asyn Training
EPICS Collaboration Meeting
April 2012

1. Overview of asyn
2. Overview of asynPortDriver
3. Writing an asyn driver Tutorial for Measurement Computing USB-1608GX-2A0

asyn: An Interface Between EPICS Drivers and Clients

Mark Rivers, Marty Kraimer, Eric Norum

University of Chicago

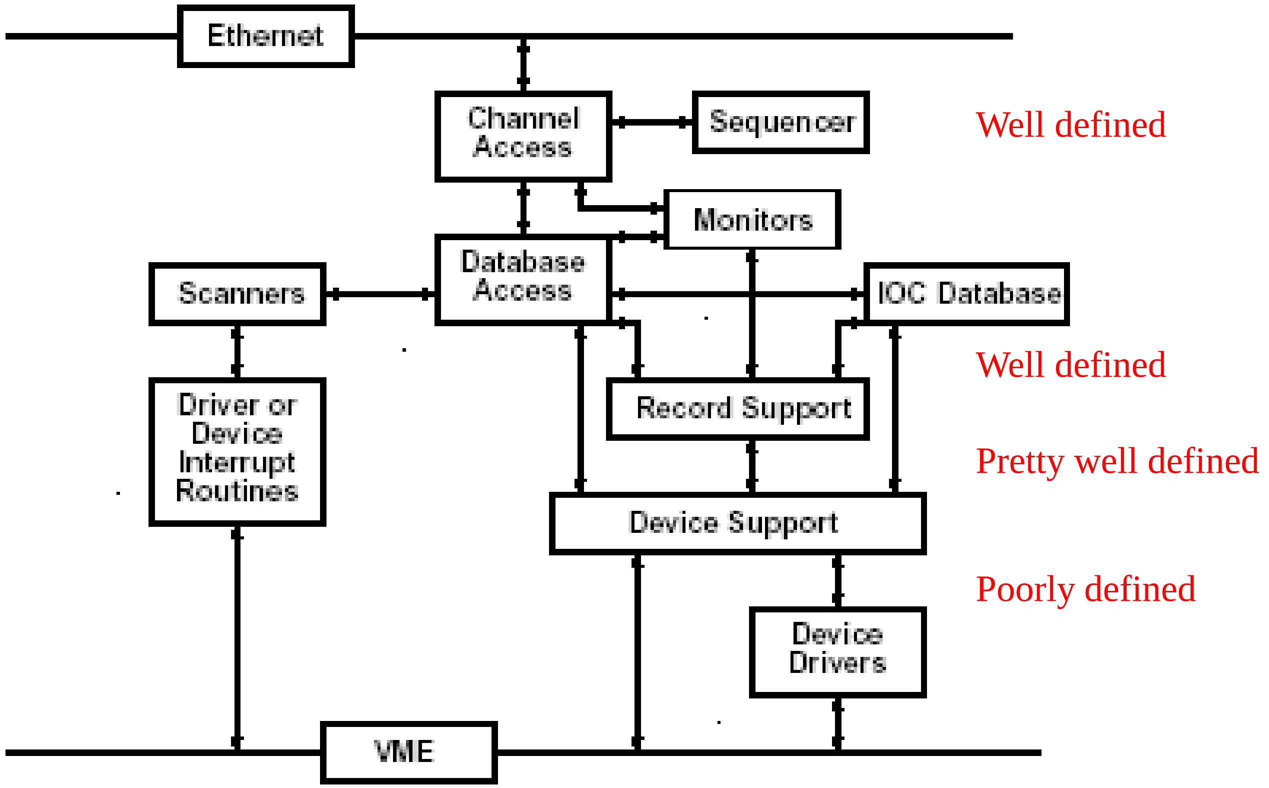
Advanced Photon Source

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



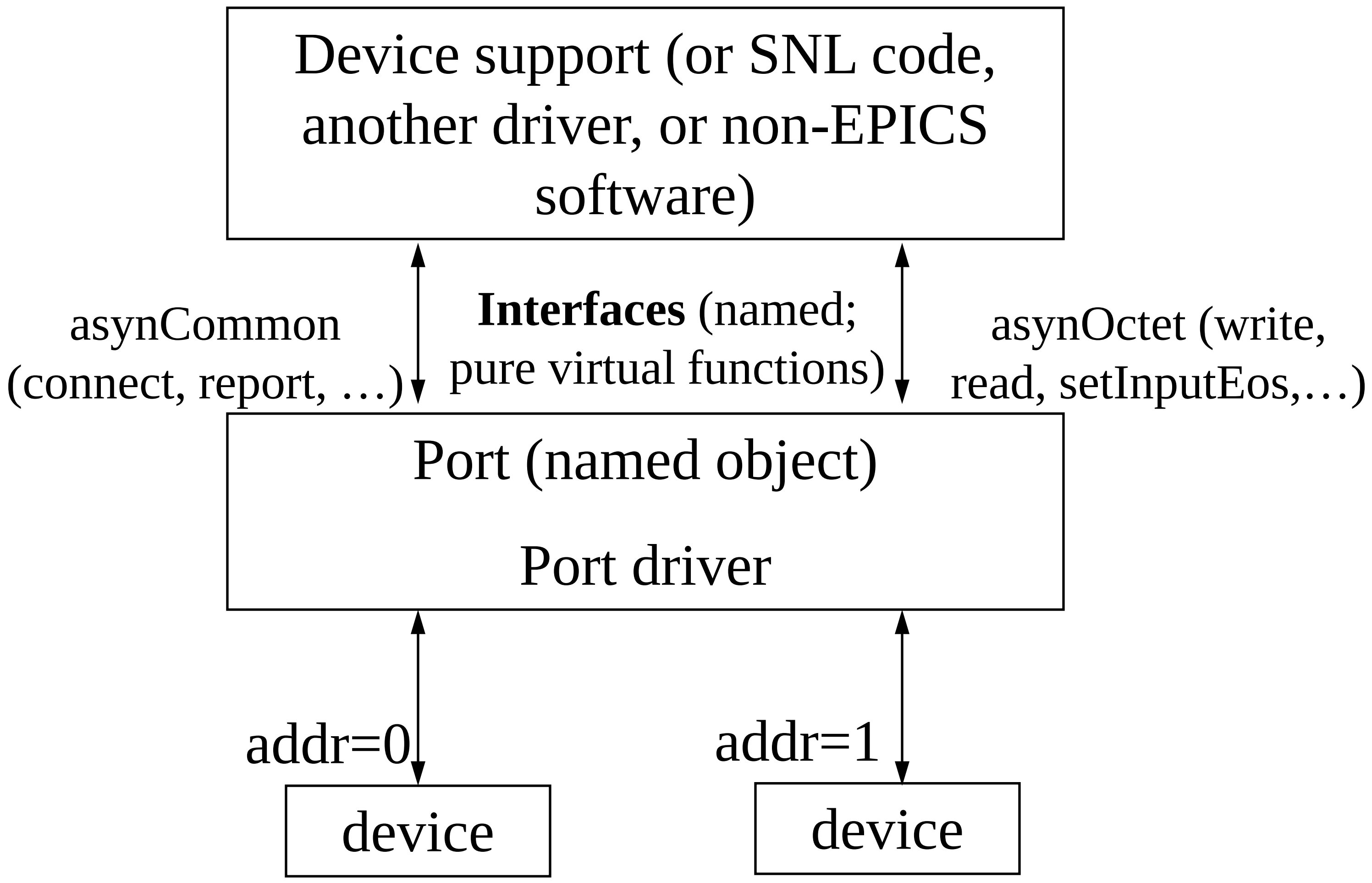
History – why the name asyn

- 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!

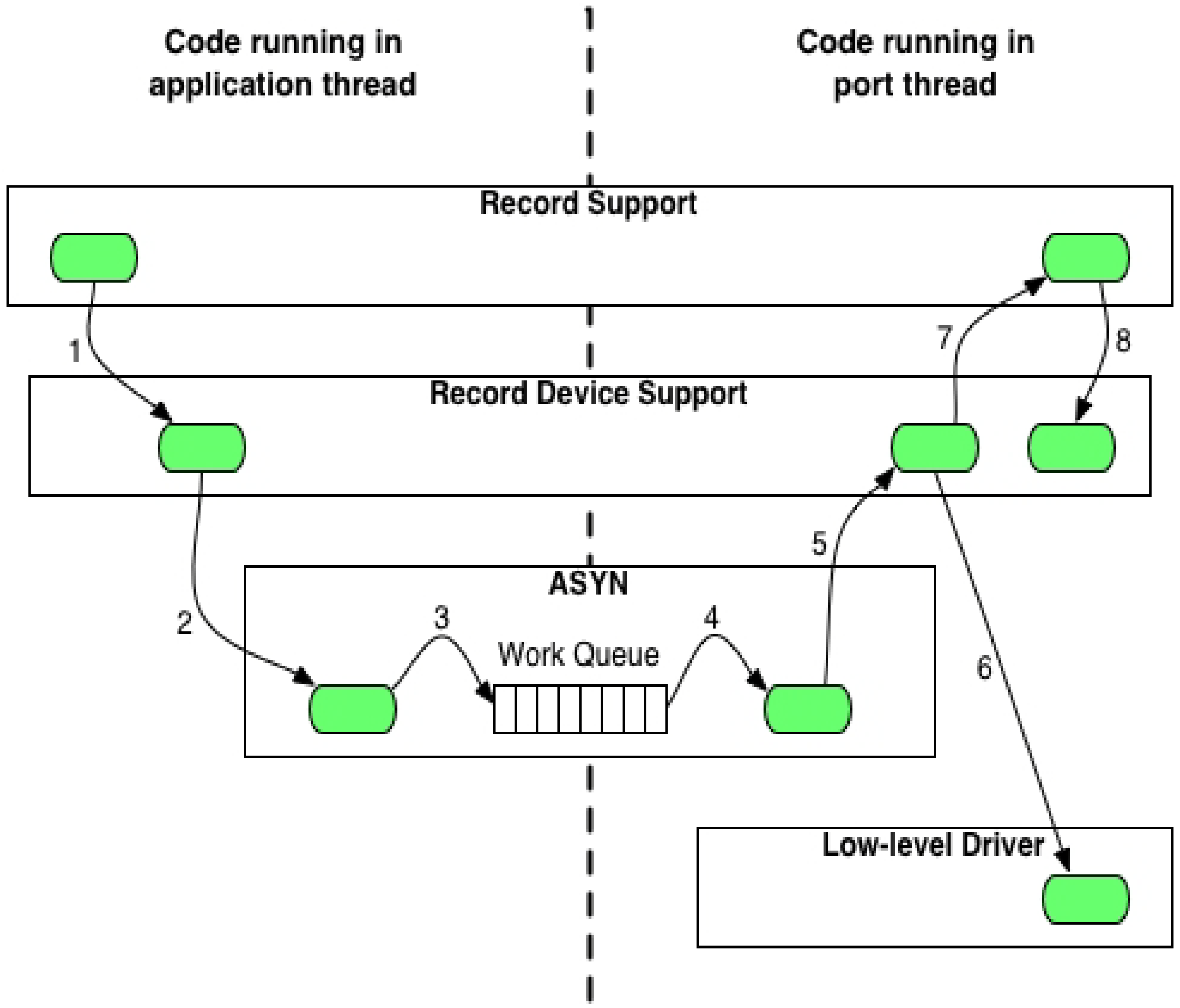
Independent of EPICS

- asyn is largely 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

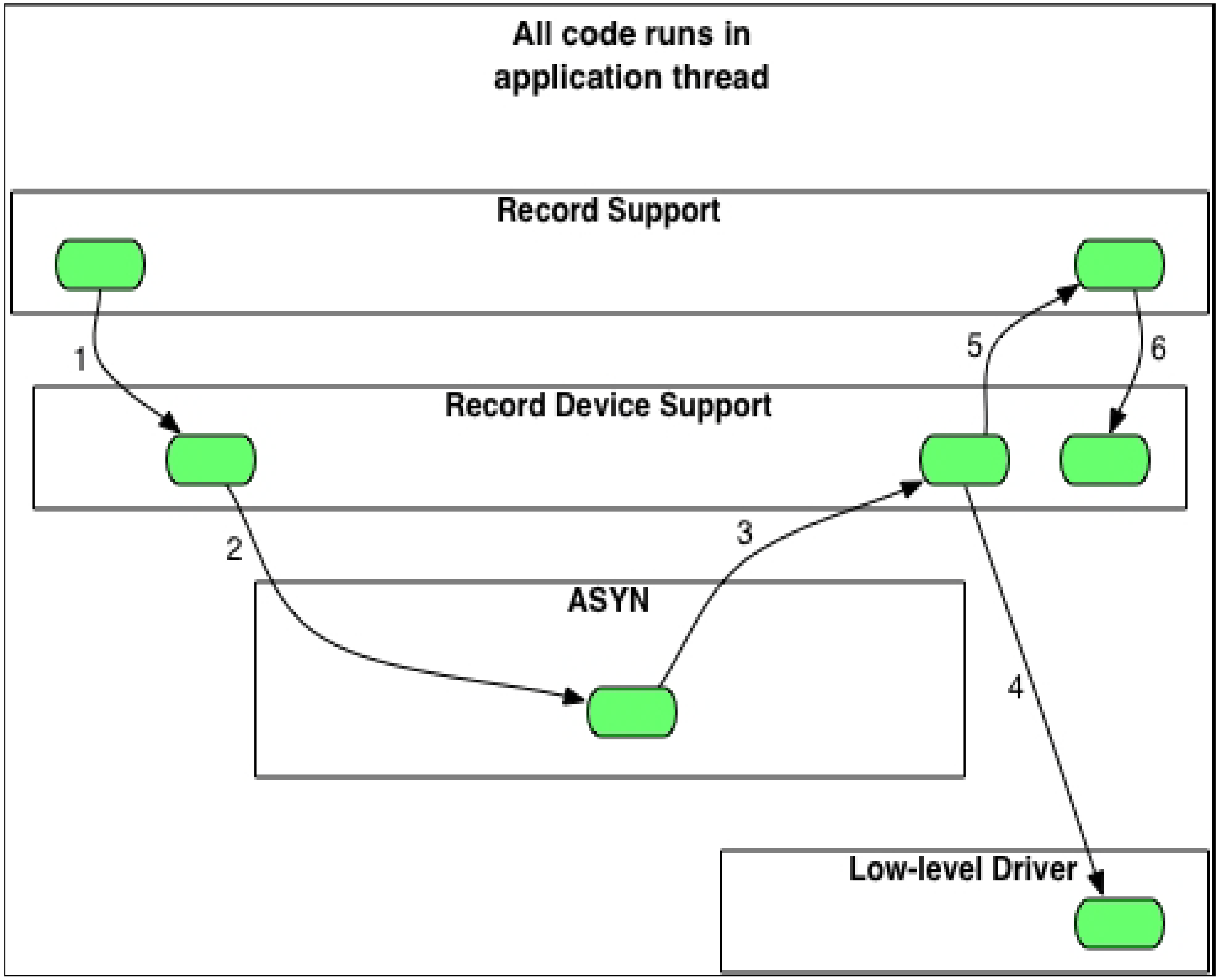
asyn Architecture (client/server)


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 (slow device)



Control flow – synchronous driver (fast device)



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:

```
record(longout, "$(P)$(R)BinX") {
   field(PINI, "YES")
   field(DTYP, "asynInt32")
   field(OUT,  "@asyn($(PORT),$(ADDR),$(TIMEOUT))BIN_X")
   field(VAL,  "1")
}

/* 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;
    /* reason is normally set by driver in drvUser->create()
     * to identify “parameter” */
    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:
- asynGenericPointer:

Miscellaneous interfaces

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

Developers are free to create new interfaces. In practice I don’t know of any!

asynStandardInterfaces.h

- Greatly reduces driver code when initializing standard interfaces. Just fill in fields in asynStandardInterfaces structure and call asynStandardInterfacesBase->initialize().

```
typedef struct asynStandardInterfaces {
    asynInterface common;
    asynInterface drvUser;
    asynInterface octet;
    int octetProcessEosIn;
    int octetProcessEosOut;
    int octetInterruptProcess;
    int octetCanInterrupt;
    void *octetInterruptPvt;
…
    asynInterface int32;
    int int32CanInterrupt;
    void *int32InterruptPvt;

    asynInterface float64Array;
    int float64ArrayCanInterrupt;
    void *float64ArrayInterruptPvt;

} asynStandardInterfaces;

typedef struct asynStandardInterfacesBase {
    asynStatus (*initialize)(const char *portName, asynStandardInterfaces
    *pInterfaces, asynUser *pasynUser, void *pPvt);
} asynStandardInterfacesBase;

epicsShareExtern asynStandardInterfacesBase *pasynStandardInterfacesBase;
```

Driver before asynStandardInterfaces

```
#include <asynDriver.h>
#include <asynInt32.h>
#include <asynInt8Array.h>
#include <asynInt16Array.h>
#include <asynInt32Array.h>
#include <asynFloat32Array.h>
#include <asynFloat64.h>
#include <asynFloat64Array.h>
#include <asynOctet.h>
#include <asynDrvUser.h>
...
typedef struct drvADPvt {
    ...
    /* Asyn interfaces */
    asynInterface common;
    asynInterface int32;
    void *int32InterruptPvt;
    asynInterface float64;
    void *float64InterruptPvt;
    asynInterface int8Array;
    void *int8ArrayInterruptPvt;
    asynInterface int16Array;
    void *int16ArrayInterruptPvt;
    asynInterface int32Array;
    void *int32ArrayInterruptPvt;
    asynInterface float32Array;
    void *float32ArrayInterruptPvt;
    asynInterface float64Array;
    void *float64ArrayInterruptPvt;
    asynInterface octet;
    void *octetInterruptPvt;
    asynInterface drvUser;
} drvADPvt;
```

Driver before asynStandardInterfaces

```
int drvADImageConfigure(const char *portName, const char
*detectorDriverName, int detector)
{
    ...
    /* Create asynUser for debugging */
    pPvt->pasynUser = pasynManager->createAsynUser(0, 0);

    /* Link with higher level routines */
    pPvt->common.interfaceType = asynCommonType;
    pPvt->common.pinterface  = (void *)&drvADCommon;
    pPvt->common.drvPvt = pPvt;
    pPvt->int32.interfaceType = asynInt32Type;
    pPvt->int32.pinterface  = (void *)&drvADInt32;
    pPvt->int32.drvPvt = pPvt;
    pPvt->float64.interfaceType = asynFloat64Type;
    pPvt->float64.pinterface  = (void *)&drvADFloat64;
    pPvt->float64.drvPvt = pPvt;
    pPvt->int8Array.interfaceType = asynInt8ArrayType;
    pPvt->int8Array.pinterface  = (void *)&drvADInt8Array;
    pPvt->int8Array.drvPvt = pPvt;
    pPvt->int16Array.interfaceType = asynInt16ArrayType;
    pPvt->int16Array.pinterface  = (void *)&drvADInt16Array;
    pPvt->int16Array.drvPvt = pPvt;
    pPvt->int32Array.interfaceType = asynInt32ArrayType;
    pPvt->int32Array.pinterface  = (void *)&drvADInt32Array;
    pPvt->int32Array.drvPvt = pPvt;
    pPvt->float32Array.interfaceType = asynFloat32ArrayType;
    pPvt->float32Array.pinterface  = (void *)&drvADFloat32Array;
    pPvt->float32Array.drvPvt = pPvt;
    pPvt->float64Array.interfaceType = asynFloat64ArrayType;
    pPvt->float64Array.pinterface  = (void *)&drvADFloat64Array;
    pPvt->float64Array.drvPvt = pPvt;
    pPvt->octet.interfaceType = asynOctetType;
    pPvt->octet.pinterface  = (void *)&drvADOctet;
    pPvt->octet.drvPvt = pPvt;
    pPvt->drvUser.interfaceType = asynDrvUserType;
    pPvt->drvUser.pinterface  = (void *)&drvADDrvUser;
    pPvt->drvUser.drvPvt = pPvt;

    ...
```

Driver before asynStandardInterfaces

```
status = pasynManager->registerInterface(portName,&pPvt->common);
if (status != asynSuccess) {
    errlogPrintf("drvAsynADConfigure ERROR: Can't register common.\n");
    return -1;
}

status = pasynInt32Base->initialize(pPvt->portName,&pPvt->int32);
if (status != asynSuccess) {
    errlogPrintf("drvAsynADConfigure ERROR: Can't register int32\n");
    return -1;
}
pasynManager->registerInterruptSource(portName, &pPvt->int32,
                                      &pPvt->int32InterruptPvt);

status = pasynFloat64Base->initialize(pPvt->portName,&pPvt->float64);
if (status != asynSuccess) {
    errlogPrintf("drvAsynADConfigure ERROR: Can't register float64\n");
    return -1;
}
pasynManager->registerInterruptSource(portName, &pPvt->float64,
                                      &pPvt->float64InterruptPvt);

status = pasynInt8ArrayBase->initialize(pPvt->portName,&pPvt->int8Array);
if (status != asynSuccess) {
    errlogPrintf("drvAsynADConfigure ERROR: Can't register int8Array\n");
    return -1;
}

pasynManager->registerInterruptSource(portName, &pPvt->int8Array,
                                      &pPvt->int8ArrayInterruptPvt);

status = pasynInt16ArrayBase->initialize(pPvt->portName,&pPvt->int16Array);
if (status != asynSuccess) {
    errlogPrintf("drvAsynADConfigure ERROR: Can't register int16Array\n");
    return -1;
}

pasynManager->registerInterruptSource(portName, &pPvt->int16Array, &pPvt->int16ArrayInterruptPvt);
```

Driver before asynStandardInterfaces

```
status = pasynInt32ArrayBase->initialize(pPvt->portName,&pPvt->int32Array);
    if (status != asynSuccess) {
        errlogPrintf("drvAsynADConfigure ERROR: Can't register int32Array\n");
        return -1;
    }

    pasynManager->registerInterruptSource(portName, &pPvt->int32Array, &pPvt->int32ArrayInterruptPvt);

    status = pasynFloat32ArrayBase->initialize(pPvt->portName,&pPvt->float32Array);
    if (status != asynSuccess) {
        errlogPrintf("drvAsynADConfigure ERROR: Can't register float32Array\n");
        return -1;
    }

    pasynManager->registerInterruptSource(portName, &pPvt->float32Array, &pPvt->float32ArrayInterruptPvt);

    status = pasynFloat64ArrayBase->initialize(pPvt->portName,&pPvt->float64Array);
    if (status != asynSuccess) {
        errlogPrintf("drvAsynADConfigure ERROR: Can't register float64Array\n");
        return -1;
    }

    pasynManager->registerInterruptSource(portName, &pPvt->float64Array, &pPvt->float64ArrayInterruptPvt);

    status = pasynOctetBase->initialize(pPvt->portName,&pPvt->octet,0,0,0);
    if (status != asynSuccess) {
        errlogPrintf("drvAsynADConfigure ERROR: Can't register octet\n");
        return -1;
    }

    pasynManager->registerInterruptSource(portName, &pPvt->octet,
                                          &pPvt->octetInterruptPvt);

    status = pasynManager->registerInterface(pPvt->portName,&pPvt->drvUser);
    if (status != asynSuccess) {
        errlogPrintf("drvAsynADConfigure ERROR: Can't register drvUser\n");
        return -1;
    }
```

Driver after asynStandardInterfaces

```
#include <asynStandardInterfaces.h>
...
typedef struct drvADPvt {
    ...
    /* Asyn interfaces */
    asynStandardInterfaces asynInterfaces;
} drvADPvt;

int drvADImageConfigure(const char *portName, const char *detectorDriverName, int detector)
{
     ...
     asynStandardInterfaces *pInterfaces;
     ...
    /* Create asynUser for debugging and for standardBases */
    pPvt->pasynUser = pasynManager->createAsynUser(0, 0);

    /* Set addresses of asyn interfaces */
    pInterfaces = &pPvt->asynInterfaces;

    pInterfaces->common.pinterface        = (void *)&drvADCommon;
    pInterfaces->drvUser.pinterface       = (void *)&drvADDrvUser;
    pInterfaces->octet.pinterface         = (void *)&drvADOctet;
    pInterfaces->int32.pinterface         = (void *)&drvADInt32;
    pInterfaces->float64.pinterface       = (void *)&drvADFloat64;
    pInterfaces->int8Array.pinterface     = (void *)&drvADInt8Array;
    pInterfaces->int16Array.pinterface    = (void *)&drvADInt16Array;
    pInterfaces->int32Array.pinterface    = (void *)&drvADInt32Array;
    pInterfaces->float32Array.pinterface  = (void *)&drvADFloat32Array;
    pInterfaces->float64Array.pinterface  = (void *)&drvADFloat64Array;

    /* Define which interfaces can generate interrupts */
    pInterfaces->octetCanInterrupt        = 1;
    pInterfaces->int32CanInterrupt        = 1;
    pInterfaces->float64CanInterrupt      = 1;
    pInterfaces->int8ArrayCanInterrupt    = 1;
    pInterfaces->int16ArrayCanInterrupt   = 1;
    pInterfaces->int32ArrayCanInterrupt   = 1;
    pInterfaces->float32ArrayCanInterrupt = 1;
    pInterfaces->float64ArrayCanInterrupt = 1;

    status = pasynStandardInterfacesBase->initialize(portName, pInterfaces,  pPvt->pasynUser, pPvt);
    if (status != asynSuccess) {
        errlogPrintf("drvADImageConfigure ERROR: Can't register interfaces: %s.\n",
                     pPvt->pasynUser->errorMessage);
        return -1;
    }
```

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 a parameter index value (e.g. mcaElapsedLive, mcaErase, etc.)
- More complex driver could set pasynUser->drvUser to a pointer to something.
- Example

```
record(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), $(ADDR), $(TIMEOUT))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 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) with new data
  - 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.
- Example interrupt drivers
  - Ip330 ADC, IpUnidig binary I/O, SIS 38XX MCS, areaDetector drivers, etc.
  - Measurement Computing devices we will study and demo later this morning

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 virtually all 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 (in and out)
  - asynXXXArray
    - waveform (in and out)

Generic Device Support

- All synApps modules I am responsible for now use standard asyn device support, and no longer have specialized device support code:
  - areaDetector 2D detectors
  - Modbus General Modbus driver
  - dxp (XIA Saturn, XMAP, Mercury spectroscopy systems)
  - Ip330 16-channel 16-bit Industry Pack ADC
  - IpUnidig 24 bit Industry Pack digital I/O module
  - quadEM APS VME quad electrometer
  - dac128V 8 channel 12-bit Industry Pack DAC
  - measComp Measurement Computing USB modules (*more later*)
  - Canberra ICB modules (Amp, ADC, HVPS, TCA)
  - SIS 38XX multichannel scalers
  - Motors: Newport XPS, Parker Aries, ACS MCB-4B

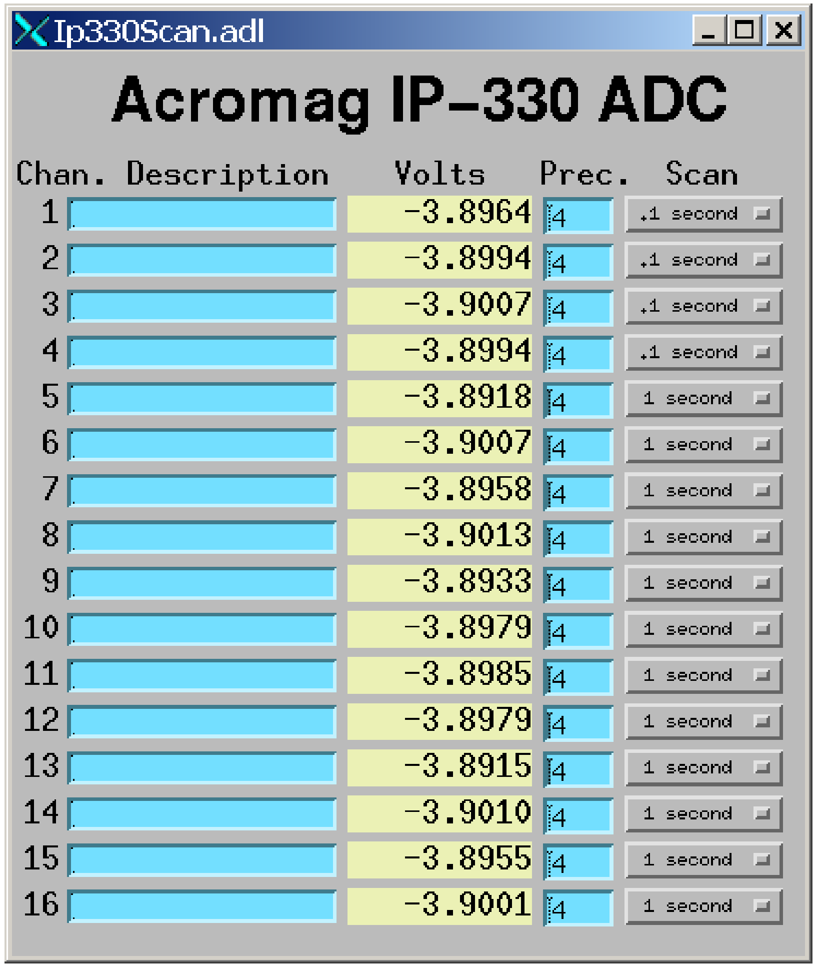
Generic Device Support

- MCA and motor records use special device support, because they are not base record types.
  - *Single device-independent device support file*
  - Only driver is device-dependent
- MCA and new motor drivers now only use the standard asyn interfaces, so it is possible (in principle) to write a database using only standard records and control any MCA driver or new motor driver

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")
}
```

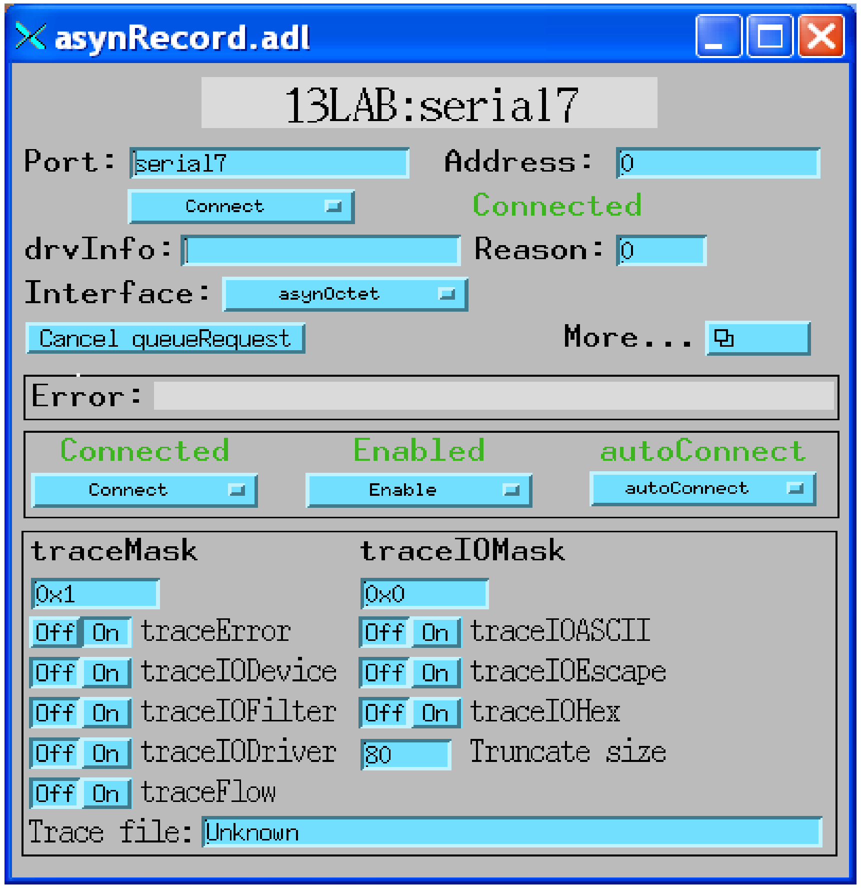

Ip330Scan.adl

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

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



asynRecord.adl

13LAB:serial7

Port: serial7 Address: 0

Connect Connected

drvInfo: Reason: 0

Interface: asynOctet

Cancel queueRequest More...

Error:

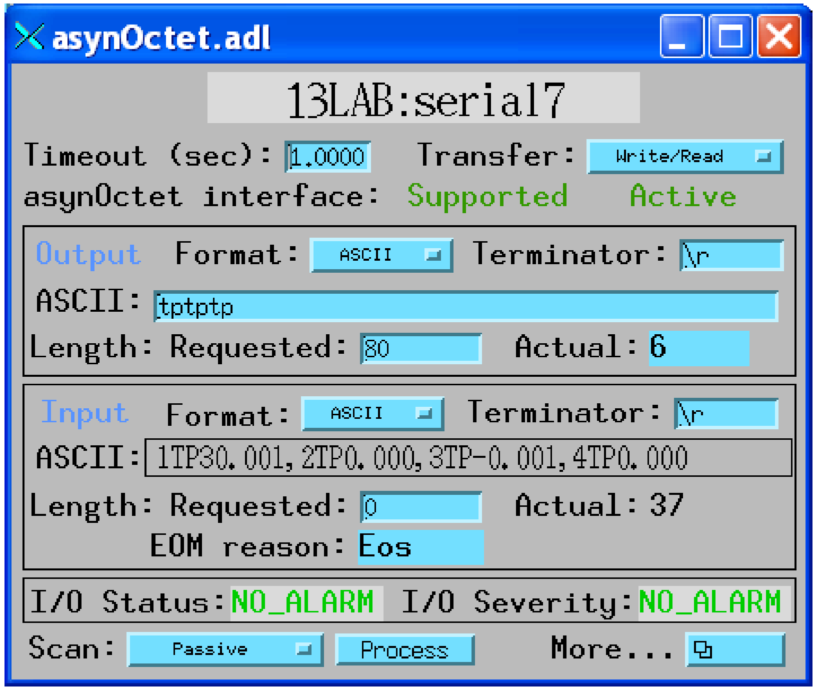
Connected	Enabled	autoConnect
Connect	Enable	autoConnect

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

asynRecord – asynOctet devices

Interactive I/O to octet devices (serial, TCP/IP, GPIB, etc.)



asynOctet.adl

13LAB:serial7

Timeout (sec): 1.0000 Transfer: Write/Read
asynOctet interface: Supported Active

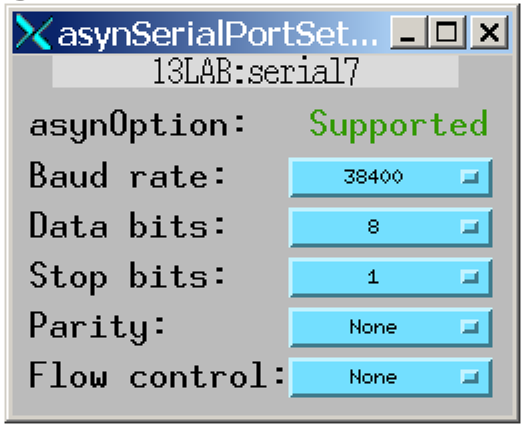
Output Format: ASCII Terminator: \r
ASCII: tptptp
Length: Requested: 80 Actual: 6

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

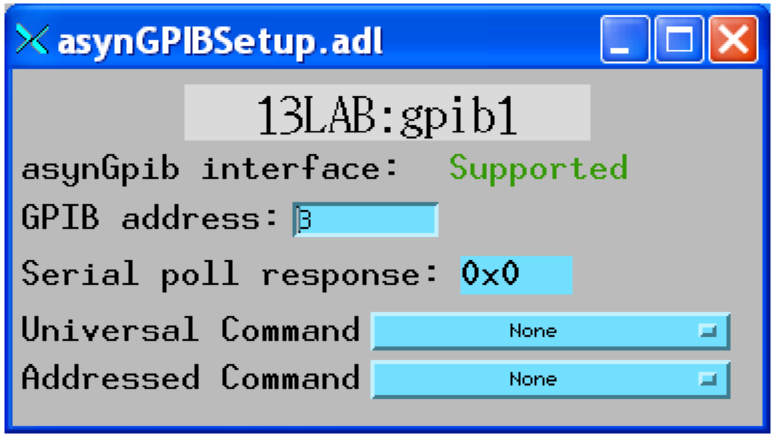


asynSerialPortSet...

13LAB:serial7

asynOption:	Supported
Baud rate:	38400
Data bits:	8
Stop bits:	1
Parity:	None
Flow control:	None

Perform GPIB specific operations



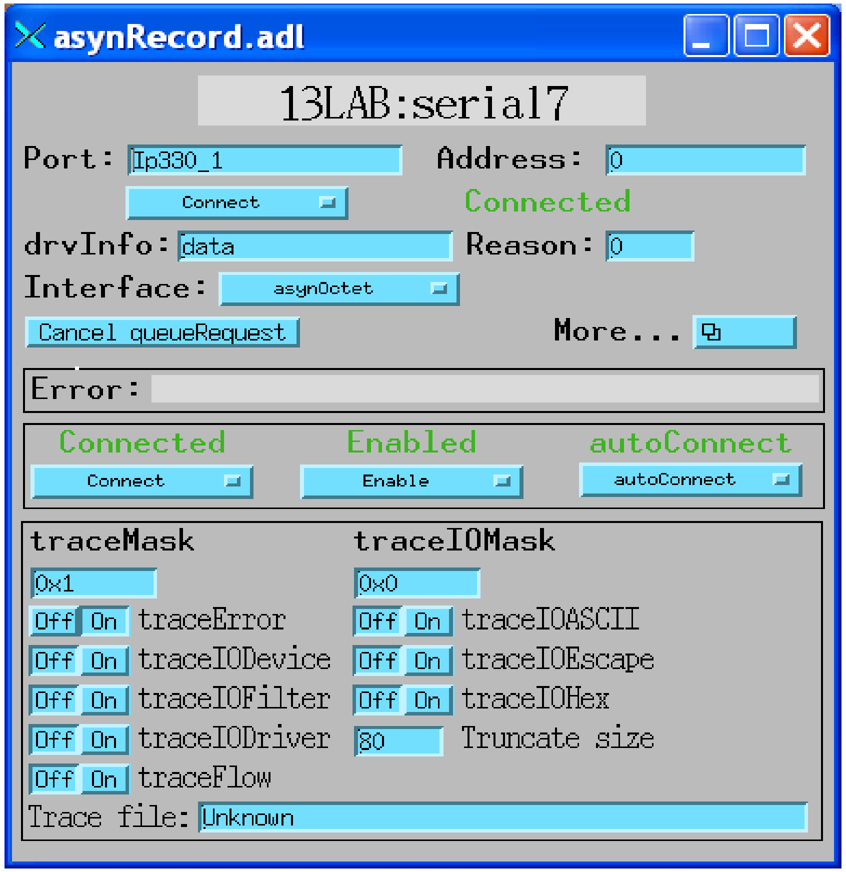
asynGPIBSetup.adl

13LAB:gpib1

asynGpib interface:	Supported
GPIB address:	3
Serial poll response:	0x0
Universal Command	None
Addressed Command	None

asynRecord – 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...

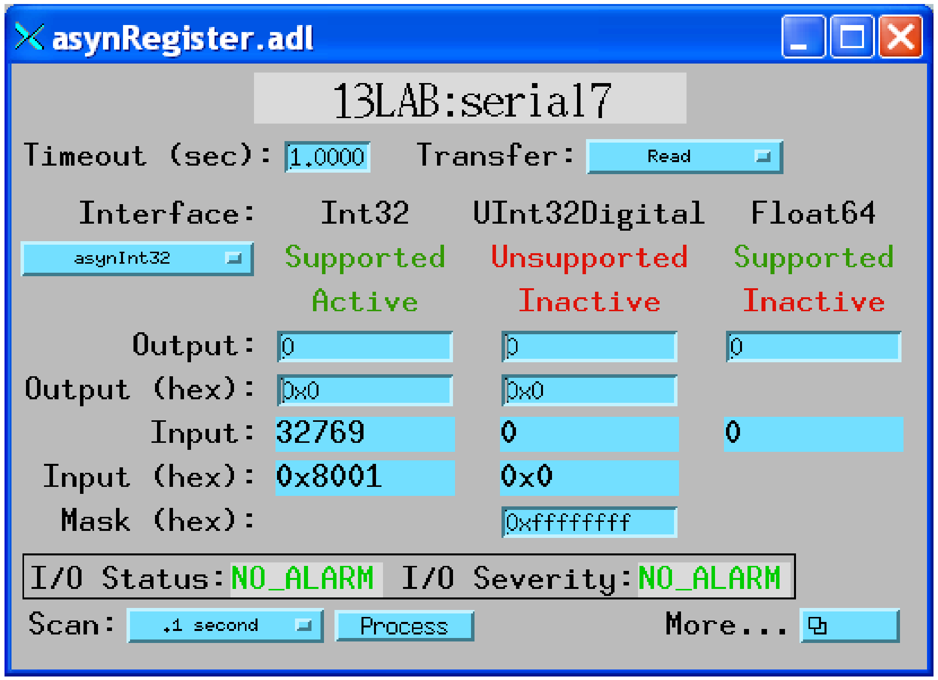
Error:

Connected	Enabled	autoConnect
Connect	Enable	autoConnect

traceMask		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

Read ADC at 10Hz with asynInt32 interface



asynRegister.adl

13LAB:serial7

Timeout (sec): 1.0000 Transfer: Read

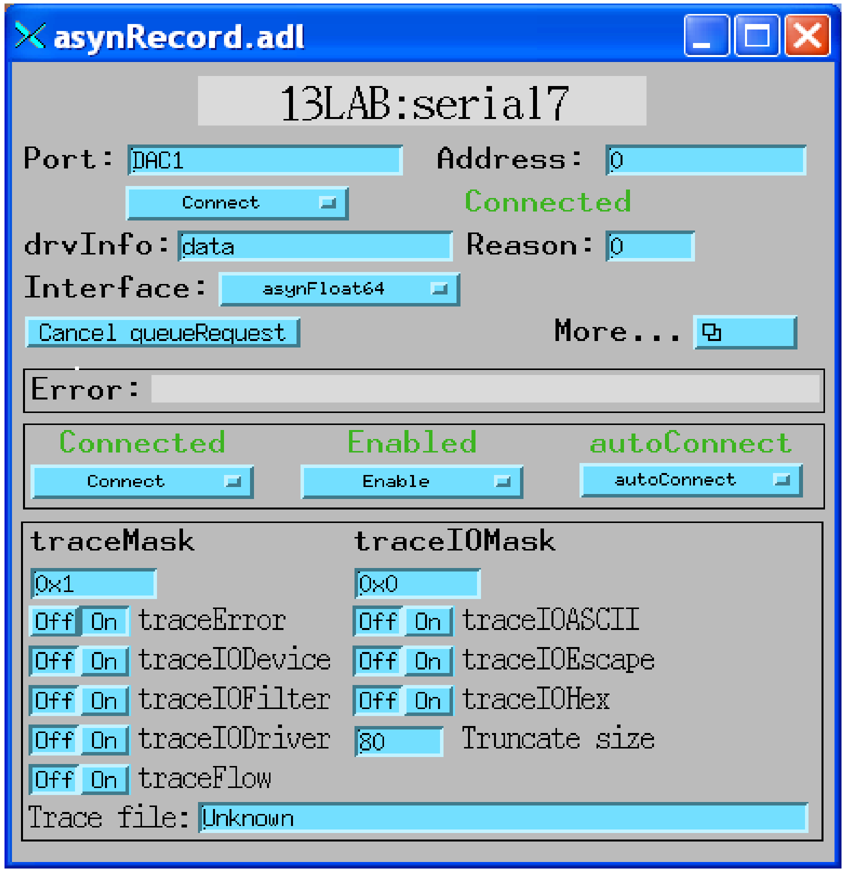
Interface:	Int32	UInt32Digital	Float64
asynInt32	Supported	Unsupported	Supported
	Active	Inactive	Inactive
Output:	0	0	0
Output (hex):	0x0	0x0	
Input:	32769	0	0
Input (hex):	0x8001	0x0	
Mask (hex):		0xffffffff	

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

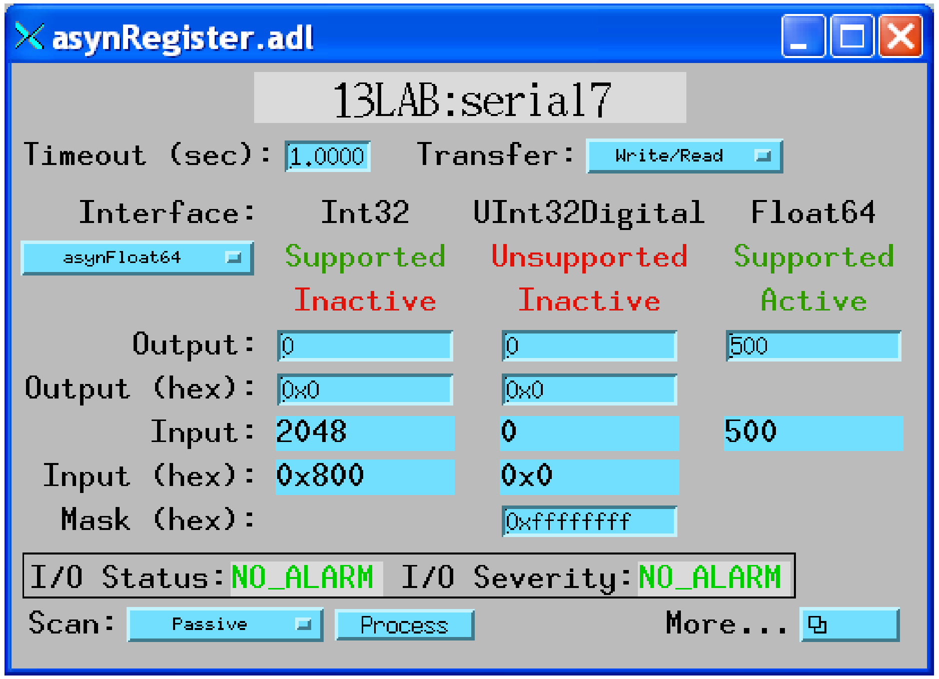
traceMask: 0x1

traceIOMask: 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...

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)
  - areaDetector driver talking to marCCD server, Pilatus camserver, etc.
  - iocsh commands

Synchronous interfaces – Tektronix scope driver example

- In initialization:

```
/* Connect to scope */
 status = pasynOctetSyncIO->connect(scopeVXI11Name,
                                    0,
                                    &this->pasynUserScope,
                                    NULL);
```

- In IO:

```
status = pasynOctetSyncIO->writeRead(pasynUserScope,
                                     sendMessage, numSend,
                                     receiveMessage,
                                     sizeof(receiveMessage),
                                     WRITE_READ_TIMEOUT,
                                     &numSent, responseLen,
                                     &eomReason);
```

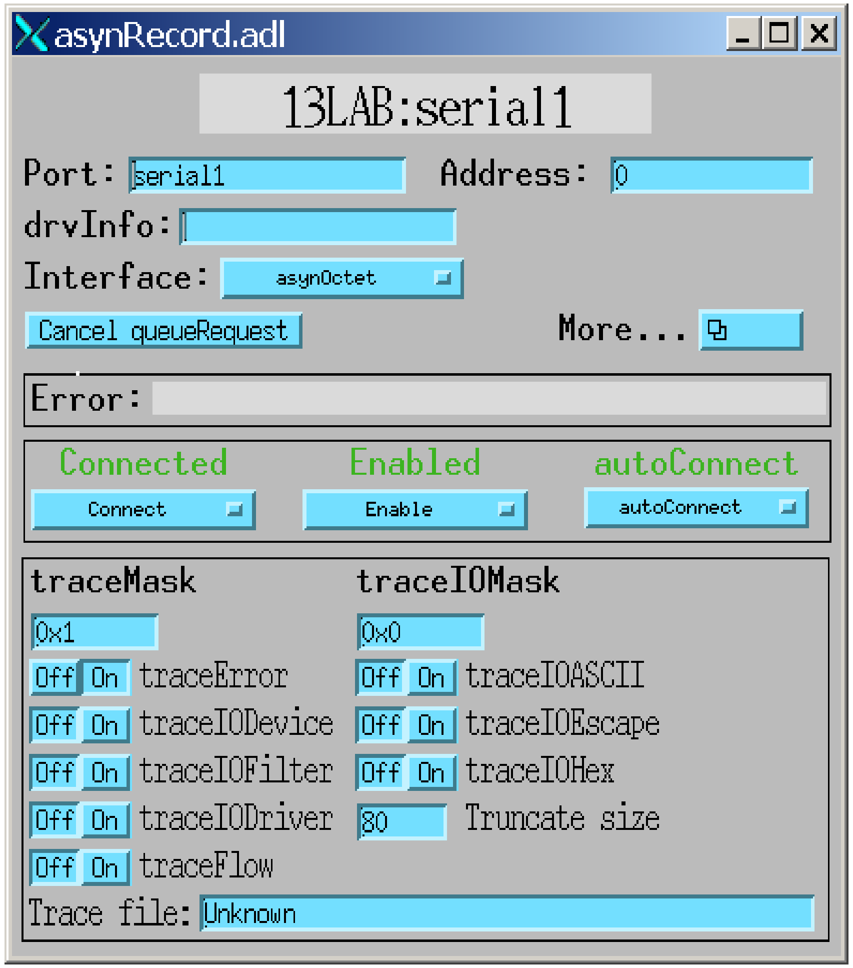
iocsh Commands

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

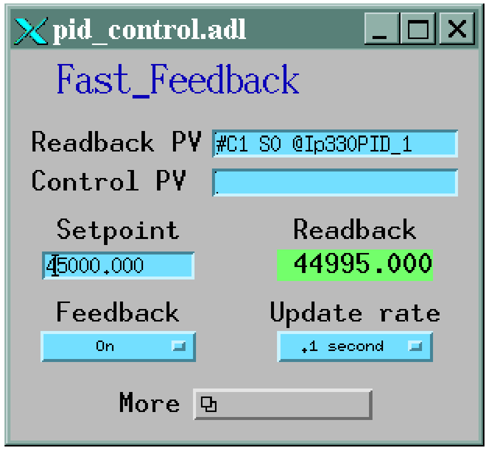
Off On traceIODriver 80 Truncate size

Off On traceFlow

Trace file: Unknown

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:

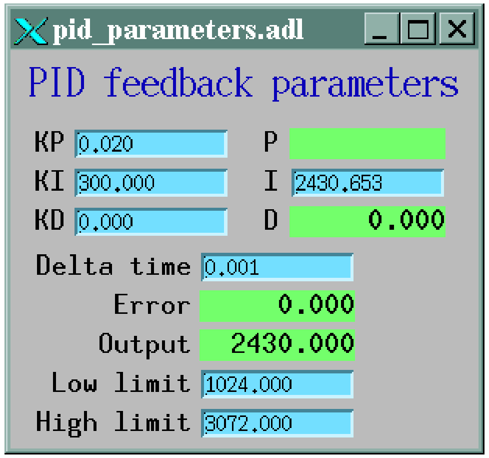
Setpoint: 45000.000

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

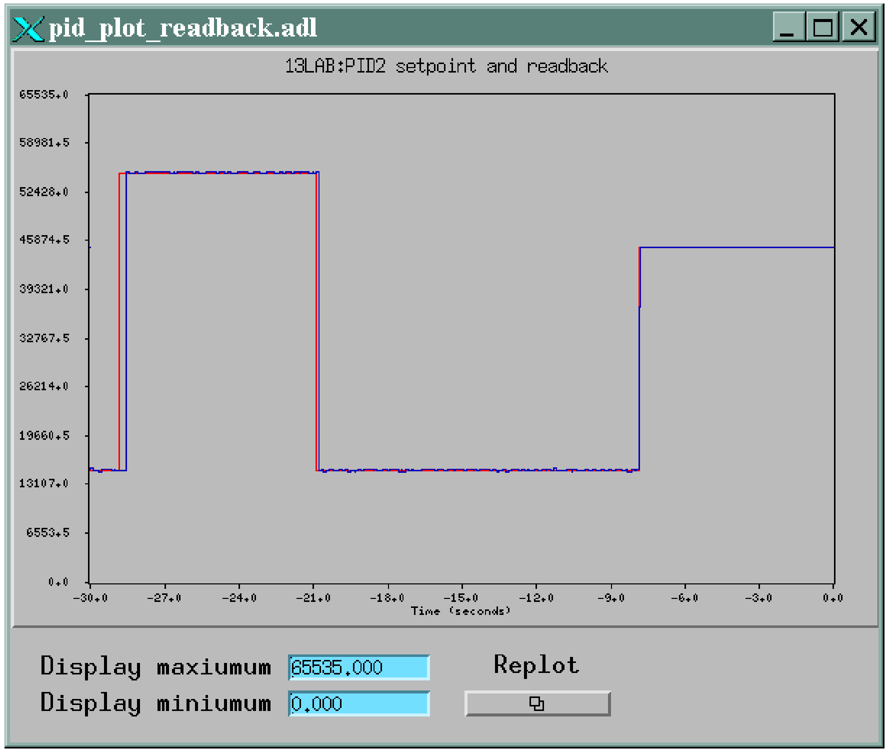
Delta time: 0.001

Error: 0.000

Output: 2430.000

Low limit: 1024.000

High limit: 3072.000


pid_plot_readback.adl

13LAB:PID2 setpoint and readback

Display maxiumum: 65535.000

Display miniumum: 0.000

Replot

asyn: Drivers Included in the Module

- drvAsynSerialPort
  - Driver for serial ports on most OS (Linux, Windows, Darwin, vxWorks, RTEMS, etc.)
  - drvAsynSerialPortConfigure("portName", "ttyName", priority, noAutoConnect, noProcessEosIn)
  - asynSetOption("portName", addr, "key", "value")

Key	**Value**
baud	9600 50 75 110 134 150 200 300 600 1200 ... 230400
bits	8 7 6 5
parity	none even odd
stop	1 2
clocal	Y N
crtscts	N Y
ixon	N Y
ixoff	N Y
ixany	N Y

asyn: Drivers Included in the Module

drvAsynIPPort – driver for IP network communications

drvAsynIPPortConfigure("portName", "hostInfo", priority, noAutoConnect, noProcessEos)

- hostInfo - The Internet host name, port number and optional IP protocol of the device (e.g. "164.54.9.90:4002", "serials8n3:4002", "serials8n3:4002 TCP" or "164.54.17.43:5186 udp").
- Protocols:
    - TCP (default)
    - UDP
    - UDP* — UDP broadcasts. The address portion of the argument must be the network broadcast address (e.g. "192.168.1.255:1234 UDP*").
    - HTTP — Like TCP but for servers which close the connection after each transaction.
    - COM — For Ethernet/Serial adapters which use the TELNET RFC 2217 protocol. This allows port parameters (speed, parity, etc.) to be set with subsequent asynSetOption commands just as for local serial ports. The default parameters are 9600-8-N-1 with no flow control.

Where does an output record (ao, longout, mbbo, etc.) get its initial value?

1. From the aoRecord.dbd file
2. From the database file you load
3. asyn device support init_record function does a read from your asyn driver to read the current value.
   - If your driver returns asynSuccess on that read then that value is used.
   - Supports bumpless reboots.
   - If using asynPort driver then if your driver has done setIntegerParam, setDoubleParam, or setStringParam before init_record (i.e. typically in constructor) then the readInt32, etc. functions will return asynSuccess. If setInt32Param has not been done then readInt32 returns asynError, and that value will not be used.
4. From save/restore
5. dbpf at the end of startup script.

asyn: Problems and Future Work

- **No support for driver providing enum strings to device support.**
  - Currently enum strings must be hardcoded in the database.
  - Sometimes they should be generated by the driver, depending on what model device is connected.
  - They can even change dynamically: the list of valid gains can change when the readout speed changes.
  - Need to add new asynEnumString interface, and it should support callbacks
- **No support for passing status information in the asyn callback functions.**
  - This means that records with SCAN=I/O Intr will not go into alarm status when the driver detects an error.
  - Subject of multiple tech-talk requests
  - Seemed quite difficult to fix, because changing the interface to add status to the callback function would break all existing asyn drivers that do callbacks.
  - But I just realized on the plane ride out that pasynUser->auxStatus could be used for this!
    - It would almost always be 0 now (=asynSuccess) so using non-zero value (e.g. asynError, asynTimeout) is unlikely to break existing code.

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 99% (?) 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.