asyn: An Interface Between EPICS Drivers and Device Support

Mark Rivers, Marty Kraimer, Eric Norum

University of Chicago

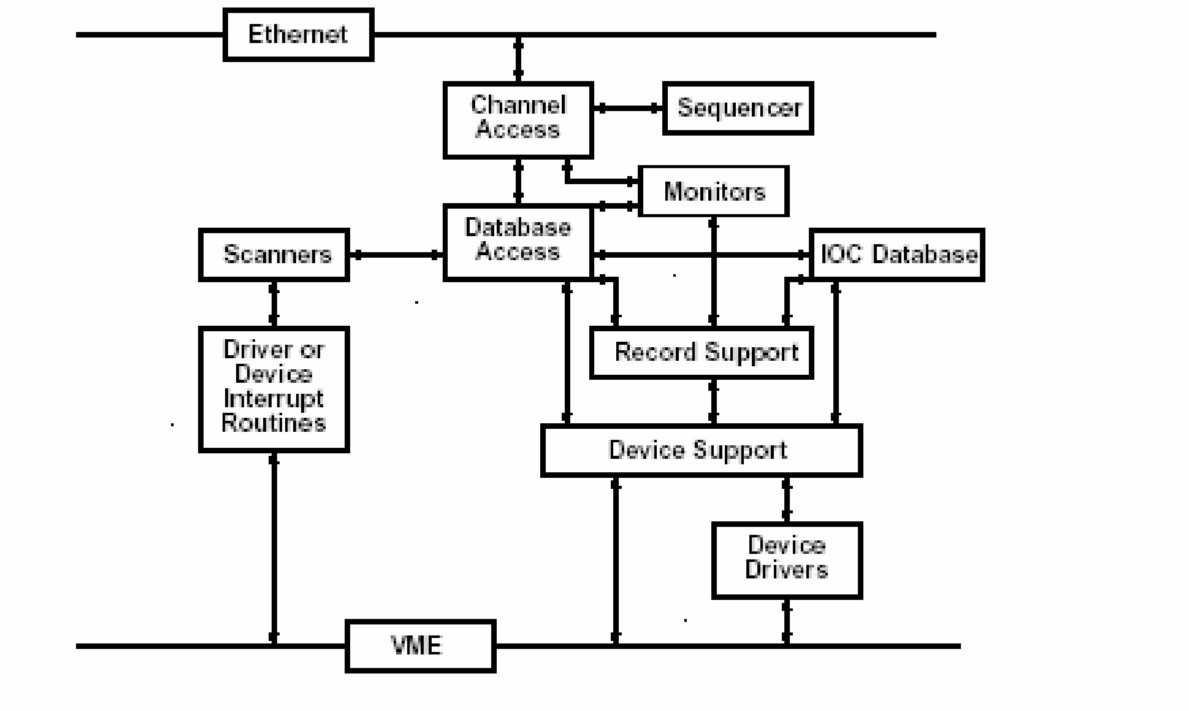
Advanced Photon Source

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!

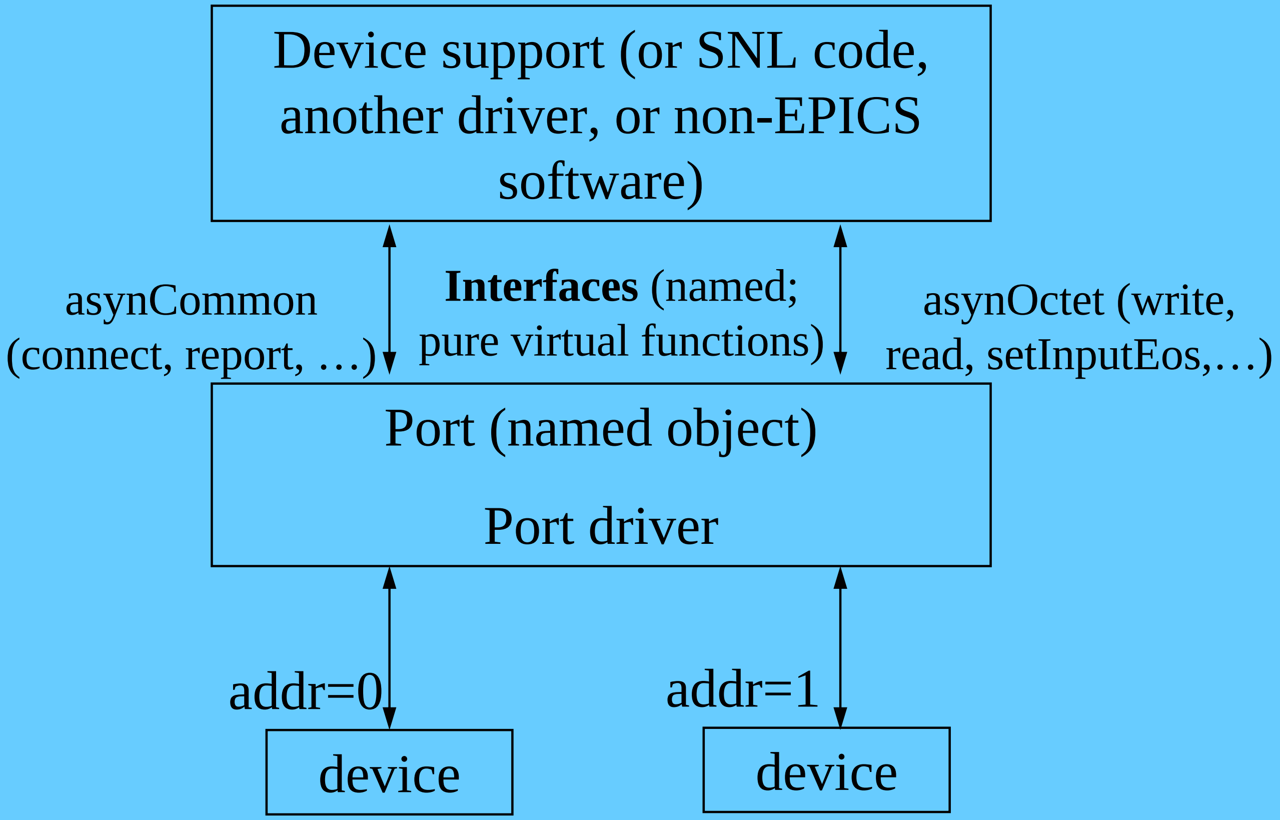
EPICS IOC architecture



History – why the name asyn

- 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!

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

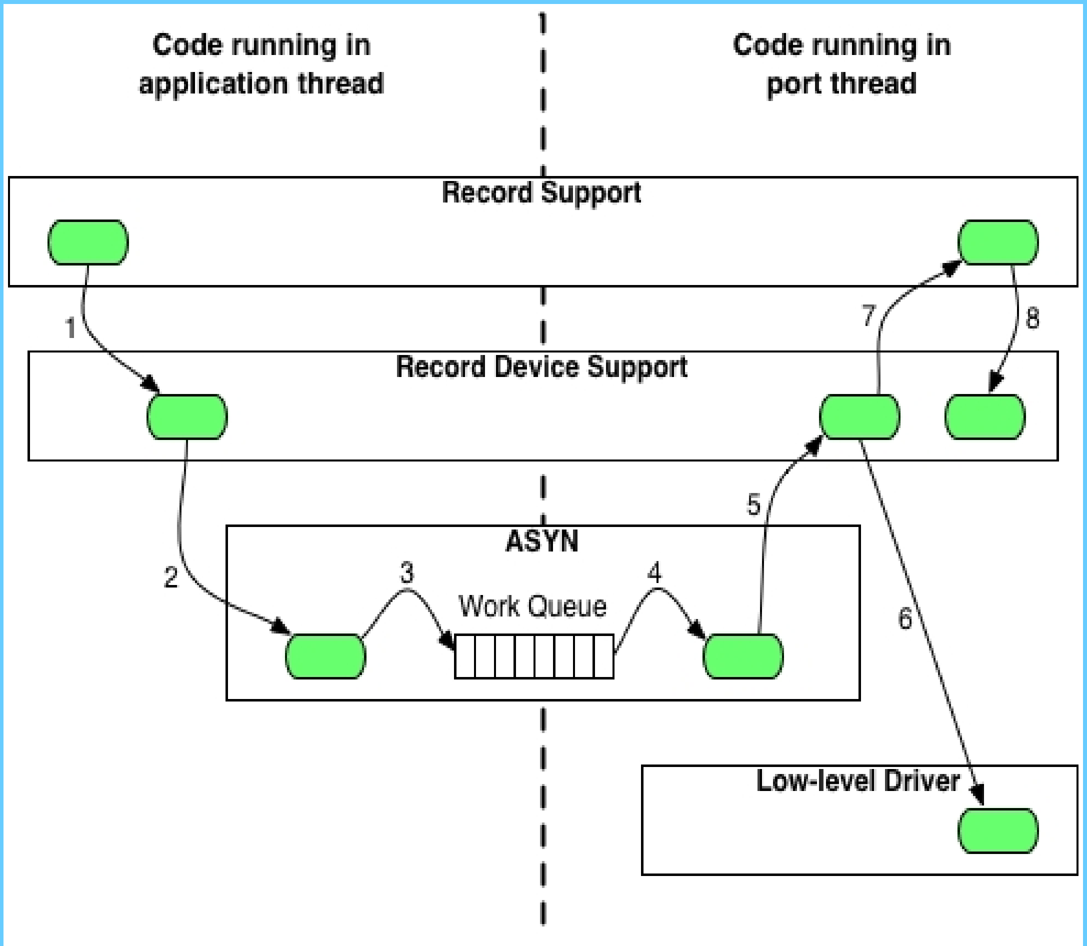
addr=0

addr=1

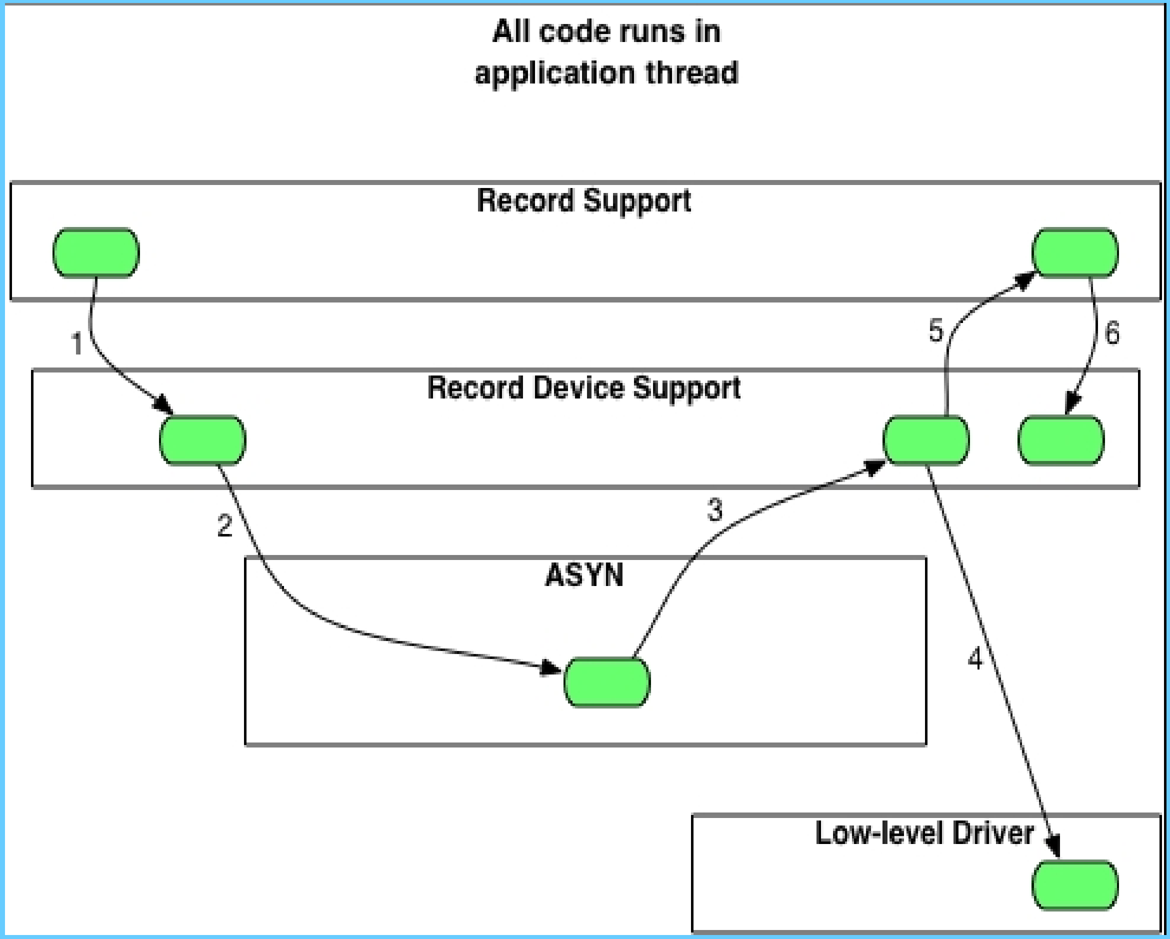
device

device

Control flow – asynchronous driver



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
- 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 Device Support

- Connect to device (port)
- Create asynUser
- 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()
- 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);
...
 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
   process,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 have write(), read(), registerInteruptUser() and cancelInterruptUser() methods
- asynOctet: writeRaw(), readRaw(), flush(), setInputEos(), setOutputEos(), getInputEos(), getOutputEos()
- asynInt32: getBounds()
- asynInt32Array:
- asynUInt32Digital:
- asynFloat64:
- asynFloat64Array:

Miscellaneous interfaces

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

Standard Interfaces - drvUser

- pdrvUser->create(void *drvPvt, asynUser *pasynUser, const char *drvInfo, const char **pptypeName, size_t *psize);
- drvInfo string is parsed by driver.
- It typically sets pasynUser->reason to an enum value (e.g. mcaElapsedLive, mcaErase, etc.)
- More complex driver could set pasynUser->drvUser to a pointer to something.
- Example

```
grecord(mbbo,"$(P)$(HVPS)INH_LEVEL") {
    field(DESC,"Inhibit voltage level")
    field(PINI,"YES")
    field(ZRVL,"0")
    field(ZRST,"+5V")
    field(ONVL,"1")
    field(ONST,"+12V")
    field(DTYP, "asynInt32")
    field(OUT,"@asyn($(PORT))INHIBIT_LEVEL")
}
status = pasynEpicsUtils->parseLink(pasynUser, plink,
                &pPvt->portName, &pPvt->addr, &pPvt->userParam);
pasynInterface = pasynManager->findInterface(pasynUser, asynDrvUserType,1);
status = pasynDrvUser->create(drvPvt,pasynUser,pPvt->userParam,0,0);
```

Support for Interrupts

- The standard interfaces asynInt32, asynInt32Array, asynUInt32Digital, asynFloat64 and asynFloat64Array 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

Support for Interrupts – Ip330 driver

```
static void intFunc(void *drvPvt)
{
...
for (i = pPvt->firstChan; i <= pPvt->lastChan; i++) {
        data[i] = (pPvt->regs->mailBox[i + pPvt->mailBoxOffset]);
    }
    /* Wake up task which calls callback routines */
    if (epicsMessageQueueTrySend(pPvt->intMsgQId, data, sizeof(data)) == 0)
...
}
static void intTask(drvIp330Pvt *pPvt)
{
while(1) {
        /* Wait for event from interrupt routine */
        epicsMessageQueueReceive(pPvt->intMsgQId, data, sizeof(data));
        /* Pass int32 interrupts */
        pasynManager->interruptStart(pPvt->int32InterruptPvt, &pclientList);
        pnode = (interruptNode *)ellFirst(pclientList);
        while (pnode) {
            asynInt32Interrupt *pint32Interrupt = pnode->drvPvt;
            addr = pint32Interrupt->addr;
            reason = pint32Interrupt->pasynUser->reason;
            if (reason == ip330Data) {
                pint32Interrupt->callback(pint32Interrupt->userPvt,
                                          pint32Interrupt->pasynUser,
                                          pPvt->correctedData[addr]);
            }
            pnode = (interruptNode *)ellNext(&pnode->node);
        }
        pasynManager->interruptEnd(pPvt->int32InterruptPvt);
...
}
```

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

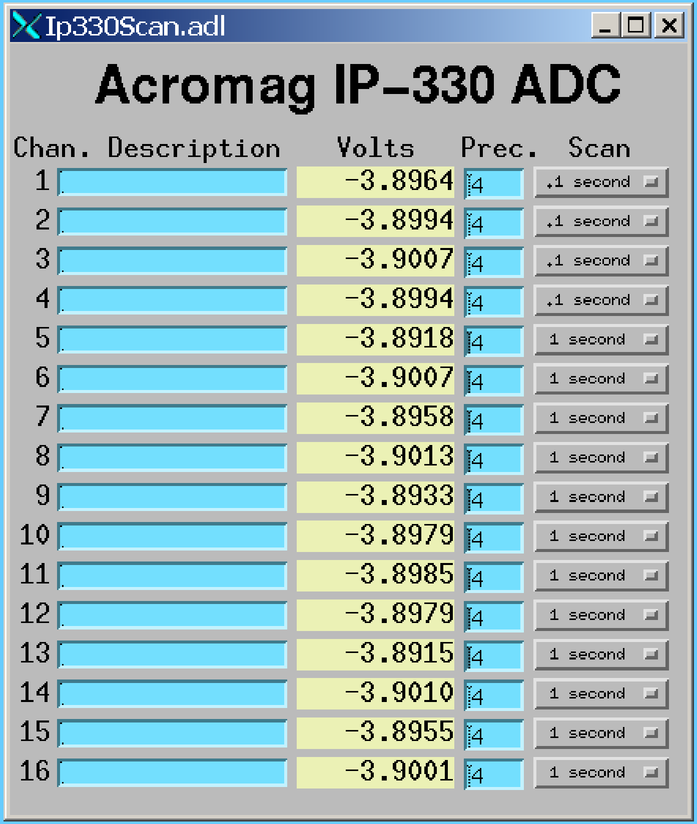
- asyn includes generic device support for many standard EPICS records and standard asyn interfaces
- Eliminates need to write device support in many cases. New hardware can be supported by writing just a driver.
- Record fields:
  - field(DTYP, “asynInt32”)
  - field(INP, “@asyn(portName, addr, timeout)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

Generic Device Support

- The following 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 DXP records use special device support, because they are not base record types
- However, the MCA drivers now only use the standard asyn interfaces, so it would be possible to write a database using only standard records and control any MCA driver (Canberra, DXP, etc.).

Generic Device Support

```
corvette> view ../Db/ip330Scan.template
record(ai,"$(P)$(R)")
{
        field(SCAN,"$(SCAN)")
        field(DTYP,"asynInt32Average")
        field(INP,"@asyn($(PORT) $(S))DATA")
        field(LINR,"LINEAR")
        field(EGUF,"$(EGUF)")
        field(EGUL,"$(EGUL)")
        field(HOPR,"$(HOPR)")
        field(LOPR,"$(LOPR)")
        field(PREC,"$(PREC)")
}

record(longout,"$(P)$(R)Gain")
{
        field(PINI,"YES")
        field(VAL,"$(GAIN)")
        field(DTYP,"asynInt32")
        field(OUT,"@asyn($(PORT) $(S))GAIN")
}
```



Ip330Scan.adl

Acromag IP-330 ADC

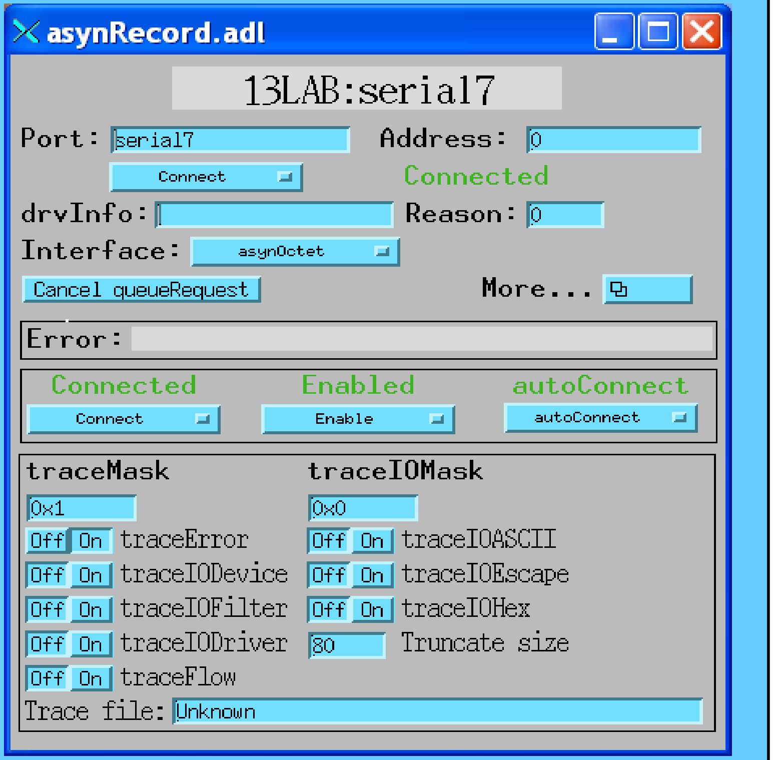
Chan.	Description	Volts	Prec.	Scan
1		-3.8964	4	.1 second
2		-3.8994	4	.1 second
3		-3.9007	4	.1 second
4		-3.8994	4	.1 second
5		-3.8918	4	1 second
6		-3.9007	4	1 second
7		-3.8958	4	1 second
8		-3.9013	4	1 second
9		-3.8933	4	1 second
10		-3.8979	4	1 second
11		-3.8985	4	1 second
12		-3.8979	4	1 second
13		-3.8915	4	1 second
14		-3.9010	4	1 second
15		-3.8955	4	1 second
16		-3.9001	4	1 second

Other Device Support

- synApps “ip” application is converted to asyn
  - devXxStrParm
  - devAiMks – MKS vacuum gauge controller
  - devMpc – MPC ion pump and TSP controller
- Love controller support being converted
- GPIB and serial support using configuration files (gpibCore)
- STREAMS and devAscii being converted

asynRecord

- New 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 poweful



asynRecord.adl

13LAB:serial7

Port: serial7 Address: 0

Connected

drvInfo: Reason: 0

Interface: asynOctet

Error:

Connected Enabled autoConnect

traceMask: 0x1

traceIOMask: 0x0

Off On traceError

Off On traceIODevice

Off On traceIOFilter

Off On traceIODriver

Off On traceFlow

Off On traceIOASCII

Off On traceIOEscape

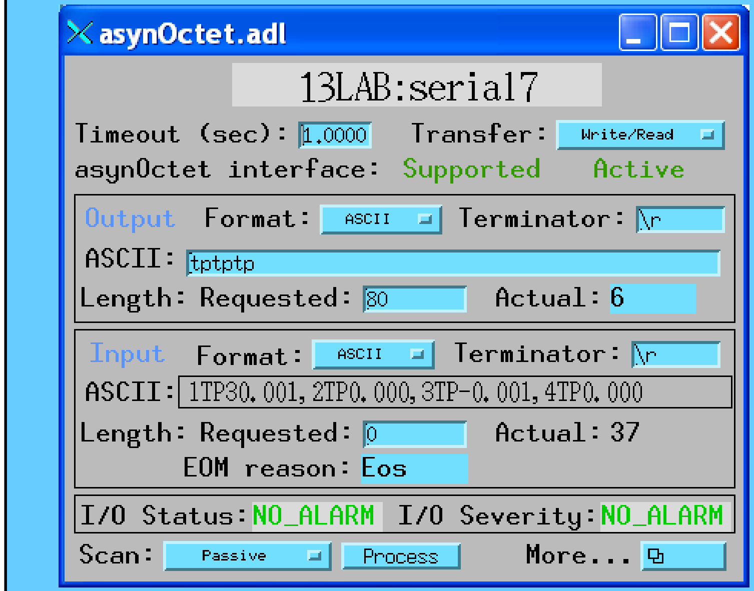
Off On traceIOHex

80 Truncate size

Trace file: Unknown

asynRecord – asynOctet devices

Interactive I/O to serial device



asynOctet.adl

13LAB:serial7

Timeout (sec): 1.0000 Transfer: Write/Read

asynOctet interface: Supported Active

Output Format: ASCII Terminator: \r

ASCII: tptptp

Length: Requested: 80 Actual: 6

Input Format: ASCII Terminator: \r

ASCII: 1TP30.001,2TP0.000,3TP-0.001,4TP0.000

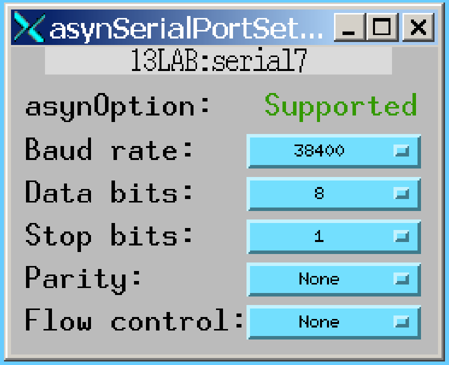
Length: Requested: 0 Actual: 37

EOM reason: Eos

I/O Status: NO_ALARM I/O Severity: NO_ALARM

Scan: Passive Process More...

Configure serial port parameters



asynSerialPortSet...

13LAB:serial7

asynOption: Supported

Baud rate: 38400

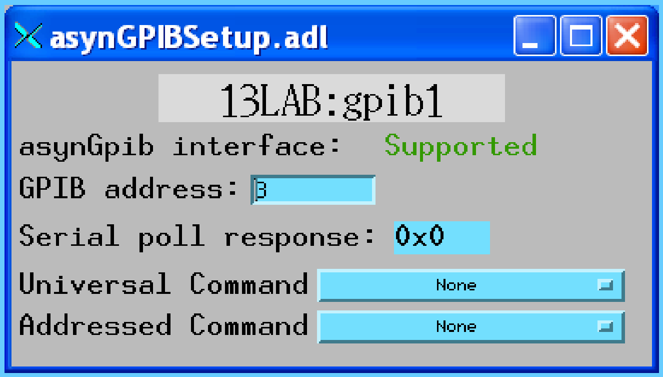
Data bits: 8

Stop bits: 1

Parity: None

Flow control: None

Perform GPIB specific operations



asynGPIBSetup.adl

13LAB:gpib1

asynGpib interface: Supported

GPIB address: 3

Serial poll response: 0x0

Universal Command None

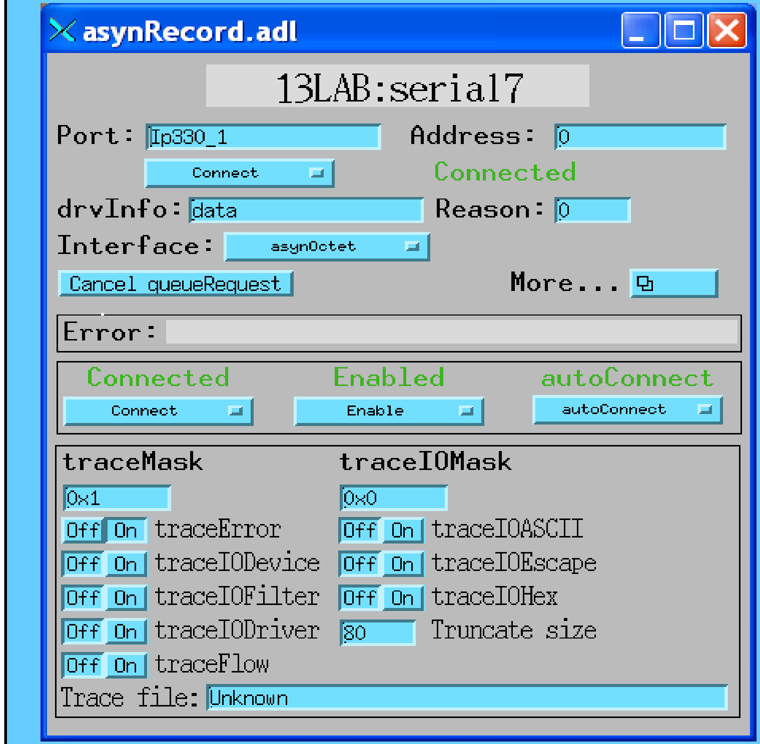
Addressed Command None

asynRecord – Differences from generic serial and generic gpib records

- ODEL field replaced by OEOS. Changed from a DBF_LONG to DBF_STRING to support multi-character terminators. The IDEL (serial) and EOS (gpib) fields eplaced by IEOS. Changed from a DBF_LONG to DBF_STRING to support multi-character terminators.
- IEOS and OEOS fields only used if modified after connecting to port. Fields set to current eos strings for the port when connecting.
- INP field replaced by PORT and ADDR fields to support run-time connection to different devices.
- AOUT and OEOS fields are processed by dbTranslateEscape before being sent to the device. In rare cases this may require changing the output strings if these contained the "\" character.
- asyn record always posts monitors on the input field (AINP or BINP) when the record processes. Older records did not post monitors on the AINP field if the value was the same as the previous read. This caused problems for some SNL programs and data acquisition applications.
- ODEL and IDEL were used even when OFMT or IFMT were in "Binary" mode. OEOS and IEOS are now ignored when OFMT or IFMT respectively are in "Binary" mode, because readRaw and writeRaw are called.
- TMOT field has changed from DBF_LONG to DBF_DOUBLE, and the units have changed from milliseconds to seconds. TMOT=-1.0 now means wait forever.

asynRecord – register devices

Same asynRecord, change to ADC port



asynRecord.adl

13LAB:serial7

Port: Ip330_1 Address: 0

Connect Connected

drvInfo: data Reason: 0

Interface: asynOctet

Cancel queueRequest More...

Error:

Connected Enabled autoConnect

Connect Enable autoConnect

traceMask: 0x1

Off On traceError

Off On traceIODevice

Off On traceIOFilter

Off On traceIODriver

Off On traceFlow

traceIOMask: 0x0

Off On traceIOASCII

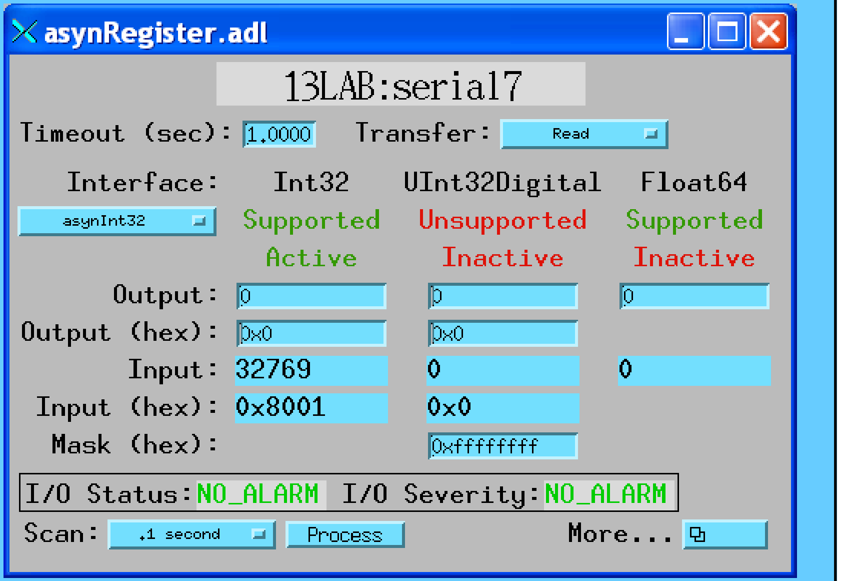
Off On traceIOEscape

Off On traceIOHex

80 Truncate size

Trace file: Unknown

Read ADC at 10Hz with asynInt32 interface



asynRegister.adl

13LAB:serial7

Timeout (sec): 1.0000 Transfer: Read

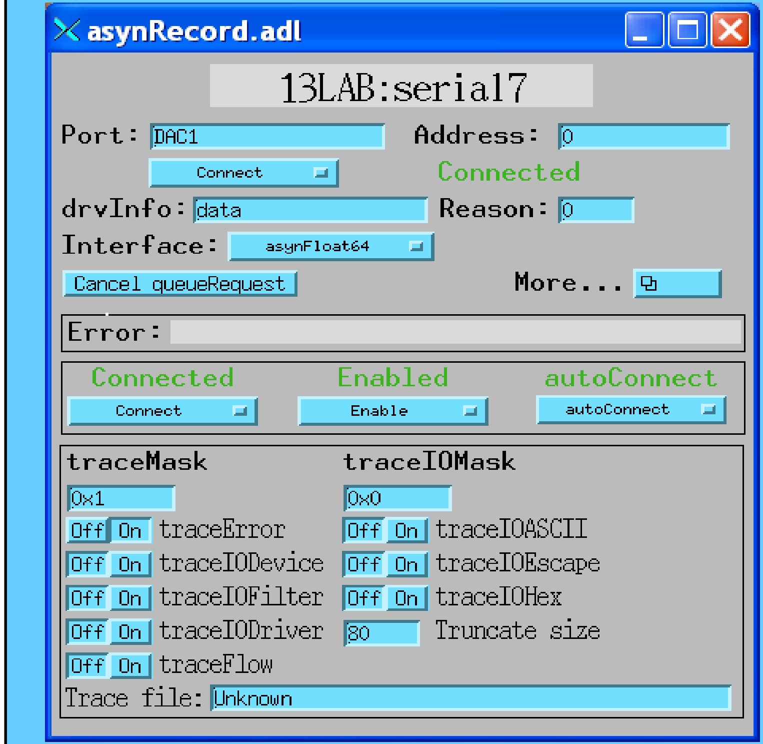
```
Interface:      Int32        UInt32Digital   Float64
asynInt32       Supported    Unsupported     Supported
                Active       Inactive        Inactive
Output:         0            0               0
Output (hex):   0x0          0x0
Input:          32769        0               0
Input (hex):    0x8001       0x0
Mask (hex):                  0xffffffff
```

I/O Status: NO_ALARM I/O Severity: NO_ALARM

Scan: .1 second Process More...

asynRecord – register devices

Same asynRecord, change to DAC port



asynRecord.adl

13LAB:serial7

Port: DAC1 Address: 0

Connect Connected

drvInfo: data Reason: 0

Interface: asynFloat64

Cancel queueRequest More...

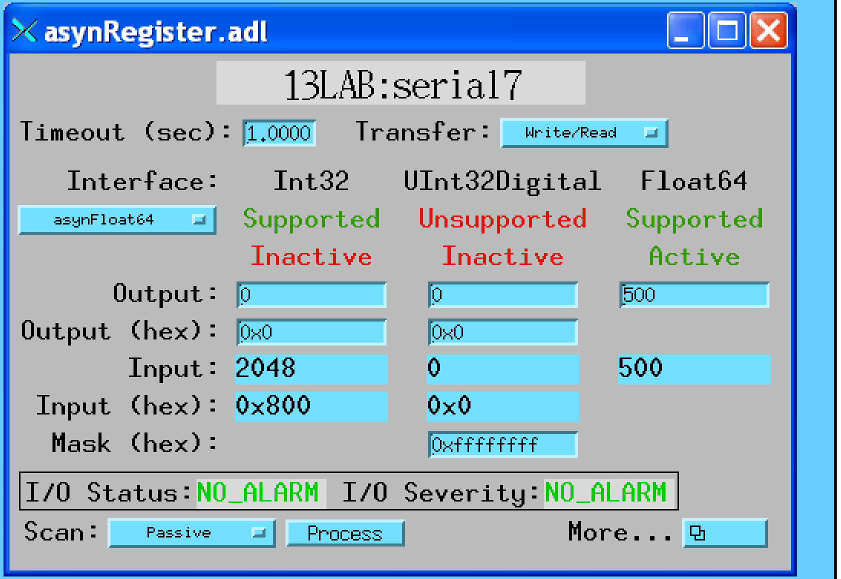
Error:

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

Write DAC with asynFloat64 interface



asynRegister.adl

13LAB:serial7

Timeout (sec): 1.0000 Transfer: Write/Read

Interface:	Int32	UInt32Digital	Float64
asynFloat64	Supported	Unsupported	Supported
	Inactive	Inactive	Active
Output:	0	0	500
Output (hex):	0x0	0x0	
Input:	2048	0	500
Input (hex):	0x800	0x0	
Mask (hex):		0xffffffff	

I/O Status: NO_ALARM I/O Severity: NO_ALARM

Scan: Passive Process More...

GeoSoilEnviroCARS

Getting Started with EPICS November 18, 2004

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
  - Motor drivers running in separate thread
  - iocsh commands

Synchronous interfaces – motor driver example

- In initialization:

```
/* Initialize communications channel */
success_rtn = pasynOctetSyncIO->connect(cntrl->asyn_port,
                        cntrl->asyn_address, &cntrl->pasynUser, NULL);
```

- In IO:

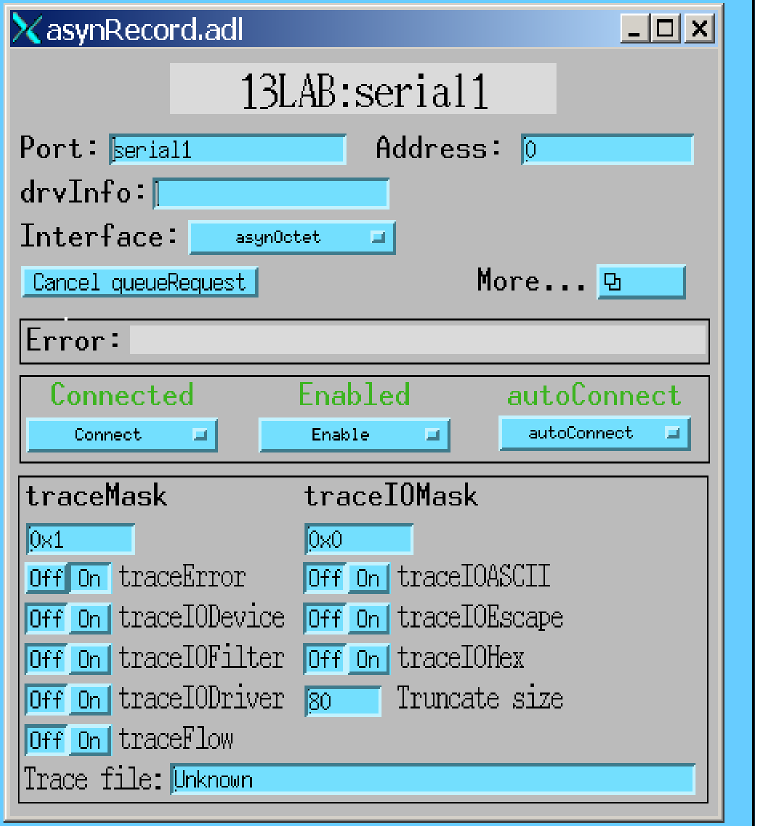
```
pasynOctetSyncIO->write(cntrl->pasynUser, com, strlen(com),
                           TIMEOUT, &nwrite);

status = pasynOctetSyncIO->read(cntrl->pasynUser, com, BUFF_SIZE,
                                   timeout, &nread, &eomReason);
```

iocsh Commands

```
asynReport(filename,level,portName)
asynInterposeFlushConfig(portName,addr,timeout)
asynInterposeEosConfig(portName,addr)
asynSetTraceMask(portName,addr,mask)
asynSetTraceIOMask(portName,addr,mask)
asynSetTraceFile(portName,addr,filename)
asynSetTraceIOTruncateSize(portName,addr,size)
asynSetOption(portName,addr,key,val)
asynShowOption(portName,addr,key)
asynAutoConnect(portName,addr,yesNo)
asynEnable(portName,addr,yesNo)
asynOctetConnect(entry,portName,addr,oeos,ieos,timeout,buffer_len)
asynOctetRead(entry,nread,flush) asynOctetWrite(entry,output)
asynOctetWriteRead(entry,output,nread) asynOctetFlush(entry)
asynOctetSetInputEos(portName,addr,eos,drvInfo)
asynOctetGetInputEos(portName,addr,drvInfo)
asynOctetSetOutputEos(portName,addr,eos,drvInfo)
asynOctetGetOutputEos(portName,addr,drvInfo)
```

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

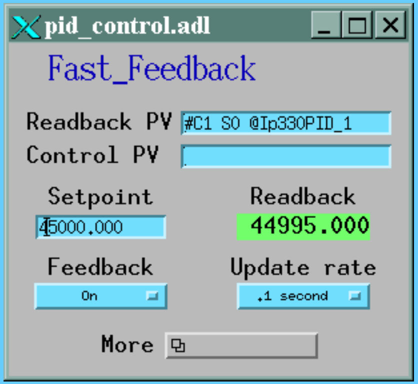
Current asyn Drivers

- Unix/Linux/vxWorks/cygwin serial ports
- TCP/IP sockets
- GPIB via National Instruments VME, Ethernet/GPIB devices, Ip488 Industry Pack modules
- VXI-11
- IpUnidig digital I/O (Industry Pack). Supports interrupts.
- dac128V digital-to-analog (Industry Pack)
- Ip330 analog-to-digital (Industry Pack). Supports interrupts.
- Canberra AIM multi-channel analyzer and ICB modules (Ethernet)
- XIA DXP DSP spectroscopy system (CAMAC, EPP, PXI soon)
- APS quad electrometer (VME). Supports interrupts.
- epid record fast feedback (float 64 with callbacks for input, float64 for output)
- Mca fast-sweep (Int32Array with callbacks)

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

GeoSoilEnviroCARS


pid_control.adl

Fast_Feedback

Readback PV #C1 S0 @Ip330PID_1

Control PV

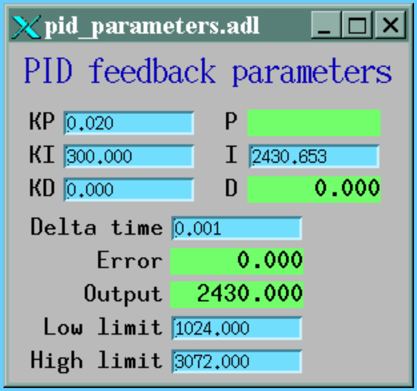
Setpoint 45000.000

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

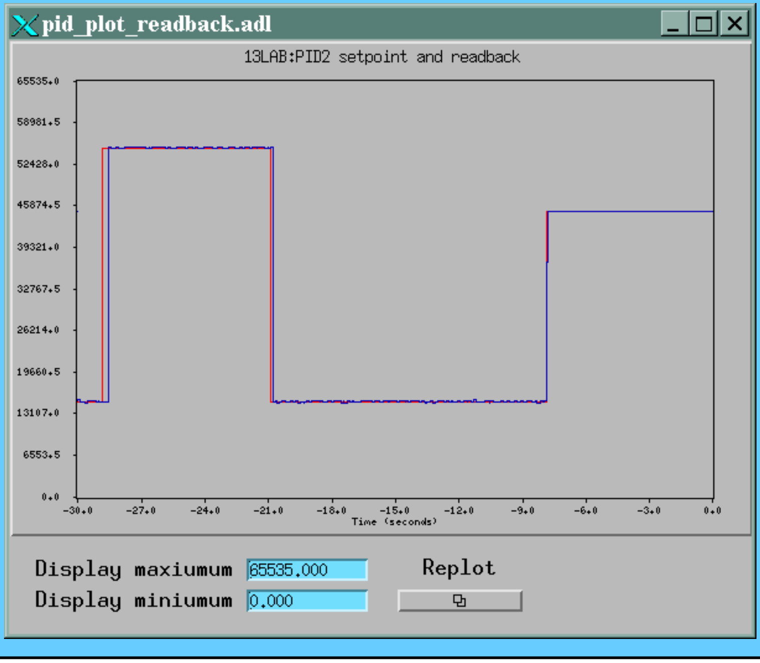
Delta time 0.001

Error 0.000

Output 2430.000

Low limit 1024.000

High limit 3072.000


pid_plot_readback.adl

13LAB:PID2 setpoint and readback

Display maxiumum 65535.000

Display miniumum 0.000

Replot

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.