Mark Rivers, Marty Kraimer, Eric Norum

University of Chicago

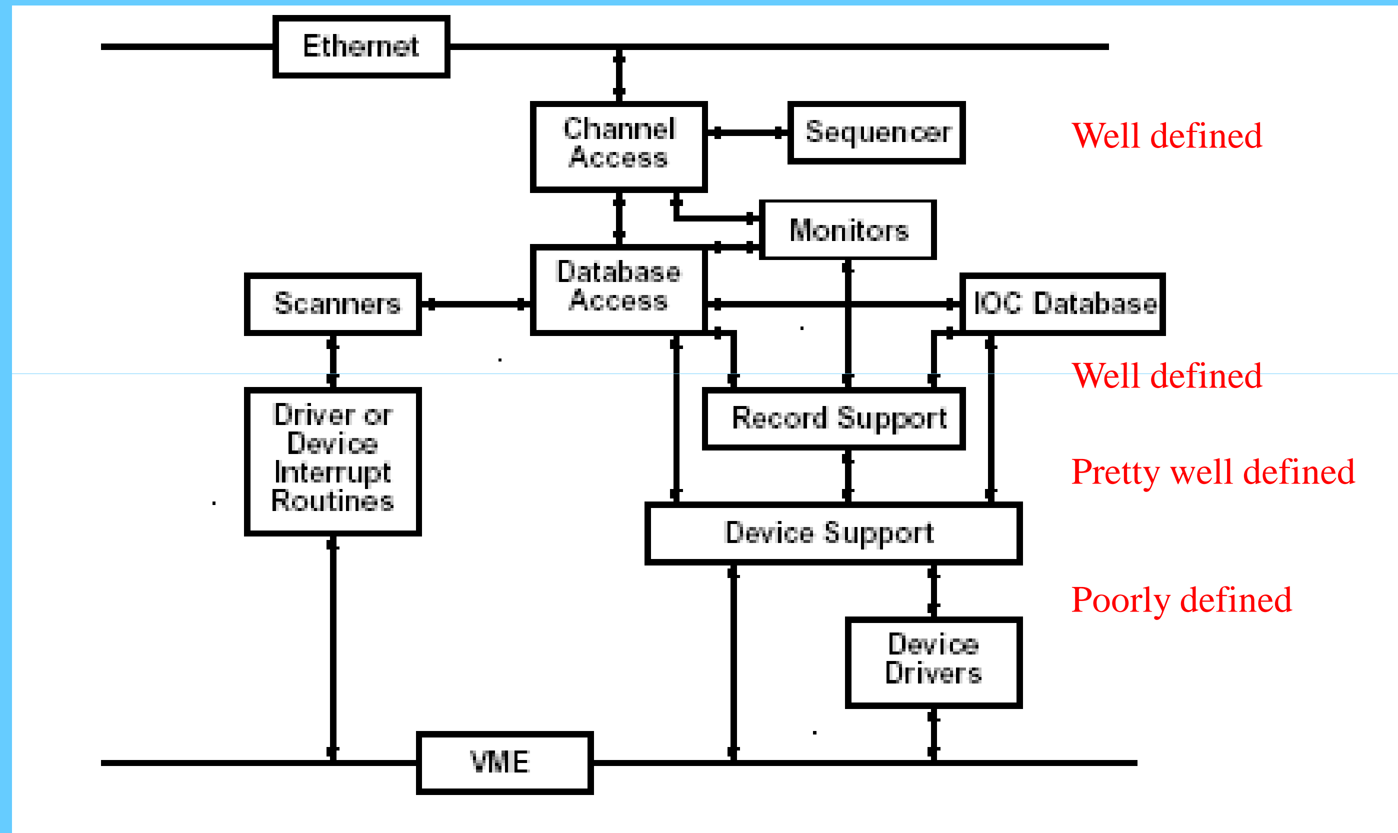
Advanced Photon Source

What is asyn and why to we need it?

Motivation

- Standard EPICS interface between device support and drivers is only loosely defined
- Needed custom device support for each driver
- asyn provides standard interface between device support and device drivers
- And a lot more too!

EPICS IOC architecture

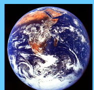


Well defined

Well defined

Pretty well defined

Poorly defined

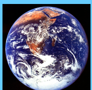


History – why the name asyn

- asyn replaces earlier APS packages called HiDEOS and MPF (Message Passing Facility)
- The initial releases of asyn were limited to “asynchronous” devices (e.g. slow devices)
  - Serial
  - GPIB
  - TCP/IP
- asyn provided the thread per port and queuing that this support needs.
- Current version of asyn is more general, synchronous (non-blocking) drivers are also supported.
- We are stuck with the name, or re-writing a LOT of code!

Independent of EPICS

- asyn is actually independent of EPICS (except for optional device support and asynRecord).
- It only uses libCom from EPICS base for OS independence in standard utilities like threads, mutexes, events, etc.
- asyn can be used in code that does not run in an IOC
  - asyn drivers could be used with Tango or other control systems



asyn Architecture

Device support (or SNL code, another driver, or non-EPICS software)

asynCommon (connect, report, …)

Interfaces (named; pure virtual functions)

asynOctet (write, read, setInputEos,…)

Port (named object)

Port driver

addr=0

addr=1

device

device

Control flow – asynchronous driver

Code running in application thread

Code running in port thread

Record Support

Record Device Support

ASYN

Work Queue

Low-level Driver

1 2 3 4 5 6 7 8

Control flow – synchronous driver

asynManager – Methods for drivers

- registerPort
  - Flags for multidevice (addr), canBlock, isAutoConnect
  - Creates thread for each asynchronous port (canBlock=1)
- registerInterface
  - asynCommon, asynOctet, asynInt32, etc.
- registerInterruptSource, interruptStart, interruptEnd
- interposeInterface – e.g. interposeEos, interposeFlush
- Example code:

```
pPvt->int32Array.interfaceType = asynInt32ArrayType;
pPvt->int32Array.pinterface  = (void *)&drvIp330Int32Array;
pPvt->int32Array.drvPvt = pPvt;
status = pasynManager->registerPort(portName,
                 ASYN_MULTIDEVICE, /*is multiDevice*/
                 1,  /*  autoconnect */
                 0,  /* medium priority */
                 0); /* default stack size */
status = pasynManager->registerInterface(portName,&pPvt->common);
status = pasynInt32Base->initialize(pPvt->portName,&pPvt->int32);
pasynManager->registerInterruptSource(portName, &pPvt->int32,
                 &pPvt->int32InterruptPvt);
```

asynManager – Methods for Clients (e.g. Device Support)

- Create asynUser
- Connect asynUser to device (port)
- Find interface (e.g. asynOctet, asynInt32, etc.)
- Register interrupt callbacks
- Query driver characteristics (canBlock, isMultidevice, isEnabled, etc).
- Queue request for I/O to port
  - asynManager calls callback when port is free
    - Will be separate thread for asynchronous port
  - I/O calls done directly to interface methods in driver
    - e.g. pasynOctet->write()

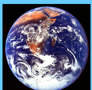
asynManager – Methods for Clients (e.g. Device Support)

Example code:

```
/* Create asynUser */
pasynUser = pasynManager->createAsynUser(processCallback, 0);
status = pasynEpicsUtils->parseLink(pasynUser, plink,
                &pPvt->portName, &pPvt->addr, &pPvt->userParam);
status = pasynManager->connectDevice(pasynUser, pPvt->portName, pPvt->addr);
status = pasynManager->canBlock(pPvt->pasynUser, &pPvt->canBlock);
pasynInterface = pasynManager->findInterface(pasynUser, asynInt32Type, 1);
status = pasynManager->queueRequest(pPvt->pasynUser, 0, 0);
```

In processCallback()

```
status = pPvt->pint32->read(pPvt->int32Pvt, pPvt->pasynUser, &pPvt->value);
```



asynManager – asynUser

- asynUser data structure. This is the fundamental “handle” used by asyn

```
asynUser = pasynManager->createAsynUser(userCallback queue,
                                        userCallback timeout);
asynUser = pasynManager->duplicateAsynUser)(pasynUser,
                                        userCallback queue,
                                        userCallback timeout);
typedef struct asynUser {
    char *errorMessage;
    int errorMessageSize;
    /* The following must be set by the user */
    double       timeout;  /* Timeout for I/O operations */
    void         *userPvt;
    void         *userData;
    /* The following is for user to/from driver communication */
    void         *drvUser;
    /* The following is normally set by driver */
    int          reason;
    /* The following are for additional information from method calls */
    int          auxStatus; /* For auxillary status /
} asynUser;
```

Standard Interfaces

Common interface, all drivers must implement

- asynCommon: report(), connect(), disconnect()

I/O Interfaces, most drivers implement one or more

- All of these have write(), read(), registerInteruptUser() and cancelInterruptUser() methods
- asynOctet: flush(), setInputEos(), setOutputEos(), getInputEos(), getOutputEos()
- asynInt32: getBounds()
- asynInt8Array, asynInt16Array, asynInt32Array:
- asynUInt32Digital:
- asynFloat64:
- asynFloat32Array, asynFloat64Array:

Miscellaneous interfaces

- asynOption: setOption() getOption()
- asynGpib: addressCommand(), universalCommand(), ifc(), ren(), etc.
- asynDrvUser: create(), free()

Support for Callbacks (Interrupts)

- The standard interfaces asynOctet, asynInt32, asynUInt32Digital, asynFloat64 and asynXXXArray all support callback methods for interrupts
- registerInterruptUser(…,userFunction, userPrivate, …)
  - Driver will call userFunction(userPrivate, pasynUser, data) whenever an interrupt occurs
  - Callback will not be at interrupt level, so callback is not restricted in what it can do
- Callbacks can be used by device support, other drivers, etc.
- Current interrupt drivers
  - Ip330 ADC, IpUnidig binary I/O, quadEM APS quad electrometer, areaDetector drivers
  - Acromag IP440/IP445, HMS simulators for labs

Support for Interrupts – Performance

- Ip330 ADC driver. Digitizing 16 channels at 1kHz.
- Generates interrupts at 1 kHz.
- Each interrupt results in:
  - 16 asynInt32 callbacks to devInt32Average generic device support
  - 1 asynInt32Array callback to fastSweep device support for MCA records
  - 1 asynFloat64 callback to devEpidFast for fast feedback
- 18,000 callbacks per second
- 21% CPU load on MVME2100 PPC-603 CPU with feedback on and MCA fast sweep acquiring.

Generic Device Support

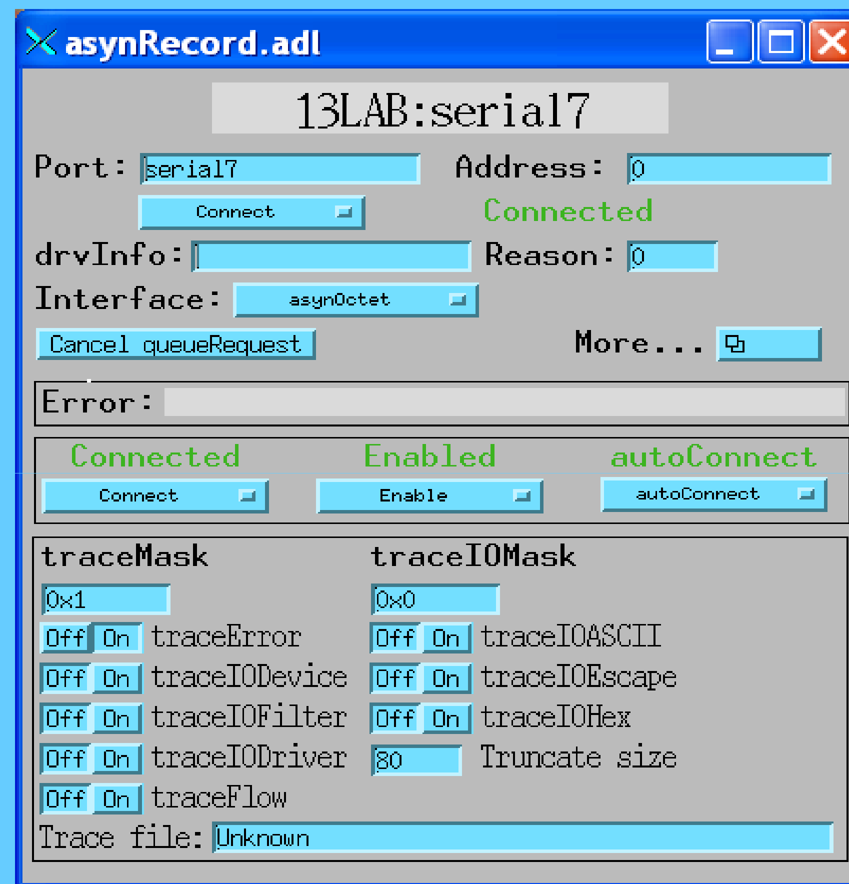
- asyn includes generic device support for many standard EPICS records and standard asyn interfaces
- Eliminates need to write device support in many cases. New hardware can be supported by writing just a driver.
- Record fields:
  - field(DTYP, “asynInt32”)
  - field(INP, “@asyn(portName, addr, timeout) drvInfoString)
- Examples:
  - asynInt32
    - ao, ai, bo, bi, mbbo, mbbi, longout, longin
  - asynInt32Average
    - ai
  - asynUInt32Digital, asynUInt32DigitalInterrupt
    - bo, bi, mbbo, mbbi, mbboDirect, mbbiDirect, longout, longin
  - asynFloat64
    - ai, ao
  - asynOctet
    - stringin, stringout, waveform
  - asynXXXArray
    - waveform (in and out)

Generic Device Support

- The following synApps modules all now use standard asyn device support, and no longer have specialized device support code:
  - Ip330 ADC
  - IpUnidig
  - quadEM
  - dac128V
  - Canberra ICB modules (Amp, ADC, HVPS, TCA)
- MCA and motor records use special device support, because they are not base record types
- However, the MCA and new motor drivers now only use the standard asyn interfaces, so it is possible to write a database using only standard records and control any MCA driver or new motor driver

asynRecord

- EPICS record that provides access to most features of asyn, including standard I/O interfaces
- Applications:
  - Control tracing (debugging)
  - Connection management
  - Perform interactive I/O
- Very useful for testing, debugging, and actual I/O in many cases
- Replaces the old generic "serial" and "gpib" records, but much more powerful



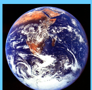
Synchronous interfaces

- Standard interfaces also have a synchronous interface, even for slow devices, so that one can do I/O without having to implement callbacks
- Example: asynOctetSyncIO
  - write(), read(), writeRead()
- Very useful when communicating with a device that can block, when it is OK to block
- Example applications:
  - EPICS device support in init_record(), (but not after that!)
  - SNL programs, e.g. communicating with serial or TCP/IP ports
  - Any asynchronous asyn port driver communicating with an underlying asynOctet port driver (e.g. motor drivers)
  - HMS simulator port driver for lab
  - areaDetector driver talking to marCCD server, Pilatus camserver, etc.
  - iocsh commands

Tracing and Debugging

- Standard mechanism for printing diagnostic messages in device support and drivers
- Messages written using EPICS logging facility, can be sent to stdout, stderr, or to a file.
- Device support and drivers call:
  - asynPrint(pasynUser, reason, format, ...)
  - asynPrintIO(pasynUser, reason, buffer, len, format, ...)
  - Reason:
    - ASYN_TRACE_ERROR
    - ASYN_TRACEIO_DEVICE
    - ASYN_TRACEIO_FILTER
    - ASYN_TRACEIO_DRIVER
    - ASYN_TRACE_FLOW
- Tracing is enabled/disabled for (port/addr)
- Trace messages can be turned on/off from iocsh, vxWorks shell, and from CA clients such as medm via asynRecord.
- asynOctet I/O from shell

asynRecord.adl
13LAB:serial1
Port: serial1 Address: 0
drvInfo:
Interface: asynOctet
Cancel queueRequest More...
Error:
Connected Enabled autoConnect
Connect Enable autoConnect
traceMask traceIOMask
0x1 0x0
Off On traceError Off On traceIOASCII
Off On traceIODevice Off On traceIOEscape
Off On traceIOFilter Off On traceIOHex
Off On traceIODriver 80 Truncate size
Off On traceFlow
Trace file: Unknown

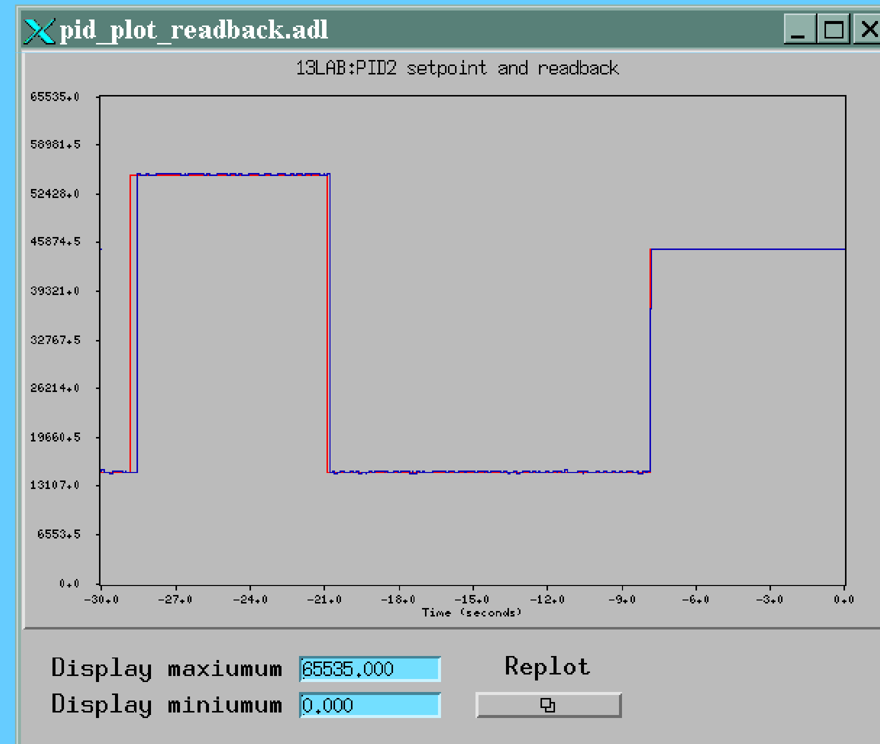


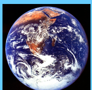
Fast feedback device support (epid record)

- Supports fast PID control
- Input: any driver that supports asynFloat64 with callbacks (e.g. callback on interrupt)
- Output: any driver that supports asynFloat64.
- In real use at APS for monochromator feedback with IP ADC/DAC, and APS VME beam position monitor and DAC
- >1kHz feedback rate

pid_control.adl
Fast_Feedback
Readback PV #C1 S0 @Ip330PID_1
Control PV
Setpoint 45000.000
Readback 44995.000
Feedback On
Update rate .1 second
More

pid_parameters.adl
PID feedback parameters
KP 0.020
P
KI 300.000
I 2430.653
KD 0.000
D 0.000
Delta time 0.001
Error 0.000
Output 2430.000
Low limit 1024.000
High limit 3072.000





2010

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 2 drivers I’ve written for this class (Acromag IP440/IP445 binary I/O and HMS simulator) use asynPortDriver
- Hides all details of registering interfaces, registering interrupt sources, doing callbacks, default connection management

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 on new value:
    - New parameter value arrives, or new data arrives from device
    - Change values of one or more parameters
    - 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 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 data type/asyn interface (int32, float32, etc.), caches the current value, maintains changed flag
  - Drivers use asynPortDriver methods to read the current value from the table, and to set new values in the table.
  - Method to call all registered callbacks for values that have changed since callbacks were last done.

asynPortDriver C++ 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);
```

- Drivers typically 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

asynPortDriver C++ 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 (typically 1)
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.

Summary- Advantages of asyn

- Drivers implement standard interfaces that can be accessed from:
  - Multiple record types
  - SNL programs
  - Other drivers
- Generic device support eliminates the need for separate device support in 90% (?) of cases
  - synApps package 10-20% fewer lines of code, 50% fewer files with asyn
- Consistent trace/debugging at (port, addr) level
- asynRecord can be used for testing, debugging, and actual I/O applications
- Easy to add asyn interfaces to existing drivers:
  - Register port, implement interface write(), read() and change debugging output
  - Preserve 90% of driver code
- asyn drivers are actually EPICS-independent. Can be used in any other control system.