

EPICS asyn Training Classes

January 10-12, 2018

- Wednesday morning session
 - Overview of asyn
 - Overview of asynPortDriver
- Wednesday afternoon session
 - Writing an asyn driver: Tutorial for Measurement Computing USB-1608GX-2A0
 - Hands-on driver development
- Thursday morning session
 - Overview of areaDetector
 - Demo of areaDetector with simDetector
- Thursday afternoon & Friday morning sessions
 - Hands-on driver development

All talks can be downloaded from

http://cars.uchicago.edu/data/Rivers/NSLS-II_asyn_class

Useful Links for EPICS Documentation

EPICS Application Developer's Guide

<https://epics.anl.gov/base/R3-15/5-docs/AppDevGuide/AppDevGuide.html>

EPICS Record Reference Manual

https://wiki-ext.aps.anl.gov/epics/index.php/RRM_3-14

Asyn driver documentation:

<https://epics.anl.gov/modules/soft/asyn/R4-32/asynDriver.html>

Or

<http://htmlpreview.github.io/?https://github.com/epics-modules/asyn/blob/master/documentation/asynDriver.html>

EPICS training videos 2014-2015

<https://epics.anl.gov/docs/training.php>

Useful Links for EPICS Documentation

State Notation Language documentation:

<http://www-csr.bessy.de/control/SoftDist/sequencer/>

areaDetector documentation

<http://cars.uchicago.edu/software/epics/areaDetectorDoc.html>

StreamDevice documentation:

<http://epics.web.psi.ch/software/streamdevice/doc/>

asyn: An Interface Between EPICS Drivers and Clients

Mark Rivers, Eric Norum, Marty Kraimer

University of Chicago

Lawrence Berkeley Laboratory

Brookhaven National Laboratory

How to deal with a new device

(My philosophy!)

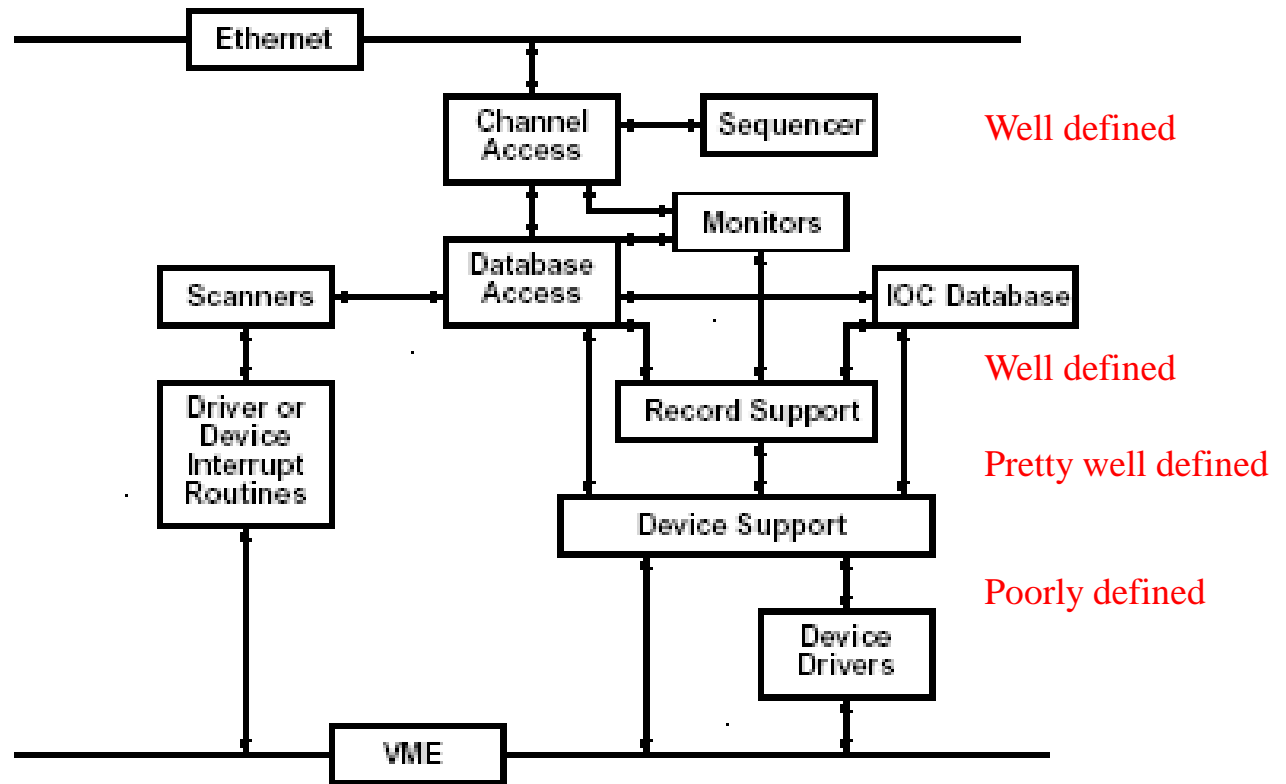
- If the device uses strings to communicate, and is not too complex, use streamDevice
 - Works well for relatively simple devices
 - Difficult to deal with more complex devices where parameters interact, since there is no “state” information
 - Uses asyn at the low-level (serial, TCP, GPIB), so it helps to understand asyn
- If the device does not use strings, or is complex, then write an asyn port driver (preferably using asynPortDriver C++ base class)
- Should not need to write device support for standard records
 - 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, call directly from SNL, or another control system altogether, etc.

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



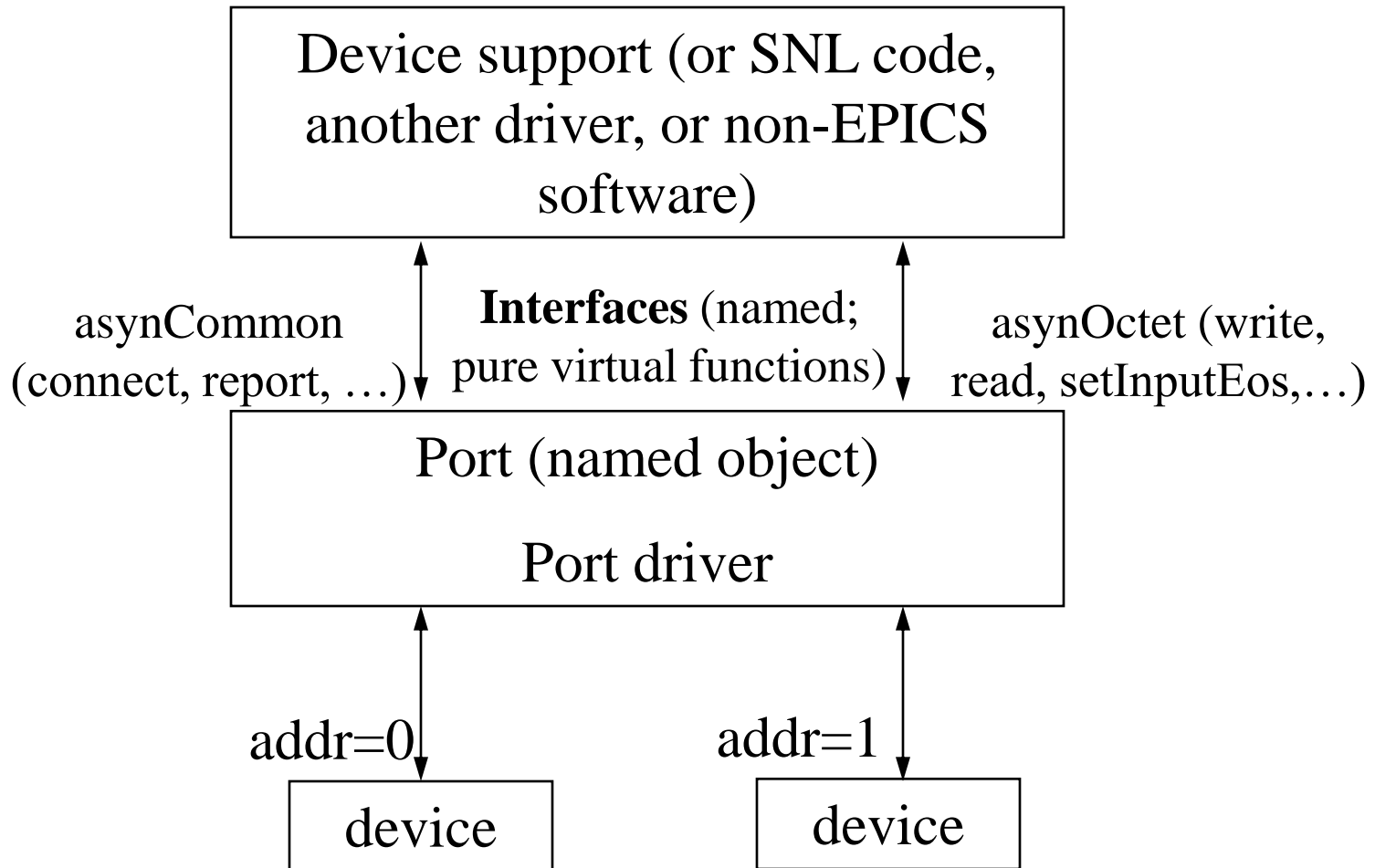
History – why the name asyn

- 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!

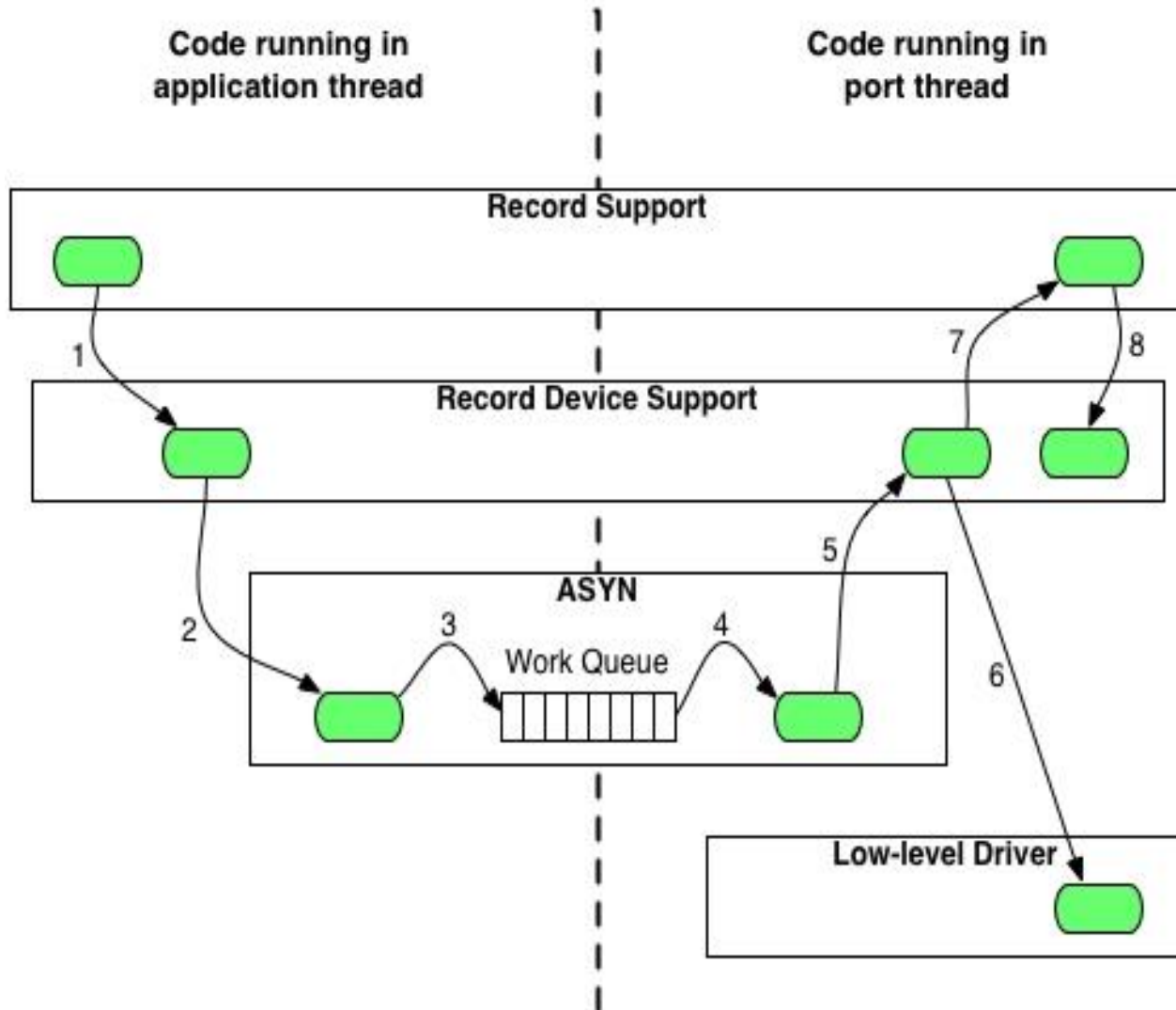
Independent of EPICS

- 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)

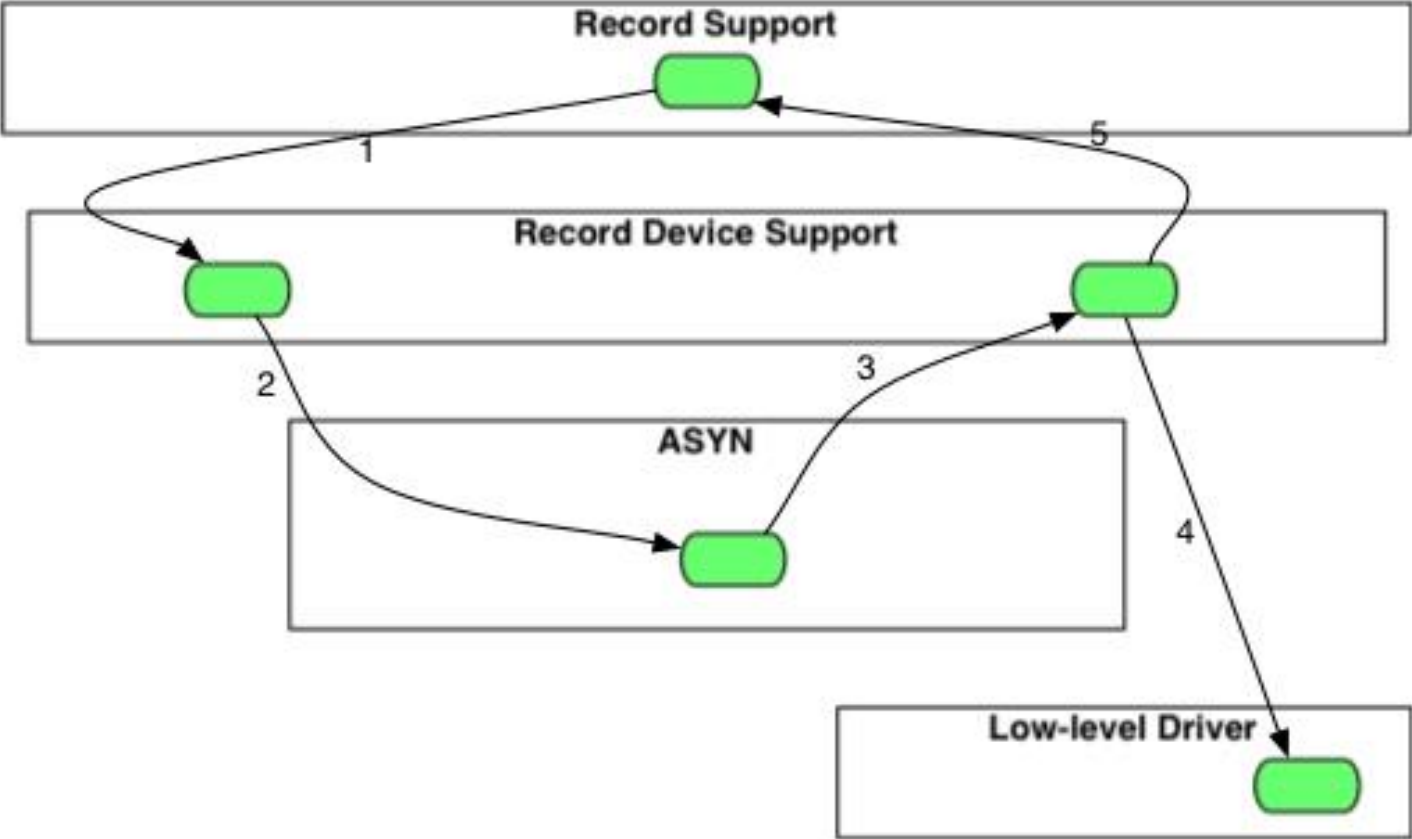


Control flow – asynchronous driver (slow device)



Control flow – synchronous driver (fast device)

All code runs in application thread



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 (synchronous driver, asynInt32 interface, can do callbacks):

```
pPvt->int32.interfaceType = asynInt32ArrayType;
pPvt->int32.pinterface = (void *)&drvIp330Int32Array;
pPvt->int32.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;  
    /* timeout 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;  
    epicsTimeStamp timestamp;  
    /* The following are for additional information from method calls */  
    int     auxStatus; /*For auxillary status*/  
    int     alarmStatus; /* Typically for EPICS record alarm status */  
    int     alarmSeverity; /* Typically for EPICS record alarm severity */  
}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(), registerInterruptUser() and cancelInterruptUser() methods
- asynOctet: flush(), setInputEos(), setOutputEos(), getInputEos(), getOutputEos()
- asynInt32: getBounds()
- asynInt8Array, asynInt16Array, asynInt32Array:
- asynUInt32Digital: setInterrupt(), clearInterrupt()
- asynFloat64:
- asynFloat32Array, asynFloat64Array:
- asynGenericPointer:
- asynEnum

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! asynGenericPointer can be used to pass any structure when other interfaces are too simple.

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 without 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 without 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 without 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 with 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`, `asynXXXArray`, `asynGenericPointer`, and `asynEnum` 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 later today

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)
 - asynEnum
 - bi, bo, mbbi, mbbo

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, NSLS and CaenEls/Elettra quad electrometers
 - 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

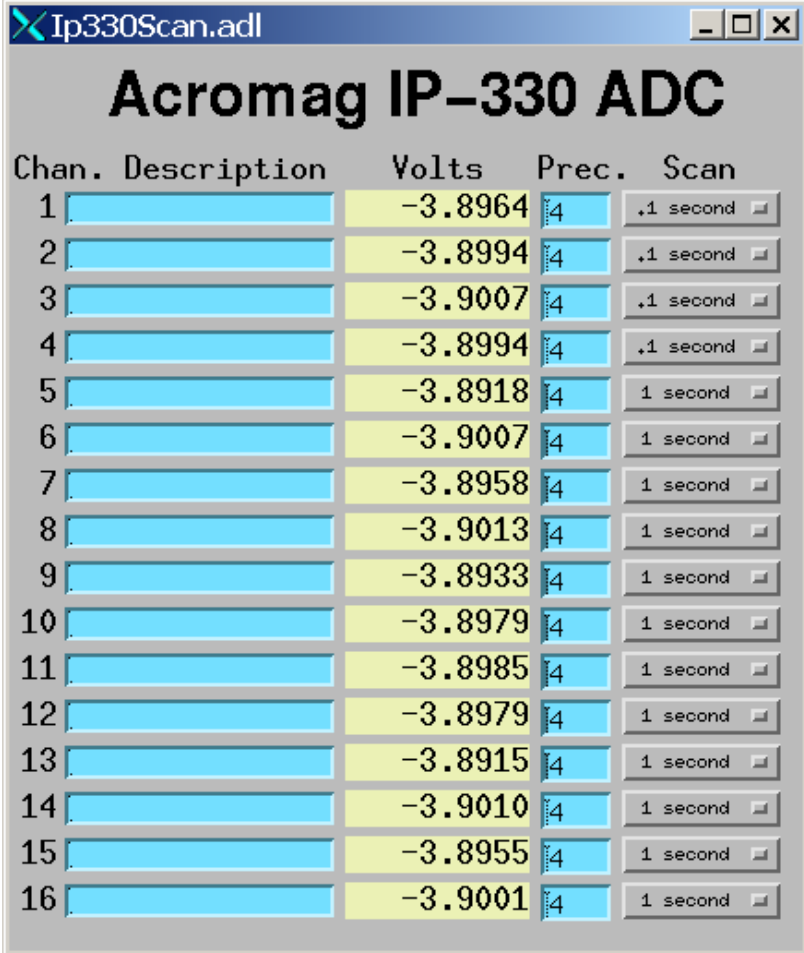
Generic Device Support

- MCA, scaler 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, scaler 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
 - However, the state logic would need to be moved from the record into a driver base class

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")
}
```



The screenshot shows a window titled "Ip330Scan.adl" with a table of ADC data. The table has four columns: "Chan.", "Description", "Volts", and "Prec. Scan". There are 16 rows of data, each with a channel number, a description, a voltage value, a precision of 4, and a scan rate of 0.1 second.

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

Generic Device Support

Ring buffers

- Ring buffers
- Input records can receive driver callbacks faster than the record can process. They can use a ring buffer to store the callback values, allowing the record to process multiple times to process all callbacks up to the ring buffer size.
- Default ring buffer size is 10, except for stringin, stringout, and waveform records default is 0.
- Can be changed with info tag in .db file:

```
info(asyn:FIFO, "20")
```

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.
 - For stringout and output waveform records this is only done if the info tag asyn:INITIAL_READBACK=1.
4. From save/restore
5. dbpf at the end of startup script.

Generic Device Support

Callbacks for Output Records

- Normally output records do not use callbacks
- Sometimes want a driver callback to update the output record value (but without sending the callback value back to the driver).

- Can be enabled with

```
info (asyn:READBACK, "1")
```

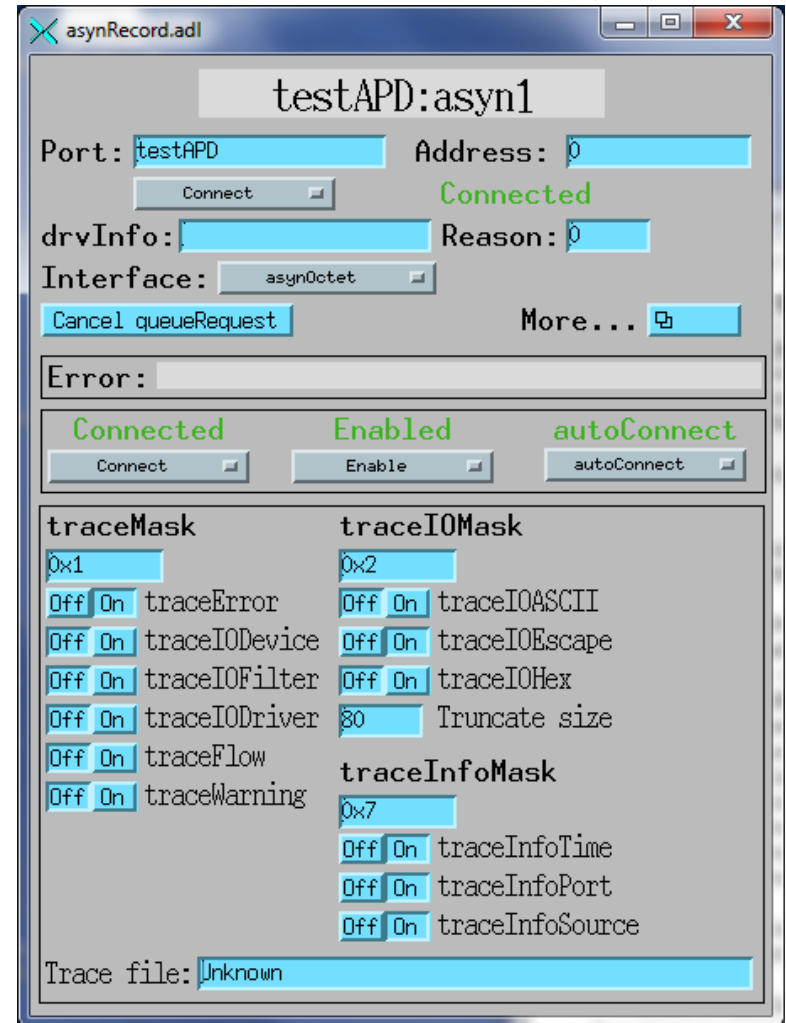
- Example: output file number in areaDetector. Allows use to set the file number, but also have it

```
record(longout, "$ (P) $ (R) FileNumber")
{
    field(PINI, "YES")
    field(DTYP, "asynInt32")
    field(OUT, "@asyn ($ (PORT) , $ (ADDR) , $ (TIMEOUT) ) FILE _ NUMBER")
    field(VAL, "1")
    info (asyn:READBACK, "1")
    info (autosaveFields, "VAL")
}
```

- Note: busy record always implicitly has asyn:READBACK in asyn device support, so it will always process the record on callbacks.

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

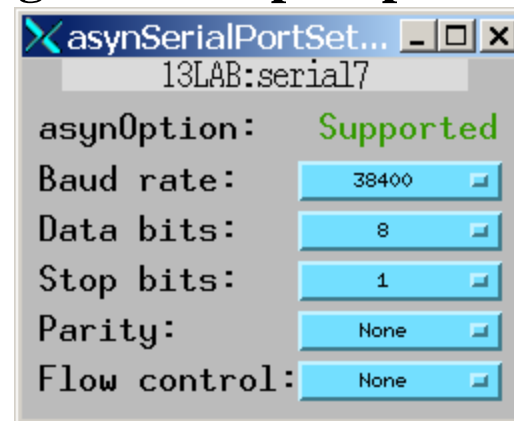


asynRecord – asynOctet devices

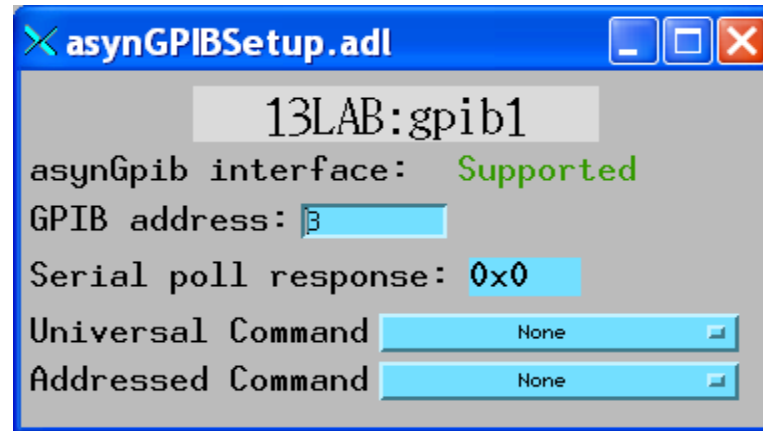
Interactive I/O to octet devices
(serial, TCP/IP, GPIB, etc.)



Configure serial port parameters

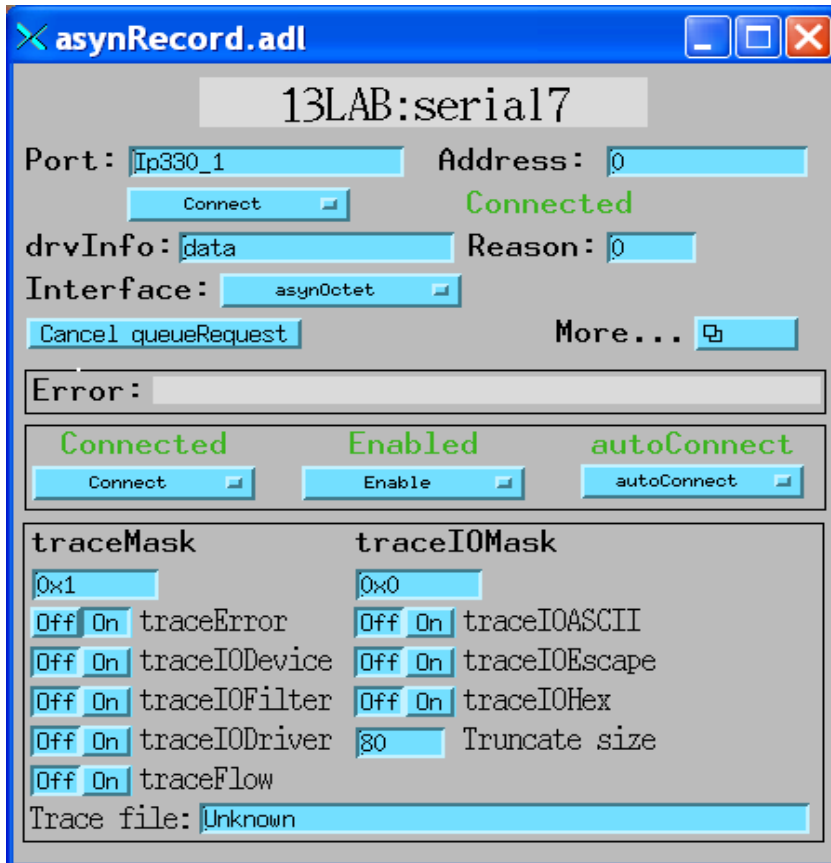


Perform GPIB specific operations



asynRecord – register devices

Same asynRecord, change to ADC port



The screenshot shows the 'asynRecord.adl' window for device '13LAB:serial7'. The 'Port' is 'Ip330_1' and 'Address' is '0'. The status is 'Connected'. The 'Interface' is 'asynOctet'. The 'Error' field is empty. Below the error field, there are three status indicators: 'Connected', 'Enabled', and 'autoConnect', each with a corresponding button. The 'traceMask' and 'traceIOMask' sections contain various options with 'On/Off' checkboxes and input fields. The 'Trace file' is 'Unknown'.

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

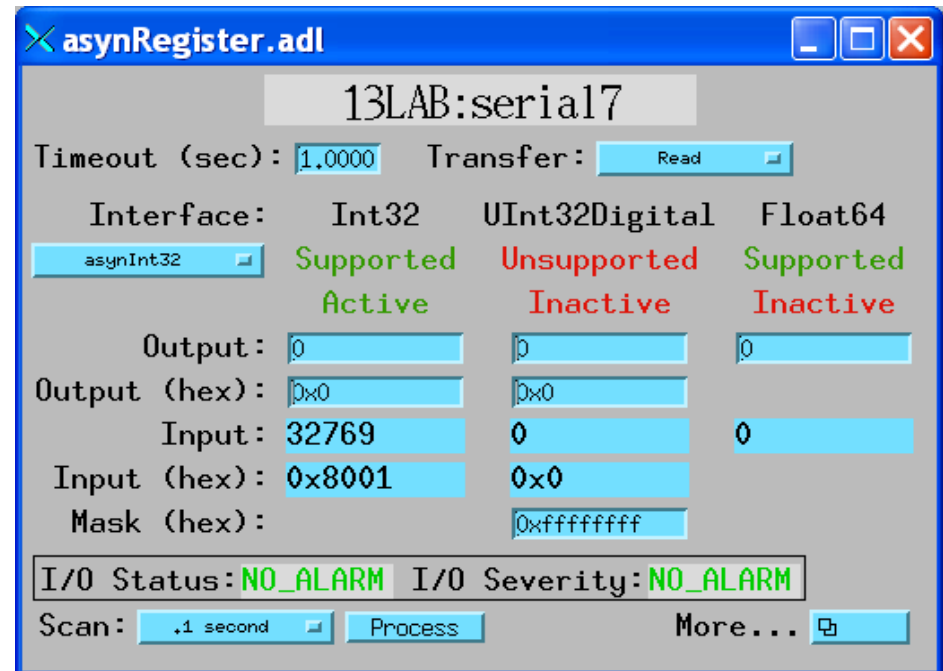
Off On traceIOFilter Off On traceIOHex

Off On traceIODriver 80 Truncate size

Off On traceFlow

Trace file: Unknown

Read ADC at 10Hz with asynInt32 interface



The screenshot shows the 'asynRegister.adl' window for device '13LAB:serial7'. The 'Timeout (sec)' is '1.0000' and 'Transfer' is 'Read'. The 'Interface' is 'asynInt32', which is 'Supported' and 'Active'. Other interfaces 'UInt32Digital' and 'Float64' are 'Unsupported' and 'Inactive'. The 'Output' is '0' and 'Input' is '32769'. The 'Mask (hex)' is '0xffffffff'. The 'I/O Status' is 'NO_ALARM' and 'I/O Severity' is 'NO_ALARM'. The 'Scan' is '.1 second' and 'Process' is active.

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 0

Input: 32769 0 0

Input (hex): 0x8001 0x0

Mask (hex): 0xffffffff

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

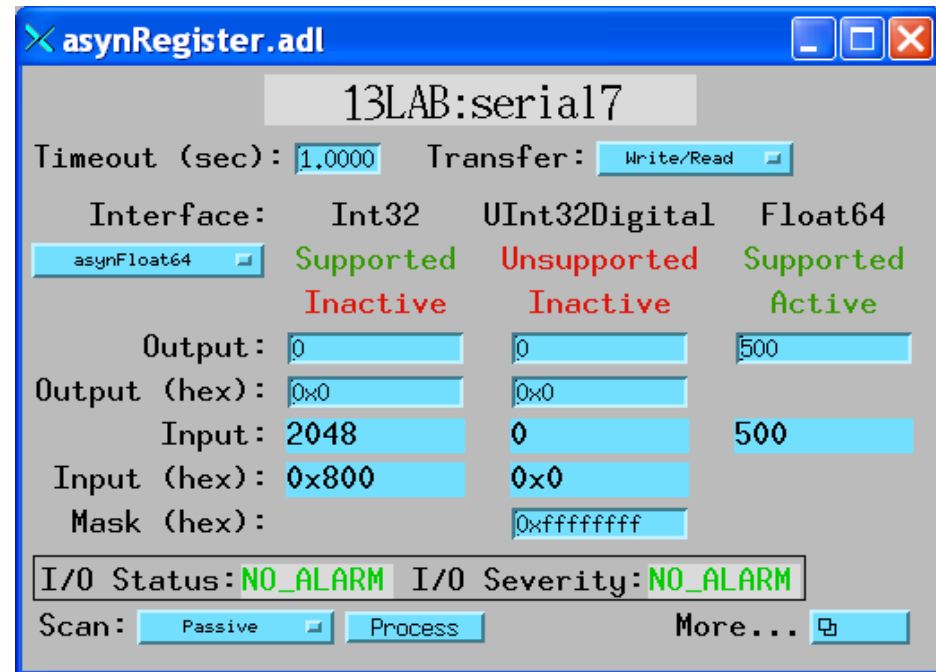
Scan: .1 second Process More...

asynRecord – register devices

Same asynRecord, change to DAC port



Write DAC with asynFloat64 interface



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:

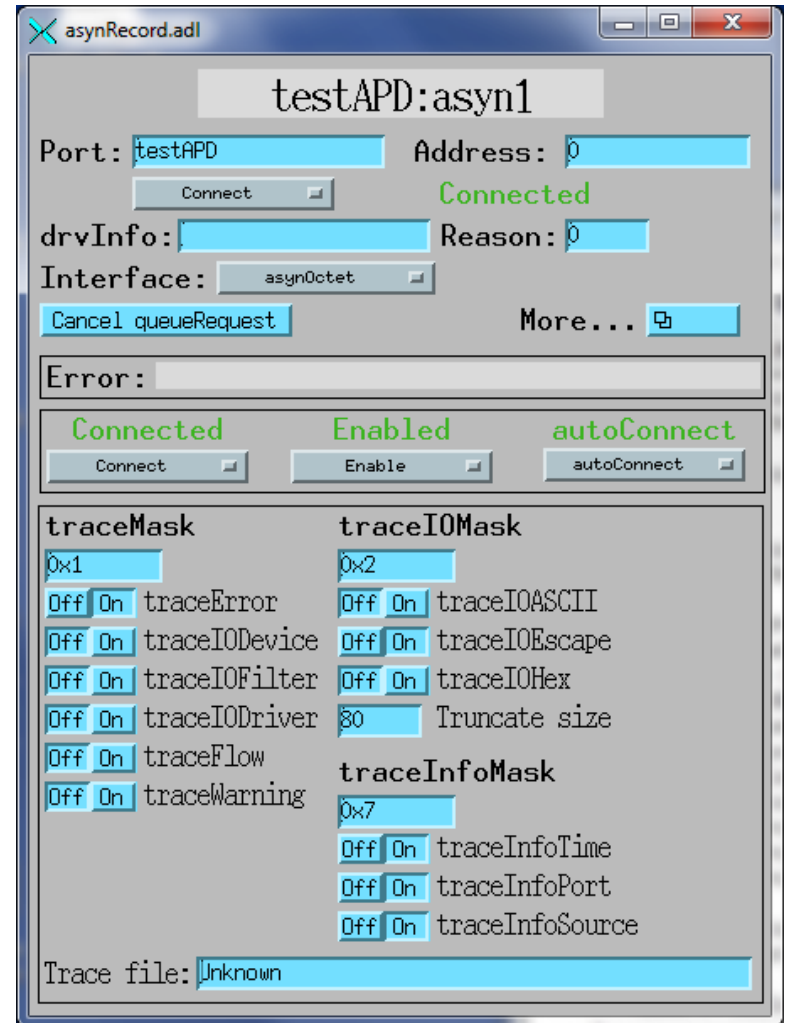
```
status = pasynOctetSyncIO->writeRead(pasynUserScope,
                                     sendMessage, numSend,
                                     receiveMessage,
                                     sizeof(receiveMessage),
                                     WRITE_READ_TIMEOUT,
             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

iocsh Commands

asynReport(level,portName)
asynSetTraceMask(portName,addr,mask)
asynSetTraceIOMask(portName,addr,mask)
asynSetTraceInfoMask(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)
asynRegisterTimeStampSource(portName, functionName)
asynUnregisterTimeStampSource(portName)
asynInterposeFlushConfig(portName,addr,timeout)
asynInterposeEosConfig(portName,addr)

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_TRACE_WARNING`
 - `ASYN_TRACE_FLOW`
 - `ASYN_TRACEIO_DEVICE`
 - `ASYN_TRACEIO_FILTER`
 - `ASYN_TRACEIO_DRIVER`
- 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



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
crtsets	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, optional local port number, and optional IP protocol of the device. The format is:

<host>:<port>[:localPort] [protocol]

(e.g. "164.54.9.90:4002", "164.54.9.90:4001:10101", "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.

asyn: Drivers Included in the Module

- `drvAsynIPServerPort` – socket listening driver for TCP or UDP network communications
 - `drvAsynIPServerPortConfigure("portName", "serverInfo", maxClients, priority, noAutoConnect, noProcessEos)`
- `drvVxi11`: GPIB over Ethernet
- GPIB drivers for vxWorks and Linux
- USB TMC (Test and Measurement Class) for some USB devices

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.
- asynPortDriver (next talk) simplifies many things, makes much less code.