PSFC/JA-20-52

Exploring MDSplus data-acquisition software and custom devices

Santoro, Fernando; Stillerman, Joshua; Lane Walsh, Stephen; Fredian, Thomas

January 2020

Plasma Science and Fusion Center
Massachusetts Institute of Technology
Cambridge MA 02139 USA

This work was funded under DOE cooperative agreement DE-SC0012470. Reproduction, translation, publication, use and disposal, in whole or in part, by or for the United States government is permitted.

Submitted to *Fusion Engineering and Design*

Exploring MDSplus data-acquisition software and custom devices

Fernando Santoro, Joshua Stillerman, Stephen Lane-Walsh, Thomas Fredian

MIT, Plasma Science and Fusion Center

Abstract

MDSplus is a software tool designed for data acquisition, storage, and analysis of complex scientific experiments. Over the years, MDSplus has primarily been used for data management for fusion experiments. This paper demonstrates that MDSplus can be used for a much wider variety of systems and experiments. We present a step-by-step tutorial describing how to create a simple experiment, manage the data, and analyze it using MDSplus and Python. To this end, a custom example device was developed to be used as the data source. This device was built on an open-source electronic hardware platform, and it consists of a microcontroller and two sensors. We read data from these sensors, store it in MDSplus, and use JupyterLab to visualize and process it.

This project and code demo are available on the GitHub site at this URL:
https://github.com/santorofer/MDSplusAndCustomeDevices

Keywords: Data acquisition, Software, Visualization, MDSplus, Custom devices, Python

1. Introduction

MDSplus has been used as one of the principal software tools to acquire and organize the vast amount of data generated by magnetic fusion energy experiments.

In order to demonstrate how MDSplus works, and specifically its devices, all that we need is a simple experiment. Our experiment consists of a microcontroller connected to two sensors. The microcontroller communicates to MDSplus through a serial port, to a Python Device class.

An MDSplus Device describes a set of nodes and methods. These nodes contain the hardware configuration, as well as the data collected from the device. These are usually organized to closely resemble the sensors on the hardware. For example, channel A might become INPUT_A. These methods operate on the hardware configuration information and are used to communicate with the device.

The standard set of methods for modern devices are INIT(), STOP(), and TREND(). INIT() is used to start streaming data collection. STOP() is used to stop streaming data collection. TREND() takes a single data point and returns, and is meant to be called repeatedly by an external process.

In order to view the data you have collected in MDSplus, you can use another tool in the MDSplus toolbox called jScope.

MDSplus events allow the various MDSplus tools to communicate while data is being collected. These events are implemented as UDP broadcast network messages, and can be produced or consumed from any MDSplus tool. During the data acquisition process, we send an event whenever new data is received. When using jScope, you can configure it to listen for an MDSplus event and update the display accordingly.

Section 2 describes in more detail the hardware used in this experiment. The MDSplus implementation is described in Section 3. In it, the MDSplus Device class, the structure of the tree, and the methods are shown. In Section 4, we explain the configuration of the system and the execution of our experiment. The steps to visualize the data are listed in Section 5. Finally, in Section 6, the process for analyzing data is described.

2. The hardware device

2.1. The microcontroller and its sensors

The hardware used in this experiment consists of an Arduino microcontroller (see Fig. 1) and two sensors.

Email addresses: fsantoro@psfc.mit.edu (Fernando Santoro), jas@psfc.mit.edu (Joshua Stillerman), slwalsh@psfc.mit.edu (Stephen Lane-Walsh), twf@psfc.mit.edu (Thomas Fredian)

The first is a LIDAR sensor, and the second is a temperature/humidity sensor (see Fig. 2). To read more about them see [1] and [2]. Microcontrolers make it very easy to set up a simple experiment such as this. This simplicity allows us to focus on creating a tool that can demonstrate the capabilities of MDSplus, and specifically the creation of an MDSplus Device.

Using the Arduino's USB serial port and Python's PySerial library, we query the sensors and the responses are stored in the MDSplus tree.

2.1.1. Arduino UNO board

We are using the open source Arduino UNO board for this demonstration. This board uses the ATmega328P microchip. Arduino UNO contains several digital input/output pins, analog inputs, and a USB connection that allows for a direct communication to a computer via an USB cable. Fig. 1 shows the board.

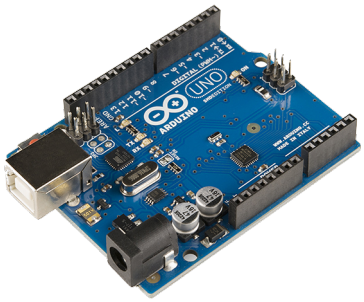

Figure 1: Arduino UNO microcontroller board. (This image is licensed under the Creative Commons Attribution 2.0 Generic license. The license can be found in https://creativecommons.org/licenses/by/2.0/. Author: SparkFun Electronics from Boulder, USA. This image can be found in Flickr.com and Wikimedia Commons).

The board can be programmed using the Arduino IDE, see following subsection. To learn more about the board, its capabilities and its specifications, refer to the Arduino's documentation [3].

2.1.2. The Arduino Sketch

Before the Arduino board can communicate with the sensors, firmware needs to be written and uploaded into the board. A sketch, using Arduino terminology [4], is the code that is uploaded into the microcontroller (*Appendix A* shows the sketch code written for this experiment.)

The sketch code can be divided into three parts: the setup() function (line #30 *Appendix A*), the getSerialCommand() function (starting on line #49) and the loop() function (starting on line #71). The setup() function initializes the two sensors by calling the begin methods of the LIDAR and the temperature/humidity libraries. The getSerialCommand() function reads input from the serial port. The loop() function queries the sensors every two milliseconds, and responds to commands from the getSerialCommand() function.

2.1.3. The sensors

Garmin LIDAR-Lite v3HP is a ranging and proximity sensor. This uses an optical distant measurement sensor with the following characteristics (among others): when sample the data the sensor updates with a frequency greater than 1 kHz who range is 5 cm to 40 m with a 905 nm wave-length and 1.3 W peak power laser. More detailed explanation of this and other specifications are found in the Gamin's manual [1]. See Figure 2 for an image of the LIDAR sensor.

Figure 2: Garmin Lidar Lite r3HP on the left, DHT11 digital temperature and humidity sensor on the right.

The DHT11 is a cheap digital temperature and humidity sensor. It has an update rate of 1 Hz. Refer to its documentation for a more exhaustive description of its technical characteristics [2]. See Figure 2 for an image of these sensors.

3. The structure of an MDSplus Device

The hardware described in the previous section is represented in the data system as an MDSplus Device. The MDSplus Device is a framework that allows the user to integrate the hardware into the software [5]. The goal of writing this Device class, is that any duplicate hardware can use the same code.

Configuration for a given device will be stored in a structure of nodes in the tree. A graphical MDSplus tool

called jTraverser allows for the exploration and editing of this structure. It is shown in Figure 3.

The code that represents the device, written in Python, can be divided in five parts:

- The parts array. This is the definition of the structure of nodes that make up this device. See Section 3.1 for a more complete description, see also *Appendix B*, where the MDSplus parts array is clearly defined (starting in line #11).
- The serial port communication methods. These are internal methods of the Device class used to communicate with the sensors. Section 3.1.1 describes them in more detailed.
- The INIT() and STOP() methods. These are two of the standard methods used by MDSplus to run the data acquisition system. In this case, the INIT() method will start the data stream, while the STOP() method will terminate the acquisition process. See *Appendix D*.
- The TREND() method. This method is executed by MDSplus when collecting data using scheduled jobs. See Section 3.1.2 for a detail explanation of this method, as well as the definition and usage of scheduled jobs and Section 4.1 for the usage in the context of running the data system. The source code can be read in *Appendix E*.
- The STREAM() method. This method is called by the INIT() method. When run, the method communicates with the device through the class query methods, receiving and storing the data that is being streamed. See Section 3.1.3 for a detail explanation and Section 4.2 for its usage. The source code can be read in *Appendix F*.

3.1. Implementing the MDSplus tree

The class that implements our Arduino-based Device defines the structure of nodes in a variable called the parts array. These nodes are where the information associated with the experiment is stored. When the device is added to the tree, the nodes described in the parts array are added to the tree as well (see *Appendix B*).

What follow is a short description of each of the nodes of the tree that represent our experiment:

- COMMENT: A string that can contain comments about the experiment.
- BAUD : A number that contains information about the baud rate, with a default value of 9600.

Figure 3: The structure of the MDSplus tree shown by the jTraverser

- TRIG_TIME: A number that represent the time in which the device starts streaming data.
- SEG_LENGTH: A number that represent the length of each segment that also corresponds to the number of samples per segment.
- MAX_SEGMENTS: A number representing the total number of segments to be stored.
- TREND_EVENT: The name of the MDSplus event used to update scopes or notify other applications that data has been added.
- STREAM_EVENT: A string that represent the event name, and similar to the TREND event. Data acquisition can then be displayed as it happens.
- RUNNING: A boolean that keeps the information of the running status of the device.
- DISTANCE: A signal node that stores the data of the LIDAR sensor as a MDSplus signal data type, which is useful to represent the time evolution of a given quantity.
- TEMPERATURE: A signal node that stores the data of the temperature sensor as a MDSplus signal data type.
- HUMIDITY: A signal node that stores the data of the humidity sensor as a MDSplus signal data type.

Once MDSplus parts array structure has been defined, the super-class has all the required information to provide the specific constructor. Fig 3 shows the results of building the MDSplus tree for the Arduino device for this example.

The following sub-sections give a short descriptions of the methods that form part of our Device class.

3.1.1. Open and query the sensors

These set of methods use the Python library called PySerial. They consist of four methods for opening and initializing the serial port to communicate with the Arduino device, query the device by sending the query question or command, and finally receiving data from it. *Appendix C* shows the four methods to control the device through the serial port.

3.1.2. The TREND() Method

The TREND() method is used to acquire data from the sensors in a continues automatic way. The method simply opens the serial socket (line #3), query the device for one data point (lines #12, 15 and 18), saves the answers into the corresponding tree nodes (making use of the MDSplus putRow() method) and finally closes the socket (lines #23, 24 and 25). This is repeated every a determined amount of time, that for this experiment is define in Section 4.1. The full code can be found in *Appendix E*.

To run this method automatically, a scheduled job script needs to be written. Section 4.1 shows how to write this script and how to tell the operating system to execute it.

3.1.3. The STREAM() Method

The STREAM() method is an internal method. It polls the sensors and writes the data to the tree. Continuous data acquisition is achieved in MDSplus via the segment concept. A segment is a record that has a start and end point time, and the associated the data. MDSplus segmented records has the ability to append new segments and to retrieve a subset of the data when necessary. To summarize, each segment is identified by the following (see lines # 31 to 33 in refstreammethod):

1. start, i.e. the starting time of the segment
2. end, i.e. the ending time of the segment
3. dimension, i.e. this is the time associated with each sample in the segment.
4. data, i.e. an array that contains the data.

See Section "Using MDSplus Segments for Continuous Data Acquisition" in [7] for detail explanation of using segments in MDSplus.

The full code that implements the STREAM() method can be found in *Appendix F*. As can be seen in the while loop (line #20 of the code), the device is sampled every certain amount of time, defined by the Arduino sketch (see line #88 of the sketch in *Appendix A* showing a 2 milliseconds delay before the next time the serial port is queried), until the segment length is reached, and repeated until we reach the maximum number of segments. In other words, the Arduino device will start acquiring data cyclically until the preset number of samples is recorded or the STOP() method is executed. In this example the data acquisition then happens at 0.5 KHz.

4. Data acquisition

The first step to data acquisition is to set up the system environment. MDSplus and Python need to know where the tree files are and where the python code representing the device is. This two variables need to be defined:

A- the path to the MDSplus tree

B- the path to the MDSplus python device

For this example, the environment is set using the following two lines:

```
import os
os.environ['arduino_path'] =
 ↪  '/ArduinoEx/'
os.environ['MDS_PYDEVICE_PATH'] =
 ↪  '/ArduinoEx/devices/'
```

Listing 1: Setting up the system environment

As an example, we used "/ArduinoEx" as the path to the tree files, and '/ArduinoEx/devices' as the path to the code.

The second step is to add the device to the model that was designed in Section 3.1.3. The following example code imports the Arduino class that was built based on the MDSplus Device class together with MDSplus Tree function that opens the tree for writing in a new model. The Arduino device is then added to the specified device model using the addDevice() method of the tree object. This function takes the name of the node as an input, where the device will be added to the tree, ie. this is the head of the node of the device, and the name of the model tree (see Listing 2).

```
from MDSplus import Tree
import mdsarduino
tree = Tree('arduino', -1,  'new')
tree.addDevice('arduino_node',
 ↪  'arduino')
tree.write()
tree.close()
```

Listing 2: Adding the device to the model

The third step is to choose between trending and streaming. When data acquisition is synchronized with system time and initiated by operating system timers, we define this as trending. Data recorded in this way can be displayed with date and time as it X-Axis. When data acquisition is associated with an experiment with its own time-base, and done by a thread polling the device on timer, this is streaming. In this case data can be displayed on the experiment time-base.

4.1. Trends: data acquisition using scheduled jobs

Data acquisition can be scheduled to start and stop every certain amount of time using timers. Depending on the operating system, a task can be schedule by systemd, in Linux systems, or by launchd, in Mac OS X systems. The following sections explain in details how to set a task using those two types of scheduling frameworks.

4.1.1. Linux systemd

In Linux operating systems, systemd is the system and service manager (see [6])

The following scripts are needed to set up an experiment as a trend.

A- a timer and a service script, to be run by systemd. For every timer file, a service file exists, both with the same name but different extension (in our example below we used *arduino_trend.timer* and *arduino_trend.service*). These files are executed by systemctl.

B- a device specific script to be called by the service script. This is the script that will execute MDSplus commands that runs the device code.

The timer script (*arduino_trend.timer*) will run the experiment for a predetermined amount of time, which in this case is 5 seconds:

```
[Unit]
Description=Timer to run trend.service
[Timer]
OnUnitActiveSec=5s
AccuracySec=1us
[Install]
WantedBy=multi-user.target
```

The service script (*arduino_trend.service*) will execute the device script, which in this example is called "arduino_trend":

```
[Unit]
Description=Trigger the TREND()
 ↪  function of the Arduino device in
 ↪  the trend tree
[Service]
Type=simple
ExecStart=/ArduinoEx/arduino_trend
User=mdsplus
[Install]
WantedBy=multi-user.target
```

The device script (in our example is called "arduino_trend") will, in turn, executes MDSplus commands and run the experiment making use of MDSTCL (the MDSplus Tree Command Language interpreter):

```
#!/bin/bash
. /usr/local/mdsplus/setup.sh
export MDS_PYDEVICE_PATH=/ArduinoEx/
export arduino_path=/ArduinoEx/

mdstcl <<EOF
    set tree trend /shot=0
    do /meth arduino trend
    exit
EOF
```

The execution of the timer and the service is done by calling systemctl. This happens in two parts: enable and start.

```
sudo systemctl enable
 ↪  arduino_trend.timer
sudo systemctl enable
 ↪  arduino_trend.service

sudo systemctl start
 ↪  arduino_trend.timer
sudo systemctl start
 ↪  arduino_trend.service
```

4.1.2. Mac OS X launchd

In Mac OS X, launchd is the service management framework, or job scheduler, for starting, stopping and

managing process and scripts on OS X. In this framework, an MDSplus Device TREND() method will be executed from a shell script, which in turn launchd will execute every predetermined amount of time. For example, the TREND() method could be executed once every 5 seconds, data is retrieved once from each of the sensors, saved in the correct tree node and the execution of the TREND() method ends. After the preset amount of time, this cycle is repeated, adding new data values into the tree.

The recommended way to execute a system script to be launched with launchd is shown below. This script is similar to to the one used in Section 4.1.1 (using MDSplus's MDSTCL command).

```
#!/bin/bash
. /usr/local/mdsplus/setup.sh
export MDS_PYDEVICE_PATH=/ArduinoEx/
export arduino_path=/ArduinoEx/

mdstcl <<EOF
    set tree arduino /shot=0
    do /meth arduino_node trend
    exit
EOF
```

The above bash script needs to be executed by Mac OS X launchctl command. This is done by first writing a OS X plist, that in this example is called "com.mdsarduino.daemon.plist". To start the service, the following command needs to be executed:

```
launchctl load ~/Library/LaunchAgents/
 ↪  com.mdsarduino.daemon.plist
```

To learn more about Mac OS X plist development, see Apple's Working with Property List Files [8].

4.2. Data Acquisition by streaming data

The device's INIT() method starts the internal STREAM() method which, in turn polls the device for samples on a timer and records them in the tree.

We first create the model tree and then we can start streaming data, using the INIT() method. We can use Python or MDSTCL for this.

A second step is to prepare and launch one of the MDSplus scopes, for example jScope (see Section 5 for more details). From the jScope GUI, the event variable name needs to be defined, so that MDSplus events can be updated by the scope and in turn the acquired data can simultaneously be displayed (see Section 5 for details on how to set the event name using jScope configuration menus). An example of what jScope display looks like when it is used during a data acquisition run can be seen in Figure 4.

As a third step we set the current shot and create the pulse. The code is as follow:

```
from MDSplus import Tree
tree = Tree('arduino', -1)
tree.setCurrent(1)
tree.createPulse(0)
tree = Tree('arduino' , 0)
```

Listing 3: Setting the shot number

The fourth step is to execute the INIT() method defined in *Appendix D*. The Python call statement is as follows:

```
tree.ARDUINO_NODE.init()
```

Listing 4: Execution of the INIT() method

Finally, the data acquisition can be stopped using the STOP() method described in *Appendix D*. The Python call is as follows:

```
tree.ARDUINO_NODE.stop()
```

Listing 5: Execution of the STOP() method

The following list summarizes what is needed to run the data acquisition software developed in the previous few sections:

1. Set up the system environment, defining the treeName_path variable to point to the location of the tree model file. (see Listing 1)
2. Create an MDSplus Model Tree by adding the device to the model. (see Listing 2)
3. Launch the MDSplus jScope. The scope event name needs to be already defined in the scope configuration.
4. The following 3 steps are repeated for each shot:
   (a) Set the current shot and create the pulse. (see Listing 3)
   (b) Execute the INIT() method to start the data acquisition. (see Listing 4)
   (c) Execute the STOP() method to stop the data acquisition. (see Listing 5)

5. Visualizing data acquisition

The results of the experiment can be visualized in two ways:

- Dynamically: simultaneous visualization using jScope (Fig. 4)
- Post data collection: using jScope or Python libraries (see Section 6).

jScope is one of the tools MDSplus provides for visualization of the data, as it happens, during and after data acquisition. It's a tool (written in Java) that allows for the definition of graph panels, zoom in parts of the waveforms, etc. One of the features that MDSplus scopes have, is the possibility of defining a field under the Customize pulldown menu, in the Global Setting choice, called Update Event. This entry contains the name of the event that causes the automatic update of the scope.

An MDSplus event is defined in the Python code of the device using the MDSplus Event class and its setevent method (see Listing 6 below).

```
MDSplus.Event.setevent(event_name)
```

Listing 6: Issuing an event from within a Python code, using the MDSplus Event class and its setevent method.

The argument event_name is one of the elements of the Arduino device class parts array (see *Appendix B*, lines #23 and 25). Also, see the TREND() Method or the STREAM() Method in *Appendix E* and *Appendix F* where the usage of the above statement can be found (see line #27 for the TREND() method and line #39 for the STREAM() method)

We can see in Figure 4 the jScope tool used during data acquisition. The three panels show the data at the same time it is being collected. Top panel represent the distance to the LIDAR sensor in cm, while the middle panel shows the ambient temperature measurements in Celsius and the lower panel shows the ambient humidify, all of them as a function of time in sec.

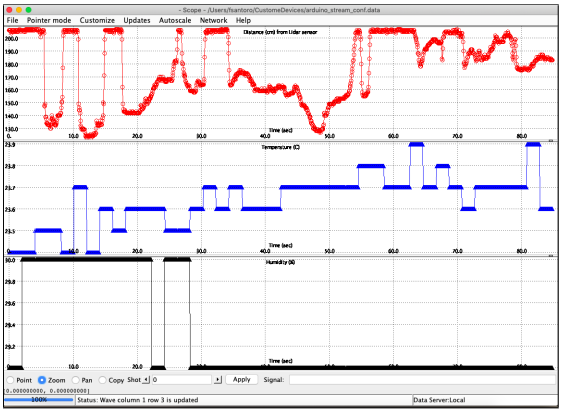

Figure 4: MDSplus jScope showing data acquisition as it happens.

6. Data Analysis

6.1. Reading the data set from an MDSplus tree

Jupyter Lab notebook allows for data analysis with the ability of performing interactive data visualization. A simple workflow to read the acquired data from the MDSplus trees into a Python variable and later on visualize the results can be summarized as follows.

Before starting, Python libraries like matplotlib, will need to be installed. In addition, interactive plots are configured when the following line is added to the Python code:

```
%matplotlib widget
```

The above line enables the jupyter-matplotlib backend, i.e. an interactive widget as a Jupyter magic.

The code showed below (Listing 7) is an example function that reads data sets from an MDSplus tree. In it, the tree name, shot number and the name of the node are arguments that are passed to the function. Once the tree is open, the node handle can be defined so that data can be retrieved. The data is then returned by the function as a Python array.

```
from MDSplus import *
%matplotlib widget
import matplotlib.pyplot as plt

def getData(treeName, shot, node):
    shotNumber= int(shot)

    tree = Tree(treeName, shotNumber)
    node = tree.getNode(node)

    nodedata = node.getData()
    data     = nodedata.data()
    return data
```

Listing 7: Opening of the MDSplus tree for reading

An example of a Python function that uses the openTree function above (Listing 7) can be seen in *Appendix G*, where the function is called arduino. This code is presented here as an example of doing data analysis with Jupyter Lab and data acquisition with MDSplus.

The code is divided in three parts. The first part deals with reading of the data directly from the MDSplus data set (see lines #40, 58 and 59). This is done by calling the arduino function with the parameters of the experiment, i.e. the shot number, tree and node names.

Secondly, data from the MDSplus tree that is returned as a float array is used to build a spline function. The SciPy library scipy.interpolate.UnivariateSpline was used, and the smoothing of the curve was done by set_smoothing_factor method (see line #50). And thirdly, the data can then be plotted using the Python plot function. Figure 5 shows distance (cm) vs number of data points taken from LIDAR sensor data set saved in the MDSplus tree. The dots are the raw data points while the orange curve represent the approximation using splines. In Jupyter Lab the plot window is interactive, supporting zoom and some graph manipulation.

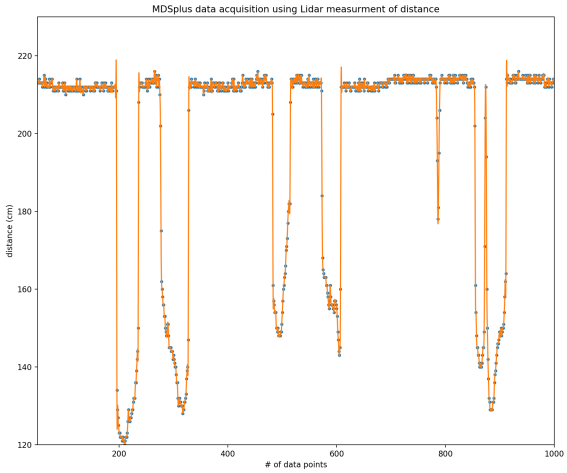

Figure 5: JupyterLab output plot. It shows the raw data collected (dots) during the data acquisition for the LIDAR sensor. The spline interpolation is shown here as orange line. Distance (cm) versus number of data points

7. Summary and Conclusions

Data acquisition and automatic analysis can be done by designing an MDSplus Device using Python. This Device is in reality a Python abstraction of the hardware itself. Once the MDSplus Device is constructed, data collection can proceed, and it can be collected into MDSplus trees automatically.

There is perception that MDSplus has a steep learning curve and is considered difficult to deploy. This paper, on the other hand, shows that this perception is far from the truth. We are able to demonstrate this by describing first the hardware used for the experiment in a concise and simple manner. Followed by an explanation of how that hardware can be translated into an MDSplus Device abstraction. This abstraction was then dissected in full, showing the different functions that the Python device code contains, with an thorough explanation of each of them.

Following the creation of the code, we presented a description of the different options for executing a data acquisition system. Starting with scheduled jobs and the TREND() method, and ending with continues data collection using the STREAM() method.

Finally, we showed two ways to produce data visualization during and after the acquisition phase: using MDSplus jScopes together with MDSplus Event class for simultaneous visualization, and using Jupyter Lab interactive visualization capabilities during an analysis phase.

JupyterLab provides a user-friendly tool to develop and both document custom devices in MDSplus and display, analyze and document results. However it is not required, it is simply a way to operate and interact with Python. The code could be developed using any programming tool-set. Data display can be done in any MDSplus supported programming language, or using the interactive tools (jScope, dwscope, dwscope_remote) provided with the MDSplus package.

Overall, the ability of Jupyter Lab to break down the different sections of the code and allow detail descriptions for every step of data acquisition process, makes it easy to demonstrate the capabilities of MDSplus Devices. Another possibility is to execute the data collection from outside Jupyter Lab and only come back to it once data analysis functions are developed, taking advantage of plot interactivity.

To summarize, we have created a Python abstraction of the hardware device, i.e. the MDSplus Device, and showed to be self-contained and simple enough that can be use as a template for the understanding of MDSplus Devices in more complex experiments and systems in general.

8. Acknowledgements

This work was funded under DOE cooperative agreement DE-SC0012470.

Appendix A. Arduino sketch designed specifically for this experiment

```
/**
* Author: Fernando Santoro and Stephen Lane-Walsh
* Date: May 01, 2019
* Based on code by: Garmin and Shawn Hymel (SparkFun Electronics)
* Date: June 29, 2017
* Read distance from LIDAR-Lite v3 over I2C
*/

#include <Adafruit_Sensor.h>
#include <DHT.h>
#include <DHT_U.h>
#include <Wire.h>
#include <LIDARLite.h>

// Globals
LIDARLite lidarLite;
int cal_cnt = 0;

DHT_Unified dht(5, DHT11);

uint32_t delayMS;

String cmd;

void setup()
{
Serial.setTimeout(50);
Serial.begin(9600);

// Lidar setup
// Set configuration to default
lidarLite.begin(0, true);
lidarLite.configure(0); // Change this number to try out alternate configurations

// Temperature Humidity sensor setup
dht.begin();
sensor_t sensor;
dht.humidity().getSensor(&sensor);
delayMS = sensor.min_delay / 1000;
}


bool getSerialCommand() {
static int i = 0;
static char line[32];
while (Serial.available()) {
char c = Serial.read();
line[i++] = c;
if (c == '\n') {
  line[i - 1] = '\0';
```

```
    cmd = String(line);
    i = 0;
    return true;
  }
  else if (i == sizeof(line) - 1) {
    line[i] = '\0';
    cmd = String(line);
    i = 0;
    return true;
  }
  }
  return false;
  }

  void loop()
  {
  int dist;
  sensors_event_t event;

  // At the beginning of every 100 readings,
  // take a measurement with receiver bias correction
  if ( cal_cnt == 0 ) {
  dist = lidarLite.distance();      // With bias correction
  } else {
  dist = lidarLite.distance(false); // Without bias correction
  }

  // Increment reading counter
  cal_cnt++;
  cal_cnt = cal_cnt % 100;

  delay(2); // in millisecs

  if (getSerialCommand()) {
  if (cmd == "DIST") {
  // Display distance
  Serial.println(dist);
  }
  else if (cmd == "READY?") {
  Serial.println("I'M READY");
  }
  else if (cmd == "DELAY") {
  Serial.println(delayMS);
  }
  else if (cmd == "TEMP") {
  dht.temperature().getEvent(&event);
  Serial.println(event.temperature);
  }
  else if (cmd == "HUMID") {
  dht.humidity().getEvent(&event);
  Serial.println(event.relative_humidity);
  }
```

```
}
}
```

Appendix B. MDSplus Device class parts array

```
from MDSplus import Event
import serial
import time
import datetime
import threading
import numpy as np
import jdc
import pdb

class ARDUINO(MDSplus.Device):
    parts=[
        {'path':':COMMENT','type':'text',
            'options':('no_write_shot',)},
        {'path':':NODE','type':'text','value':'',
            'options':('no_write_shot')},
        {'path':':BAUD','type':'numeric', 'value': 9600,
            'options':('no_write_shot',)},
        {'path':':SEG_LENGTH','type':'numeric', 'value': 5,
            'options':('no_write_shot',)},
        {'path':':MAX_SEGMENTS','type':'numeric', 'value': 1000,
            'options':('no_write_shot',)},
        {'path':':TREND_EVENT','type':'text', 'value':'ARDUINO_TREND',
             'options':('no_write_shot',)},
        {'path':':STREAM_EVENT','type':'text','value':'ARDUINO_STREAM',
            'options':('no_write_shot')},
        {'path':':RUNNING','type':'any', 'options':('no_write_model',)},
    ]

    parts.append({'path':':DISTANCE','type':'signal',
        'options':('no_write_model','write_once',)})
    parts.append({'path':':TEMPERATURE','type':'signal',
        'options':('no_write_model','write_once',)})
    parts.append({'path':':HUMIDITY','type':'signal',
        'options':('no_write_model','write_once',)})
```

Appendix C. Sensor queries through serial port

Appendix C.1. Initializing Arduino serial port

```
def openSerial(self):
# Initializing Arduino serial port
print('Openning serial port')
try:
    self.socket = serial.Serial()
    self.socket.port     = '/dev/cu.usbmodem14101'
    self.socket.baudrate = 9600
    self.socket.parity = serial.PARITY_NONE
```

```
    self.socket.bytesize = serial.EIGHTBITS
    self.socket.stopbits = serial.STOPBITS_ONE
    self.socket.timeout  = 1
    self.socket.xonxoff  = False
    self.socket.rtscts   = False
    self.socket.dsrdtr   = False
    self.socket.open()
    # Wait for handshake and skip garbage data
    self.socket.readline()
    self.socket.write("\n")
    self.socket.flush()
    self.socket.readline()
except serial.SerialException:
    print('Port ' + self.socket.port + ' not available.')
    return
return
```

Appendix C.2. Asking questions to the sensor.

```
def queryCommand(self, cmd):
    self.sendCommand(cmd)
    return self.recvResponse()
```

Appendix C.3. Sending queries to the sensor.

```
def sendCommand(self,cmd):
    self.socket.write(cmd + "\n")
    self.socket.flush() # Flush and wait until all data is written.
```

Appendix C.4. Receiving answers from the sensor.

```
def recvResponse(self):
    msg = ""
    try:
        msg = self.socket.readline()
        return msg
    except serial.SerialException:
        print('Problem reading socket')
        return None
```

Appendix D. The INIT() and STOP() methods

```
def init(self):
    self.running.on=True
    self.stream()
INIT=init

def stop(self):
    print('Stopping now')
    self.running.on = False
STOP=stop
```

Appendix E. The TREND() method

```
def trend(self):
    print("Starting trend")
    self.openSerial()
    sampleTime=time.time()

    event_name = self.trend_event.data()

    chan_1 = self.__getattr__('DISTANCE')
    chan_2 = self.__getattr__('TEMPERATURE')
    chan_3 = self.__getattr__('HUMIDITY')

    distance    = MDSplus.Float32(float(self.queryCommand('DIST')))
    print(distance)

    temperature = MDSplus.Float32(float(self.queryCommand('TEMP')))
    print(temperature)

    humidity    = MDSplus.Float32(float(self.queryCommand('HUMID')))
    print(humidity)

    MDSsampleTime = MDSplus.Int64(sampleTime*1000.)

    chan_1.putRow(1000, distance   , MDSsampleTime)
    chan_2.putRow(1000, temperature, MDSsampleTime)
    chan_3.putRow(1000, humidity   , MDSsampleTime)

    MDSplus.Event.setevent(event_name)
    self.socket.close()
TREND=trend
```

Appendix F. The STREAM() Method

```
def stream(self):
    print("Starting streamer")
    self.openSerial()
    segment = 0
    start_time = time.time()

    seg_length = int(self.seg_length.data())
    event_name = self.stream_event.data()
    print(event_name)

    dist  = np.zeros(seg_length)
    temp  = np.zeros(seg_length)
    humi  = np.zeros(seg_length)
    times = np.zeros(seg_length)

    chan_1 = self.__getattr__('DISTANCE')
    chan_2 = self.__getattr__('TEMPERATURE')
    chan_3 = self.__getattr__('HUMIDITY')
```

```
19
20      while self.running.on and segment < self.max_segments.data():
21
22          for sample in range(seg_length):
23              previous_time = time.time()
24              times[sample] = previous_time - start_time
25              distStr = self.queryCommand('DIST')
26              if 'nack' not in distStr:
27                  dist[sample] = MDSplus.Float32(float(distStr))
28              temp[sample] = MDSplus.Float32(float(self.queryCommand('TEMP')))
29              humi[sample] = MDSplus.Float32(float(self.queryCommand('HUMID')))
30
31          chan_1.makeSegment(times[0], times[-1], times, dist)
32          chan_2.makeSegment(times[0], times[-1], times, temp)
33          chan_3.makeSegment(times[0], times[-1], times, humi)
34          segment +=1
35
36          if segment%100 == 0:
37              print('Segment #' + str(segment) + ': Added to tree nodes.')
38
39          Event.setevent(event_name)
40
41      self.socket.close()
42      print("%s\tAll Sampling Done"%(datetime.datetime.now(),))
43
44      print("%s\tDevice added"%(datetime.datetime.now(),))
```

Appendix G. Data analisys using Python, MDSplus and JupyerLab

```
1  import numpy as np
2  import os
3  import sys
4
5  from scipy.interpolate import UnivariateSpline
6
7  %matplotlib widget
8  import matplotlib.pyplot as plt
9  import matplotlib
10
11 import math
12
13 import array
14 from MDSplus import *
15
16 def arduino(treeName, shot, node):
17
18     shotNumber= int(shot)
19     treeName  = treeName
20     inputNode = node
21
22     print('Opening tree ' + treeName + ' in shot #' + str(shotNumber) + ' and input Node
       ↪  ' + inputNode)
```

```
23      tree = Tree(treeName, shotNumber)
24      node = tree.getNode(inputNode)
25
26      print('Reading Data...')
27      nodedata = node.getData()
28      data     = nodedata.data()
29      print('Finished reading Data...')
30
31      return data
32
33
34  def main():
35
36      shotNumber = 1
37
38      # Lidar: DISTANCE
39      ydata   = arduino('arduino', shotNumber, "arduino_node:distance")
40      xdata   = np.linspace(0., ydata.size-1, num=ydata.size, endpoint=True)
41
42      print('Building Spline')
43      fspl = UnivariateSpline(xdata, ydata)
44
45      plt.interactive(False)
46      xnew = np.linspace(0., ydata.size-1, num=6*ydata.size, endpoint=True)
47
48      print('Smoothing...')
49      fspl.set_smoothing_factor(.5)
50
51      fig=plt.figure(figsize=(12, 10), dpi= 80, facecolor='w', edgecolor='k')
52      plt.plot(xdata, ydata, '.', xnew, fspl(xnew), '-')
53      plt.show()
54
55      # Temperature and Humidity sensor
56      ydata_t   = arduino('arduino', shotNumber, "arduino_node:temperature")
57      ydata_h   = arduino('arduino', shotNumber, "arduino_node:humidity")
58      xdata   = np.linspace(0., ydata.size-1, num=ydata.size, endpoint=True)
59
60      print('Building Spline')
61      fspl_t = UnivariateSpline(xdata, ydata_t)
62      fspl_h = UnivariateSpline(xdata, ydata_h)
63
64      plt.interactive(False)
65      xnew = np.linspace(0., ydata_t.size-1, num=6*ydata_t.size, endpoint=True)
66
67      print('Smoothing...')
68      fspl_t.set_smoothing_factor(.5)
69      fspl_h.set_smoothing_factor(.5)
70
71      fig=plt.figure(figsize=(12, 10), dpi= 80, facecolor='w', edgecolor='k')
72      plt.plot(xdata, ydata_t, '.', xnew, fspl_t(xnew), '-')
73      plt.show()
74
```

```
75      fig=plt.figure(figsize=(12, 10), dpi= 80, facecolor='w', edgecolor='k')
76      plt.plot(xdata, ydata_h, '.', xnew, fspl_h(xnew), '-')
77      plt.show()
78
79      return 1
80
81  if __name__ == '__main__':
82      main()
```

List of source codes

References

[1] Garmin LIDAR-Lite v3HP Operation Manual and Technical Specifications, 2018
[2] Adafruits basic temperature-humidity sensor, 2019, https://www.adafruit.com/product/386
[3] Arduino's Website Model UNO, 2019, https://store.arduino.cc/usa/arduino-uno-rev3
[4] Arduino's Website, Sketches, 2019, https://www.arduino.cc/en/tutorial/sketch
[5] Developing MDSplus Devices, MDSplus tutorial, 2019, http://www.mdsplus.org/index.php/Documentation:Tutorial:Devices
[6] Ubuntu Manual, 2019, http://manpages.ubuntu.com
[7] MDSplus manual, MDSplus tutorial, 2019, http://www.mdsplus.org/index.php/Introduction
[8] Working with Property List Files, 2019, https://developer.apple.com
[9] Espressif's website Model ESP8266EX, 2019, https://www.espressif.com/en/products/hardware/esp8266ex/overview
[10] ESP8266 Wikipedia Website, 2019, https://en.wikipedia.org/wiki/ESP8266