IDM UID
Q3WFFU
VERSION CREATED ON / VERSION / STATUS
23 Jan 2017 / 1.1 / Approved
EXTERNAL REFERENCE / VERSION

Report

D5.1 NDS Developer Manual

NDS documentation including enhanced plugin functionality

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 v1.0**	**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 Developer Manual (Q3WFFU)			
Version	***Latest Status***	***Issue Date***	***Description of Change***
v0.0	In Work	21 Dec 2016	
v1.0	Signed	29 Dec 2016	* Updated '3.7. EPICS Record Handling' (with emphasis on 3.7.1) according to the latest changes * Almost completely rewrote chapter '8. NDS Plugins' due to the poor content in the previous version and to update according to the latest changes * Fixed chapter/section numbering * Added table of contents * Fixed various minor issues throughout the document
v1.1	Approved	23 Jan 2017	Fixed the hyperlinks to the referenced documents, and corrected the descriptions of device (A1.1.7) and channel messages (A2.1.5).

Nominal Device Support, Developer's Manual

Developer'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 document covers the usage of the Nominal Device Support (NDS) which represents the C++ implementation of the Nominal Device Model (NDM) [RD1]. NDS is a library which generalizes EPICS [RD2] device support for generic DAQ and Timing devices. This document describes the sets of interfaces, solutions and best practices of device integration for EPICS provided by NDS and guides developer from template instantiation to complete solution.

This document is targeted to EPICS device support developers.

1.1. System Overview

The Nominal Device Support (NDS), the C++ implementation of the Nominal Device Model (NDM), generalizes EPICS device support for generic DAQ and Timing devices. The NDS library provides sets of interfaces, solutions and best practices of device integration for EPICS.

The NDS library is also provided with online documentation. It comes installed with CODAC Core System full development role and is accessible via the `codac-help` command. It can be found under the *Fast controllers: I/O* category of *CODAC Core System Documentation*.

1.2. Organization of the Manual

Topics of the manual are grouped by the following chapters:

- Common NDS practices
- State machine
- NDS Tools
- NDS Timing device
- DAQ system

1.3. Abbreviations

AI	Analog Input
AO	Analog Output
CODAC	Control, Data Access and Communication
DAQ	Data acquisition

FIR	Finite Impulse Response
GPIO	General Purpose Input/Output
IIR	Infinite Impulse Response
IOC	Input/Output Controller
SDD	Self-Description Data
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): Users'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] ITER Numbering System for Parts/Components (28QDBS)

[RD8] NI-RIO EPICS Module: RIOEM, Engineering Design Document (6WU76X)

[RD9] asynSIS8300-epics-driver - Programmer's Guide (6RK5ST)

[RD10] NI PXI-6259 EPICS Device Support User's Guide (3DEY52)

[RD11] Time Representation in CODAC software (7GDNSX v1.1)

[RD12] How to include a new I/O Moudle in SDD (A4WQDZ v2.0)

[RD13] CODAC Core System Application Development Manual (33T8LW v5.3)

[RD14] CODAC Core System Self-Description Data Editor User Manual (32Z4W2)

[RD15] NI Sync EPICS Driver User's Guide (33Q5TX)

2. Summary

Nominal Device Support (NDS) is the implementation of Nominal Device Model (NDM) [RD1]. It provides core for device integration to EPICS control system. It defines EPICS database [RD4] interfaces which intend to be common for the all devices of similar type.

NDS library provides specification for 2 device types:

- NDS DAQ device
- NDS Timing device

Core functionality is shared between these types. Description of function per specific device type could be found in Appendix A .

This document oriented to the EPICS device support developers and it is supposed that developers are familiar with NDM principles and ideas which could be found in [RD1].

In general device integration is consists from the following steps:

- Choosing device type to inherit: DAQ/Timing device. (Up to NDS v.2.3.5 Image acquisition is a part of the DAQ device).
- EPICS template initialization for that device (see section 3.1).
- Defining device structure and life cycle (see section 3.3).
- Implementation of device initialization (see section 3.5).
- Implementation of device specific functionality.
- If required device integration to DAQ system must be foreseen (see chapter 7).

NDS forces developer to use predefined state machines for a device and channels. There is a description of state machines and how to program them in chapter 4.

NDS provides set of tools to simplify and generalize threads handling, data polling, IOC's functions, and modules intercommunication (see chapter 5).

NDS timing device defines internal interface which allows to timestamp data without passing timestamp through EPICS (see section 6.2).

3. Common NDS practices

3.1. Getting Started

3.1.1. Using Code Generators

To facilitate learning and developing of NDS, it provides sets of EPICS templates:

- nds
- ndsTime

The first template, 'nds' is an example of the general DAQ device, while the second 'ndsTime' is an example of timing device.

In addition to example templates NDS provides empty application templates for EPICS device support module. Output of these templates is a system library. They suits to CODAC packaging requirements and should be used by CODAC developers. The templates are:

- nds
- ndsTime

ndsSupport template is for general DAQ device support modules. Second template 'ndsTime-Support' is for Timing device support modules. Templates include the skeleton for a full driver. The developer must then trim-down the database template files and the C code of the driver to match the functionalities of the particular device.

3.1.1.1. EPICS Code Generator

To generate a skeleton driver, use the NDS template for the makeBaseApp.pl script in a newly created directory:

```
$ mkdir ndsExample
$ cd ndsExample
$ makeBaseApp.pl -t nds ndsExample
```

Apart from the driver, this also creates an example EPICS application that uses the driver. To run the application, you can also create an IOC for it using NDS IOC boot template:

```
$ makeBaseApp.pl -i -t nds ndsExample
$ chmod a+x iocBoot/iocndsExample/st.cmd
```

3.1.1.2. CODAC Code Generator

The Maven tool provides a way to generate a module template for future development.

Note: Name of the EPICS support module should be started from 'epics' prefix, according CODAC best practices.

```
$ mvn iter:newunit -Dunit=m-epics-drv
[INFO] MODULE UNIT 'm-nds-example' CREATED
$ cd m-epics-drv
$ mvn iter:newapp -Dapp=drv -Dtype=nds
[INFO] EPICS APPLICATION 'drv' CREATED
$ mvn include -Dtype=module -Dname=pipe
[INFO] SPECIFIED 'MODULE' ELEMENT INCLUDED FOR PACKAGING
```

Then, edit pom.xml and ensure that the compiled device driver will be packaged for installation on development and runtime machines by specifying the following install package (for details on packaging see [RD13]):

```
   <packaging>
      ...
      <package>
         <include type="file"
                  source="main/epics/lib"
                  target="epics/modules/nds/lib" />
      </package>
```

The CODAC unit prepared in this way can then be packaged in RPMs:

```
$ mvn package
make -C ./configure install
...
+ exit 0
[INFO] Successfully packaged:
codac-core-5.3-epics-drv-5.3.0.v0.0a1-0.el6.x86_64.rpm
[INFO] PACKAGING COMPLETED (see for .rpm files under 'target' directory)
```

The following RPMs are obtained this way:

- `codac-core-5.3-epics-drv-5.3.0.v0.0a1-0.el6.x86_64.rpm`: RPM containing the shared library. This RPM should be installed on the development and target machines.

It is possible to check content of the package with following command:

```
$ rpm -qlp
target/codac-core-5.3-epics-pipe-5.3.0.v0.0a1-0.el6.x86_64.rpm
/opt/codac-5.3
/opt/codac-5.3/epics
/opt/codac-5.3/epics/modules
...
```

Note: CODAC templates don't include sample application. Sample application should be created separately as a normal CODAC module. CODAC requires explicit naming of sample application. Name should have '***-sample***' suffix.

E.g. for the sample application which uses library above, name should be: ***epics-drv-sample***:

```
$ mvn iter:newunit -Dunit=m-epics-drv-sample
$ cd m-epics-drv-sample
$ mvn iter:newapp -Dapp=drv-sample
$ mvn iter:newioc -Dioc=drv-sample -Dapp=drv-sample
```

You have to include drv module to sample application as a normal EPICS module.

3.1.2. Directory Structure of an NDS Device-specific Driver

The directory structure of a NDS device-specific driver is as follows:

```
configure # EPICS Makefiles
   ...
iocBoot # IOC boot configuration and scripts
   iocndsExample
      envSystem
      Makefile
      README
      st.cmd
   Makefile
Makefile # The main Makefile
ndsExampleApp # Driver with example application
   Db # EPICS database templates
      ndsExampleAnalogChannel.template # One file per channel
      ...
      ndsExampleImageChannel.template
   driver # Source code of the driver, it will produced a library
      Makefile # The driver's Makefile
      ndsExampleADIOChannel.cpp # ... A/D I/O channel
      ndsExampleADIOChannel.h
      ndsExampleDevice.cpp # ... device-level logic
      ndsExampleDevice.h
      ndsExample.h
      ndsExampleImageChannel.cpp # ... image acquisition channel
      ndsExampleImageChannel.h
   Makefile
   src # IOC application for testing
       Makefile
       ndsExampleMain.cpp
```

```
      ndsExample.substitutions
 x.txt
```

Depicted structure is valid for NDS example templates.

3.1.3. Minimal Driver

Code of a minimal device driver is the following:

```
/* The include files for the NDS-based device support. */

#include <ndsManager.h>
#include <ndsDevice.h>

/* Our device-specific driver must extend nds::Device. */
class ExampleDevice: public nds::Device {
public:
  /* The constructor. Delegates to the base 'class constructor and initializes
   * internal data structures. */
  ExampleDevice(const std::string& name): nds::Device(name) {
  }

  /* Construct objects associated with the device (channel groups, channels). */
  virtual ndsStatus createStructure(const char portName, const char params) {
      return asynSuccess;
  }
};

/* Register the device driver. */
nds::RegisterDriver<ExampleDevice> exampleDevice("ExampleDevice");
```

The RegisterDriver helper class in the last line registers the ExampleDevice class with the NDS port driver under the name *ExampleDevice*. So whenever a device of type ExampleDevice is created, an instance of the ExampleDevice class is constructed and its createStructure function is called to initialize it.

EPICS database templates should be provided as well. Templates for records described in Appendix A are standardized, so for a particular device driver only the subset that is supported should be used. The templates are available in the Db directory when generating an NDS device support project (see section 3.1.1).

Makefile to build the minimal driver is:

```
TOP = ..
include $(TOP)/configure/CONFIG
```

```
# The library to produce.
LIBRARY_IOC += ndsExample

# Dependencies of the library. EPICS base + NDS port driver.
ndsExample_LIBS += $(EPICS_BASE_IOC_LIBS)
ndsExample_LIBS += asyn
ndsExample_LIBS += ndsCPP

# List of source files.
ndsExample_SRCS += ndsExampleDevice.cpp

include $(TOP)/configure/RULES
```

3.1.4. Building an EPICS Application

To build an EPICS application, *asynDriver*, the *NDS* library and the *NDS* driver for a particular device must all be specified in the Makefile. This is achieved by adding the following libraries:

```
ndsExampleAppSupport_LIBS += asyn
ndsExampleAppSupport_LIBS += NDS
ndsExampleAppSupport_LIBS += ndsExample
```

and the following database definition files:

```
ndsExampleApp_DBD += asyn.dbd
ndsExampleApp_DBD += ndsExample.dbd
```

Here, *ndsExampleApp* is the name of the EPICS application and *ndsExample* is the name of the device-specific *NDS* driver.

3.2. NDS Tracing System

NDS provides the following macros for debugging, informational and error output:

```
#define NDS_CRT(args...) // Critical error, further processing impossible
#define NDS_ERR(args...) // Error
#define NDS_WRN(args...) // Warning
#define NDS_INF(args...) // Informational output
#define NDS_DBG(args...) // Debug output
#define NDS_TRC(args...) // Tracing output
#define NDS_STK(args...) // Marking entering and exiting from code section
#define NDS_CRT_CHECK (expression, args...)
```

`NDS_CRT` uses the `cantProceed` function to lock IOC execution. Before putting the IOC into this state it is useful to provide detailed description of the problem.

`NDS_CRT_CHECK (expression, args...)` checks an expression and if it is true then calls `NDS_CRT()`.

3.2.1. Managing Trace Level

Verbosity of driver logs can be changed by the following IOC function:

```
epics> ndsSetTraceLevel(traceLevel)
```

List of trace levels can be checked by:

```
epics> ndsTraceLevels
1: CRT
2: ERR
3: WRN
4: INF
5: DBG
6: TRC
7: STK
```

3.3. Device Structure

The device structure is described in Figure 1.

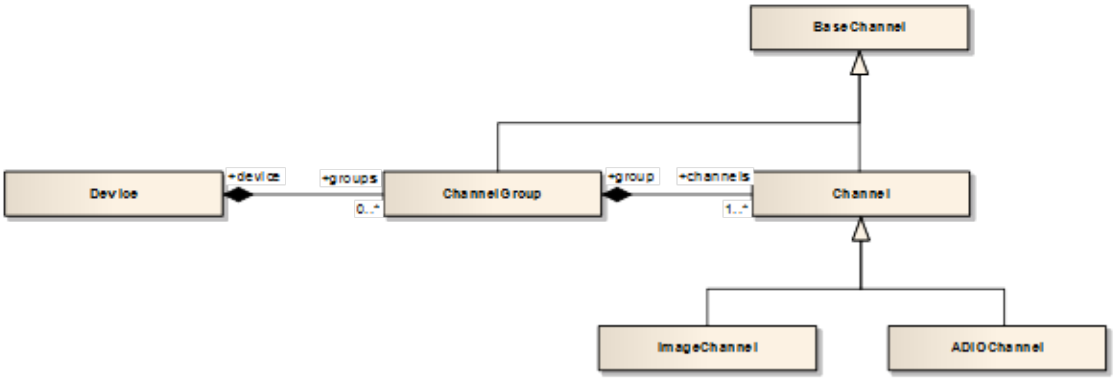

Figure 1: Classes diagram

A specific device class can only be inherited from the nds::Device class.

A device specific channel group can be inherited from ChannelGroup or AutoChannelGroup.

A device specific channel can be inherited from any class including nds::Channel and below.

3.3.1. Device Class

A device class provides common functionality for device objects.

3.3.1.1. ChannelGroup Iteration

```
for(ChannelGroupContainer::iterator itr = _nodes.begin();
    itr != _nodes.end();
    ++itr)
{

nds::ChannelGroup gr = itr->second->getBase();
}
```

3.3.1.2. Finding Channels

The nds::Device class provides the following functions to find ChannelGroups and Channels:

```
/*** Returns channel group instance by its name. */
ndsStatus getChannelGroup(const std::string& groupName, ChannelGroup);

/*** Returns channel instance by its name. */
ndsStatus getChannelByName(const std::string& channelName, Channel);

/*** Returns channel instance by group name and channel index. */
ndsStatus getChannel(const std::string& groupName, int idx, Channel);
```

3.3.2. ChannelGroup

ChannelGroup is a standard class which organizes channels into a group with some specific parameters. Simple groups are organized by channels type Analog/Digital/Image, Input/Output.

Channels organization can be also done from a logical perspective, e.g. channels, dependent on a concrete trigger, represent a trigger group.

nds::ChannelGroup is a base class for such an object. There is also AutoChannelGroup which provides advanced device configuration options. Channels can be added to the group automatically at the IOC initialization stage from the EPICS database.

3.3.2.1. Iteration of Channels

Channels can be iterated in the following ways:

[NDS v. 2.2.3] Iterating vector:

```
std::for_each(_nodes.begin(),
              _nodes.end(),
              boost::bind(&Channel::on, _1)));
```

[NDS v. 2.2.4] Iterating map:

```
std::for_each(_nodes.begin(),
              _nodes.end(),
              boost::bind(&Channel::on,
              boost::bind<Channel*>(&ChannelContainer::value_type::second, _1)));
```

In the general case, any container can be iterated through a type's iterator:

```
for(ChannelContainer::iterator itr = _nodes.begin();
    itr != _nodes.end();
    ++itr)
{
    ... (itr->second)
}
```

3.3.3. Channel

The nds::Channel class is the base class for any kind of channel object.

The nds::ADIOChannel object defines parameters which are common to the Analog and Digital I/O channels.

3.3.3.1. Accessing a Device from a Channel

The device can be accessed from nds::ChannelGroup or nds::Channel through the following field:

```
Device *_device;
```

3.4. Lifecycle of a Device Driver

A device driver distinguishes between 3 phases of IOC functioning: IOC configuration, IOC initialization, operating and exiting (Figure 2).

The IOC configuration state is used to configure the structure of the device, i.e. to define static parameters which will not be changed over the life cycle of the device. In this stage all classes are created, e.g. Device, ChannelGroup, Channels, etc. This stage includes all lines in st.cmd before the iocInit function call.

The structure of the device is defined immediately after the IOC shell command ndsCreateDevice is executed (see Figure 1 for the details). At that time, the constructor is called, and immediately afterwards the createStructure Device's member function. The whole device structure should be created inside the createStructure member: required ChannelGroups and

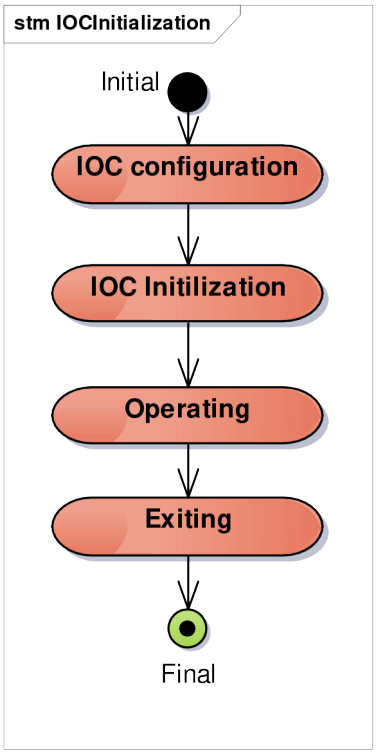

Figure 2: IOC state machine.

Channels. ChannelGroup must be registered within a device. If it is not registered, then processing of channel groups is not started. Similarly for Channels, each Channel must be registered within a relevant channel group. After registration `_portAddr` and `_portName` fields are populated by relevant values. A Channel's number within a group can be got through the `getChannelNumber()` method after the channel is registered.

In the case that the createStructure member returns an ndsError then IOC execution will be locked. This problem is treated as critical and further processing is impossible.

IOC initialization happens when the iocInit function is called. In this state all connections between EPICS records and NDS PV handlers (getter and setter) will be established. All EPICS I/O interrupts will be registered. AutoChannelGroup will fill their channels list on this stage.

Exiting happens when exit is explicitly requested or by Ctrl-C. Then, NDS cancels all existing tasks and calls the destroy() function member for each class and then deletes objects.

When IOC initialization begins, the state of all objects is changed to IOCInitialization. This is a good point to take any action that needs to be done before initialization starts and can be done by registering the state event handler (see chapter 4). For example, the configuration done via IOC shell commands will already have been done at this point and all objects will have relevant operation to operate (`_portAddr`, `_portName`).

When IOC initialization is complete, the state of all objects is changed to OFF. At this stage,

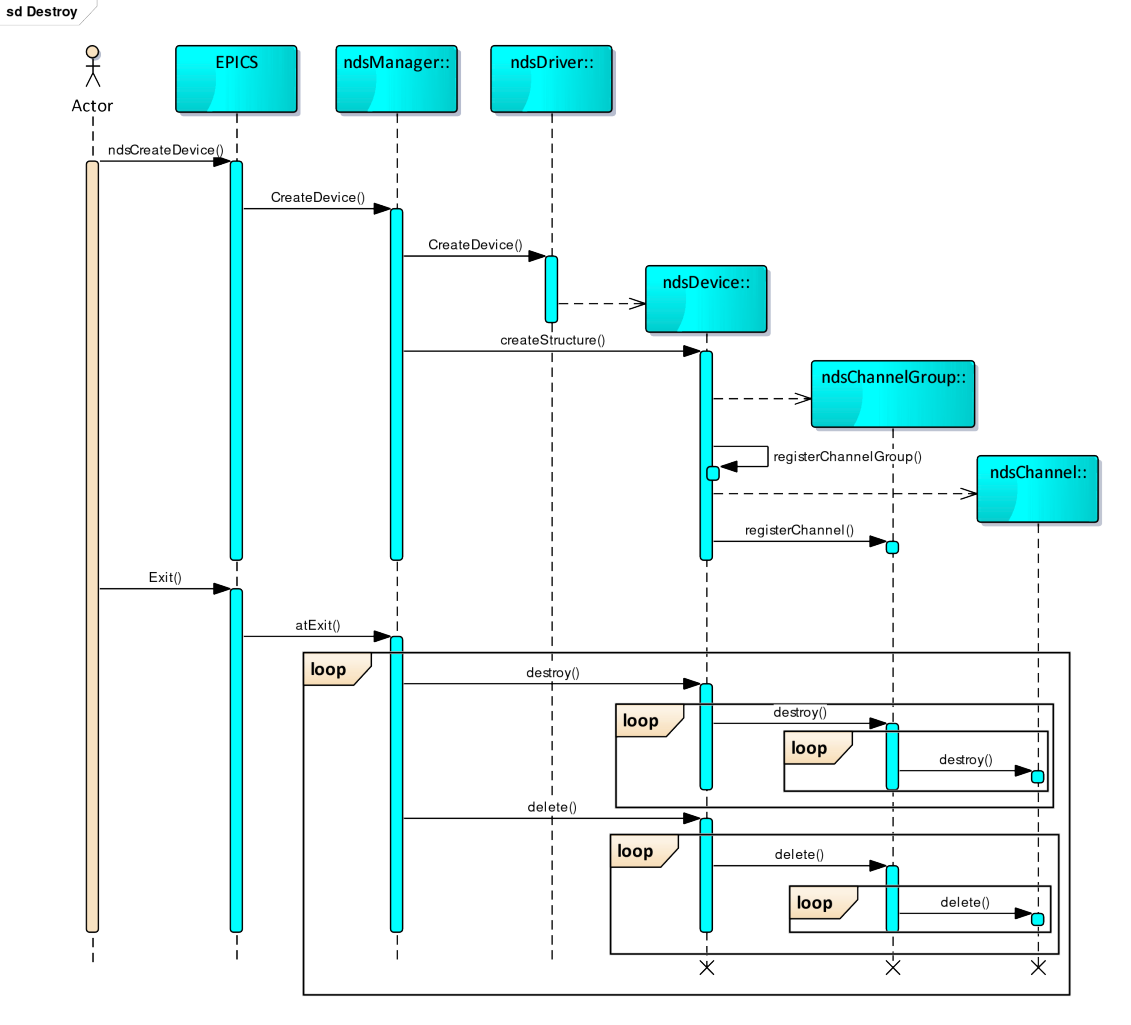

Figure 3: Creation and destruction of objects.

the initial processing of records as defined in the IOC's EPICS database is complete. Thus, the records will have been fully configured. This is a good place to

- check validity of configuration,
- spawn any threads that may be needed,
- initialize data structures,
- read information (versions, model, etc.) from the device,
- allocate memory,
- etc.

When the EPICS process is about to exit, the destroy function is called. In destroy, the device driver should:

- put device in a safe state,
- disconnect from the device and terminate the threads.

It is recommended that it waits for the threads to complete, i.e., signals them to terminate, and then waits for them to actually terminate.

3.5. Creating Device Structure

Device structure is created inside nds::Device::createStructure. This method is called after the device object is created. At call time, the device object has all parameters parsed and everything is prepared to create children objects. The CreateStructure function is only used to create a virtual device structure. It is not the place to open the device or to start the configuration process of the device.

3.5.1. Accessing Initialization Parameters

To access initialization parameters that were set with parameters of the ndsCreateDevice IOC shell command, use the following functions:

```
std::string strParam = getStrParam("PARAM1", "default value");
int intParam = getIntParam("PARAM2", 15);
```

These parameters are available throughout most of the lifecycle of the device object: from the time when the createStructure function was called, to the time when destroy finishes executing. Usually, however, the device-specific driver will process initialization parameters in createStructure (e.g., by initializing internal data structures) and will no longer need to consult them at a later time.

The getStrParam function returns the value of initialization parameter PARAM1 as a string, or the provided default value if the parameter is not specified in the call to ndsCreateDevice. Similarly, getIntParam returns the integer value of the requested parameter.

For example, if st.cmd defines:

```
ndsCreateDevice "_APPNAME_", "$(PORT)",
"FILE=/tmp/q,N_AI=2,N_AO=3,N_DI=4,N_DO=5,N_DIO=6,N_IMAGE=7"
```

Then parameters can be accessed:

```
std::string strParam = getStrParam("FILE", "/dev/testdev");
int numAI = getIntParam("N_AI", 0);
int numAO = getIntParam("N_AO", 0);
int numDI = getIntParam("N_DI", 0);
...
```

3.6. Defining Channels

A device can consist of multiple **channel groups**, and each channel group consists of multiple **channels**. Usually, but not necessarily, all channels within a channel group are of the same type (e.g., analog inputs, analog outputs, digital inputs/outputs, etc.).

Each channel group must be assigned a unique name. For each channel group, the NDS creates an asyn port [RD5] that is used to communicate with channels within this group. The port name is formed by the device port name plus channel group name (lowercase) separated by dots.

For example, for a channel group "AI", containing analog inputs and a device registered with port name "pxi6368.0", the port name of the "AI" channel group will be "pxi6368.0.ai". NDS will build port name and register port for channel group automatically.

Device's functions are accessible though the device asyn port and asyn address is 0.

ChannelGroup's functions are accessible through ChannelGroup's asyn port with address is -1.

Channels have sequencing numbers and are numbered automatically according to its registration, starting from 0. Channels are on the ChannelGroups asyn port [RD5].

- PortName:0 – device functions.
  - PortName.ChannelGroupName:-1 – channel group's functions.
    - ◇ PortName.ChannelGroupName:ID – functions of channel with ID.

Example of port:address structure relatively to object's functions for device port named: "pxi6368.0".

- pxi6368.0:0 – device functions.
- pxi6368.0.ai:-1 – "AI" channel group functions.
  - pxi6368.0.ai:0 – AI0 channel's functions.

  - pxi6368.0.ai:1 – AI1 channel's functions.
  - ...
  - pxi6368.0.ai:31 – AI31 channel's functions.
- pxi6368.0.dio:-1 – "DIO" channel group functions.
  - pxi6368.0.dio:0 – DIO0 channel's functions.
  - pxi6368.0.dio:1 – DIO1 channel's functions.
  - ...
  - pxi6368.0.dio:15 – DIO15 channel's functions.

There are two technical implications that need to be considered when assigning channels to channel groups:

Meaning of asyn addresses. Combination of asyn port and address must be unique. Thus, it is possible for AI channel 0 and DIO channel 0 since both have asyn address of 0, provided that they are in different channel groups. If instead the AI0 and DIO0 channels were both part of the same channel group, they would necessarily have different asyn addresses to avoid conflict (e.g., AI0 would be 0 and, DIO0 would be, say, 32, if there were 32 AI channels using addresses from 0 to 31). This would make the EPICS database more difficult to configure, as channel-asyn address mapping would need to be considered. Concurrency.

Whenever the asynDriver makes a call to a function of a port driver, it locks the port. Thus, two functions of the same port cannot be executed concurrently. This implies that functions of the same channel group cannot execute concurrently. While in most cases such a locking strategy is correct and optimal, that is not necessarily always the case:

- If locking is needed across channel groups, then the developer of the device-specific NDS driver must use locking explicitly. epicsMutex from EPICS' libCom library [RD3] is the recommended mechanism to implement such locking.
- If all channels are independent (i.e., no other channels need to be locked when code for a particular channel is executing), then the channels can be put in separate channel groups to improve concurency. The extreme scenario is when each channel would be put in its own channel group – then, there would be no locking among channels.

3.6.1. Manual Channel Creation and Registration

The place in the device-specific NDS driver code where the structure of the device is defined is the createStructure function, which is overridden in the device's derivative of the NDS Device base class. This function must create channel groups and channel objects, and register channel objects with channel group objects. It might look like this:

```
ndsStatus ExampleDevice::createStructure(const char *portName,
        const char *params)
{
    /* This device has two channel groups. */
    nds::ChannelGroup cgAI = new nds::ChannelGroup("AI");
    nds::ChannelGroup cgDIO = new nds::ChannelGroup("DIO");

    /* Both channel groups have 32 channels each. */
    for(int i = 0; i < 32; ++i) {
        cgAI->registerChannel(new ExampleAIChannel());
        cgDIO->registerChannel(new ExampleDIOChannel());
    }

    return asynSuccess;
};
```

In this example, 2 additional asyn ports will be created for “AI” and “DIO” channel groups respectively. AI channel group will have address 1 and DIO channel group will have address 2.

Classes ExampleAIChannel and ExampleDIOChannel must extend the Channel base class.

3.6.2. Automatic Channel Object Creation and Registration

NDS provides a method to create and register channels in an automatic mode. In this mode NDS waits for EPCIS DB initialization and creates the required channels automatically. This allows for keeping a required minimum number of active channels.

To allow NDS to configure channels automatically nds::AutoChannelGroup should be used. The developer should provide a fabric function to define how the channel should be created. The fabric function should return an instance of nds::Channel. This fabric is passed to AutoChannelGroup, on its creation, and will be automatically called for each required channel.

```
/* Fabric function for automatic channels creation. */
nds::Channel *createPFI()
{
    return new Terminal();
}

ndsStatus _APPNAME_Device::createStructure(const char *portName,
        const char *params)
{
    // Creating AutoChannelGroup object.
    nds::ChannelGroup *channelGroup =
            new nds::AutoChannelGroup("trg", &createPFI );
    return ndsSuccess;
}
```

createPFI is a fabric function which defines which channel class should be instantiated. **AutoChannelGroup** constructor accepts fabric function as a parameter.

3.7. EPICS Record Handling

NDS provides a way to connect EPICS record to C++ callback functions. These functions are called record handlers. For each record NDS registers a getter and a setter (see Figure 4 Handling CA requests.). Getters and setters are used for synchronous record processing.

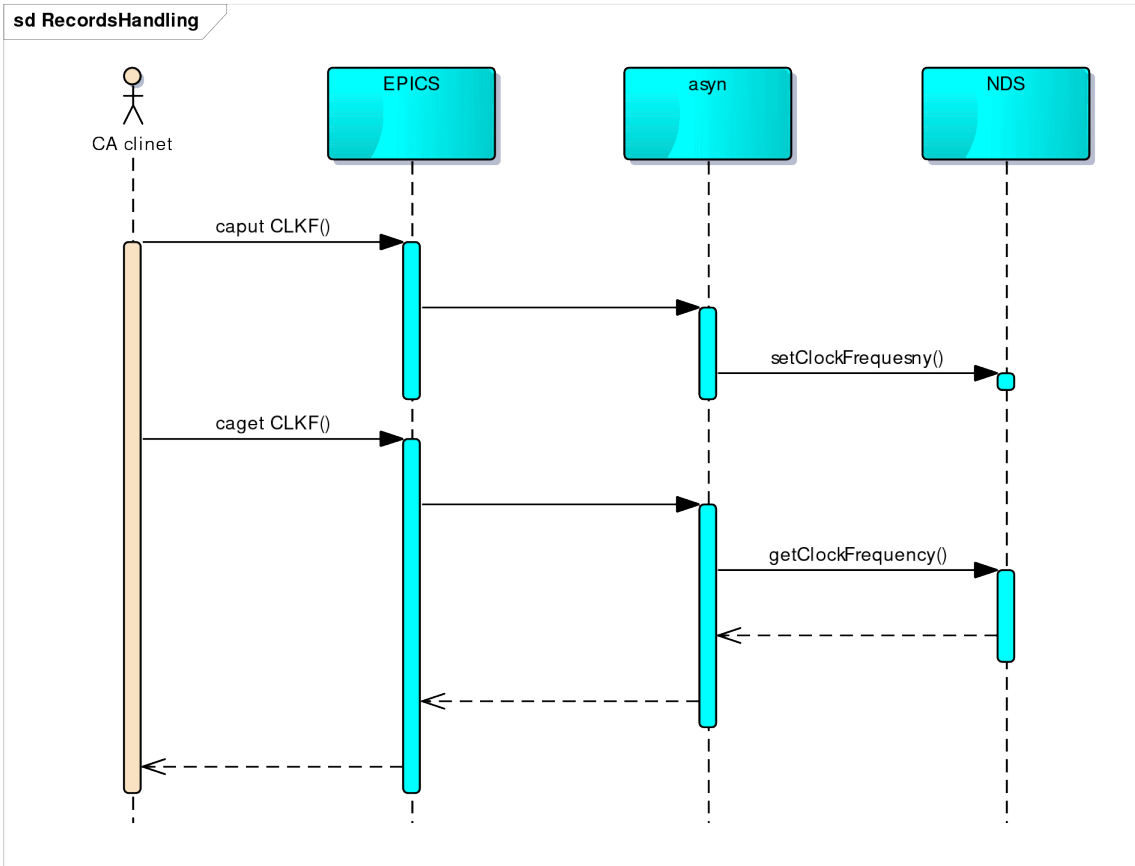

Figure 4: Handling CA requests.

Several EPICS records representing the common NDS signals are already registered by the NDS base classes to enforce the interface towards the users of the device. The list of signals is given in Appendix A, where column *Function* provides the suffix of the implementing getter and/or setter function names in C++. Functions are grouped by device/channel type.

NDS also provides asynchronous record updating through EPICS interrupts system. Each record whose SCAN field is set to "I/O Intr" is registered within NDS. Registered means that NDS owns interrupt ID which could be used to update record through `doCallbacks` functions (see Figure 5 and section 3.7.5 for more information).

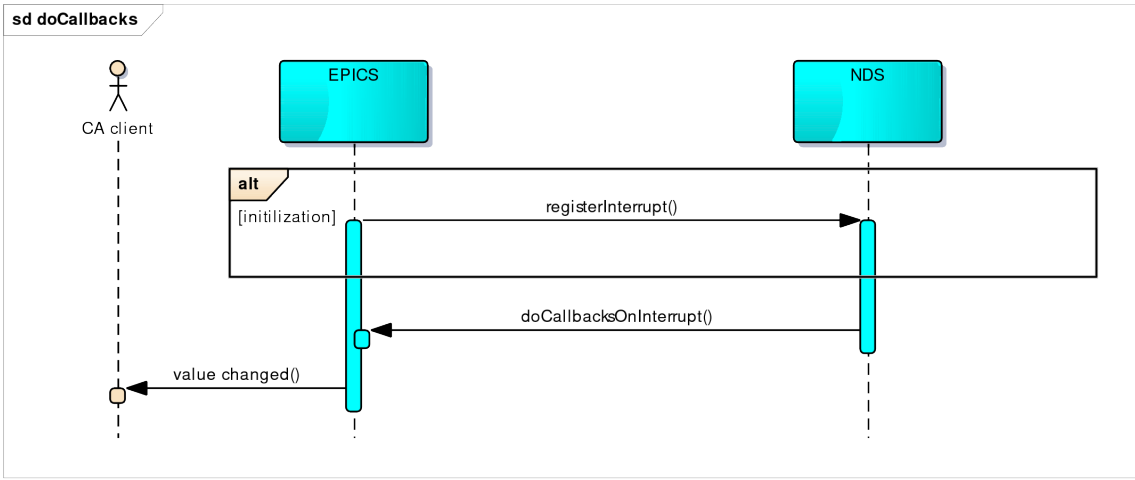

Figure 5: Asynchronous record update.

3.7.1. Synchronous EPICS Record Handling

Developers can easily add new PVs and register their handlers by overloading `registerHandlers` function:

```
virtual ndsStatus registerHandlers(PVContainers *pvContainers);
```

This member can be overloaded for Device, ChannelGroup and Channel.

The following macro definitions can be used for registering the handlers:

- `NDS_PV_REGISTER_OCTET(reason, writer, reader, InteruptId, size = -1)`
- `NDS_PV_REGISTER_INT32(reason, writer, reader, InteruptId)`
- `NDS_PV_REGISTER_FLOAT64(reason, writer, reader, InteruptId)`
- `NDS_PV_REGISTER_INT8ARRAY(reason, writer, reader, InteruptId, size = -1)`
- `NDS_PV_REGISTER_INT16ARRAY(reason, writer, reader, InteruptId, size = -1)`
- `NDS_PV_REGISTER_INT32ARRAY(reason, writer, reader, InteruptId, size = -1)`
- `NDS_PV_REGISTER_FLOAT32ARRAY(reason, writer, reader, InteruptId, size = -1)`
- `NDS_PV_REGISTER_FLOAT64ARRAY(reason, writer, reader, InteruptId, size = -1)`
- `NDS_PV_REGISTER_UINT32DIGITAL(reason, writer, reader, InteruptId)`

Macro definitions accept the following parameters:

`reason`	Asyn reason name for the record. Reason name is used in the INP or OUT fields of the record (see [RD5]).
`writer`	Pointer to a setter function. It must be a member of the class which makes the registration or `NULL` if not writable.
`reader`	Pointer to a getter function. It must be a member of the class which makes the registration or `NULL` if not readable.

`interruptId`	Pointer to an integer variable which will retrieve the record's interruptId. For each reason the NDS driver will assign an integer code that is required by asynDriver. This code will be stored in this variable and should later be used together with `doCallback` functions to initiate interrupt (see asynPortDriver documentation).
`size`	[OPTIONAL] Only applicable for Octets and array PVs, representing the number of array elements or -1 (default) if size is unknown. This information can later be retrieved with `nds::Base::findDataSource` function used by, e.g., NDS plugins for which it may be required to know the size necessary for the buffer memory allocation.

Table 2: PV_REGISTER arguments

Depending on the record data type, the signature of the getter and setter functions for reason *XXX* should correspond to:

- If record is a simple type *T*:

```
ndsStatus setXXX(asynUser *pasynUser, T value)
ndsStatus getXXX(asynUser *pasynUser, T *value)
```

- If record is an array of simple type *T* elements:

```
ndsStatus getXXX(asynUser *pasynUser, T *outArray, size_t nelem,
    size_t *nIn)
ndsStatus setXXX(asynUser *pasynUser, T *inArray, size_t nelem)
```

- If record is an octet:

```
ndsStatus getXXX(asynUser *pasynUser, char *data, size_t maxchars,
    size_t *nbytesTransfered, int *eomReason)
ndsStatus setXXX(asynUser *pasynUser, const char *data, size_t numchars,
    size_t *nbytesTransfered)
```

For example:

```
ndsStatus registerHandlers(PVContainers *pvContainers)
{
  nds::<BaseClass>::registerHandlers(pvContainers);
  NDS_PV_REGISTER_OCTET("FirmwareVersion", NULL, &getFirmwareVersion,
       &_interruptIdFirmwareVersion);
}

ndsStatus getFirmwareVersion(asynUser *pasynUser, char *buf, size_t size, size_t
    *pcount, int *eomReason)
```

If a given register is not intended to be read from or written to from EPICS, it may specify `NULL` for `reader` and/or `writer` argument. The use of fake handlers which are still provided in the `nds::Base` class for legacy reasons should be avoided as of NDS v.2.3.10.

3.7.2. Getters Call on Initialization

[Supported from: v.2.2.3]

During IOC initialization EPICS calls getter functions for all registered records. This allows setting of initial values to the record. If a default value was already set by another tool or defined in the DB file, then it will be overwritten. To prevent overwriting of default values, the NDS getter should return an ndsError status. The example below provides a solution for this case:

```
ndsStatus Channel::getStreamingURL(asynUser *pasynUser,
       char *data, size_t maxchars, size_t *nbytesTransfered,
       int *eomReason)

{
    if(getCurrentState() == CHANNEL_STATE_IOC_INITIALIZATION)
    {
        // Prevent overwriting of VAL value on initialization stage
        return ndsError;
    }
    return ndsSuccess;
}
```

3.7.3. Overriding a Standard Function

NDS standardizes several functions so as to enforce the interface towards the users of the device. The list of functions is given in Appendix A , where column *Function* provides the suffix of the name of the function in C++ that implements it. Two C++ functions correspond to each row in the table:

- The getter (prefix get), which retrieves the value from the device.
- The setter (prefix set), which applies the value to the device.

All functions are defined as virtual functions in the base classes, so they are meant to be overridden in the derived classes by the device-specific NDS driver.

3.7.3.1. Example: Reading Value from Channel

For example, to overwrite the channel's read function (i.e., acquire data from device), you would declare it in the derived channel class as follows:

```
// The .h file.
class ExampleChannel: public nds::ADIOChannel {
    ...
public:
    virtual ndsStatus getValueFloat64(asynUser* pasynUser, epicsFloat64 *value);
    ...
};

// The .cpp file.

ndsStatus ExampleChannel::getValueFloat64(asynUser* pasynUser,
        epicsFloat64 *value)
{
    *value = ...; /* whatever code is needed to get the value from the hardware */

    return ndsSuccess;
}
```

3.7.4. Asynchronous EPICS Record Handling

[Supported from: v.2.2.3]

Requires: asynDriver v.4.21

There are set of functions for asynchronous records update. There is a set of doCallbacks functions for each asynType, for example:

```
ndsStatus doCallbacksInt8Array(...);
ndsStatus doCallbacksInt16Array(...);
ndsStatus doCallbacksInt32Array(...);
ndsStatus doCallbacksFloat32Array(...);
ndsStatus doCallbacksFloat64Array(...);
ndsStatus doCallbacksGenericPointer(...);
ndsStatus doCallbacksInt32(...);
ndsStatus doCallbacksFloat64(...);
ndsStatus doCallbacksOctet(...);
ndsStatus doCallbacksUInt32Digital(...);
```

There are overloading methods for this list which serve customer needs (see Appendix B).

3.7.5. Using Asynchronous Update

1. Record should be present in DB files.
   It can from NDS standard records or a device specific record.
   1. AsynType of record is defined in DB file:

```
field(DTYP, "asynInt32")
```

2. Record should be configured to for I/O interrupt scanning type [X] (see EPICS RRM [RD4]):

```
field(SCAN, "I/O Intr")
```

3. Record should be connected to the required asyn reason:

```
field(INP, "@asyn($(ASYN_PORT).$(CHANNEL_PREFIX), $(ASYN_ADDR)) Event")
```

For example:

```
record(longin, "$(PREFIX)-$(CHANNEL_ID)-EVT") {
    field(DESC, "Event")
    field(DTYP, "asynInt32")
    field(INP, "@asyn($(ASYN_PORT).$(CHANNEL_PREFIX),
      $(ASYN_ADDR)) Event")
    field(SCAN, "I/O Intr")
    field(TSEL, "$(PREFIX)-$(CHANNEL_ID)-TMST.TIME")
}
```

2. Record should be registered within NDS.

To do asynchronous update, the interrupt id is required.

```
ndsStatus Instance::registerHandlers(nds::PVContainers* pvContainers)
{
   nds::<BaseClass>::registerHandlers(pvContainers);
   ...
   NDS_PV_REGISTER_INT32("Event",
      &nds::Base::setInt32,
      &ExADIOChannel::getInt32,
      &idEvent); // getting interrupt ID for ''Event reason.
   ...
    return ndsSuccess;
}
```

3. Now everything is ready to update record asynchronously.

```
doCallbacksInt32(value, // new value
       idEvent); // interrupt ID
```

3.7.6. Asyn Reason Arguments

NDS provides a way to pass arguments to asyn reason from the EPICS DB. It means that the OUT/INP parameter of a record can be:

```
@asyn($(ASYN_PORT).$(CHANNEL_PREFIX), $(ASYN_ADDR)) Reason(arg1)
```

Here arg1 is the argument which will be accessible from the device support handlers.

NDS parses the asyn reason string and populates fields of the DriverCommand class instance:

```
class DriverCommand
{
public:
    /// Vector of reason's arguments
    std::vector<std::string> Args;

    /// Reason's getter. Returns clear reason.
    std::string getReason() { return _reason; }

    ...
};
```

Here:

getReason() returns a reason, without arguments, which will be used to call handlers;
Args is a vector of arguments which were passed to reason.

A pointer to the DriverCommand instance will be stored in the userData field of the asynUser instance and can be accessed from setter/getter in the following way:

```
ndsStatus Terminal::setReason(asynUser* pasynUser, epicsInt32 value)
{
   // Reason's arguments is accessible through DriverCommand class
   nds::DriverCommand *command =
      (nds::DriverCommand*)pasynUser->userData;
}
```

3.7.7. Get asynAddr from get/set member

Sometimes it is required to get the original asynAddr from within EPICS record's setter or getter functions. To do this, the following procedure could be used:

```
int portAddr;
pasynManager->getAddr(pasynUser, &portAddr);
```

3.8. Records Timestamping

EPICS allows timestamping of PVs from the device support module. This allows timestamping of records synchronously through setters/getters and asynchronously through the interrupt mechanism. The EPICS record should be configured to select the requested time stamping

source (see EPICS Record Reference Manual [RD4]). Each EPICS record has a TSE field which indicate the time stamp source. In this case, the value must be "-2".

Other records can use this record as a timestamp source by referencing it through the TSEL field. The TSEL field should point to the TIME field of the timestamp record:

```
field(TSEL, "\$(PREFIX)-\$(CHANNEL\_ID)-TMST.TIME")
```

3.8.1. Synchronous Timestamp Update

The synchronous method uses a record's setter and getter to update data and timestamps. They will be called synchronously when the EPICS channel access client sets or reads a value accordingly. Each setter and getter has an asynUser argument which is used to propagate the actual timestamp.

The EPICS record should be configured to read timestamps from device support:

```
field(TSE, "-2")
```

Example of record's timestamp setting:

```
ndsStatus Device::getTime(asynUser *pasynUser, epicsInt32 *value,
       size_t nelements, size_t *nIn)
{
    epicsTime time = epicsTime::getCurrent();
    epicsTimeStamp stamp = (epicsTimeStamp)time;
    pasynUser->timestamp = stamp;

    *nIn = 2;
    value[0] = stamp.secPastEpoch;
    value[1] = stamp.nsec;
    return ndsSuccess;
}
```

3.8.2. Asynchronous Timestamp Update

The asynchronous data update mechanism uses EPICS interrupts to propagate data.

To use this mechanism, an EPICS record's scan option should be configured to "I/O Intr" and device support (-2) should be selected as a timestamp source:

```
field(SCAN, "I/O Intr")
field(TSE, "-2")
```

Now the EPICS record is ready to receive interrupts and set the desired timestamp.

NDS provides the following function set to update data and timestamps simultaneously:

```
ndsStatus doCallbacksInt8Array(epicsInt8 *value, size_t nElements,
       int reason, int addr, epicsTimeStamp timestamp);

ndsStatus doCallbacksInt16Array(epicsInt16 *value, size_t
       nElements, int reason, int addr, epicsTimeStamp timestamp);

ndsStatus doCallbacksInt32Array(epicsInt32 *value, size_t
       nElements, int reason, int addr, epicsTimeStamp timestamp);

ndsStatus doCallbacksFloat32Array(epicsFloat32 *value, size_t
       nElements, int reason, int addr, epicsTimeStamp timestamp);

ndsStatus doCallbacksFloat64Array(epicsFloat64 *value, size_t
       nElements, int reason, int addr, epicsTimeStamp timestamp);

ndsStatus doCallbacksInt32(epicsInt32 value, int reason, int
       addr, epicsTimeStamp timestamp);

ndsStatus doCallbacksFloat64(epicsFloat64 value, int reason,
       int addr, epicsTimeStamp timestamp);

ndsStatus doCallbacksOctet(char *data, size_t numchars,
       int eomReason, int reason, int addr, epicsTimeStamp
       timestamp);
```

Each function has a timestamp argument to provide the actual update of the time.

3.9. Read Back Value

The EPICS out record does not allow setting of its value from with the setter function. It means that if the value passed to it cannot be applied and needs to be corrected within the handler, then handler is unable to notify the user about the actual value. The read back value PV could help. This solution includes 2 PV's: system parameter (PV in Figure 6) and "read back value" (PV_RBV in Figure 6). PV is an output EPICS record and PV_RBV is an input EPICS record with scanning type I/O interrupt events. Asyn does not provide a way to return an output record value from a device support module and this is why the second PV is required. When the actor sets a parameter from the CA tool, the relevant setter for the selected PV is called (setPV in Figure 6). The setter processes the request and then calls the doCallbacks function with actualValue which was set to update read back value.

DB code:

```
record(longout, "$(PREFIX)-TEST") {
    field(DESC, "TEST of RBV")
```

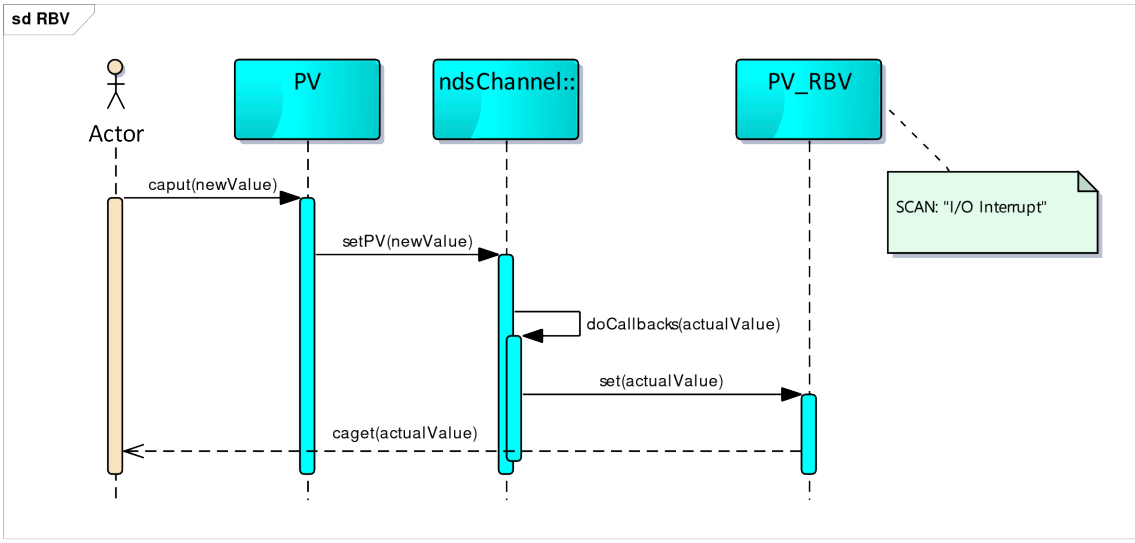

Figure 6: Read back values.

```
    field(DTYP, "asynInt32")
    field(OUT, "@asyn($(ASYN_PORT), $(ASYN_ADDR))TestRBV")
}

record(longin, "$(PREFIX)-TESTRBV") {
    field(DESC, "TEST of RBV")
    field(DTYP, "asynInt32")
    field(INP, "@asyn($(ASYN_PORT), $(ASYN_ADDR))TestRBV")
    field(SCAN, "I/O Intr")
}
```

Source code:

```
ndsStatus ExDevice::registerHandlers(nds::PVContainers* pvContainers)
{
    // Example of additional PV Record handlers' registration

    // IMPORTANT! It is important to call parent register function
    // to register all default handlers.
    nds::Device::registerHandlers(pvContainers);

    ...

    NDS_PV_REGISTER_INT32(
            "TestRBV",
            &ExDevice::setTestRBV,
            &ExDevice::getTestRBV,
            &idTestRBV);
    ...
```

```
    ndsSuccess;

}

ndsStatus ExDevice::setTestRBV(asynUser* user, epicsInt32 value)
{
    NDS_INF("ExDevice::setTestRBV = %d", value);
    doCallbacksInt32(101, idTestRBV, \_portAddr);
    return ndsSuccess;
}
```

3.10. Messaging Mechanism

3.10.1. Handling Messages

NDS provides a means for users of NDS devices to send messages (e.g., commands) to the device or its channel, so that it can take some action (e.g., trigger a state transition, perform self-tests or upload firmware). Messages are received automatically by the NDS (i.e., a single Channel Access put is performed to deliver a message).

To register a handler for messages, use the function registerMessageWriteHandler. This function is defined in class Base, which is the base class of both Device and Channel – therefore, it is available at both device and channel levels.

Example:

```
// The .cpp file. Corresponding .h file not shown.

// A good place to register the message handlers is in the constructor:
ExampleDevice::ExampleDevice(const std::string& name): nds::Device(name)
{
    ...
    registerMessageWriteHandler(
      "MY_MESSAGE", /// Name of the message type
      boost::bind(
          &ExampleDevice::onMyMessage, /// Address of the handler
          this, /// Object which owns the handler
          _1,_2) /// Stub functors (see boost::bind documentation for details.)
    );
    ...
}

ndsStatus ExampleDevice::onMyMessage(asynUser* pasynUser, const
       nds::Message& msg) {
```

```
    Message response;
    response.messageType = msg.messageType;
    response.insert("TEXT", "Not implemented");
    response.insert("CODE", "-1");
    doCallbacksMessage(response);
    return ndsSuccess;
}
```

3.10.2. Standard Message Types

NDS predefines the following message types and handlers for them.

- Message types commons to all objects
  - RESET – handler: Base::handleResetMsg. It requests resetting process to be initiated.
- Device specific messages
  - ON – handler: Base::handleOnMsg. It requests switch Device ON (see state machine discription for details).
  - OFF – handler: Base::handleOffMsg. It requests switch Device OFF (see state machine description for details).
- ChannelGroup and Channel specific message types
  - START – handler: BaseChannel::handleStartMsg. It requests to start Channel's (ChannelGroup) processing. ChannelGroup should be in processing state to allow activate channel. START command doesn't depend on object global state (ENABLED/DISABLED).
  - STOP - handler: BaseChannel::handleStopMsg. I requests to stop Channel's (ChannelGroup) processing. Object will be put to DISABLED state. STOP command doesn't depend on object global state (ENABLED/DISABLED).

All handlers could be overloaded. The overloading method should provide a common messaging response.

3.10.3. CSS Messaging Support

Messaging (MSGS/MSGR) is represented by a waveform record. CSS's "Text Input" and "Text Update" components should be used to display the waveform string. Format for these components should be "String" to see the text. Selecting "Default" will show the elements in ASCII format.

3.11. Loading FPGA code

When the user sets the FWUP record, the NDS driver loads the binary image referred to in the meta-information XML file and checks it (SHA1 checksum, compatibility of the target, etc). If all is OK, a device-specific function is called that receives the binary image as parameter, and must pass it to the hardware (Figure 7).

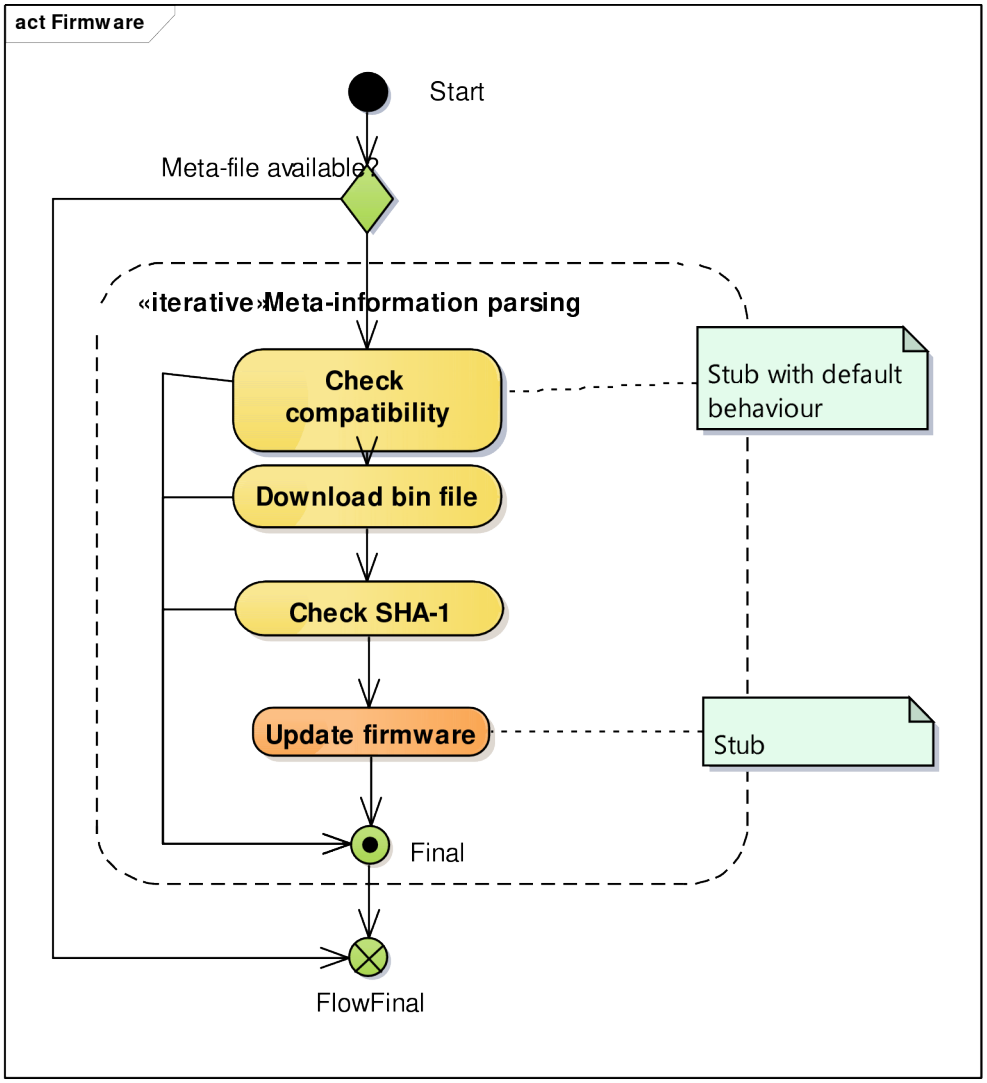

Figure 7: Firmware update activity diagram.

3.11.1. NDS v2.0 Implementation

This mechanism is not yet implemented in NDS v2.0, so in this version the name of the file is passed through the FWUP variable and the uploading function must be called from the PV's handler.

3.11.2. NDS v2.1 Implementation

NDS v2.1 fully supports xml parsing.

The update process will iterate through the firmware sections of the meta-information file (see Appendix D) and check if target for each section is compatible with the hardware. If a firmware compatible binary image is downloaded, the SHA-1 sum for the image is checked and the update firmware procedure is called. The check compatibility function has a default implementation which compares hardware model, hardware revision, and firmware revision (see A.1). The update firmware stub has the following signature:

```
virtual ndsStatus updateFirmware(const std::string& module,
const std::string& image, const std::string& method);
```

where:

- module – is a target module name (device dependent);
- image – path to image file;
- method – update method (device dependent);

```
ndsStatus ExDevice::updateFirmware(const std::string& module,
       const std::string& image, const std::string& method)
{
    NDS_INF("Updating firmware.");
    NDS_INF("Module to update: %s Current firmware version: %s ",
          module.c_str(), _firmwareVersion.c_str());
    NDS_INF("Image file: %s", image.c_str());

    if(!boost::filesystem::exists(image))
    {
         NDS_ERR("Update procedure was not able to find : %s", image.c_str());
         return ndsError;
    }
    <firmware loading>
    return ndsSuccess;
}
```

A specific NDS device support module is responsible for implementing the update procedure.

4. State Machines

Each device and each channel has a state, as shown in Figures 10 and 11. The device-specific driver interacts with the state machines in two ways:

- By setting the state. For example, if a fault is detected at the level of a channel, the channel should go to the ERROR state.
- Reacting on a change of state. For example, when the state of the channel goes to OFF, all activities on the channel should stop.

For requesting transition to a state, the following device-level functions are available in the Device base class:

- on(): request transition to the ON state.
- off(): request transition to the OFF state.
- error(): unconditional transition to the ERROR state.
- fault():unconditional transition to the FAULT state.
- reset():unconditional transition to the RESETTING state.

In the BaseChannel base class of all channels, the following channel-level functions are defined:

- disable(): request transition to the DISABLE state.
- start(): request transition to the PROCESSING state in which the data acquisition or waveform generation of the channel is in progress.
- stop(): request transition to the ON state.
- error(): unconditional transition to the ERROR state.
- reset():unconditional transition to the RESETTING state.

Some transitions are conditional – i.e., they may be vetoed by the state transition listeners, while others are unconditional.

To react to a change, a handler function that is called during the state transition must be registered. Three kinds of handlers can be registered:

- registerOnRequestStateHandler: the handler is called to confirm state transition.
- registerOnLeaveStateHandler: the handler is called when a state is exited.
- registerOnEnterStateHandler: the handler is called when a state is entered.

These functions are defined in the AbstractStateMachine class, from which both Device and BaseChannel base classes for devices and channels, respectively, are derived.

The following example registers handlers for confirming state transitions and for entering states:

```
// The .cpp file. Corresponding .h file not shown.

// A good place to register the state transition handlers is in the constructor:
ExampleDevice::ExampleDevice(const std::string& name): nds::Device(name)
{
    ...
    registerOnRequestStateHandler(
        nds::DEVICE_STATE_INIT,
        nds::DEVICE_STATE_ON,
        boost::bind(&ExampleDevice::onSwitchOnRequest, this, _1, _2));
    registerOnEnterStateHandler(
        nds::DEVICE_STATE_OFF,
        boost::bind(&ExampleDevice::onSwitchOff, this, _1, _2));
    ...
}

ndsStatus exDevice::onSwitchOnRequest(nds::DeviceStates from,
        nds::DeviceStates to)
{
    // Handling switch ON device request
    // If this function returns ndsError device will not be switched ON.
    return ndsSuccess;
}
```

The example above shows how to define the onRequestState handler onSwitchOnRequest. When the user calls the on() function, this handler will be called (see Figure 8). If this handler returns ndsSuccess then the device successfully goes to the ON state. If it returns ndsFault, then the device will go the FAULT state.

The NDS state machine provides 2 error states ERROR and FAULT (see Figure 10). FAULT state is used when operation of device is impossible, ERROR state for all other cases.

There are 4 NDS's status codes which can redirect transition from transitions request operation (see Figure 8):

- ndsSuccess – transition proceeds to requested state
- ndsError – result state ERROR
- ndsFault – result state FAULT
- ndsBlockTransition – object stays at the same state which was before transition requested.

The developer can register as many state events' handlers as possible. These handlers will be called in order of registration. In case of transition request handling, all handlers should return a status of ndsSuccess as it is only in this case that transitions will be treated as allowed.

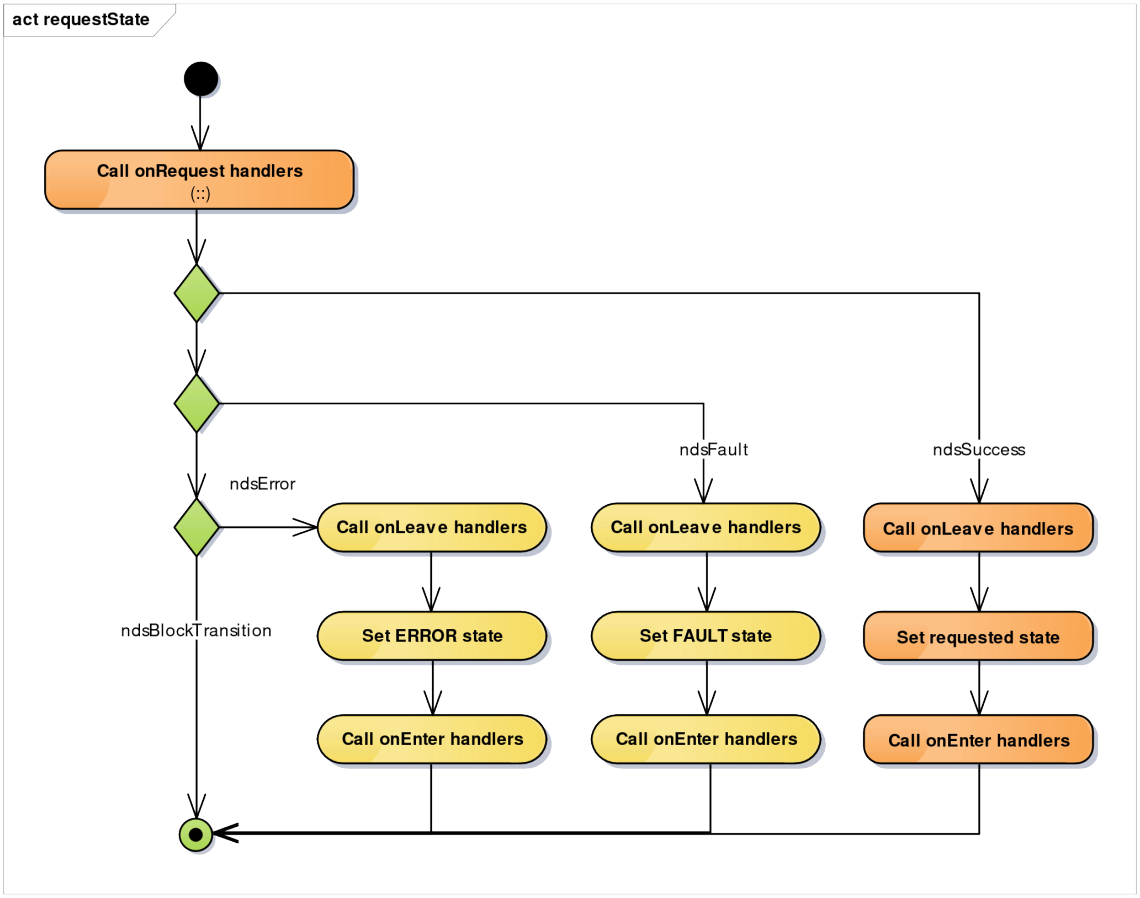

Figure 8: Request state activity diagram.

4.1. Enable Objects

Each object has an ENBL record which describes the global state of the object (Figure 9).

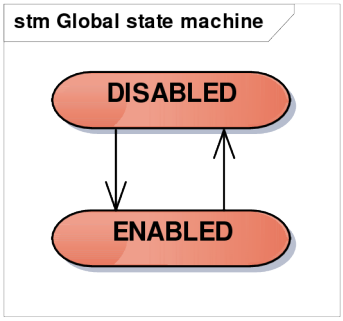

Figure 9: Global state machine.

An enabled status of this record means that the object is configured and ready to be used. This record could be switched independently for each object. If an object has this record enabled, then it means that as soon as the parent object switches to the active state (ON for Device objects, PROCESSING for ChannelGroup and Channel objects) this object will be forced to also switch to the active state.

NDS provides **isEnabled()** method to get the global status of the object.

4.2. Device's State Machine

ERROR and FAULT states are accessible from any other state.

4.2.1. Fast init

An ON message sent to a device forces the state machine to switch to the INIT state (when device is in OFF state). If the device does not require the INIT procedure ("Fast Init") then the device can be automatically switched to the ON state, by call this function from device constructor:

enableFastInit();

It will register an onEnter handler for INIT state. This handler requests the ON state.

4.2.1.1. Example of One Device File Handling

If the device is using one device file (/dev/device_name) to access all components (Segments, ChannelGroups, Channels) then the device object should take care of the lifecycle of the device file inside NDS. This case requires 2 transition handlers on NDS::Device level: onRequestState and onEnterState.

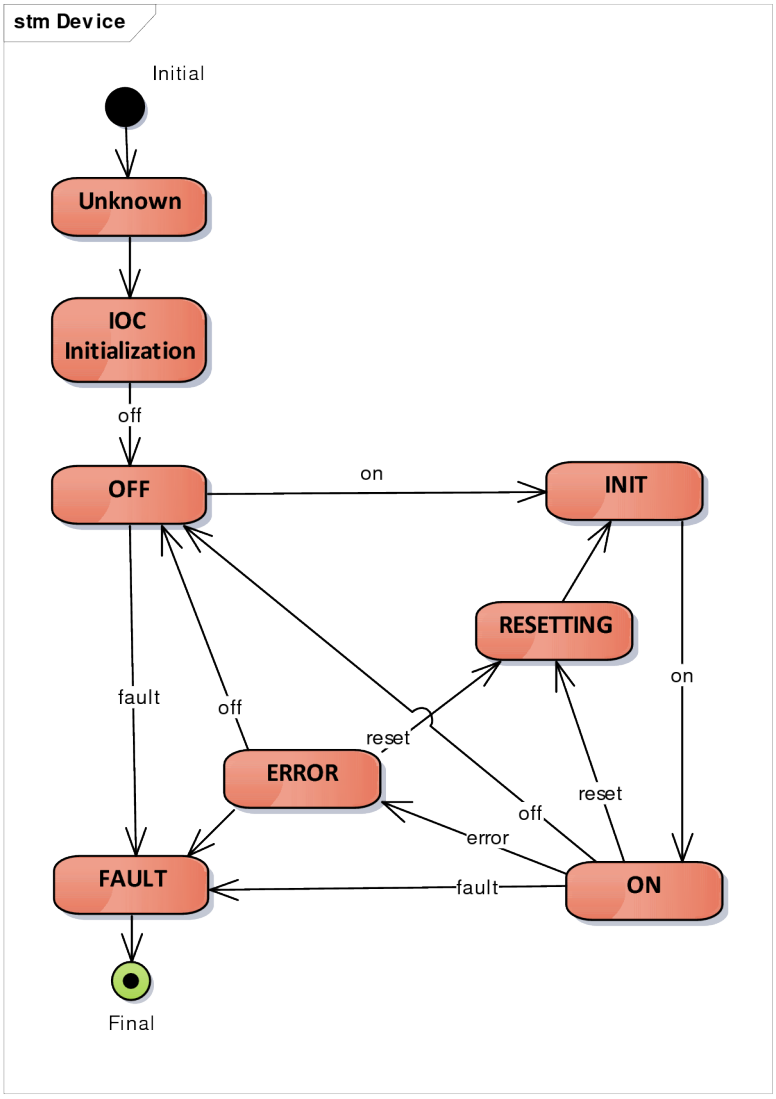

Figure 10: Device's state machine.

- onRequestState will open the device file and check if all works properly.
- onEnterState will pass the opened device pointer to all ChannelGroups. It will be called only in the case that the request state succeeds.

Steps

1. Inherit your ChannelGroup from nds::ChannelGroup.
2. Define setter for device file pointer inside your ChannelGroup class.

```
class yourChannelGroup: public nds::ChannelGroup
{
  DeviceAPI* _deviceAPI;

public:
  // Setter for device file pointer.
  setDeviceAPI(deviceAPI)
    {
      _deviceAPI = deviceAPI;
    }
}
```

3. Define openDevice member inside your device class and implement it. Your device object is inherited from nds::Device. This member will be called when the required transition is requested.

```
ndsStatus openDevice()
{
    _deviceAPI = open(...); // new DeviceAPI() in case of _deviceAPI is
        class
    if (deviceAPI)
         return ndsSuccess;
    return ndsError;
}
```

4. Define setDeviceAPIPointer member inside your device class. This member will be called when the actual transition happens and the device file pointer exists.

```
ndsStatus setDeviceAPIPointer()
{
  ChannelGroupMap::iterator itr;
  for(itr = _nodes.begin(), itr != _node.end(), ++itr)
  {
    try
    {
      yourChannelGroup *group = dynamic_cast<yourChannelGroup*>
          (itr.second->getBase());
      group->setDeviceAPI(_deviceAPI);
```

```
    }
    catch(...)
    {
      NDS_CRT"('Cant process".);
    }
    return ndsSuccess;
  }
}
```

5. Define closeDevice member inside your device class. This member will be called, when the device enters the OFF state, to close device.

```
ndsStatus closeDevice()
{
    if (_deviceAPI)
        close(_deviceAPI ); // delete _deviceAPI; in case deviceAPI is class
    return ndsSuccess;
}
```

6. register `onRequestState(init, on, openDevice)`
7. register `onEnterState(on, setDeviceAPIPointer)`
8. register `onLeaveState(on, closeDevice)`

4.3. ChannelGroup and Channel State Machine Implementation

ERROR and FAULT states are accessible from any state.

4.4. How to Block Undesired Transitions

Sometimes it is required to block a transition from, for example, the RESETTING to the ON state, to allow only transition through INIT state.

There are default onStateRequest handlers for this case, namely onWrongStateRequested, for nds::Device, nds::BaseChannel classes which could be registered on undesired transition. These handlers print out warning messages and return ndsBlockTransition status. As it was described earlier the ndsBlockTransition status prevents switching of instance state.

Example:

```
// If you would like to forbid transition
// from OFF state to ON state uncomment code below.
// This default request handler returns ndsBlockTransition,
// so device will stay in the same state (OFF).
registerOnRequestStateHandler(
```

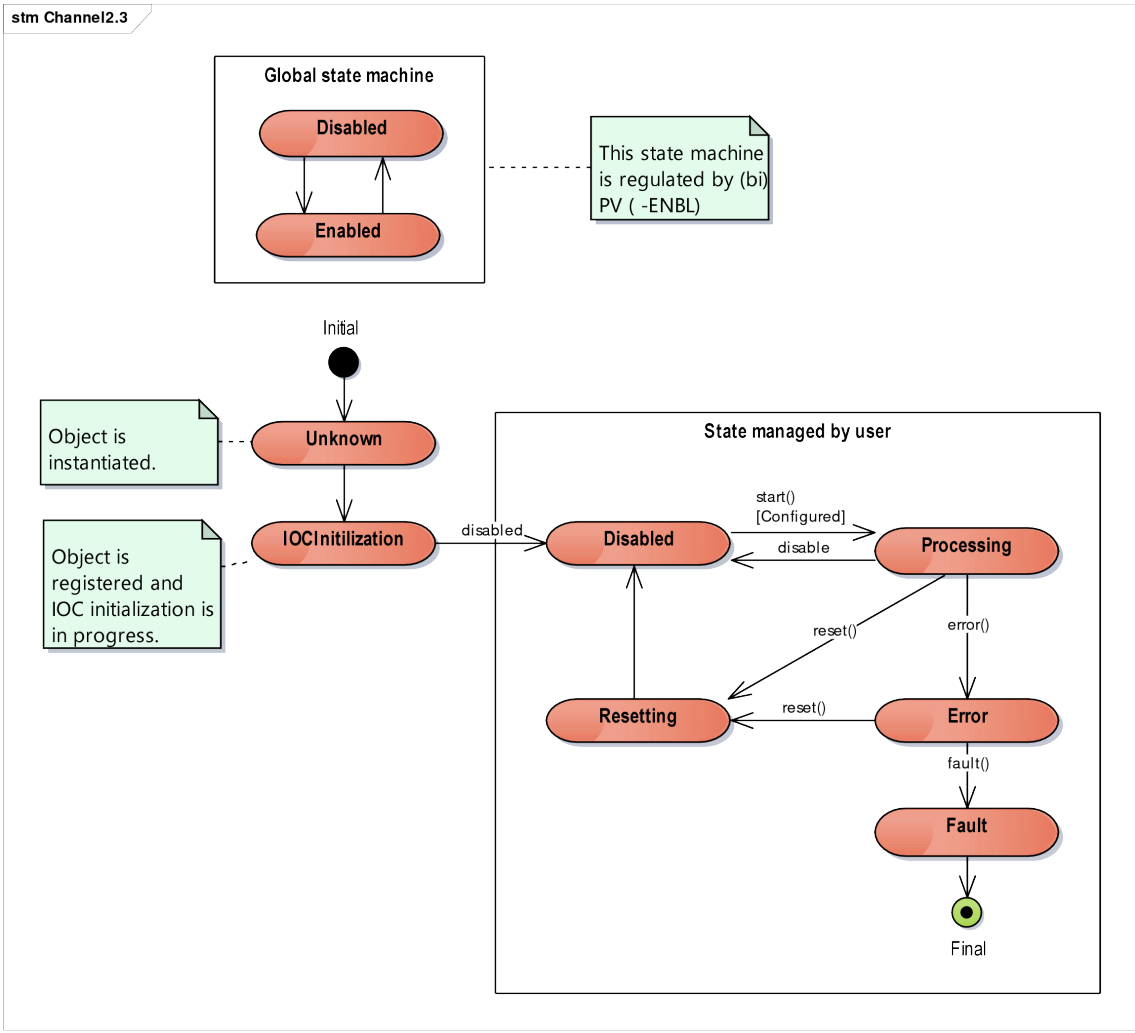

Figure 11: Channel state machine.

```
    nds::DEVICE_STATE_RESETTING, // Current state
    nds::DEVICE_STATE_ON, // Requested state
    boost::bind(&nds::Device::onWrongStateRequested, this, _1, _2));
```

4.5. Reset Operation

Supported from: v.2.2.1

A general way of performing resetting is to start the resetting operation from an onEnterState handler and start an nds::ThreadTask which will track the device status and when device is ready, after resetting, will switch the device to the required state (OFF, INIT, ON).

4.5.1. Switching from the RESETTING State

The RESETTING state is accessible by the reset() command, but switching from the reset() command is blocked through the standard functions on(), off(), and init(). Transition to the required state could be requested in the following way:

Example: Requesting ON state for Device instance

```
getCurrentStateObj()->requestState(this, nds::DEVICE_STATE_ON);
```

For Channel/ChannelGroup instance (requesting ENABLED state):

```
getCurrentStateObj()->requestState(this, nds::CHANNEL_STATE_ENABLED);
```

If requesting an unconditional transition is required, without calling onRequestState handlers, then the following code could be used:

```
getCurrentStateObj()->setMachineState (this, nds::CHANNEL_STATE_ENABLED);
```

The ERROR or DEFUNCT states could be requested in the normal way through calling the error() and defunct() functions respectively.

4.6. Automatic ChanelGroups and Channels switching

Sometimes the state of the children should be synchronized with parent's state. For example when device is switched ON all enabled ChannelGroups could be switched to PROCESSING state automatically or when ChannelGroups is switched to PROCESSING all enabled Channels will be switched in its turn to PROCESSING automatically.

Developer is responsible to enable this functionality. It could be easily done by registering predefined state handlers.

Note: State handlers are executed in the order of registration. This should be taken in to account when children switching handlers are registered.

4.6.1. For ChannelGroups

The following handlers could be registered in Device state machine:

```
// Enabled ChannelGrous will be started automatically when device is switched ON
// NOTE: handlers are called according to registration order
registerOnEnterStateHandler(nds::DEVICE_STATE_ON,
      boost::bind(&nds::Device::startChannelGroups, this, _1, _2));

// ChannelGrous will be disabled automatically when device is switched OFF
// NOTE: handlers are called according to registration order
registerOnEnterStateHandler(nds::DEVICE_STATE_OFF,
      boost::bind(&Device::disableChannelGroups, this, _1, _2));
```

4.6.2. For Channels

The following handlers could be registered in ChannelGroup state machine:

```
// Enabled Channels within the ChannelGroup will be started automatically
// NOTE: handlers are called according to registration order
registerOnEnterStateHandler(nds::CHANNEL_STATE_PROCESSING,
      boost::bind(&ChannelGroup::startChannels, this, _1, _2) );

// Channels within the ChannelGroup will be DISABLED automatically
// NOTE: handlers are called according to registration order
registerOnEnterStateHandler(nds::CHANNEL_STATE_DISABLED,
      boost::bind(&ChannelGroup::disableChannels, this, _1, _2) );
```

5. NDS Tools

5.1. epicsMutex and AsynDriver Locker

Supported from: v.2.2.3

NDS provides lockers for epicsMutex and asynDriver which will automatically unlock resources when the processor goes out of its defined scope (RAII).

List of macroses serving locking/unlocking is shown below:

```
#define LOCK_ASYN_PORT(port)
#define UNLOCK_ASYN_PORT(port)

#define LOCK_MUTEX(mutex)
#define UNLOCK_MUTEX(mutex)
```

5.2. Define Driver Specific IOC Function

Supported from: v.2.2.2

The nds::Manager has the findDevice member function which can be used to get a pointer to a nds::Device instance. It allows for the definition of device specific functions which can be accessible from the IOC.

```
/**
 * @param portName - the device port name which was used to create device by nds
     ::createDevice.
 * @param device -
 * @return ndsSuccess - if device found
 * @return ndsError - if device is not found
 */
ndsStatus findDevice(const char *portName, Device** device);
```

For example:

```
int configureDevice (const char *devicePort, int param)
{
    nds::Device *ptr;
    nds::Manager::getInstance().findDevice(devicePort, *ptr);

    // Device specific function (not implemented in NDS)
    ptr->setParam(param)
```

```
}
```

5.3. NDS Tasks

NDS provides a common interface for asynchronous tasks (nds::BaseTask). This allows managing tasks in a single way. All asynchronous tasks are registered within an NDS task manager. The task manager takes care of tasks cancellation when an exit is requested. The task manager requires unique task names. If a task name is not unique, then this is treated as a critical error and the task manager blocks IOC execution.

The task manager provides a name generator to help device integrators create unique names.

The BaseTask interface gives 2 main functions to manage tasks: start() and cancel(). The first starts task execution, while the second cancels task execution. BaseTask has an internal state machine to prevent a task's double starting or double cancelling, so start() and cancel() methods cannot be overwritten directly. The interface includes 2 private functions to override startPrv() and cancelPrv().

BaseTask sets a task's state machine to *ndsThreadStopping* and *ndsThreadStarting*. Each concrete task should transfer task to final states *ndsThreadStopped* and *ndsThreadStarted*, to provide normal task processing.

5.3.1. nds::TaskService

[NDS v.2.3, ndsTaskService.h]

nds::TaskService is a service class for NDS tasks. The main goal of this class is to simplify the cancellation of tasks. There are 4 main operations which could be performed on a TaskService object:

- Request task cancelling:

  ```
  void cancel()
  ```

- Check if task was cancelled:

  ```
  volatile bool isCancelled()
  ```

- Initiate sleep procedure:

  ```
  WaitStatus sleep(double seconds)
  ```

  The task service provides a special sleep procedure which could be interrupted by a cancel request. Only this sleep procedure should be used in a task's handlers to allow normal cancellation process.
  The sleep procedure can have following statuses:

ndsWaitStatusEvent means that a cancel event was received.
ndsWaitStatusError means that the task was not able to start the wait procedure.
ndsWaitStatusTimeout is returned when the sleep timeout expires.

- Clean cancelled status:

```
void restore()
```

Cleaning of a cancelled status will not restart the task.

5.3.2. nds::ThreadTask Class

[NDS v.2.3, ndsThreadTask.h]

This is a simple thread task which provides an easy method of registering a thread handler. A thread handler will be run in a separate thread once. When a thread handler returns, ThreadTask stops.

ThreadTask could be created through the factory method:

```
   static ThreadTask create(const std::string& name,
          unsigned int stackSize,
          unsigned int priority,
          TaskBody body);
```

TaskBody is a task handler which has following signature:

```
   void (*)(TaskServiceBase& service)
```

The thread task could be restarted. For each start of the task, a new instance of the EPICS thread will be created.

For example:

```
#include <ndsThreadTask.h>

class ExDevice: public nds::Device
{
private:
  nds::ThreadTask resetTask;

public:
  ...
  /// emulation of resetting process
  ndsStatus resetProcess(nds::TaskServiceBase &service);
}

ExDevice::ExDevice(const std::string& name):nds::Device(name)
```

```
{
  ...
  resetTask = nds::ThreadTask::create(
      nds::TaskManager::generateName("Resetting"),
      epicsThreadGetStackSize(epicsThreadStackSmall),
      epicsThreadPriorityMedium,
      boost::bind(&ExDevice::resetProcess, this, _1));
}

ndsStatus ExDevice::onReset(nds::DeviceStates, nds::DeviceStates)
{
    NDS_INF("Start resetting.");
    resetTask->start();
    return ndsSuccess;
}

ndsStatus ExDevice::resetProcess(nds::TaskServiceBase &service)
{
    doCallbacksMessage("RESET",0,"Resetting in progress.");
    service.sleep(5.0);
    doCallbacksMessage("RESET",0,"Resetting complete.");
    getCurrentStateObj()->requestState(this, nds::DEVICE_STATE_ON);
    return ndsSuccess;
}
```

5.3.3. nds::PeriodicTask Class

[NDS v.2.3, ndsPeriodicTask.h]

nds::PeriodicTask is a task which repeats execution of the task body with the requested period.

An instance of the nds::PeriodicTask could be created through the fabric method:

```
static PeriodicTask* create(const std::string& name,
       unsigned int stackSize,
       unsigned int priority,
       TaskBody body);
```

TaskBody is a task handler which has the following signature:

```
void (*)(TaskServiceBase& service)
```

The task could be cancelled and restarted again.

5.3.4. nds::PollingTask Class

[NDS v.2.3, ndsPollingTask.h]

nds::PollingTask provides a polling core which correctly handles interruptions.

The polling task could be created through the factory method create(), which has the following functionality:

```
static PollingTask* create(
       const std::string& name,
       unsigned int priority,
       unsigned int stackSize);
```

Where:

name is a task name. priority
epicsThread priority. stackSize
epicsthread stack size.

A file descriptor's registration is done through the following method:

```
ndsStatus addFile(int fileFd, FileEventHandler handler,
       uint32_t events = EPOLLIN);
```

Where:

- fileFd is a valid file descriptor.
- handler is a callback method to handle event on this file descriptor.

The handler will be called on an event on this file descriptor. The developer should handle all possible events which could occur. nds::PollingThread does not handle any file events, it just calls the handler.

The polling task handler has following signature:

```
void (*)(TaskServiceBase& service, const struct
       epoll_event& event);
```

The handler receives a service object to serve the handler's needs and an event object which the epoll_wait function returns.

The polling task could be restarted. Each time the PollingTask starts, a new instance of the thread will be created. Note: the polling task frees a list of file handlers on cancelling, so before a new iteration file descriptors should be reregistered within a polling task. It allows for the closing of file handlers when a polling task is cancelled.

5.3.4.1. nds::PollingTask Example

This example describes channel processing for a device which has separate device files for each channel. The channel's active state is PROCESSING. All state handlers mentioned relate to the PROCESSING state. A simplified activity list is:

1. Open device file from onRequestState handler. If device file cannot be opened then the handler should return ndsError which means channel cannot be processed.
2. Add files and starting polling from onEnter handler.
3. Cancelling polling from onLeave handler. Close files.

For example:

```
#include "ndsPeriodicTask.h"
#include "ndsPollingTask.h"

class ExADIOChannel: public nds::ADIOChannel
{
protected:
    ...
    nds::PeriodicTask *taskPeriodic;
    nds::PollingTask *taskPolling;

public:
    ...
    /// Concrete channel specific member
    void processSamplingBody(nds::TaskServiceBase& service);

    /// Concrete channel specific member
    void processReadBody(nds::TaskServiceBase& service);
}

ExADIOChannel::ExADIOChannel…()
{
    ...
    if (!_isOutput)
    {
        snprintf(timerName, sizeof(timerName), "pipe-reader-%s", fileName);

        taskPolling = nds::PollingTask::create(
                timerName,
                epicsThreadGetStackSize(epicsThreadStackSmall),
                epicsThreadPriorityMedium);
    }

    snprintf(timerName, sizeof(timerName), "pipe-sampler-%s", fileName);

    taskPeriodic = nds::PeriodicTask::create(
            timerName,
            epicsThreadGetStackSize(epicsThreadStackSmall),
            epicsThreadPriorityMedium,
            boost::bind(
```

```
                &ExADIOChannel::processSamplingBody,
                this,
                _1));
}

ndsStatus ExADIOChannel::onSwitchOn(nds::ChannelStates prevState,
       nds::ChannelStates currState)
{
    ...
    // Starting periodic task
    if (taskPeriodic)
    {
        taskPeriodic->start(epicsTime::getCurrent()+10, clockPeriod()/1.0e9);
    } else {
        NDS_DBG("Periodic task is not defined.");
    }

    if (!_isOutput)
    {
        if(taskPolling)
        {
            // Opening file
            if(fileFD == -1)
            {
                fileFD = open(fileName, O_RDONLY | O_NONBLOCK);
                NDS_DBG("Openning file: %s fd: %d", fileName, fileFD );
                if(fileFD == -1) {
                    NDS_ERR("Can't open '%s' for reading.", fileName);
                    return ndsError;
                }
            }

            // File descriptor registration within a polling thread

            if(taskPolling->addFile(fileFD,
                   boost::bind(&ExADIOChannel::processReadBody, this, _1, _2)) ==
                   ndsSuccess )
            {
                 // Start polling
                 taskPolling->start();
            } else {
                return ndsError;
            }
        } else {
            NDS_DBG("Polling task is not defined.");
```

```
        }
    }
    return ndsSuccess;
}

ndsStatus ExADIOChannel::stopProcessing(nds::ChannelStates
        prevState,
        nds::ChannelStates currState)
{
    NDS_INF("ExADIOChannel::stopProcessing");

    eventTimer->cancel();
    if (!_isOutput)
    {
        // Cancelling polling task
        if(taskPolling)
        {
            taskPolling->cancel();
        } else {
            NDS_DBG("Polling task is not defined.");
        }
    }
    // Cancelling periodic task
    if (taskPeriodic)
    {
        taskPeriodic->cancel();
    } else {
        NDS_DBG("Periodic task is not defined.");
    }

    return closeFiles();
}
```

5.3.5. nds::Timer Class

[NDS v.2.2.3, ndsTimer.h]

This is a wrapper over the EPICS Timer class, which synchronizes the timer interface with the nds::BaseTask. All nds::Timer objects are registered within the nds::TaskManager and will be automatically cancelled during the exit procedure.

```
TimerPtr create(const std::string &name, OnTimeHandler p);
TimerPtr create(const std::string &name, epicsTimerQueue &queue,
        OnTimeHandler p);
```

Where:

- name – is a timer name, unique within IOC.
- OnTimerHandler has the following signature:

```
epicsTimerNotify::expireStatus (*)(TaskService& service,
       const epicsTimestamp timestamp);
```

6. NDS Timing Device

6.1. Time Events

A time event is an abstract event which can happen on a terminal line. Each event is defined by event id (the event name can be used as the event id) and source terminal. The quantity of events is not limited.

Each time event represents one event of the following types:

- Future Time Event (FTE)
- Pulse
- Clock

The time event structure is hierarchical. FTE is a base. Pulse and clock extend it consecutively.

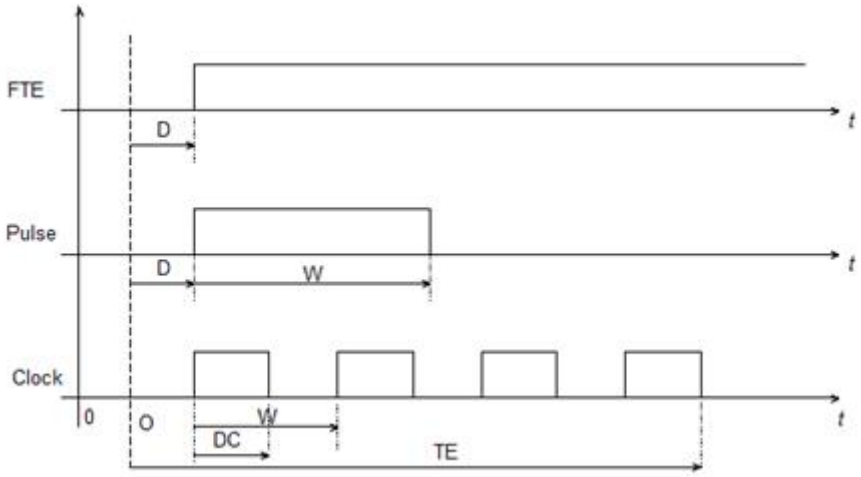

Figure 12: Timing diagram

FTE has following PV's:

- O – origin time - expressed as a double and representing an absolute time in seconds with a maximum resolution of a millisecond;
- D – delay - expressed as a double and representing a time in seconds with the resolution of a nanosecond and with an absolute limit of one day and representing delay of this occurrence from the time origin.
- E – enabled/disabled;
- L – level of scheduled time event;
- A – acquisition - provides the user with the information that the timing has been triggered with a time measurement expressed as a double and representing delay of this occurrence from the time origin.

Pulse adds width PV to this list:

- W – width of event in seconds with the resolution of a nanosecond and representing the delay from the pulse start time;

Clock adds duty cycle and generation end time:

- DC – duty cycle in seconds with the resolution of a nanosecond;
- TE – clock generation end time in seconds with the resolution of a nanosecond and with an absolute limit of one day and representing delay of this occurrence from the time origin.

NDS provides templates for each type of time event. Each required event template should be instantiated separately.

Example of FTE instantiation from st.cmd

```
## Loading FTE's events
# EVENT_NAME - human readable event name
# EVENT_ID - event identification
# TERMINAL_ID - event's terminal identification

dbLoadRecords "db/_APPNAME_FTE.template", "PREFIX=_APPNAME_,
ASYN_PORT=$(PORT), ASYN_ADDR=0, EVENT_NAME=FTE, EVENT_ID=1,
TERMINAL_ID=PXITRG1"
```

NDS implements following classes to support event records for time events:

- TimeEvent (ndsTimeEvent.h)
- Pulse (ndsPulse.h)
- Clock (ndsClock.h)

Concrete NDS device support should override methods of these classes to implement the event's functionality.

6.2. TimeStamping interface

6.2.1. Implementation

Sequence diagram of timestamping is represented on Figure 13. When next buffer is ready to be populated within a channel (DAQChannel on the Figure 13) actor is calling bufferXXXProcessed() function. XXX represents standards asyn array types here:

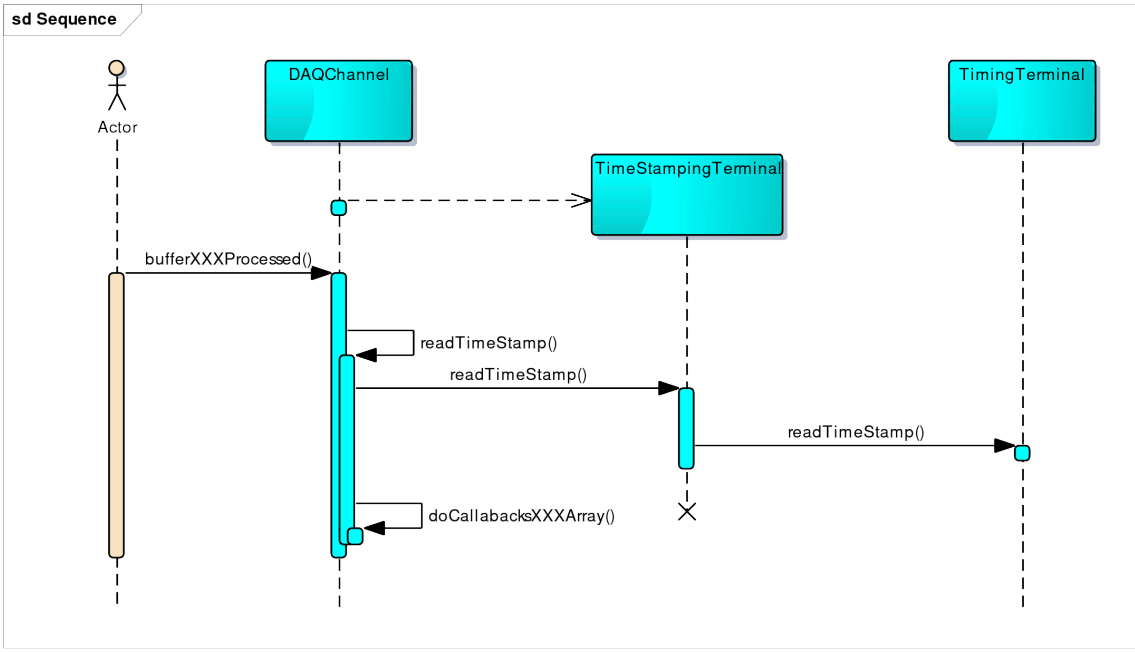

Figure 13: Timestamping sequence

```
ndsStatus bufferInt8Processed(epicsInt8 *value, size_t nElements,
       ndsStatus status=ndsSuccess);

ndsStatus bufferInt16Processed(epicsInt16 *value, size_t
       nElements, ndsStatus status=ndsSuccess);

ndsStatus bufferInt32Processed(epicsInt32 *value, size_t
       nElements, ndsStatus status=ndsSuccess);

ndsStatus bufferFloat32Processed(epicsFloat32 *value, size_t
       nElements, ndsStatus status=ndsSuccess);

ndsStatus bufferFloat64Processed(epicsFloat64 *value, size_t
       nElements, ndsStatus status=ndsSuccess);
```

bufferXXXProcessed() invokes internal **readTimeStamp()** method which must implement concreate timestamp functionality. By default (see Figure 13) this method is calling **readTimestampMethod()** of a **TimeStampingTerminal** instance which is a private field of a **nds::Channel**. If **TimeStampingTerminal** is configured it will read timestamp from required **TimingTerminal** otherwise will return ndsError.

Developer of concrete device support is interested in readTimeStamp() method of a channel. This method is responsible to obtain accurate timestamp what is valid for both DAQ and Timing channels.

In case of Timing channel readTimeStamp() method must return best available latest times-

tamp for this timing terminal. The method **must be implemented in thread safe manner**, as this method can be called directly from other threads, running in other device support modules that are configured to read time stamps from this terminal. It is advisable to read the time stamps from the device in a separate thread and in that thread update the latest time stamp from the terminal continuously. The latest time stamp for the terminal is then returned by the implemented `readTimeStamp` method. The Timing Channel's `readTimeStamp` method must be **thread safe**, **fast** and **must not interact with the device**.

In case of DAQ channel this methods obtains timestamp from relevant timing terminal, through NDS layer, and performs required calculation. Two cases are possible : 1. The Timing device provides the exact time stamp that the DAQ device needs 2. The DAQ Device's channel must perform certain calculations to compute the correct time stamp. For example : the timing device provides the time stamp for the beginning of the sampling with certain frequency, and the DAQ device must calculte the correct time stamp from the index of the sample and the initial time stamp

The pre-requisite for this is that the DAQ and Timing device are configured to work in a NDS DAQ System.

In both cases, the time stamping terminal / line should be correctly configured in the `TTRM` EPICS record of the DAQ channel. For correct configuration, see the DAQ System Extended Functionality chapter.

In the first case this is all the developer should do. The default implementation of `Channel ::readTimeStamp` already does the correct thing : obtains the time stamp from the timing device.

In the second case, the DAQ Device's Channel class should, in its implementation of `readTimeStamp` method, first obtain the time stamp from the Timing device by calling `Channel::readTimeStamp` (parent class's `readTimeStamp`), and then modify the obtained time stamp to account for the offset given by the index of the sample for example.

7. DAQ System

7.1. DAQ System architecture

DAQ system consists at least from DAQ device support and Timing device support libraries. These 2 libraries must be linked in one EPICS application to support internal NDS time stamping interface.

Chapter provides example of DAQ system with dummy DAQ and timing devices. Next 2 sections describes in details how to instantiate DAQ and timing dummy devices from EPICS templates. If it is required to create a DAQ System with real devices, there is no need to instantiate dummy devices, so next 2 sections could be skipped and developer can immediately start from the section 7.4.

7.2. DAQ support library

Instantiate dummy DAQ device support from NDS template.

```
mvn iter:newunit -Dunit=m-epics-pipeexample
cd m-epics-pipeexample
mvn iter:newapp -Dapp=pipeexample -Dtype=nds
```

Configure packaging

```
<configuration>
  <packaging>
    <package name="lib">
      <include type="file"
          source="main/epics/lib"
          target="epics/modules/pipeexample/lib" />

      <include type="file"
          source="main/epics/db"
          target="epics/modules/pipeexample/db" />

      <include type="file"
          source="main/epics/dbd"
          target="epics/modules/pipeexample/dbd" />
    </package>
  </packaging>
</configuration>
```

Compile and install device support library.

7.3. Timing device support

Instantiate dummy timing device support library.

```
mvn iter:newunit -Dunit=m-epics-timing
cd m-epics-timing/
mvn iter:newapp -Dapp=timing -Dtype=ndsTime
```

Configure packaging

```
<configuration>
  <packaging>
    <package name="lib">
      <include type="file"
          source="main/epics/lib"
          target="epics/modules/timing/lib" />

      <include type="file"
          source="main/epics/db"
          target="epics/modules/timing/db" />

      <include type="file"
          source="main/epics/dbd"
          target="epics/modules/timing/dbd" />

    </package>
  </packaging>
</configuration>
```

Compile and install device support library.

7.4. Building DAQ System application

Instantiating DAQ System application

```
mvn iter:newunit -Dunit=m-epics-daqsys
cd m-epics-daqsys/
mvn iter:newapp -Dapp=daqsys
mvn iter:newioc -Dioc=daqsys -Dapp=daqsys -Dtype=nds
```

Edit source Makefile to include compiled and installed libraries:

Note: Example shows case when 2 dummy devices are used. In case real devices are used name of files must be fixed accordingly.

```
daqsys_DBD += timing.dbd
daqsys_LIBS += timing

daqsys_DBD += pipeexample.dbd
daqsys_LIBS += pipeexample
```

These lines make sure that required devices are included to DAQ Sytem.

DAQ System application must be configured to initialize required devices. Edit st.cmd file to add DAQ device support initialization:

```
ndsCreateDevice "ndsPipeExample", "ndsTestPIPE",
       "FILE=/tmp/q,N_AI=2,N_AO=3,N_DI=4,N_DO=5,N_DIO=6,N_IMAGE=7"

dbLoadRecords "db/pipeexample.db", "PREFIX=PIPE, IDX=0, MODULEIDX=0,
       ASYN_PORT=ndsTestPIPE, TIMEOUT=1"
```

Add timing device support initialization:

```
ndsCreateDevice "timing", "ndsTiming", "TRG=2"

## Load record instances
dbLoadRecords "db/timing.db", "PREFIX=TM, IDX=0, MODULEIDX=0,
       ASYN_PORT=ndsTiming, TIMEOUT=1"
```

Note: Example shows configuration of dummy devices. For real devices these lines must be fixed accordingly.

After DAQ system is compiled it is ready to be used.

7.5. DAQ System Extended Functionality

Deploying two *NDS* built device devices on the same EPICS IOC allows for the devices to extend their functionality beyond the functionality available to one device alone.

In particular, it is possible for one device to obtain timestamps from another device. A common usage of a *NDS* DAQ System would be to deploy two devices, an Timing device and an DAQ device on the same IOC and by doing this allow the devices to exchange time stamps through *NDS* layer.

This functionality might be necessary when the DAQ device needs to obtain the timestamps from another device (Timing device), faster than EPICS can provide because of high throughput of samples needing a timestamp, low latency requirements or because the device is publishing the data to a high performance network.

7.5.0.1. Exchanging Time Stamps Through NDS Layer

The configuration that allows the device to obtain the time stamps from another device is performed during runtime, through EPICS. A common use case is to make the DAQ device obtain the time stamps form a timing device.

The configuration set with

```
$ caput DAQ-AI0-TTRM `pfi0`
```

configures the DAQ device's channel 0 to obtain the time stamps from the `pfi0` backplane line. The timing device, hosted on the same IOC, should have a channel group named `pfi`. The channel 0 of group `pfi` will provide the timestamps.

The time stamps obtained from the source configured in this way can be modified in the device support layer of the DAQ device, to, for instance, account for index of the sample in case the timing device only provides the time stamp of the first sample in the series.

After any possible device-support level modifications, the time stamp is available in EPICS.

8. *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).

8.1. Developing an *NDS* Plugin

To develop an NDS Plugin, it is advisable to start from the provided *NDS* plugin template. To create a CODAC project using the *NDS* plugin template, use the commands below and substitute the generic names for the module (`m-epics-dummkplugin`, `ndsDummyPlugin`) with the names for the plugin that you are developing.

```
$ mvn newunit -Dunit=m-epics-dummyplugin
$ cd m-epics-dummyplugin/
$ mvn newapp -Dapp=ndsDummyPlugin -Dtype=ndsPlugin
$ mvn include -Dtype=module -Dname=ndsDummyPlugin
```

When implementing an *NDS* Plugin, keep in mind that the plugin is an *NDS* Device, so all standard *NDS* functionality is available in the plugin classes.

The following chapters highlight the added functions that the plugin implementation can use to implement the functionality as a plugin device.

8.1.1. *NDS* Plugin Device

The *NDS* Plugin Device implementation class has to inherit from the `PluginDevice`. A sample definition (from the header) is provided below.

```
class ndsDummyPluginDevice : public nds::PluginDevice
{
public:
  /// Constructor of the Device class
  ndsDummyPluginDevice(const std::string &name);
};
```

At Plugin Device construction a channel creation function must be provided to the `PluginDevice` constructor. This function is then internally used by `AutoChannelGroup` to instantiate the plugin channels. Example of the plugin creation function is provided below.

```
nds::Channel *createndsDummyPluginChannel()
{
  return new ndsDummyPluginChannel();
}

ndsDummyPluginDevice::ndsDummyPluginDevice(const std::string &name) :
  nds::PluginDevice(name, "inv", boost::bind(&createndsDummyPluginChannel))
{}
```

8.1.2. *NDS* Plugin Channel

NDS Plugin Channel implements the plugin specific data processing logic.

Each plugin channel is assigned a data source through `SRC` PV. Data source can be any array record on an arbitrary device, channel group or channel. How exactly the data sources are specified is better described in section 8.2.

NDS Plugin Channel inherits from *NDS* Channel and adheres to the same state machine states and transitions. *NDS* Plugin Channel is processing the data from the device only when it is in the `PROCESSING` state. It attaches to the target data source when it transfers to the `PROCESSING` state, and detaches when it leaves this state.

To perform use-case specific tasks before / after the plugin is attached / detached to / from the target channel the following functions of `PluginChannel` can be overridden:

- Invoked before the plugin is registered to the source:

  ```
  virtual ndsStatus preRegisterPlugin();
  ```

- Invoked after the plugin is registered to the source:

  ```
  virtual void afterRegisterPlugin();
  ```

- Invoked before the plugin is unregistered from the source:

  ```
  virtual void preDeRegisterPlugin();
  ```

- Invoked after the plugin is unregistered from the source:

  ```
  virtual void afterDeRegisterPlugin();
  ```

For example in ITER DAN plugin, the DAN stream is initialized in the `preRegisterPlugin` function and closed in the `afterDeRegisterPlugin` function.

To process the data published by the attached channel, the following functions of the `PluginChannel` can be overridden:

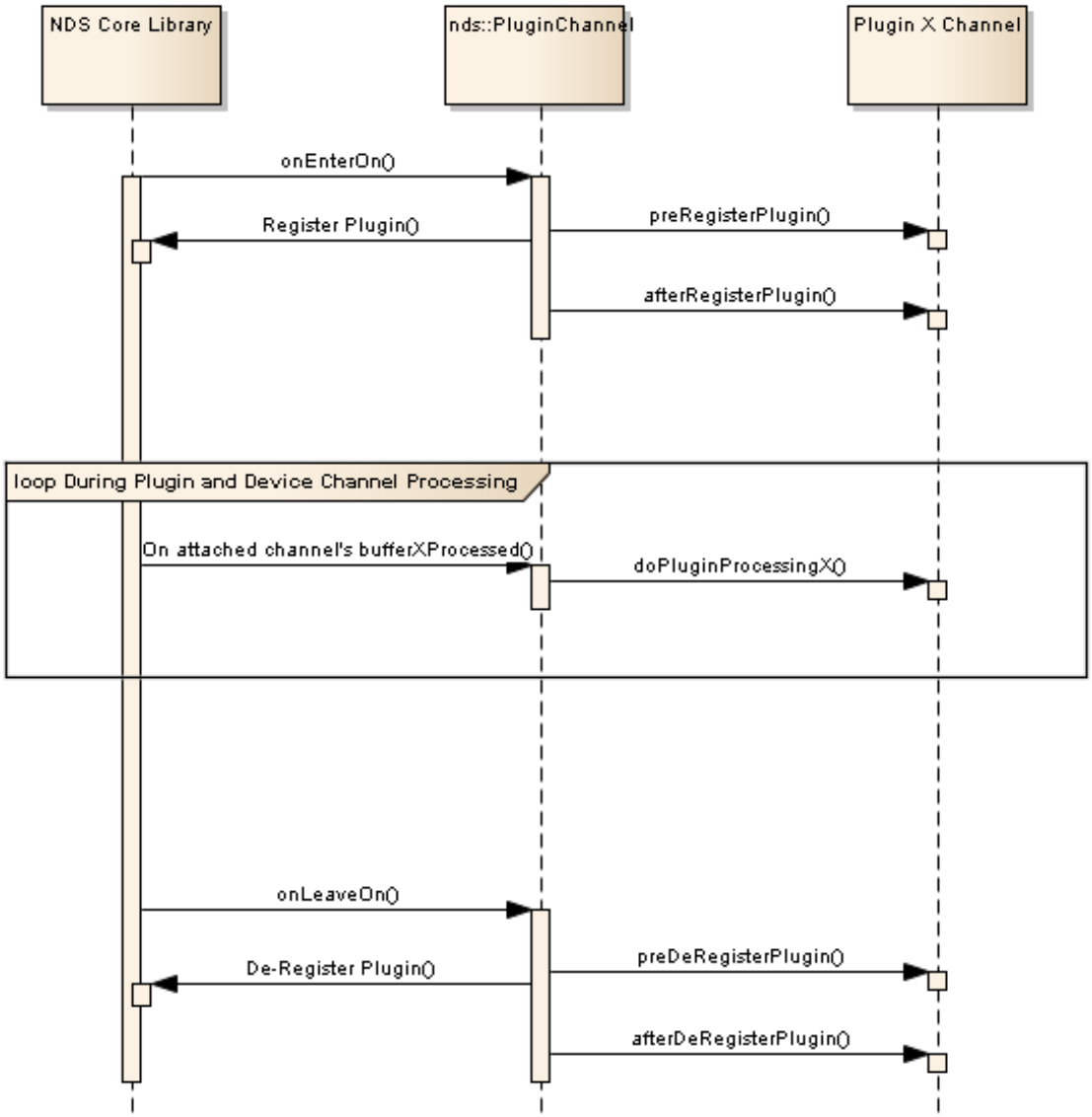

Figure 14: *NDS* Plugin Channel functions as called by *NDS*

- Invoked when data source of `epicsInt8` array type is ready for processing:

```
virtual ndsStatus doPluginProcessingInt8Array(epicsTimeStamp timeStamp,
    epicsInt8 const *value, size_t nElements, ndsStatus status);
```

- Invoked when data source of `epicsInt16` array type is ready for processing:

```
virtual ndsStatus doPluginProcessingInt16Array(epicsTimeStamp timeStamp,
    epicsInt16 const *value, size_t nElements, ndsStatus status);
```

- Invoked when data source of `epicsInt32` array type is ready for processing:

```
virtual ndsStatus doPluginProcessingInt32Array(epicsTimeStamp timeStamp,
    epicsInt32 const *value, size_t nElements, ndsStatus status);
```

- Invoked when data source of `epicsFloat32` array type is ready for processing:

```
virtual ndsStatus doPluginProcessingFloat32Array(epicsTimeStamp timeStamp,
    epicsFloat32 const *value, size_t nElements, ndsStatus status);
```

- Invoked when data source of `epicsFloat64` array type is ready for processing:

```
virtual ndsStatus doPluginProcessingFloat64Array(epicsTimeStamp timeStamp,
    epicsFloat64 const *value, size_t nElements, ndsStatus status);
```

Other data source types are currently not supported. Only the function matching the data source type will be invoked. Plugin can implement all functions or only some of them depending on what type of data can be expected.

The following functions for accessing data source information are provided by the `PluginChannel` base class:

- Data source name (corresponds to the value of `SRC` PV):

```
const std::string& nds::PluginChannel::getDataSourceName() const;
```

- Data source type:

```
ndsDataType nds::PluginChannel::getDataSourceType() const;
```

- Size of the data source (i.e. the number of data source typed elements):
  - for channel buffers, it is the value assigned to the `_bufferXXXSize` variable
  - for other arrays, this is the number of array elements when provided with `NDS_PV_REGISTER_XXXARRAY()` or -1 otherwise
  - for simple types it is always 1

```
nds::PluginChannel::size_t getDataSourceSize() const;
```

  To calculate the actual size in Bytes, data source size needs to be multiplied by the size of data type element (`getDataSourceSize()*ndsElementSize(getDataSourceType())`).
- Device providing the data source or `NULL` if data source is not assigned:

```
Device* nds::PluginChannel::getDataSourceDevice() const;
```

- ChannelGroup providing the data source or `NULL` if data source is not provided by channel or channel group:

```
ChannelGroup* nds::PluginChannel::getDataSourceChannelGroup() const;
```

- Channel providing the data source or `NULL` if data source is not provided by channel:

```
Channel* nds::PluginChannel::getDataSourceChannel() const;
```

8.2. *NDS* Plugin Data Sources

As mentioned in the previous section, each *NDS* plugin channel is assigned a data source.

Data source can be any array record on an arbitrary device, channel group or channel, registered with one of the `NDS_PV_REGISTER_XXXARRAY` macros (see section 3.7.1). Data source is specified through the plugin channel's `SRC` PV 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 specified as the first argument of the `NDS_PV_REGISTER_XXXARRAY` macro.

8.3. *NDS* Plugin Integration with Other *NDS* Devices

8.3.1. Exposing a channel buffer data source

Properly written *NDS* devices fulfilling the requirements mentioned later in this section, already natively expose their channel buffers to be used as *NDS* plugin data sources.

Existing devices should already make use of the `bufferXXXProcessed` functions to signal when *NDS* channel buffer is ready for EPICS processing (see section 6.2).

Before actually triggering the EPICS processing, these functions first invoke the corresponding `dataProcessed<XXX>` function to allow the channel buffer data to also be processed by the plugins. It is even possible to completely bypass the EPICS processing and only have the plugins do their part by setting the `EPI` PV to `DISABLED`.

For *NDS* plugins it is important that the data source channel defines the buffer type and (depending on the plugin) also the buffer size. It is expected that every *NDS* device implementation in it's channel constructor specifies the buffer type by invoking `setChannelDataType` function and assigns the buffer size (the number of elements) to appropriate `_BufferXXXSize` variable, e.g.:

```
ndsDummyDaqDeviceChannel::ndsDummyDaqDeviceChannel(...) : nds::Channel(), ...
{
    setChannelDataType(CHANNEL_DATA_TYPE_FLOAT32_ARRAY);
    _BufferFloat32Size = DUMMY_FLOAT32_BUFFER_SIZE;
    ...
}
```

If the device support is expected to publish a lot of data to be later processed by the attached plugins, it might be necessary to disable EPICS processing during runtime. For example, when publishing a lot of data to a scientific data archive (i.e. ITER DAN network).

To facilitate such use case EPICS publishing can be disabled by changing the value of the `EPI` PV to `DISABLED` on all channels that do not need to publish the data to EPICS.

8.3.2. Exposing an arbitrary array data source

Arbitrary array records (simple types are currently not supported) require to be explicitly exposed as *NDS* plugin data sources from the *NDS* device implementation.

First of all, in order for any data to be referable as a data source, a record needs to be registered with the `NDS_PV_REGISTER_XXXARRAY` macros (see section 3.7.1). Such record may or may not be linked with an actual EPICS PV. If it has no EPICS PV representation, the record will only be usable for the plugin processing with `dataProcessed<XXX>`. In the opposite case, in order to trigger the EPICS processing, `doCallback` function will need to be invoked independently (see section 3.7.4).

For example, let's consider a data source that is specified as `pxi6368-0.ai0[CData]`. Here `pxi6368-0` is the device name, `pxi6368-0.ai` is the channel group name and `pxi6368-0.ai0` is the first channel on the `pxi6368-0.ai` channel group. `CData` is an array record registered on the channel.

In the *NDS pxi6368* device channel implementation class, `registerHandlers` would be overridden to register the `CData` record as follows:

```
NDS_PV_REGISTER_INT8ARRAY("CData", NULL, NULL, &channelDataBufferInt8ID,
    channelDataBufferSize);
```

In this example `CData` does not provide any getter or setter function, meaning that it is not represented by any EPICS PV; this record will be used exclusively for communication with plugins. In order to trigger the processing of `CData`'s buffer `channelData` on the plugins, the *pxi6368* device channel needs to invoke:

```
dataProcessed<epicsInt8>(channelDataBufferInt8ID, (epicsInt8*)&channelData,
    channelDataBufferSize, ts_current, ndsSuccess);
```

Where `ts_current` is the EPICS timestamp, `channelData` is the buffer structure prepared for plugin processing, `channelDataBufferSize` corresponds to `sizeof(channelData)` and is the same value as used with `NDS_PV_REGISTER_INT8ARRAY`. `channelDataBufferInt8ID` is the handler obtained when the record was registered with `NDS_PV_REGISTER_INT8ARRAY`.

9. SDD Integration Of NDS Device Support

To integrate an NDS based EPICS Device Support Module into ITER SDD Editor the document [RD12] should be referred to as primary source of information.

This chapter focuses on NDS specifics of integration of EPICS device support modules, and aims not to duplicate any information in [RD12].

For reference, a developer can use the `ndsDAN` SDD integration, available from ITER SVN

```
$ svn co https://svnpub.iter.org/codac/iter/codac/contracts/2015/DNFM_UseCase/
    System/trunk/m-epics-nds-dan/src/main/sdd/ ./sdd-sample
```

When referring to the NDS DAN device's SDD integration keep in mind that the templates provide SDD integration of a DAN plugin device, therefore it would be better to build the Hardware description xml from a device that more closely resembles the device you are working with from the hardware perspective.

9.1. I/O Module Description

The Hardware description SDD xml does not have any NDS specific information, as it focuses on the hardware description.

9.2. EPICS Device Support

EPICS Device Support for all NDS based devices share quite a few common properties. The document [RD12] provides the technical information on the meaning of the following xml tags, this chapter will focus on practical implementation.

As all NDS based device supports depend on NDS library and its dependencies, the following information must be in the device support template :

```
<libraryName>asyn</libraryName>
<libraryName>nds</libraryName>
<libraryName>ndsXXX</libraryName>
```

where `ndsXXX` is the name of the device support to be integrated into SDD.

The `INP` and `OUT` links syntax in EPICS records is very similar across all NDS device support implementations, as *NDS* provides a common interface between EPICS records and the device support layer.

The provided commands are also similar across all *NDS* based device supports.

For functionality that is obtained in the device support by deriving of the implementation classes from the NDS base classes, the configuration can be copied from the `ndsDAN`'s SDD integration files.

9.3. EPICS Templates

Because of the similarity of EPICS templates for NDS based device support modules, the records providing the common functions can be copied from the `ndsDAN`'s SDD integration files. Keep in mind to adopt the integration files to your specific hardware.

The `ndsDAN` is a plugin device, and as such does not have any hardware-specific limits on its processing, values it can write to the hardware, or any limitations that originate from the hardware.

A. Signal List

This appendix lists all configuration parameters that a nominal device can provide.

The first section lists the configuration parameters related to a Device and the second section lists the configuration parameters related to a channel.

The function describing tables contain the following columns:

- The first column, *Function*, specifies the name of the function within the driver which is called when the value of the record is set. By our convention, this name corresponds to the string that is used in the INP or OUT field of the record. For example, the EPICS database would have the following definition for the value of these fields:

```
field(INP, "@asyn($(port),$(addr)) Function")
```

- The *PV suffix* column defines the suffix of the EPICS process variable name. If a suffix is given in parentheses – e.g., (WF) – it is actually blank, but is stated in the table just for purpose of cross-referencing throughout this document.
- The *Record type* column specifies the record type in the EPICS database.
- Finally the *Version* column specifies in which NDS version this function was first supported.

Not all devices are required to support all of the functions and configuration parameters. For those that are not optional, the *Function* is followed by [R]. Also, if some configuration parameters only apply to a certain kind of channel, the abbreviations of channel types are listed in the brackets, e.g., [R: AI, C] means that the parameter is required for all analog input and camera channels.

A.1. Device Functions

Function	Get/Set	PV suffix	Record type	Version
State [R]	get	(STAT)	mbbi	1.0
Model [R]	get	IMDL	stringin	1.0
Serial	get	ISN	stringin	1.0
HardwareRevision	get	IHW	stringin	1.0
FirmwareVersion	get	IFW	stringin	1.0
SoftwareVersion [R]	get	ISW	stringin	1.0
Message	get	MSGR	waveform	1.0
Message	set	MSGS	waveform	1.0
Emulation	get/set	EMUL	bo	1.0
Status	get	STS	mbbi	1.0
DeviceEnabled [R]	get/set	ENBL	bo	1.0

Table 3: Generic device functions.

A.1.1. Device function descriptions

A.1.1.1. State

The state of the device. The following values are possible:

OFF (0) The device is offline – even powered off, if possible by the hardware. To perform power cycling, first set the device state to OFF, and then back to ON.

ON (1) The device is ready to perform. Note: setting the state to ON might not be immediate. If device initialization is not immediate, the device will pass through the INIT state.

RESET (2) Reset the device. When set, a soft reset of the device will be performed. If device takes longer time to reset, it might pass through the INIT state.

INIT (3) The device is initializing. When initialization is complete, it will go to the ON state. All settings will be reset to initial/default values. The device is in the INIT state also when it is temporarily unavailable e.g., due to a firmware update that is in progress. It is not possible to explicitly set the PV to this value.

ERROR (4) A device-level error occurred. The device needs to be reset.

DEFUNCT (5) Hardware malfunction. Not even a reset/power-cycle would help and the device needs to be serviced or replaced. This state can also be a consequence of a broken firmware update. In the ERROR and DEFUNCT states, alarm severity (SEVR

field) is set to MAJOR and alarm state (STAT) is set to STATE. Additional information about the error can be found in the IOC's error log.

A.1.1.2. Model

Device model name.

A.1.1.3. Serial

Device serial number.

A.1.1.4. HardwareRevision

Device hardware revision.

A.1.1.5. FirmwareVersion

Device firmware version.

A.1.1.6. SoftwareVersion

Device software version. Contains version of NDS and device support concatenated by -, e.g., NDS 2.3.1 and Device Support 8.0 would be `2.3.1-8.0`. May also contain additional version strings, also separated by -.

A.1.1.7. Message

Get Retrieve the last message from the device. The message is of the format:

```
<message> <correlation-id>
```

Set Send a message to the device. Message string has the format:

```
<message> [<correlation-id> [<param> [<param>] ...]]
```

where `<message>` is the type of the message to convey to the device. `<correlation-id>` is used to correlate a response message with the request message. A number of parameters can then be given. Separator is a space. If parameter value contains a space, the parameter needs to be enclosed in quotes.
Standardized messages are:

TEST Perform self-test of the device.
LIST List all supported messages.

ON Switch device to ON state.
OFF Switch device to OFF state.
RESET Reset the device.

If during a test, problems are found, the state of the device (STAT record) is set to the ERROR or DEFUNCT state. More information can be found in the IOC error log.

Individual devices may implement their own device specific messages. For more information refer to device documentation.

A.1.1.8. Emulation

TRUE (0) Allow software emulation of functions that are not supported by hardware.
FALSE (1) Disallow software emulation.

A.1.1.9. Status

Device status:

No error (0)
Initializing (1)
Resetting (2)
Hardware error (3)
Brd unavailable (4)
Static conf error (5)
Dync conf error (6)
Reserved (7)
FIFO overflow (8)
FPGA not ready (9)
Ref clk no lock (10)
Buffer overflow (11)

A.1.1.10. Enabled

Device operation status:

DISABLED (0)
ENABLED (1)

A.2. Channel Functions

Function	Get/Set	PV suffix	Record type	Version
Buffer [R: AI, C, DI, DO, DIO]	get/set	(WF)	`waveform`	
BufferSize [R: AI, C, DI, DO, DIO]	get	BUFS	`longout`	
ChannelState [R]	get	STAT	`mbbo`	
Enabled [R]	get/set	ENBL	`bo`	
Message	get	MSGR	`waveform`	
Message	set	MSGS	`waveform`	
Quality	get	Q	`ai`	
Value	get/set	OUT	`ao, bo`	
ValueFloat64	get/set	IN	`ai, bi`	

Table 4: Generic channel functions

A.2.1. Channel function Descriptions

A.2.1.1. Buffer

- Output channels: the contents of the buffer to output to the channel. The values are subject to conversion from engineering units to raw units.
- Input channels: retrieve the next buffer. The NORD field specifies how many samples are read. The NELM field specifies the maximal number of samples.

A.2.1.2. BufferSize

Size of the buffer. Used to set the NELM field of the (WF) record.

A.2.1.3. ChannelState

Get the state of the channel. The following values are possible:

OFF (0) The channel is disabled. In this state, no operation can be performed on the channel, and the state must first be set to ON.

ON (1) The channel is ready to be configured. This is the initial state of the channel.

RESET (2) Reset the channel. When set, the channel will be configured to its default configuration. When reconfigured, its state will be set to ON.

BUSY (3) Channel is busy (acquiring analog input or frames, or generating output). To stop the activity, state can be set to ON.

ERROR (4) A channel-level error occurred. The channel needs to be reset.

DEFUNCT (5) Hardware malfunction of the channel. Not even a reset of the channel or power-cycle of the device would help. Channel needs to be re-cabled or replaced. If any channel is DEFUNCT, the device state is also set to DEFUNCT.

A.2.1.4. Enabled

Sets whether the channel should be processed or not when the channel group which it belongs to is being processed. It is used to select which channels should be active during an acquisition.

Can be set to:

DISABLED (0) This channel will not change to `PROCESSING` if the parent channel group changes to `PROCESSING`.

ENABLED (1) This channel will change to `PROCESSING` if the parent channel group changes to `PROCESSING`.

A.2.1.5. Message

Description for the device messages (A.1.1.7) also applies to channels.

A.2.1.6. Quality

Quality of the measurements on the channel (e.g., measure of signal-to-noise).

A.2.1.7. Value

Set a value of the output channel. The value is subject to conversion from engineering units to raw units.

A.2.1.8. ValueFloat64

Read the current value of the channel. The value is subject to filtering and conversion from raw units to engineering units.

A.3. Triggering functions

Function	Get/Set	PV suffix	Record type	Version
TriggerChannel	get/set	TRGC	waveform	

Function	Get/Set	PV suffix	Record type	Version
TriggerDelay	get/set	TRGD	ao	
TriggerRepeat	get/set	TRGR	longout	
Trigger	get/set	TRG	bo	

A.3.1. Function Descriptions

A.3.1.1. TriggerChannel

Set another channel as the trigger.

A.3.1.2. TriggerDelay

Set the delay for the trigger. The delay is given as the number of seconds past the trigger event.

A.3.1.3. TriggerRepeat

Set the number of times that the trigger should be triggered.

A.3.1.4. Trigger

Trigger data acquisition immediately.

A.4. Filtering functions

Function	Get/Set	PV suffix	Record type	Version
Filter	get/set	FILT	waveform	
LowPass	get/set	LOWP	ao	
HighPass	get/set	HGHP	ao	
FilterType	get/set	FT	mbbo	
Smoothing	get/set	SMOO	ao	

A.4.1. Function Descriptions

A.4.1.1. Filter

Set the filter's polynomial coefficients.

A.4.1.2. LowPass

High cut-off border of bandwidth of a low pass filter.

A.4.1.3. HighPass

Low cut-off border of bandwidth of a high pass filter.

A.4.1.4. FilterType

Atomic function to enable/disable filtering. The available options are:

DISABLED (0)
SIMPLE (1)
POLYNOMIAL (2)
SMOOTHING (3)

A.4.1.5. Smoothing

Coefficient of smoothing filter. It is an infinite impulse response (IIR) filter with $a_0 = \texttt{SMOO}$ and $b_0 = (1 - \texttt{SMOO})$.

A.5. FFT functions

Function	Get/Set	PV suffix	Record type	Version
FFTBuffer	set	FFT	waveform	
FFTOverlap	get/set	FFTO	ao	
FFTSize	get/set	FFTN	ao	
FFTSmoothing	get/set	FFTS	ao	
FFTWindow	get/set	FFTW	mbbo	

A.5.1. Function Descriptions

A.5.1.1. FFTBuffer

The Fourier transform of the acquired signal.

A.5.1.2. FFTOverlap

Configure the amount of overlap between consecutive frames for the Fourier transform.

A.5.1.3. FFTSize

Configure size of the frame for the Fourier transform.

A.5.1.4. FFTSmoothing

Configure the smoothing factor for averaging the Fourier transform.

A.5.1.5. FFTWindow

Select the windowing function for the Fourier transform. Possible values are:

NONE (0) (No window, i.e., $\mathrm{window}[i] = 1.$)
BARLETT (1)
BLACKMAN (2)
FLATTOP (3)
HANN (4)
HAMMING (5)
TUKEY (6)
WELCH (7)

A.6. Signal generation functions

Function	Get/Set	PV suffix	Record type	Version
SignalFrequency	get/set	SGNF	ao	
SignalOffset	get/set	SGNO	ao	
SignalPhase	get/set	SGNP	ao	

Function	Get/Set	PV suffix	Record type	Version
SignalType	get/set	SGNT	mbbo	
Spline	get/set	SPLN	waveform	

A.6.1. Function Descriptions

A.6.1.1. SignalFrequency

Frequency of those types of signal that are periodic. See the SGNT record for specification of the signal type. Frequency is specified in Hz as a floating point number.

A.6.1.2. SignalOffset

Offset to add to the output signal, e.g., the sine signal would range from SGNO-SGNA to SGNO+SGNA. Offset is specified in terms of the output's engineering units.

A.6.1.3. SignalPhase

Phase of the generated output signal. Phase is relative to the epoch of the time base (e.g., the TCN).

A.6.1.4. SignalType

Set the type of output signal that is generated by the output channel. It can be one of the following values:

WAVEFORM (0) Waveform as defined in WF record.
SPLINE (1) Spline (piecewise-cubic function) as defined in SPLN record
SIN (2) Sine curve (amplitude SGNA, offset SGNO, frequency SGNF, phase SGNP).
PULSE (3) Pulse train (amplitude SGNA, offset SGNO, frequency SGNF, phase SGNP, duty cycle SGNP).
SAWTOOTH (4) Sawtooth output (amplitude SGNA, offset SGNO, frequency SGNF, phase SGNP, rising for fraction SGNP, falling for fraction 1-SGNP).

A.6.1.5. Spline

Description of a spline to generate on the output signal.

A.7. ADIO Channel Functions

Function	Get/Set	PV suffix	Record type	Version
Bandwidth	get	BW	ai	
ClockFrequency	get/set	CLKF	ao	
ClockMultiplier	get/set	CLKM	ao	
ClockSource	get/set	CLKS	mbbo	
Conversion	get/set	CVT	waveform	
Coupling	get/set	ACDC	bo	
DecimationFactor	get/set	DECF	ao	
DecimationOffset	get/set	DECO	ao	
Differential	get/set	DIFF	bo	
DIODirection [R:DIO]	get/set	DIR	bo	
Gain	get/set	GAIN	ao	
Ground	get/set	GND	bo	
Impedance	get	IMPD	ai	
MeasuredOffset	get	MOFS	ai	
SignalAmplitude	get/set	SGNA	ao	
SignalDutyCycle	get/set	SGND	ao	

Table 9: ADIO Channel Functions.

A.7.1. Function Descriptions

A.7.1.1. Bandwidth

Get the bandwidth of an analog input channel.

A.7.1.2. ClockFrequency

Set the frequency of the internal clock selected with CLKSRC record. The frequency is specified in Hz.

For camera channels, this parameter specifies the number of frames per second.

If a clock source other than an internal clock source is selected, or if the selected clock source does not support the requested frequency, the alarm severity (SEVR field) of this record is set

to MINOR and the alarm state (STAT) is set to WRITE. If internal clock source is selected, the frequency is set to the nearest smaller frequency.

A.7.1.3. ClockMultiplier

Set the clock multiplier of the external clock selected with CLKSRC record. The multiplier is given as a factor, e.g., 1.0 to keep the external source's frequency, 0.25 to divide it by 4, and 4 to multiply it with 4, etc.

If a clock source other than an external clock source is selected, or if the selected clock source does not support the requested multiplier factor, the alarm severity (SEVR field) of this record is set to MINOR and the alarm state (STAT) is set to WRITE. If external clock source is selected, the multiplier is set to the nearest smaller multiplier.

A.7.1.4. ClockSource

Set the clock source for sampling (input channel), frame acquisition (camera) or clocking digital-to-analog converter (output channel).

Possible values for the clock source are:

INT (0) Internal clock. [REQUIRED]
INT1 (1) Second internal clock.
INT2 (2) Third internal clock.
INT3 (3) Fourth internal clock.
TCN (4) External clock. Usually from the timing module.
EXT1 (5) Second external clock.
EXT2 (6) Third external clock.
EXT3 (7) Fourth external clock.

If the channel does not support a particular clock source, the alarm severity (SEVR field) of this record is set to MINOR, the alarm state (STAT) is set to WRITE, and the clock source remains unaffected.

A.7.1.5. Conversion

Define conversion from raw values to engineering values. Conversion is defined as a segmented cubic spline. Waveform array consists of five-tuples, whose elements are the start of the segment, followed by the four cubic spline coefficients.

If an empty array is given, conversion is performed according to the LINR, EGUF and EGUL fields of the IN and OUT records.

A.7.1.6. Coupling

Set the coupling (AC or DC) of the analog input channel. Values are:

AC (0) Coupling for alternating current.
DC (1) Coupling for direct current.

A.7.1.7. DecimationFactor

Set decimation factor.

For input channels, only every DECF-th sample (or frame) will be sampled.

For output channels outputting a waveform, only every DECF-th sample will be output.

A.7.1.8. DecimationOffset

Specify the index of the first sample that is not decimated.

A.7.1.9. Differential

Set the type of input: differential or single-ended.

A.7.1.10. DIODirection

For general purpose input/output digital channels, specifies whether the channel is an input or an output channel. Values are:

IN (0) Input channel.
OUT (1) Output channel.

A.7.1.11. Gain

Gain of a channel.

For analog input channels, this is the amplification factor by which the signal is amplified before it is digitized.

For analog output channels, this is the amplification factor by which the signal is amplified just after it has been converted to analog.

If the channel does not support a particular gain factor, the alarm severity (SEVR field) of this record is set to MINOR and the alarm state (STAT) is set to WRITE. The gain is set to the nearest supported smaller value.

A.7.1.12. Ground

Specify whether or not to ground the input.

A.7.1.13. Impedance

Get the impedance of the analog input channel.

A.7.1.14. MeasuredOffset

Return the measured offset of an analog input channel.

A.7.1.15. SignalAmplitude

Amplitude of the output signal, e.g., the sine signal without offset would range from –SGNA to +SGNA. Amplitude is specified in terms of the output's engineering units.

A.7.1.16. SignalDutyCycle

Set the duty cycle of the output signal. Duty cycle is expressed as a fraction of the whole cycle when the output is in the *high* state (pulse train) or rising (sawtooth).

A.8. Timing Device's Functions

Function	Get/Set	PV suffix	Record type	Version
SyncStatus	get	SYNC	mbbi	
SecsSinceSync	get	SYNCLOST	longin	
MaxScheduledFtes	get	FTEMAX	longin	
NumPendingFtes	get	FTENUM	longin	
CheckFteLevels	get	FTEERRS	longin	
AbortAllFtes	set	FTEAALL	bo	
Time	get	BDTM	waveform	
TimeText	get	HBDTM	stringin	
OriginTime	set/get	O	longin	

A.8.1. Function Descriptions

A.8.1.1. SyncStatus

Synchronization status:

NOT_SYNCED (0)
SYNCING (1)
SYNCED (2)
LOST_SYNC (3)

A.8.1.2. SecsSinceSync

Seconds since lost synchronization

A.8.1.3. MaxScheduledFtes

Max number of scheduled FTEs

A.8.1.4. NumPendingFtes

Number of pending FTEs

A.8.1.5. CheckFteLevels

Check number of FTE level errors

A.8.1.6. AbortAllFtes

Abort all pending FTEs

A.8.1.7. Time

Board time

A.8.1.8. TimeText

Board Time

A.8.1.9. OriginTime

Origin time in seconds. Reference time base.

A.9. Timing Terminal's Functions

Function	Get/Set	PV suffix	Record type	Version
TSEnable	get/set	TSEN	bo	
TSEdge	get/set	TSCG	bo	
TSDecimation	get/set	TSDC	longout	
TimeStamp	get	TS	waveform	

A.9.1. Function Descriptions

A.9.1.1. TSEnable

Enable time stamping on the terminal

A.9.1.2. TSEdge

Time stamping edge:

A.9.1.3. TSDecimation

Timestamp decimation

A.9.1.4. TimeStamp

Read timestamps

A.10. Abstract Future Time Event Functions

Function	Get/Set	PV suffix	Record type	Version
Delay	get/set	D	ao	

Function	Get/Set	PV suffix	Record type	Version
Enabled	get/set	E	bo	
Timestamp	get	A	ai	
Level	get/set	L	bo	
Terminal	get/set	TERM	stringout	

A.10.1. Function Descriptions

A.10.1.1. Delay

Delay from the origin time.

A.10.1.2. Enabled

Enabled flag.

A.10.1.3. Timestamp

Actual time when it was generated.

A.10.1.4. Level

FTE level.

A.10.1.5. Terminal

Terminal on which this event will be issued.

A.11. Abstract Pulse Functions

Function	Get/Set	PV suffix	Record type	Version
Delay	get/set	D	ao	
Enabled	get/set	E	bo	
Generated	get	A	ai	
Width	get/set	W	ao	

Function	Get/Set	PV suffix	Record type	Version
Terminal	get/set	TERM	stringout	

A.11.1. Function Descriptions

A.11.1.1. Delay

Event delay from origin time.

A.11.1.2. Enabled

Disabled (0)
Enabled (1)

A.11.1.3. Generated

Actual timestamp when event was generated.

A.11.1.4. Width

Pulse width

A.11.1.5. Terminal

Terminal on which this event will be issued.

A.12. Abstract Clock Functions

Function	Get/Set	PV suffix	Record type	Version
Delay	get/set	D	ao	
Enabled	get/set	E	bo	
Timestamp	get	A	ai	
Width	get/set	P	ao	
DutyCycle	get/set	DC	ao	
TimeEnd	get/set	TE	ao	
Terminal	get/set	TERM	stringout	

A.12.1. Function Descriptions

A.12.1.1. Delay

Delay of clock generation start from origin time.

A.12.1.2. Enabled

Disabled (0)
Enabled (1)

A.12.1.3. Timestamp

Actual timestamp when generation started.

A.12.1.4. Width

Clock period

A.12.1.5. DutyCycle

Clock duty cycle

A.12.1.6. TimeEnd

End of clock generation. It is a delay from the origin time.

A.12.1.7. Terminal

Terminal on which this event will be issued.

A.13. Time Event Sequence

It supports messaging (see messaging description).

Function	Get/Set	PV suffix	Record type	Version
Delay		D	ao	
State	get	STAT	bo	
OriginTime	get/set	O	ao	

A.13.1. Function Descriptions

A.13.1.1. Delay

A.13.1.2. State

Sequence state:

DISABLED (0)
PROCESSING (1)

A.13.1.3. OriginTime

Origin time. This value overwrites device's value.

A.14. Image specific functions

Function	Get/Set	PV suffix	Record type	Version
–	get	(WF)	subArray	
Buffer	get	RAW	waveform	
BufferSize	get/set	SIZE	longout	
ImageType	get/set	IMGT	mbbo	
Resolution	get/set	RES	longin	
SamplesPerPixel	get	SPP	longin	
OriginX	get/set	ORGX	longout	
OriginY	get/set	ORGY	longout	
Height	get/set	HGHT	longout	
Width	get/set	WDTH	longin	
BinX	get/set	BINX	longout	
BInY	get/set	BINY	longout	
ReverseX	get/set	REVX	bo	
ReverseY	get/set	REVY	bo	

A.14.1. Function Descriptions

A.14.1.1. –

Cropped region. Region cropping is done on the EPICS level. If cropping is done on the device support level, then this PV must be replaced by relevant wave form record.

A.14.1.2. Buffer

Raw full image

A.14.1.3. BufferSize

Size of the buffer. Used to set the NELM field of the (WF) record.

A.14.1.4. ImageType

Type of the image's raw format.

GRAYSCALE (0) Grayscale (implies SPP=1).
RGB (1) Red-green-blue (implies SPP=3).
BGR (2) Blue-green-red (implies SPP=3).
RGBG (3) Bayer format (implies SPP=4).
GRGB (4) Bayer format (implies SPP=4).
RGGB (5) Bayer format (implies SPP=4).

A.14.1.5. Resolution

Resolution of the channel (number of bits per sample). For camera, number of bits per color.

A.14.1.6. SamplesPerPixel

Number of samples per pixel. Depends on selected image type. It is used to calculate region size.

A.14.1.7. OriginX

X coordinate of the image's upper-left corner (for cropping).

A.14.1.8. OriginY

Y coordinate of the image's upper-left corner (for cropping).

A.14.1.9. Height

Requested height of the camera image.

A.14.1.10. Width

Requested width of the camera image.

A.14.1.11. BinX

Binning function

A.14.1.12. BInY

Binning function

A.14.1.13. ReverseX

NORMAL (0) image is not reversed
REVERSED (1) image is reversed

A.14.1.14. ReverseY

NORMAL (0) image is not reversed
REVERSED (1) image is reversed

A.15. Image acquisition functions

Function	Get/Set	PV suffix	Record type	Version
ShutterStatus	Get	SHST	bi	
Exposure	Set/Get	EXP	longout	
MinExposure	Get	EMIN	longin	
MaxExposure	Get	EMAX	longin	
ExposureStep	Get	ESTP	longin	
Delay	Set/Get	DLY	longout	
MinDelay	Get	DMIN	longin	

Function	Get/Set	PV suffix	Record type	Version
MaxDelay	Get	DMAX	longin	
DelayStep	Get	DSTP	longin	
Hotpixels	Set/Get	HP	waveform	
HotpixelsCorrection	Set/Get	HPE	bi	
NumberOfLostFrames	Get	NLF	login	
ShutterMode	Set/Get	SHMD	mbbi	
BlackLevel	Set/Get	BLCK	longin	
ImageSourceType	Set/Get	IMST	mbbi	
ImageSource	Set/Get	IMGS	waveform	
ImageCounter	Get	IMGC	longin	
AcqusitionType	Set/Get		longout	

A.15.1. Function Descriptions

A.15.1.1. ShutterStatus

CLOSED (0) Shutter is closed.
OPEN (1) Shutter is opened.

A.15.1.2. Exposure

Acquisition time. Time during which shutter is opened and sensor is exposed.

A.15.1.3. MinExposure

Minimum exposure which is supported by camera. Provided by hardware.

A.15.1.4. MaxExposure

Maximum exposure which is supported by camera. Provided by hardware.

A.15.1.5. ExposureStep

Exposure step which is supported by camera. Provided by hardware.

A.15.1.6. Delay

Delay between trigger received and shutter is open. Provided by hardware.

A.15.1.7. MinDelay

Minimum possible delay which is supported by hardware. Provided by hardware.

A.15.1.8. MaxDelay

Maximum possible delay which is supported by hardware. Provided by hardware.

A.15.1.9. DelayStep

Minimum possible delay step which is supported by hardware. Provided by hardware.

A.15.1.10. Hotpixels

Description of hot pixels.

A.15.1.11. HotpixelsCorrection

Enable/disable hotpixel correction.

A.15.1.12. NumberOfLostFrames

Number of lost frames.

A.15.1.13. ShutterMode

Shutter mode

Global (0) Global shutter mode
Running (1) Running shutter mode

A.15.1.14. BlackLevel

Sets level of black pixels for this channel

A.15.1.15. ImageSourceType

Definition of image source

Camera (0) images obtained from camera.
File (1) images obtained from specified file/folder.
Plugin (2) Images obtained from plugin.

A.15.1.16. ImageSource

Definition of image source. Behavior of this PV depends on the IMST value.

A.15.1.17. ImageCounter

Continues image counter.

A.15.1.18. AcqusitionType

Image acquisition type:

Single (0) Single image per acquisition session
Predefined (1) Predefined number of images will be obtained and then acquisition stops.
Continues (2) Images will be obtained continuously while acquisition is not stopped explicitly.

B. List of doCallbacks Functions

This section includes a list of the doCallbacks functions for all possible EPICS type variations.

Event dispatching functions. Theses functions should be used by the user to notify asyn driver when data is ready to be handled. Majority of the functions has interruptId and port address parameters. These parameters are used to select correspondent handler to process data. InterruptId is stored by the handler registering function (last parameter). Asyn address is stored during initialization in protected field _portAddr

```
ndsStatus doCallbacksInt8Array(epicsInt8 *value, size_t nElements, int reason,
    int addr);
ndsStatus doCallbacksInt8Array(epicsInt8 *value, size_t nElements, int reason,
    int addr, epicsTimeStamp timestamp);
ndsStatus doCallbacksInt8Array(epicsInt8 *value, size_t nElements, int reason);
ndsStatus doCallbacksInt8Array(epicsInt8 *value, size_t nElements, int reason,
    epicsTimeStamp timestamp);

ndsStatus doCallbacksInt16Array(epicsInt16 *value, size_t nElements, int reason,
    int addr);
ndsStatus doCallbacksInt16Array(epicsInt16 *value, size_t nElements, int reason,
    int addr, epicsTimeStamp timestamp);
ndsStatus doCallbacksInt16Array(epicsInt16 *value, size_t nElements, int reason);
ndsStatus doCallbacksInt16Array(epicsInt16 *value, size_t nElements, int reason,
    epicsTimeStamp timestamp);

ndsStatus doCallbacksInt32Array(epicsInt32 *value, size_t nElements, int reason,
    int addr);
ndsStatus doCallbacksInt32Array(epicsInt32 *value, size_t nElements, int reason,
    int addr, epicsTimeStamp timestamp);
ndsStatus doCallbacksInt32Array(epicsInt32 *value, size_t nElements, int reason);
ndsStatus doCallbacksInt32Array(epicsInt32 *value, size_t nElements, int reason,
    epicsTimeStamp timestamp);

ndsStatus doCallbacksFloat32Array(epicsFloat32 *value, size_t nElements, int
    reason, int addr);
ndsStatus doCallbacksFloat32Array(epicsFloat32 *value, size_t nElements, int
    reason, int addr, epicsTimeStamp timestamp);
ndsStatus doCallbacksFloat32Array(epicsFloat32 *value, size_t nElements, int
    reason);
ndsStatus doCallbacksFloat32Array(epicsFloat32 *value, size_t nElements, int
    reason, epicsTimeStamp timestamp);
```

```
ndsStatus doCallbacksFloat64Array(epicsFloat64 *value, size_t nElements, int
    reason, int addr);
ndsStatus doCallbacksFloat64Array(epicsFloat64 *value, size_t nElements, int
    reason, int addr, epicsTimeStamp timestamp);
ndsStatus doCallbacksFloat64Array(epicsFloat64 *value, size_t nElements, int
    reason);
ndsStatus doCallbacksFloat64Array(epicsFloat64 *value, size_t nElements, int
    reason, epicsTimeStamp timestamp);

ndsStatus doCallbacksGenericPointer(void *pointer, int reason, int addr);
ndsStatus doCallbacksGenericPointer(void *pointer, int reason);

ndsStatus doCallbacksInt32(epicsInt32 value, int reason, int addr);
ndsStatus doCallbacksInt32(epicsInt32 value, int reason, int addr, epicsTimeStamp
     timestamp);
ndsStatus doCallbacksInt32(epicsInt32 value, int reason);
ndsStatus doCallbacksInt32(epicsInt32 value, int reason, epicsTimeStamp timestamp
    );

ndsStatus doCallbacksFloat64(epicsFloat64 value, int reason, int addr);
ndsStatus doCallbacksFloat64(epicsFloat64 value, int reason, int addr,
    epicsTimeStamp timestamp);
ndsStatus doCallbacksFloat64(epicsFloat64 value, int reason);
ndsStatus doCallbacksFloat64(epicsFloat64 value, int reason, epicsTimeStamp
    timestamp);

ndsStatus doCallbacksOctet(char *data, size_t numchars, int eomReason, int reason
    , int addr);
ndsStatus doCallbacksOctet(char *data, size_t numchars, int eomReason, int reason
    , int addr, epicsTimeStamp timestamp);
ndsStatus doCallbacksOctet(char *data, size_t numchars, int eomReason, int reason
    );
ndsStatus doCallbacksOctet(char *data, size_t numchars, int eomReason, int reason
    , epicsTimeStamp timestamp);
ndsStatus doCallbacksOctet(const std::string &data, int eomReason, int reason,
    int addr);
ndsStatus doCallbacksOctet(const std::string &data, int eomReason, int reason,
    int addr, epicsTimeStamp timestamp);
ndsStatus doCallbacksOctet(const std::string &data, int eomReason, int reason);
ndsStatus doCallbacksOctet(const std::string &data, int eomReason, int reason,
    epicsTimeStamp timestamp);
ndsStatus doCallbacksOctet(const std::string &data, int reason);
ndsStatus doCallbacksOctet(const std::string &data, int reason, epicsTimeStamp
    timestamp);
```

```
ndsStatus doCallbacksUInt32Digital(epicsUInt32 value, epicsUInt32 mask, int
    reason, int addr);
ndsStatus doCallbacksUInt32Digital(epicsUInt32 value, epicsUInt32 mask, int
    reason, int addr, epicsTimeStamp timestamp);
ndsStatus doCallbacksUInt32Digital(epicsUInt32 value, epicsUInt32 mask, int
    reason);
ndsStatus doCallbacksUInt32Digital(epicsUInt32 value, epicsUInt32 mask, int
    reason, epicsTimeStamp timestamp);
```

C. areaDetector Support

AreaDetector is an EPCIS module which describes the standard interface for the area detectors (cameras). areaDetector supports plugins for image handling: region-of-interest (ROI), various data format exports (JPEG, TIFF, STAT, etc.).

NDS provides areaDetector plugin support: i.e., an areaDetector plugins can be connected to an NDS image channel. This is shown in Figure 15, where areaDetector's ROI and TIFF plugins make use of an image captured by an NDS image channel.

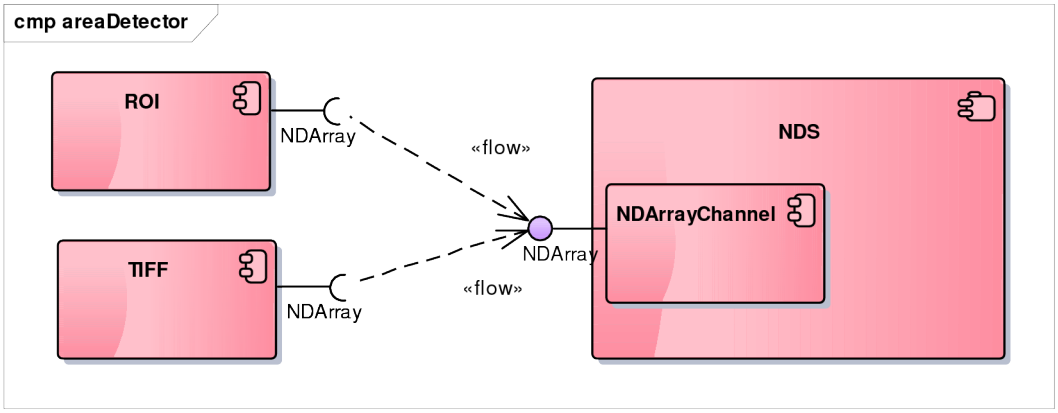

Figure 15: Run-time data flow involving AreaDetector plugins.

This is achieved by NDS also exposing areaDetector's NDArray interface for every NDS image channel. NDArrayChannel class provides NDArray interface for the areaDetector plugins.

Image buffer is treated as being in a raw format. NDS will convert the data to NDArray and publish it to all registered plugins as shown in activity diagram in Figure 17.

C.1. Installing AreaDetector

The areaDetector module is required to build NDS with areaDetector plugins' support.

The following modules are required that are not (yet) part of CODAC.

- busy: version 1.4
  http://www.aps.anl.gov/bcda/synApps/busy/busy.html
- sscan: version 2.8.1
  http://www.aps.anl.gov/bcda/synApps/sscan/sscan.html
- calc: version 3.0
  http://www.aps.anl.gov/bcda/synApps/calc/calc.html

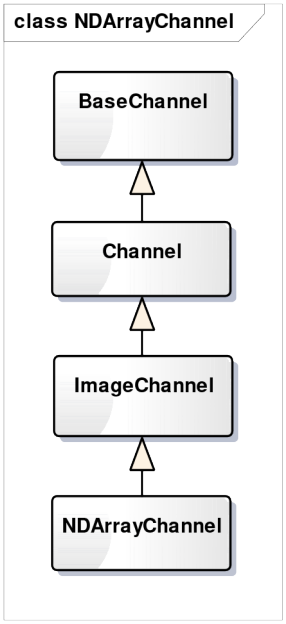

Figure 16: NDS image channel class hierarchy.

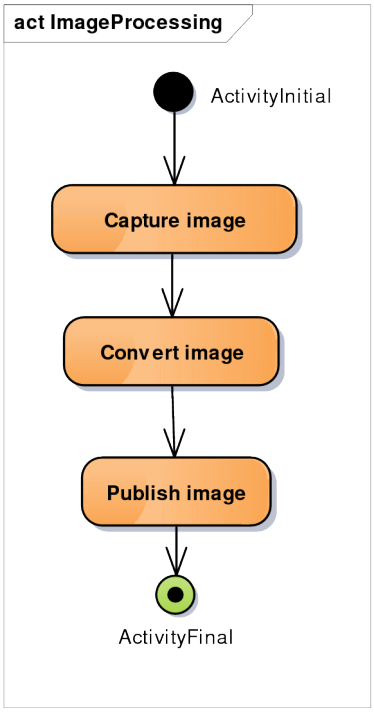

Figure 17: Activity diagram for publishing images to AreaDetector.

- areaDetector: version 1.8
  http://cars9.uchicago.edu/software/epics/areaDetector.html

These modules are needed as they are dependencies of areaDetector. They should be built using standard EPICS module build procedure, in the same order as listed above. You will need to edit `configure/RELEASE` file to specify the following locations of EPICS base and modules where they are due:

```
EPICS_BASE=/opt/codac/epics/base
EPICS_MODULES=/opt/codac/epics/modules
ASYN=$(EPICS_MODULES)/asyn
SSCAN=$(EPICS_MODULES)/sscan
BUSY=$(EPICS_MODULES)/busy
CALC=$(EPICS_MODULES)/calc
AUTOSAVE=$(EPICS_MODULES)/autosave
AREA_DETECTOR=$(EPICS_MODULES)/areaDetector
```

Before moving every module, move it to the corresponding directory `$EPICS_MODULES` as indicated above. You will need to have root privileges to move it there.

C.2. Building NDS with areaDetector Support

Define the path to areaDetector and the required modules in the configuration file `configure/RELEASE`. The following line should be added:

```
AREA_DETECTOR=${EPICS_MODULES}/areaDetector
BUSY=${EPICS_MODULES}/busy
CALC=${EPICS_MODULES}/calc
SSCAN=${EPICS_MODULES}/sscan
```

Then, the standard NDS build procedure should be followed (mvn package, install RPMs).

C.3. Building an NDS application with areaDetector Support

1. Define the path to areaDetector in the configuration file `configure/RELEASE`. The following lines should be added:

   ```
   AREA_DETECTOR=${EPICS_MODULES}/areaDetector
   BUSY=${EPICS_MODULES}/busy
   CALC=${EPICS_MODULES}/calc
   SSCAN=${EPICS_MODULES}/sscan
   ```

2. Check a list of required areaDetector's plugins in Makefile:

   ```
   <appName>/src/Makefile
   ```

3. Uncomment the NDArrayChannel in the project substitution file:

```
src/<appName>.substitutions
```

4. Check required plugins configuration and database loading:

```
iocBoot/ioc<appName>/st.cmd
```

5. Uncomment AREA_DETECTOR macro definition.
6. Uncomment required plugins (e.g., TIFF).

D. 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 18.

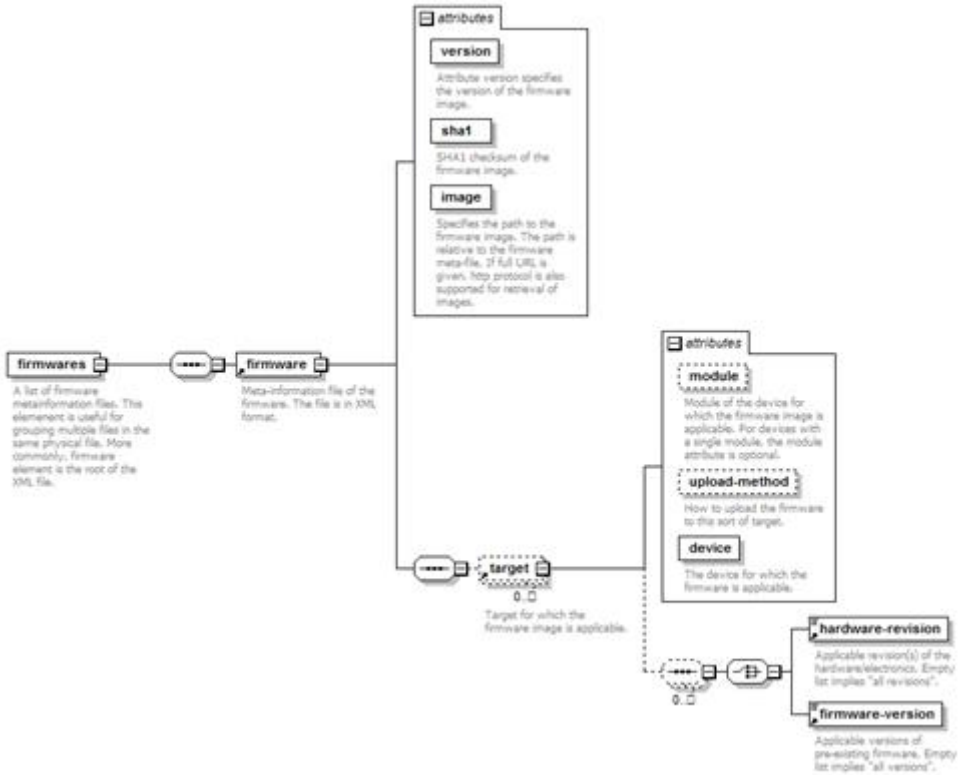

Figure 18: Schema of the firmware meta-information file.

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

```
<firmware image="http://server/ni6529-fw-2.3.1.bin"
        version="2.3.1"
        xmlns="http://www.cosylab.com/NDS/FirmwareMetaInfo/2012">

 <target device="NI6529" module="fpga-3">
   <hardware-revision>rev0</hardware-revision>
   <hardware-revision>rev1</hardware-revision>
   <firmware-version>2.2.0</firmware-version>
   <firmware-version>2.2.1</firmware-version>
```

```
    <firmware-version>2.2.2</firmware-version>
  </target>
</firmware>
```