

# Table of Contents

<b>1</b>	<b>INTRODUCTION.....</b>	<b>2</b>
1.1	System Overview .....	2
1.2	Organization of the Manual.....	2
1.3	Definition .....	2
1.4	References .....	3
<b>2</b>	<b>GETTING STARTED.....</b>	<b>4</b>
2.1	Using Code Generators.....	4
2.1.1	EPICS Code Generator .....	4
2.1.2	CODAC Code Generator .....	4
2.2	Directory Structure of an NDS Device-specific Driver .....	5
2.3	Minimal Driver.....	6
2.4	Building an EPICS Application.....	7
<b>3</b>	<b>NDS TRACING SYSTEM .....</b>	<b>8</b>
3.1	Managing Trace Level.....	8
<b>4</b>	<b>DEVICE STRUCTURE .....</b>	<b>9</b>
4.1	Device Class .....	9
4.1.1	ChannelGroup Iteration .....	9
4.1.2	Finding Channels .....	9
4.2	ChannelGroup.....	10
4.2.1	Iteration of Channels.....	10
4.3	Channel.....	11
4.3.1	Accessing a Device from a Channel .....	11
<b>5</b>	<b>LIFECYCLE OF A DEVICE DRIVER .....</b>	<b>12</b>
<b>6</b>	<b>NDS::DEVICE::CREATESTSTRUCTURE .....</b>	<b>15</b>
6.1	Accessing Initialization Parameters .....	15
<b>7</b>	<b>DEFINING CHANNELS .....</b>	<b>16</b>
7.1	Manual Channel Creation and Registration .....	17
7.2	Automatic Channel Object Creation and Registration.....	17
<b>8</b>	<b>EPICS RECORDS' HANDLING.....</b>	<b>19</b>
8.1	Synchronous EPICS Record Handling .....	20
8.2	Getters' Call on Initialization .....	21
8.3	Overriding a Standard Function .....	21
8.3.1	Example: Reading Value from Channel .....	22
8.4	Asynchronous EPICS Record Handling.....	22
8.5	Using Asynchronous Update.....	22
8.6	Asyn Reason Arguments .....	23
8.7	Records Timestamping.....	24

8.7.1	Synchronous Timestamp Update .....	24
8.7.2	Asynchronous Timestamp Update .....	25
8.8	Read Back Value .....	26
8.9	Get asynAddr from get/set Function .....	27
9	STATE MACHINES .....	28
9.1	Enable Objects .....	30
9.2	Device's State Machine .....	31
9.2.1	Fast init .....	31
9.3	ChannelGroup and Channel State Machine Implementation .....	34
9.4	How to Block Undesired Transitions .....	34
9.5	Reset Operation .....	35
9.5.1	Switching from the RESETTING State .....	35
10	MESSAGING MECHANISM .....	36
10.1	Handling Messages .....	36
10.2	Standard Message Types .....	36
10.3	CSS Messaging Support .....	37
11	LOADING FPGA CODE .....	38
11.1	NDS v2.0 Implementation .....	38
11.2	NDS v2.1 Implementation .....	38
12	TIME EVENTS .....	40
13	NDS TOOLS .....	42
13.1	epicsMutex and AsynDriver Locker .....	42
14	NDS TASKS .....	43
14.1	nds::TaskService .....	43
14.2	nds::ThreadTask Class .....	44
14.3	nds::PeriodicTask Class .....	45
14.4	nds::PollingTask Class .....	45
14.4.1	nds::PollingTask Example .....	46
14.5	nds::Timer Class .....	48
15	DEFINE DRIVER SPECIFIC IOC FUNCTION .....	50
16	SOFTWARE USER MANUAL REVIEW CHECKLIST .....	51

# 1 INTRODUCTION

This Developer's Manual provides the information necessary for a developer to effectively implement an EPICS device support with usage of NDS.

## 1.1 System Overview

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

The NDS library is also provided with online documentation. To access it on CODAC systems, make sure that the `codac-core-X.Y-epics-nds-doc` RPM is installed where `X.Y` is the CODAC version. The documentation is then available at location:

```
file:///opt/codac/epics/modules/nds/doc/index.html
```

## 1.2 Organization of the Manual

This manual covers the following topics:

- Getting Started
- NDS Tracing System
- Device Structure
- Lifecycle of a Device Driver
- `nds::Device::createStructure`
- Defining channels
- EPICS records' handling
- State machines
- Messaging mechanism
- Loading FPGA code
- Time events
- NDS tools
- NDS tasks
- Define driver specific IOC function
- Software User Manual Review Checklist
- Function Reference
- List of `doCallbacks` functions (for all possible EPICS type variations)
- `areaDetector` support

## 1.3 Definition

AI	Analog Input
AO	Analog Output

<b>CODAC</b>	Control, Data Access and Communication
<b>DAQ</b>	Data acquisition
<b>FIR</b>	Finite Impulse Response
<b>GPIO</b>	General Purpose Input/Output
<b>IIR</b>	Infinite Impulse Response
<b>IOC</b>	Input/Output Controller
<b>SDD</b>	Self-Description Data
<b>NDM</b>	Nominal Device Model
<b>NDS</b>	Nominal Device Support
<b>SEU</b>	Single Event Upset
<b>TCN</b>	Time Communication Network
<b>UML</b>	Unified Modeling Language

## 1.4 References

- [RD1] CODAC Core System Self-Description Data Editor User Manual ([32Z4W2](#))
- [RD2] NI Sync EPICS Driver User's Guide ([33Q5TX](#))
- [RD3] asynDriver: Asynchronous Driver Support  
<http://www.aps.anl.gov/epics/modules/soft/asyn/>
- [RD4] GigE Vision: Video Streaming and Device Control Over Ethernet Standard, v2.0
- [RD5] ITER Numbering System for Parts/Components ([28QDBS](#))
- [RD6] NI-RIO EPICS Module: RIOEM, Engineering Design Document ([6WU76X](#))
- [RD7] asynSIS8300-epics-driver - Programmer's Guide ([6RK5ST](#))
- [RD8] NI PXI-6259 EPICS Device Support User's Guide ([3DEY52](#))
- [RD9] EPICS Application Developer's Guide ([AppDevGuide](#))
- [RD10] Time Representation in CODAC software ([7GDNSX v1.1](#))
- [RD11] EPICS 3-14 Record Reference Manual ([RRM-3.14](#))
- [RD12] How to include a new I/O Module in SDD ([A4WQDZ v1.0](#))
- [RD13] CODAC Core System Application Development Manual ([33T8LW v4.4](#))

## 2 GETTING STARTED

### 2.1 Using Code Generators

NDS provides the following EPICS templates:

- nds
- ndsTime

The first template, 'nds' is a general template for DAQ devices, while the second 'ndsTime' is a template for timing devices.

#### 2.1.1 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
```

The template includes 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.

#### 2.1.2 CODAC Code Generator

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

```
$ mvn iter:newunit -Dunit=m-nds-example
[INFO] MODULE UNIT 'm-nds-example' CREATED
$ cd m-nds-example
$ mvn iter:newapp -Dapp=ndsExample -Dtype=nds
[INFO] EPICS APPLICATION 'ndsExample' CREATED
$ mvn iter:newioc -Dioc=nds-example -Dapp=ndsExample -Dtype=nds
Using target architecture linux-x86_64 (only one available)
[INFO] EPICS IOC 'ndsExample' CREATED
[INFO] IOC 'ndsExample' 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-3.1-nds-example-
3.1.0.v0.0a1-1.el6.x86_64.rpm
[INFO] PACKAGING COMPLETED (see for .rpm files under 'target'
directory)

```

The following RPMs are obtained this way:

- `codac-core-3.1-nds-example-3.1.0.v0.0a1-1.el6.x86_64.rpm`: RPM containing the shared library. This RPM should be installed on the development and target machines.
- `codac-core-3.1-nds-example-ioc-3.1.0.v0.0a1-1.el6.x86_64.rpm`: RPM containing the test IOC application. This RPM should only be installed on the target machines used for testing.

## 2.2 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 type
│   │   ├── ...
│   │   └── 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

```

## 2.3 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 2.1).

*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
```

## 2.4 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 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().

### 3.1 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
```

## 4 DEVICE STRUCTURE

The device structure is described in Figure 1.

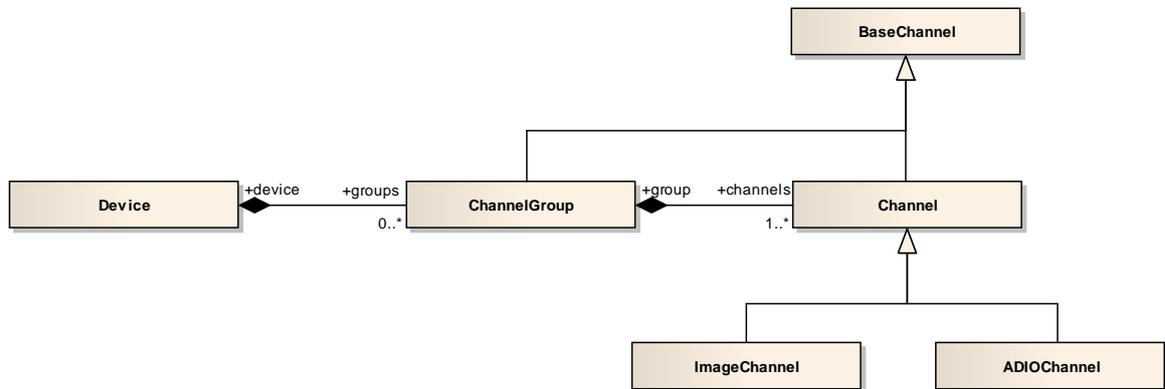


Figure 1 Classes diagram

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.

### 4.1 Device Class

A device class provides common functionality for device objects.

#### 4.1.1 ChannelGroup Iteration

```
for( ChannelGroupContainer::iterator itr = _nodes.begin();
    itr != _nodes.end();
    ++itr)
{
    nds::ChannelGroup *gr = itr->second->getBase();
}
```

#### 4.1.2 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**);

```

## 4.2 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.

### 4.2.1 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)
}

```

## 4.3 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.

### 4.3.1 Accessing a Device from a Channel

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

```
Device* _device;
```

## 5 LIFECYCLE OF A DEVICE DRIVER

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

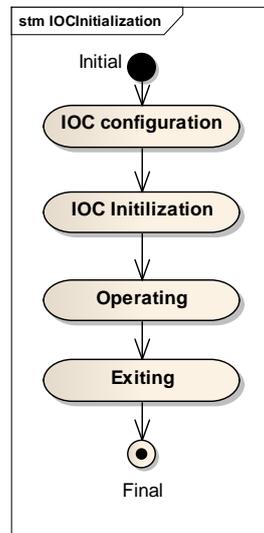
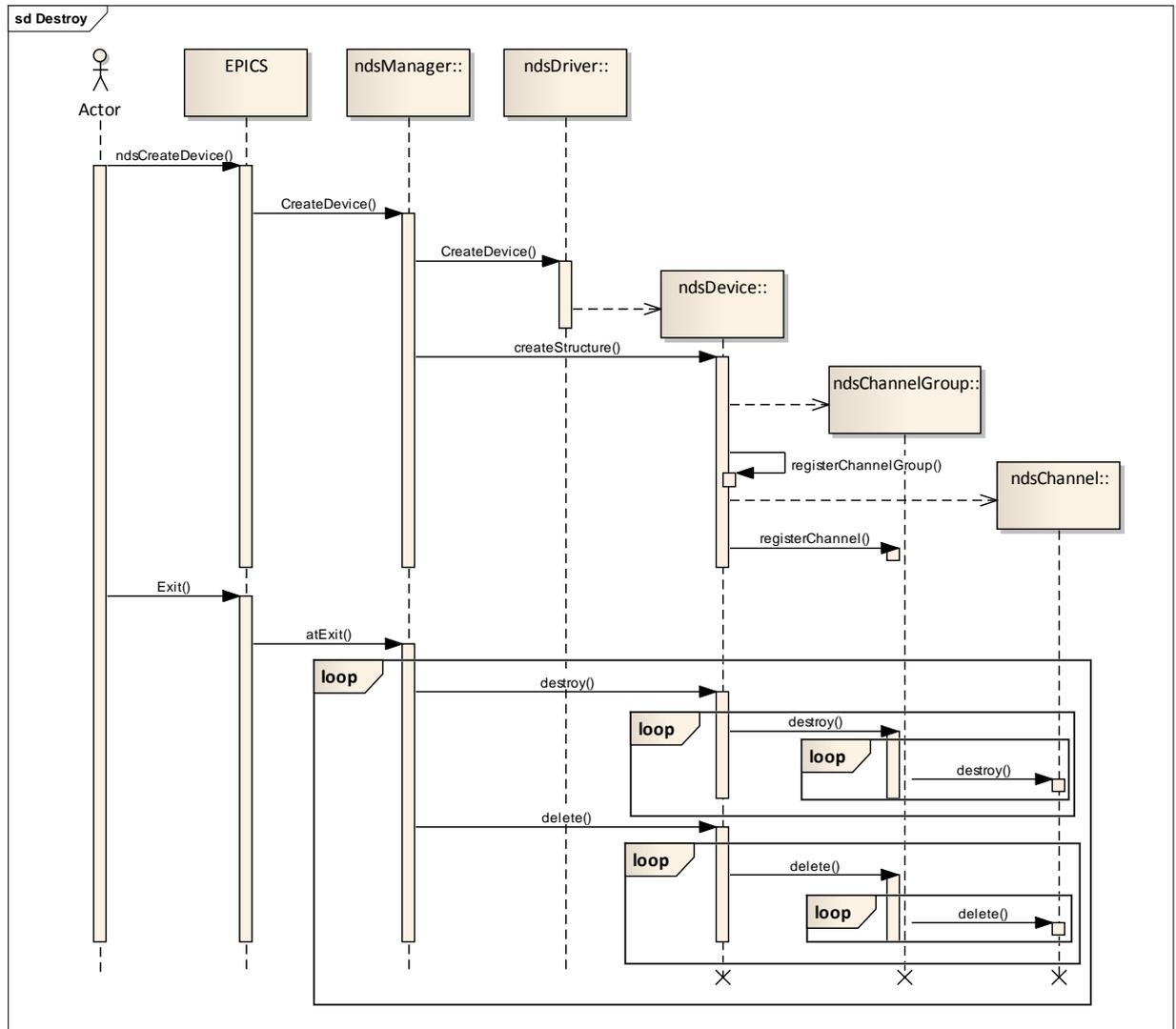


Figure 2 IOC state machine

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 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.



**Figure 3 Creation and destruction of objects**

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 Section 9). 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, 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.

## 6 NDS::DEVICE::CREATESTRUCTURE

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.

### 6.1 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);
...
```

## 7 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 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”

Device and channel group functions are accessible through the main device asyn port through different addresses. The device’s address is 0, while addresses of the channel groups start from 1. The channel group’s address is assigned automatically according to the order of the channel groups’ registration.

Channels have sequencing numbers and are numbered automatically according to its registration.

Example of port structure relatively to driver object’s functions (port:address)

- pxi6368.0:0 – device functions.
- pxi6368.0:1 – “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.
- pxi6368.0:2 – “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 [RD9] 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 concurrency. The extreme scenario is when each channel would be put in its own channel group – then, there would be no locking among channels.

## 7.1 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");

    int i;
    /* Both channel groups have 32 channels each. */
    for(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.

## 7.2 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 EPICS 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.

## 8 EPICS RECORDS' HANDLING

NDS provides a way to connect EPICS record to c++ call-back functions. These functions called record's handlers. For each record NDS provides 2 functions: getter and setter (see Figure 4 Handling CA requests.). Getter and setter are used for synchronous record processing.

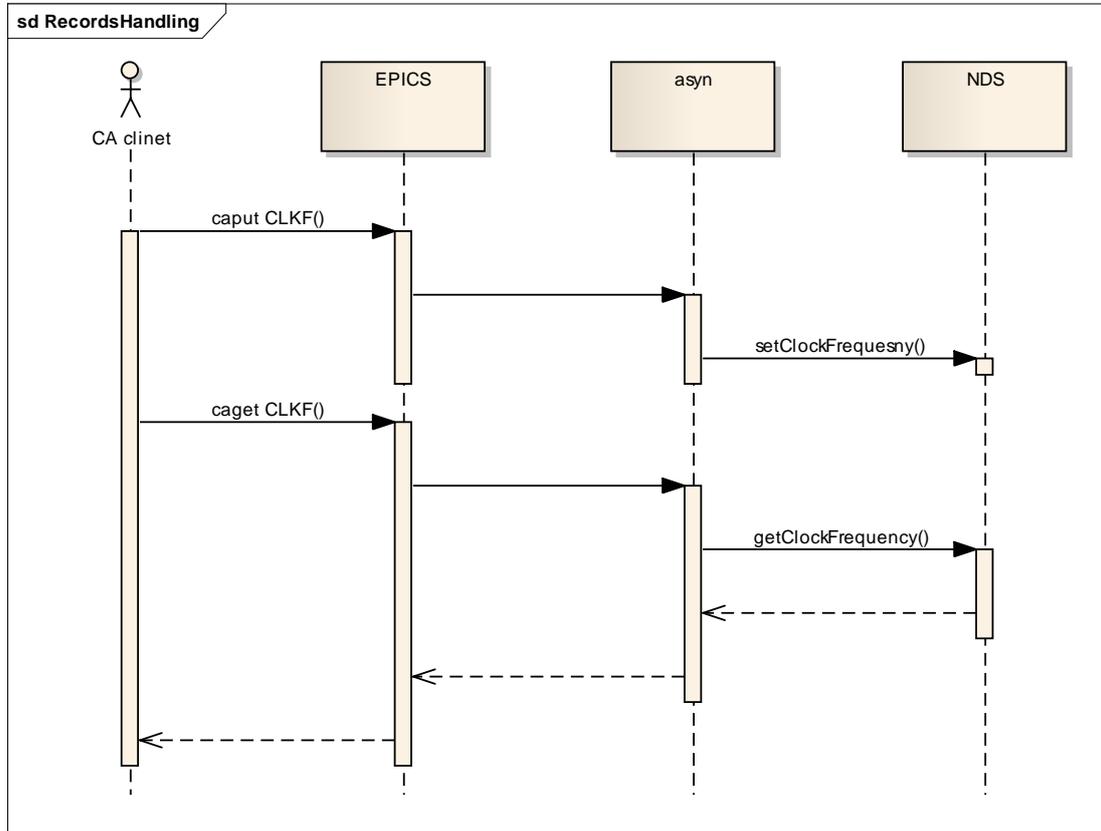


Figure 4 Handling CA requests.

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.

NDS also provides asynchronous record updating through EPICS interrupts system. Each record marked with SCAN field equivalent "I/O Interrupt" is registered within NDS. Registered means that NDS owns interrupt ID which could be used to update record through doCallbacks functions (see Figure 5 Asynchronous record update).

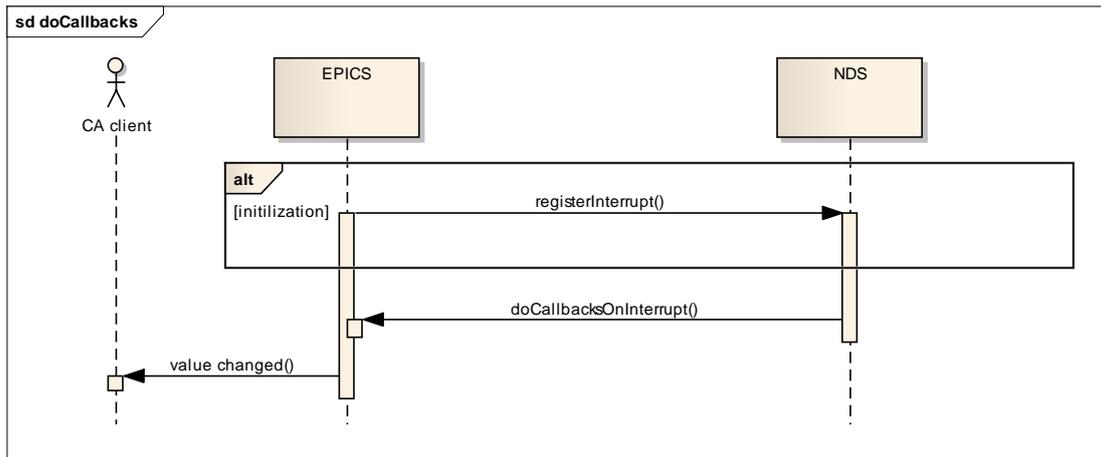


Figure 5 Asynchronous record update

## 8.1 Synchronous EPICS Record Handling

Developers can easily add new PVs and register handlers for them by overloading **registerHandlers** member. This function accepts a PV container as argument.

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

This member can be overloaded for the Device, ChannelGroups, and Channels.

[There are macro definitions for the handler registration:

- NDS\_PV\_REGISTER\_OCTET
- NDS\_PV\_REGISTER\_INT32
- NDS\_PV\_REGISTER\_FLOAT64
- NDS\_PV\_REGISTER\_INT8ARRAY
- NDS\_PV\_REGISTER\_INT16ARRAY
- NDS\_PV\_REGISTER\_INT32ARRAY
- NDS\_PV\_REGISTER\_FLOAT32ARRAY
- NDS\_PV\_REGISTER\_FLOAT64ARRAY
- NDS\_PV\_REGISTER\_UINT32DIGITAL

Macro definitions accept the following parameters:

reason	This is a text reason which will be used in INP or OUT fields of the record.
writer	Pointer on a write function.*
reader	Pointer on a read function.*
interruptId	This is an address of integer variable to store interruptId for this reason. For each string reason, the NDS driver will assign an integer code that is required by asynDriver. This code will be stored in this variable. This code should be used together with doCallbacks functions to initiate interrupt (see asynPortDriver documentation).

Read and write functions should match prototypes (see Appendix A )

For example,

```
NDS_PV_REGISTER_OCTET("Firmware", &Device::writeOctet,
&Device::getFirmware, &interrupt);
```

Registration requires both writer and reader, but occasionally it is useless to have both. In this case, the developer can use fake handlers which are stubs. The following functions are defined which can be used as a stub:

- writeOctet / readOctet
- writeInt32 / readInt32
- writeFloat64 / readFloat64
- writeInt8Array / readInt8Array
- writeInt16Array / readInt16Array
- writeInt32Array / readInt32Array
- writeFloat32 / readFloat32
- writeFloat64 / readFloat64
- writeUInt32Digital / readUInt32Digital

## 8.2 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 *nbytesTransferred, int *eomReason)
{
    if ( getCurrentState() == CHANNEL_STATE_IOC_INITIALIZATION )
    {
        // Prevent overwriting of VAL value on initialization
stage
        return ndsError;
    }
    return ndsSuccess;
}
```

## 8.3 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.

### 8.3.1 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;
}
```

## 8.4 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](#)).

## 8.5 Using Asynchronous Update

### 1. Record should be present in DB files.

It can from NDS standard records or a device specific record.

#### 1.1. AsynType of record is defined in DB file:

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

**1.2.** Record should be configured to for I/O interrupt scanning type [X] (see EPICS RRM [RD11]):

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

**1.3.** 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
```

## 8.6 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;

```

## 8.7 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 [RD11]). 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")
```

### 8.7.1 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(asyUser *pasyUser, epicsInt32 *value,
size_t nElements, size_t *nIn)
{
    epicsTime time = epicsTime::getCurrent();
    epicsTimeStamp stamp = (epicsTimeStamp)time;
    pasyUser->timestamp = stamp;

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

## 8.7.2 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.

## 8.8 Read Back Value

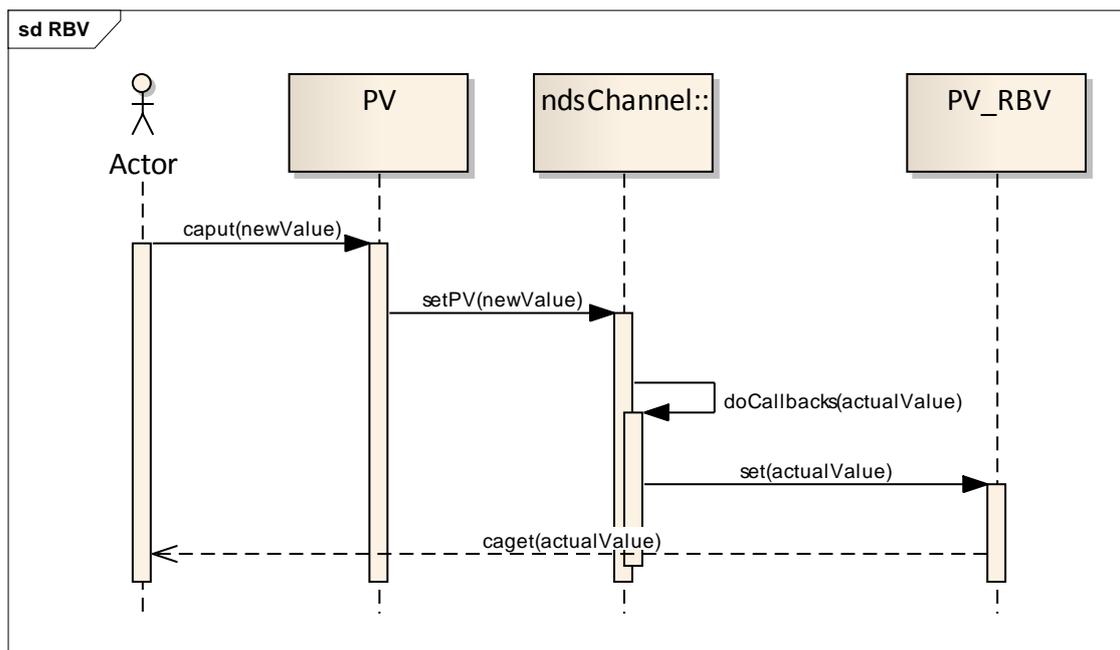


Figure 6: Read back values

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, "$(PREFIX)-TEST") {
    field(DESC, "TEST of RBV")
    field(DTYP, "asynInt32")
    field(OUT, "@asyn($(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;
}

```

## 8.9 Get asynAddr from get/set Function

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);

```

## 9 STATE MACHINES

Each device and each channel has a state, as described in sections 9.1 and 9.3. 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 7). 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 9). **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 7):

- **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.

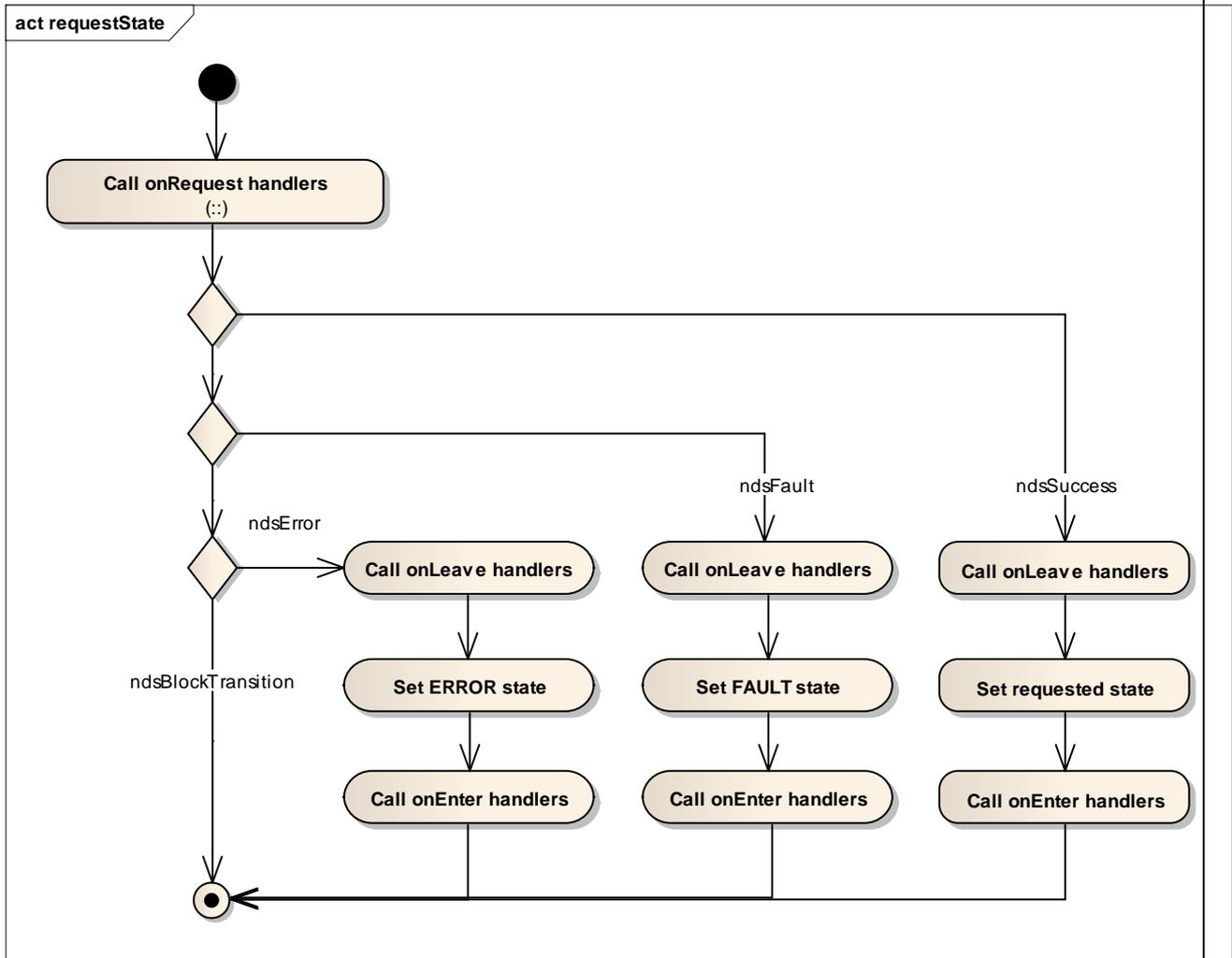


Figure 7 Request state activity diagram.

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.

## 9.1 Enable Objects

Each object has an ENBL record which describes the global state of the object (Figure 8).

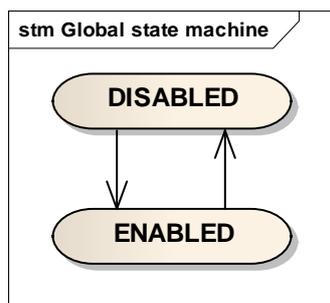


Figure 8: Global state machine

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.

## 9.2 Device's State Machine

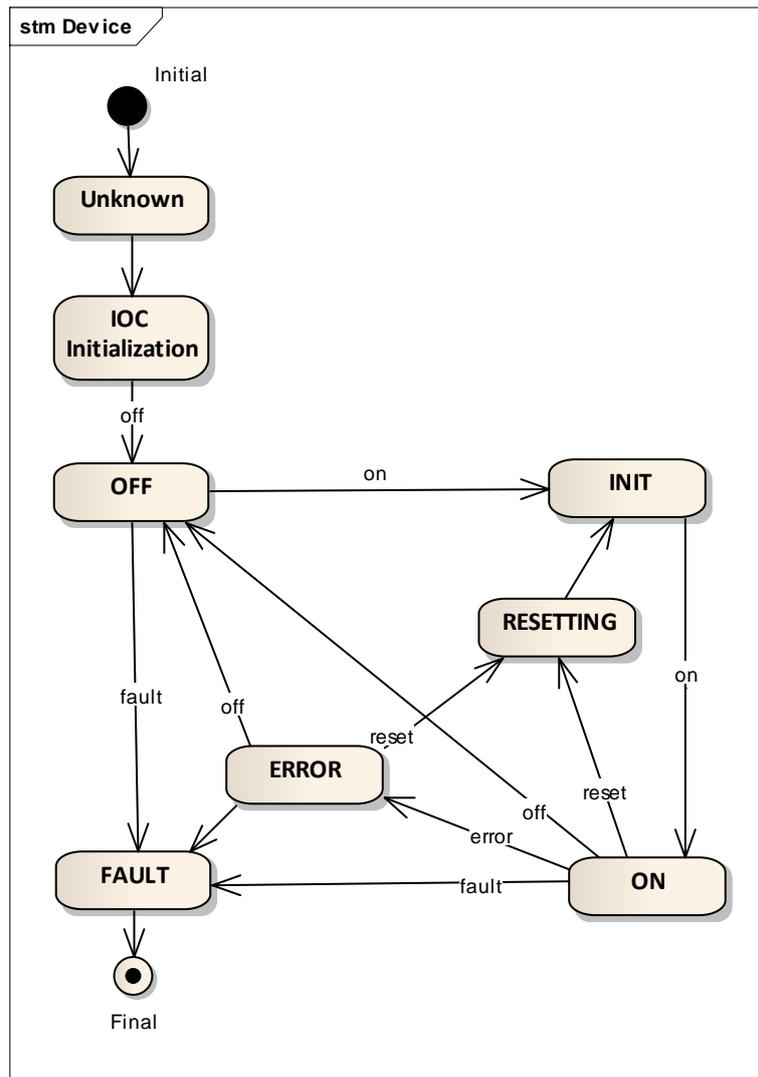


Figure 9 Device's state machine

ERROR and FAULT states are accessible from any other state.

### 9.2.1 Fast init

An ON message sent to a device forces the state machine to switch to the INIT state. If the device does not require the INIT procedure 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.

### 9.2.1.1 *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.

- 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("Can't 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)

## 9.3 ChannelGroup and Channel State Machine Implementation

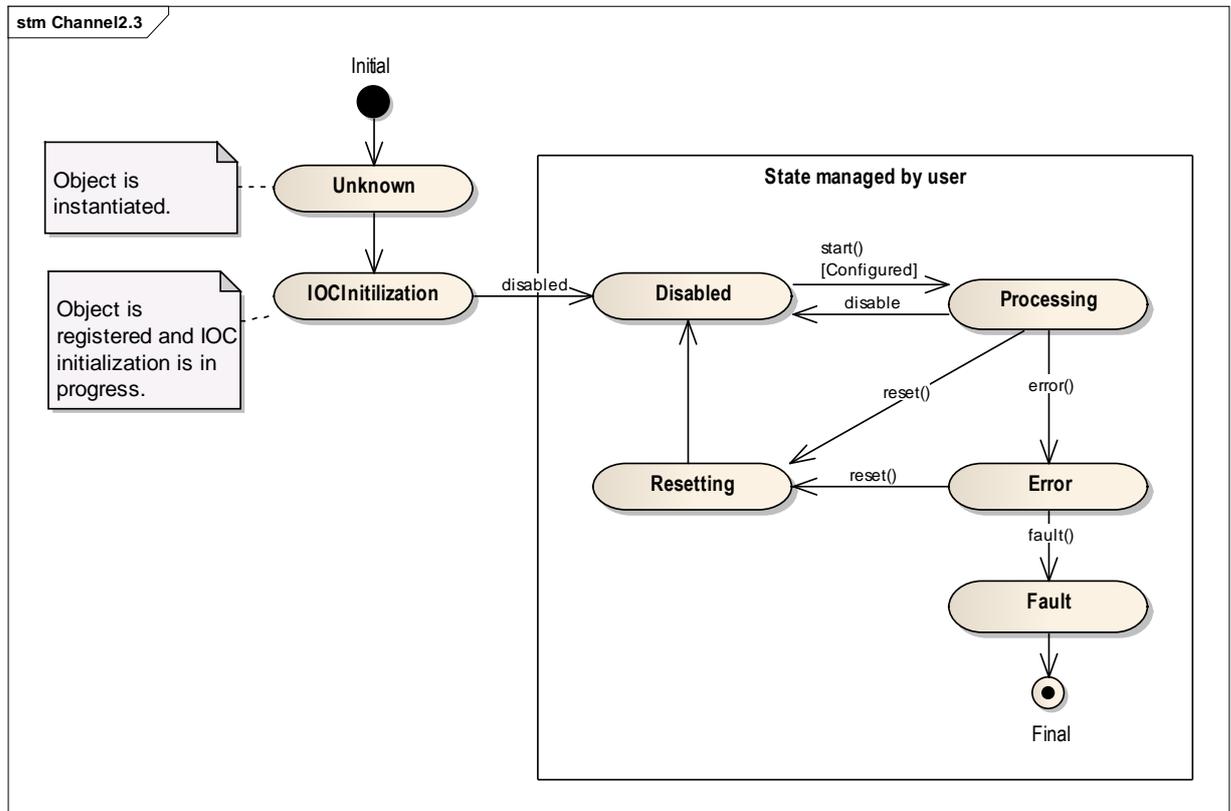


Figure 10: Channel state machine.

ERROR and FAULT states are accessible from any state.

## 9.4 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(
        nds::DEVICE_STATE_RESETTING, // Current state
        nds::DEVICE_STATE_ON, // Requested state
    )
  
```

```
boost::bind(&nds::Device::onWrongStateRequested, this,
_1, _2) );
```

## 9.5 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).

### 9.5.1 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.

# 10 MESSAGING MECHANISM

## 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;
}
```

## 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.

### **10.3 CSS Messaging Support**

Messaging (MSG/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.

# 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 11).

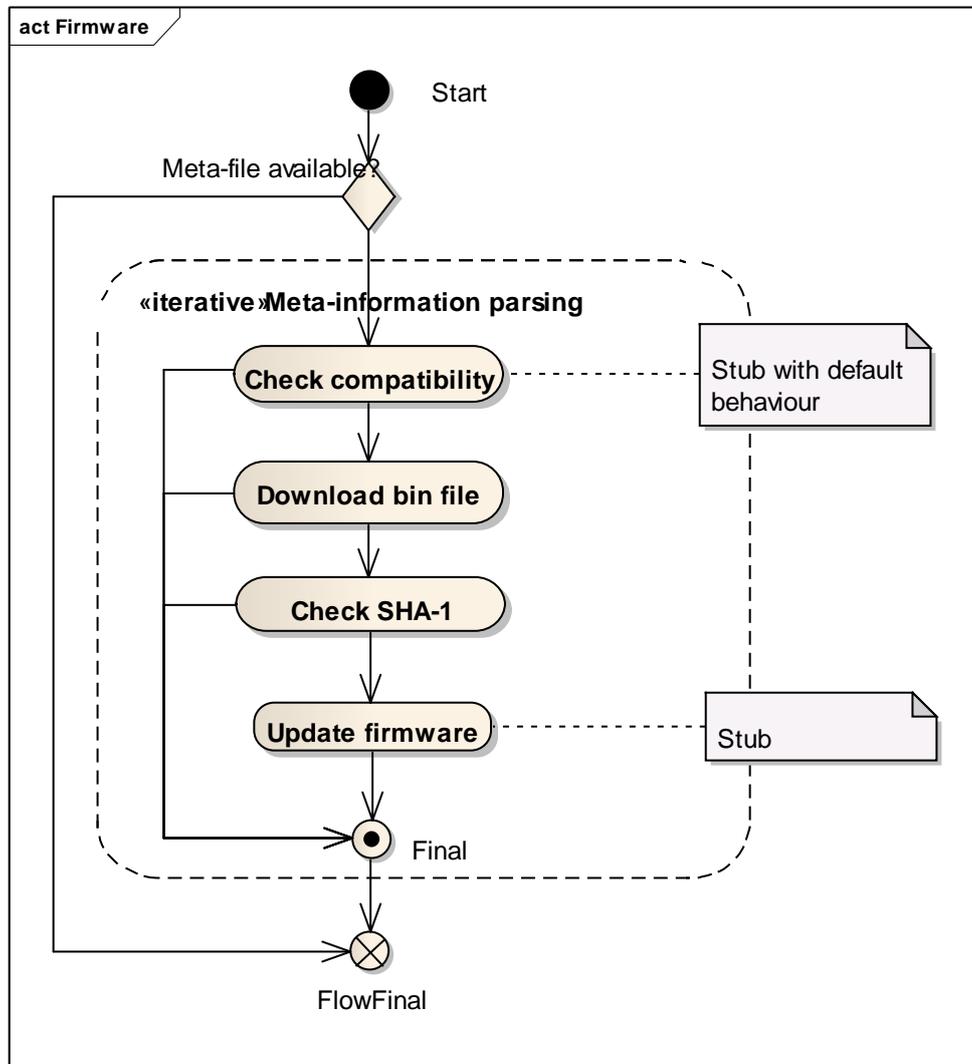


Figure 11 Firmware update activity diagram

## 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.

## 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.2). 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.

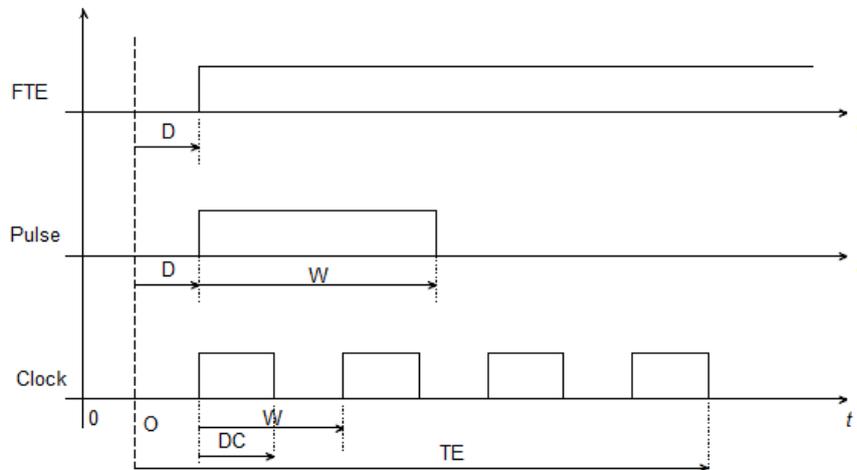
## 12 TIME EVENTS

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.



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.

## **13 NDS TOOLS**

### **13.1 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).

# 14 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.

## 14.1 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.

## 14.2 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;
    }

```

## 14.3 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);

```

TaskBody is a task handler which has the following signature:

```

void (*) (TaskServiceBase& service)

```

The task could be cancelled and restarted again.

## 14.4 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 – epicstthread 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.

#### 14.4.1 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("Opening 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();
}

```

## 14.5 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** epicsT

## 15 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)
}
```

# 16 SOFTWARE USER MANUAL REVIEW CHECKLIST

[Print this checklist separately to be included in the SUM Review Report]

Review Of Software User Manual	
Project Name	[Software Project Title]
Document	Software User Manual [IDM Link]
Reviewer	[V&V Team]

Criteria	Yes/No/ NA
<b>Document Standard</b>	
1 Were standards/guidelines and naming conventions established for the document?	
1a Does the document format conform to the specified standard/guideline?	
1b Are the standards and naming conventions established followed throughout the document?	
<b>Operations Overview</b>	
2 Does the document contain a section that describes a high level purpose and main capabilities of the software, and its overall operation in terms of the following:	
2a Function?	
2b Options?	
2c System performance considerations (i.e., restrictions & limitations)?	
<b>Detailed Description of Functions</b>	
3 Is there a detailed description of the overall subsystem(s) or major functional capability?	
4 Have assumptions and restrictions to processing been addressed?	
5 Have high-level diagrams of subsystems, including interfaces, data flow, and communications for each processing mode been provided?	
6 Is a high-level description of input and output provided?	
7 Have detailed descriptions of processing keyed to operator-specified input and actions in terms of points of control, functions performed, and results obtained (both normal and abnormal, i.e., error processing and recovery) been addressed?	
8 Have samples of displays in the order in which they are generated been provided?	
9 Have sample hardcopy output in the order in which they are produced been provided?	
10 Have numbered messages with explanations of system's and user's actions been provided?	

Criteria	Yes/No/ NA
11 Have descriptions of inputs from any other sources other than users that may affect its interface with the user been addressed?	
<b>Installation and Initialization</b>	
12 Does the document explain in detail the procedures for installing, tailoring, and initiating the software, including:	
12a Equipment set-up?	
12b Power-on and power-off?	
12c Bootstrap and load?	
12d Initiation commands?	
12e Interrupt/recovery/restart?	
12f Initialization of files, variables, or other data?	
12g Tailoring, reconfiguration, adaptation?	
12h Re-initialization after failure?	
<b>Startup and Termination</b>	
13 Does the document describe how to start and terminate operations normally, and how to determine whether normal termination has occurred?	
14 Does the document include procedures to address:	
14a Trouble indications and corrective actions?	
14b On-line interventions?	
14c Trap recovery?	
14d Operating communications?	
14e Fault isolation techniques?	
14f Conditions requiring software abort or equipment shut-down?	
15 Does the document include procedures for restarting after both normal and abnormal termination?	
16 If recovery procedures are required for restarting after abnormal termination, do they address:	
16a Check points?	
16b Collection of failures data?	
16c Restoring files?	

Criteria	Yes/No/ NA
16d Restoring devices to operational mode?	
<b>Error and Warning Messages</b>	
17 Does the document contain a list and explanations for each possible error condition and associated messages that may be encountered along with the corresponding corrective actions to be taken?	
18 Does the document identify agency or point-of-contacts for assistance?	
<b>Recovery Steps</b>	
19 Does the document explain recovery procedures the user may employ?	

## Appendix A REFERENCE

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

The first section lists the configuration parameters related to a Channel and the second section lists the configuration parameters related to a device.

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.

*Function* column 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")
```

Not all devices are required to support all of these functions and configuration parameters. For those that are not optional, the *Description* column explicitly states [REQUIRED]. 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., [REQUIRED, AI, C] means that the parameter is required for all analog input and camera channels.

## A.1. Channel Functions

Function	PV suffix	Record type	Description
Buffer	(WF)	waveform	[REQUIRED: DO; OPTIONAL: AO] Output channels: the contents of the buffer to output to the channel. The values are subject to conversion from engineering units to raw units.
BufferSize	BUFS	longout	[REQUIRED: AI, DI, DIO, C] Size of the buffer. Used to set the <b>NELM</b> field of the (WF) record.
getBandwidth	BW	ai	[AI] Get the bandwidth of an analog input channel.
getBuffer	(WF)	waveform	[REQUIRED: AI, DI, DIO, C] Input channels: retrieve the next buffer. The <b>NORD</b> field specifies how many samples are read. The <b>NELM</b> field specifies the maximal number of samples.
getImpedance	IMPD	ai	[AI] Get the impedance of the analog input channel.
getMeasuredOffset	MOFS	ai	[AI] Return the measured offset of an analog input channel.
getSamplesPerPixel	SPP	longin	[C] Number of samples per pixel.
Quality	Q	ai	[AI] Quality of the measurements on the channel (e.g., measure of signal-to-noise).
readFFT	FFT	waveform (double type)	[AI] The Fourier transform of the acquired signal.
receiveMessage	MSGR	stringin	Retrieve the last message from the device. The message is of the format <message> <correlation-id>

Function	PV suffix	Record type	Description
<b>sendMessage</b>	<b>MSGGS</b>	<b>stringout</b>	<p>Send a message to the device. Message string has the format:</p> <p><b>&lt;message&gt; [&lt;correlation-id&gt; [&lt;param&gt; [&lt;param&gt;] ...]]</b></p> <p>where &lt;message&gt; is the type of the message to convey to the device. &lt;correlation-id&gt; 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.</p> <p>Standardized messages are:</p> <p><b>STOP</b> Stop any action that is in progress at the device (e.g., test execution).</p> <p><b>TEST-QUICK</b> Perform a quick self-test of the device. <b>TEST-FULL</b> Perform an intensive self-test of the device. The test checks all the test-points defined by the device, therefore it may take a longer time to complete.</p> <p>If during a test, problems are found, the state of the device (STAT record) is set to the <b>ERROR</b> or <b>DEFUNCT</b> state. More information can be found in the IOC error log.</p>
<b>setChannelState</b>	<b>STAT</b>	<b>mbbo</b>	<p><b>[REQUIRED]</b> Set the state of the channel. The following values are possible:</p> <p><b>0 OFF</b> The channel is disabled. In this state, no operation can be performed on the channel, and the state must first be set to <b>ON</b>.</p> <p><b>1 ON</b> The channel is ready to be configured. This is the initial state of the channel.</p> <p><b>2 RESET</b> Reset the channel. When set, the channel will be configured to its default configuration. When reconfigured, its state will be set to <b>ON</b>.</p> <p><b>3 BUSY</b> Channel is busy (acquiring analog input or frames, or generating output). To stop the activity, state can be set to <b>ON</b>.</p> <p><b>4 ERROR</b> A channel-level error occurred. The channel needs to be reset.</p> <p><b>5 DEFUNCT</b> 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 <b>DEFUNCT</b>, the device state is also set to <b>DEFUNCT</b>.</p>

Function	PV suffix	Record type	Description
<b>setClockFrequency</b>	<b>CLKF</b>	<b>ao</b>	<p>[AI, AO, DI, DO, DIO, C] Set the frequency of the internal clock selected with <b>CLKSRC</b> record. The frequency is specified in Hz.</p> <p>For camera channels, this parameter specifies the number of frames per second.</p> <p>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 (<b>SEVR</b> field) of this record is set to <b>MINOR</b> and the alarm state (<b>STAT</b>) is set to <b>WRITE</b>. If internal clock source is selected, the frequency is set to the nearest smaller frequency.</p>
<b>setClockMultiplier</b>	<b>CLKM</b>	<b>ao</b>	<p>[AI, AO, DI, DO, DIO, C] Set the clock multiplier of the external clock selected with <b>CLKSRC</b> 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.</p> <p>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 (<b>SEVR</b> field) of this record is set to <b>MINOR</b> and the alarm state (<b>STAT</b>) is set to <b>WRITE</b>. If external clock source is selected, the multiplier is set to the nearest smaller multiplier.</p>

Function	PV suffix	Record type	Description
<b>setClockSource</b>	<b>CLKS</b>	<b>mbbo</b>	<p>[<b>AI, AO, DI, DO, DIO, C</b>] Set the clock source for sampling (input channel), frame acquisition (camera) or clocking digital-to-analog converter (output channel).</p> <p>Possible values for the clock source are:</p> <p>0 <b>INT</b> Internal clock. <b>[REQUIRED]</b></p> <p>1 <b>INT1</b> Second internal clock. Optional.</p> <p>2 <b>INT2</b> Third internal clock. Optional.</p> <p>3 <b>INT3</b> Fourth internal clock. Optional.</p> <p>4 <b>TCN</b> External clock. Usually from the timing module.</p> <p>5 <b>EXT1</b> Second external clock. Optional.</p> <p>6 <b>EXT2</b> Third external clock. Optional.</p> <p>7 <b>EXT3</b> Fourth external clock. Optional.</p> <p>If the channel does not support a particular clock source, the alarm severity (<b>SEVR</b> field) of this record is set to <b>MINOR</b>, the alarm state (<b>STAT</b>) is set to <b>WRITE</b>, and the clock source remains unaffected.</p>
<b>setConversion</b>	<b>CVT</b>	<b>waveform</b> (double type)	<p>[<b>AI, AO</b>] Define conversion from raw values to engineering values as described in section <b>Error! Reference source not found.</b> Conversion is defined as a segmented cubic spline. aveform array consists of five-tuples, whose elements are the start of the segment, followed by the four cubic spline coefficients.</p> <p>If an empty array is given, conversion is performed according to the <b>LINR, EGUF</b> and <b>EGUL</b> fields of the <b>IN</b> and <b>OUT</b> records.</p>
<b>setCoupling</b>	<b>ACDC</b>	<b>bo</b>	<p>[<b>AI</b>] Set the coupling (AC or DC) of the analog input channel. Values are:</p> <p>0 <b>AC</b> Coupling for alternating current.</p> <p>1 <b>DC</b> Coupling for direct current.</p>

Function	PV suffix	Record type	Description
<b>setDecimationFactor</b>	<b>DECF</b>	<b>ao</b>	[AI, AO, DI, DO, DIO, C] Set decimation factor. For input channels, only every <b>DECF</b> -th sample (or frame) will be sampled. For output channels outputting a waveform, only every <b>DECF</b> -th sample will be output.
<b>setDecimationOffset</b>	<b>DECO</b>	<b>ao</b>	[AI, AO, DI, DO, DIO, C] Specify the index of the first sample that is not decimated.
<b>setDifferential</b>	<b>DIFF</b>	<b>bo</b>	[AI] Set the type of input: differential or single-ended.
<b>setDIODirection</b>	<b>DIR</b>	<b>bo</b>	[REQUIRED: DIO] For general purpose input/output digital channels, specifies whether the channel is an input or an output channel. Values are: 0 <b>IN</b> Input channel. 1 <b>OUT</b> Output channel.
<b>setFFTOverlap</b>	<b>FFTO</b>	<b>ao</b>	[AI] Configure the amount of overlap between consecutive frames for the Fourier transform.
<b>setFFTSize</b>	<b>FFTN</b>	<b>ao</b>	[AI] Configure size of the frame for the Fourier transform.
<b>setFFTSmoothing</b>	<b>FFTS</b>	<b>ao</b>	[AI] Configure the smoothing factor for averaging the Fourier transform.
<b>setFFTWindow</b>	<b>FFTW</b>	<b>mbbo</b>	[AI] Select the windowing function for the Fourier transform. Possible values are: 0 <b>NONE</b> No window (i.e., $window[i] = 1$ ). 1 <b>BARLETT</b> 2 <b>BLACKMAN</b> 3 <b>FLATTOP</b> 4 <b>HANN</b> 5 <b>HAMMING</b> 6 <b>TUKEY</b> 7 <b>WELCH</b>

Function	PV suffix	Record type	Description
<b>setFilter</b>	<b>FILT</b>	<b>waveform</b> (double type)	[AI] Set the filter for the analog input signal as described in section <b>Error! Reference source not found.</b> . If an empty array is given, the <b>SMOO</b> field of the <b>IN</b> record is used.
<b>setGain</b>	<b>GAIN</b>	<b>ao</b>	[AI, AO] 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 ( <b>SEVR</b> field) of this record is set to <b>MINOR</b> and the alarm state ( <b>STAT</b> ) is set to <b>WRITE</b> . The gain is set to the nearest supported smaller value.
<b>setGround</b>	<b>GND</b>	<b>bo</b>	[AI] Specify whether or not to ground the input.
<b>setHeight</b>	<b>HGHT</b>	<b>longin</b>	[C] Requested height of the camera image.
<b>setImageType</b>	<b>IMGT</b>	<b>mbbo</b>	[C] Type of the image's raw format. 0 <b>GRAYSCALE</b> Grayscale (implies <b>SPP=1</b> ). 1 <b>RGB</b> Red-green-blue (implies <b>SPP=3</b> ). 2 <b>BGR</b> Blue-green-red (implies <b>SPP=3</b> ). 3 <b>RGBG</b> Bayer format (implies <b>SPP=4</b> ). 4 <b>GRGB</b> Bayer format (implies <b>SPP=4</b> ). 5 <b>RGGB</b> Bayer format (implies <b>SPP=4</b> ).

Function	PV suffix	Record type	Description
<b>setOffset</b>	<b>OFS</b>	<b>ao</b>	<p>[AI, AO] Offset voltage of a channel.</p> <p>For analog input channels, the offset voltage is subtracted before it is amplified.</p> <p>For analog output channels, the offset voltage is added after it is amplified.</p> <p>If the channel does not support a particular offset, the alarm severity (<b>SEVR</b> field) of this record is set to <b>MINOR</b> and the alarm state (<b>STAT</b>) is set to <b>WRITE</b>. The offset is set to the nearest supported smaller value.</p>
<b>setOriginX</b>	<b>ORGX</b>	<b>longin</b>	[C] X coordinate of the image's upper-left corner (for cropping).
<b>setOriginY</b>	<b>ORGY</b>	<b>longin</b>	[C] Y coordinate of the image's upper-left corner (for cropping).
<b>setResolution</b>	<b>RES</b>	<b>longin</b>	[AI, C] Resolution of the channel (number of bits per sample). For camera, number of bits per color.
<b>setSignalAmplitude</b>	<b>SGNA</b>	<b>ao</b>	[AO] Amplitude of the output signal, e.g., the sine signal without offset would range from <b>-SGNA</b> to <b>+SGNA</b> . Amplitude is specified in terms of the output's engineering units.
<b>setSignalDutyCycle</b>	<b>SGND</b>	<b>ao</b>	[AO] 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 <i>high</i> state (pulse train) or rising (sawtooth).
<b>setSignalFrequency</b>	<b>SGNF</b>	<b>ao</b>	[AO] Frequency of those types of signal that are periodic. See the <b>SGNT</b> record for specification of the signal type. Frequency is specified in Hz as a floating point number.
<b>setSignalOffset</b>	<b>SGNO</b>	<b>ao</b>	[AO] Offset to add to the output signal, e.g., the sine signal would range from <b>SGNO-SGNA</b> to <b>SGNO+SGNA</b> . Offset is specified in terms of the output's engineering units.
<b>setSignalPhase</b>	<b>SGNP</b>	<b>ao</b>	[AO] Phase of the generated output signal. Phase is relative to the epoch of the time base (e.g., the TCN).

Function	PV suffix	Record type	Description
setSignalType	SGNT	mbbo	<p>[AO] Set the type of output signal that is generated by the output channel. It can be one of the following values:</p> <p>0 <b>WAVEFORM</b> Waveform as defined in WF record.</p> <p>1 <b>SPLINE</b> Spline (piecewise-cubic function) as defined in SPLN record</p> <p>2 <b>SIN</b> Sine curve (amplitude SGNA, offset SGNO, frequency SGNF, phase SGNP).</p> <p>3 <b>PULSE</b> Pulse train (amplitude SGNA, offset SGNO, frequency SGNF, phase SGNP, duty cycle SGNP).</p> <p>4 <b>SAWTOOTH</b> Sawtooth output (amplitude SGNA, offset SGNO, frequency SGNF, phase SGNP, rising for fraction SGNP, falling for fraction 1-SGNP).</p>
setTriggerChannel	TRGC	stringout	[AI, AO, DI, DO, DIO, C] Set another channel as the trigger. Please refer to section <b>Error! Reference source not found.</b> for syntax of this record.
setTriggerDelay	TRGD	ao	[AI, AO, DI, DO, DIO, C] Set the delay for the trigger. The delay is given as the number of seconds past the trigger event.
setTriggerRepeat	TRGR	longout	[AI, AO, DI, DO, DIO, C] Set the number of times that the trigger should be triggered.
setWidth	WDTH	longin	[C] Requested width of the camera image.
Spline	SPLN	waveform	[AO] Description of a spline to generate on the output channel (see section <b>Error! eference source not found.</b> ).
trigger	TRG	stringout	[AI, AO, DI, DO, DIO, C] Trigger data acquisition immediately or at specified time.
Value	OUT	ao, bo	[AO, DO] Set a value of the output channel. The value is subject to conversion from engineering units to raw units.

Function	PV suffix	Record type	Description
ValueFloat64	IN	ai, bi	[AI, DI, DIO] Read the current value of the channel. The value is subject to filtering and conversion from raw units to engineering units.

## A.2. Device Functions

Function	PV suffix	Record type	Description
(getState)	(STAT)	mbbi	<p>[REQUIRED] The state of the device. The following values are possible:</p> <ul style="list-style-type: none"> <li>0 <b>OFF</b> The device is offline – even powered off, if possible by the hardware. To perform power cycling, first set the device state to <b>OFF</b>, and then back to <b>ON</b>.</li> <li>1 <b>ON</b> The device is ready to perform. <b>Note:</b> setting the state to <b>ON</b> might not be immediate. If device initialization is not immediate, the device will pass through the <b>INIT</b> state.</li> <li>2 <b>RESET</b> Reset the device. When set, a <i>soft reset</i> of the device will be performed. If device takes longer time to reset, it might pass through the <b>INIT</b> state.</li> <li>3 <b>INIT</b> The device is initializing. When initialization is complete, it will go to the <b>ON</b> state. All settings will be reset to initial/default values. The device is in the <b>INIT</b> 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.</li> <li>4 <b>ERROR</b> A device-level error occurred. The device needs to be reset.</li> <li>5 <b>DEFUNCT</b> 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.</li> </ul>

Function	PV suffix	Record type	Description
			In the <b>ERROR</b> and <b>DEFUNCT</b> states, alarm severity ( <b>SEVR</b> field) is set to <b>MAJOR</b> and alarm state ( <b>STAT</b> ) is set to <b>STATE</b> . Additional information about the error can be found in the IOC's error log.
<b>getModel</b> <b>getSerial</b> <b>getHardwareRevision</b> <b>getFirmwareVersion</b> <b>getSoftwareVersion</b>	<b>IMDL</b> <b>ISN</b> <b>IHW</b> <b>IFW</b> <b>ISW</b>	<b>stringin</b>	<p>[<b>REQUIRED</b>: model and software info] Retrieve information about the device.</p> <p><b>IMDL</b> is the model of the device. <b>ISN</b> is the device serial number. <b>IHW</b> is the hardware revision ID. <b>IFW</b> is the firmware version number. <b>ISW</b> is the software (device driver) version number.</p> <p><b>Note for device driver implementers:</b> rather than overriding the <b>get*</b> functions, set the <b>modelName</b>, <b>serialNumber</b>, <b>hardwareVersion</b>, <b>firmwareVersion</b> and <b>softwareVersion</b> fields of the <b>NDSDevice</b> structure during driver initialization.</p>
<b>receiveMessage</b>	<b>MSGR</b>	<b>stringin</b>	Retrieve the last message from the device. The message is of the format <message> <correlation-id>
<b>sendMessage</b>	<b>MSGS</b>	<b>stringout</b>	<p>Send a message to the device. Message string has the format:</p> <p><b>&lt;message&gt; [&lt;correlation-id&gt; [&lt;param&gt; [&lt;param&gt;] ...]]</b></p> <p>where &lt;message&gt; is the type of the message to convey to the device. &lt;correlation-id&gt; 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.</p> <p>Standardized messages are:</p> <p><b>STOP</b> Stop any action that is in progress at the device (e.g., test execution).  <b>TEST-QUICK</b> Perform a quick self-test of the device. <b>TEST-FULL</b> Perform an intensive self-test of the device. The test checks all the test-points defined by the device, therefore it may take a longer time to complete.</p>

Function	PV suffix	Record type	Description
			If during a test, problems are found, the state of the device (STAT record) is set to the <b>ERROR</b> or <b>DEFUNCT</b> state. More information can be found in the IOC error log.
<b>setClockFrequency</b>	<b>CLKF</b>	<b>ao</b>	<p>[AI, AO, DI, DO, DIO, C] Set the frequency of the internal clock selected with <b>CLKSRC</b> record. The frequency is specified in Hz.</p> <p>For camera channels, this parameter specifies the number of frames per second.</p> <p>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 (<b>SEVR</b> field) of this record is set to <b>MINOR</b> and the alarm state (<b>STAT</b>) is set to <b>WRITE</b>. If internal clock source is selected, the frequency is set to the nearest smaller frequency.</p>
<b>setEmulation</b>	<b>EMUL</b>	<b>bo</b>	Allow ( <b>TRUE</b> ) or disallow ( <b>FALSE</b> ) software emulation of functions that are not supported by hardware.

## Appendix B LIST OF DOCALLBACKS FUNCTIONS

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

```
/**
 * Event dispatching functions.
 * These 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
 */

/**
 * asynInt8Array interrupt dispatching function.
 * Dispatch Int8 array value to registered handlers.
 * \param value is a new data (array)
 * \param nElements is a number of elements in the array
 * \param reason is an interruptId of the reason code to
process new data
 * \param addr is a port address to select correct value
 * \param timestamp is a timestamp value to update record
process time
 * \return It returns status of the operation. If operation
success it returns ndsSuccess vice versa ndsError
 *
 */
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);

/**
 * asynInt16Array interrupt dispatching functions
 * Dispatch Int16 array value to registered handlers.
 */
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);

/**
 * asynInt32Array interrupt dispatching function
 * Dispatch Int32 array value to registered handlers.
 */
    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);

/**
 * asynFloat32Array interrupt dispatching function
 * Dispatch Float32 array value to registered handlers.
 */
    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);

/**
 * asynFlaot64Array interrupt dispatching function
 * Dispatch Float64 array value to registered handlers.
 */
    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);

/**
 * GenericPointer interrupt dispatching function
 * \param pointer is an new data
 * \param reason is a reason code (numeric) to process new
data
 * \param addr is a port address to select handler for the
reason
 * \return It returns status of the operation

```

```

*
* Reason and port address are used to select appropriate
handler to process data.
*/
    ndsStatus doCallbacksGenericPointer(void *pointer, int
reason, int addr);
    ndsStatus doCallbacksGenericPointer(void *pointer, int
reason);

/**
* asynInt32 interrupt dispatching function
* Dispatch Int32 value to registered handlers.
*/
    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);

/**
* asynFloat64 interrupt dispatching function
* Dispatch Flaot64 value to registered handlers.
*/
    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);

/**
* asynOctet interrupt dispatching function.
* Dispatch Octet value to registered handlers.
*/
    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);
```

## Appendix 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 12, where areaDetector's ROI and TIFF plugins make use of an image captured by an NDS image channel.

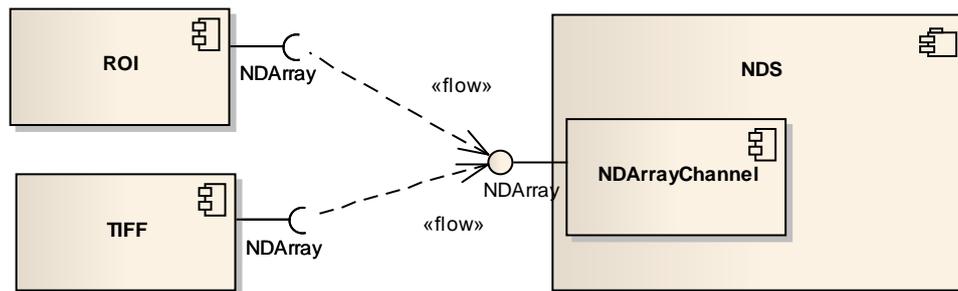


Figure 12: Run-time data flow involving AreaDetector plugins.

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.

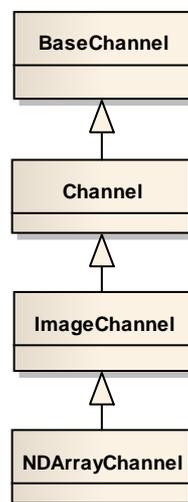


Figure 13: NDS image channel class hierarchy.

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 14.

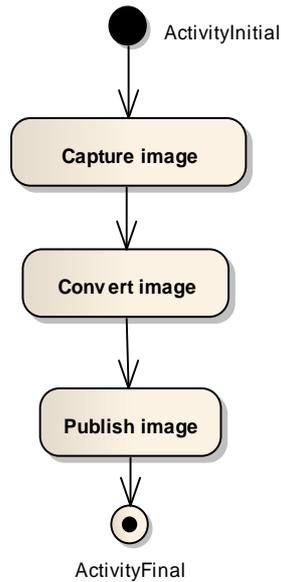


Figure 14: Activity diagram for publishing images to AreaDetector.

## 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>
- **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` files 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.1.1. Building NDS with areaDetector Support

Define the path to areaDetector and the required modules in the configuration file `epics-nds-2.0/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.1.2. Building 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).

## Appendix 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 15.

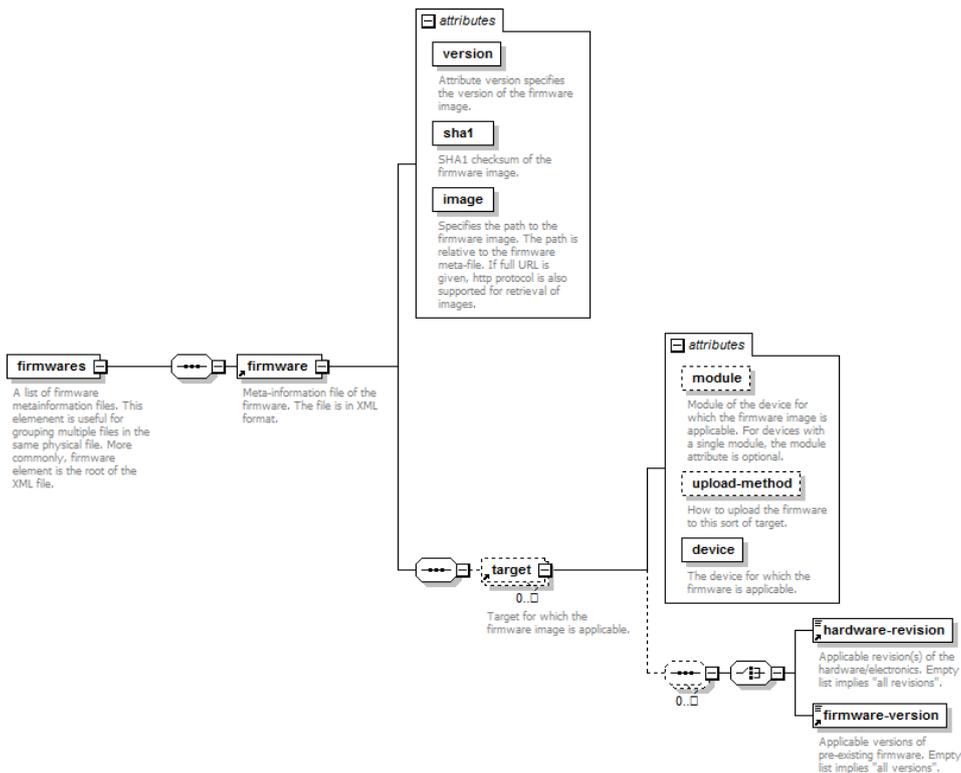


Figure 15: Schema of the firmware meta-information file.

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

```

<!--
! Meta-information file of the firmware. The file is in XML
format.
!
! The root element's image attribute specifies the path to the
firmware
! image. The path is relative to the firmware meta-file. If full
URL
! is given, http protocol is also supported for retrieval of
images.
!
! Attribute version specifies the version of the firmware image.
!-->
<firmware image="http://server/ni6529-fw-2.3.1.bin"
version="2.3.1"

xmlns="http://www.cosylab.com/NDS/FirmwareMetaInfo/2012">
<!--
! device and module refer to the device and its

```

```
! module(s) for which the firmware is applicable.
! For devices with a single module, the module
! attribute is optional.
!-->
<target device="NI6529" module="fpga-3">
  <!--
    ! Applicable revision(s) of the hardware/electronics.
    ! Empty list implies "all revisions".
    !-->
    <hardware-revision>rev0</hardware-revision>
    <hardware-revision>rev1</hardware-revision>
  <!--
    ! Applicable versions of pre-existing firmware.
    ! Empty list implies "all versions".
    !-->
    <firmware-version>2.2.0</firmware-version>
    <firmware-version>2.2.1</firmware-version>
    <firmware-version>2.2.2</firmware-version>
  </target>
</firmware>
```

## Appendix E    **TEMPLATE FOR DEVICE-SPECIFIC DOCUMENTATION**

<b>Function</b>	<b>Support Level<sup>1</sup></b>	<b>Comment</b>
	F/P/D/S	

---

<sup>1</sup>Support level column has the following meaning: F – full compliance to the Nominal Device Model; P - partial compliance to the NDM (comment column must have details); D – default Nominal device Model implementation (e.g.: software simulation); S- device-specific function.