting and `None` when dismounting.

`get_port_state`(*port*)
Returns the current state of the specified port.

`is_mountable`(*port*)
Returns true if the specified port can be mounted without issues. Does not guarantee that the port is actually mountable, only that the the port was marked as mountable since the last probe operation.

`is_mounted`(*port=None*)
Returns true if the specified port is currently mounted. If no port is specified, return true if any port is mounted.

`parse_states`(*state*)
: This method sets up all the container types and states from a DCSS type status string. If no status string.

**class** `bcm.device.automounter.ManualMounter`
: Basic Automounter objects. Contains a number of Automounter Containers.

  **Signals:**

  ***mounted*:** a signal emitted when a sample is mounted. The data transmitted is a tuple of the form (<port no.>, <barcode>) when mounting and `None` when dismounting.

### 2.1.3 Counters

Counter Device objects

A counter device enables counting and averaging over given time periods.

Each Counter device obeys the following interface:

methods:

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

### 2.1.4 Detector

### 2.1.5 Diagnostics

Created on Jun 1, 2010

@author: michel

**class** `bcm.device.diagnostics.DiagnosticBase`(*descr*)
: Base class for diagnostics.

### 2.1.6 Diffractometer

**exception** `bcm.device.diffractometer.DiffractometerError`
: Base class for errors in the diffractometer module.

### 2.1.7 Goniometer

### 2.1.8 Multi-Channel Analyzer

### 2.1.9 Monochromator

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

`stop()`

`wait()`

### 2.1.10 Motors

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

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

### 2.1.11 Video

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

### 2.1.12 Miscelanous

**class** `bcm.device.misc.MotorShutter`(*motor*)
Used for CMCF1 cryojet Motor

`is_open()`
Convenience function for open state

## 2.2 Beamline

### 2.2.1 Interfaces

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

### 2.2.2 MX Beamline

MX Beamline(Macromolecular Crystallography Beamline) objects

This module creates MXBeamline objects (class MXBeamline) from a python configuration file 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.

**class** `bcm.beamline.mx.MXBeamline(`*console=False*`)`
An MX Beamline

`MXBeamline.setup()`
Set up and register the beamline devices.

### 2.2.3 Remote Client

**class** `bcm.beamline.remote.BeamlineClient`

`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