IDM UID
Q79UMH
VERSION CREATED ON / VERSION / STATUS
23 Jan 2017 / 1.2 / Approved
EXTERNAL REFERENCE / VERSION

Report

D5.1 NDS User Manual

NDS user documentation including the SDN and DAN plugin usage instructions

Approval Process			
	Name	*Action*	*Affiliation*
Author	**Zagar A.**	**23 Jan 2017:signed**	
Co-Authors			
Reviewers	**Ghate A.**	**23 Jan 2017:recommended (Fast Track)**	**IO/DG/COO/SCOD/CSD/PCI**
Previous Versions Reviews	**Lange R.**	**16 Jan 2017:reviewed v1.1**	**IO/DG/COO/SCOD/CSD/CDC**
Approver	**Simrock S.**	**27 Jan 2017:approved**	**IO/DG/COO/SCOD/CSD/CDC**
Document Security: Internal Use *RO: Simrock Stefan*			
Read Access	**LG: LSDUC, AD: ITER, AD: External Collaborators, AD: IO_Director-General, AD: EMAB, AD: OBS - Control System Division (CSD), AD: OBS - CODAC Section (CDC), AD: Auditors, AD: ITER Management Assessor, project administrator, RO**		

Change Log			
D5.1 NDS User Manual (Q79UMH)			
Version	***Latest Status***	***Issue Date***	***Description of Change***
v0.0	In Work	21 Dec 2016	
v1.0	Signed	29 Dec 2016	* Completely rewrote chapter '17. NDS Plugins' due to the poor content in the previous version and to update according to the latest changes * Added better description for the '17.2 DAN Plugin' and the new '17.3 SDN Plugin' * Added disclaimer about the ITER naming convention in '5. PV Record naming' * Fixed chapter/section numbering * Added table of contents and the list of references (1.4) * Fixed various minor issues throughout the document
v1.1	Signed	04 Jan 2017	Updated also the rest of chapter 5. 'PV Record naming' to comply with ITER standards.
v1.2	Approved	23 Jan 2017	Replaced the example for SDN plugin with the actual topic structure used in D1-I9 use case, following the diagnostics requirements.

Nominal Device Support, User's Manual

User's Manual

Revision:	2.3.10
Status:	Release
Project:	Nominal Device Support
Last modification:	Thu Jan 19 14:23:47 2017 +0100
Created:	Mon May 28 21:47:21 2012 +0000

Prepared by

Matjaz Bercic
Niklas Claesson
Slava Isaev
Anze Zagar
Klemen Zagar

COSYLAB
CONTROL SYSTEM LABORATORY

Contents

1. Introduction

This User's Manual provides the information necessary for the user to effectively use the interfaces of a nominal device.

1.1. System Overview

The Nominal Device Support (NDS) Library, the C++ implementation of the Nominal Device Model (NDM), generalizes EPICS device support for data acquisition and timing devices. It provides sets of interfaces, solutions and best practices of device integration for EPICS.

The rationale behind having a common interface is that a large class of devices offers similar capabilities. If each such device were to have a different interface, engineers using them would need to be familiar with each device specifically, requiring more time for familiarization, inhibiting transfer of knowledge from working with one device to another and increasing the chance of engineering errors due to a misunderstanding or incorrect assumptions.

Specifically, this document describes how to use a device once the IOC that hosts it is running. The device **must be** configured and running according to the instructions given in section 4.1.

Some examples are given that make use of the operating system's shell and assume that `DEVICE` and `CHANNEL` environment variables are set to the device and channel process variable prefix, respectively. In the context of an EPICS database, `${DEVICE}` and `${CHANNEL}` also refer to the device and channel record names, respectively. The naming convention for `DEVICE` and `CHANNEL` is given in the Developer's Manual.

1.2. Organization of the Manual

This manual covers the following topics:

- Model description
- EPICS IOC Shell and Database
- PV Records
- Clocks and Triggering
- Messages
- Conversion between Raw and Engineering Units
- Signal Filtering
- Fourier Transform
- Decimation

- Function Generator
- Firmware Update
- Streaming of Acquired Data
- Error Handling

A reference of the functions is in the Developer's Guide.

1.3. Abbreviations

Abbreviation	Description
AI	Analog Input
AO	Analog Output
CBS	Control Breakdown Structure
CODAC	Control, Data Access and Communication
DAN	Data Archiving Network
DAQ	Data acquisition
FIR	Finite Impulse Response
GPIO	General Purpose Input/Output
IIR	Infinite Impulse Response
IOC	Input/Output Controller
SDD	Self-Description Data
SDN	Synchronous Databus Network
NDM	Nominal Device Model
NDS	Nominal Device Support
SEU	Single Event Upset
TCN	Time Communication Network
UML	Unified Modeling Language

Table 1: Abbreviations

1.4. References

[RD1] Nominal Device Support (NDS): Developer's Manual

[RD2] EPICS Home page (http://www.aps.anl.gov/epics/)

[RD3] EPICS Application Developer's Guide (http://www.aps.anl.gov/epics/base/R3-14/12-docs/AppDevGuide/)

[RD4] EPICS 3-14 Record Reference Manual (https://wiki-ext.aps.anl.gov/epics/index.php/RRM_3-14)

[RD5] asynDriver: Asynchronous Driver Support (http://www.aps.anl.gov/epics/modules/soft/asyn/)

[RD6] GigE Vision: Video Streaming and Device Control Over Ethernet Standard, v2.0

[RD7] CODAC Core System Self-Description Data Editor User Manual (32Z4W2)

[RD8] Plant Control Design Handbook (27LH2V v7.0)

[RD9] I&C signal and variable naming convention (2UT8SH v8.1)

[RD10] Diagnostics Plant I&C Design Documentation (JQLRRK v1.9)

[RD11] Functional Breakdown for Diagnostics Plant I&C (LAJF9S v1.1)

[RD12] DAN User Manual (Q6GULS, v2.5)

[RD13] SDN Tools User Manual (RGSWHS, v1.5)

[RD14] SDN Archiver User Manual (SD29MG, v1.2)

2. The Device Meta-Model

In this document, we consider *devices* that may have zero or more *channels*. The device and its channels are called *objects* – as such, they have a *name* that uniquely identifies them within their scope, and a list of functions that may be invoked upon them.

In addition, a device can have a number of *initialization parameters* that define the device (e.g., the hardware address or device file of the low-level driver).

A function can be invoked upon an object. The actual object and function that is invoked is defined with the following rules:

- A function *handler* is invoked.
- If a handler is not found, the function *default handler* is invoked.

A special function is the *send message*, which allows the user (or another process) to send a message to an object, instructing it to perform an action, change state or similar.

This meta-model is shown in Figure 1.

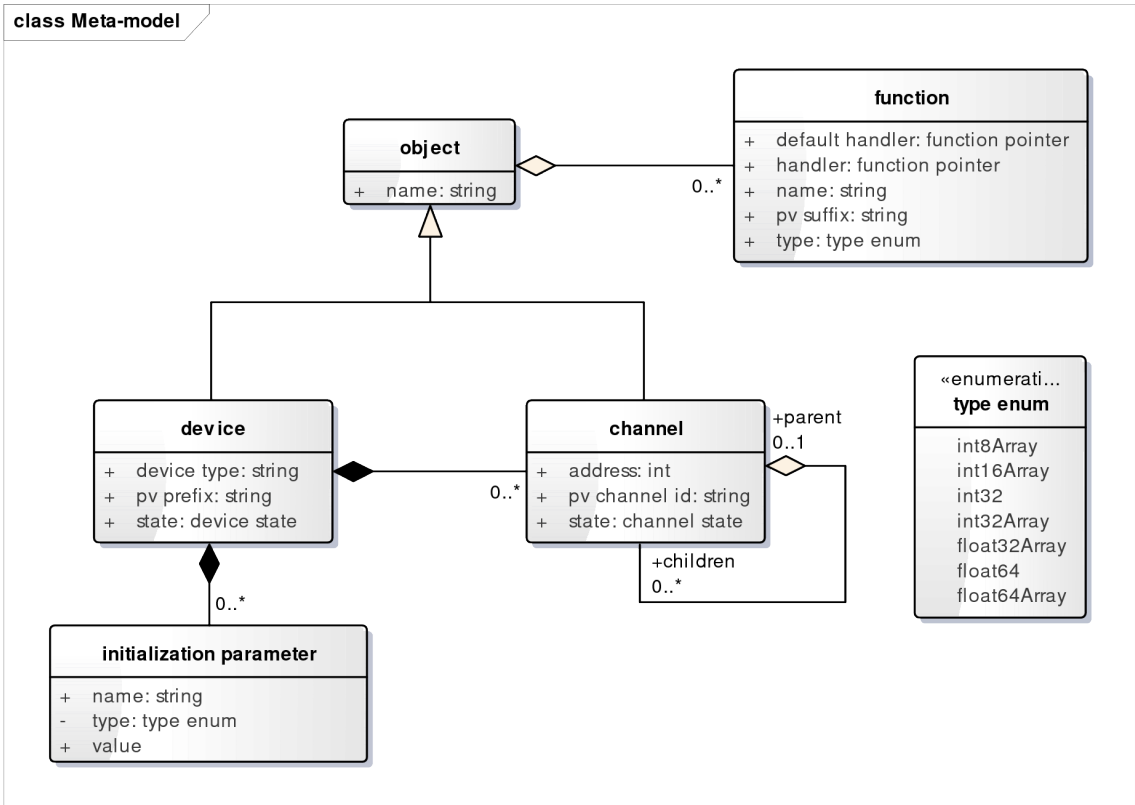

Figure 1: The device meta-model

The meta-model defined in this way covers a wide range of devices:

- Data acquisition (DAQ).
- Signal generators (analog output).
- Digital input and output devices.
- Cameras and other image acquisition devices.
- Timing boards (a combination of a signal generator and digital input/output device).
- Motion controllers (each motion axis, managed by the controller, is modelled as a channel).

The meta-model has the following implementation in mind:

- The EPICS *asynDriver* is used to implement the device driver. In this implementation, the *name* of the device corresponds to the *asynDriver's* port name, and channel's *address* corresponds to *asynDriver's address*.
- Functions of objects can be made accessible via the EPICS Channel Access. The names of process variables are inferred from the *pv prefix*, *pv channel id* and *pv suffix* attributes of device, channel and function, respectively (for device-level process variables, an empty string is considered as the *pv channel id*).
  - The rules by which *pv prefix* and *pv channel id* are defined are not within the scope of the nominal device (meta-)model and can thus be adapted to site-specific conventions.
  - The *handler* of the function is invoked when the process variable is processed in EPICS terms.

2.1. The Device Model

In the present scope of the device model, we consider a class of input/output devices that have the following characteristics:

- A channel can deal either with **digital** or **analog** data.
  - Special cases of analog data are **camera channels**, which are also supported.
- Each channel can be either an **input channel** or an **output channel**.
  - In case of general purpose input/output (GPIO) digital channels, the same channel might be able to serve both directions, but not at the same time.
- A channel's input sampling or output production can be **synchronized** in various ways, e.g., with an external trigger, with a distributed timing system, from software, etc.
- Data processing can be performed on the channel, including: decimation, filtering and conversion from raw acquired data to engineering units.
- The behavior of the device or channel can be configured by invoking functions upon it.

Due to the implementation of a device, some channels may be coupled and share some of the configuration parameters – i.e., while the parameters can be configured, they cannot be set for an individual channel, but rather apply to a group of channels. This is modeled with a **channel group**, which provides those configuration parameters that are common to more than one channel. To the users, the channel group exposes the same interface as a channel would.

Example: Let us assume that a device has 8 analog input channels (`AI0` thru `AI7`), but only two clock sources that dictate the sampling rate – one for channels `AI0` thru `AI3`, and the other for `AI4` thru `AI7`. If each channel were to have a CLKF configuration parameter, the user could inadvertently set `AI0-CLKF` and then `AI1-CLKF`, unaware that the latter action would have overridden the `AI0-CLKF`. Rather, the device would provide two channel groups – `CG0` for channels `AI0` thru `AI3`, and `CG1` for channels `AI4` thru `AI7`. Thus, setting the clock frequency for channels `AI0` thru `AI3` would be performed through `CG0-CLKF`, and for `AI4` thru `AI7` via `CG1-CLKF`.

Example: NI pxi6259 (M series) board's segments are channel groups in NDs terminology.

Note: The model proposed here allows each channel to belong to exactly one channel group. A scenario where there are two or more sets of shared configuration parameters, each applicable to a different subset of channels, and the subsets of channels are not disjoint, would not fit this model.

A UML class diagram depicting the nominal device model is shown in the following figure.

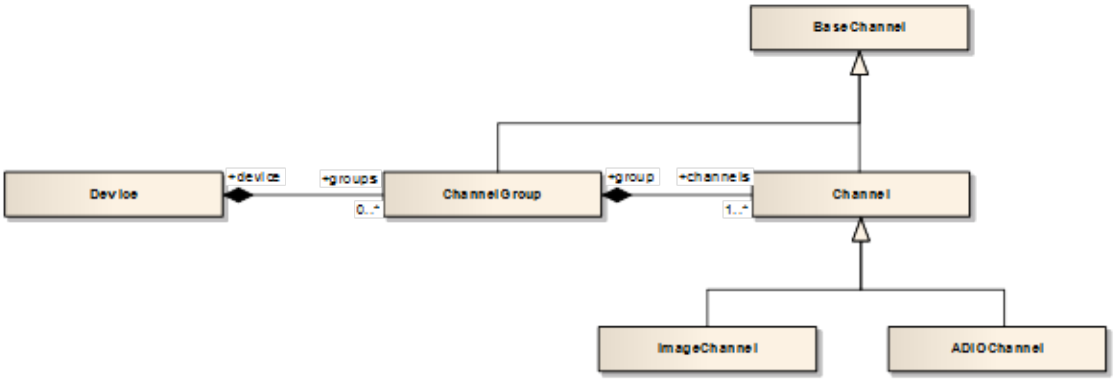

Figure 2: NDS class diagram.

2.1.1. Device State Machine

A device is modeled as a state machine as depicted in the following figure, by an abstract, high-level model. Details of the actual implementation are contained in the Developer's Manual. The arrows in the diagram show logical transitions while the conditions are described within brackets.

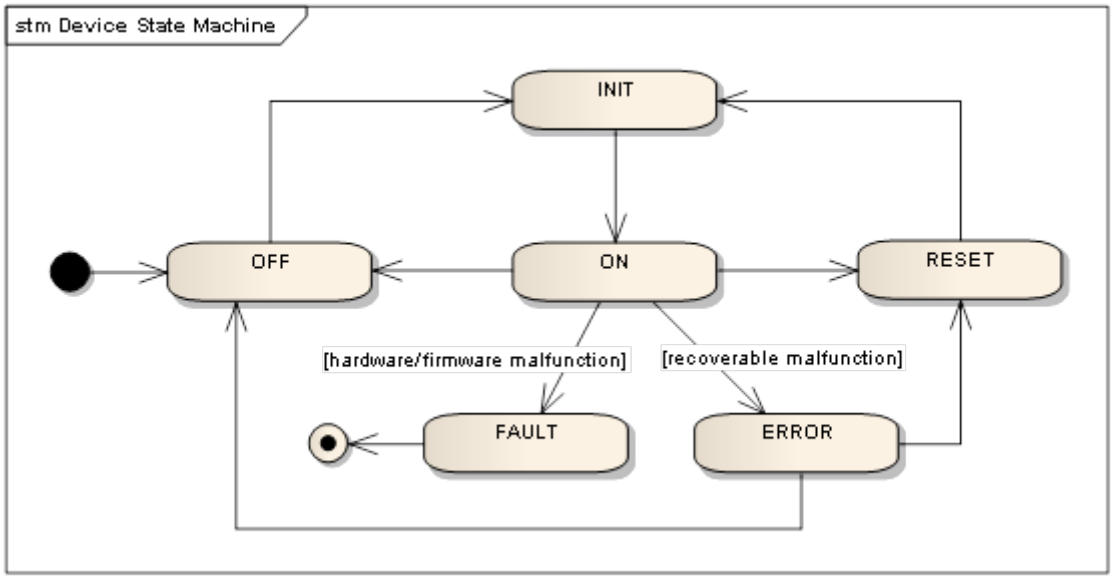

Figure 3: Abstract, high level model of the state machine of the device. The condition description is contained within brackets.

The state machine starts in the `OFF` state, where the device is powered off (if it can be) and the device driver is not connected to it.

When `ON` message is sent to the device, it is requested to be moved to the `ON` state. If initialization is quick, the state machine proceeds directly to the `ON` state, otherwise, it is goes to the `INIT` state while it is initializing. As the device enters the `INIT` state, all of the device's and its channels' configuration parameters are set to their default values. Some of the parameters might only be permitted to change in the `INIT` state. When all initialization conditions are met and all intended configuration is done, the device may be transitioned from `INIT` to `ON`.

Neither the device nor its channels can be used in the `OFF` or `INIT` states. When the device is in the `ON` state some configuration might still be allowed and the device can be used.

The device can be reset in two ways: either by sending a `RESET` message, or a sequence consisting of `OFF` and `ON` messages. In the first case, a soft reset is performed. In the second case, the power of the device is cycled. Depending on hardware support, both methods may actually be implemented in the same way. If the device has no provisions for a soft reset, resetting simply means resetting configuration parameters to the default values.

There are two error states: a recoverable `ERROR` state which can be recovered with a reset, and a non-recoverable hardware or firmware malfunction `FAULT`, which cannot be recovered, and requires manual intervention (e.g., firmware upgrade or replacement of the hardware). The `FAULT` state also reflects the state of the application when the device is not present or found in the system.

While in an error state, neither the device nor its channels can be used. Configuration param-

eters can still be set, however. In particular, firmware upgrade may be attempted in the FAULT state.

Note: Only transitions from the ON state to the error states are depicted in the above diagram. Transitions from other states are also possible, but this is not shown in the diagram for clarity of presentation.

2.1.2. Channel and Channel Group State Machine

This section applies to both channels and channel groups. Each channel can be in one of several possible states. The channel state machine is shown in the following figure.

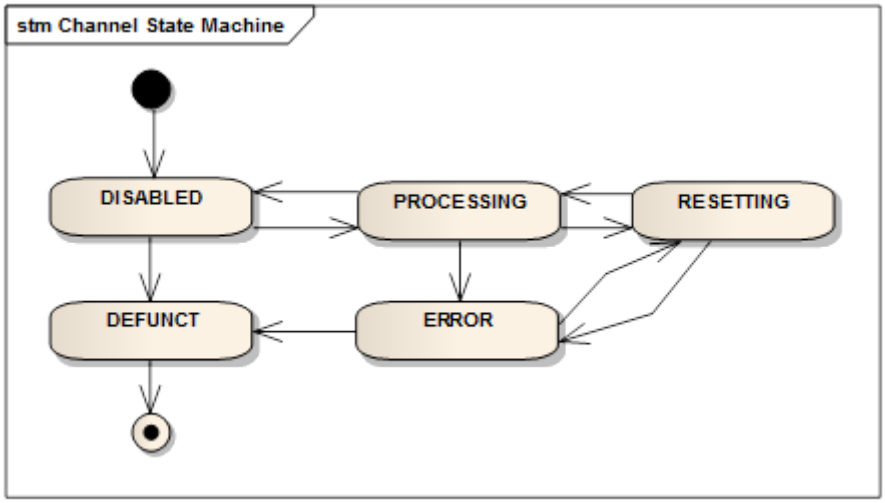

Figure 4: State machine of a channel.

A channel starts in the DISABLED state, where it is disabled and will not perform any operations.

Note: Channel can only be configured while it is in the DISABLED state and the device is **not** in the INIT state.

The channel state transitions to DISABLED either when the device state becomes OFF (cf. section 2.1.1) or when the user sends a DISABLE message to the channel.

Once the channel has been configured and the device is in the ON state, it can be transitioned to the PROCESSING state. Now it is ready to start performing the operations it is configured to do (e.g., acquire data once trigger condition is met).

Note: By using the ENABLED property (see the next section), a channel may be transitioned to PROCESSING state automatically when the channel group is transitioned.

The channel can be reset to the default configuration by sending it the RESET message. If a hardware-level reset is supported for that individual channel, it is also performed.

The ERROR and FAULT states model error conditions. The ERROR state indicates a transient error condition which can be recovered either by sending a RESET message or an ON/OFF message

sequence to the channel. The FAULT state indicates a fatal error which can only be fixed by servicing the hardware.

If a device is in the FAULT state, all channels are also in a FAULT state. The reverse is not true: if a single channel is in a FAULT, and other channels are unaffected, the state of the device remains unaffected.

2.1.3. Enabled property

[v2.3] This feature's purpose is to let a channel group only process channels which are ENABLED. It can be thought of as an "Active" property, where only active channels are processed.

ENABLED means that the object is configured and ready to be used. For a device it means that device will be switched ON when the IOC starts. For other objects, such as channels, it means that they will transition to PROCESSING as required.

DISABLED means that the object hasn't been configured yet and it is not ready to be started on application start.

Note: DISABLED is not the same as the state DISABLED.

This property is implemented through an EPICS record. It allows specification of channels which are already configured and save this indication between application restarts. By introducing this, NDS allows use of the archiver, SDD [RD7] or any other tool to save and automatically restore device configuration.

2.2. Architecture

Nominal Device Support is implemented as a port driver for the *asynDriver* [RD5].

NDS provides:

- Definition and implementation of the device model and its behavior as described in the previous section.
- Mechanism for registration of a port with the *asynDriver*.
- Implementation of:
  - asynInt32,
  - asynFloat64,
  - asynUInt32Digital,
  - asynOctet,
  - asynGenericPointer,
  - asynFloat32Array,
  - asynInt8Array,

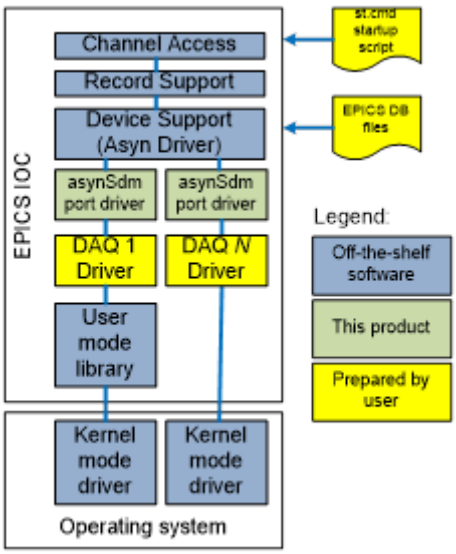

Figure 5: Architecture of the NDS port driver.

  - asynInt16Array and
  - asynInt32Array interfaces.

  The calls on these interfaces are interpreted and mapped in function-oriented calls on the device-specific *NDS* driver as defined in requirements specification. This also enforces the interface towards the EPICS database.
- API and its implementation through which a device-specific driver provides data (e.g., acquired samples) and states (e.g., error conditions).
- Support for management of threads (e.g., for polling).
- Implementation of operations in software when they are not supported by hardware.
- Definition of IOC shell functions for initialization of the device driver.

The device instance and its channel groups has one *asynDriver* port each. By default, the ports are configured for asynchronous mode of operation. In this mode, all device-specific functions are allowed to block without interfering with EPICS database processing.

3. Use Cases

This section provides an overview of the kind of devices which can be standardized with NDS. All devices expose similiar functionality through the same interface, which increases usability of the system built of these components.

3.1. Data Acquisition Device Support

Data acquisition device support consists from the entities which are organized in structure as described above (see chapter 2). Depending on device functionality channels entities could include functionality described in next section: Filtering, Conversion, Streaming, Triggering.

3.2. Image Acquisition Device Support

Image acquisition device support consists from the entities which are organized in structure as described above (see chapter 2). Device entity is representing camera or image grabber device. Channel represents 1 image channel. Depending on device functionality channels entities could include functionality described in next section: Streaming, Triggering.

3.3. Timing Device Support

Timing device support consists from the entities which are organized in structure as described above (see chapter 2). Timing terminals are represented by the child of the nds::Channel class which is called nds::Terminal. Timing device could have multiple virtual events which are assigned to specific timing terminal. Virtual Event could be organized in groups (through ChannelGroup entity).

Timing device entity is responsible for global functions of timing board: synchronization, health management and device state machine.

3.4. Health Management Device Support

Health management device consists from the entities which are organized in structure as described above (see chapter 2). Depending on health management level Device could represent Chassis, CPU, Cubicle, System. At the same time channel might represent: CPU, Memory,

HardDisc, Ethernet interface, different boards and so on. In the Health Management System channels represents different components of this system.

3.5. DAQ System

DAQ system combines functionality of the Timing and Data Acquisition devices and consist of at least 2 entities of such devices. DAQ System is represented by a single EPICS IOC.

Timing and DAQ device support for required devices must be developed as a separate modules and then combined to the DAQ system.

As far device instances are situated inside one IOC they can communicate each other through NDS internals. In general case standard PV handlers could be used as an interface between devices for internal communication. Internal communication means that devices communicates through C++ function calls without passing data through EPICS. Also NDS defines special purpose interfaces for internal communication.

E.g. Timing terminal exposes functionality to obtain latest timestamp. Common nds::Channel provides interface to specify time stamping source and default implementation of timestamp obtaining. This mechanics are used by analog/digital input channel to timestamp acquired data.

4. EPICS IOC Shell and Database

4.1. Device Initialization

The EPICS device support developer provides a device-specific NDS driver, which represents an abstract device which is not instantiated. Instantiation of a specific instance of the device must be done from the `st.cmd` file.

Device instantiation starts with the IOC shell command:

```
ndsCreateDevice("type", "device", "[name1=value1[,name2=value2,...]]")
```

This command **must** appear before the `iocInit` command. It specifies that an *NDS* device of the given *type* is to be instantiated, and that its name is *device*. *device* also corresponds to the name of the *asynDriver* port assigned to the device. Instantiation parameters are given with a comma-separated list of name-value pairs.

The *type* is a unique string defined by the device-specific driver. Consult the documentation of the driver to determine what it is.

If *type* refers to a device type that is not registered, or if *device* is already in use, or if *name* does not refer to a parameter name supported by the device driver, an error message is written to the IOC log.

Example: the following `st.cmd` snippet would define a device of type *ndsExample* with name *ndsExample-0* and the POSIX file descriptor of the device driver is `/tmp/nds0`:

```
ndsCreateDevice("ndsExample", "ndsExample-0", "FILE=/tmp/nds0")
```

4.1.1. Initialization Parameters

In addition to the list of configuration parameters, the following initialization parameters, for the st.cmd file, are also standardized:

Initialization parameter	Description
`FILE`	Provides the path to the file descriptor for *NDS* device drivers that communicate with their low-level device drivers via device nodes in the file-system.
`SN`	Identify the associated *NDS* device by its supplier's serial number. The *NDS* device driver will need to enumerate all the devices to find the one with the requested serial number.

Initialization parameter	Description
IDX	Logical number of the device in the operating system. (Unique controller index within IOC.)
CHASSIS	Logical number of the chassis. By convention, the top one in the rack is 0 (when applicable).

Table 2: List of standard initialization parameters.

4.2. NDS IOC Debug Functions

ndsListDrivers Prints a list of registered NDS drivers ndsListDevices

Prints a list of registered device instances ndsTraceLevels

Prints a list of trace levels ndsSetTraceLevel [traceLevel]

Sets trace level ndsListTasks

Prints a list of NDS tasks ndsStartTask [task]

Manual starting of selected NDS task ndsCancelTask [task]

Manual cancelation of selected NDS task ndsPrintStructure [device name]

Prints device structure

5. PV Record naming

IMPORTANT

This section describes the *NDS* internal PV record naming. Note that at ITER all NDS PVs must be aliased with the names corresponding to the ITER I&C naming conventions ([RD8] section 4.3, [RD9]). For more detailed information on the naming and functional breakdown structure of diagnostic I&Cs refer also to [RD10] and [RD11].

Aliasing is achieved through the use of EPICS record aliasing features [RD3].

The name of EPICS database records associated with *NDS* devices have the general form:

prefix[-*channelid*][-*suffix*]

prefix Any site-specific naming convention prefix for device, channel group or channel.

At ITER the *prefix* would be `$(CBS1)-$(CBS2)-HWCF:`*devtype*`-$(BOARDTYPEIDX)`, where *devtype* is the device specific identifier.

channelid Channel ID depends on what kind of object it is:

- For a device it is an empty string.
- For a channel it is the *channel name prefix* immediately followed by the *channel number*.
- For a channel group it is the *channel group name*.

At ITER the *channelid* would be `$(CHANNEL_NAME)$(CHIDX)` for channels, or just `$(CHANNEL_NAME)` for channel groups.

suffix Suffix is used to specify which functionality to access. The list of common functions is given in the reference in the Developer's Guide.

Substitution macro	Description
`BOARDTYPEIDX`	Device index, unique for a given device/board type
`CBS1`	CBS level 1 identifier.
`CBS2`	CBS level 2 identifier.
`CHANNEL_NAME`	[channel, channel group] The channel name prefix or the channel group name. Some NDS devices do not expose this macro because the value is fixed and cannot be changed.
`CHIDX`	[channel] Channel index, unique for a given channel group.

Substitution macro	Description
MODULEIDX	Module index, unique for a given device/board type. Some NDS devices do not expose this macro because the value is fixed and cannot be changed.
PREFIXBOARDTYPE	Device/board specific type prefix. Some NDS devices do not expose this macro because the value is fixed and cannot be changed.

Table 4: List of standardized substitution macros.

In addition, the template may choose to set initial values for configuration parameters of the device's or channel's records. In this case, the name of the macro substitution parameter is equal to the function suffix of the record, as given in Developer's guide.

PREFIXBOARDTYPE, MODULEIDX and CHANNEL_NAME are used together to identify the *asynDriver* port name ($(PREFIXBOARDTYPE)$(MODULEIDX) for devices and $(PREFIXBOARDTYPE)$(MODULEIDX).$(CHANNEL_NAME) for channel groups and channels). *asynDriver* port name of the device should also correspond to the name of the device instance, as used in the call to ndsCreateDevice IOC shell function when the device was created.

The *asynDriver* address for channels corresponds to CHIDX. For channel groups it is -1, and for devices it is 0.

The database template can be loaded with the standard IOC shell command dbLoadRecords, for example:

```
dbLoadRecords("/opt/codac/epics/db/ndsExampleDevice.template",
  "BOARDTYPEIDX=0,
  CBS1=D1,
  CBS2=I9,
  PREFIXBOARDTYPE=example,
  MODULEIDX=1")

dbLoadRecords("/opt/codac/epics/db/ndsExampleDeviceChannel.template",
  "BOARDTYPEIDX=0,
  CBS1=D1,
  CBS2=I9,
  PREFIXBOARDTYPE=example,
  MODULEIDX=1,
  CHANNEL_NAME=raw,
  CHIDX=0")
```

Alternatively, the macros can also be provided through the use of EPICS substitution files. See [RD3] for more information.

6. Clocks and Triggering

Data acquisition starts (or a camera frame is grabbed) when a trigger condition is met. *NDS* models two aspects of triggering:

- when the first trigger occurs
- recurrence of the trigger condition.

The trigger condition can be based on an event, i.e., trigger when some channel (or combination of channels) reaches a certain value.

6.1. Time Trigger [removed]

This feature was removed. Use hardware timing triggers instead.

6.2. Event Trigger

An event trigger condition occurs when the input on a channel or a set of channels reaches a certain value.

To trigger on an event, the `TRGC` record is used to specify the trigger condition. `TRGC` is a space-separated list of expressions. Expression can have the following formats:

- *channel-id*: true when the digital input channel *channel-id* is in the "one" state.
- !*channel-id*: true when the digital input channel *channel-id* is in the "zero" state.
- *channel-id*>*level*: true when the analog input channel *channel-id* is below value *level*.
- *channel-id*<*level*: true when the analog input channel *channel-id* is above value *level*.
- *channel-id**level*: true only on transition through selected *level,* [rising edge].
- *channel-id*/*level*: true only on transition through selected level *level,* [falling edge].

Here, *level* is the value of the channel in engineering units (i.e., after conversion). *channel-id* is the ID of the channel (see Previous chapter for the channel naming convention).

In addition to the standard channel types used for the channel ID, type TCN can also be used which is an alias for a digital signal coming from the timing board that is synchronized with the distributed time.

The trigger condition is met when all of the expressions match (logical `AND`). For example, trigger condition:

```
TCN3 !DI2 AI0>5 AI1<7
```

occurs when the fourth digital input wired with the timing board (TCN3) is 1, digital input 2 is 0, analog input 0 exceeds 5 and analog input 1 is below 7.

Note: 'AND' and 'OR' conditions on trigger can generate meta-stable states.

6.3. Software triggering

Software (manual) trigger event occurs when the TRG record is processed. This can be achieved by writing 1 to PROC field of the record.

6.4. Accurate Time-stamping

To ensure that the timestamp of the acquired data is set to match the time when the trigger condition was met, set the TSE field of the ID of the event that will be picked-up by EPICS general time and that provides the time of the trigger.

6.5. Continuous vs.One-shot Trigger

Configuration parameter TRGR specifies whether the trigger repeats, and if so, how many times.

- Any non-negative value specifies the number of remaining repetitions. It decreases by one whenever a trigger occurs until 0 is reached.
- -1 means continuously and is the default value.
- 0 means don't trigger.

6.6. Trigger Delay

A trigger can be configured to occur a given delay after the trigger condition is met. The delay is configured with the TRGD record, which specifies the time in seconds, expressed as a double value.

It is also possible to set a negative value for the trigger delay (*pre-trigger*). However, a negative value for the trigger delay requires that the data is buffered and is therefore limited by the amount of available RAM.

6.7. Lack of Hardware Support for Triggers

Not all devices support all the triggering functions described in the previous sections. If a function is not supported by the device, software implementation in *NDS* will be used instead. This implies that continuous sampling is employed, leading to a high utilization of resources. Also, time accuracy of the trigger is of the order of the sampling period (defined with the CLKF parameter), which could be significantly less than what would be achievable in hardware.

Whenever one from trigger's configuration parameters `TRGR`, `TRGC` or `TRGD` are set to a value that is not supported by hardware, the parameter is set to MINOR alarm state, and a warning is logged in the IOC's log.

7. Messages

Messages are a mechanism to send commands to the driver and control it. It is address either to the device, the channel group or an individual channel. The driver will then do the action that is required – including doing a transaction with the device, if needed. This is implemented through the records `MSGS` (send a message) and `MSGR` (receive a message): an external process writes a string to the `MSGS` record to send a message and read the `MSGR` to see any responses.

Both records are of the type waveform and limited in length to 255 characters. The format of a message consists of the message type, followed by a space and then a comma-separated list of name-value pairs supplying the parameters of the message:

Note: It must be a standard NULL terminated c-string.

message-type[$name_1$=$value_1$[,$name_2$=$value_2$...]]

The initial processing of the message is done by *NDS*. Depending on the selected message type, *NDS* passes processing to registered handlers. If it is required, message processing could be done in hardware.

Request-response communication is also possible. To correlate a response to a request, a special parameter, named `ID`, can be used to identify the request, and the device/channel will also put this parameter into the response. For example, if the user sends a message to the device like this:

```
$ caput -S DAQ-MSGS "TEST ID=123,QUICK=NO,VERBOSE=YES"
Old: DAQ-MSGS HELLO
New: DAQ-MSGS TEST ID=123,QUICK=NO,VERBOSE=YES
```

Monitoring the response could result in something like:

```
$ camonitor -S DAQ-MSGR
DAQ-MSGR 2012-02-13 12:03:05.182332 TEST
ID=123,TEST=Device present?,RESULT=
DAQ-MSGR 2012Par-02-13 12:12:04.241258 INIT
DAQ-MSGR 2012-02-13 12:12:04.423246 RESET
DAQ-MSGR 2012-02-13 12:12:04.653452 INIT
DAQ-MSGR 2012-02-13 12:12:04.872345 ON
```

7.1. Predefined Message Types

The following pre-defined message types are generic:

LIST Returns a list of registered message types. RESET

Request to perform a reset of the object. FWUP

Update firmware.

Request parameters:

URL The URL of the meta-information file.

Response parameters:

TEXT Completion status of the firmware update provided as text. CODE

Numeric code identifying firmware completion status. MODULE

Identification of the module to whose update the message refers. Not required if the whole device is being updated.

Possible completion statuses are (CODE and TEXT pairs):

0: Firmware update completed.

1: Firmware update in progress.

2: No matching firmware found.

3: Firmware update failed.

4: Firmware image '%s' couldn't be downloaded.

5: Firmware meta-file '%s' not found.

6: Firmware checksum didn't match.

The following pre-defined message types are device specific:

ON Request device to transition to `ON`. OFF

Request device to transition to `OFF`. TEST

Execute a self-test of the device.

Request parameters:

QUICK Whether to perform a quick self-test (**QUICK=YES**) or a full test (**QUICK=NO**, which is the default). The quick test may be incomplete, but is guaranteed to finish within one second. VERBOSE

Whether to report intermediate steps of the test (**VERBOSE=YES**) or just the final result (**VERBOSE=NO**, which is the default).

Response parameters:

TEST Identification of the test. Omitted if this is the final result of the test. TEXT

Textual description of the test result. CODE

Numeric code identifying the test result. 0 means all is OK, and other values indicate problems.

The following pre-defined message types are channel or channel group specific:

START Request channel or channel group to start processing. STOP

Request channel or channel group to stop processing. SET_IMG_FMT

Set format for the channel's image.

Request parameters:

IDX the zero-based index of the format, as returned by **LIST_IMG_FMTS**.

Response parameters:

TEXT Textual description of the error. CODE

Numeric code identifying the result. 0 means all is OK, and other values indicate problems.

LIST_IMG_FMTS list all image formats.

Request parameters:

ID request identification is required to allow associating responses (which can be many) with the request.

Response parameters:

ID request identification. IDX

the zero-based index of the format. N

total number of supported formats. WDTH

width of the image. HGHT

height of the image. RES

resolution (bits per sample). SPP

samples-per-pixel. IMGT

type of image's raw format (see IMGT record in Developer's guide).

7.2. User Defined Message Types

There can also be user-defined, device-specific, messages.

8. Conversion between Raw and Engineering Units

The NDS driver supports conversion of acquired raw data from analog input channels into engineering units. Conversion is performed on:

- an input channel's waveform record, `${CHANNEL}`
- an input channel's sample read-out, `${CHANNEL}-IN`
- an output channel's waveform record, `${CHANNEL}`
- an output channel's single output value, `${CHANNEL}-OUT`

Conversion during analog input is performed as

$$\mathrm{VAL} = f(\mathrm{RVAL})$$

where VAL is the value in engineering units, RVAL is the acquired raw value, and f is a function that can be broken down in a number of segments, each of which is a third-order polynomial[1]. More precisely, for every segment i where x is between its bounds x_i and x_{i+1}, $f(x)$ can be expressed as:

$$f(x) = a_{i,3}(x - x_i)^3 + a_{i,2}(x - x_i)^2 + a_{i,1}(x - x_i) + a_{i,0}$$

Similarly, for analog output conversion, the function

$$\mathrm{RVAL} = g(\mathrm{VAL})$$

is used, which can also be broken down into segments of third-order polynomials.

Conversion is configured with the `${CHANNEL}-CVT` record. This record is a waveform (array) of double values, grouped in tuples of five. The elements of each tuple are:

- element `[i*5 + 0]`: start of the segment, x_i
- element `[i*5 + 1]`: polynomial coefficient a_0
- element `[i*5 + 2]`: polynomial coefficient a_1
- element `[i*5 + 3]`: polynomial coefficient a_2
- element `[i*5 + 4]`: polynomial coefficient a_3

If records `${CHANNEL}-CVT` is not set, standard EPICS' fields `LINR`, `EGUF` and `EGUL` are used from `${CHANNEL}-IN` (input) and `${CHANNEL}-OUT` (output). In that case, the conversion is linear (`LINR` field has value `LINEAR`).

[1] A third order polynomial was chosen because it is the lowest order polynomial whose first derivative can be made continuous even at segment break-points.

9. Signal Filtering

The NDS supports 3 types of filtering:

- Simple filtering
- Polynomial filter definition
- Smoothing filtering

Type of the filter could be chosen through `${CHANNEL}-FT`

If the requested filter is implemented in hardware, the filter parameters are used to configure the hardware. Otherwise, *NDS* implements the filter in software. In this case, the `FILT` record is put in the `MINOR` alarm state and a warning is output to the IOC error log.

9.1. Polynomial and Smoothing Filtering

The *NDS* port driver supports infinite impulse response (IIR) filters for filtering analog input channels and pixels in camera frames. Filtering is performed through the channel's waveform record, `${CHANNEL}`, and through the record for individual sample read-out, `${CHANNEL}-IN`.

General form of an IIR filter is:

$$y[n] = -\sum_{K=1}^{N} a_k y[n-k] + \sum_{K=0}^{M} b_k x[n-k]$$

Here, $y[n]$ is the n-th output of the filter, and $x[n]$ is the n-th input into the filter. In the special case where N=0, the filter is a *finite impulse response* filter (FIR).

The coefficients a_k and b_k are specified with the `${CHANNEL}-FILT` waveform record. The `FILT` record contains elements of type double, and its structure is:

- element [0]: number of coefficients a (N in the above equation)
- element [1]: number of coefficients b (M +1 in the above equation)
- elements [2] through [N+1]: coefficients a_1 through a_N
- elements [N+2] through [N+M+2]: coefficients b_0 through b_M

The *NDS* port driver allows to use smoothing filter, which is equivalent to the default EPICS smoothing filter. `SMOO` field of the channel's `IN` record defines coefficient. It is an IIR filter with a_0=`SMOO` and b_0=(1-`SMOO`). In the special case when `SMOO` is 0, there is no filtering of the signal (i.e., $y[n] = x[n]$).

If one were to compute $y[0]$, then $y[-N]$ thru $y[-1]$ and $x[-M]$ thru $x[0]$ would need to be known – i.e., the filter needs to be initialized. The *NDS* software filter implementation performs the initialization by setting all unknown parameters to $x[0]$. By comparison, the EPICS analog input record implementation rather skips the evaluation of $y[0]$ and starts with $y[1]$ (setting $y[0]$ to $x[0]$). Behavior of a hardware implementation depends on the particular device.

9.2. Simple Signal Filtering

Simple filtering function could be defined by the following PVs: `LOWP`, `HGHP`. PVs are available per each channel or channel group. Implementation depends on the device.

LOWP low pass parameter of the filter specifies high cut-off frequency. It means that frequencies above this value are attenuated at least on -3 db. (see figure 1.a)

HGHP high pass parameter of the filter specifies low cut-off frequency. It means that frequencies below this value are attenuated at least on -3 db. (see figure 1.b)

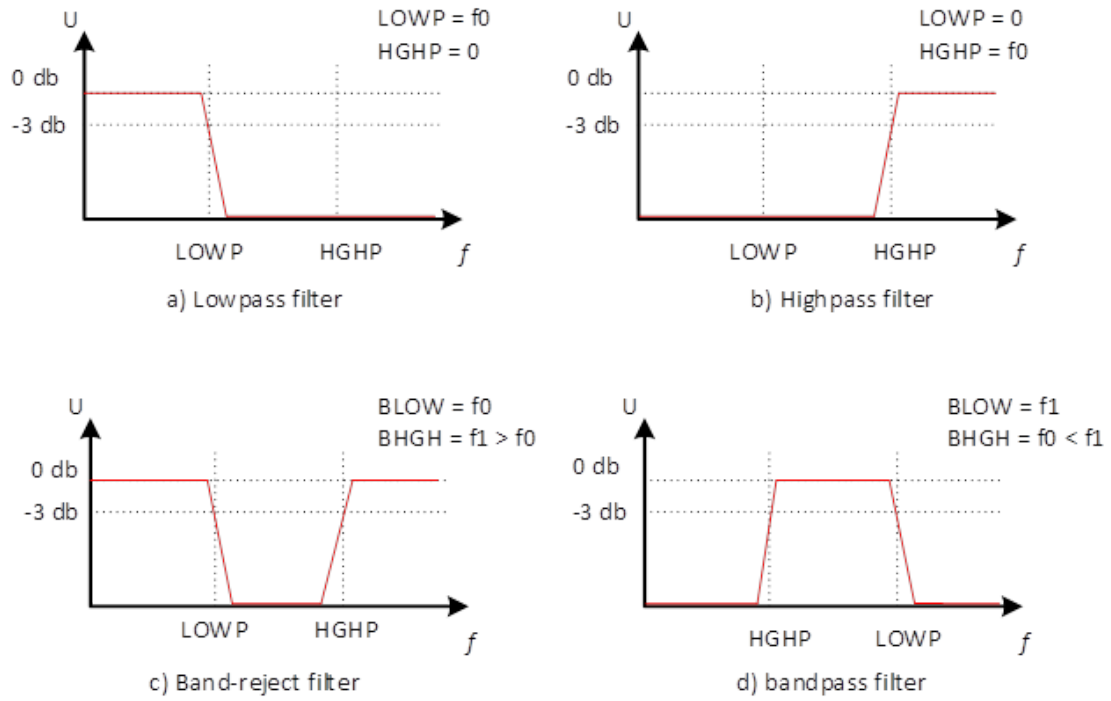

Figure 6

To check current status of the filtering read back PVs could be used: `rLOWP` and `rHGHP`.

Note: Readback PV names are site-specific.

10. Fourier Transform

Analog input channels can perform the Fourier transform and is controlled with four parameters[1]:

FFTN size of the frame for the Fourier transform (number of samples). FFTO
number of samples that overlap between consecutive frames. FFTW
windowing function. FFTS
smoothing factor for averaging the Fourier transform.

The result of the Fourier transform is available in the FFT record, which is a waveform array of double values.

A Fourier transform is in effect if the size of the frame, FFTN, is greater than 0. It is recommended that FFTN is a power of 2 so that the fast Fourier transform algorithm can be used to improve efficiency.

The value of the FFT record is computed as follows:

1. A frame consisting of FFTN samples is prepared. The first frame is available when FFTN samples are acquired after the start of the acquisition was triggered. Let **frame**$[i]$ denote a frame, where $0 \leq i <$ FFTN.
2. The frame is multiplied by the selected window function:

$$\textbf{frame'}[i] = \textbf{frame}[i] \cdot \textbf{window}[i]$$

3. The Fourier transform is computed. The result is a waveform of double values, say **fft**$[i]$. This array, too, has FFTN elements.
4. If this is the first iteration, then the FFT record is the **fft**$[i]$ array. Otherwise, a new value of the FFT record is computed as:

$$\textbf{FFT}[i] = \text{FFTS} \cdot \textbf{FFT}[i] + (1 - \text{FFTS}) \cdot \textbf{fft}[i]$$

5. The frame for the next iteration of the Fourier transform is prepared with subsequent samples. The overlap parameter, FFTO, determines the first sample that goes in the next frame. If there is no overlap, **frame**[0] is **sample**[FFTN]. Otherwise, **frame**[0] is **sample**[FFTN − FFTO], **frame**[1] is **sample**[FFTN − FFTO + 1], etc.

When supported in hardware, the device is configured to perform the Fourier transform. If not supported by hardware, or if the hardware does not support the requested features, NDS implements the transform in software. In this case, the SEVR field of result record is set to the MINOR alarm state and a warning message is logged in the IOC log.

[1] The names of these records imply that the fast Fourier transform is always used, which may not necessarily be the case. However, the abbreviation FFT is so commonly used that it has been chosen for the purpose of naming these records anyway.

11. Decimation

Decimation refers to effectively reducing the sampling rate by keeping only every DECF-th sample, where DECF is a configurable parameter (*decimation factor*). Configuration parameter DECO (*decimation offset*) specifies how many samples are skipped from beginning of the sampling period.

Decimation is applicable both for input channels, where it reduces sampling frequency, as well as output channels, where it can manipulate the frequency of reproduction of the waveform that is to be generated.

Suppose that waveform **WF'** were collected without decimation. Then, the decimation process results in a waveform **WF**, such that

$$\textbf{WF}[i] = \textbf{WF'}[i \cdot \texttt{DECF} + \texttt{DECO}]$$

For the output channel, if waveform **WF** is requested to be played out, then **WF'** is played out such that:

$$\textbf{WF'}[i] = \textbf{WF}[i \cdot \texttt{DECF} + \texttt{DECO}]$$

Example: Let WF contain a sine signal of frequency 100 Hz sampled at 2 kHz (i.e., 20 samples per period). If decimation factor (DECF) were set to 10 and offset (DECO) were set to 5, sine wave advanced by phase $\pi/2$ (5/20 of a period) and frequency of 1000 Hz would actually be generated. In addition, GAIN parameter can be used to adjust the amplitude.

12. Function Generator

For analog output channels, instead of setting the **WF** record that specifies the samples to output, a spline can be specified in the SPLN record.

Splines allow for description of piecewise-cubic functions. Specification of the spline is the same as specification of the conversion function between raw and engineering units described in section 7.1, except that x_i corresponds to the starting time of the segment relative to the trigger.

Example: Let the SPLN record be set to the array:

```
0 0 0 0 0 # from time 0, f(x) = 0 * (x-0)^3 + 0 * (x-0)^2 + 0 * (x-0) + 0
1 1 0 0 0 # from time 1, f(x) = 0 * (x-1)^3 + 0 * (x-1)^2 + 0 * (x-1) + 1
2 1 1 0 0 # from time 2, f(x) = 0 * (x-2)^3 + 0 * (x-2)^2 + 1 * (x-2) + 1
3 2 0 1 0 # from time 3, f(x) = 0 * (x-3)^3 + 1 * (x-3)^2 + 0 * (x-3) + 2
4 3 0 0 1 # from time 4, f(x) = 1 * (x-4)^3 + 0 * (x-4)^2 + 0 * (x-4) + 3
5 4 0 0 0 # from time 5, f(x) = 0 * (x-5)^3 + 0 * (x-5)^2 + 0 * (x-5) + 4
```

The amplitude-versus-time function generated by the output channel would be:

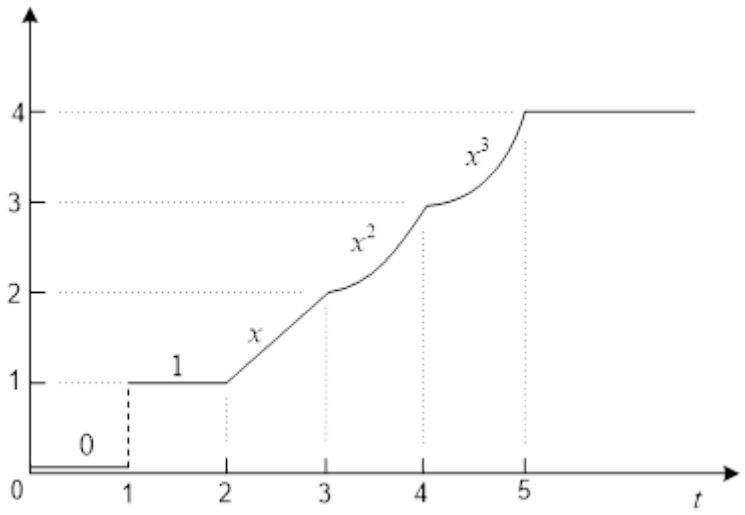

Figure 7: Spline output example

13. Firmware Update

Some devices allow their firmware to be updated.

To decrease the probability of an inadvertent firmware update, it is necessary to supply meta-information to every firmware image. The meta-information specifies what hardware can be updated, and which firmware must be installed.

The meta-information is stored in an XML file whose schema is shown in Figure.

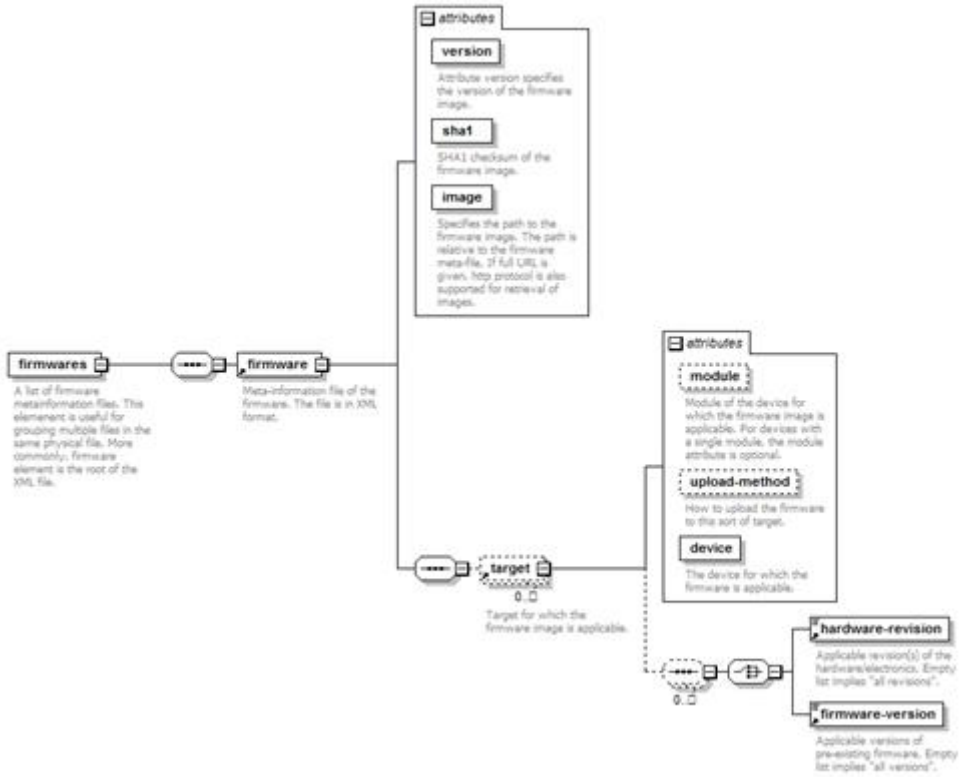

Figure 8: Schema of the firmware meta-information file.

For example, the firmware meta-information could look as follows:

```
<!--
 ! Meta-information file of the firmware. The file is in XML format.
 !
 ! The root element s image attribute specifies the path to the firmware
 ! image. The path is relative to the firmware meta-file. If full URL
 ! is given, http protocol is also supported for retrieval of images.
 !
 ! Attribute version specifies the version of the firmware image.
 !-->
```

```
<firmware image="http://server/ni6529-fw-2.3.1.bin"
        version="2.3.1"
        xmlns="http://www.cosylab.com/NDS/FirmwareMetaInfo/2012" />
  <!--
   ! device and module refer to the device and its
   ! module(s) for which the firmware is applicable.
   ! For devices with a single module, the module
   ! attribute is optional.
   !-->
  <target device="NI6529" module="fpga-3">
    <!--
     ! Applicable revision(s) of the hardware/electronics.
     ! Empty list implies "all revisions".
     !-->
    <hardware-revision>rev0</hardware-revision>
    <hardware-revision>rev1</hardware-revision>
    <!--
     ! Applicable versions of pre-existing firmware.
     ! Empty list implies "all versions".
     !-->
    <firmware-version>2.2.0</firmware-version>
    <firmware-version>2.2.1</firmware-version>
    <firmware-version>2.2.2</firmware-version>
  </target>
</firmware>
```

To update the firmware of the device, first make sure that the firmware is available on the machine where the IOC runs, or use a HTTP URL that refers to the location of the firmware meta-information and the image file.

The recommended way to achieve this is by placing it on a network file system, e.g., /codac/-firmware, and pointing the CODAC_FW environment variable on the IOC machine to that directory.

To trigger the update, put the path to the firmware meta-information file in the device's FWUP record:

```
$ caput ${DEVICE}-FWUP '$CODAC_FW/fw-meta-info.txt'
```

Note: If a HTTP URL is used, ensure that it is accessible from the target machine (e.g., that the web proxy is properly configured).

Example:

Assume that the firmware meta-information and image are available. If on the IOC machine one were to list files in the $CODAC_FW directory, one would get:

```
$ ls -1 $CODAC_FW
```

```
ni6529-fw-2.3.1.bin
ni6529-fw-2.3.1.txt
```

To update the firmware, use the command:

```
$ caput ${DEVICE}-FWUP $CODAC_FW/ni6529-fw-2.3.1.txt
Old : ${DEVICE}-FWUP
New : ${DEVICE}-FWUP $CODAC_FW/ni6529-fw-2.3.1.txt
```

Observing the state of the device during update might look like this:

```
$ camonitor ${DEVICE}-STAT
${DEVICE}-STAT 2012-02-13 12:03:05.182332 FAULT
${DEVICE}-STAT 2012-02-13 12:12:04.241258 INIT
${DEVICE}-STAT 2012-02-13 12:12:04.423246 RESET
${DEVICE}-STAT 2012-02-13 12:12:04.653452 INIT
${DEVICE}-STAT 2012-02-13 12:12:04.872345 ON
```

Initially, the device was in the `FAULT` state, e.g., because the previous firmware was defective. Then the caput command was issued, and firmware started being updated, transitioning the board to the `INIT` state. When the update was complete, a soft reset of the board was performed (`RESET` state). The device then initialized, and finally was back to the `ON` state, ready for operation.

14. Streaming of Acquired Data

As EPICS Channel Access is not designed for low-latency or high-throughput communication, *NDS* devices also support more direct means to stream acquired data towards their intended targets.

Streaming is configured with the `STRM` configuration parameter, which holds a string value. If the parameter is an empty string, no streaming is performed. Otherwise, it is interpreted as the address for streaming. The address takes the form of an URL:

```
protocol://address[:port][/path[?param1=val1[&param2=…val2]]]
```

The following protocols are supported:

gvsp UDP packets formatted according to the GigE Vision's GVSP standard [RD6] are sent to IP *address* and *port*. Multicast IP addresses can also be used.

udp UDP packets formatted according to *NDS* UDP packet format (see section 14.1) are sent to IP *address* and *port*. Multicast IP addresses can also be used.

path can optionally be used to identify the stream.

Parameter `bin` specifies whether the acquired data should be sent in binary format (`true`/1, which is the default) or in human-readable ASCII format (`false`/0).

14.1. *NDS* UDP Packet Format

For streaming of acquired data, NDS users two types of frames, *info* and *data*:

14.1.1. Info frame format

Byte 0-3 Contains a flag idenfitying the frame as an info frame, `0xFFFFFFFF`.

Byte 4- ASCII-encoded text where each line gives a value of one configuration parameter. The name of the parameter is followed by a colon and its value. Values of parameters `STRM`, `CLKF`, `RES`, `WDTH`, `HGHT`, `SPP`, `CVT` and `FILT` are usually provided.

14.1.2. Data frame format

Byte 0-3 A 32-bit unsigned network-ordered sequential number of the frame. Starts with 0 and increments by 1 for each UDP packet sent. Can be used to detect packet loss.

Byte 4-11 Timestamp of the data. Two 32-bit values are provided. The first is the number of seconds since the UNIX epoch, and the second is the number of nanoseconds.

Byte 12- Number of bytes per sample depends on the RES parameter, and is aligned to one, two, four or eight bytes. For example, if RES is 1 (for digital input channels), each sample occupies one byte. If RES is 24 and no conversion is configured, each sample is four bytes.

15. Error Handling

Implementation of error handling depends on the manner in which the API is provided:

- If the user tries to set an invalid value to an output record, or a value that is not supported by the device, the alarm severity (SEVR field) of the record is set to MINOR and the alarm state (STAT) is set to WRITE.
  - If there is a record that conveys a read-back value corresponding to the output record, the read-back record indicates what value was actually applied. The actually applied value is device-specific, and it might be the nearest value that is supported, the nearest smaller value that is supported, no change, etc.
- When a message is sent that supplies invalid values of parameters, the CODE parameter of the response indicates the numeric code of the fault (0 indicates success), and TEXT parameter provides textual description of the failure.

In the IOC's error log, a warning message is logged to provide more detailed information.

16. Device Health Management

NDS provides standard set of error codes to report device status. Device status is reported through **STS** PV. The possible error codes can be seen in the table below.

Error code	Error string
0	No error
1	Initializing
2	Resetting
3	Hardware error
4	Brd unavailable
5	Static conf error
6	Dync conf error
8	FIFO overflow
9	FPGA not ready
10	Ref clk no lock
11	Buffer overflow

Table 5: Error codes.

17. *NDS* Plugins

NDS plugins allow extending the functionality of existing *NDS* devices in a generic and pluggable way. They can be attached to arbitrary *NDS* devices or even chained together to combine different functionality.

NDS plugins can either be used to perform additional processing on the *NDS* device data before the data gets processed by EPICS, or they can stream the data directly out of the *NDS* bypassing EPICS altogether. The later scenario is especially useful for implementing plugins for high-throughput data streaming such as scientific data archiving (e.g. ITER DAN) or synchronous low-latency data communication (e.g. ITER SDN).

NDS plugins are configured through EPICS and can consist of arbitrary number of *NDS* plugin channels. Each channel is mapped to a single data source, i.e., a stream of data produced and exposed by individual *NDS* device. Each *NDS* device already inherently exposes its channel buffers as attachable data sources (see section 8.3.1 of [RD1]). However, it may also be extended (requires code modification) to expose additional data sources (see section 8.3.2 of [RD1]).

17.1. Usage of *NDS* Plugins

Instantiation and configuration of *NDS* plugins is done in the same way as for ordinary *NDS* devices. In order to use a plugin in an IOC the plugin must first be created with the standard *NDS* device `st.cmd` instruction:

```
ndsCreateDevice("type", "device", "[name1=value1[,name2=value2,...]]")
```

EPICS configuration is normally done through the use of EPICS device and channel substitution files referring to the corresponding *NDS* plugin specific EPICS database template files. Channel substitution file consists of a list of *NDS* plugin channels. For each element in the list, one plugin channel will be instantiated and configured with the provided configuration parameters (substitution macros). The list of common *NDS* plugin device and channel substitution macros is provided in Table 6. Each plugin may additionally expect also other plugin specific parameters.

Substitution macro	Description
`BOARDTYPEIDX`	Plugin device index, unique for a given plugin type
`CBS1`	CBS level 1 identifier.

Substitution macro	Description
CBS2	CBS level 2 identifier.
CHIDX	[channel] Plugin channel index, unique for a given plugin device.
ENBL	Enabled flag specifying whether the plugin device or channel should be started automatically when the plugin is loaded.
MODULEIDX	Module index, corresponding to the index used with ndsCreateDevice.
SRC	[channel] Channel data source (see 17.1.1).

Table 6: List of common *NDS* plugin substitution macros.

17.1.1. Data Source

As already mentioned, each *NDS* plugin channel is assigned a data source.

Data source can be any array record registered on an arbitrary *NDS* device, channel group or channel. Data source is specified through the plugin channel's SRC PV (also configured in the plugin channel substitution file) using one of the following syntaxes:

- Device record: DEVICE[RECORD] (e.g pxi6368-0[DData])
- Channel group record: DEVICE-CG[RECORD] (e.g pxi6368-0.ai[CGData])
- Channel record: DEVICE-CGX[RECORD] (e.g pxi6368-0.ai0[CData])
- Channel buffer record: DEVICE-CGX (e.g. pxi6368-0.ai0)

Where:

- DEVICE is the name of *NDS* device (asyn port: DEVICE; asyn address: 0).
- DEVICE-CG is the name of *NDS* channel group belonging to the DEVICE device (asyn port: DEVICE-CG; asyn address: -1).
- DEVICE-CGX is the name of *NDS* channel belonging to the DEVICE-CG channel group (asyn port: DEVICE-CG; asyn address: X).
- RECORD is the name of an array record belonging to a device, channel group or channel. The record name corresponds to the reason name which was used to register the record in the *NDS* device implementation (see section 8.3.2 of [RD1] for more details).

17.1.2. EPICS Processing

As one of the primary purposes of *NDS* plugins is to allow processing or streaming of high-throughput or low-latency data, it might not be desirable to have the data also processed in EPICS.

When data source is a channel buffer on an existing *NDS* device, EPICS processing is enabled by default unless configured otherwise in the EPICS database. However, it may also be disabled at runtime through the use of `EPI` PV on the *NDS* device channel.

For example, if the *NDS* device channel used as the data source is named `DAQ-AI0`, EPICS processing can be disabled with the following command:

```
$ caput DAQ-AI0-EPI "DISABLED"
```

Similarly it can be re-enabled (e.g. for debugging purposes) with:

```
$ caput DAQ-AI0-EPI "ENABLED"
```

For other arbitrary data sources, the EPICS processing is managed by the *NDS* device implementation. For more information on how to control what data should get processed in EPICS and what should not, refer to section 8.3.2 of [RD1].

17.2. DAN Plugin

DAN plugin is used to stream a given *NDS* device data source to the ITER's Data Archiving Network (DAN) for the purpose of scientific data archiving.

In order to have DAN plugin integrated into an EPICS IOC the following is required:

- **EPICS IOC application built with *ndsDAN* device support**
  The *ndsDAN* device support's database definition file (DBD) and the *so* library have to be included in the EPICS application's `Makefile`: (along with the *NDS* core and any other *NDS* device supports already used by the IOC)

  ```
  <appname>_DBD += ndsDAN.dbd
  <appname>_LIBS += ndsDAN
  ```

- **EPICS IOC substitution files for *ndsDAN* device support**
  EPICS IOC substitution files for instantiating and configuring the *NDS* DAN plugin must be prepared. Substitution files should define the common *NDS* plugin substitution macros from Table 6 along with the DAN specific macros from Table 7.

Substitution macro	Description
`STRM`	[channel] DAN stream name.

Table 7: List of DAN plugin specific substitution macros.

Here is an example of `ndsDANDevice.substitution`:

```
file "/opt/codac/epics/db/ndsDANDevice.template" {
```

```
pattern
  { BOARDTYPEIDX, CBS1, CBS2, ENBL, MODULEIDX }
  { "0", "D1", "I9", "0", "0" }
}
```

And an example of `ndsDANChannel.substitution`:

```
file "/opt/codac/epics/db/ndsDANChannel.template" {
pattern
  { BOARDTYPEIDX, CBS1, CBS2, CHIDX, ENBL, MODULEIDX, SRC, STRM }
  { "0", "D1", "I9", "0", "0", "0", "7966R-0.raw0", "D1-I9-BCA0:DAN-0-CH0" }
  { "0", "D1", "I9", "1", "0", "0", "7966R-0.raw1", "D1-I9-BCA0:DAN-0-CH1" }
  { "0", "D1", "I9", "2", "0", "0", "7966R-0.raw2", "D1-I9-BCA0:DAN-0-CH2" }
  { "0", "D1", "I9", "3", "0", "0", "7966R-0.raw3", "D1-I9-BCA0:DAN-0-CH3" }
}
```

This example will independently stream the buffers of the four `7966R-0` device's channels (`7966R-0.raw{0..3}`) to the `D1-I9-BCA0:DAN-0-CH{0..3}` DAN streams.

- **EPICS IOC configured to start with *ndsDAN* device support**
  The *NDS* plugin device must be instantiated and the substitution files loaded from the EPICS IOC `st.cmd`. In ITER EPICS templates where `st.cmd` is split into multiple *cmd* files, the devices should be instantiated from `userPreDriverConf.cmd`:

  ```
  ndsCreateDevice "ndsDAN", "DAN0", ""
  ```

  And the database substitution files should be loaded from `dbToLoad.cmd`:

  ```
  dbLoadTemplate("ndsDANDevice.substitution")
  dbLoadTemplate("ndsDANChannel.substitution")
  ```

  This *cmd* file should also provide the list of aliases for the record names compliant with the ITER naming conventions (see the notice in chapter 5).
- **DAN configuration file**
  DAN additionally requires the DAN API configuration XML to be prepared and deployed in the DAN configuration directory. Refer to [RD12] for more information.

Once the IOC is built and running, the DAN plugin is started and managed through PVs in exactly the same way as any other *NDS* device. When data starts arriving from the attached data sources it will get automatically transmitted over the DAN network. In order to verify the content of the DAN stream, standard DAN tools like `dan_archiver_writer` and `dan-plot` may be used (see [RD12]).

17.3. SDN Plugin

SDN plugin is used to publish SDN topic payloads composed of one or many *NDS* device data sources on the ITER's Synchronous Databus Network (SDN) for the purpose of low-latency communication with other fast controllers.

In order to have SDN plugin integrated into an EPICS IOC the following is required:

- **EPICS IOC application built with *ndsSDN* device support**
  The *ndsSDN* device support's database definition file (DBD) and the *so* library have to be included in the EPICS application's `Makefile`: (along with the *NDS* core and any other *NDS* device supports already used by the IOC)

```
<appname>_DBD += ndsSDN.dbd
<appname>_LIBS += ndsSDN
```

- **EPICS IOC substitution files for *ndsSDN* device support**
  EPICS IOC substitution files for instantiating and configuring the *NDS* SDN plugin must be prepared. Substitution files should define the common *NDS* plugin substitution macros from Table 6 along with the SDN specific macros from Table 8.

Substitution macro	Description
`TOPIC`	[channel] SDN topic name.
`ATTR`	[channel] SDN topic attribute name. If an empty string, the data source attached to the plugin channel will provide a full SDN topic payload instead of individual SDN topic attribute value.
`PUBL`	[channel] SDN publish flag specifying whether the SDN topic should be published when the plugin channel data is processed.

Table 8: List of SDN plugin specific substitution macros.

Here is an example of `ndsSDNDevice.substitution`:

```
file "/opt/codac/epics/db/ndsSDNDevice.template" {
pattern
  { BOARDTYPEIDX, CBS1, CBS2, ENBL, MODULEIDX }
  { "0", "D1", "I9", "0", "0" }
}
```

And an example of `ndsSDNChannel.substitution`:

```
file "/opt/codac/epics/db/ndsSDNChannel.template" {
pattern
  { BOARDTYPEIDX,CBS1,CBS2,CHIDX,ENBL,MODULEIDX,SRC,TOPIC,ATTR,PUBL }
  { "0","D1","I9","0","0","0","7966R-0.pdm[PdmData]",
    "D1_I9_BCA0_DPMEFPMDTO_MRDLM","","1" }
}
```

Another example of `ndsSDNChannel.substitution`:

```
file "/opt/codac/epics/db/ndsSDNChannel.template" {
pattern
```

```
 { BOARDTYPEIDX,CBS1,CBS2,CHIDX,ENBL,MODULEIDX,SRC,TOPIC,ATTR,PUBL }
 { "0","D1","I9","0","0","0","7966R-0.pdm0[Timestamp]",
   "D1_I9_BCA0_DPMEFPMDTO_MRDLM","archive_timestamp","0" }
 { "0","D1","I9","1","0","0","7966R-0.pdm0[Id]",
   "D1_I9_BCA0_DPMEFPMDTO_MRDLM","id","0" }
 { "0","D1","I9","2","0","0","7966R-0.pdm0[ProductionState]",
   "D1_I9_BCA0_DPMEFPMDTO_MRDLM","productionState","0" }
 { "0","D1","I9","3","0","0","7966R-0.pdm0[QualityTag]",
   "D1_I9_BCA0_DPMEFPMDTO_MRDLM","qualityTag","0" }
 { "0","D1","I9","4","0","0","7966R-0.pdm0",
   "D1_I9_BCA0_DPMEFPMDTO_MRDLM","totalFusionPower","1" }
}
```

Both examples of `ndsSDNChannel.substitution` files will be streaming the same SDN topic (*D1_I9_BCA0_DPMEFPMDTO_MRDLM*), the difference is only how they will aggregate the topic payload data.

In the first example, SDN plugin will expect that the entire payload is already aggregated internally by the `7966R-0` device – respecting the prescribed SDN topic structure (structure is defined in `D1_I9_BCA0_DPMEFPMDTO_MRDLM.xml` file, shown later in this section). Every time the `PdmData` data source array is processed by the `7966R-0.pdm` channel group, the processed data will be copied to the topic payload, and the topic will be published on the SDN network (`PUBL` substitution macro tells whether to publish the topic every time the SDN channel is processed or not).

In the second example, SDN plugin will aggregate the payload data by itself. It will independently receive each topic attribute through the corresponding SDN plugin channel, whenever the configured data source gets processed (the `SRC` substitution macro). The last SDN channel is linked with the `7966R-0.pdm0` channel buffer data source, providing the `totalFusionPower` attribute value for the SDN topic. This channel is also configured for publishing (with the `PUBL` substituion macro), implying that when `7966R-0.pdm0` channel buffer is processed, the SDN plugin will also publish the topic payload on the SDN network. By that time it is very important that the rest of topic attributes have already been written to the payload, meaning that for this scenario, all SDN plugin channels need to be processed synchronously with the publishing channel being processed the last.

- **EPICS IOC configured to start with *ndsSDN* device support**

  The *NDS* plugin device must be instantiated and the substitution files loaded from the EPICS IOC `st.cmd`. In ITER EPICS templates where `st.cmd` is split into multiple *cmd* files, the devices should be instantiated from `userPreDriverConf.cmd`:

  ```
  ndsCreateDevice "ndsSDN", "SDN0", ""
  ```

  As with all *NDS* devices, the last argument is used to specify SDN device parameters (shown in Table 9).

Initialization parameter	Description
IFACE	[optional] Specifies the SDN interface name. By default it is obtained from the SDN_INTERFACE_NAME environment variable.

Table 9: List of SDN plugin specific initialization parameters.

The database substitution files should be loaded from dbToLoad.cmd:

```
dbLoadTemplate("ndsSDNDevice.substitution")
dbLoadTemplate("ndsSDNChannel1.substitution")
```

This *cmd* file should also provide the list of aliases for the record names compliant with the ITER naming conventions (see the notice in chapter 5).

- **SDN topic definition file**

  SDN topic definition file is used to define the structure of SDN topic. One such XML file for each SDN topic should be deployed in the SDN configuration directories of both the publishing and the subscribing side. These files are normally generated by SDD.

  Here is an example of D1_I9_BCA0_DPMEFPMDTO_MRDLM.xml (the definition file for D1_I9_BCA0_DPMEFPMDTO_MRDLM SDN topic):

```
<?xml version="1.0" encoding="UTF-8"?>
<topic name="D1_I9_BCA0_DPMEFPMDTO_MRDLM" version="2">
  <attribute rank="1" name="archive_timestamp" dataType="uint64_t" qualifier
      ="timestamp" unit="ns - nanosecond" />
  <attribute rank="2" name="id" dataType="uint64_t" qualifier="samplenb" />
  <attribute rank="3" name="productionState" dataType="uint32_t" />
  <attribute rank="4" name="qualityTag" dataType="uint32_t" />
  <attribute rank="5" name="totalFusionPower" dataType="float" unit="W -
      Watt" />
</topic>
```

  Beware that for the first example of ndsSDNChannel.substitution file, the total size and the structure (i.e. attribute ordering, data types and sizes) of the buffer data received from the 7966R-0.pdm[PmdData] data source must exactly match the size and structure of the SDN topic as defined in this SDN topic definition file.

  On the other hand, for the second example of ndsSDNChannel.substitution file, data provided by individual attribute-providing data sources must match the attribute sizes (sizeof(<dataType>)) and data types as defined in this SDN topic definition file.

Once the IOC is built and running, the SDN plugin is started and managed through PVs in exactly the same way as any other *NDS* device. When data starts arriving from the attached data sources it will get automatically transmitted over the SDN network. In order to verify the content of the SDN topic traffic, standard SDN tools like sdn-record, sdn-archive and sdn-plot may be used (see [RD13], [RD14]).