areaDetector: EPICS Area Detector Support Documentation

Release 3-1

University of Chicago

Jul 14, 2017

Contents

Release 3-1

July 3, 2017

Mark Rivers

University of Chicago

CHAPTER 1

Content

Overview

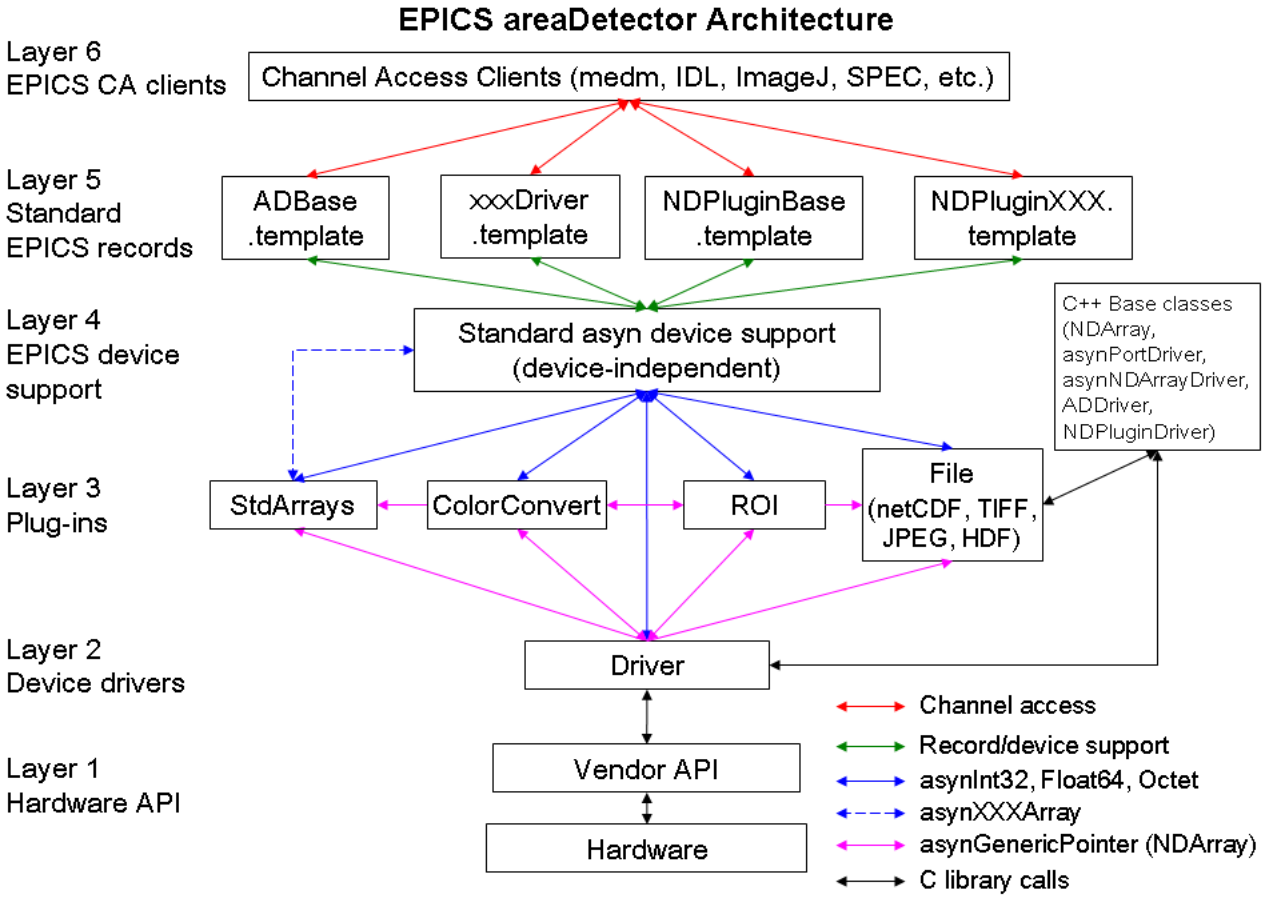
The areaDetector module provides a general-purpose interface for area (2-D) detectors in EPICS. It is intended to be used with a wide variety of detectors and cameras, ranging from high frame rate CCD and CMOS cameras, pixel-array detectors such as the Pilatus, and large format detectors like the Perkin Elmer flat panels.

The goals of this module are:

- Minimize the amount of code that needs to be written to implement a new detector.
- Provide a standard interface defining the functions and parameters that a detector driver must support.
- Provide a set of base EPICS records that will be present for every detector using this module. This allows the use of generic EPICS clients for displaying images and controlling cameras and detectors.
- Allow easy extensibility to take advantage of detector-specific features beyond the standard parameters.
- Have high-performance. Applications can be written to get the detector image data through EPICS, but an interface is also available to receive the detector data at a lower-level for very high performance.
- Provide a mechanism for device-independent real-time data analysis such as regions-of-interest and statistics.
- Provide detector drivers for commonly used detectors in synchrotron applications. These include Prosilica GigE video cameras, IEEE 1394 (Firewire) cameras, ADSC and MAR CCD x-ray detectors, MAR-345 online imaging plate detectors, the Pilatus pixel-array detector, Roper Scientific CCD cameras, Perkin-Elmer amorphous silicon detector, and many others.

Architecture

The architecture of the areaDetector module is shown below.



From the bottom to the top this architecture consists of the following:

- Layer 1. This is the layer that allows user written code to communicate with the hardware. It is usually provided by the detector vendor. It may consist of a library or DLL, of a socket protocol to a driver, a Microsoft COM interface, etc.
- Layer 2. This is the driver that is written for the areaDetector application to control a particular detector. It is written in C++ and inherits from the ADDriver class. It uses the standard asyn interfaces for control and status information. Each time it receives a new data array it can pass it as an NDArray object to all Layer 3 clients that have registered for callbacks. This is the only code that needs to be written to implement a new detector. Existing drivers range from about 800 to 2600 lines of code.
- Layer 3. Code running at this level is called a "plug-in". This code registers with a driver for a callback whenever there is a new data array. The existing plugins implement file saving (NDPluginFile), region-of-interest (ROI) calculations (NDPluginROI), statistics calculations (NDPluginStats, DNPluginROIStat), image processing (NDPluginProcess), geometric transformations (NDPluginTransform), buffering and triggering (NDPluginCircularBuff), color mode conversion (NDPluginColorConvert), graphics and text overlays (NDPluginOverlay), exporting NDArray attributes as scalar and waveform records (NDPluginAttribute), and conversion of detector data to standard EPICS array types for use by Channel Access clients (NDPluginStdArrays). Plugins are written in C++ and inherit from NDPluginDriver. Existing plugins range from about 300 to 3000 lines of code.
- Layer 4. This is standard asyn device support that comes with the EPICS asyn module.
- Layer 5. These are standard EPICS records, and EPICS database (template) files that define records to communicate with drivers at Layer 2 and plugins at Layer 3.

- Layer 6. These are EPICS channel access clients, such as MEDM that communicate with the records at Layer 5. areaDetector includes two client applications that can display images using EPICS waveform and other records communicating with the NDPluginStdArrays plugin at Layer 3. One of these clients is an ImageJ plugin, and the other is a freely runnable IDL application.

The code in Layers 1-3 is essentially independent of EPICS. In principle there are only 2 EPICS dependencies in this code.

1. libCom from EPICS base provides operating-system independent functions for threads, mutexes, etc.
2. asyn is a module that provides interthread messaging services, including queueing and callbacks.

In particular it is possible to eliminate layers 4-6 in the architecture shown in Figure 1. This means that it is not necessary to run an EPICS IOC or to use EPICS Channel Access when using the drivers and plugins at Layers 2 and 3. This is demonstrated in the simDetectorNoIOC application in ADSimDetector and in the unit tests in ADCore/ADApp/pluginTests.

The plugin architecture is very powerful, because new plugins can be written for application-specific purposes. For example, a plugin could be written to analyze images and do some application specific functions, and such a plugin would then work with any detector driver. Plugins are also powerful because they can be reconfigured at run-time. For example the NDPluginStdArrays can switch from getting its array data from a detector driver to an NDPluginROI plugin. That way it will switch from displaying the entire detector to whatever sub-region the ROI driver has selected. Any Channel Access clients connected to the NDPluginStdArrays driver will automatically switch to displaying this subregion. Similarly, the NDPluginFile plugin can be switched at run-time from saving the entire image to saving a selected ROI, just by changing its input source. Plugins can be used to form an image processing pipeline, for example with a detector providing data to a color convert plugin, which feeds an ROI plugin, which feeds a file saving plugin. Each plugin can run in its own threads, and hence in its own cores on a modern multi-core CPU.

The use of plugins is optional, and it is only plugins that require the driver to make callbacks with image data. If there are no plugins being used then EPICS can be used simply to control the detector, without accessing the data itself. This is most useful when the vendor provides an API has the ability to save the data to a file and an application to display the images.

What follows is a detailed description of the software, working from the bottom up. Most of the code is object oriented, and written in C++.

Implementation Details

The areaDetector module depends heavily on asyn. It is the software that is used for interthread communication, using the standard asyn interfaces (e.g. asynInt32, asynOctet, etc.), and callbacks. In order to minimize the amount of redundant code in drivers, areaDetector has been implemented using C++ classes. The base classes, from which drivers and plugins are derived, take care of many of the details of asyn and other common code.

asynPortDriver

Detector drivers and plugins are asyn port drivers, meaning that they implement one or more of the standard asyn interfaces. They register themselves as interrupt sources, so that they do callbacks to registered asyn clients when values change. They inherit from the asynPortDriver base C++ class that is provided in the asyn module. That base class handles all of the details of registering the port driver, registering the supported interfaces, and registering the required interrupt sources. It also provides a parameter library for int, double, and string parameters indexed by the integer index values defined in the driver. The parameter library provides methods to write and read the parameter values, and to perform callbacks to registered clients when a parameter value has changed. The asynPortDriver class documentation describes this class in detail.

NDArray

The NDArray (N-Dimensional array) is the class that is used for passing detector data from drivers to plugins. An NDArray is a general purpose class for handling array data. An NDArray object is self-describing, meaning it contains enough information to describe the data itself. It can optionally contain "attributes" (class NDAttribute) which contain meta-data describing how the data was collected, etc.

An NDArray can have up to ND_ARRAY_MAX_DIMS dimensions, currently 10. A fixed maximum number of dimensions is used to significantly simplify the code compared to unlimited number of dimensions. Each dimension of the array is described by an NDDimension structure. The NDArray class documentation describes this class in detail.

NDArrayPool

The NDArrayPool class manages a free list (pool) of NDArray objects. Drivers allocate NDArray objects from the pool, and pass these objects to plugins. Plugins increase the reference count on the object when they place the object on their queue, and decrease the reference count when they are done processing the array. When the reference count reaches 0 again the NDArray object is placed back on the free list. This mechanism minimizes the copying of array data in plugins. The NDArrayPool class documentation describes this class in detail.

NDAttribute

The NDAttribute is a class for linking metadata to an NDArray. An NDattribute has a name, description, data type, value, source type and source information. Attributes are identified by their names, which are case-sensitive. There are methods to set and get the information for an attribute.

It is useful to define some conventions for attribute names, so that plugins or data analysis programs can look for a specific attribute. The following are the attribute conventions used in current plugins:

Conventions for standard attribute names		
Attribute name	Description	Data type
ColorMode	"Color mode"	int (NDColorMode_t)
BayerPattern	"Bayer pattern"	int (NDBayerPattern_t)

Attribute names are case-sensitive. For attributes not in this table a good convention would be to use the corresponding driver parameter without the leading ND or AD, and with the first character of every "word" of the name starting with upper case. For example, the standard attribute name for ADManufacturer should be "Manufacturer", ADNumExposures should be "NumExposures", etc.

The NDAttribute class documentation describes this class in detail.

NDAttributeList

The NDAttributeList implements a linked list of NDAttribute objects. NDArray objects contain an NDAttributeList which is how attributes are associated with an NDArray. There are methods to add, delete and search for NDAttribute objects in an NDAttributeList. Each attribute in the list must have a unique name, which is case-sensitive.

When NDArrays are copied with the NDArrayPool methods the attribute list is also copied.

IMPORTANT NOTE: When a new NDArray is allocated using NDArrayPool::alloc() the behavior of any existing attribute list on the NDArray taken from the pool is determined by the value of the global variable eraseNDAttributes. By default the value of this variable is 0. This means that when a new NDArray is allocated from the pool its attribute list is **not** cleared. This greatly improves efficiency in the normal case where attributes for a given driver are defined once at initialization and never deleted. (The attribute **values** may of course be changing.) It eliminates allocating and deallocating attribute memory each time an array is obtained from the pool. It is still possible to add new attributes

to the array, but any existing attributes will continue to exist even if they are ostensibly cleared e.g. asynNDArrayDriver::readNDAttributesFile() is called again. If it is desired to eliminate all existing attributes from NDArrays each time a new one is allocated then the global variable eraseNDAttributes should be set to 1. This can be done at the iocsh prompt with the command:

```
var eraseNDAttributes 1
```

The NDAttributeList class documentation describes this class in detail.

PVAttribute

The PVAttribute class is derived from NDAttribute. It obtains its value by monitor callbacks from an EPICS PV, and is thus used to associate current the value of any EPICS PV with an NDArray. The PVAttribute class documentation describes this class in detail.

paramAttribute

The paramAttribute class is derived from NDAttribute. It obtains its value from the current value of a driver or plugin parameter. The paramAttribute class is typically used when it is important to have the current value of the parameter and the value of a corresponding PVAttribute might not be current because the EPICS PV has not yet updated. The paramAttribute class documentation describes this class in detail.

functAttribute

The functAttribute class is derived from NDAttribute. It obtains its value from a user-written C++ function. The functAttribute class is thus very general, and can be used to add almost any information to an NDArray. ADCore contains example code, myAttributeFunctions.cpp that demonstates how to write such functions. The functAttribute class documentation describes this class in detail.

asynNDArrayDriver

asynNDArrayDriver inherits from asynPortDriver. It implements the asynGenericPointer functions for NDArray objects. This is the class from which both plugins and area detector drivers are indirectly derived. The asynNDArrayDriver class documentation describes this class in detail.

The file asynNDArrayDriver.h defines a number of parameters that all NDArray drivers and plugins should implement if possible. These parameters are defined by strings (drvInfo strings in asyn) with an associated asyn interface, and access (read-only or read-write). There is also an integer index to the parameter which is assigned by asynPortDriver when the parameter is created in the parameter library. The EPICS database NDArrayBase.template provides access to these standard driver parameters. The following table lists the standard driver parameters. The columns are defined as follows:

- **Parameter index variable:** The variable name for this parameter index in the driver. There are several EPICS records in ADBase.template that do not have corresponding parameter indices, and these are indicated as Not Applicable (N/A).
- **asyn interface:** The asyn interface used to pass this parameter to the driver.
- **Access:** Read-write (r/w) or read-only (r/o).
- **drvInfo string:** The string used to look up the parameter in the driver through the drvUser interface. This string is used in the EPICS database file for generic asyn device support to associate a record with a particular parameter. It is also used to associate a paramAttribute with a driver parameter in the XML file that is read by asynNDArrayDriver::readNDAttributesFile

- **EPICS record name:** The name of the record in ADBase.template. Each record name begins with the two macro parameters $(P) and $(R). In the case of read/write parameters there are normally two records, one for writing the value, and a second, ending in _RBV, that contains the actual value (Read Back Value) of the parameter.
- **EPICS record type:** The record type of the record. Waveform records are used to hold long strings, with length (NELM) = 256 bytes and EPICS data type (FTVL) = UCHAR. This removes the 40 character restriction string lengths that arise if an EPICS "string" PV is used. MEDM allows one to edit and display such records correctly. EPICS clients will typically need to convert such long strings from a string to an integer or byte array before sending the path name to EPICS. In IDL this is done as follows:

```
; Convert a string to a null-terminated byte array and write with caput
IDL> t = caput('13PS1:TIFF1:FilePath', [byte('/home/epics/scratch'),0B])
; Read a null terminated byte array
IDL> t = caget(`13PS1:TIFF1:FilePath`, v)
; Convert to a string
IDL> s = string(v)
```

In SPEC this is done as follows:

```
array _temp[256]
# Setting the array to "" will zero-fill it
_temp = ""
# Copy the string to the array.  Note, this does not null terminate, so if array␣
↪already contains
# a longer string it needs to first be zeroed by setting it to "".
_temp = "/home/epics/scratch"
epics_put("13PS1:TIFF1:FilePath", _temp)
```

Note that for parameters whose values are defined by enum values (e.g NDDataType, NDColorMode, etc.), drivers can use a different set of enum values for these parameters. They can override the enum menu in ADBase.template with driver-specific choices by loading a driver-specific template file that redefines that record field after loading ADBase.template.

	Parameter Definitions in asynNDArrayDriver.h and EPICS Record Definitions in NDArrayBase.template (file-relat				
		Information about the version of ADCore and the plugin or driver			
Parameter index variable	**asyn interface**	**Access**	**Description**	**drvInfo string**	**EPICS record name**
NDADCoreVersion	asynOctet	r/o	ADCore version number. This can be used by Channel Access clients to alter their behavior depending on the version of ADCore that was used to build this driver or plugin.	ADCORE_VERSION	(P)(R)ADCoreVersion_RBV
NDDriverVersion	asynOctet	r/o	Driver or plugin version number. This can be used by Channel Access clients to alter their behavior depending on the version of the plugin or driver.	DRIVER_VERSION	(P)(R)DriverVersion_RBV
		Information about the asyn port			
NDPortNameSelf	asynOctet	r/o	Aasyn port name	PORT_NAME_SELF	(P)(R)PortName_RBV
		Data Type			
NDDataType	asynInt32	r/w	Data type (NDDataType_t).	DATA_TYPE	(P)(R)DataType
					(P)(R)DataType_RBV
		Color Mode			
NDColorMode	asynInt32	r/w	Color mode (NDColorMode_t).	COLOR_MODE	(P)(R)ColorMode
					(P)(R)ColorMode_RBV
NDBayerPattern	asynInt32	r/o	Bayer pattern (NDBayerPattern_t) of NDArray data.	BAYER_PATTERN	(P)(R)BayerPattern_RBV

Detector Drivers

Here is an example of linking documentation across repositories.

Credits

Citations

We kindly request that you cite the following article *[A1]* if you use project.

References

Bibliography

[A1] Mark L. Rivers. Areadetector: software for 2d detectors in epics. *AIP Conference Proceedings*, 1234(1):51–54, 2010. URL: http://aip.scitation.org/doi/abs/10.1063/1.3463256, doi:10.1063/1.3463256.

[B1] Mark L. Rivers. Areadetector: software for 2d detectors in epics. *AIP Conference Proceedings*, 1234(1):51–54, 2010. URL: http://aip.scitation.org/doi/abs/10.1063/1.3463256, doi:10.1063/1.3463256.