ASYN Device Support Framework

W. Eric Norum

2006-11-20

Argonne NATIONAL LABORATORY

... for a brighter future

U.S. Department of Energy

UChicago Argonne LLC

Office of Science U.S. DEPARTMENT OF ENERGY

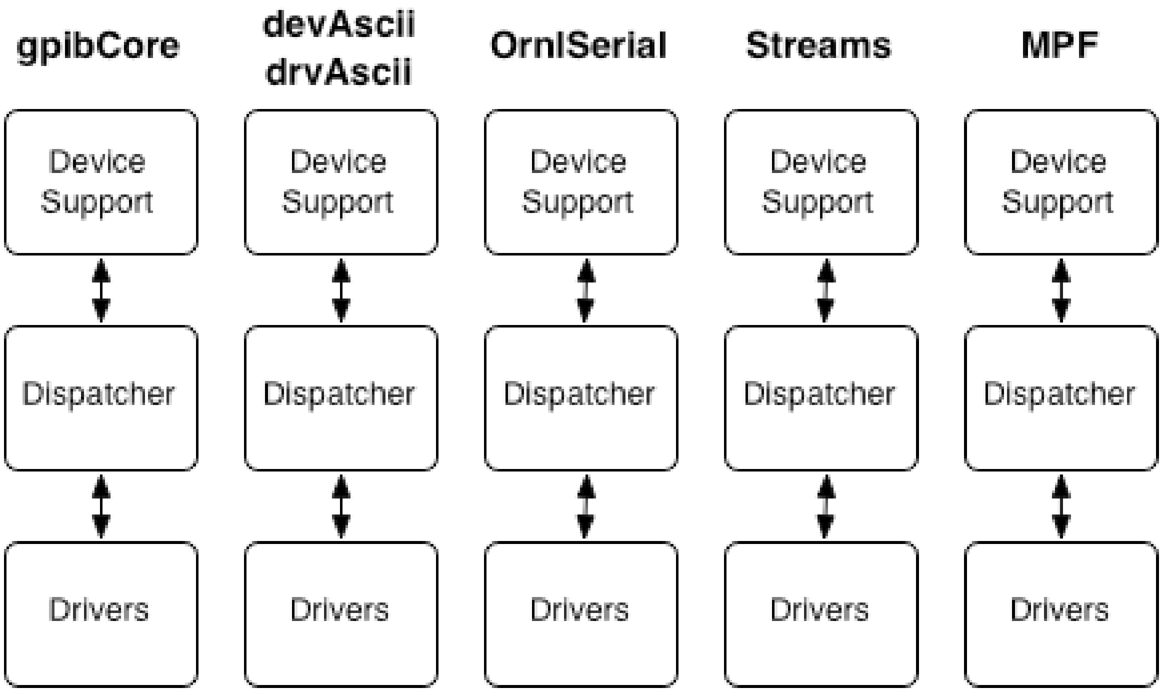
ASYN

- What is it?
- What does it do?
- How does it do it?
- How do I use it?

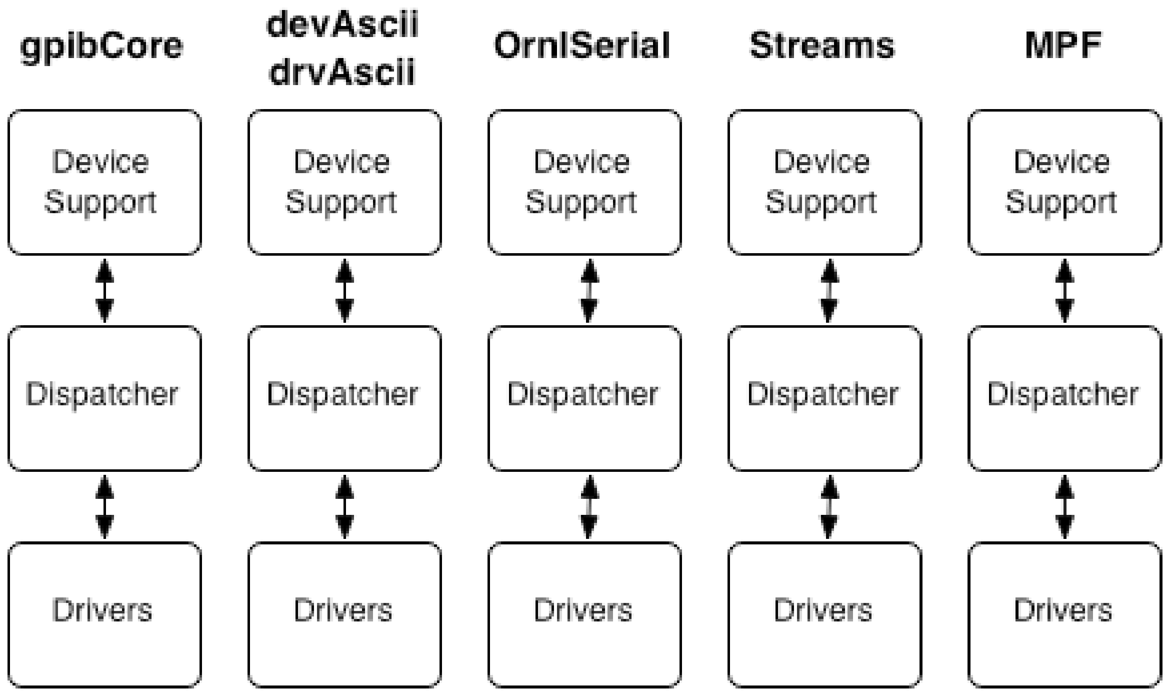
What is it?

Asynchronous Driver Support is a general purpose facility for interfacing device specific code to low level communication drivers

The problem – Duplication of effort

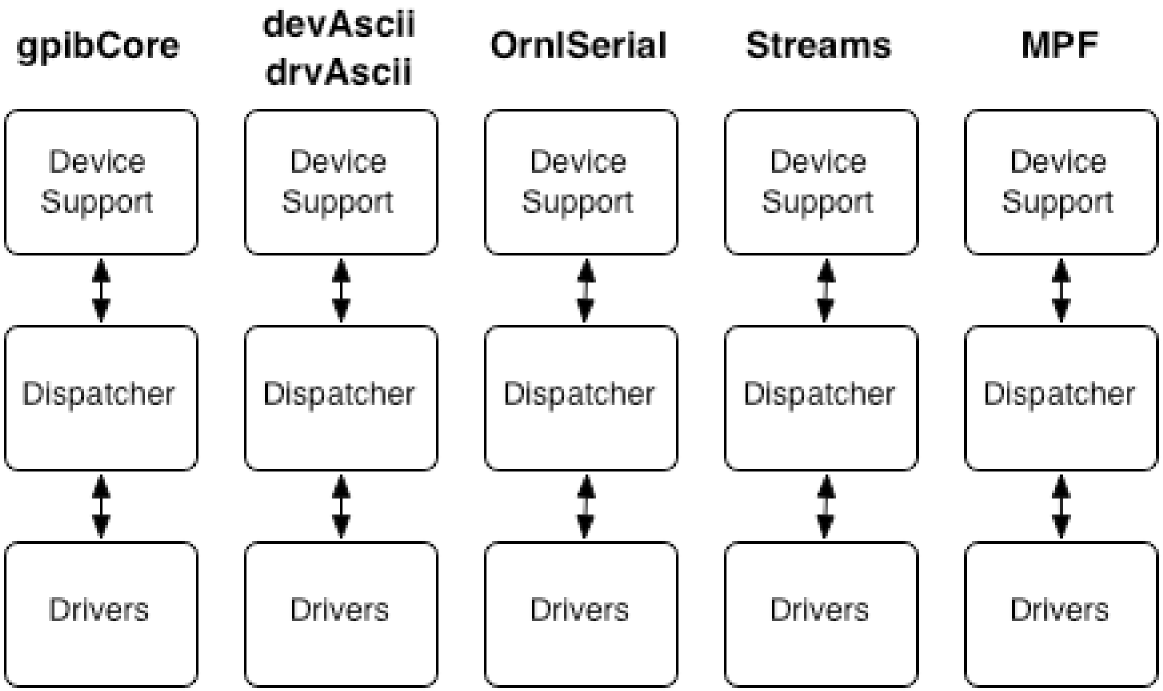


The problem – Duplication of effort



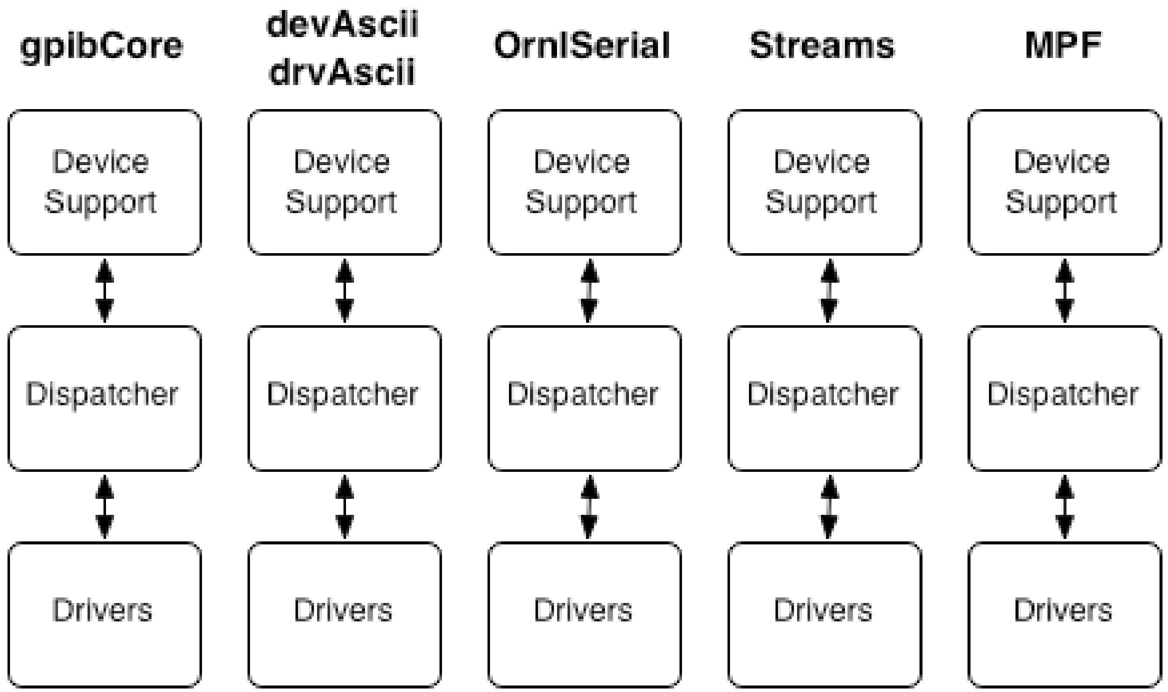
- Each device support has its own asynchronous I/O Dispatcher
  – All with different degrees of support for message concurrency and connection management

The problem – Duplication of effort



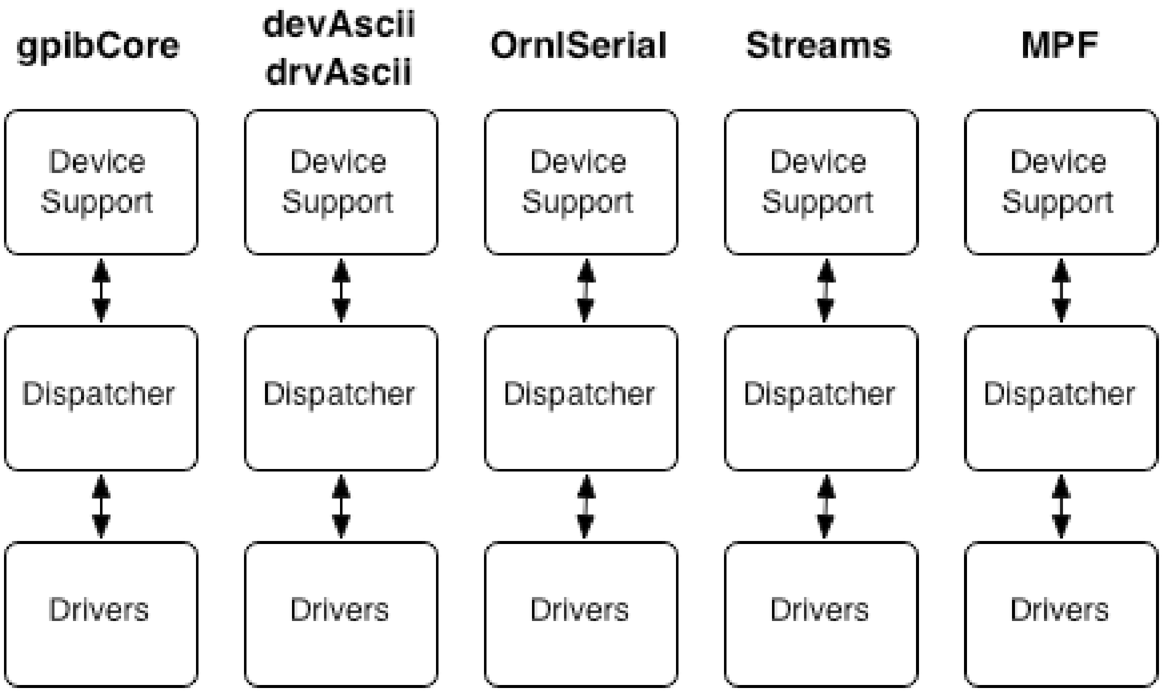
- Each device support has its own set of low-level drivers
  - All with different driver coverage

The problem – Duplication of effort



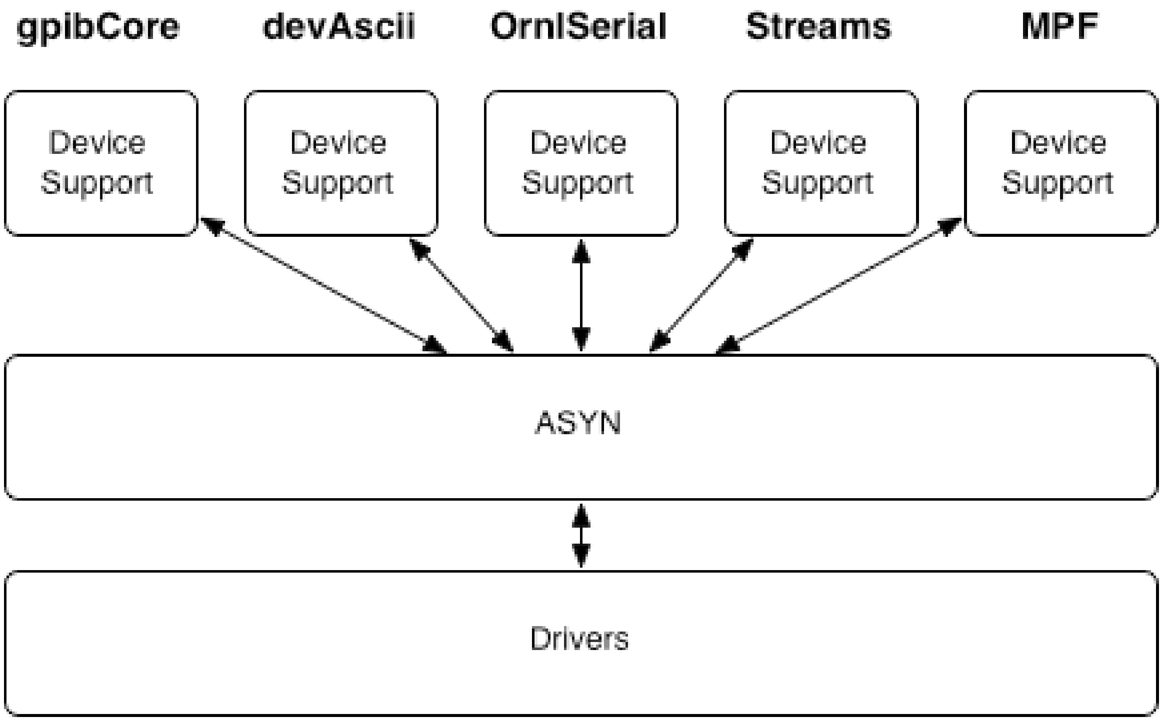
- Not possible to get all users to switch to one devXXX
  - Many 10s of thousands of record instances
  - 100s of device support modules

The problem – Duplication of effort



- R3.14 makes the situation a whole lot worse:
  - Adds another dimension to the table – multiple architectures
  - vxWorks, POSIX (Linux, Solaris, OS X), Windows, RTEMS

The solution – ASYN



The solution – ASYN

- Cost
  - Device support code must be rewritten

The solution – ASYN

- Cost
  - Device support code must be rewritten
  - Drivers must be rewritten

The solution – ASYN

- Cost
  - Device support code must be rewritten
  - Drivers must be rewritten
  - Hmmm….sounds like, “Be reasonable, do it my way”.

    Have we just added another column to the ‘problem’ figure?

The solution – ASYN

- Cost
  - Device support code must be rewritten
  - Drivers must be rewritten
  - Hmmm….sounds like, “Be reasonable, do it my way”

    Have we just added another column to the ‘problem’ figure?
- Benefit
  - Rewrite driver once – works with *all* types of device support
  - Drivers are now an *O(1)* problem rather than an *O(n)* problem
    - *Several drivers done – O(0) problem*

The solution – ASYN

- Cost
  - Device support code must be rewritten
  - Drivers must be rewritten
  - Hmmm….sounds like, “Be reasonable, do it my way”.

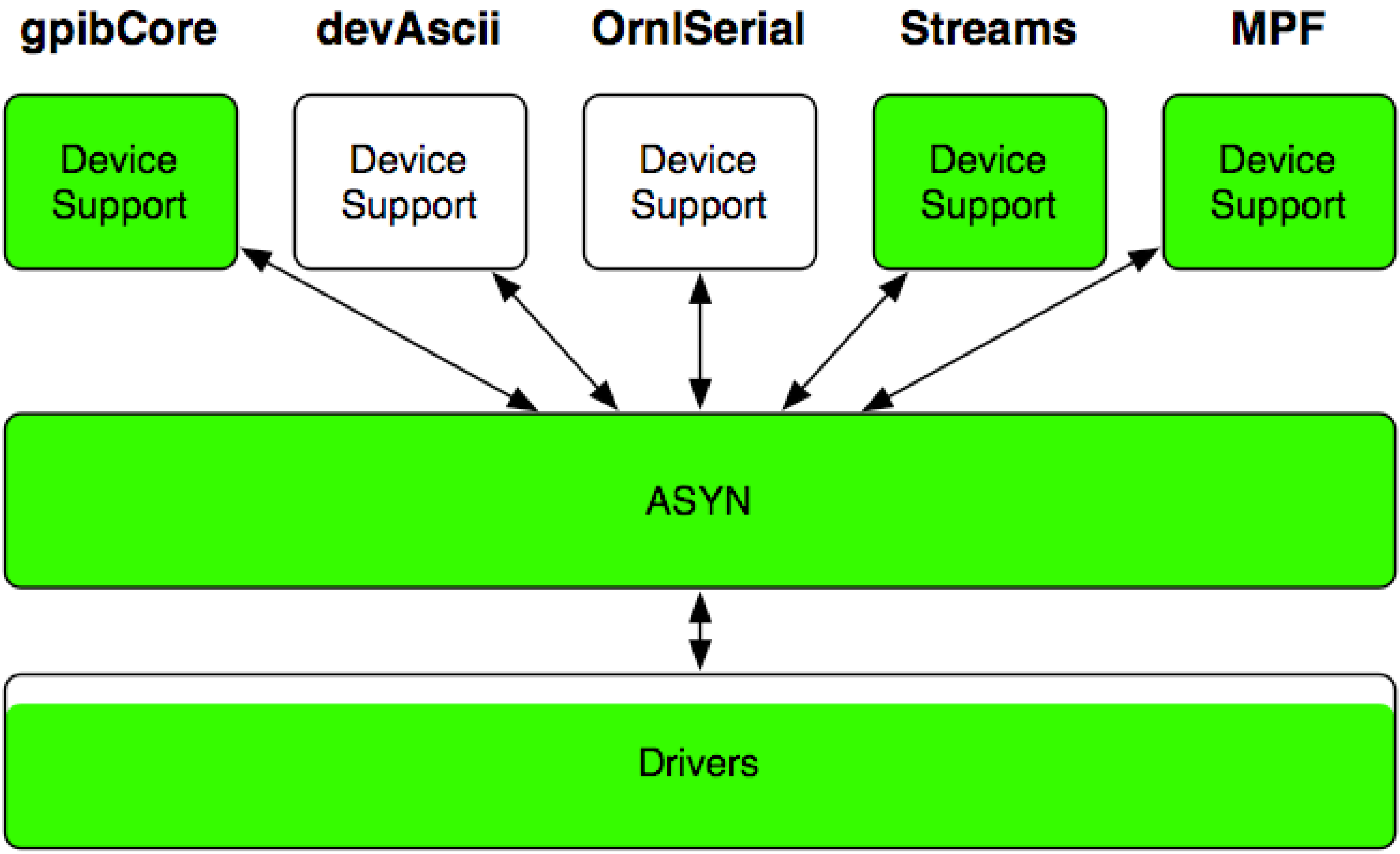
    Have we just added another column to the ‘problem’ figure?
- Benefit
  - Rewrite driver once – works with *all* types of device support
  - Drivers are now an *O(1)* problem rather than an *O(n)* problem
    - *Several drivers done – O(0) problem*
  - Common connection management

The solution – ASYN

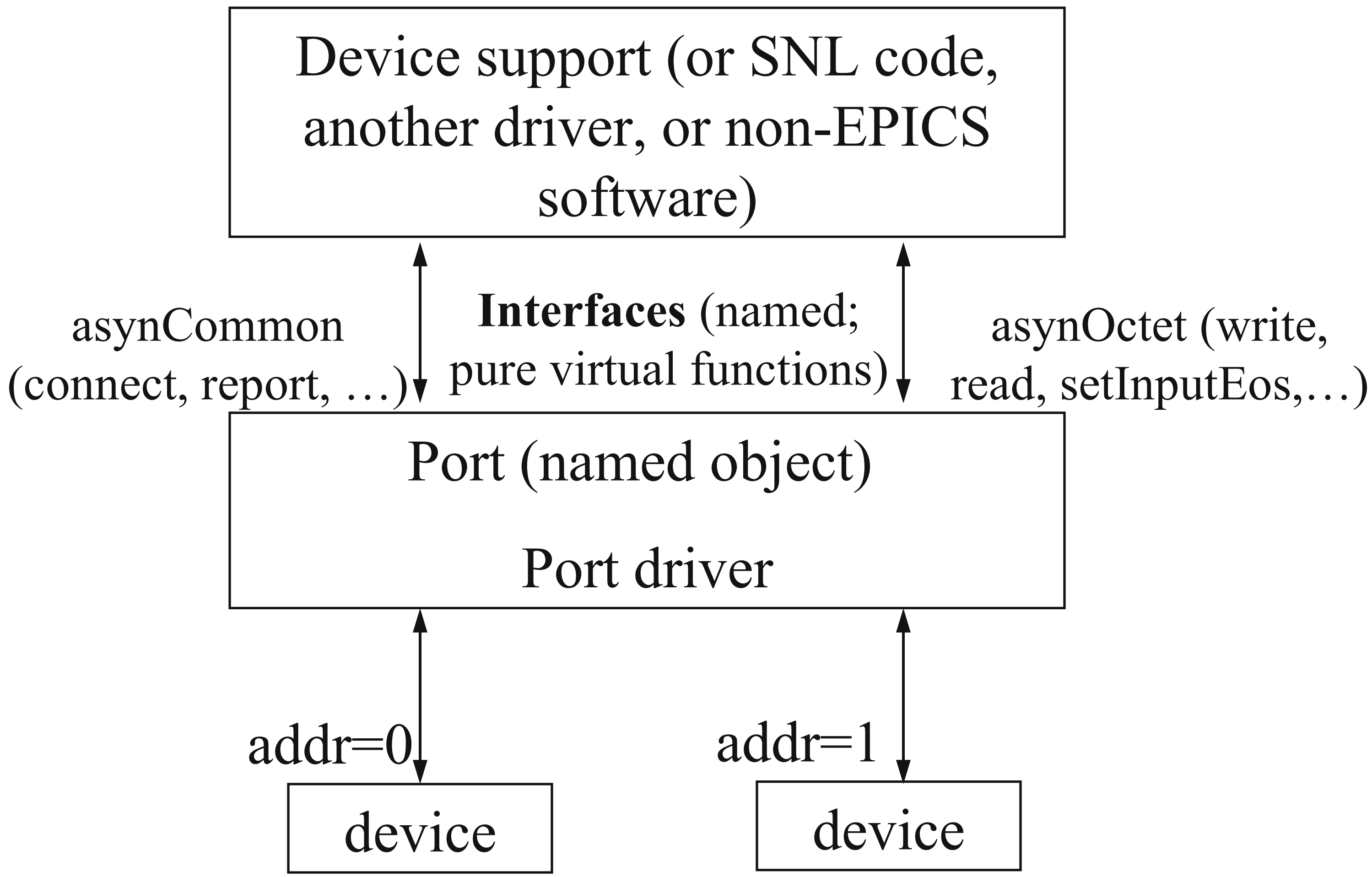
- Cost
  - Device support code must be rewritten
  - Drivers must be rewritten
  - Hmmm….sounds like, “Be reasonable, do it my way”.

    Have we just added another column to the ‘problem’ figure?
- Benefit
  - Rewrite driver once – works with *all* types of device support
  - Drivers are now an *O(1)* problem rather than an *O(n)* problem
    - *Several drivers done – O(0) problem*
  - Common connection management
    - *And it even works! – Passes the ‘Dalesio’ test*

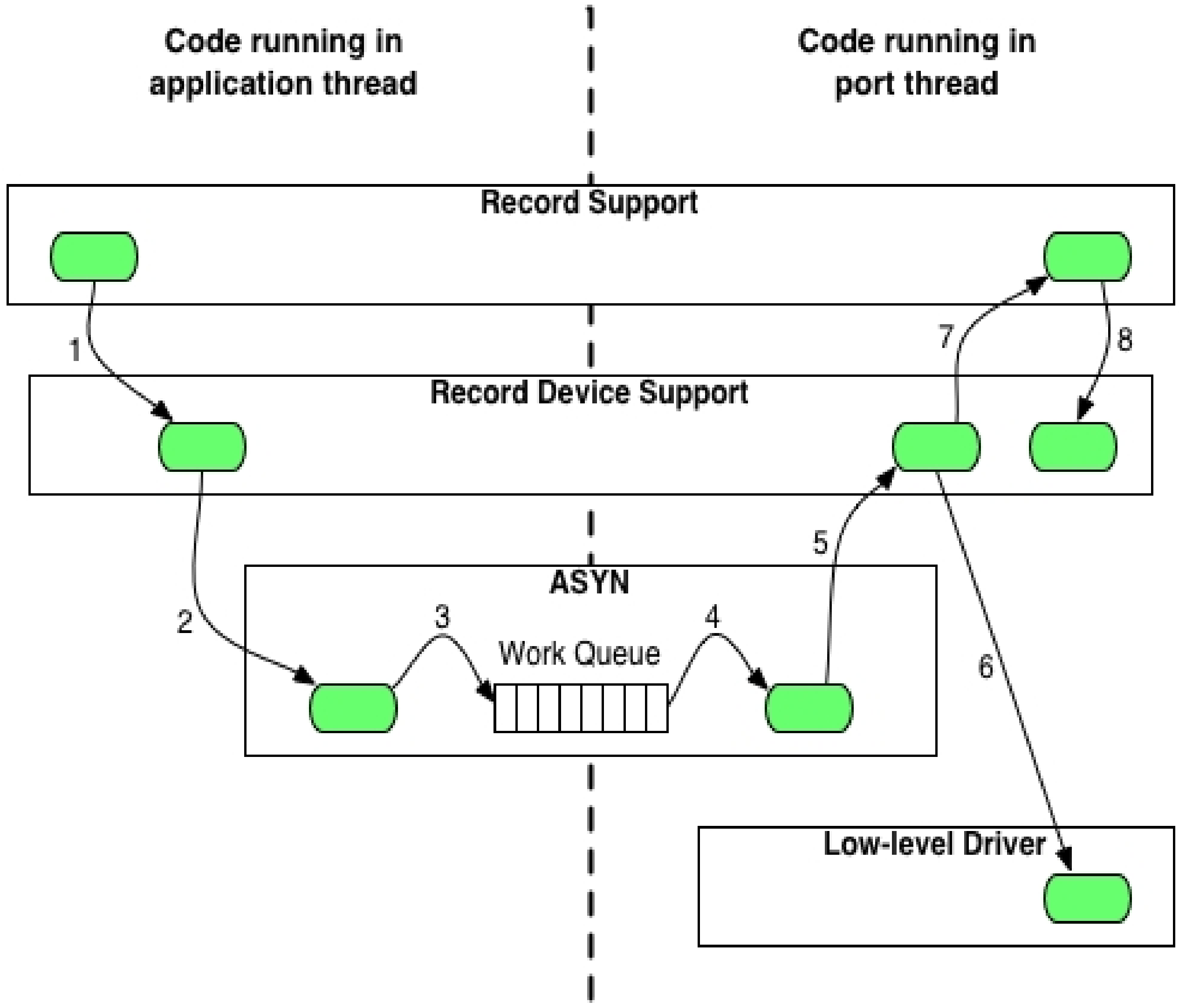
ASYN status


gpibCore
devAscii
OrnlSerial
Streams
MPF
Device Support
Device Support
Device Support
Device Support
Device Support
ASYN
Drivers

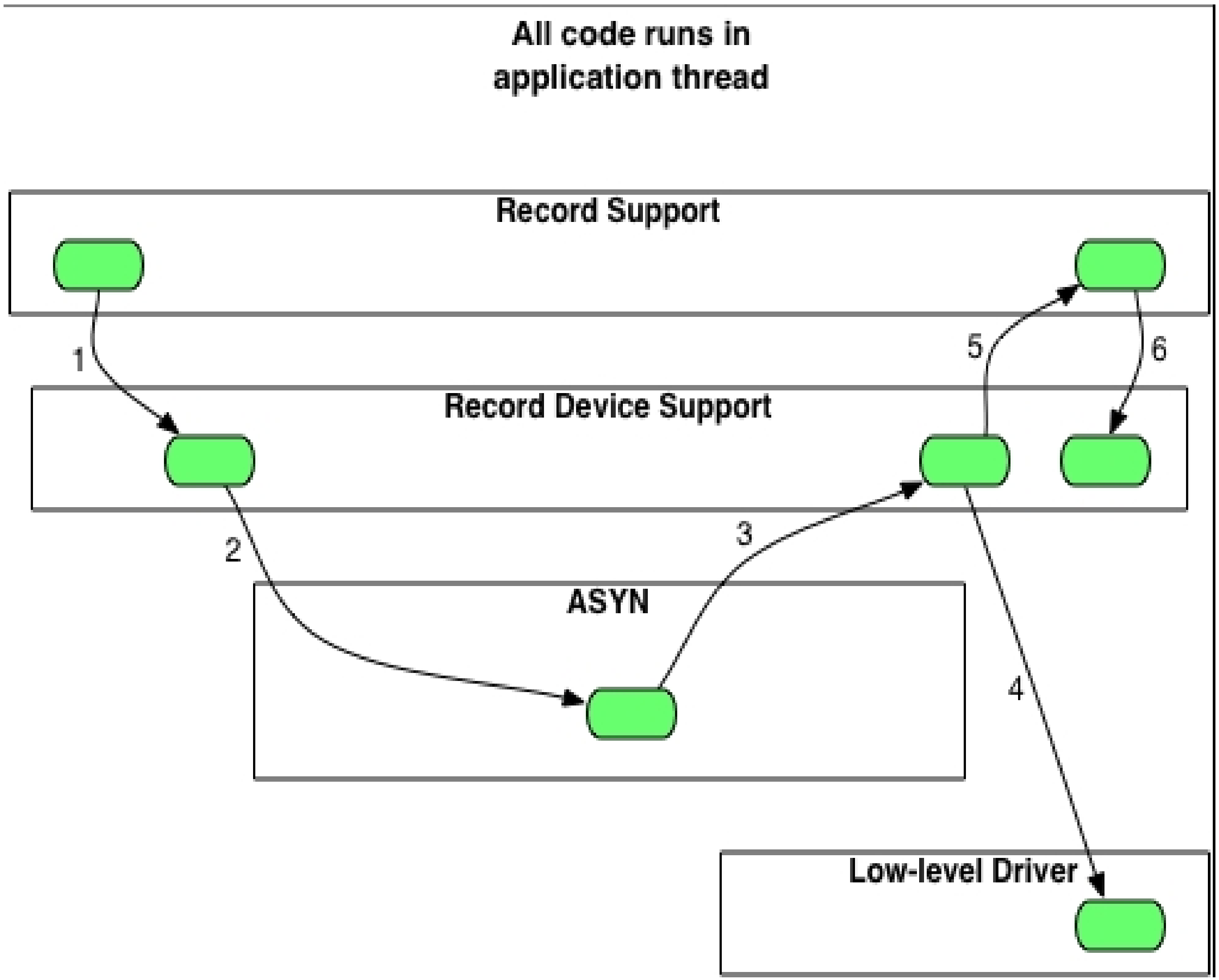
asyn Architecture



Control flow – asynchronous driver



Control flow – synchronous driver



ASYN Components – asynManager

- Provides thread for each communication interface
  - All driver code executes in the context of this thread
- Provides connection management
  - Driver code reports connect/disconnect events
- Queues requests for work
  - Nonblocking – can be called by scan tasks
  - User-supplied callback code run in worker-thread context makes calls to driver
  - Driver code executes in a single-threaded synchronous environment
- Handles registration
  - Low level drivers register themselves
  - Can 'interpose' processing layers

ASYN Components – asynCommon

- A group of methods provided by all drivers:
  - Report
  - Connect
  - Disconnect
  - Set option
  - Get option
    - *Options are defined by low-level drivers*
    - *e.g., serial port rate, parity, stop bits, handshaking*

ASYN Components – asynOctet

- Driver or interposed processing layer
- Methods provided in addition to those of asynCommon:
  - Read
  - Write
  - Set end-of-string character(s)
  - Get end-of-string character(s)
- All that's needed for serial ports, 'telnet-style' TCP/IP devices
- The single-threaded synchronous environment makes driver development much easier
  - No fussing with mutexes
  - No need to set up I/O worker threads

ASYN Components – asynGpib

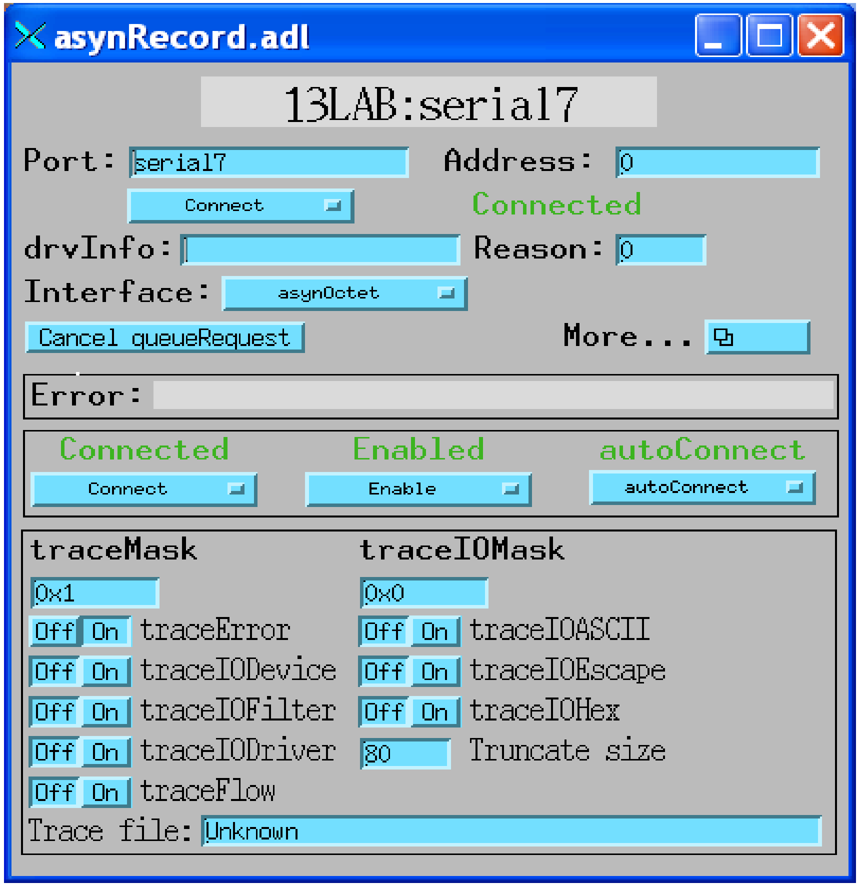
- Methods provided in addition to those of asynOctet:
  - Send addressed command string to device
  - Send universal command string
  - Pulse IFC line
  - Set state of REN line
  - Report state of SRQ line
  - Begin/end serial poll operation
- Interface includes asynCommon and asynOctet methods
  - Device support that uses read/write requests can use asynOctet drivers. Single device support source works with serial and GPIB!

ASYN Components – asynRecord

- Diagnostics
  - Set device support and driver diagnostic message masks
  - No more ad-hoc ‘debug’ variables!
- General-purpose I/O
  - Replaces synApps serial record and GPIB record
- Provides much of the old ‘GI’ functionality
  - Type in command, view reply
  - Works with **all** asyn drivers
- A single record instance provides access to all devices in IOC

asynRecord

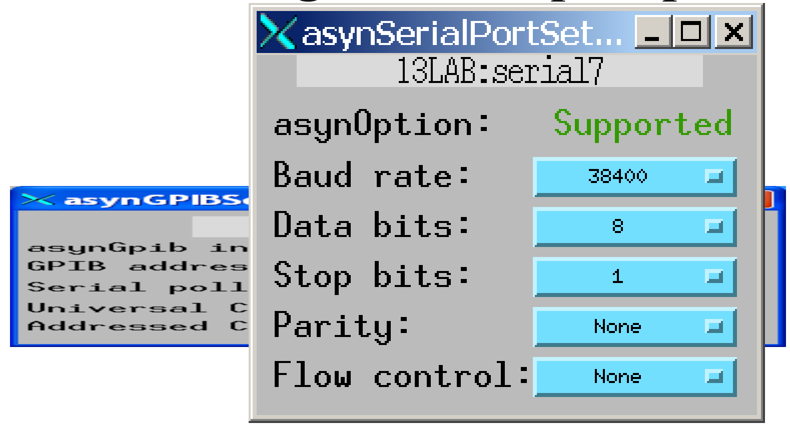
- EPICS record that provides access to most features of asyn, including standard I/O interfaces
- Applications:
  - Control tracing (debugging)
  - Connection management
  - Perform interactive I/O
- Very useful for testing, debugging, and actual I/O in many cases
- Replaces the old generic “serial”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



asynRecord – asynOctet devices

Configure serial port parameters

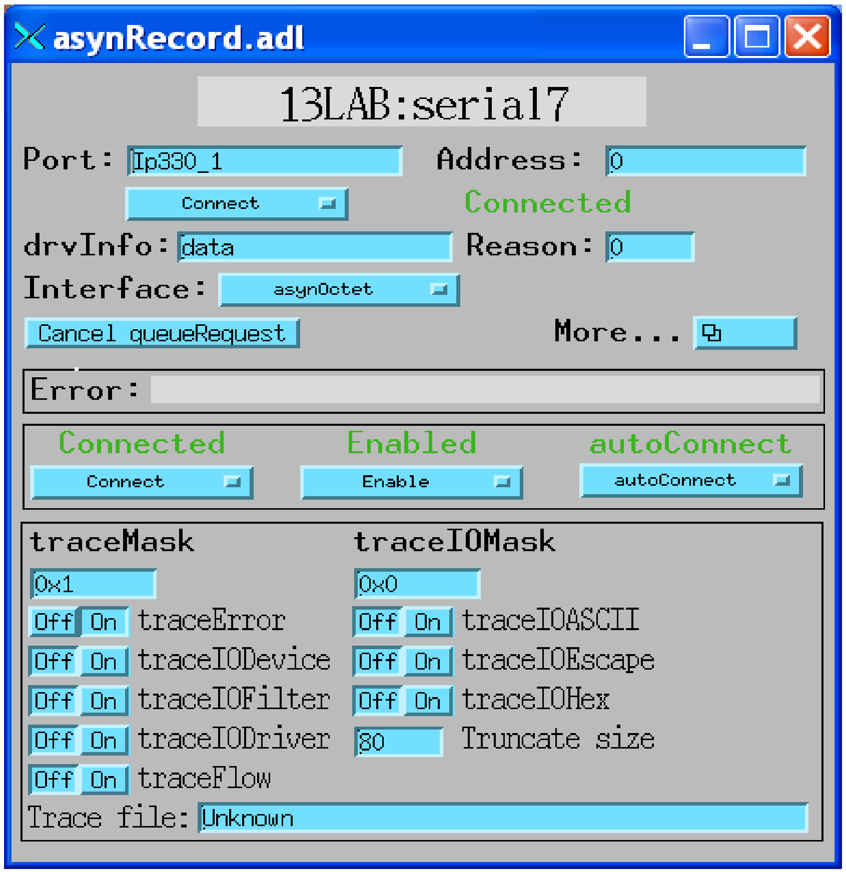
Interactive I/O to serial device



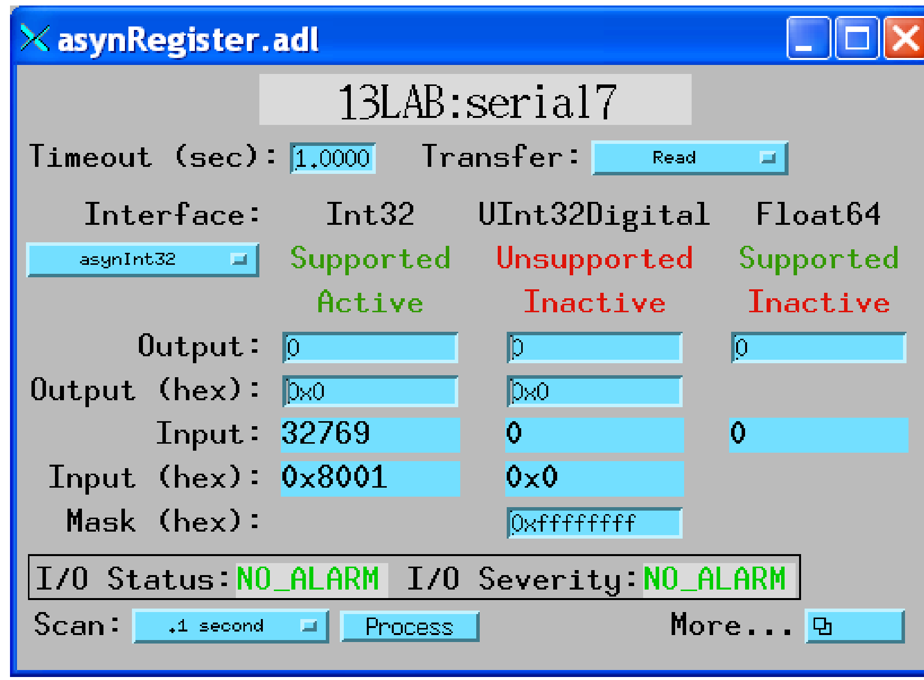
Perform GPIB specific operations

asynRecord – register devices

Same asynRecord, change to ADC port

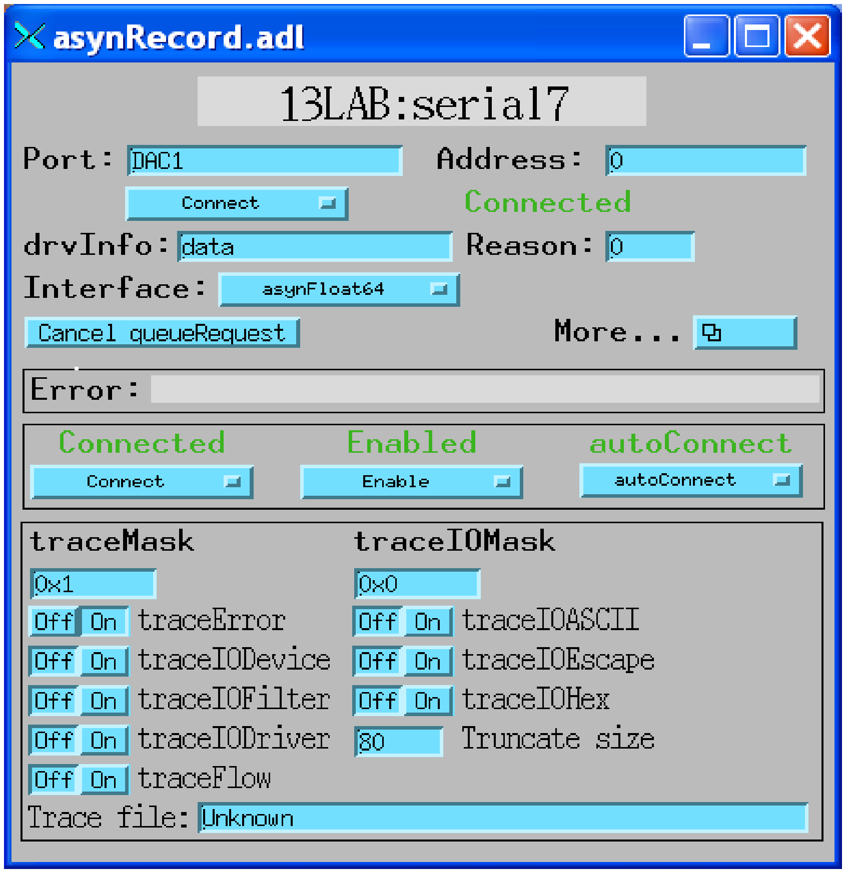


Read ADC at 10Hz with asynInt32 interface

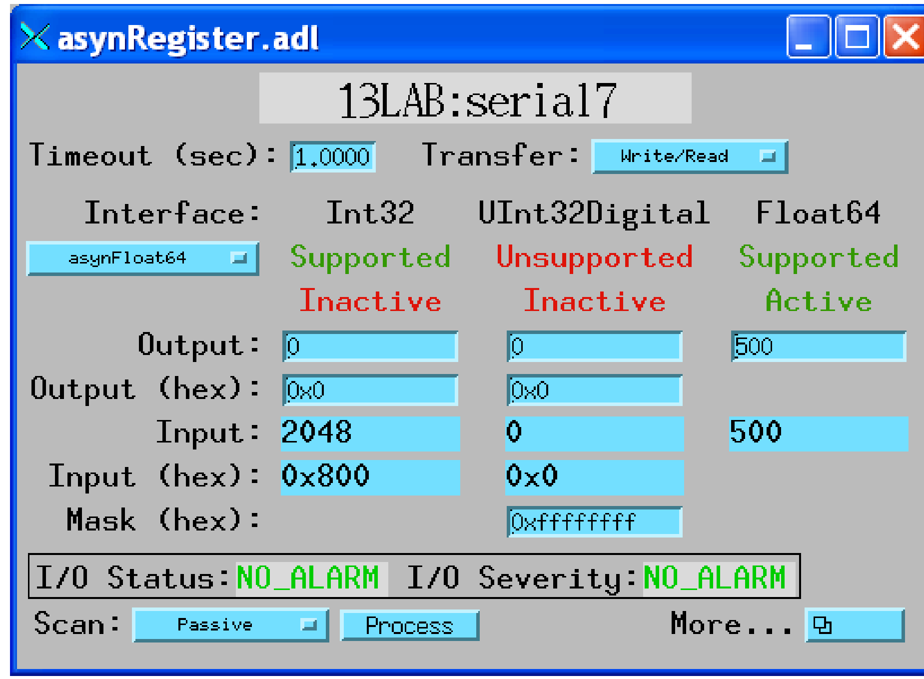


asynRecord – register devices

Same asynRecord, change to DAC port

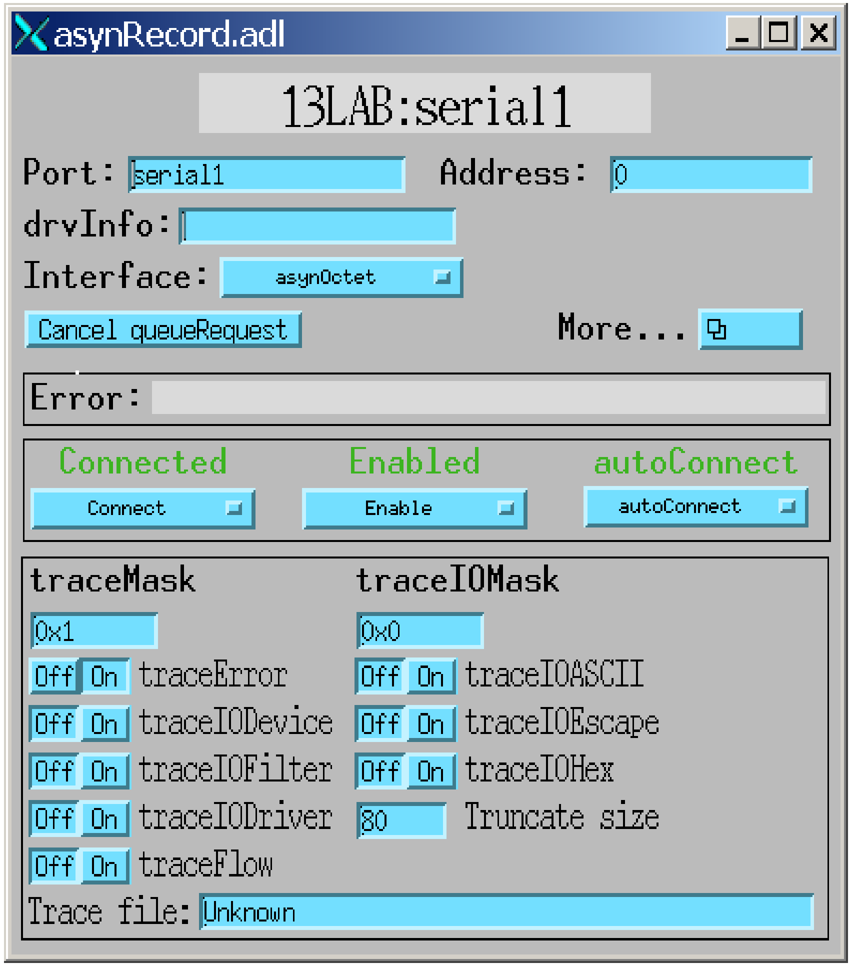


Write DAC with asynFloat64 interface



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



Great – So how do I use it?

- Adding existing device support to an application
- Writing support for a message-based (asynchronous) device
  - devGpib
  - Streams
  - Custom
- Writing support for a register-based (synchronous) device
- Dealing with interrupts
  - ‘Completion’ interrupts
  - ‘Trigger’ (unsolicited) interrupts

Adding ASYN instrument support to an application

Adding ASYN instrument support to an application

- This is easy because the instrument support developers always follow all the guidelines – right?
- The following procedure is taken from:

  How to create EPICS device support for a simple serial or GPIB device

Make some changes to configure/RELEASE

- Edit the configure/RELEASE file created by makeBaseApp.pl
- Confirm that the EPICS_BASE path is correct
- Add entries for ASYN and desired instruments
- For example:
  - `AB300    =/home/EPICS/modules/instrument/ab300/1-1`
  - `ASYN     =/home/EPICS/modules/soft/asyn/3-2`
  - `EPICS_BASE=/home/EPICS/base`

Modify the application database definition file

- If you are building your application database definition from an xxxInclude.dbd file, then include the additional database definitions in that file:

```
include "base.dbd"
include "devAB300.dbd"
include "drvAsynIPPort.dbd"
include "drvAsynSerialPort.dbd"
```

Modify the application database definition file

- If you are building your application database definition from the application Makefile, you specify the additional database definitions there:

  .

  .

  *xxx*_DBD += base.dbd

  *xxx*_DBD += devAB300.dbd

  *xxx*_DBD += drvAsynIPPort.dbd

  *xxx*_DBD += drvAsynSerialPort.dbd

  .

  .

Add support libraries to the application

- You must link the instrument support library and the ASYN library with the application
- Add the lines
  *xxx*_LIBS += devAB300
  *xxx*_LIBS += asyn
  before the
  *xxx*_LIBS += $(EPICS_BASE_IOC_LIBS)
  line in the application Makefile

Modify the application startup script

dbLoadRecords(“db/devAB300.db”,”P=AB300:,R=,L=0,A=0”)

- P,R - PV name prefixes – PV names are (P)(R)name
- L - Link number from corresponding devxxxxConfigure command

  drvAsynIPPortConfigure("L0","192.168.3.137:4001",0,0,0)
- A - Device address

Writing ASYN instrument support

Guidelines for converting or writing instrument support

- Strive to make the instrument support useful by others
- Try to support all the capabilities of the instrument
- Keep names and functions as general as possible
- Stick to the prescribed source/library layout

Converting or writing instrument support?

- Strive to make the instrument support useable by others
- Try to support all the capabilities of the instrument
- Keep names and functions as general as possible
- Stick to the prescribed source/library layout
- Maybe even ship some documentation with your support

Recommended source file arrangement

- Instrument support is not tied to EPICS base
- Support should not depend upon other instrument support
- Support should not influence other instrument support
- Which means that:
  - Instrument support is placed in CVS repository in
    - <*xxxxx*>`/modules/instrument/`<*instrumentname*>`/`
  - Each <*instrumentname*> directory contains
    - `Makefile`
    - `configure/`
    - <*InstrumentName*>`Sup/`
    - `documentation/`
    - `License`

There's a script to make this a little easier

- **mkdir *xxxx*/modules/instrument/*myinst***
- **cd *xxxx*/modules/instrument/*myinst***
- ***xxxx*/modules/soft/asyn/bin/<*arch*>/makeSupport.pl -t devGPIB *MyInst***

```
Makefile
configure/
    CONFIG          Makefile        RULES           RULES_TOP
    CONFIG_APP      RELEASE         RULES_DIRS
MyInstSup/
    Makefile devMyInst.c devMyInst.db devMyInst.dbd
documentation/
   devMyInst.html
```

- **A few changes to the latter 4 files and you're done!**

Converting devGpib instrument support

Converting existing devGpib instrument support

See “Updating devGPIB instrument support to ASYN” in the ASYN documentation

- Use makeSupport.pl to create a new instrument support area
- Copy the existing ‘.c’, ‘.db’ and ‘.dbd’ files to the new support area
- Make some changes to the ‘.c’ file
  - Remove a bunch of lines
  - Make a minor change to each command table entry
  - Change the device-support initialization
- Make some minor changes to the ‘.db’ file
- Build -- test -- release

Example of converted instrument support

- Simple digital voltmeter – Keithley 196
- ~130 lines removed
- 2 lines added
- ~22 lines changed
- More complex device would have about the same number of lines removed and added, but would have more lines changed
  - mostly by rote
- Changes shown on following pages – don't worry about the details
- Somewhat artificial example
  - Very simple device
  - Didn't abide by "Make generally useful; Fully support" rules

Writing devGpib instrument support

Applies to serial and network devices too!

For instruments such as:

- Those connected to local GPIB ports (vxWorks-only)
  - IP-488
  - NI-1014
- Those connected to remote GPIB ports
  - Agilent E5810, E2050
  - Tektronix AD007
- Those connected to local serial ports (e.g. COM1:, /dev/ttyS0)
- Those connected to remote serial ports (e.g. MOXA box)
- Serial-over-Ethernet devices (‘telnet-style’)
- VXI-11 Ethernet devices (e.g., Tektronix TDS3000 oscilloscopes)

New support for a message-based instrument (devGPIB)

- /<*path*>/makeSupport.pl -t devGpib <*InstrumentName*>
- Confirm configure/RELEASE entries for ASYN and BASE
- Modify InstrumentNameSup/devInstrumentName.c
  - Specify appropriate TIMEOUT and TIMEWINDOW values
  - Specify tables of command/response strings and record initialization strings (if needed)
  - Write any custom conversion or I/O routines
  - Set respond2Writes as appropriate (in `init_ai` routine)
  - Fill in the command table
    - *dset, type, priority, command, format, rsplen, msglen, convert, P1, P2, P3, pdevGpibNames, eos*

New support for a message-based instrument (devGPIB)

dset, type, priority, command, format, rsplen, msglen, convert, P1, P2, P3, pdevGpibNames, eos

- /* Param 0 - Identification string */
  {&DSET_SI,GPIBREAD,IB_Q_LOW,"*IDN?","%39[^\n]",0,80,0,0,NULL,NULL,NULL},
- /* Param 3 -- Set frequency */
  {&DSET_AO,GPIBWRITE,IB_Q_LOW,NULL,"FRQ %.4f HZ",0,80,NULL,0,0,NULL,NULL,NULL},
- static char *setDisplay[] = {"DISP:TEXT 'WORKING'","DISPLAY:TEXT:CLEAR",NULL};
  /* Param 2 Display Message: BO */
  {&DSET_BO,GPIBEFASTO,IB_Q_HIGH,NULL,NULL,0,0,NULL,0,0,setDisplay,NULL,NULL},
- /* Param 3 Read Voltage: AI */
  {&DSET_AI,GPIBREAD,IB_Q_HIGH,"MEAS:VOLT:DC?","%lf",0,80,NULL,0,0,NULL,NULL,NULL},
- /* Param 20 -- read amplitude */
  {&DSET_AI,GPIBREAD,IB_Q_LOW,"IAMP",NULL,0,60,convertVoltage,0,0,NULL,NULL,NULL},

New support for a message-based instrument (devGPIB)

```
static int
convertVoltage(gpibDpvt *pgpibDpvt, int P1, int P2, char **P3)
{
    aiRecord *pai = (aiRecord *)pgpibDpvt->precord;
    asynUser *pasynUser = pgpibDpvt->pasynUser;
    double v;
    char units[4];

    if (sscanf(pgpibDpvt->msg, P1 == 0 ? "AMP %lf %3s" : "OFS %lf %3s", &v, units) != 2) {
        epicsSnprintf(pasynUser->errorMessage, pasynUser->errorMessageSize, "Scanf failed");
        return -1;
    }
    if (strcmp(units, "V") == 0) {
    } else if (strcmp(units, "MV") == 0) {
        v *= 1e-3;
    } else {
        epicsSnprintf(pasynUser->errorMessage, pasynUser->errorMessageSize, "Bad units");
        return -1;
    }
    pai->val = v;
    return 0;
}
```

Writing ASYN instrument support

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 – 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()

ASYN API

- Hey, what with terms like ‘methods’ and ‘instances’ this looks very object-oriented – howcome the API is specified in C?
- "I made up the term 'object-oriented', and I can tell you I didn't have C++ in mind" – Alan Kay (The inventor of Smalltalk and of many other interesting things), OOPSLA '97

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 – ledDriver.c

1-10 – Standard headers (cantProceed.h for callocMustSucceed, devLib.h for devWriteProbe)

12-15 – Define location of 8-bit I/O port in CPU memory space

20-24 – Driver private storage declaration. One *asynInterface* structure for each interface provided by this driver.

30-47 – asynCommon methods. All must be present even if empty. Connect and disconnect methods call back to asynManager to register the connection state.

52-60 – asynInt32 methods. Only those needed for this device need be present (see line 98 for why this is true).

65 – Registration routine. Called from within startup script command: `xxx_registerRecordDeviceDriver(pdbbase)`

72 – Allocate the driver private storage (why not static??)

74-77 – Verify that hardware really exists

80-84 – Register the port (single-address, synchronous, auto-connect)

86-93 – Register the asynCommon support provided by this driver

95-102 – Register the asynInt32 support provided by this driver. Note that the pasynInt32Base initialize method is invoked. This provides default methods for all methods not mentioned on line 60 and then invokes registerInterface.

103 – Export the registration routine (so it gets called from IOC startup script)

Generic Device Support – ledDriver.dbd

```
registrar(ledDriverDeviceSupportRegistrar)
```

Generic Device Support – ledDriver.db

```
record(longout,"leds") {
    field(DTYP,"asynInt32")
    field(OUT,"@asyn(ledDriver 0 0)")
}
```

Generic Device Support – acquisitionControl.c

- 14 - uint32Digital – since no mbbiDirect, mbboDirect in asynInt32
- 41 - Probe in connect method rather than registration routine
- 47 - Multiple addresses per port
- 78 - Read method
- 149 - Register port with multiple-address attribute
- 165 - Invoke registerInterface directly (all needed methods provided)

Generic Device Support – acquisitionControl.db

```
record(mbbiDirect, "$(P)ClockFaultMBBI") {
    field(DESC, "Clock status")
    field(DTYP, "asynUInt32Digital")
    field(INP,  "@asynMask(acquisitionControlReg,0,0xFFFF,0)")
    field(SCAN, "2 second")
}
record(bo, "$(P)ClockFaultRbkFrc") {
    field(DESC, "Force clock fault readback")
    field(OUT, "$(P)ClockFaultMBBI.PROC")
}
record(longout, "$(P)ClockFaultClrLO") {
    field(DESC, "Reset clock faults")
    field(DTYP, "asynUInt32Digital")
    field(OUT,  "@asynMask(acquisitionControlReg,0,0xFFFF,0)")
    field(FLNK, "$(P)ClockFaultRbkFrc”)
}
```

Generic Device Support – acquisitionControl.db

```
record(mbbiDirect, "$(P)P0SelectMBBI") {
    field(DESC, "P0 selection")
    field(DTYP, "asynUInt32Digital")
    field(INP,  "@asynMask(acquisitionControlReg,1,0xFFFF,0)")
    field(SCAN, "2 second") }
record(bo, "$(P)P0SelectRbkFrc") {
    field(DESC, "Force P0 select readback")
    field(OUT, "$(P)P0SelectMBBI.PROC") }
record(mbbo, "$(P)P0SelectMBBO") {
    field(DESC, "P0 selection")
    field(DTYP, "asynUInt32Digital")
    field(OUT,  "@asynMask(acquisitionControlReg,1,0x1,0)")
    field(ZRVL, 0)   field(ZRST, "PLL C0")
    field(ONVL, 1)   field(ONST, "PLL C3")
    field(FLNK, "$(P)P0SelectRbkFrc") }
```

Generic Device Support – fpgaProgrammingInfo.c

- 12 - asynOctet – but synchronous
- 26 - another place for the table of methods
- 56 - read configuration information from FPGA ROM
- 88 - IOCshell command rather than EPICS registrar for configuration
- 137 - Set up table of methods
- 164-169 - Register IOCshell command

Generic Device Support – fpgaProgrammingInfo

```
record(stringin, "$(P)$(R)FPGACompileTimeSI") {
    field(DESC, "FPGA compile date/time")
    field(DTYP, "asynOctetRead")
    field(INP,  "@asyn($(PORT) 0 0)")
    field(SCAN, "Passive")
    field(PINI, 1)
}
```

```
#####################################################################
# FPGA version information
devFpgaInfoConfigure("fpgaInfo",0x3800)
dbLoadRecords("db/fpgaProgrammingInfo.db","P=$(P),R=,PORT=fpgaInfo”)
```

Dealing with interrupts

‘Solicited’ interrupts

- e.g., command/response completion
- e.g., txEmpty/rxFull
- Easy to deal with – driver works in blocking, single-threaded environment
- Use devConnectInterruptVME to associate handler with hardware interrupt
- Call epicsEventSignal from low-level interrupt handler
- Driver write method might look like:

```
for(i = 0 ; i < numchars ; i++) {
    send next character to device
    epicsEventWaitWithTimeout(………);
}
```

‘Unsolicited’ interrupts

- Not quite as easy
- e.g., a trigger which will cause records with SCAN(“I/O Intr”) to process
- Driver initialization creates an task which waits for signal from low-level interrupt handler (ASYN routines must **not** be called from low-level handler)
- Configuration must invoke ASYN manager registerInterruptSource
  - Allows subsequent use of interruptStart/End
- The standard interfaces asynInt32, asynInt32Array, asynUInt32Digital, asynFloat64 and asynFloat64Array all support callback methods for interrupts
- Callbacks can be used by device support, other drivers, etc.

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);
...
}
```

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);
```

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);
```

Lab session – Control ‘network-attached device’

TCP Port 24742

- *IDN?
  - Returns device identification string (up to 200 characters long)
- LOADAV?
  - Returns three floating-point numbers (1, 5, 15 minute load average)
- CLIENT?
  - Returns information about client
- VOLTAGE?
  - Returns most recent voltage setting
- VOLTAGE *x.xxxx*
  - Sets voltage