areaDetector: EPICS software for area detectors

Table of Contents

areaDetector: EPICS Area Detector Support

Release 1-7

August 9, 2011

Mark Rivers

University of Chicago

Contents

Overview

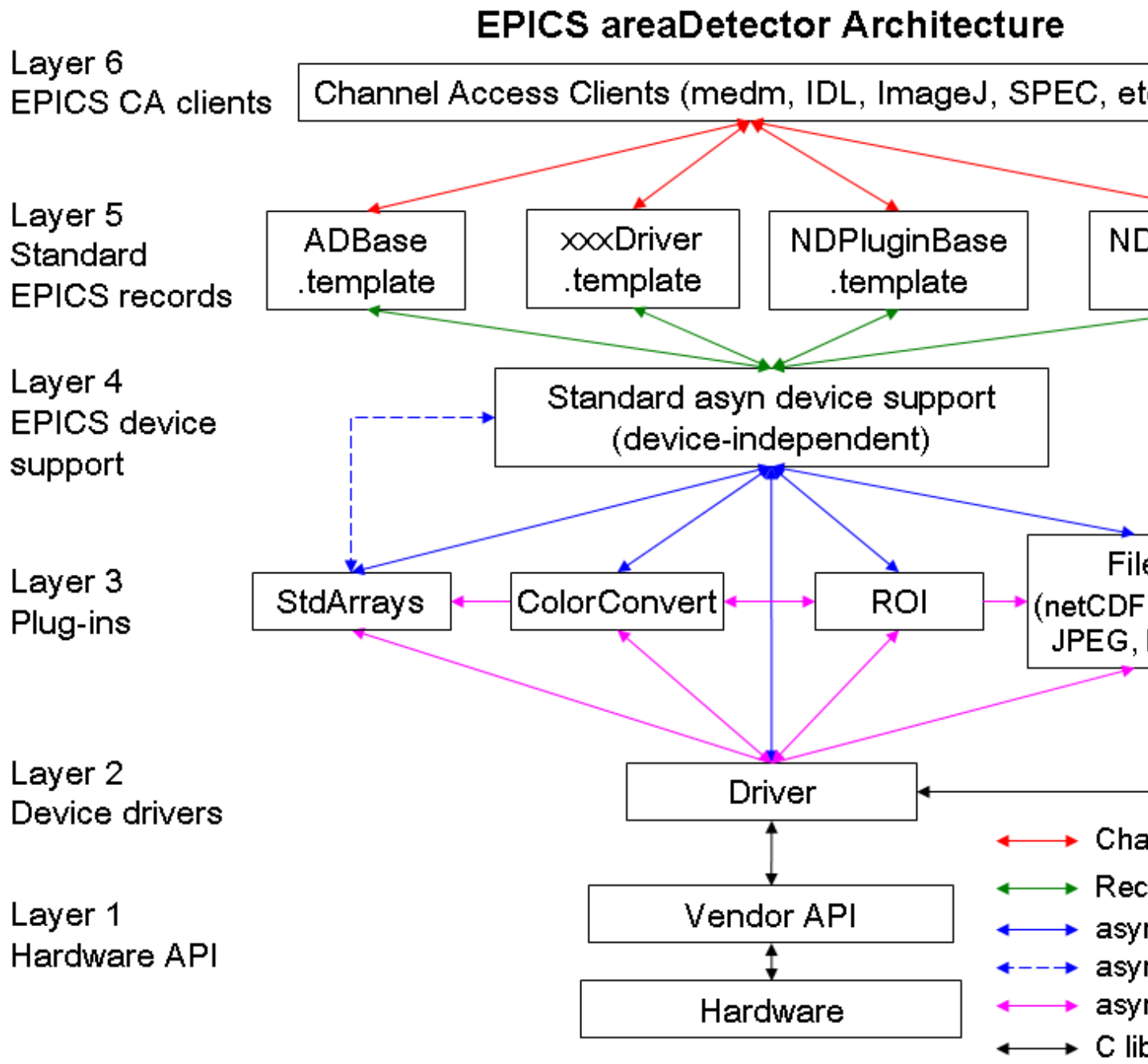
The areaDetector module provides a general-purpose interface for area (2-D) detectors in EPICS. It is intended to be used with a wide variety of detectors and cameras, ranging from high frame rate CCD and CMOS cameras, pixel-array detectors such as the Pilatus, and large format detectors like the MAR-345 online imaging plate.

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, and the Perkin-Elmer amorphous silicon detector.

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 800 to 1800 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), color mode conversion (NDPluginColorConvert), and conversion of detector data to standard EPICS array types for use by Channel Access clients (NDPluginStdArrays). Plugins are normally written in C++ and inherit from NDPluginDriver. Existing plugins range from 300 to 800 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. There are only 2 EPICS dependencies in this code.

1. libCom. libCom from EPICS base provides operating-system independent functions for threads, mutexes, etc.
2. asyn. 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, providing there is a programs such as the high-performance GUI shown in Layer 3. 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.

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 find the center of the beam, 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 thread, and hence in its own core 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-insensitive. 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-insensitive in the areaDetector software, but external software may not be case-insensitive so the attribute names should generally be used exactly as they appear above. 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-insensitive.

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.

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 ADBase.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
# 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 ADBase.template (fi				
Parameter index variable	**asyn interface**	**Access**	**Description**	**drvInfo s**

Information about the asyn port				
NDPortNameSelf	asynOctet	r/o	asyn port name	PORT_NAME_SELF
Data type				
NDDataType	asynInt32	r/w	Data type (NDDataType_t).	DATA_TYPE
Color mode				
NDColorMode	asynInt32	r/w	Color mode (NDColorMode_t).	COLOR_MODE
Actual dimensions of array data				
NDArraySizeX	asynInt32	r/o	Size of the array data in the X direction	ARRAY_SIZE_X
NDArraySizeY	asynInt32	r/o	Size of the array data in the Y direction	ARRAY_SIZE_Y
NDArraySizeZ	asynInt32	r/o	Size of the array data in the Z direction	ARRAY_SIZE_Z
NDArraySize	asynInt32	r/o	Total size of the array data in bytes	ARRAY_SIZE
File saving parameters (records are defined in NDFile.template)				
NDFilePath	asynOctet	r/w	File path	FILE_PATH
NDFilePathExists	asynInt32	r/o	Flag indicating if file path exists	FILE_PATH_EXISTS
NDFileName	asynOctet	r/w	File name	FILE_NAME
NDFileNumber	asynInt32	r/w	File number	FILE_NUMBER

NDFileTemplate	asynOctet	r/w	Format string for constructing NDFullFileName from NDFilePath, NDFileName, and NDFileNumber. The final file name (which is placed in NDFullFileName) is created with the following code:
			
			`epicsSnprintf(`
			`  FullFilename,`
			`  sizeof(FullFilename),`
			`  FileTemplate, FilePath,`
			`  Filename, FileNumber);`
			
			FilePath, Filename, FileNumber are converted in that order with FileTemplate. An example file format is `"%s%s%4.4d.tif"`. The first %s converts the FilePath, followed immediately by another %s for Filename. FileNumber is formatted with %4.4d, which results in a fixed field with of 4 digits, with leading zeros as required. Finally, the .tif
				FILE_TEMPLATE

			extension is added to the file name. This mechanism for creating file names is very flexible. Other characters, such as _ can be put in Filename or FileTemplate as desired. If one does not want to have FileNumber in the file name at all, then just omit the %d format specifier from FileTemplate. If the client wishes to construct the complete file name itself, then it can just put that file name into NDFileTemplate with no format specifiers at all, in which case NDFilePath, NDFileName, and NDFileNumber will be ignored.	
NDFullFileName	asynOctet	r/o	Full file name constructed using the algorithm described in NDFileTemplate	FULL_FILE_NA
NDAutoIncrement	asynInt32	r/w	Auto-increment flag. Controls whether FileNumber is automatically incremented by 1 each time a file is saved (0=No, 1=Yes)	AUTO_INCRE
NDAutoSave	asynInt32	r/w	Auto-save flag (0=No, 1=Yes) controlling whether a file is automatically saved each time acquisition completes.	AUTO_SAVE
NDFileFormat	asynInt32	r/w	File format. The format to write/read data in (e.g. TIFF, netCDF, etc.)	FILE_FORMAT
NDWriteFile	asynInt32	r/w	Manually save the most recent array to a file when value=1	WRITE_FILE
NDReadFile	asynInt32	r/w	Manually read a file when value=1	READ_FILE
NDFileWriteMode	asynInt32	r/w	File saving mode (Single, Capture, Stream)(NDFileMode_t)	WRITE_MODE
NDFileCapture	asynInt32	r/w	Start (1) or stop (0) file capture or streaming	CAPTURE
NDFileNumCapture	asynInt32	r/w	Number of frames to acquire in capture or streaming mode	NUM_CAPTUR
NDFileNumCaptured	asynInt32	r/o	Number of arrays currently acquired capture or streaming mode	NUM_CAPTUR
NDFileDeleteDriverFile	asynInt32	r/w	Flag to enable deleting original driver file after a plugin has re-written the file in a different format. This can be useful for detectors that must write the data to disk in order for the areaDetector driver to read it back. Once a file-writing plugin has rewritten the data in another format it can be desireable to then delete the original file.	DELETE_DRIV
Array data				
NDArrayCallbacks	asynInt32	r/w	Controls whether the driver does callbacks with the array data to registered plugins.	ARRAY_CALL

			0=No, 1=Yes. Setting this to 0 can reduce overhead in the case that the driver is being used only to control the device, and not to make the data available to plugins or to EPICS clients.	
NDArrayData	asynGenericPointer	r/w	The array data as an NDArray object	NDARRAY_DATA
NDArrayCounter	asynInt32	r/w	Counter that increments by 1 each time an array is acquired. Can be reset by writing a value to it.	ARRAY_COUNTER
N/A	N/A	r/o	Rate at which arrays are being acquired. Computed in the ADBase.template database.	N/A
Array attributes				
NDAttributesFile	asynOctet	r/w	The name of an XML file defining the PVAttributes and paramAttributes to be added to each NDArray by this driver or plugin. The format of the XML file is described in the documentation for asynNDArrayDriver::readNDAttributesFile().	ND_ATTRIBUTES_FIL
Debugging control				
N/A	N/A	N/A	asyn record to control debugging (asynTrace)	N/A

ADDriver

ADDriver inherits from asynNDArrayDriver. This is the class from which area detector drivers are directly derived. It provides parameters and methods that are specific to area detectors, while asynNDArrayDriver is a general NDArray driver. The ADDriver class documentation describes this class in detail.

The file ADDriver.h defines the parameters that all areaDetector drivers should implement if possible.

Parameter Definitions in ADDriver.h and EPICS Record Definitions in ADBase.te					
Parameter index variable	**asyn interface**	**Access**	**Description**	**drvInfo string**	
Information about the detector					
ADManufacturer	asynOctet	r/o	Detector manufacturer name	MANUFACTURER	$(
ADModel	asynOctet	r/o	Detector model name	MODEL	$(
ADMaxSizeX	asynInt32	r/o	Maximum (sensor) size in the X direction	MAX_SIZE_X	$(
ADMaxSizeY	asynInt32	r/o	Maximum (sensor) size in the Y direction	MAX_SIZE_Y	$(
ADTemperature	asynFloat64	r/w	Detector temperature	TEMPERATURE	$($(
Detector readout control including gain, binning, region start and size, revers					

ADGain	asynFloat64	r/w	Detector gain	GAIN
ADBinX	asynInt32	r/w	Binning in the X direction	BIN_X
ADBinY	asynInt32	r/w	Binning in the Y direction	BIN_Y
ADMinX	asynInt32	r/w	First pixel to read in the X direction. 0 is the first pixel on the detector.	MIN_X
ADMinY	asynInt32	r/w	First pixel to read in the Y direction. 0 is the first pixel on the detector.	MIN_Y
ADSizeX	asynInt32	r/w	Size of the region to read in the X direction	SIZE_X
ADSizeY	asynInt32	r/w	Size of the region to read in the Y direction	SIZE_Y
ADReverseX	asynInt32	r/w	Reverse array in the X direction (0=No, 1=Yes)	REVERSE_X
ADReverseY	asynInt32	r/w	Reverse array in the Y direction (0=No, 1=Yes)	REVERSE_Y
Image and trigger modes				
ADImageMode	asynInt32	r/w	Image mode (ADImageMode_t).	IMAGE_MODE
ADTriggerMode	asynInt32	r/w	Trigger mode (ADTriggerMode_t).	TRIGGER_MODE
Frame type				
ADFrameType	asynInt32	r/w	Frame type (ADFrameType_t).	FRAME_TYPE
Acquisition time and period				
ADAcquireTime	asynFloat64	r/w	Acquisition time per image	ACQ_TIME
ADAcquirePeriod	asynFloat64	r/w	Acquisition period between images	ACQ_PERIOD
Number of exposures and number of images				
ADNumExposures	asynInt32	r/w	Number of exposures per image to acquire	NEXPOSURES
ADNumImages	asynInt32	r/w	Number of images to acquire in one acquisition sequence	NIMAGES
Acquisition control				
ADAcquire	asynInt32	r/w	Start (1) or stop (0) image acquisition. This is an EPICS busy record that does not process its forward link until acquisition is	ACQUIRE

			complete. Clients should write 1 to the Acquire record to start acquisition, and wait for Acquire to go to 0 to know that acquisition is complete.		
Status information					
ADStatus	asynInt32	r/o	Acquisition status (ADStatus_t)	STATUS	$(
ADStatusMessage	asynOctet	r/o	Status message string	STATUS_MESSAGE	$(
ADStringToServer	asynOctet	r/o	String from driver to string-based vendor server	STRING_TO_SERVER	$(
ADStringFromServer	asynOctet	r/o	String from string-based vendor server to driver	STRING_FROM_SERVER	$(
ADNumExposuresCounter	asynInt32	r/o	Counter that increments by 1 each time an exposure is acquired for the current image. Driver resets to 0 when acquisition is started.	NUM_EXPOSURES_COUNTER	$(
ADNumImagesCounter	asynInt32	r/o	Counter that increments by 1 each time an image is acquired in the current acquisition sequence. Driver resets to 0 when acquisition is started. Drivers can use this as the loop counter when ADImageMode=ADImageMultiple.	NUM_IMAGES_COUNTER	$(
ADTimeRemaining	asynFloat64	r/o	Time remaining for current image. Drivers should update this value if they are doing the exposure timing internally, rather than in the detector hardware.	TIME_REMAINING	$(
ADReadStatus	asynInt32	r/w	Write a 1 to this parameter to force a read of the detector status. Detector drivers normally read the status as required, so this is usually not necessary, but there may be some circumstances under which forcing a status read may be needed.	READ_STATUS	$(
Shutter control					
ADShutterMode	asynInt32	r/w	Shutter mode (None, detector-controlled or EPICS-controlled) (ADShutterMode_t)	SHUTTER_MODE	$($(
ADShutterControl	asynInt32	r/w	Shutter control for the selected (detector or EPICS) shutter (ADShutterStatus_t)	SHUTTER_CONTROL	$($(
ADShutterControlEPICS	asynInt32	r/w	This record processes when it receives a callback from the driver	SHUTTER_CONTROL_EPICS	$(

			to open or close the EPICS shutter. It triggers the records below to actually open or close the EPICS shutter.	
N/A	N/A	r/w	This record writes its OVAL field to its OUT field when the EPICS shutter is told to open. The OCAL (and hence OVAL) and OUT fields are user-configurable, so any EPICS-controllable shutter can be used.	N/A
N/A	N/A	r/w	This record writes its OVAL field to its OUT field when the EPICS shutter is told to close. The OCAL (and hence OVAL) and OUT fields are user-configurable, so any EPICS-controllable shutter can be used.	N/A
ADShutterStatus	asynInt32	r/o	Status of the detector-controlled shutter (ADShutterStatus_t)	SHUTTER_STATUS
N/A	N/A	r/o	Status of the EPICS-controlled shutter. This record should have its input link (INP) set to a record that contains the open/close status information for the shutter. The link should have the "CP" attribute, so this record processes when the input changes. The ZRVL field should be set to the value of the input link when the shutter is closed, and the ONVL field should be set to the value of the input link when the shutter is open.	N/A
ADShutterOpenDelay	asynFloat64	r/w	Time required for the shutter to actually open (ADShutterStatus_t)	SHUTTER_OPEN_DELAY
ADShutterCloseDelay	asynFloat64	r/w	Time required for the shutter to actually close (ADShutterStatus_t)	SHUTTER_CLOSE_DELAY

Guidelines and rules for drivers

The following are guidelines and rules for writing areaDetector drivers

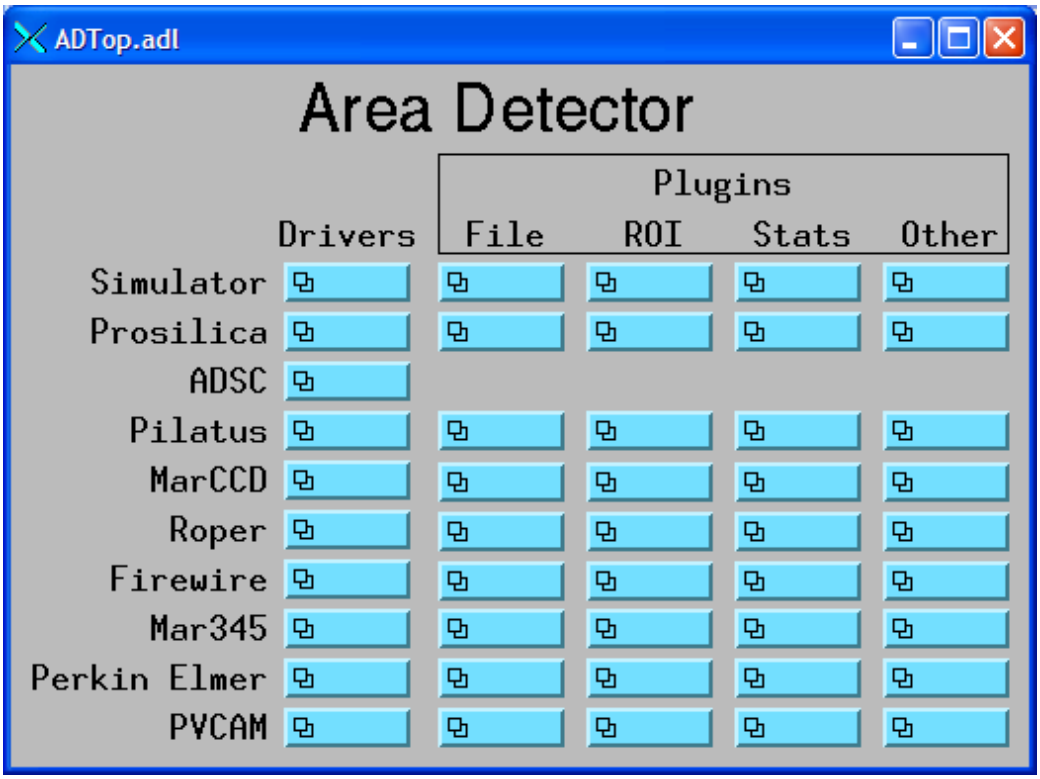
- Drivers will generally implement one or more of the writeInt32(), writeFloat64() or writeOctet() functions if they need to act immediately on a new value of a parameter. For many parameters it is normally sufficient to simply have them written to the parameter library, and not to handle them in the writeXXX() functions. The parameters are then retrieved from the parameter library with the getIntParam(), getDoubleParam(), or getStringParam() function calls when they are needed.

- If the writeInt32(), writeFloat64() or writeOctet() functions are implemented they **must** call the base class function for parameters that they do not handle and whose parameter index value is less than the first parameter of this class, i.e. parameters that belong to a base class.
- Drivers will need to call the createParam() function in their constructor if they have additional parameters beyond those in the asynPortDriver or ADDriver base classes.
- Drivers will generally need to create a new thread in which they run the acquisition task. Some vendor libraries create such a thread themselves, and then the driver must just implement a callback function that runs in that thread (the Prosilica is an example of such a driver).
- The acquisition thread will typically monitor the acquisition process and perform periodic status update callbacks. The details of how to implement this will vary depending on the specifics of the vendor API. There are many existing detector drivers that can be used as examples of how to write a new driver.
- If the detector hardware does not support fixed-period acquisition or muliple-image acquisition sequence (ADNumImages parameter) then these should be emulated in the driver. The simDetector, marCCD and other drivers can be used as examples of how to do this.
- If NDArrayCallbacks is non-zero then drivers should do the following:
    - Call asynNDArrayDriver::getAttributes to attach any attributes defined for this driver to the current array.
    - Call doCallbacksGenericPointer() so that registered clients can get the values of the new arrays. Drivers must release their mutex by calling this->unlock() before they call doCallbacksGenericPointer(), or a deadlock can occur if the plugin makes a call to one of the driver functions.

MEDM screens

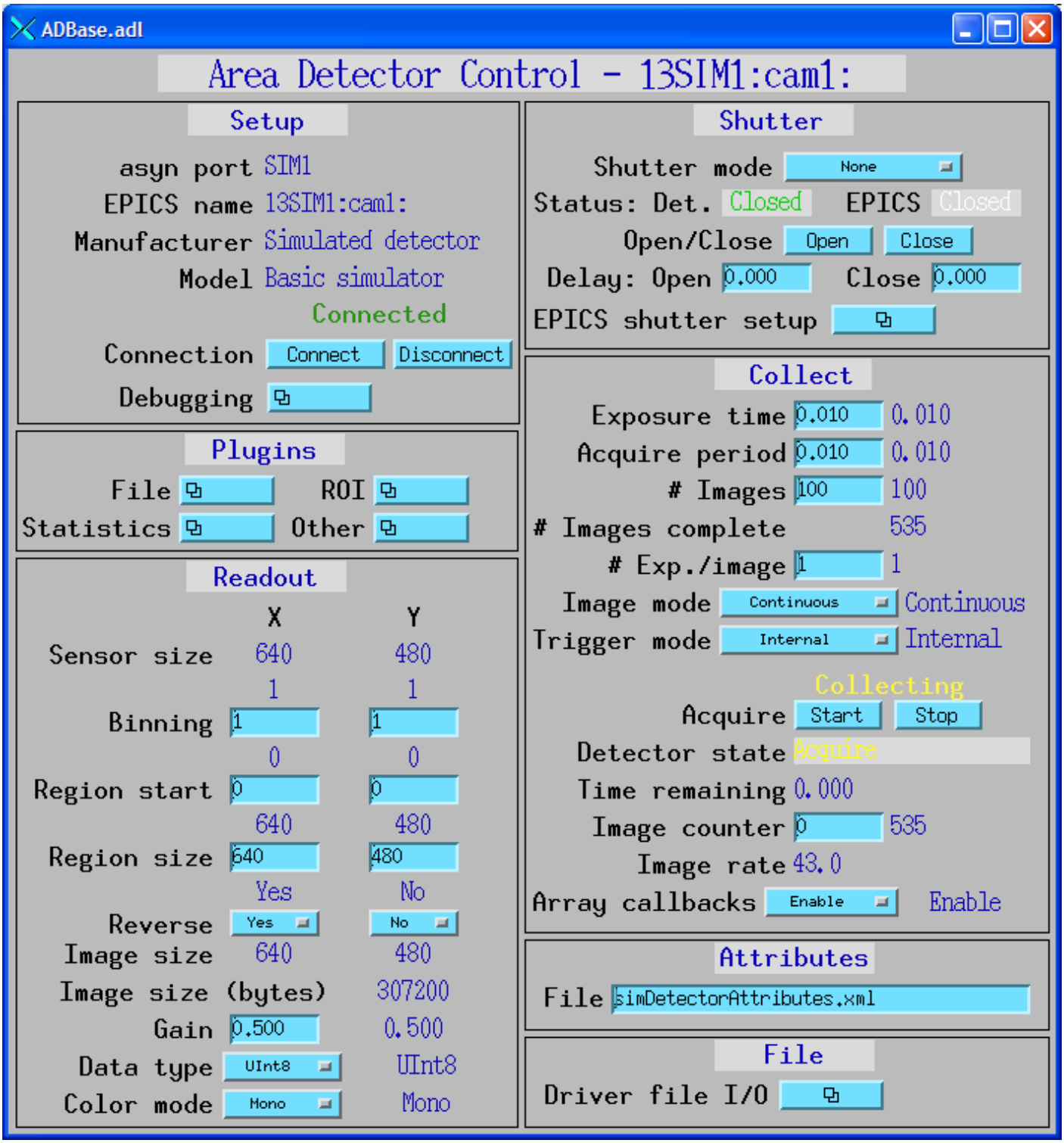
The following is the top-level MEDM screen that provides links to the screens for most of the detectors and plugins that areaDetector supports. This screen is useful for testing, and as a source for copying related-display menus to be placed in application-specific MEDM screens.

ADTop.adl



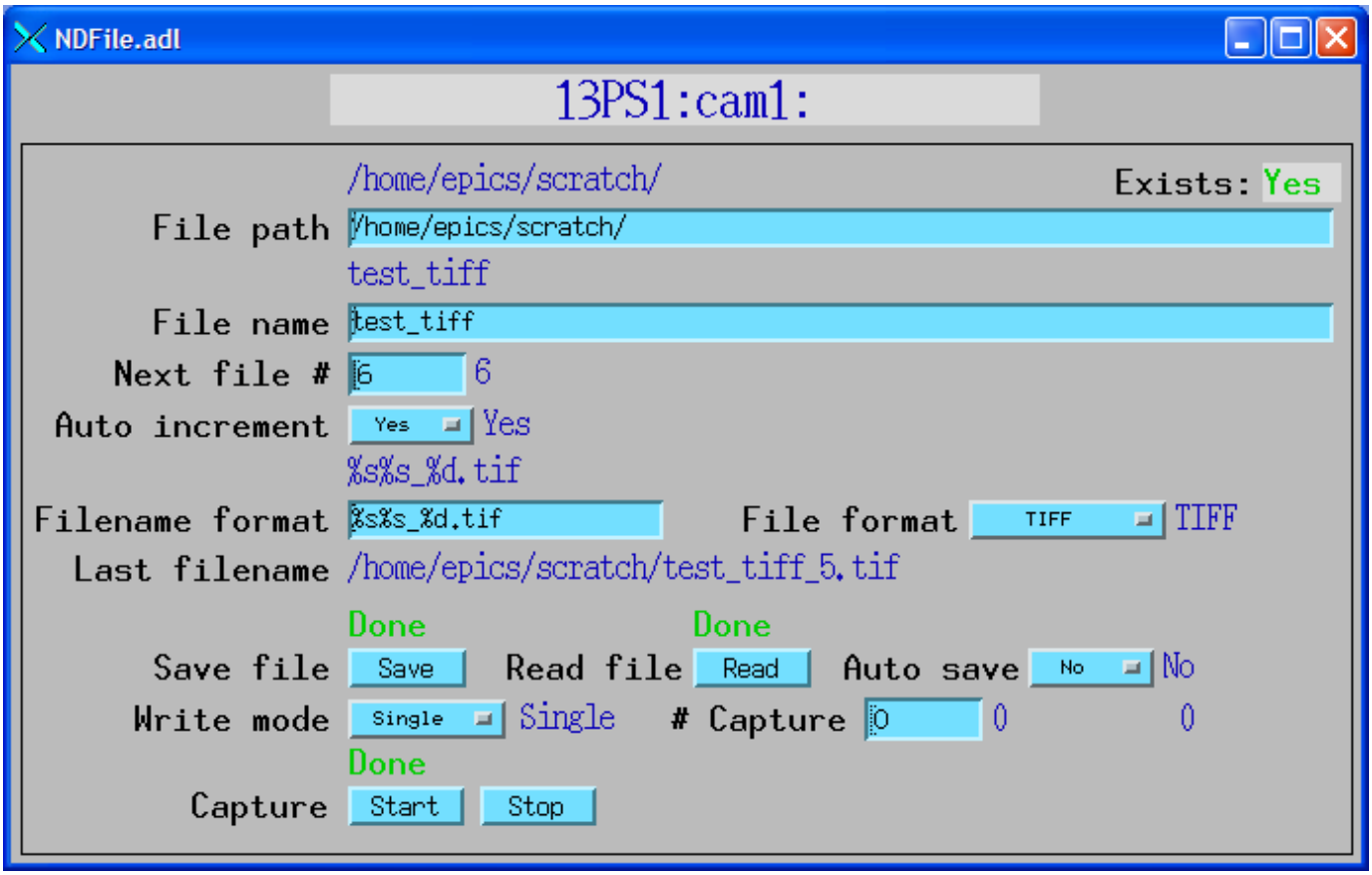
The following is the MEDM screen that provides access to the parameters in asynNDArrayDriver.h and ADDriver.h through records in ADBase.template. This is a top-level MEDM screen that will work with any areaDetector driver. Note however that many drivers will not implement all of these parameters, and there will usually be detector-specific parameters not shown in this screen, so detector-specific MEDM screens should generally be created that display the EPICS PVs for the features implemented for that detector.

ADBase.adl



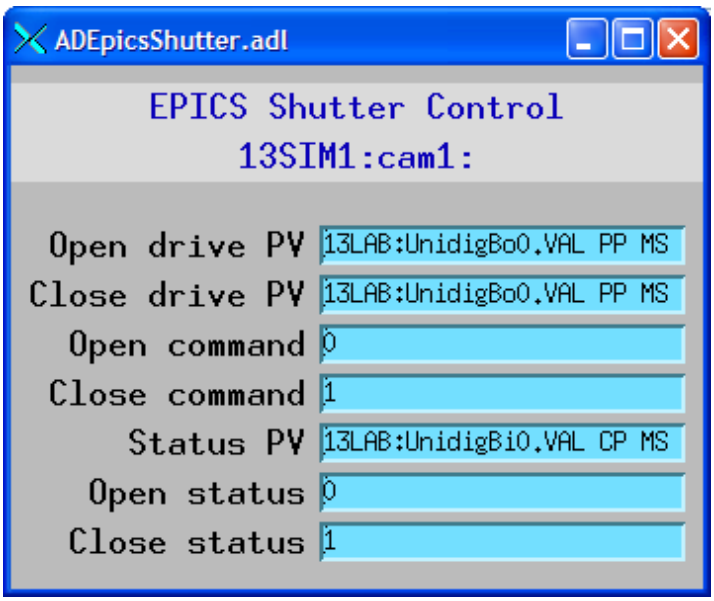
The following is the MEDM screen that provides access to the file-related parameters in asynNDArrayDriver.h through records in NDFile.template. This screen is for use with detector drivers that directly implement file I/O.

NDFile.adl



The following is the MEDM screen that provides access to the EPICS shutter parameters in ADDriver.h through records in ADBase.template. This screen allows one to define the EPICS PVs to open the shutter, close the shutter, and determine the shutter status. The values of these PVs for open and close drive and status can also be defined. Note that in many cases the same PV will be used for open and close drive, but in some cases (e.g. APS safety shutters) different PVs are used for open and close.

ADEpicsShutter.adl



Installation, configuration, and running

Installation: source code version

After obtaining a copy of the distribution, it must be installed and built for use at your site. These steps only need to be performed once for the site (unless versions of the module running under different releases of EPICS and/or the other required modules are needed).

1. Create an installation directory for the module, usually this will end with

   ```
   .../support/
   ```
2. Place the distribution file in this directory. Then issue the commands (Unix style)

   ```
   tar xvzf areaDetectorRX-Y.tgz
   ```

   where X-Y is the release.
3. This creates a <top> application.

   ```
   .../support/areaDetectorRX-Y
   ```
4. Download all required supporting modules if you don't already have them. This includes asyn, autosave, busy, etc.
5. Edit the `configure/RELEASE` file and set the paths to your installation of EPICS base and to your versions of supporting modules.
6. Run `gnumake` in the top level directory and check for any compilation errors.

Installation: prebuilt version

Prebuilt versions of areaDetector are provide for Windows (win32-x86), Cygwin (cygwin-x86), and Linux (linux-x86). Follow these steps to use the prebuilt version:

1. Create an installation directory for the module. On Windows I typically use C:\EPICS\support. On Linux I typically use /home/ACCOUNT/epics/support, where ACCOUNT is the name of the account that is normally used to run the detector software, e.g. marccd on a marCCD detector, mar345 on a mar345 detector, etc.
2. Place the distribution file in this directory. Then issue the commands (Unix style)

   ```
   tar xvzf areaDetectorPrebuilt_RX-Y.tgz
   ```
3. In the iocBoot directory make a **copy** of the example ioxXXX directory for the detector you are using and give it a new local name. By doing this you will be able to update to later versions of areaDetector without overwriting modifications you make in the iocXXX directory.
4. In the new iocXXX directory you just created edit st.cmd to change the PV prefix $(P) to one that is unique to your site. PV prefixes must be unique on the subnet, and if you use the default prefix there could be a conflict with other detectors of the same type.
5. In the same iocXXX directory edit the file envPaths to point to the locations of all of the support modules on your system. Normally this is handled by the EPICS build system, but when using the prebuilt version this must be manually edited. Do not worry about the path to EPICS_BASE, it is not required.

Installation: medm

areaDetector provides display/control screens for the medm Motif Editor and Display Manager. A prebuilt version of medm for Windows can be found in the EPICS WIN32 Extensions. For Linux one can build medm from source code, which requires downloading and building EPICS base first. Alternatively I provide a prebuilt version of medm for Linux in the EPICS_Linux_binraries.tar package. To use this version copy the medm executable to some location in your PATH, e.g. /usr/local/bin, or ~/bin, etc. Copy the 2 shareable libraries libCom.so and libca.so to a location which is in your LD_LIBRARY_PATH. To use either the source code or prebuilt version you need to have the OpenMotif package installed. This typically is **not** installed by default with recent versions of Linux. Go to www.openmotif.org and download and install the appropriate RPM package for your Linux version.

Configuration

Before running an areaDetector application it is usually necessary to configure a number of items.

- EPICS environment variables. There are several environment variables that EPICS uses. I suggest setting these in the .cshrc (or .bashrc) file for the account that will be used to run the detector.
  - EPICS_CA_AUTO_ADDR_LIST and EPICS_CA_ADDR_LIST. These variables control the IP addresses that EPICS clients use when searching for EPICS PVs. The default is EPICS_CA_AUTO_ADDR_LIST=YES and EPICS_CA_ADDR_LIST to be the broadcast address of all networks connected to the host. Some detectors, for example the marCCD and mar345, come with 2 network cards, which are on 2 different subnets, typically a private one connected to the detector and a public one connected to the local LAN. If the default value of these variables is used then EPICS clients (e.g. medm) running on the detector host computer will generate many errors because each EPICS PV will appear to be coming from both networks. The solution is to set these variables as follows:

```
setenv EPICS_CA_AUTO_ADDR_LIST NO
setenv EPICS_CA_ADDR_LIST localhost:XX.YY.ZZ.255
```

    where XX.YY.ZZ.255 should be replaced with the broadcast address for the public network on this computer.
  - EPICS_CA_MAX_ARRAY_BYTES. This variable controls the maximum array size that EPICS can transmit with Channel Access. The default is only 16kB, which is much too small for most detector data. This value must be set to a large enough value on both the EPICS server computer (e.g. the one running the areaDetector IOC) and client computer (e.g. the one running medm, ImageJ, IDL, etc.). This can simply be set to a very large value like 100MB.

```
setenv EPICS_CA_MAX_ARRAY_BYTES 100000000
```

  - EPICS_DISPLAY_PATH. This variable controls where medm looks for .adl display files. If the recommendation below is followed to copy all adl files to a single directory, then this environment variable should be defined to point to that directory. For example:

```
setenv EPICS_DISPLAY_PATH /home/mar345/epics/adls
```

- medm display files. I find it convenient to copy all medm .adl files to a single directory and then point the environment variable EPICS_DISPLAY_PATH to this directory. The alternative is to point EPICS_DISPLAY_PATH to a long list of directories where the adl files are located in the distributions,

which is harder to maintain. On the mar345, for example, I create a directory called /home/mar345/epics/adls, where I put all of the adl files. To simplify copying the adl files to that location I use the following one-line script, which I place in /home/mar345/bin/sync_adls.

```
find /home/mar345/epics/support -name '*.adl' -exec cp -fv {} /home/mar345/epics/adls \;
```

This script finds all adl files in the epics/support tree and copies them to /home/mar345/epics/adls. That directory must be created before running this script. Similar scripts can be used for other Linux detectors (marCCD, Pilatus, etc.) and can be used on Windows as well if Cygwin is installed. Each time a new release of areaDetector is installed remove the old versions of each support module (areaDetector, asyn, autosave, etc.) and then run this script to install the latest medm files.

Running the IOC

Each example IOC directory comes with a Linux script (start_epics) or a Windows batch file (start_epics.bat) or both depending on the architectures that the detector runs on. These scripts provide simple examples of how to start medm and the EPICS IOC. For example, for the mar345 iocBoot/iocMAR345/start_epics contains the following:

```
medm -x -macro "P=13MAR345_1:, R=cam1:, I=image1:, ROI=ROI1:, NETCDF=netCDF1:, TIFF=TIFF1:, JPEG=JPEG1:
../../bin/linux-x86/mar345App st.cmd
```

This script starts medm in execute mode with the appropriate medm display file and macro parameters, running it in the background. It then runs the IOC application. This script assumes that iocBoot/iocMAR345 is the default directory when it is run, which could be added to the command or set in the configuration if this script is set as the target of a desktop shortcut, etc. The script assumes that EPICS_DISPLAY_PATH has been defined to be a location where the mar345.adl and related displays that it loads can be found. You will need to edit the script in your copy of the iocXXX directory to change the prefix (P) from 13MAR345_1: to whatever prefix you chose for your IOC. The start_epics script could also be copied to a location in your PATH (e.g. /home/mar345/bin/start_epics). Add a command like `cd /home/mar345/epics/support/areaDetector/1-5/iocBoot/iocMAR345` at the beginning of the script and then type `start_epics` from any directory to start the EPICS IOC.

Acknowledgements and licenses

"This software is based in part on the work of the Independent JPEG Group".