asyn lab handout

Mark Rivers, University of Chicago
January 27, 2015
Advanced Photon Source
Room E1100/1200

Overview

This lab is intended to introduce writing an asyn port driver using the asynPortDriver C++ base class. The device to be controlled is a Newport XPS motor controller. In this lab we will be controlling the analog inputs and outputs, and the digital inputs and outputs. We will not be controlling the motors. The online documentation for the Newport XPS can be found here:
http://assets.newport.com/webDocuments-EN/images/XPS-Q8_XPSDocumentation.pdf

The information about the analog and digital I/O functions are documented on pages 206-211 of that manual.

The XPS is controlled via ASCII commands over TCP/IP sockets. Newport provides a C library that hides these commands from the programmer. For this exercise however, we will not be using their library but will communicating directly with the controller over a TCP port.

The XPS listens for TCP socket connections on port 5001. The XPS allows multiple socket clients to connect on this port, so each student can be connecting to the actual hardware. The only limitation is that if one student writes to an output, all other students will see that change.

Setup

The development will be done on our Linux machine. 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 XPSExample source code

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

```
$ cd student1
```

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

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

Change to the XPSExample directory:

```
$ cd XPSExample/
```

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 iocBoot/iocXPSExample directory:

```
$ cd iocBoot/iocXPSExample/
```

Edit the files st.cmd and XPSExample.substitutions and replace the string XPS_TEACHER: with XPS_STUDENT_N:, where N the appropriate number for your student number, i.e. XPS_STUDENT_1: for student1, etc.

You must also change your medm setup so that it uses your PV prefix.

Change to the XPSExampleApp/op/adl directory:

```
$ cd ../../XPSExampleApp/op/adl
```

Edit XPSExampleTop.adl either with medm or a text editor. Replace all occurrences of XPS_TEACHER: with XPS_STUDENT_1:, etc.

XPS Commands

The XPS listens for TCP socket connections on port 5001. The XPS allows multiple socket clients to connect on this port, so each student can be connecting to the actual hardware. The only limitation is that if one student writes to an output, all other students will see that change.

The following are the commands that we will be using for this exercise:

GPIOAnalogGet(device, double *)
There are 4 analog inputs, and their device names are:
GPIO2.ADC1
GPIO2.ADC2
GPIO2.ADC3
GPIO2.ADC4
The "double *" argument to the command is a string literal: that string must be sent to the XPS. It is a strange syntax, I know.

If the command is successful the response will be similar to the following:
0,1.494854661444,EndOfAPI
So the value read back comes after the first comma in the response.

GPIOAnalogSet(device, value)
There are 4 analog outputs, and their device names are:
GPIO2.DAC1
GPIO2.DAC2
GPIO2.DAC3
GPIO2.DAC4

Value is a double value in volts. For example
GPIBAnalogSet(GPIO.DAC1, 1.5)
If the command is successful the response will be the following:
0,,EndOfAPI

GPIODigitalGet(device, unsigned short *)
We will be using the 6 digital input bits on device GPIO3.DI.

GPIODigitalSet(device, mask, value)
We will be using the 6 digital output bits on device GPIO3.DO.
If the command is successful the response will be the following:
0,,EndOfAPI

FirmwareVersionGet(char *)
This reads the firmware version of the XPS

For this exercise I have wired analog output 1 to analog input 1, 2 to 2, etc. This means that the analog input value read should be very close to the last value written to the corresponding analog output. I have also wired the digital output bits to the corresponding digital input bit, so you can test whether you are reading back the correct value.

The XPS does not terminate each reply with a normal terminator like \r or \n, or \r\n. Rather it terminates the reply with the string “EndOfAPI”. This terminator is too long to use the asyn EOS features. Fortunately the XPS does not require terminators on input commands, and always sends its replies in a single TCP message. So for this exercise we will just disable the EOS processing of the drvAsynIPPort driver completely, by setting the flag noProcessEos to 1.

XPSExample.cpp

The goal of this lab is to understand how to write a driver, XPSDriver.cpp to communicate with the XPS to read and write analog and digital values. The XPSDriver will use the asynPortDriver C++ base class.

EPICS database
The database files for the driver are in `XPSExample/XPSExampleApp/Db/`

```
XPSAi.db  analog input
XPSAo.db  analog output
XPSBi.db  binary input
XPSBo.db  binary output
XPSLi.db  long   input
XPSLo.db  long   output
XPSFirmwareVersion stringin input
```

These are fully functional databases. They should be studied to see how different types of records are handled: bo, bi, ao, ai, longout, longin, etc.

Driver source code
The source code for the driver is in XPSExample/XPSExampleApp/src. There are two versions of the driver in that location: XPSDriver.cpp and XPSDriverTemplate.cpp. XPSDriver.cpp is a fully functional driver, and so is an example to use. XPSDriverTemplate.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 template driver
#XPSExample_SRCS += XPSDriverTemplate.cpp
# Use this line for the fully developed driver
XPSExample_SRCS += XPSDriver.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

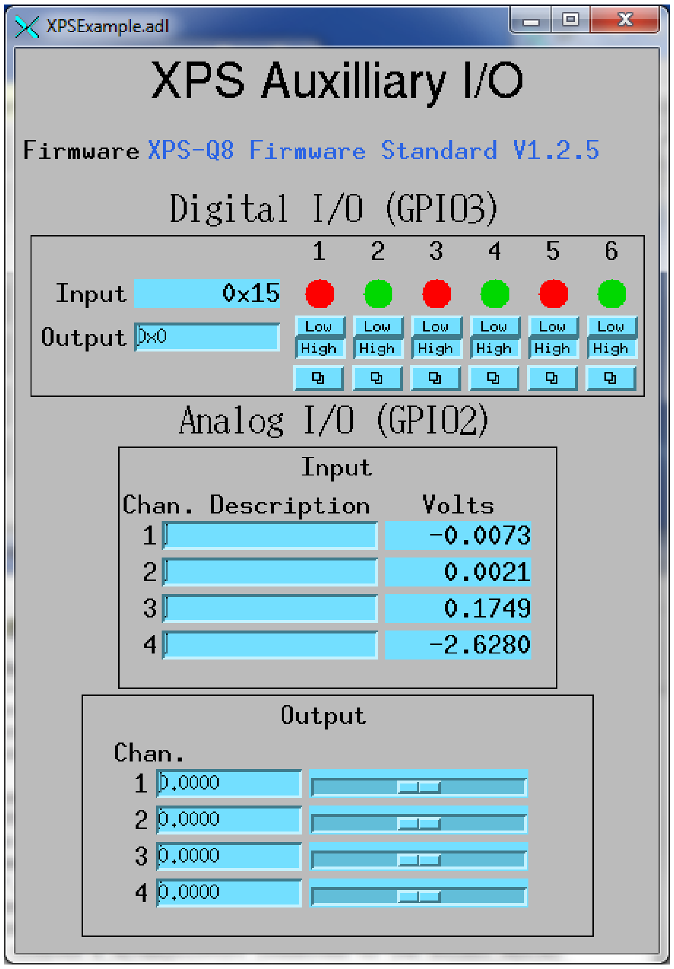
```
$ cd XPSExample/XPSExampleApp/op/adl

$ medm -x XPSExampleTop.adl &
```

This will start the top-level MEDM display. Click on the XPSAux button to open the XPSExample.adl related display. The fields will all be white until you start your IOC.

Start the IOC

```
$ cd XPSExample/iocBoot/iocXPSExample/
$ ../../bin/linux-x86_64/XPSExample st.cmd
```

Your MEDM screen should now be connected and look like the following:



XPS Auxilliary I/O

Firmware XPS-Q8 Firmware Standard V1.2.5

Digital I/O (GPIO3)

	1	2	3	4	5	6
Input 0x15	🔴	🟢	🔴	🟢	🔴	🟢
Output 0x0						

Analog I/O (GPIO2)

Input

Chan.	Description	Volts
1		-0.0073
2		0.0021
3		0.1749
4		-2.6280

Output

Chan.	
1	0.0000
2	0.0000
3	0.0000
4	0.0000

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
```

This listing shows that there are 2 asyn ports: XPSSocket, and XPSDriver.

Get a detailed report on XPSDriver:

```
epics> asynReport 10 XPSDriver
…
```

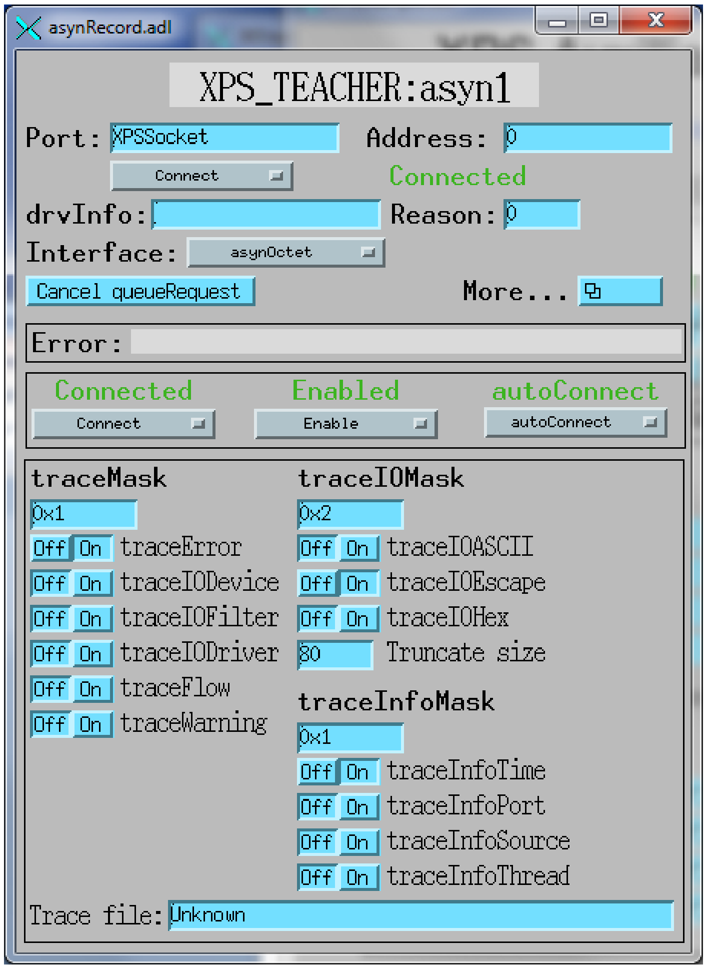
Note that it shows that there are 4 addresses for this driver, and that it shows the contents of the parameter library for each address.

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

You can enable asynTrace messages on the XPSSocket driver at the IOC prompt:

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

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


asynRecord.adl

XPS_TEACHER:asyn1

Port: XPSSocket Address: 0

Connected

drvInfo: Reason: 0

Interface: asynOctet

Error:

traceMask	traceIOMask
0x1	0x2
traceError	traceIOASCII
traceIODevice	traceIOEscape
traceIOFilter	traceIOHex
traceIODriver	80 Truncate size
traceFlow	traceInfoMask
traceWarning	0x1
	traceInfoTime
	traceInfoPort
	traceInfoSource
	traceInfoThread

Trace file: Unknown

This shows the communication over the XPSSocket port.

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. Connect to the server in the constructor

2. Read the firmware version in the constructor, so the callbacks so the records get the initial values.

3. Implement writeReadXPS to be able to send commands to the XPS and read the response

4. Implement writeUInt32Digital() for the bo and longout records

5. Implement writeFloat64 for the ao records

6. Implement pollerThread to read the digital and analog inputs

7. Implement the report() function to report the status of the driver

Additional things to try:

1) Note that the values of the output records do not correctly reflect the status of the hardware when the IOC starts. Why not? Can you modify the driver to make this work?
2) Note that the state of the longout record does not correctly change when the value of the bo records change, and the bo records do not change when the longout changes. This application is actually built with the latest SVN HEAD version of asyn, which will soon be released as R4-26. This has the capability of having output records change state when the driver does a callback, if the record has the info tag info(asyn:Readback, “1”). Try this and see if it works.
3) Note that when you exit the IOC it sends many error messages. Why? How can this be fixed? Hint: use epicsAtExit().