A more flexible design for MDSplus Device drivers

Fernando Santoro[a], Stephen Lane-Walsh[a], Joshua Stillerman[a], Mark Winkel[a]

[a]*MIT Plasma Science and Fusion Center, 77 Massachusetts Avenue, NW17, Cambridge, MA 02139*

The traditional approach to building MDSplus Device drivers is rigid and lacks the ability to meet changing needs. We introduce a novel paradigm for Device driver development that allows the tree structure to dynamically change. This allows device drivers that can reconfigure to automatically reflect the hardware it represents, or a device that implements a variable number of queries to an external database. We have created a driver using this paradigm that communicates with a digitizer, queries the modules attached, and builds a MDSplus tree structure to utilize them. Additionally, this driver can reconfigure to match changes in the digitizer, by adding or deleting nodes using overwrite and/or delete modes. We also wrote a method for verifying both the setting provided and that the hardware matches the last known state. We have added fields to help validate settings such as min/max limits, and a list of allowed values. The definitions of the nodes which make of the device have been augmented to include help, tool tips and validation ranges. This will facilitate automated user interface generation.

Keywords: Data Acquisition, Digitizers, Hardware, MDSplus, MDSplus Devices, Software.

1. Introduction

MDSplus Devices [1, 2], which enable the use of data acquisition systems through representations of hardware, are historically very rigid. Originally developed in C and TDI [3], and now developed in Python, MDSplus Devices allow you to interact with the tree and hardware simultaneously. With the flexibility offered by Python and MDSplus, we introduce a new way to design these Devices.

2. Data Acquisition and Devices

One of the most critical parts of any scientific experiment is data acquisition. This involves communicating with hardware using many different APIs and storing the resulting data. Due to the wide range of hardware used, various codes and abstractions are often developed to communicate with and represent these devices. These abstractions are useful because a single device code can be developed to support multiple similar devices.

Devices often represent specific models of physical devices, like digitizers, but could also represent more abstract systems. For example, they could represent databases or services, anything with an API.

Some experiments make use of EPICS [4] (Experimental Physics and Industrial Control System) as the framework for data acquisition. EPICS is actively used in projects like ITER Fusion device [5], UKAEA (the UK Atomic Energy Authority) [6], KSTAR (Korean Superconducting Tokamak Advanced Research) [7] and many other non-fusion energy research. It has the concept of StreamDevice which provides support for controlling or reading data from serial or network attached devices. Although ITER adopted EPICS as control and data acquisition system, some diagnostics like SPIDER [8] also uses MDSplus for acquiring data.

2.1 MDSplus Devices

Since MDSplus was primarily intended for use with data acquisition systems it provides its own framework for interfacing with devices for data acquisition. This involves creating a group of nodes in the MDSplus tree that represents a given device. These nodes contain the configuration, data and metadata for the device. Depending on the amount of data and the rate it was acquired, the device can even be used to buffer data before writing it to the tree. The Device utilizes whatever protocols, packages, or drivers are required for interfacing with a given piece of hardware. These Device codes then serve as a uniform API for accessing and controlling data acquisition hardware during an MDSplus shot cycle.

When you add a device to a tree it will generate the nodes automatically. Then necessary information must be entered to communicate with the device. For example, a networked device would require you to fill in the IP address.

MDSplus Devices allow you to call methods on a device which means calling a function in some language to do something with the associated set of nodes. For example, some common methods are: init(), arm(), store(), abort(), etc. Additionally, you can configure methods to be automatically called at certain points during the experiments using actions.

3. Past and Present Design

Historically, each Device was designed as a rigid representation of a specific device model, such as ACQ216, ACA800 [9], or J221. Devices and manuals would be brought to the MDSplus developers, and codes would be written in the language of the era, and then used for experiments. The following subsections describe the different languages that has been used to create those Devices in more detail.

author's email: fsantoro@mit.edu

3.1 C and TDI Devices

MDSplus Devices were originally written in C, as shared libraries with well-defined symbol names. A device named ABC would live in one of the predefined device libraries, such as *libMitDevices.so* (or *MIT$DEVICES* in the days of VAX/VMS), and would contain symbols named *abc__init()* and *abc__add()*. This evolved naturally into TDI, the language built-in to MDSplus. This allowed defining devices in a higher-level language, without the need for strict variable typing or memory management. The example of an ABC device would now be defined as a collection of files named *abc__init.fun*, *abc__add.fun*, and so on.

3.2 Python Devices

As time progressed and MDSplus began to embrace Python, it became worthwhile to engineer a way to write device codes in it. The infrastructure was added to the MDSplus Python API [10] to define Devices as classes, subclassing *MDSplus.Device*. These Python classes would be stored on the search path defined in the *MDS_PYDEVICE_PATH* environment variable and found at runtime. This provided a modern and easy to learn method for writing device codes. Additionally, the plethora of libraries available to Python makes it a great choice for this. In fact, many devices already have APIs available to Python for interacting with them.

3.3 Rigid Representation

The underlying structure of an MDSplus Device is a collection of nodes, this has already been explained. However, the piece that is missing is how interconnected these nodes actually are. Designed for efficiency back in the 80s, accessing the nodes that constitute a device was done by offset, not with string comparisons. Put differently, the position of a node in the list defines how it is used, not the name. This can be counterintuitive if you want to add a new node in the middle of the list, or modify the list of nodes in any way. Effectively, once the device is added to the MDSplus tree, the structure of the nodes that represent the hardware becomes *immutable*. For most applications, and for devices who's features never change, this was sufficient, but this proves problematic when updates are made that require nodes to be added or removed. The only solutions are to create a new Device with a different name, or to delete and re-add the existing Device from the tree. Both of these solutions can cause issues as the number of devices can grow out of control.

4. New Flexible Design

To solve the limitations described above, the following approach was taken. To start, an immutable set of nodes is still required. However, these will only be the minimum set needed to connect to the device and define operational modes. These nodes will not change with each reconfiguration, and will usually represent things that are static.

To start, an immutable set of nodes is still required. However, these will only be the minimum set needed to connect to the device and define operational modes. These nodes will not change with each reconfiguration, and will usually represent things that are static (see Figure 1), for example, the network address, the model and modules of the hardware.

4.1 Configuring

Once those minimal set of nodes are inserted into the tree, you can then start the configuration process (see Figure 1). This involves running a method while the tree is open for edit, allowing the method to change the tree structure. It is during this process that the flexibility is most apparent.

In order to determine the details required to represent a given device, it can be useful to simply ask. By querying the device for installed modules and available features, we can programmatically generate a list of required nodes.

As an example, we describe the effects of configuring and reconfiguring a D-tACQ digitizer. If we query an ACQ2106 [11] chassis and it informs us that there is an ACQ435ELF module installed in site 1, we can add a *SITE1* node with 32 *INPUT_XX* nodes to store the recorded data, and nodes for any module-specific configuration options.

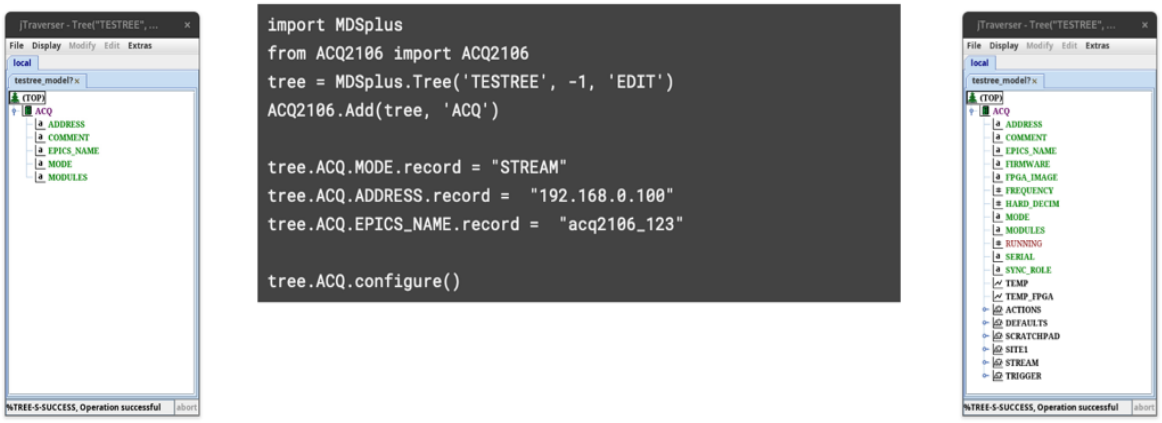

Figure 1: Example of the state of the MDSplus device in the tree. On the left, the device shown has just been added. On the right, the device shown has been configured in a streaming mode.

author's email: fsantoro@mit.edu

Once the list of nodes is determined, the device will ensure each node exists. If the node already exists, this will do nothing. Any default values will only be applied to newly created nodes, or to all nodes if the *overwrite=True* parameter is specified.

The device code keeps track of all nodes that were requested, and then handles all of the extraneous nodes, if any. These nodes are usually left over from a previous configuration. By default, these nodes will simply be turned off, in order to prevent the accidental loss of data. However, you can pass *delete_nodes=True* in order to fully clean any existing nodes out of your tree structure.

4.2 Reconfiguring

This structure can then be changed at any time, for any reason. By calling the configure method again, the tree can update to match changes in the hardware or settings. For example, if you install a new module in your ACQ2106 digitizer chassis, it will detect that and add the required nodes.

For devices that support multiple operational modes, this is especially important. Figure 2 shows a device that was originally configured for streaming data capture being reconfigured for transient data capture, and back again.

4.3 Node Tooltips and Metadata

Working with devices this complex stretched our ability to document them in a way that a user can digest. A way to ease this pain is by adding tooltips to most nodes (See Figure 3), containing information such as intended use, side effects, and pitfalls. Additionally, we've added the ability to define constraints on the value of a node that will be discussed further in the Validation section. All of these new attributes have been added as Extended Node Characteristics (XNCI) to the nodes as they are added to the tree. We intend to formalize them and add support to the various traverser applications to include this information as part of the actual tooltips.

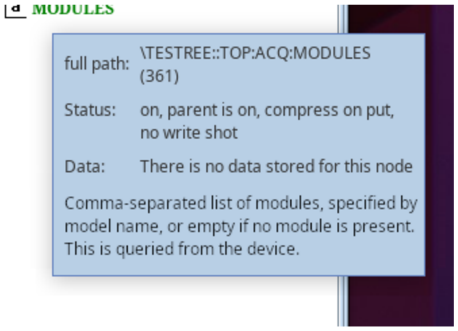

Figure 2: An example of tooltips while hovering over the *MODULES* node of the tree.

4.4 Validation

As a pleasant side effect of adding so much metadata about the nodes to the driver, we were also able to implement a validation method. The method will first check if communication with the device is working. Then, for each node with data we verify that the value is within the min/max (if specified), or in the list of approved values (if specified). Additionally, we check if any properties have changed that would require you to reconfigure the device, such as the modules, firmware, or FPGA image. Any issues that are found are printed to the console, so it is intended to be run directly by a user before an experiment.

5. Conclusions

We found that this method works very well for devices such as the ACQ2106 that are so highly configurable. For devices that are entirely static, we still recommend using the existing designs for device codes. However, the additional metadata for nodes is useful regardless, as is the validation method, and we recommend those be widely adopted. Due to the limitations presents in the original rigid design it is impossible to convert an existing device to the new flexible design without rewriting it, and removing/readding it everywhere its used.

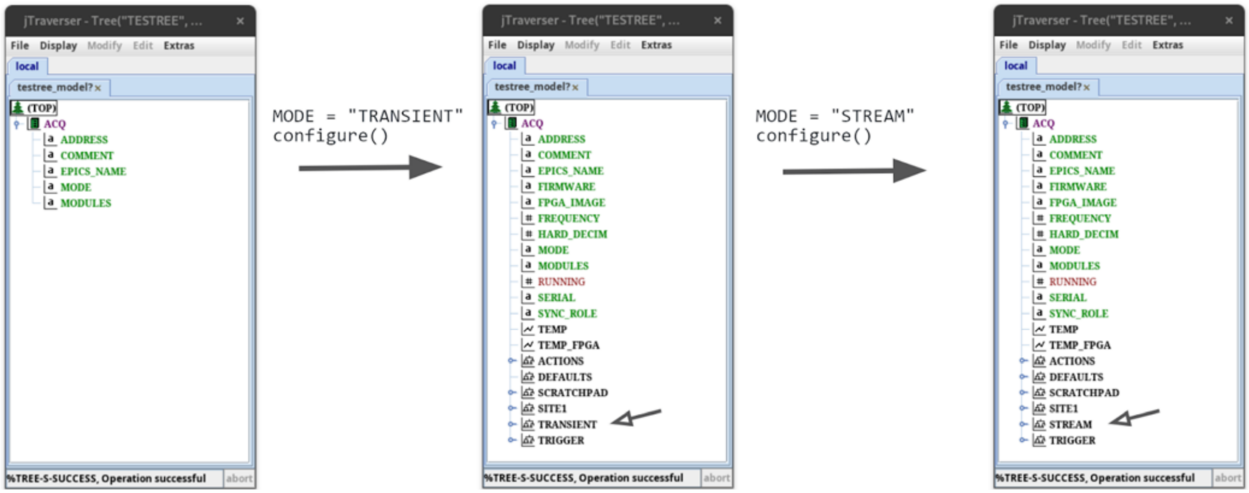

Figure 3: The figure above shows the initial state of the tree right after the device is added (left window). Then, the state of the tree after configuring for transient data capture (center window). Finally, the state of the tree after configuring for streaming data capture (right window).

author's email: fsantoro@mit.edu

We look forward to seeing what the community will do with this new design.

Acknowledgements

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

References

[1] Stillerman, Joshua, and Thomas W. Fredian. "The MDSplus Data Acquisition System, Current Status and Future Directions." Fusion Eng. Des. Vol. 43, Issue 3, January 1999. https://doi.org/10.1016/S0920-3796(98)00398-6.

[2] Fernando Santoro, Joshua Stillerman, Stephen Lane-Walsh, Thomas Fredian. "Exploring MDSplus data-acquisition software and custom devices". Fusion Engineering and Design, Volume 155, Issue C, June 2020. https://doi.org/10.1016/j.fusengdes.2020.111534.

[3] TDI (Tree Data Interface) Information, https://mdsplus.org/index.php?title=Documentation:Reference:TDI&open=101601206494100979775&page=Documentation%2FReference%2FTDI (accessed 26 September 2024)

[4] EPICS Experimental Physics and Industrial Control System. https://epics-controls.org/ (accessed 14 November 2024)

[5] P.-H Rebut, the ITER Joint Central Team and Home Teams, "ITER: the first experimental fusion reactor". Fusion Eng. Des., Vol. 27, March 1995. https://www.sciencedirect.com/science/article/abs/pii/0920379695901132

[6] The UK Atomic Energy Authority (UKAEA). https://www.gov.uk/government/organisations/uk-atomic-energy-authority (accessed 12 November 2024)

[7] Oh Y.K. et al, "Commissioning and initial operation of KSTAR superconducting tokamak", Fusion Eng. Des. Vol 84, Issues 2–6, June 2009. https://doi.org/10.1016/j.fusengdes.2008.12.099

[8] V. Toigo, et al., "The ITER neutral beam test facility towards SPIDER operation", Nucl. Fusion 57 (2017), 086027. https://doi.org/10.1088/1741-4326/aa7490

[9] Basler https://www.baslerweb.com/en-us/shop/aca800-200gc/ (accessed 12 November 2024)

[10] T. Fredian , J. Stillerman , G. Manduchi. "MDSplus objects—Python implementation". Fusion Engineering and Design, Vol 85, Issues 3–4, July 2010. https://doi.org/10.1016/j.fusengdes.2010.04.029

[11] D-tAcq, "ACQ2106 6 site ELF carrier". https://www.d-tacq.com/acq2106.shtml (accessed 26 September 2024)

author's email: fsantoro@mit.edu