MView Manual

Noah Meltzer

Table of Contents

What is MView?

At the highest level, MView is a framework that can be used to interface with virtually any external data source. In short, MView takes care of the overhead involved with running a GUI and device communication so that the end user can concentrate on their data.

Using MView, a variety of control and display elements can be easily configured. These include:

- Buttons
- Numerical/Textual Readouts
- Plots

In addition, some backend features include:

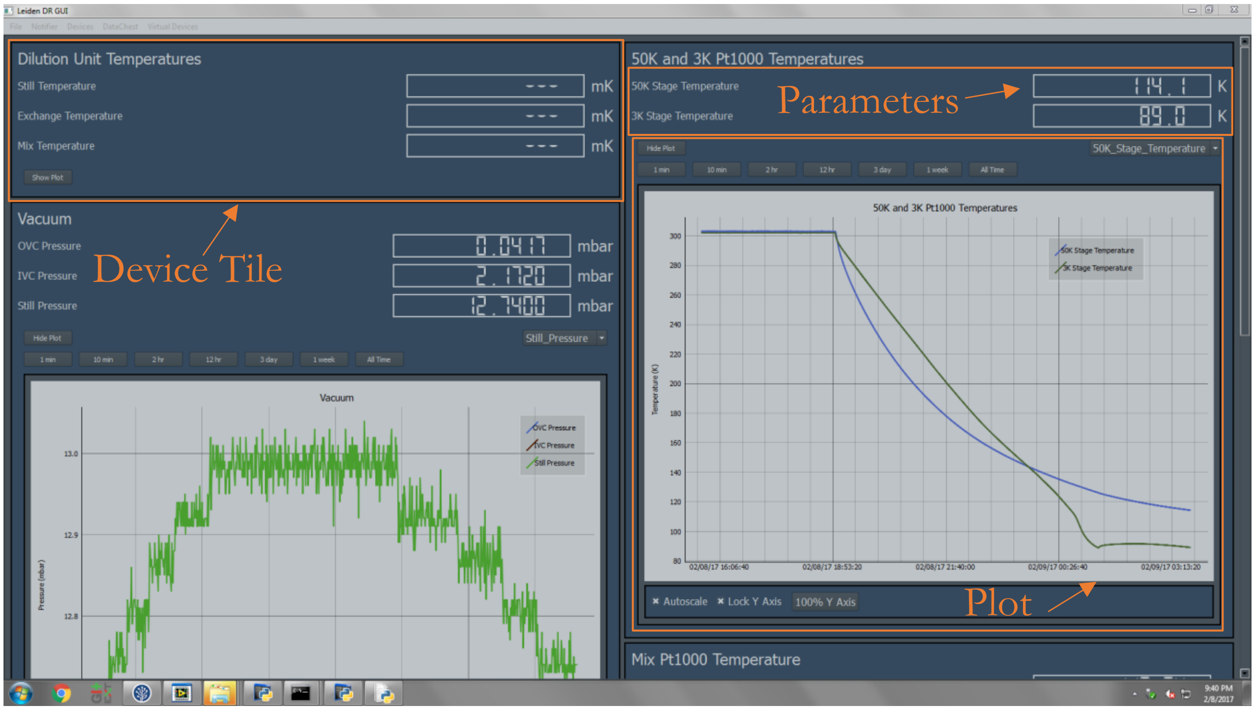
- Datalogging using Datachest
- Email/Text notifications
- Logic

MView accomplishes this by wrapping low-level communication and overhead into a **device driver** that abstracts all sources of data (**devices**) into an **MDevice** object. All MDevice objects implement a common software interface. This allows even the most complex devices to be handled in a simple manner.

With this system in place, MView implements a range of error-checking, graphing, and logging functionality which the end user can use with ease.

While not nearly as powerful as LabVIEW, MView is easy to set up and it works. This makes it ideal for simple monitoring tasks where the massive overhead of LabView is not needed. Additionally MView is also accessible by anyone who knows even the smallest amount of python without the need to learn G.

The MView Interface



MView for Our Purposes

Setting up a simple GUI for LabRad Devices

Setting up a new MView GUI is generally very easy. In this section, we will configure a simple MView project that monitors data from a LabRad device.

In order to create a GUI using MView, we must set up a program that configures our devices and tells MView how to behave.

Before You start:

- ***Please refer to the DataChest manual for DataChest setup.***
- ***Please refer to the comments in telecomm.py for telecomm server setup.***

Step 1: Imports

The first thing that must be done is to import the necessary MView libraries.

```
import MGui # Handles all GUI operations. Independent of LabRAD.
from MDevices.Device import Device # This is the device driver that represents a LabRad
server
```

MGui: Handles the overhead of initializing MView

MDevices.Device: All device drivers are stored in the MDevices folder. Device is the device driver that talks to LabRad servers.

Step 2: Initialization

Next, we need to write the class that initializes MView as well as all Devices.

```
class MyGuiClass:
  my_gui = self.gui = MGui.MGui()
```

my_gui: Holds a reference to the MVeiw Gui.

Step 3: Create a LabRad and Telecomm Server Connection

A connection to LabRad is created so that it can be passed to devices.

```
    try:
      # Attempt to establish a labrad connection.
      cxn = labrad.connect()
    except:
      # If no connection can be made, abort with an error message.
      print("Please start the LabRAD manager")
      time.sleep(2)
      sys.exit(0)
    try:
      # As of writing, there is one class in MView itself that is dependent
      # on LabRad, and it requires the telecomm server to be running.
      # This is subject to change.
      tele = cxn.telecomm_server
    except:
      # If no connection can be made, abort with an error message.
      print("Please start the telecomm server")
      time.sleep(2)
      sys.exit(1)
```

NOTE: The code in steps 1-3 is the same for all MView GUIs.

Step 4: Initializing Devices

A) Instantiating a new device

We must now initialize the LabRad Devices. Let's create a device that represents a CP2800 compressor.

First, we instantiate a new Device:

```
    Compressor = Device("Compressor")
```

There are optional keyword arguments which can also be specified:

- `lock_logging_settings = False`
  Prevent the user from editing the datalogging settings on the GUI. Note that MView still has full control over locked settings.
- `default_log_location = None`
  The default datalogging location, must be inside DataChest root. This will override the normal restoring of previous log locations when MView is opened and closed. i.e. MView will automatically switch its log location here upon opening.

This creates a device called "Compressor." Not much else happens until we tell it how to communicate.

Second, we pass it a reference to our connection:

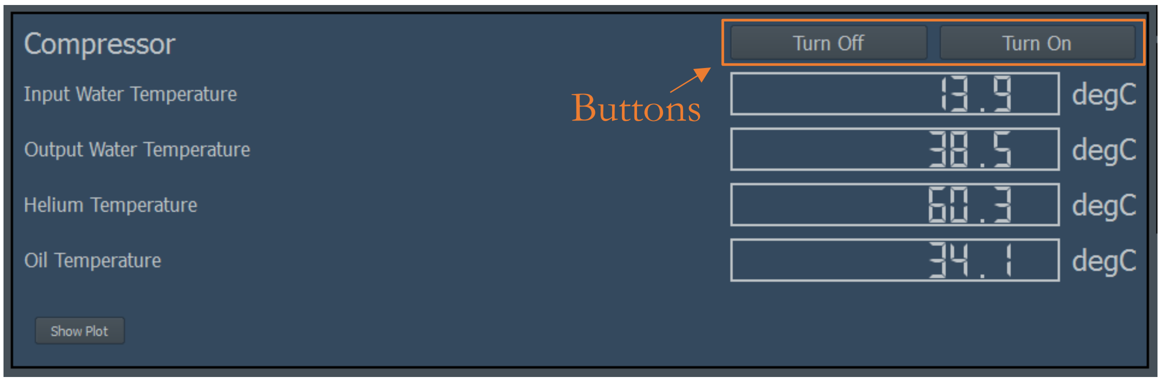
```
Compressor.connection(cxn)
```

Third, we tell it what the name of the server is. The name of the server for the CP2800 is "cp2800_compressor."

```
Compressor.setServerName("cp2800_compressor")
```

Now we have a basic MDevice. We next need to define buttons (if we want them), and parameters.

B) Adding Buttons



Next, let's add a button that turns off the compressor when we click it. This is done using the MDevice.addButton(label, message, setting, setting arguments) method.

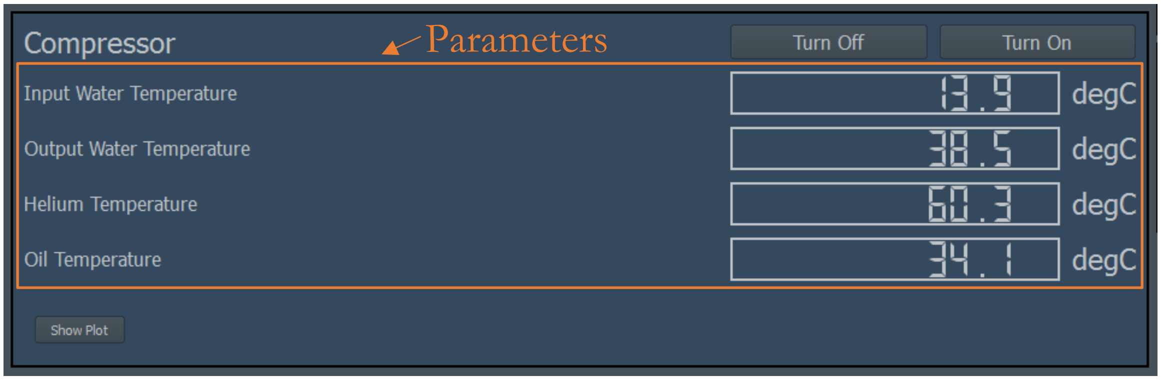
```
MDevice.addButton(label, message, setting, setting arguments)
```

Here is how it should look in the context of this example:

```
Compressor.addButton("Turn Off",
   "You are about to turn the compressor off.",
   "stop", None)
Compressor.addButton("Turn On",
   "You are about to turn the compressor on.",
   "start", None)
```

The first argument sets the button's text. The second argument is the text to be displayed in a warning popup, if no warning is to be displayed, then the second argument should be None. The third argument is the LabRad server setting that is triggered when the button is pushed, and the fourth argument is an array of arguments for the LabRad setting, None if no arguments.

C) Adding Parameters



It is now time to add parameters to the device's tile. This is done with the MDevice.addParameter() method.

```
MDevice.addParameter(Label, LabRad Setting, Labrad Setting Arguments)
```

Optional Keyword arguments include:

- `show = True`
  Display the parameter on the GUI.
- `units = None`
  The recommended units, used to override current unit. Default LabRad server units are used if this is not specified, as well as if the specified units are incompatible with the measurement.
- `Precision = 2`
  The precision of the measurement **on the display**. Full precision is used by MView internally.
- `Index = None`
  If the reading is part of an array, this is the index.
- `Log = True`
  Log this parameter.

Here is how it should look in the context of this example:

```
Compressor.addParameter("Input Water Temperature",
   "current_temperatures_only", None, index=0)
Compressor.addParameter("Output Water Temperature",
   "current_temperatures_only", None, index=1)
Compressor.addParameter("Helium Temperature",
   "current_temperatures_only", None, index=2)
Compressor.addParameter("Oil Temperature",
   "current_temperatures_only", None, index=3)
```

Step 5: Selecting the Device

Just as with any LabRad device, we must call the 'select_device' command. This is done in the following way:

```
Compressor.selectDeviceCommand("select_device", 0)
```

This selects device 0.

Step 6: The Plot

To add a plot to our graph, the MDevice.addPlot() method can be called.

```
Compressor.addPlot()
```

To set the y-axis label, the following command is used:

```
MDevice.setYLabel(y-axis label, custom units = None)
```

The first argument is the label displayed on the y-axis, and the second is an optional override of the default units. For example, it is a good idea to use this override when the server does give units. In our case, this takes the form of

```
Compressor.setYLabel("Temperature")
```

Step 7: Begin()

The next thing we must do is to tell the device to start. This is done with the MDevice.begin() method.

```
Compressor.begin()
```

Step 8: Add Device to Gui

We can now add our device to the MView gui.

```
self.gui.addDevice(Compressor)
```

Step 8: Starting MView

We start MView using the MGui.startGui() method.

```
startGui(self, title, tele, autostart=True):
```

- **title:** The title on the gui.
- **tele:** Reference to the telecom server.
- **autostart:** Allows gui to run when startGui() is called. The purpose of this option will be discussed in a later section.

```
self.gui.startGui('Leiden DR GUI',tele)
```

Step 9: Calling __init__()

As with any python class, we must call our init method **outside** of the main class.

```
class myGuiClass
...
viewer = myGuiClass()
viewer.__init__()
```

Step 10: Putting it All Together

Here is how our new piece of code should look.

```
# Copyright (C) 2016 Noah Meltzer
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 2 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program. If not, see <http://www.gnu.org/licenses/>.

import sys
import time
from tendo import singleton

import labrad

from dataChestWrapper import *
import MGui # Handles all GUI operations. Independent of LabRAD.
from MDevices.Device import Device
from MDevices.Mhdf5Device import Mhdf5Device

class nViewer:
  gui = None
  devices = []
  def __init__(self, parent = None):
    # Establish a connection to LabRAD.
    try:
      # Thiss will sys.exit(-1) if other instance is running.
      me = singleton.SingleInstance()
    except:
      print("Multiple instances cannot be running")
      time.sleep(2)
      sys.exit(1)
    try:
      cxn = labrad.connect() # Attempt to establish a labrad connection.
    except:
      # If no connection can be made, abort with an error message.
      print("Please start the LabRAD manager")
      time.sleep(2)
      sys.exit(0)
    try:
      # As of writing, there is one class in MView itself that is dependent
      # on LabRad, and it requires the telecomm server to be running.
      # This is subject to change.
      tele = cxn.telecomm_server
    except:
      # If no connection can be made, abort with an error message.
      print("Please start the telecomm server")
      time.sleep(2)
      sys.exit(1)
    self.gui = MGui.MGui()
```

```
    Compressor = Device("Compressor")
    Compressor.connection(cxn)
    Compressor.setServerName("cp2800_compressor")
    Compressor.addButton("Turn Off",
       "You are about to turn the compressor off.",
       "stop", None)
    Compressor.addButton("Turn On",
       "You are about to turn the compressor on.",
       "start", None)
    Compressor.addButton("Elapsed Time",
       None,
       "elapsed_time", None)
    Compressor.addParameter("Input Water Temperature",
       "current_temperatures_only", None, index =0)
    Compressor.addParameter("Output Water Temperature",
       "current_temperatures_only", None, index =1)
    Compressor.addParameter("Helium Temperature",
       "current_temperatures_only", None, index =2)
    Compressor.addParameter("Oil Temperature",
       "current_temperatures_only", None, index =3)
    Compressor.selectDeviceCommand("select_device", 0)
    Compressor.setYLabel("Temperature")
    Compressor.addPlot()
    Compressor.begin()
    self.gui.addDevice(Compressor)
    # Create the gui.

    self.gui.startGui('Leiden DR GUI',tele)


# In Python, the main class's __init__() IS NOT automatically called.
viewer = nViewer()
viewer.__init__()
```

Devices in MView

MView Device Structure

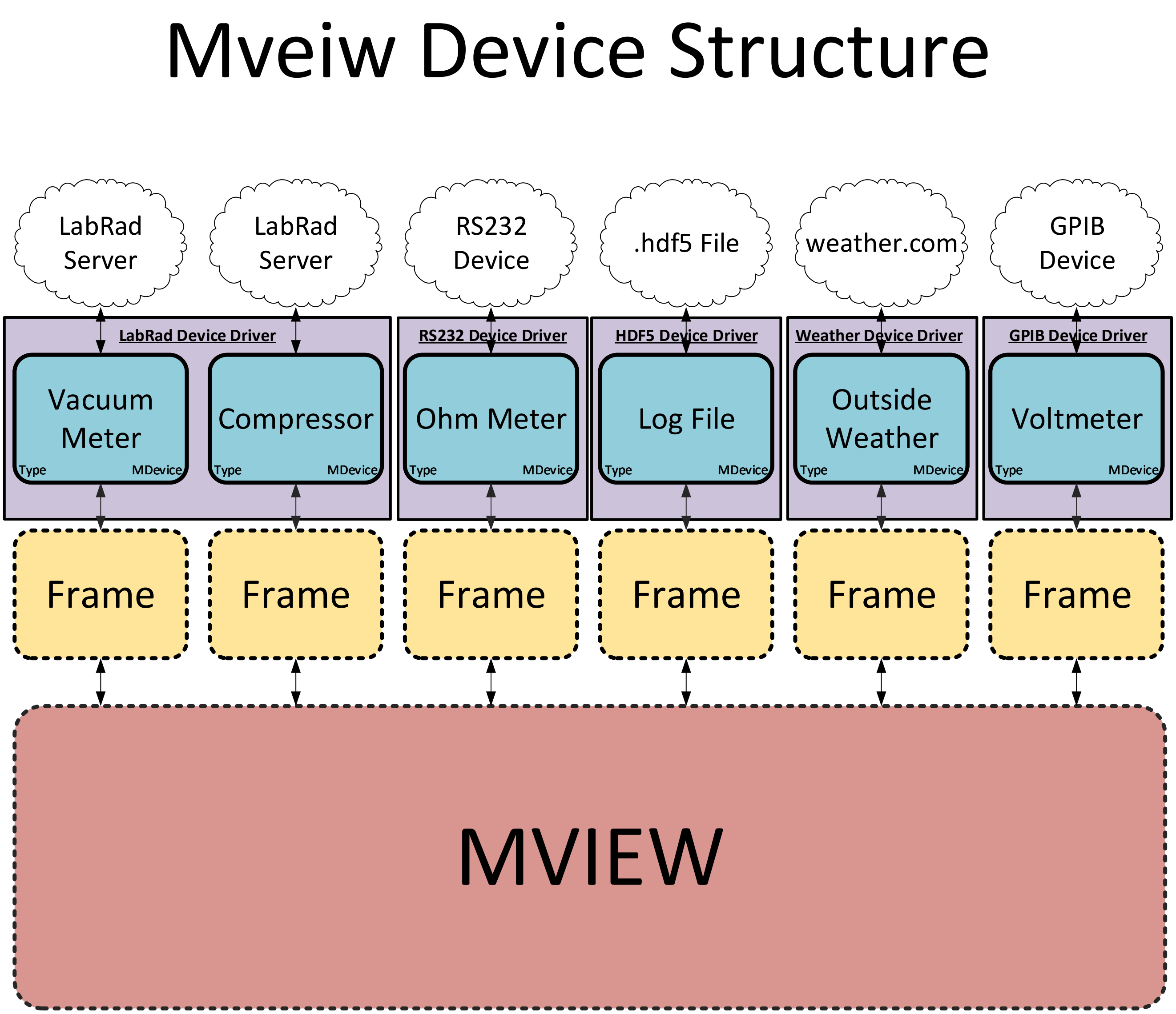


Figure 1: The structure of devices in MView

The MDevice Class

MView uses the MDevice class to give all sources of data a common interface with which to interact in the context of MView. These sources of data can be anything including but not limited to LabRad servers, RS232 devices, GPIB Devices, they can even represent the contents of .hdf5 files.

Devices in MView are created by instantiating their device drivers. For example, if there are two RS232 devices, we create two instances of the RS232 device driver. This means that only one generic device driver needs to be created for each interface (RS232, LabRad Servers, HDF5 files, etc.) and it can then be applied to all devices that use the same interface.

Special fuctions

There are a few special functions that are made available to use when the parent class is MDevice.

- `MDevice.onLoad()`

  Called at the end of MGui.startGui(), when the main MView GUI has finished loading. This allows the MDevice to configure pieces of MView only available once the program has fully loaded.

- `MDevice.onBegin()`

  Called at the end of MDevice.begin(). This is called before MView starts. This allows us to configure settings that MView might use while starting. This might include datalog locations or device-specific information.

- `MDevice.onAddParameter(*args)`

  Called when when a new parameter is added. It is passed whatever MDevice.addParameter() is passed. (Note: MDevice.onAddParameter() and MDevice.addParameter() are different).

  Warning: Never override the MDevice.addParameter() method. The MDevice.onAddParameter() method is meant to be overridden. All arguments and keyword arguments passed to MDevice.addParameter() are also passed to MDevice.onAddParameter().

- `MDevice.query()`

  Automatically called periodically, determined by MDevice.Mframe.getRefreshRate(). There is also a MDevice.Mframe.setRefreshRate() function with which the refresh rate can be configured. This is where any MDevice Query's its hardware.

- `MDevice.prompt(button)`

  Called when a device's button is pushed. Button is an array which is associated with the button. The array is constructed in the device driver code, and the PyQT button is then appended to the end by MView. The array associated with the button is passed to prompt()

in the device driver. The device driver then determines what to do based on the button pushed.

The MFrame Class

Each device has a something called a 'frame,' which is an instance of the MFrame class. The frame serves a few purposes:

- Note the 'Frame' between MView and MDevice shown in figure 1. Because the all MDevices run on their own thread, they cannot update the GUI directly. The MFrame class provides a thread-safe way for MView to interact with MDevices by forcing all information about a device into exclusively thread-safe data structures which can be accessed either by MView or and MDevice at any time.
- The MFrame holds all information about an MDevice. This includes everything from current readings to a reference to the MDeviceContainer that represents the device on the GUI to all of the logged data and more. Think of an MDevice as representing student taking notes in class, the MFrame represents that student's notebook. The notebook holds all information beyond what it makes sense for the student to hold in their memory. While it can be argued that the MFrame and MDevice classes should be combined, the two classes were separated for the sake of semantics.

Warning: Although this is a more advanced topic that the end user probably won't need to worry about, it is worth mentioning that MDevice is a child of QThread. This means that all devices run on a thread separate from one another and from the main MView GUI. These threads are known as worker threads as opposed to the main thread which runs the GUI. This is important because functions in the main thread *absolutely should not* be called by worker threads, doing so causes undefined behavior. More simply, never try to update the GUI directly from a device driver, always use the frame.

MDevice API

See website.

Writing a Device Driver

We will now write a device driver that communicates with RS232 devices.

Things you should know

MView was written to be expandable by the user. In order to make this work, creation of new device drivers is easy and do-able without extensive knowledge of python or MView itself.

All device drivers are children of the MDevice class, which handles much of the overhead when interacting with MView and handling threading.

All devices device drivers should go in the mView/MDevices/ folder.

What it should do

Let’s think about what this device driver must be able to do. In order to communicate with a serial device, we need to do the following:

1. Connect the device’s respective COM port.
2. Send the device a serial command.
3. Receive a serial response.

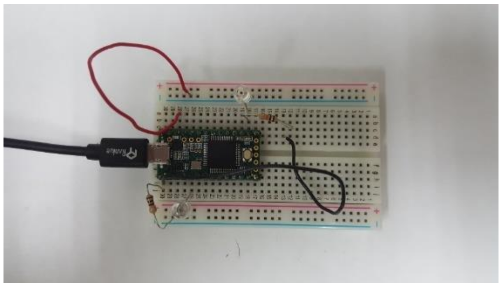
How we will do it

PySerial is a good tool for serial communication in python, so it is what we will use for the device driver.

The Device

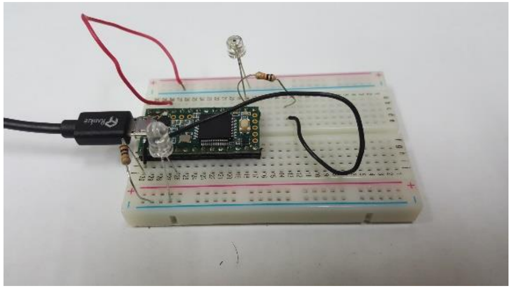
We will use a simple serial device to test our new device driver. Let’s call this device the Light Meter 3000, it has a light sensor, as well as an led.. Note that this device is just to test our driver; however, our driver should be generic and work with all serial devices. To read from the light sensor, the character ‘s’ is sent to the device, and a value terminated by ‘\r\n’ is returned.

```
>>> import serial
>>> device = serial.Serial("COM7", timeout=1)
>>> device.write('s')
1L
>>> lightlevel = device.readline()
>>> lightlevel
'17\r\n'
>>> int(lightlevel.strip())
17
```

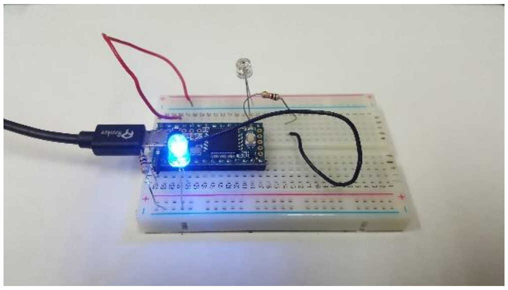


The led can be lit to 10 different brightness’s by sending the character ‘b’ immediately followed by a number 0-9.

```
>>> device.write('b0')
2L
```



```
>>> device.write('b9')
2L
```



Writing The Code

NOTE: This device driver is a simple example, for the sake of simplicity, this tutorial will not include error-checking or recovery from errors.

Before we write the actual class, lets insert a code header and import statements.

```
# Copyright (C) 2016 Noah Meltzer
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 2 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program. If not, see <http://www.gnu.org/licenses/>.

__author__ = "Noah Meltzer"
__copyright__ = "Copyright 2016, Noah Meltzer, McDermott Group"
__license__ = "GPL"
__version__ = "0.0.1"
__maintainer__ = "Noah Meltzer"
__status__ = "Beta"

# Import the parent class
from MDevice import MDevice
# We will use the pyserial class to interface with COM ports.
import serial
import time
# Traceback is good for printing errors and debugging.
import traceback
# Import necessary Qt libraries
from PyQt4 import QtGui
```

Next, let's create the class. Notice that MDevice is the parent class

```
class RS232Device(MDevice):
```

Initializing Variables

The __init__ function is called when the device is instantiated. We will use it to initialize some variables.

```
def __init__(self, *args, **kwargs):
  '''Initialize variables. Most of the device parameters cannot yet be initialized.'''
  super(RS232Device, self).__init__(*args)
  # The name of the device is passed as the first argument.
  self.name = args[0]
  # The name of the port (i.e. COMx) is given as the second argument.
  self.portname = args[1]
  # This dictionary will hold basic information about different parameters.
  # This dictionary will just be used by us.
  self.paramInfo = {}
  # This will hold a reference to the pySerial instance that
  # is used to communicate with the device.
  self.port = None
  # Is the device connected?
  self.connected = False
  # The serial timeout, default 10ms.
  self.timeout = kwargs.get("timeout",10)
  # Default baud rate is 9600.
  self.baud = kwargs.get("baud", 9600)
```

Implementing onAddParameter()

Time to handle a new parameter being added. We will override the MDevice.onAddParameter() function for this one. Remember, **never override MDevice.addParameter().**

```
def onAddParameter(self, paramName, command, *args, **kwargs):
  # Look for keyword arguments
  precision = kwargs.get("precision", None)
  units = kwargs.get("units", None)
  # Add a key to our dictionary that holds another dictionary.
  self.paramInfo[str(paramName)] = {}
  thisParam = self.paramInfo[str(paramName)]
  # Add some more keys to our dictionary.
  thisParam["name"] = paramName
  thisParam["command"] = command
  thisParam["precision"] = precision
  thisParam["units"] = units
```

Implementing Buttons

Now it is time to allow for buttons to be added. To do this, we implement the addButton() method which creates a button. A button is a list whose contents are defined by the user. There is no strict requirement for buttons other than that they are of type list. The list holds any commands or warning messages to be used by our prompt() function later.

```
def addButton(self, text, command, **kwargs):
  # Look for a message keyword argument
  message = kwargs.get("message", None)
  # Build the list.  The QPush button object will be appended to the end of this list.
  button = []
  # We will make the first element of the list hold
  # the text that is displayed on the button.
  button.append(text)
  # The second element will be the command sent to the device
  # if the button is clicked.
  button.append(command)
  # We can include a warning message too.
  button.append(message)
  # Add the button to the gui
  self.addButtonToGui(button)
```

The MDevice.prompt() function is called whenever a button is pressed, the button we created in the addButton() function is also passed to prompt. This means we have access to all messages, commands, and even the PyQt pushbutton which means that we can do things like change the color of the button based on state. Here is the implementation:

```
def prompt(self, button):
  # We stored the warning message in the 3rd element of the button array.
  # If it is None, do not display a warning message, if it is not, display about
  # warning using the QMessageBox class.
  if button[2] is not None:
    # Create a QMessageBox
    msg = QtGui.QMessageBox()
    # Set the icon to an exclamation point.
    msg.setIcon(QtGui.QMessageBox.Warning)
    # Our warning message text is in the 3rd element of our array.
    msg.setText(button[2])
    # Add ok and cancel buttons.
    msg.setStandardButtons(QtGui.QMessageBox.Ok | QtGui.QMessageBox.Cancel)
    # The window title.
    msg.setWindowTitle("Warning")
    # Execute the class and retrieve the value clicked.
    retval = msg.exec_()
    # If ok was clicked then send the command to the serial device.
    if retval == QtGui.QMessageBox.Ok:
      # The command was put into the 2nd element of the array.
      self.port.write(button[1])
  # If no warning message was given, then just go ahead and send the command.
  else:
    # The command was put into the 2nd element of the array.
    self.port.write(button[1])
```

Implementing setPort and setYLabel

The setPort and setYLabel methods are implemented as follows:

```
def setPort(self, portname, timeout = 10):
  '''Set the name of the port. i.e. COMx. Keyword args: timeout = 10.'''
  self.portname = portname
  self.timeout = timeout

def setYLabel(self, yLbl, units=''):
  '''Set the label to be displayed on the independent variable axis.'''
  self.frame.setYLabel(yLbl, units)
```

Implementing initialization

We will now implement the MDevice.onBegin() function that is called when the MDevice.begin() function is called. This happens once the device is set up, and it should initiate a the last steps that need to be completed in order to have a working device. This includes connecting to the device.

```
def onBegin(self):
  '''Begin the device.'''
  self.connect()
```

Connecting

Now, we implement the connect function. This should open a port.

```
def connect(self):
  '''Open the serial port and try to connect to it.'''
  try:
    self.port = serial.Serial(self.portname,
                              int(self.baud),
                              timeout=self.timeout,
                              write_timeout = 10)
    self.connected = True
    return self.port
  except:
    traceback.print_exc()
    self.connected = False
    print "ERROR: port:", self.port, "will try again."
    return None
```

Querying the device

The query function is called with a period defined by MDevice.frame.getRefreshRate(). At the end it should set the readings by calling self.frame.setReadings([readings]). Where [readings] is an array of readings.

```
def query(self):
```

```
'''The query function is called with a period defined
     by MDevice.frame.getRefreshRate().  At the end it
     should set the readings by calling self.frame.setReadings([readings]).
     Where [readings] is an array of readings.
  '''
try:
  readings = []
  if not self.connected:
    # if not connected, connect.
    self.connect()
  # If self.port is not None.
  if self.port is not None:
    # For each parameter, get readings
    for param in self.getParameters():
      print "param:",param
      # Flush anything that might on the port.
      self.port.flush()
      # Write the command to the port.
      command = self.paramInfo[param]['command']
      if command != None:
        self.port.write(self.paramInfo[param]['command'])
        # Wait while there is nothing on the port.
        while(not self.port.in_waiting > 0):
          time.sleep(0.01)
        # While there is stuff on the port, read it.
        reading = self.port.readline()
        # Strip whitespace and newline characters off the received value.
        reading = int(reading.strip())
      # Tell the device what the readings were.
        self.setReading(param, reading)
except:
  try:
    self.port.close()
  except:
    pass
  traceback.print_exc()
  self.connected = False
```

Closing the Port

The MDevice.close() method is called when MView is closed. In that function we tidy up after ourselves, in this case we should close the port we opened.

```
def close(self):
  '''Stop the device. This includes closing the port.'''
  try:
    self.port.close()
  except:
    print "Could not close port."
```

The Associated GUI

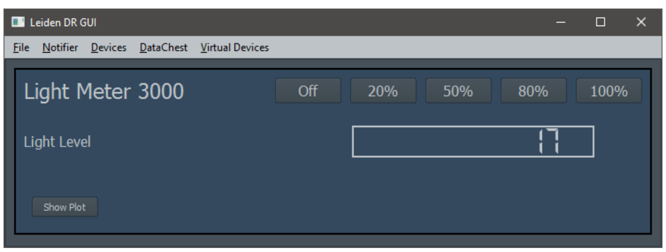
Here is the python program that will initialize the light meter 3000's MView GUI.

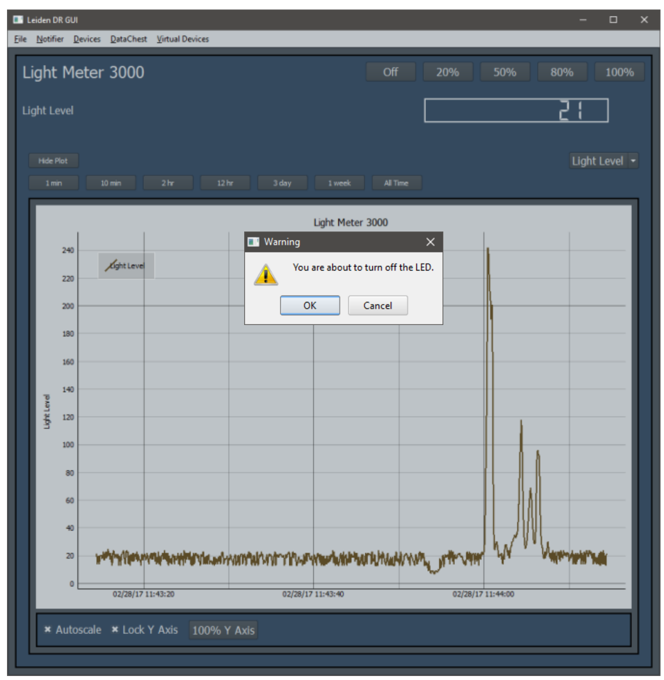
```
# Copyright (C) 2016 Noah Meltzer
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 2 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program. If not, see <http://www.gnu.org/licenses/>.


"""
version = 1.1.1
description = Demonstration
"""

import sys
sys.dont_write_bytecode = True
import MGui  # Handles all GUI operations. Independent of  LabRAD.

from MDevices.RS232Device import RS232Device

from MNodeEditor import MNodeTree
from MNodeEditor.MNodes import MDeviceNode
from MNodeEditor.MNodes import runningAverage

import labrad
import time

from tendo import singleton

class mViewer:
  gui = None
  devices = []

  def __init__(self, parent = None):
    # Establish a connection to LabRAD.
    try:
      me = singleton.SingleInstance() # will sys.exit(-1) if other instance is running
    except:
      print("Multiple instances cannot be running")
      time.sleep(2)
      sys.exit(1)
    try:
      cxn = labrad.connect()
    except:
      print("Please start the LabRAD manager")
      time.sleep(2)
      sys.exit(0)
    try:
      tele = cxn.telecomm_server
    except:
      print("Please start the telecomm server")
      time.sleep(2)
      sys.exit(1)

    self.gui = MGui.MGui()
    lm3000 = RS232Device("Light Meter 3000", "COM7", baud = 115200)
    lm3000.addButton("Off", 'b0', message = "You are about to turn off the LED.")
    lm3000.addButton("20%", 'b2')
    lm3000.addButton("50%", 'b5')
```

```
    lm3000.addButton("80%", 'b8')
    lm3000.addButton("100%", 'b9')
    lm3000.addParameter("Light Level", "s")
    lm3000.setYLabel("Light Level")
    lm3000.addPlot()
    lm3000.begin()
    self.gui.addDevice(lm3000)

    self.nodeTree = MNodeTree.NodeTree()

    lightMeterNode = MDeviceNode.MDeviceNode(lm3000)
    self.nodeTree.addNode(lightMeterNode)
    rawLightOutput = lightMeterNode.getAnchorByName("Light Level")
    avgLight = lightMeterNode.addAnchor(name = "Average Light Level", type = "input", terminate = True)

    avg = runningAverage.runningAverage()
    avg.setWindowWidth(100)
    avgInput = avg.getAnchorByName("data")
    avgOutput = avg.getAnchorByName("running avg")

    self.nodeTree.connect(rawLightOutput, avgInput)
    self.nodeTree.connect(avgOutput, avgLight)
    # self.nodeTree.connect(output, virtAvgInput)

    # Create the gui.

    self.gui.startGui('Leiden DR GUI', tele)


# In Python, the main class's __init__() IS NOT automatically called.
viewer = mViewer()
viewer.__init__()
```

The Interface

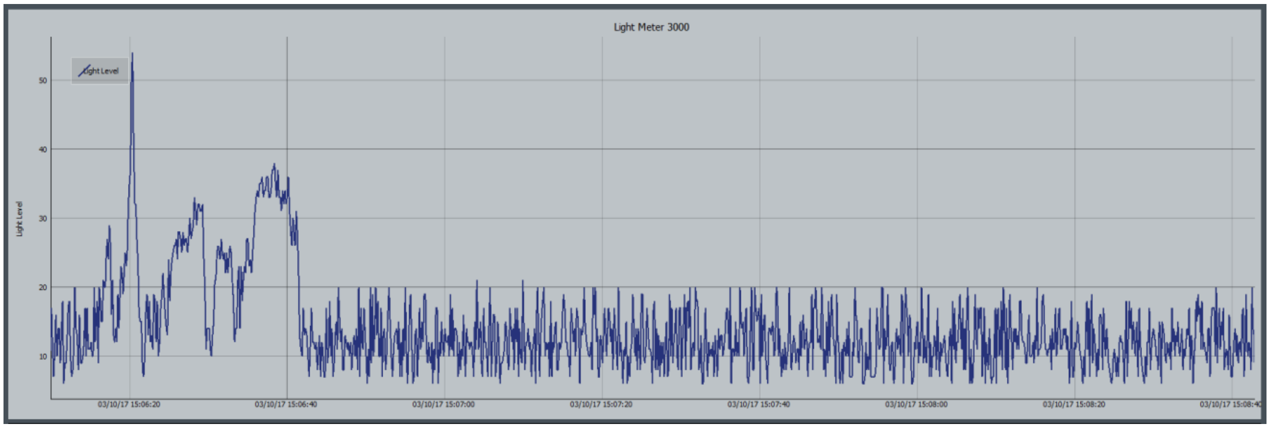
Here is what we have created!





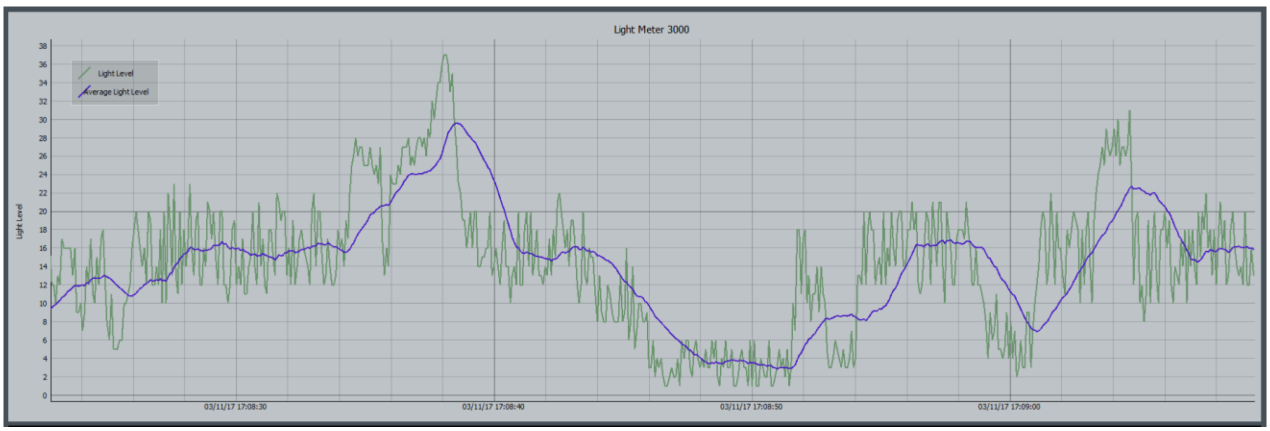
A Segway to the Cool Stuff

Lets take a closer look at the data we get from out light meter.



Notice the amount of noise in the signal. Up until this point in the guide, this noise would either have had to be ignored or dealt with manually outside of MView. There is; however, a better way and it is provided by something called the node tree.

In the following section, we will look at how to obtain the following plot automatically.



The Node Tree

What is the node tree?

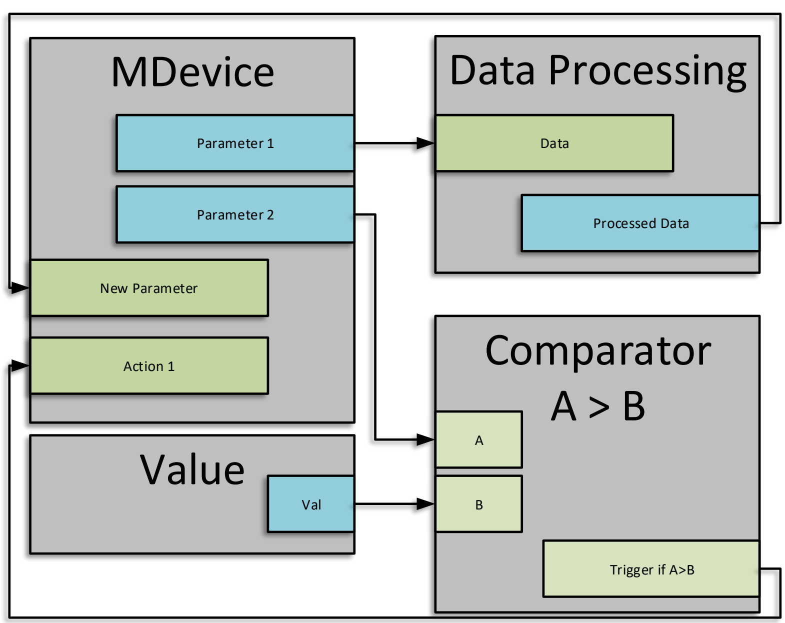
MView's node tree is without a doubt MView's most useful and coolest feature. It takes MView from being just a dashboard to a tool that can perform application-specific tasks.

MView accomplishes this through a nodal logic editor, based on a programming paradigm called **flow-based programming (FPB)**. A quick Google search should provide a much more in depth explanation of the idea, but here is the tl;dr:

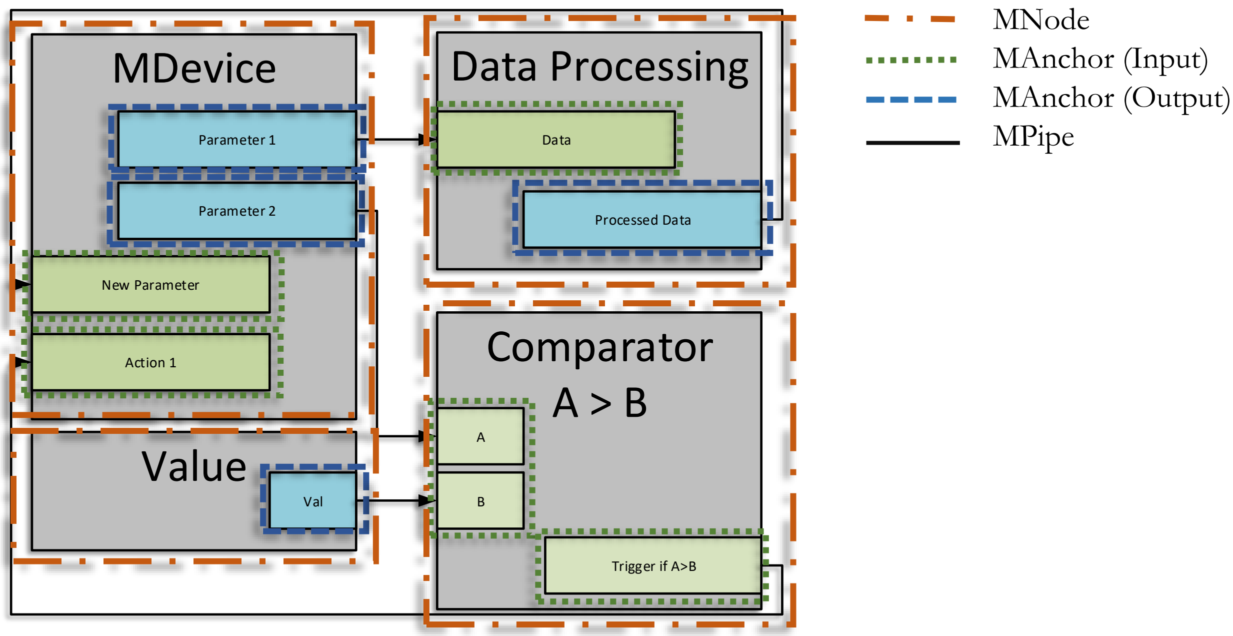
Elements in MView such as plots, devices, etc. can be represented as black boxes which are related through a network of connections. These connections communicate data between black boxes, allowing elements of MView to communicate.

Possible applications of MView's FBP include but are not limited to, data processing, feedback loops, device control. The FBP allows equipment to be controlled and monitored in much the same way as LabView, the difference being reduced complexity along with the familiarity of python.

This is enough information for now, the idea is simple and as we use it more and more, and you will get a better understanding of what it can be used for. A description of how MView's FPB engine was written will be at the end of the document.

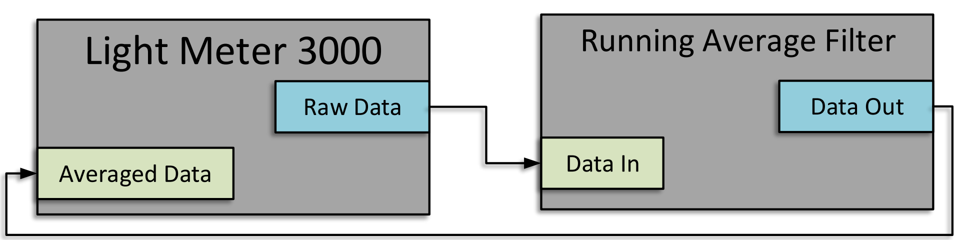


Elements of MView's Node Tree



Getting Rid of the Light Meter's Noise

We will now use the node editor to put a running average of the light meter readings on the plot. We will do this using the following map:



Note: It is always good to draw out the node tree maps before trying to implement them.