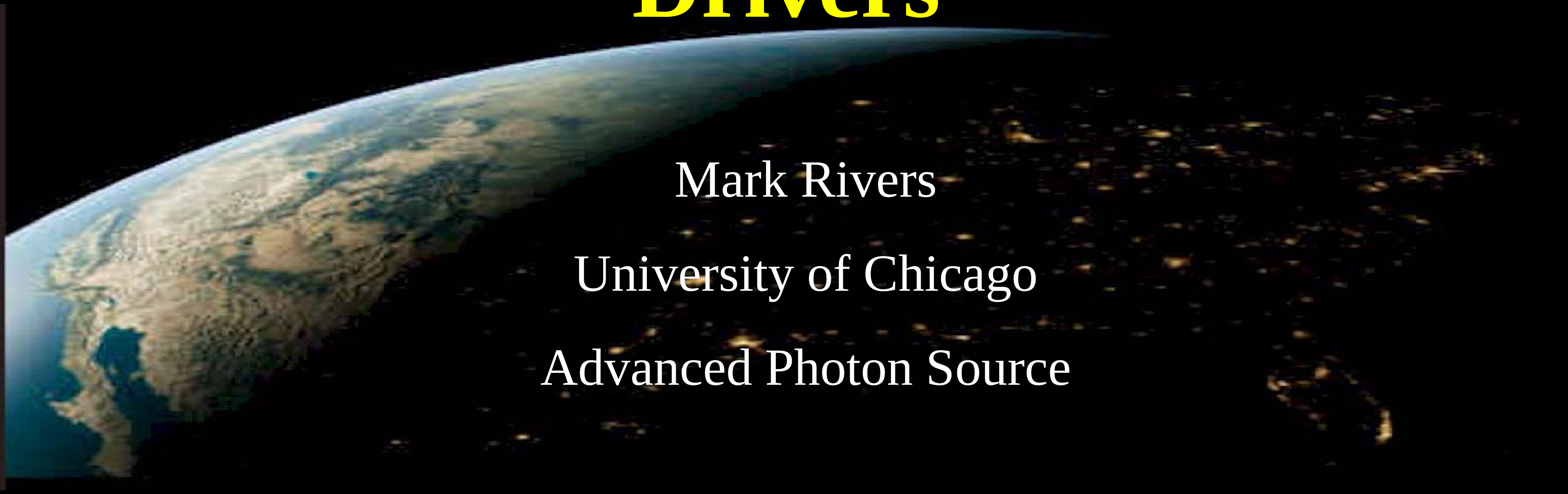
asynPortDriver

C++ Base Class for asyn Port Drivers



Mark Rivers

University of Chicago

Advanced Photon Source

How to deal with a new device (My philosophy!)

- If the device uses strings to communicate, and is not too complex, use streamDevice
  - Works well for relatively simple devices
  - Difficult to deal with more complex devices where parameters interact, since there is no “state” information
  - Uses asyn at the low-level (serial, TCP, GPIB)
- If the device does not use strings, or is complex, then write an asynPortDriver
- Should not need to write device support
  - Device support is difficult, since you need to understand the record
  - Writing device support is constraining to the developer, because you have decided what records to support
  - If you write an asyn driver the developer can chose the record types, or indeed not to use records at all, maybe call directly from SNL, etc.

asyn

- Well defined interface between EPICS device support and driver
- Standard asyn device support that can be used in nearly all cases
- In last 8 years I have written *many* new drivers and I have written almost no device support, just use standard asyn device support
- I believe asyn should be used to write *all* EPICS device drivers, not just “asynchronous” drivers like serial, GPIB and TCP/IP.
  - All of my drivers use asyn

asynPortDriver

- New C++ base class that greatly simplifies writing an asyn port driver
  - Initially developed as part of the areaDetector module
  - Moved from areaDetector into asyn itself in asyn 4-11
  - All of my areaDetector, D/A, binary I/O, and most recently motor drivers now use asynPortDriver
  - The drivers in the demos for this class (Measurement Computing 1608GX-2A0) use asynPortDriver
- Hides all details of registering interfaces, registering interrupt sources, doing callbacks, default connection management
- Why C++ ? Things that are hard in C:
  - Inheritance: virtual base class functions that can be overridden or enhanced by derived classed
  - Template functions: single function can handle any data type. Used extensively in areaDetector which supports 8 data types for NDArrays

asynPortDriver C++ Base Class

- Parameter library
  - Drivers typically need to support a number of parameters that control their operation and provide status information. Most of these can be treated as int32, int32Digital, float64, or strings. Sequence for new value:
    - New parameter value arrives from output record, or new data arrives from device
    - Change values of one or more parameters in object
    - For each parameter whose value changes set a flag noting that it changed
    - When operation is complete, call the registered callbacks for each changed parameter

asynPortDriver C++ Base Class

- asynPortDriver provides methods to simplify the above sequence
  - Each parameter is assigned an index based on the string passed to the driver in the drvUser interface
  - asynPortDriver has table of parameter values, with associated data type & asyn interface (int32, float32, etc.), caches the current value, maintains changed flag
  - There is a separate table for each asyn “address” that the driver supports
  - Drivers use asynPortDriver methods to read the current value from the table, and to set new values in the table.
  - Methods to call all registered callbacks for all values that have changed since callbacks were last done.

asynPortDriver Constructor

```
asynPortDriver(const char *portName, int maxAddr,
    int paramTableSize, int interfaceMask,
    int interruptMask, int asynFlags, int autoConnect,
    int priority, int stackSize);
```

portName:	Name of this asynPort
maxAddr:	Number of sub-addresses this driver supports
paramTableSize:	Number of parameters this driver supports
interfaceMask:	Bit mask of standard asyn interfaces the driver supports
interruptMask:	Bit mask of interfaces that will do callbacks to device support
asynFlags:	ASYN_CANBLOCK, ASYN_MULTIDEVICE
autoConnect:	Yes/No
priority:	For port thread if ASYN_CANBLOCK
stackSize:	For port thread if ASYN_CANBLOCK

Based on these arguments base class constructor takes care of all details of registering port driver, registering asyn interfaces, registering interrupt sources, and creating parameter library.

asynPortDriver C++ Parameter Library Methods

```
virtual asynStatus createParam(const char *name, asynParamType type, int *index);

virtual asynStatus setIntegerParam(          int index, int value);
virtual asynStatus setIntegerParam(int list, int index, int value);
virtual asynStatus setDoubleParam(          int index, double value);
virtual asynStatus setDoubleParam(int list, int index, double value);
virtual asynStatus setStringParam(          int index, const char *value);
virtual asynStatus setStringParam(int list, int index, const char *value);

virtual asynStatus getIntegerParam(          int index, int * value);
virtual asynStatus getIntegerParam(int list, int index, int * value);
virtual asynStatus getDoubleParam(          int index, double * value);
virtual asynStatus getDoubleParam(int list, int index, double * value);
virtual asynStatus getStringParam(          int index, int maxChars, char *value);
virtual asynStatus getStringParam(int list, int index, int maxChars, char *value);

virtual asynStatus callParamCallbacks();
virtual asynStatus callParamCallbacks(int addr);
```

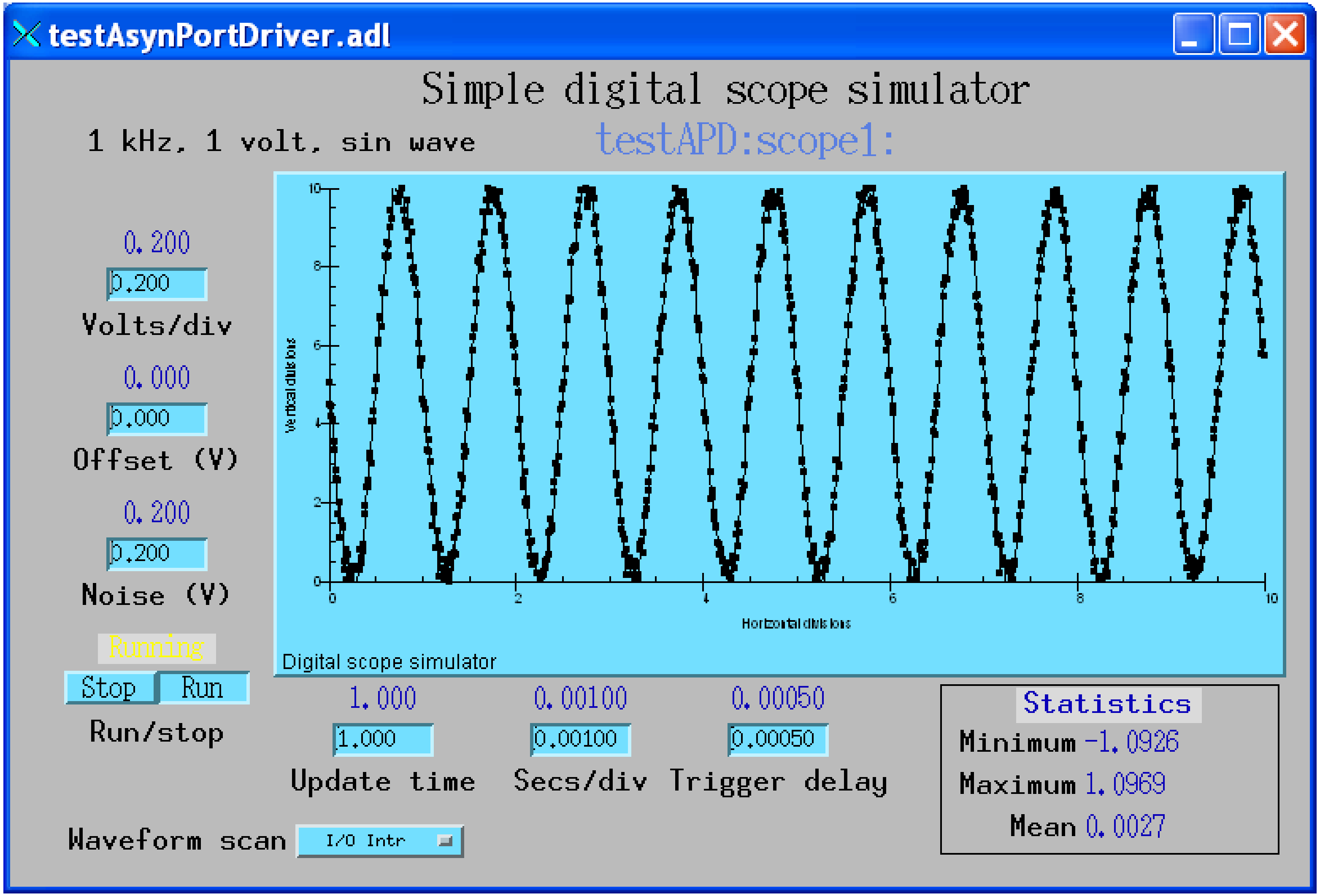
- These are the methods to write and read values from the parameter library, and to do callbacks to clients (e.g. device support) when parameters change

asynPortDriver Write/Read Methods

```
virtual asynStatus readInt32(asynUser *pasynUser, epicsInt32 *value);
virtual asynStatus writeInt32(asynUser *pasynUser, epicsInt32 value);
virtual asynStatus readFloat64(asynUser *pasynUser, epicsFloat64 *value);
virtual asynStatus writeFloat64(asynUser *pasynUser, epicsFloat64 value);
virtual asynStatus readOctet(asynUser *pasynUser, char *value, size_t maxChars,
                             size_t *nActual, int *eomReason);
virtual asynStatus writeOctet(asynUser *pasynUser, const char *value,
                              size_t maxChars,size_t *nActual);
virtual asynStatus readInt16Array(asynUser *pasynUser, epicsInt16 *value,
                                  size_t nElements, size_t *nIn);
virtual asynStatus writeInt16Array(asynUser *pasynUser, epicsInt16 *value,
                                   size_t nElements);
virtual asynStatus doCallbacksInt16Array(epicsInt16 *value, size_t nElements,
                                         int reason, int addr);
```

- These are the methods that device support calls to write a new value from an output record or to read a new value for an input record, (or initial read of an output record at iocInit).
- Drivers often don’t need to implement the readXXX functions, base class takes care of everything, i.e. get cached value from parameter library
- Need to implement the writeXXX methods if any immediate action is needed on write, otherwise can use base class implementation which just stores parameter in library

testAsynPortDriver
Digital Oscilloscope Simulator



testAsynPortDriver.adl

Simple digital scope simulator

1 kHz, 1 volt, sin wave

testAPD:scope1:

0.200
0.200
Volts/div

0.000
0.000
Offset (V)

0.200
0.200
Noise (V)

Running
Run/stop

1.000
1.000
Update time

0.00100
0.00100
Secs/div

0.00050
0.00050
Trigger delay

Statistics
Minimum -1.0926
Maximum 1.0969
Mean 0.0027

Waveform scan I/O Intr

testAsynPortDriver
Digital Oscilloscope Simulator

- 18 records (ao, ai, bo, bi, longin, waveform)
- All input records are I/O Intr scanned
  - Waveform can be switched I/O Intr or periodic
- Only 340 lines of well-commented C++ code
- Look in asyn\testAsynPortDriverApp\src

testAsynPortDriver Database

```
###################################################################
#  These records are the time per division                        #
###################################################################
record(ao, "$(P)$(R)TimePerDiv") {
   field(PINI, “YES")
   field(DTYP, "asynFloat64")
   field(OUT,  "@asyn($(PORT),$(ADDR),$(TIMEOUT))SCOPE_TIME_PER_DIV")
   field(PREC, "5")
}

record(ai, "$(P)$(R)TimePerDiv_RBV") {
   field(DTYP, "asynFloat64")
   field(INP,  "@asyn($(PORT),$(ADDR),$(TIMEOUT))SCOPE_TIME_PER_DIV")
   field(PREC, "5")
   field(SCAN, "I/O Intr")
}
```

DTYP=asynFloat64, standard asyn device support for ao record

drvInfo=SCOPE_TIME_PER_DIV;
  Defines which parameter this record is connected to

testAsynPortDriver Constructor

```
testAsynPortDriver::testAsynPortDriver(const char *portName, int maxPoints)
   : asynPortDriver(
       portName,  /* Name of port */

       1, /* maxAddr */

       NUM_SCOPE_PARAMS,  /* Number of parameters, computed in code */

       /* Interface mask */
       asynInt32Mask | asynFloat64Mask | asynFloat64ArrayMask | asynDrvUserMask,

       /* Interrupt mask */
       asynInt32Mask | asynFloat64Mask | asynFloat64ArrayMask,

       /* This driver does not block and it is not multi-device, so flag is 0 */
       0, /* Setting ASYN_CANBLOCK is all that is needed to make an
           * asynchronous driver */
       1, /* Autoconnect */
       0, /* Default priority */
       0) /* Default stack size*/
```

testAsynPortDriver Parameter creation

```
#define P_TimePerDivisionString    "SCOPE_TIME_PER_DIV"    /* asynFloat64,  r/w */
#define P_VoltsPerDivisionString   "SCOPE_VOLTS_PER_DIV"   /* asynFloat64,  r/w */
#define P_VoltOffsetString         "SCOPE_VOLT_OFFSET"     /* asynFloat64,  r/w */
#define P_TriggerDelayString       "SCOPE_TRIGGER_DELAY"   /* asynFloat64,  r/w */
#define P_NoiseAmplitudeString     "SCOPE_NOISE_AMPLITUDE" /* asynFloat64,  r/w */
#define P_UpdateTimeString         "SCOPE_UPDATE_TIME"     /* asynFloat64,  r/w */
#define P_WaveformString           "SCOPE_WAVEFORM"   /* asynFloat64Array,  r/o */

createParam(P_RunString,                asynParamInt32,         &P_Run);
createParam(P_MaxPointsString,          asynParamInt32,         &P_MaxPoints);
createParam(P_VoltOffsetString,         asynParamFloat64,       &P_VoltOffset);
createParam(P_TriggerDelayString,       asynParamFloat64,       &P_TriggerDelay);
createParam(P_UpdateTimeString,         asynParamFloat64,       &P_UpdateTime);
createParam(P_WaveformString,           asynParamFloat64Array,  &P_Waveform);
createParam(P_TimeBaseString,           asynParamFloat64Array,  &P_TimeBase);
createParam(P_MinValueString,           asynParamFloat64,       &P_MinValue);
createParam(P_MaxValueString,           asynParamFloat64,       &P_MaxValue);
createParam(P_MeanValueString,          asynParamFloat64,       &P_MeanValue);
```

testAsynPortDriver writeFloat64 method

```
asynStatus testAsynPortDriver::writeFloat64(asynUser *pasynUser,
  epicsFloat64 value)
{
    int function = pasynUser->reason;
    asynStatus status = asynSuccess;
    int run;
    const char *paramName;
    const char* functionName = "writeFloat64";

    /* Set the parameter in the parameter library. */
    status = (asynStatus) setDoubleParam(function, value);
```

testAsynPortDriver writeFloat64 method

```
if (function == P_UpdateTime) {
    /* Make sure the update time is valid.
     * If not change it and put back in parameter library */
    if (value < MIN_UPDATE_TIME) {
        value = MIN_UPDATE_TIME;
        setDoubleParam(P_UpdateTime, value);
    }
    /* If the update time has changed and we are running then wake
     * up the simulation task */
    getIntegerParam(P_Run, &run);
    if (run) epicsEventSignal(this->eventId);
} else {
    /* All other parameters just get set in parameter list, no need to
     * act on them here */
}

/* Do callbacks so higher layers see any changes */
status = (asynStatus) callParamCallbacks();
```

Example of Advantage of asynPortDriver
Acromag IP440/IP445 Digital I/O Modules

Traditional approach: xy2440 and xy2445 EPICS modules

```
devXy2440.c   459 lines
drvXy2445.h   189 lines
drvXy2445.c   939 lines
TOTAL        1587 lines

devXy2445.c   425 lines
drvXy2445.h   107 lines
drvXy2445.c   489 lines
TOTAL        1021 lines
```

Using asynPortDriver

```
drvIP440.cpp 211 lines  7.5 times fewer lines of code!!!
drvIP445.cpp 192 lines  5.3 times fewer lines of code!!!
```

Simple example: Acromag IP440/IP445 Digital I/O Modules

- Reasons for much less code using asynPortDriver:
  - Don’t need to write device support, we use standard asyn device support, eliminating the code in devXy2240.c and devXy2445.c
  - Don’t need to define the interface between driver and device support, eliminating drvXy2440.h and drvXy2445.h
  - Lots of features that asynPortDriver provides (callback support, etc.) that eliminates code from driver

- Additional features:
  - To turn on debugging in traditional version requires editing source code, recompiling and rebuilding the application
  - asynTrace allows turning on debugging in a standard way with asynTrace
  - asynReport provides base class in asynPortDriver for reporting many of the standard things the driver should report

asynPortDriver: Problems and Future Work

- asynPortDriver does not have a way for a driver to set an error status in the parameter library.
  - If the base class implementation of readInt32() is being used, for example, then it will always return asynSuccess if the parameter has ever been written to the library.
  - This is easy to fix by adding a new setParamStatus() function to asynPortDriver. Will be done in next release.
- asynPortDriver was my first real C++ project
  - It does not use C++ exceptions
  - Requires clumsy checking for status on every call to access the parameter library, etc.
  - A number of other things should be improved
  - However, too much code is based on the existing class to change it
  - I will make a new asynPortDriver2 class for new drivers (and converting existing drivers as time permits) that use exceptions and have other incompatible improvements