

# XAL Accelerator Class Hierarchy

## 1 Introduction

XAL is a Java class hierarchy representing an accelerator. It is meant to provide a programming framework with a “physical” view of the accelerator, as viewed from an accelerator physics perspective. For example, the physicist thinks of the accelerator as having “sequences” which are in turn composed of beamline elements. Generally, the beamline elements consist of different kinds of magnets, RF cavities and diagnostics. The magnets consist of dipoles, quadrupoles, etc. Within this accelerator framework, for each beamline element type abstract methods are provided, so that the programmer does not need to know specifics about element or signal names.

There are two primary external sources of information for this framework: 1) a database containing the static accelerator configuration (e.g. lattice configuration of an accelerator sequence), and 2) the EPICS control system providing the real time values of dynamic accelerator properties (e.g. BPM position reading).

This document is not meant to be a comprehensive reference, but rather tries to give general information about the class structure, and explain meaning and usage by example snippets. For detailed class views, member definitions etc., the JavaDoc manual [1.1] should be referred to.

## 2 Overall class structure

The XAL class structure is shown at a high level in Fig. 1.1 . Each of these branches is described in more detail in the sections below, and a general description is provided here.

### 2.1 smf Package

The “standard machine format” or “smf” package contains classes that describe the basic accelerator framework, namely the accelerator, accelerator sequence and accelerator nodes. Several other packages are contained in the smf package, as described below.

#### *impl Package*

This package contains the actual implementations of the abstract AcceleratorNode class. For example, implementations such as quadruples, beam position monitors, etc. These classes describe the components that actually make up an accelerator beamline. The unique features and methods of each beamline type device are contained in the appropriate class.

#### *xdxf Package*

The xdx package contains services useful for parsing the smf structure from an XML structured file.

#### *chan Package*

This package contains important links between the accelerator class structure and the actual machine. Information from the machine (magnets, diagnostics, rf, etc.) is provided by the EPICS control system, in particular through the EPICS channel

access client<sup>1</sup>). The EPICS interface is on a signal-by-signal basis, with no hierarchy other than possible naming conventions of the signals. The mapping of the signals to common interfaces for AcceleratorNode types is done in this package. Each AcceleratorNode type is assigned a ChannelSuite, which is a collection of channels, with built-in interfaces to hide the actual mechanics of making connections to the EPICS control system.

#### *attr Package*

The attribute package, which is a mechanism to hold “attribute sets” of related information for different AcceleratorNode types. For instance, all AcceleratorNodes have an alignment attribute-set containing information about their installed position relative to the design position. Also, all objects of the Magnet sub-class have an attribute set containing information about their multipole field levels. Having a generalized attribute-set container for related information of a certain class offers advantages in parsing XML files.

## **2.2 XAL Channel Access**

Within the XAL framework, channel access to EPICS is handled in the `xal.ca` package. This package makes extensive use of the `jca` Java package by Eric Boucher. The `jca` package is essentially a JNI (Java Native Interface) wrapper to the EPICS channel access library written in C. Thus in the current version of XAL, it is necessary to have `jca` installed on the target platform, with all its required shared libraries and corresponding EPICS channel access shared libraries.

Although channel access in XAL (i.e., the `xal.ca` package) sits atop the `jca` package, the later is essentially hidden from the user. Thus, one needs to know nothing of `jca` to use `xal.ca`. This condition is also present to facilitate future upgrades. The current architecture is one that was quickest to implement. For performance reasons, it may be necessary to modify, or even remove, the `jca` package. The `xal.ca` package presents a clean, object-oriented picture of channel access. As such, future modifications can be made without disrupting the user’s view of channel access.

### **2.2.1 The Channel Class**

Most all of channel access in XAL is handled through the `Channel` class. This class

As a default, the `ca_pend_io()` and `ca_pend_event()` functions of EPICS channel access are called after every operation that involves channel access. Thus, it appears to the users of `xal.ca` that the channel access requests are queued and flushed automatically in one get or put operation. If this becomes a performance issue, it is possible to stop this automatic flushing of the channel access request queue by calling the `Channel` class method `setSyncRequest(true)`. After this call, the request buffer must be flushed externally using the static methods `Channel.pendIO(double)` or `Channel.pendEvent(double)`, which ever is appropriate.

---

<sup>1</sup> Following EPICS nomenclature, we refer to each individual “signal” from the machine as a channel. This is also referred to as a process variable or PV sometimes.

In the case of monitors, a polling daemon is started which continually polls EPICS channel access for monitor events. Once all monitors have been stopped, the daemon is automatically terminated and polling stops. One may access the polling daemon directly using the static methods of the class `PendDaemon`.

### 2.2.1.1 A Note About Data Types

When retrieving data from the `Channel` class, or EPICS channel access in general, you can receive the data in a variety of different data types and shapes. However, the EPICS database record describing the process variable determines the native type of that channel. When asking for the channel value as data type other than that of the native type, the value is formatted to the requested type on the IOC then sent to the requestor. Thus, additional CPU time is consumed on the IOC casting the value to the appropriate type. This additional CPU burden might cause performance problems on a heavily loaded IOC. It is typically safest to ask for the value of the process variable in its native type format, and then cast to the desired type on the client computer. The `Channel` method `nativeType()` returns an enumerated description of the channel's native type on the IOC computer.

### 2.2.1.2 Connecting

To connect to a process variable (PV) on an EPICS IOC the `Channel` class must be supplied with the EPICS channel name. This can be accomplished with either the initializing constructor `Channel(String)` or by using the `setChannelName(String)` method. For example, the following code excerpt creates a `Channel` object, connects to the EPICS process variable labeled "MEBT\_Mag:QH01:fieldRB", then disconnects:

```
Channel    chan = new Channel("MEBT_Mag:QH01:fieldRB");
Boolean    bResult;

bResult = chan.connect();
if (!bResult) {
    System.out.println("could not connect");
    return;
} else
    System.out.println("connected to channel!");
chan.disconnect();
return;
}
```

### 2.2.1.3 Process Variable Get's

To get the value of a process variable, the `Channel` class provides a suite of methods. These methods have the form `getXY()`, where X is the shape of the data (either scalar or array) and Y is the type of the data (e.g., byte, int, float).

```
fltVal = m_chan.getValFlt();
m_chan.getValFltCallback(this);
```

#### 2.2.1.4 Notifications

It is possible to receive notifications from channel connections and process variable puts. To do this use the callback form of the appropriate Channel method and the IEventSinkNotify interface. For example, suppose you have created a class Notifications that supports the IEventSinkNotify interface. In order to set a process variable, represented by the object chan, to the value 2.13 and have notification sent to an instance of Notifications you could use the following code excerpt:

```
Notifications notes = new Notifications();
chan.putValCallback(2.13, notes);
```

Channel access would then call the method eventNotify(Channel, int) of the object notes after the put operation was completed.

Notice that the same class, Notifications, could be used to handle a connection notification event. The only requirement is that the class supports the IEventNotifySink interface, what the class does with the notification is arbitrary. Such a situation might look like the following:

```
Channel      chan = new
Channel("MEBT_Mag:QH01:fieldRB");
Notification notes = new Notification();
Boolean      bResult;

bResult = chan.connectCallback(notes);
```

#### 2.2.1.5 Monitors

To monitor a process variable is to receive continual updates of its value. Setting an appropriate parameter in the monitor controls the conditions as to what time or condition the updates come. (Also, the EPICS database record can contain information specific to these updates.) Monitoring in XAL is accomplished by calling the appropriate addMonitorXXX() method of the Channel class, and maintaining a reference to the Monitor object returned by that call. As in the situation of process variable gets, there exists one form of the addMonitorXXX() method for each type of EPICS data supported. For example, to monitor a process variable and receive the data as an array of integers, you would use addMonitorArrInt().

This class provides an object view of an EPICS channel access point. ... Insert Chris Allen's input)

### 3 *Units and coordinate systems*

In general MKS units are employed. Table 1.1 lists the units for many parameters. Note that while the unit for length is m, many displays may convert to mm for display purposes. Also, the units for values coming from the EPICS control system may not be

the same as described here – in this case conversions are done for the internal XAL representation.

	<u>Quantity</u>	<u>Unit</u>
Length		m (along the reference orbit + global coordinates) mm (transverse direction from the beam)
Time		sec
Angle		degrees
Voltage (e.g. RF)		kV
Magnetic Field		T
Current		A

## Magnetic Fields

The multipole fields in the magnets are characterized by:

$$B_{normal} = \sum_{n=1} B_n \left( \frac{r}{R_{ref}} \right)^{n-1}$$

$$B_{skew} = \sum_{n=1} A_n \left( \frac{r}{R_{ref}} \right)^{n-1}$$

where  $B_n$  and  $A_n$  have units of (T/m<sup>n-1</sup>) and n=1 refers to the dipole field<sup>2</sup>. Methods that return field quantities (i.e. *getField()* methods) return only the primary normal (skew) component for normal (skew) magnets. The effective magnetic length is not included in these return values.

## 4 References

1.1 – The XAL JavaDoc Manual – see

<http://www.sns.gov/APGroup/appProg/xalDoc/index.html>

---

<sup>2</sup> . Note that the magnet measurement data are typically for integrated field quantities, and have the additional magnetic length included.

