ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUBR01

NOMINAL DEVICE SUPPORT (NDSv3) AS A SOFTWARE FRAMEWORK FOR MEASUREMENT SYSTEMS IN DIAGNOSTICS*

R. Lange†, ITER Organization, St. Paul lez Durance, France
M. Astrain, V. Costa, D. Rivilla, M. Ruiz, Grupo de Investigación en Instrumentación y Acústica Aplicada, Universidad Politécnica de Madrid, Madrid, Spain
J. Moreno, D. Sanz, GMV Aerospace and Defence, Tres Cantos, Spain

Abstract

Software integration of diverse data acquisition and timing hardware devices in diagnostics applications is very challenging. While the implementation should manage multiple hardware devices from different manufacturers providing different applications program interfaces (APIs), scientists would rather focus on the high-level configuration, using their specific environments such as Experimental Physics and Industrial Control System (EPICS), Tango, the ITER Real-Time Framework or the MARTe2 middleware.

The Nominal Device Support (NDSv3) C++ framework, conceived by Cosylab and under development at ITER for use in its diagnostic applications, uses a layered approach, abstracting specific hardware device APIs as well as the interface to control systems and real-time applications.

ITER CODAC and its partners have developed NDS device drivers using both PCI express extension for instrumentation (PXIe) and Micro Telecommunications Computing Architecture (MTCA) platforms for multifunction data acquisition (DAQ) devices, timing cards and field-programmable gate array (FPGA) based solutions. In addition, the concept of an NDS-System encapsulates a complex structure of multiple NDS device drivers, combining functions of the different low-level devices and collecting all system-specific logic, separating it from generic device driver code.

INTRODUCTION

The Instrumentation and Control Systems (I&C) used in big science facilities (BSF) are based on the use of multi-tier software applications. For example, advanced DAQ and timing systems include complex hardware elements that need software elements to configure all their functionalities. In the last years, the use of field-programmable gate arrays (FPGAs), System on a Chip circuits (SoC) and new development tools have demonstrated that software is a key part of implementing these systems [1]. The key points of using software in advanced I&C systems are adaptability, reusability and maintainability over the entire BSF project lifetime.

The Nominal Device Support (NDS) software framework has been implemented to meet these three goals. Initially developed by Cosylab [2], it was recently improved and extended by the ITER Organization, working with Universidad Politécnica de Madrid and GMV Aerospace and Defence. NDS is a driver development software framework for diagnostics measurement systems [3], focusing on data acquisition and timing devices. NDS device drivers are instantiated and configured to build complex systems, designed to solve specific applications. The applied methodology simplifies code reusability and testability, achieving high levels of software quality. Doxygen documentation, automated tests and static code analysis are used in all NDS modules. Specifically, the NDS framework provides a simplified solution for device driver development in I&C systems that use the Experimental Physics and Industrial Control System (EPICS) [4-6].

NDS SOFTWARE LAYERS

Figure 1 shows the basic structure of a device driver in the NDSv3 framework. The application, called "control system", uses the generic NDS control system interface to communicate with NDS device drivers and extensions. The NDS device drivers use the base and helper classes from the NDS-core library to access the hardware through the operating system's low level drivers.

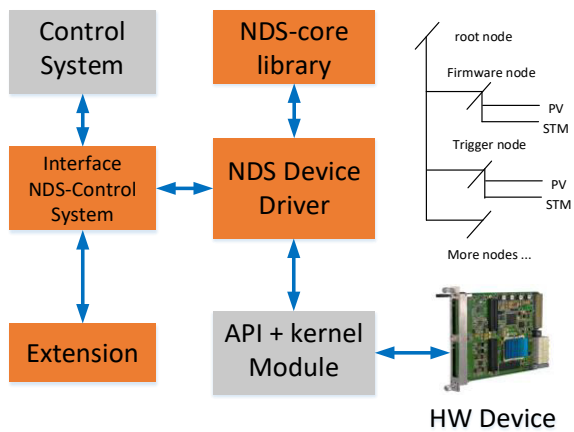

Figure 1: NDSv3 framework elements and basic layers.

NDS-Core

The NDS-core layer (NDS-core library) provides a collection of C++ classes and helpers that standardise and simplify the implementation of the software device driver (NDS device driver) for a specific hardware device or communication interface.

* This work was supported in part by the Spanish Ministry of Science, Innovation, and Universities Projects under Grant ENE2015-64914-C3-3-R and Grant PID2019-108377RB-C33, in part by the Spanish Ministry of Science, Innovation, and Universities Mobility under Grant PRX19/00449, and in part by the Comunidad de Madrid under Grant PEJD-2018-PRE/TIC-8571 and Grant PEJ-2019-AI/TIC-14507.

† ralph.lange@iter.org

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUBR01

A device driver implemented with NDS-core can be used by different kinds of applications called "control systems". Examples for such applications are EPICS and the Real-Time Framework developed by ITER [7].

The basic objects of an NDS driver are *nodes* and *process variables (PVs)*. As with previous versions of NDS (see NDSv2 [2]), every variable has a unique name. The structure of this name represents the variable's place in the facility, system, or subsystem. In NDSv3 this concept is maintained, but through abstraction much of the complexity is hidden from the end user. NDS PVs are named data or structures of data relevant to the experiment. To organise them, they are appended to a named node. Subsequently, nodes can be appended to other nodes, creating a tree-like structure that represents the device.

For example, a device might have Analog Input and Analog Output functionalities; these would be appended to a root node or port node to generate the following names: device(node)-ai(node)-ch0(node)-data(PV); device(node)-ao(node)-ch0(node)-data(PV).

Note that two different hardware devices manufactured by different companies implementing the same or similar functionalities can have an NDS device driver with a big part of the code in common. The most crucial point is that they have the same NDS PVs: if these match in name and functionality; software management and device interchangeability are simplified. This is one of the key concepts in the design of NDS based systems at ITER.

NDS-core provides several node and PV type classes to cover most functionality required in a device:

- *Base*: The base node is just a name holder.
- *Port*: A special node that connects with the NDS control system interface layer (see below). The names of PVs are generated from the ports they are appended to.
- *State Machine*: A node with a particular set of functions implementing finite state machine behaviour. The state machine node has a generic set of PVs that control the transitions or propagate states to other state machines below to form hierarchical state machines.

The NDS-core library allows the creation of different NDS PV types. NDS PVs can store a variable or trigger a specific function call, named *delegate*. The read/write operation of an NDS PV allows to read/write the data or trigger the delegate function. The assigned direction, i.e., the definition as an "input" or "output", is determined from the control system's point of view. The NDS PV types (classes) are:

- *Variable In*: This PV contains a value obtained from the hardware device to be sent to the control system. There are two different methods to deliver the data: The control system executes a read operation, accessing the scalar or array value, or the device driver executes a push operation, creating an interruption to the control system.
- *Delegate In*: This PV does not have an attribute, but calls a method when executing a read operation (defined in NDS as a *delegate method*). Typically, this method operates on the hardware through the specific system device driver to read a value.
- *Variable Out* and *Delegate Out*: These PVs contain values received from the control system to be used by the device driver or support the trigger of a function that performs a write operation into the system device driver.

The NDS drivers built in this way are able to communicate using the NDS control system interface layer. This layer has to be implemented once for each control system that needs to use NDS and can be used with all NDS devices drivers.

To control the instantiation of a device and ensure uniqueness of PV naming, the NDS-core layer manages the creation and destruction of device driver instances using a factory (following a singleton pattern implementation). Through the factory, it is possible to get the information about registered drivers and instances. In addition, it gives access to the driver's nodes (hierarchical), the NDS PVs and the state machines.

An important functionality of the NDS-core is the subscription and replication of PVs. When two NDS PVs are connected with a subscription (only possible from an NDS PV In to an NDS PV Out), pushing a value in the NDS PV In generates an update in the NDS PV Out. Similarly, two NDS PV In can be connected by a replication, where pushing a value in one NDS PV In replicates its value to the other. Note that both mechanisms, subscription and replication, are unidirectional. They allow sharing of information between drivers and nodes without the intervention of the control system, which is crucial for nodes related to archiving and real-time communication.

The NDS-core software module consists of:

- The source code implemented in C++ 11.
- Test code implemented with the help of Google-Test [8]. This test code includes a basic implementation of an NDS "control system" (see below) for test and debugging purposes.
- Doxygen [9] support files.

The NDS-core layer does not depend on other software modules (particularly from the EPICS project) and uses the C++ Standard Library. NDS-core can be built on Linux and Windows operating systems.

Control System Interface

The control system interface of the NDSv3 framework implements the interface to communicate with the application that runs the NDS device. Currently, the NDSv3 framework contains two interfaces: one exclusively for running tests of the implemented device drivers (the test control system mentioned) and one to interface with the EPICS control system. The NDS-EPICS control system interface module connects EPICS process database records to NDS PVs. NDS-EPICS is based on the use of the asynDriver module [10, 11], which implements the EPICS Device Support for different records using standardised interfaces. Using NDS-EPICS, the user only needs to define the record templates and the substitutions files. (Examples

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUBR01

contained in the software unit.) Creating the interface to a hardware device in EPICS is straightforward if the NDS device driver provides such EPICS templates.

A third implementation of the control system interface, for the TANGO control system toolkit [12], exists in the original NDSv3 development by Cosylab, but has not been used or tested in the context of the ITER developments.

The NDS-EPICS software module consists of:

- The source code of the NDS-EPICS interface.
- An application generating an EPICS Input Output Controller (IOC).
- Test code, implemented using Python with the help of the pyEpics library. These tests verify all implemented asynDriver interfaces.
- Doxygen [9] support files.

DEVICE DRIVERS

The NDS device drivers are plugin libraries. Changing or adding a device drivers does not require to change or recompile the core modules of NDS.

Adding a Device Driver

The development of an NDS device driver using the NDS-core library requires the following steps:

- Analysis of the functional blocks available in the hardware device. This often means a mapping of the Application Program Interface (API) functions provided by the manufacturer in charge of managing the different hardware parts.
- Definition of the hierarchical driver organisation using a tree-like structure of nodes. These nodes can be new classes created by the user or complex nodes from the NDS-core library, e.g., DAQ, Waveform Generation (WFG), Digital Input Output (DIO), Trigger and Clock, Routing, Timestamp or Future Time Event.
- Definition of the operations to be executed in the state machine transitions available in each node. NDS-core defines the states UNKNOWN, INITIALIZING, OFF, ON, RUNNING and FAULT.
- Implementation of the driver constructor. This code includes the initialisation of the resources needed and the creation of the driver hierarchy. In addition, creating the nodes (defined by the user or from the NDS-core library) requires implementing the methods associated with the state machine transitions and the getters/setters of the NDS PVs of type delegate in and out.
- Completion of the methods that call the API of the low level driver for the specific hardware.
- Implementation of the test code, using the test control system and the GoogleTest library.
- Source code documentation using Doxygen.
- Generation of the packages for driver distribution.

The steps to add the interface with EPICS using the NDS-EPICS module are:

- Creation of a software unit including an EPICS application and an EPICS IOC. This application needs to be compiled/linked against the NDSv3 libraries (nds-core, nds-epics) and the asynDriver module.
- Development of the specific EPICS database templates using the examples included in the NDS-EPICS module. Creation of the EPICS substitutions files.
- Configuration of the IOC using the st.cmd file, instantiating the driver and loading the databases.
- Verification of the IOC through an Operator Interface (OPI) or Channel Access operations.
- The NDS-EPICS layer offers the possibility to easily extend the driver implementation with functionalities specific to the EPICS Control System (e.g. execution of init or exit hooks)..

Existing Device Drivers

Table 1 shows a set of device drivers for the PXIe and MTCA platforms that have been developed at different facilities. Functionally, they can be split into four different groups:

- Timing cards (using the PTP standard IEE1588)
- Multi-function DAQ cards
- High sampling rate DAQ cards
- FPGA-based DAQ cards (including a customisable platform).

Of the two solutions in the last group, one is using the FlexRIO device from National Instruments (NI) that can be configured using LabVIEW/FPGA, and the other is based on a device from Advanced Industrial Electronic Systems (AIES), using Hardware Description Language (HDL) with XILINX Vivado.

Table 1: NDS Device Drivers

Devices	PXIe	MTCA
Timing IEEE1588	NI PXI6683H	PTM1588
DAQ	NI X-Series NI PXIe636x	Teledyn ADQ14 MFMC-FMC168 STRUCK SIS300
DAQ-FPGA	FlexRIO (LV FPGA)	MFMC

All device drivers packages contain the driver source code (see Figure 2), Doxygen-based generated documentation, test code developed with GoogleTest, a fully configured EPICS IOC application and Python code testing the EPICS IOC and OPI panels to use the driver using Control System Studio [13].

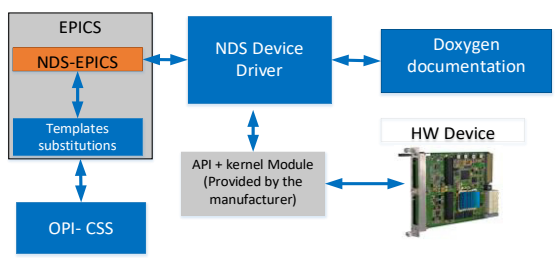

Figure 2: Elements in an NDS Driver software package.

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUBR01

Normally, such NDS device driver modules do not contain information about any specific use or application. They are generic drivers, meant to be integrated into bigger systems that implement specific use cases and include the specific knowledge. This is achieved by combining different driver modules using the extensions introduced in the next section.

EXTENSIONS

Expanding the basic NDS concepts, the ITER Organization and its partners have contributed to the NDS open-source project by improving the existing and providing additional extensions.

NDS Complex Nodes

The main functionalities required to create diagnostics and control instruments were identified and converted into pre-defined functional NDS nodes, called NDS complex nodes. With the intention to homogenise the driver development, the NDS-core library contains nodes supporting common functionalities: DAQ, WFG, DIO, Timing, Triggering, Clock and Routing. In these nodes, the PVs already have defined names, identifying their specific functionality. Therefore, an NDS device driver is in charge of implementing the configuration, status management and data acquisition with the help of software tiers provided by the manufacturer (kernel module and user space API) and the NDS-core library. Additionally, the complex nodes provide an API to simplify calls to update PV values and timestamps. Thus, while the set of PVs provided by the complex nodes is fixed, the final driver implementation can add and manage more PVs for additional functionality.

NDS Plugins

The implementation of a specific application requires the use of multiple device drivers as well as other communication interfaces. In the case of the ITER experiment, there are two such interfaces: the synchronous data bus network (SDN) is oriented to data interchange using a network with a controlled latency and the data archiving network (DAN) is used to stream high throughput data to the archiving engines.

Figure 3 shows the basic architecture for the NDS interfaces to both DAN and SDN.

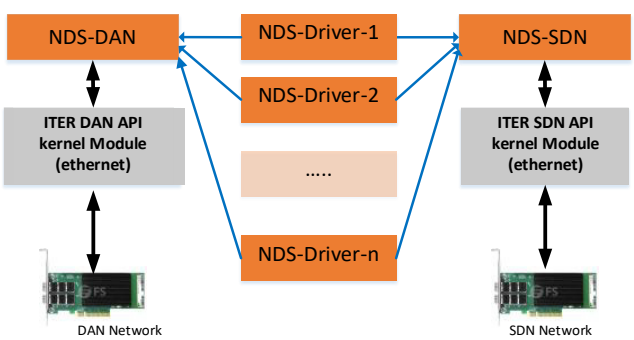

Figure 3: Plugins in an NDS-based application. The blue arrows show the NDS subscriptions.

In order to simplify the use of both interfaces for the developer, the NDS framework defines the concept of an NDS plugin. An NDS plugin is an NDS device driver that requires configuration by the user with the specific parameters of the user's application. This configuration is done through a file that contains all the configuration details in extended mark-up language (XML).

For the SDN plugin, the XML file contains the identification of the SDN topic with its data organisation as well as the mechanism of when to publish the topic. The topic can contain scalar or array fields (of different data types) and structures of these. Each field in the topic is connected to an NDS PV (using the subscription method). The topic is published either when a field changes or periodically with a publishing period defined in the XML file.

For the DAN plugin, the XML configuration similarly contains the details of the NDS PVs (from device drivers) that provide the values (scalar or arrays) that need to be archived.

Besides these two plugins, there is an additional one that implements the access to the EPICS pvAccess protocol. This solution provides a method for NDS based drivers to access industrial or legacy equipment that is interfaced using EPICS IOCs with standard drivers or other applications like middle layer services that provide a pvAccess interface. All EPICS PVs published by these IOCs or software applications can be accessed by NDS applications for reading, writing, and monitoring. This plugin is named NDS-PVXS because it uses the library developed by the PVXS Project [14].

NDS Systems

Developing a diagnostic or an instrument in a big science facility requires integrating multiple hardware devices using different software device drivers. The application developer needs to orchestrate all these hardware elements to implement a correct sequence of operations, i.e., trigger configuration, data acquisition, time stamping, archiving, etc.

In the NDS framework, we have defined the concept of an NDS system, which is a special NDS device driver that manages hierarchically other NDS device drivers (that have already been implemented and tested). This is a key concept, because the developer does not need to program or change anything in the existing NDS device drivers but only configures them for their application. Using this approach, the developer is merely a user of the NDS device drivers and NDS plugins, responsible for configuring and coordinating their use. This noticeably reduces the development time and simplifies the integration of the final solution.

An NDS system is implemented for a specific use case solving the specific requirements. It is not a generic application, and it can be designed to only expose the specific NDS PVs needed, reducing the number of PVs that need to be managed by the control system.

Figure 4 shows the block diagram of an NDS system. A set of C++ classes for the different device drivers implement the subscriptions and replications of NDS PVs to

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUBR01

manage an NDS device driver from the NDS system level. The shown structure is the diagram of the system that has been developed as a working example for the NDS system extension.

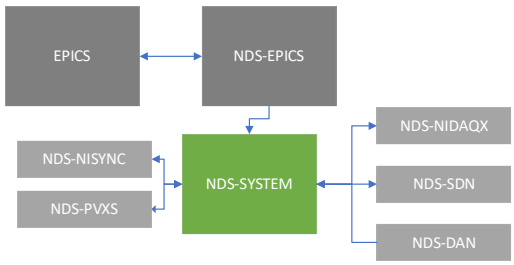

Figure 4: NDS device drivers and NDS plugins in an NDS system.

ADVANCED FPGA-BASED SYSTEMS

The use of advanced FPGA-based DAQ devices, which implement functionalities that can be customised, requires software tools that simplify development and integration. For such applications, the existing NDS framework provides two different solutions.

The first one is based on National Instruments FlexRIO technology, which includes an FPGA that can be configured using the LabVIEW/FPGA tool. In addition, the IRIO Project [15] provides a library that simplifies the integration of these devices. On top of that library, NDS implements an NDS device driver named NDS-IRIO. To implement a custom algorithm in the FPGA using this set of tools, the developer would follow the IRIO design methodology, which automatically connects the FPGA code to the application and generates PVs with the names of the LabVIEW variables, requiring no modifications of the driver code for different use cases.

The second approach is based on the use of the new FPGA, SoC and Adaptive Compute Acceleration Platform (ACAP) with design methodologies that use High Level Synthesis and the OpenCL standard. The NDS-IRIO-OpenCL module implements a software layer that simplifies the integration of such systems [16].

SOFTWARE QUALITY

The software modules implemented in the NDS framework have been developed using the Software Quality model implemented at ITER CODAC. The model is based on the ISO/IEC 15288 standard and there are documents for the different elements: the Software Requirements Specification (SRS), the Software Architecture and Design Description (SADD), the Software Test Plan (STP), the Software Test Report (STR) and the Software User Manual (SUM).

The most meaningful data are the results of the static code analysis and the test coverage of the different modules, obtained with the lcov software utility and a SonarQube [17] installation. The average test coverage value across the different modules is around 70% and has been increased with every new release. Static analysis shows no issues; the cumulated number of reported code smell problems is less than 1%.

CONCLUSIONS

The NDSv3 framework provides a complete set of software modules and documentation to develop complex diagnostics solutions involving timing, data acquisition and other advanced communication interfaces. The NDS implementation model is based on the use of specific device drivers and configurable plugins. The driver developer and the diagnostician can work separately, increasing efficiency and reliability by separating the software layers that contain device specific and application specific knowledge.

The ITER NDS complex nodes standardise most of the PV names, easing the usage of different hardware devices, which should share most PVs. As an added benefit, porting the device driver from one manufacturer API to another should re-use more than 80% of the code.

The new generation of NDS drivers makes use of the different plugins interchangeably and is able to connect to multiple control systems and other software applications. The extensively tested drivers are reaching a very respectable percentage of verification following ITER guidelines, paving the way to a successful deployment in the facility.

PROJECT CONTRIBUTORS

NDSv3 was conceived and originally implemented by Cosylab.

The institutions involved in the recent developments are the ITER Organization, Universidad Politécnica de Madrid (grupo de Investigación en Instrumentación y Acústica Aplicada), GMV Aerospace and Defence, the European Spallation Source and UKAEA-JET/MAST.

The NDS source code for the initial version of NDSv3 is available at GitHub [18, 19]. The extensions implemented by ITER, the device drivers, the plugins and other contributions are available at the ITER Git server [20].

REFERENCES

[1] M. Astrain *et al.*, "Real-Time Implementation of the Neutron/Gamma Discrimination in an FPGA-Based DAQ MTCA Platform Using a Convolutional Neural Network", *IEEE Transactions on Nuclear Science*, vol. 68, no. 8, pp. 2173-2178, Aug. 2021. doi:10.1109/TNS.2021.3090670

[2] V. Isaev, N. Claesson, M. Plesko, and K. Žagar, "EPICS Data Acquisition Device Support" in *Proc. 14th Int. Conf on Accelerator and Large Experimental Physics Control Systems (ICALEPCS2013)*, San Francisco, CA, USA, Oct. 2013, paper TUPPC059, pp. 707-709.

[3] J. Vega *et al.*, "New developments at JET in diagnostics, real-time control, data acquisition and information retrieval with potential application to ITER", *Fusion Engineering and Design*, vol. 84, issue 12, pp. 2136-2144, Dec. 2009. doi:10.1016/j.fusengdes.2009.02.055

[4] EPICS Project, https://epics-controls.org

[5] L. R. Dalesio *et al.*, "EPICS 7 Provides Major Enhancements to the EPICS Toolkit", in *Proc. 16th Int. Conf. on Ac-*

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUBR01

celerator and Large Experimental Physics Control Systems (ICALEPCS2017), Barcelona, Spain, Oct. 2017, pp. 22-26. doi:10.18429/JACoW-ICALEPCS2017-MOBPL01

[6] A. N. Johnson *et al.*, "EPICS 7 Core Status Report", in *Proc. 17th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS2019)*, New York, NY, USA, Oct. 2019, pp. 923-927. doi:10.18429/JACoW-ICALEPCS2019-WECPR01

[7] L. Zabeo *et al.*, "Work-flow process from simulation to operation for the Plasma Control System for the ITER first plasma", *Fusion Engineering and Design*, vol. 146, part B, pp. 1446-1449, 2019. doi:10.1016/j.fusengdes.2019.02.101

[8] GoogleTest/GoogleMock, https://google.github.io/googletest

[9] Doxygen, https://www.doxygen.nl/index.html

[10] M. R. Kraimer *et al.*, "EPICS: Asynchronous Driver Support", in *Proc. 10th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS2005)*, Geneva, Switzerland, Oct. 2005, paper PO2.074-5.

[11] asynDriver, https://epics-modules.github.io/master/asyn

[12] Tango Controls, https://www.tango-controls.org

[13] Control System Studio, https://controlsystemstudio.org

[14] PVXS PVA client/server library, https://mdavidsaver.github.io/pvxs

[15] M. Ruiz *et al.*, "IRIO technology: Developing applications for advanced DAQ systems using FPGAs", *2016 IEEE-NPSS Real Time Conference (RT)*, 2016, pp. 1-5, doi:10.1109/RTC.2016.7543090

[16] M. Astrain *et al.*, "A methodology to standardize the development of FPGA-based high-performance DAQ and processing systems using OpenCL", *Fusion Engineering and Design*, vol. 155, 2020, 111561, doi:10.1016/j.fusengdes.2020.111561

[17] SonarQube, https://www.sonarqube.org

[18] NDSv3-core on GitHub, https://github.com/Cosylab/nds3

[19] NDSv3-EPICS on GitHub, https://github.com/Cosylab/nds3_epics

[20] ITER Git Repositories (account required), https://git.iter.org/projects/CCS