areaDetector driver lab handout

Mark Rivers, University of Chicago

January 29, 2015
Advanced Photon Source
Room E1100/1200

Overview

This lab is intended to introduce writing an areaDetector driver using the asynDriver C++ base class. The device to be controlled is simplified version of the simulation simDetector. The new simDetector plots a noisy sin wave.

Setup

The development can either be done locally on a student's laptop or removely on our Linux machine called corvette.cars.aps.anl.gov.

For working on corvette each student has his/her own subdirectory where they will be working, /home/epics_class/student1, /home/epics_class/student2, etc. The assignment of student numbers will be done during the class.

You should bring a laptop to the lab with the following capabilities:

- X11 server
- ssh client configured to tunnel X11

The ssh/X11 connection should be tested prior to the class by logging into corvette using the credentials above and typing the command "xclock &" or "medm &". If you see the application then things are configured correctly.

Linux host: corvette.cars.aps.anl.gov
Username: (given out in class)
Password: (given out in class)

Copy and build the areaDetector/ADCore source code

Once you are logged in go to the appropriate subdirectory, e.g. student1.

```
$ cd student1
```

Copy the areaDetector source code for the lab from the /home/epics_class/teacher directory to this directory:

```
$ cp -rp ../teacher/areaDetector .
```

Change to the areaDetector/configure directory:

```
$ cd areaDetector/configure
```

Edit RELEASE_PATHS.local and change the location of areaDetector, changing this line:

```
AREA_DETECTOR= /corvette/home/epics_class/teacher/areaDetector
```

to

```
AREA_DETECTOR= /corvette/home/epics_class/student1/areaDetector
```

Replacing student1 with your student number.

Change to the areaDetector/ADCore directory.

```
cd ../ADCore
```

Clean the source:

```
$ make –sj clean uninstall
```

Build the source:

```
$ make –sj
```

The –s flag means that make runs silently, so you only see errors and warnings. –j means that the make is run in parallel, doing as many tasks as possible at the same time.

Customize the setup for your student number

All of the soft IOCs are running on the same machine and subnet, so each soft IOC must use a different PV prefix. It is very important that you change your setup to use the appropriate PV prefix. This is done as follows:

Change to the ADCore/iocs/simDetectorIOC/iocBoot/iocSimDetector directory:

```
$ cd iocs/simDetectorIOC/iocBoot/iocSimDetector/
```

Edit the file st.cmd and change PREFIX from SIM_1: to SIM_N:, where N is your student number, i.e. SIM_2: for student2, etc.

simDetector2

You will be creating a simple version of the simDetector that computes an image with a plot of a noisy sin wave. For simplicity only the following parameters from ADDriver.h will be implemented

ADSizeX : the number of X pixels
ADSizeY : the number of Y pixels
ADGain : the intensity value of the plot

In addition your driver will implement 3 new parameters for the simDetector2 class:

SimGainX : the gain in the X direction = number of sin wave periods across the image
SimGainY : the gain (amplitude) in the Y direction. Gain=1 will exactly fill the Y range
SimNoise : noise in the Y direction. Expressed as a fraction of SimGainY.

EPICS database
The database files for the driver are in `ADCore/ADApp/Db/simDetector2.template. It contains only the records for the 3 simDetector2 parameters above.`

Driver source code
The source code for the driver is in ADCore/ADApp/simDetector/src. There are two versions of the driver in that location: simDetector2.cpp and simDetector2Template.cpp. simDetector2.cpp is a fully functional driver, and so is an example to use. simDetector2.cpp Template.cpp is a skeleton driver. It contains just enough code to allow the IOC to run and for the EPICS records to connect to the driver, but it does nothing useful. Edit the Makefile in this directory to control which version of the driver you compile.

```
# Use this line for the original version of simDetector
#LIB_SRCS += simDetector.cpp
# Use this line for the full version of simDetector2
LIB_SRCS += simDetector2.cpp
# Use this line for the template version of simDetector2
#LIB_SRCS += simDetector2Template.cpp
```

Each student may want to approach the lab differently, depending on experience and goals. One approach would be to first start with the fully developed driver, and run the MEDM display to understand how the driver and device work.

Start the MEDM display

```
$ medm -x –macro “P=SIM_1:, R=cam1:” simDetector.adl &
```

The fields will all be white until you start your IOC.

Start the IOC

```
$ cd iocs/simDetectorIOC/iocBoot/iocSimDetector/
$ ../../bin/linux-x86_64/simDetectorApp st.cmd
```

Use asynReport to print information about the drivers

Type asynReport at the IOC prompt to see a brief report of each asyn driver in the IOC.

```
epics> asynReport
```

Get a detailed report on SIM1

```
epics> asynReport 10 SIM1
…
```

Use asynTrace to print information about messages between the XPSDriver and the XPSServer over the XPSSocket driver.

You can enable asynTrace messages at the IOC prompt:

```
epics> asynSetTraceIOMask SIM1 0 2
epics> asynSetTraceMask SIM1 0 9
```

Altenatively and more conveniently you can open the asynRecord.adl MEDM display for this port. You can then press traceIOEscape and traceIODriver to do the same thing as the above commands.

You should study the database and the functional driver code to understand how it works.

Building a real driver from the skeleton driver

Once you have worked with the functional driver, switch to building and running the skeleton driver. Initially it will not do anything. Try adding the following features one at a time. You can try to write them yourself or copy the code from the functional driver. Make sure you understand each step, and ask questions if you don't!

1. Add the code to the constructor to create the epicEvents, create parameters, set defaults, create the simulation task

2. Implement the code in writeInt32 to handle ADAcquire, ADSizeX, and ADSizeY..

3. Implement the simTask to start acquiring, update the image, set the NDArray metadata, and do the callbacks.

4. Implement the computeImage task to actually compute a new image.