PAUL SCHERRER INSTITUT

PSI

SLS

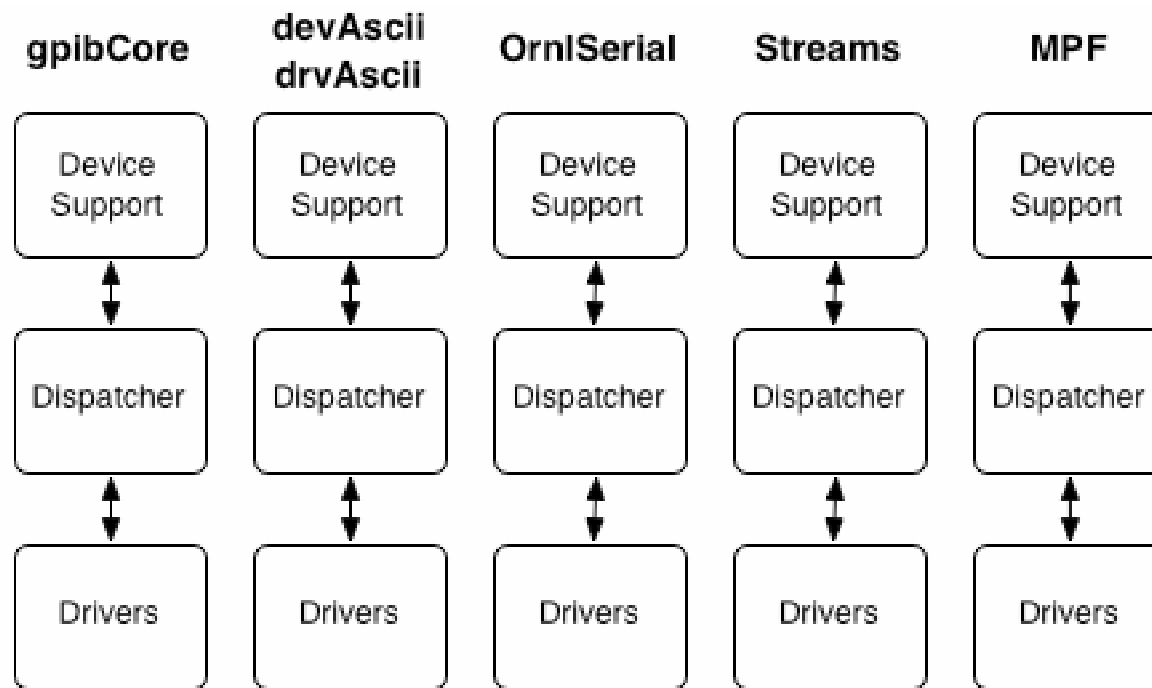
# Introduction to asynDriver

Dirk Zimoch, 2007

# What is asynDriver?

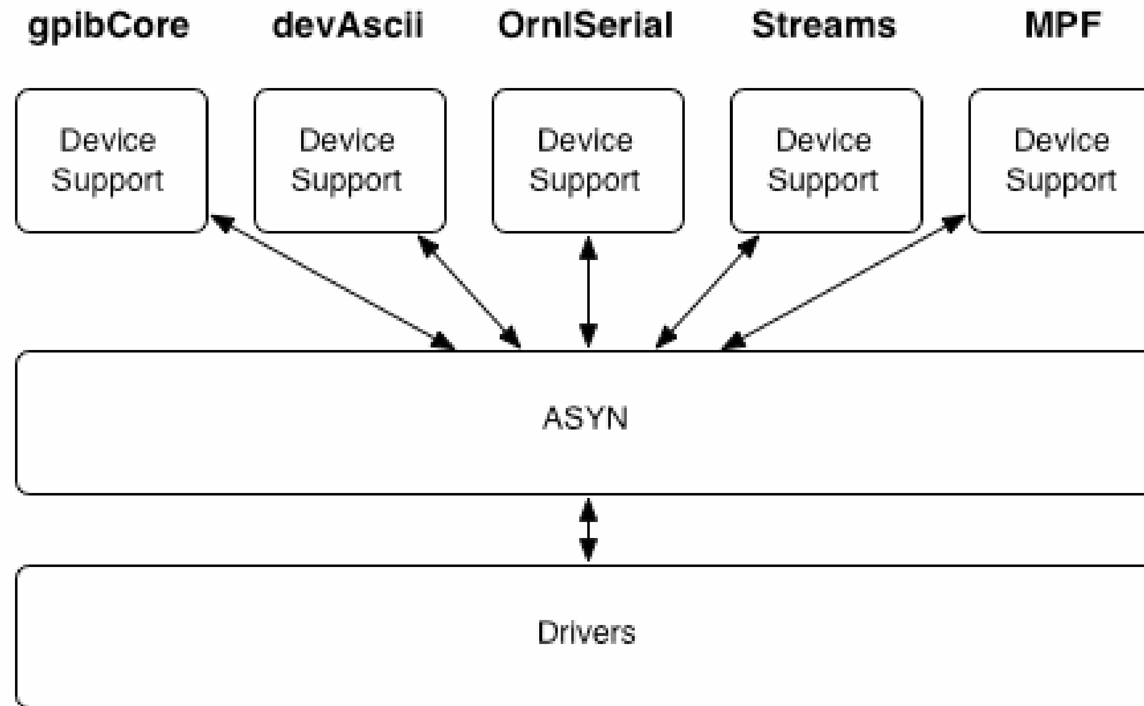
- *"asynDriver is a general purpose facility for interfacing device specific code to low level drivers."*
- What does that mean?
  - It is not a driver – it is a driver framework:
    Interface definitions and a collection of utilities.
- What does it define?
  - Interfaces to different classes (not brands) of hardware.
- What does it provide?
  - Functionalities common to all (or many) drivers.

# The problem




- Separate (incompatible) sets of drivers and device supports.
- Much effort duplicated but different sets of features.

# The plan




- Every device supports works with every driver.
- Much work went to ASYN, less work to do for drivers.

# Provided functionalities

- Dispatcher
  - Thread for asynchronous I/O
  - Interrupt subscription and handling
  - Connection management
  - Message concurrency
  - Configuration (shell) functions
- Debug tools
  - Trace messages, trace files, trace levels
  - General purpose (debug) hardware access
- Set of simple device supports

## Interface definitions

- Old (bad): Device support talks to drivers.
  - Different drivers for different hardware have different interfaces.
  - Need special device support for each type of hardware.
  - No support for other clients than device support.
- New (good): Clients talk to abstract interfaces.
  - Not limited to device supports.
    - Shell (debug) functions
    - Any C (and SNL) code
  - Different device supports can talk to the same hardware.
  - Need only one device support for any type of hardware.

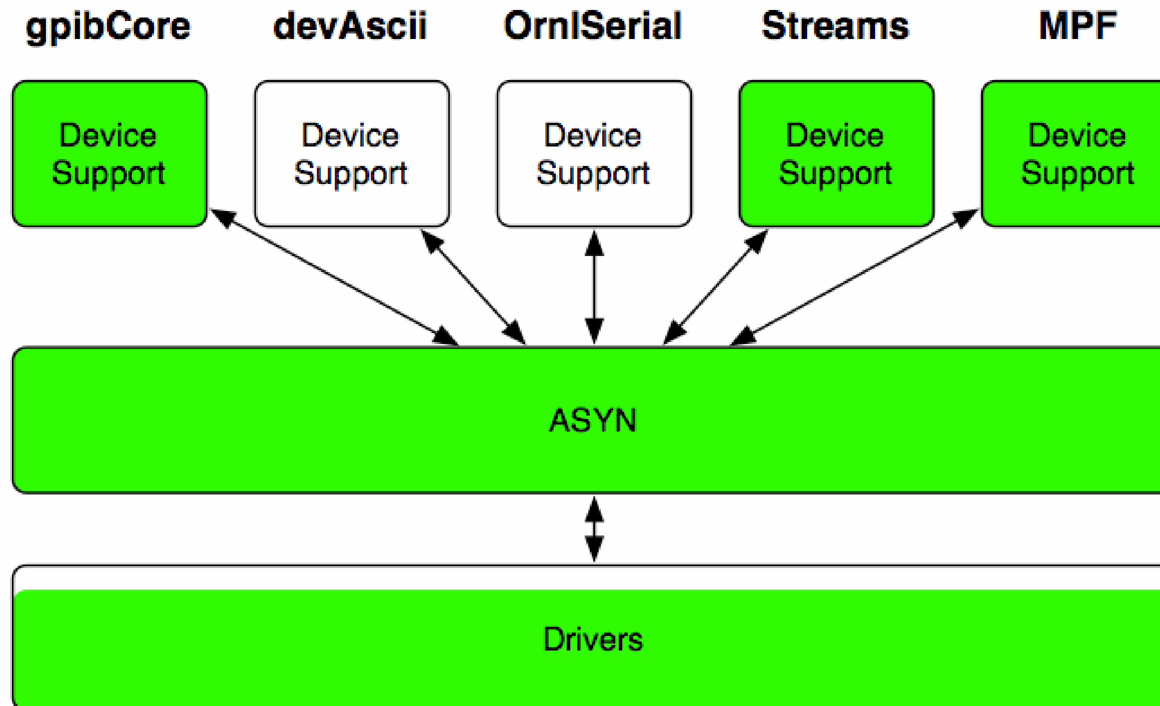
# The cost

- Device supports need to be modified
  - Talk to asyn interfaces instead of driver
- Driver needs to be modified
  - Remove all “private” dispatcher code
  - Use asyn library
  - Implement interfaces for asyn
  - Example: Simple digital voltmeter – Keithley 196
    - ~130 lines removed
    - 2 lines added
    - 22 lines changed

# Benefits

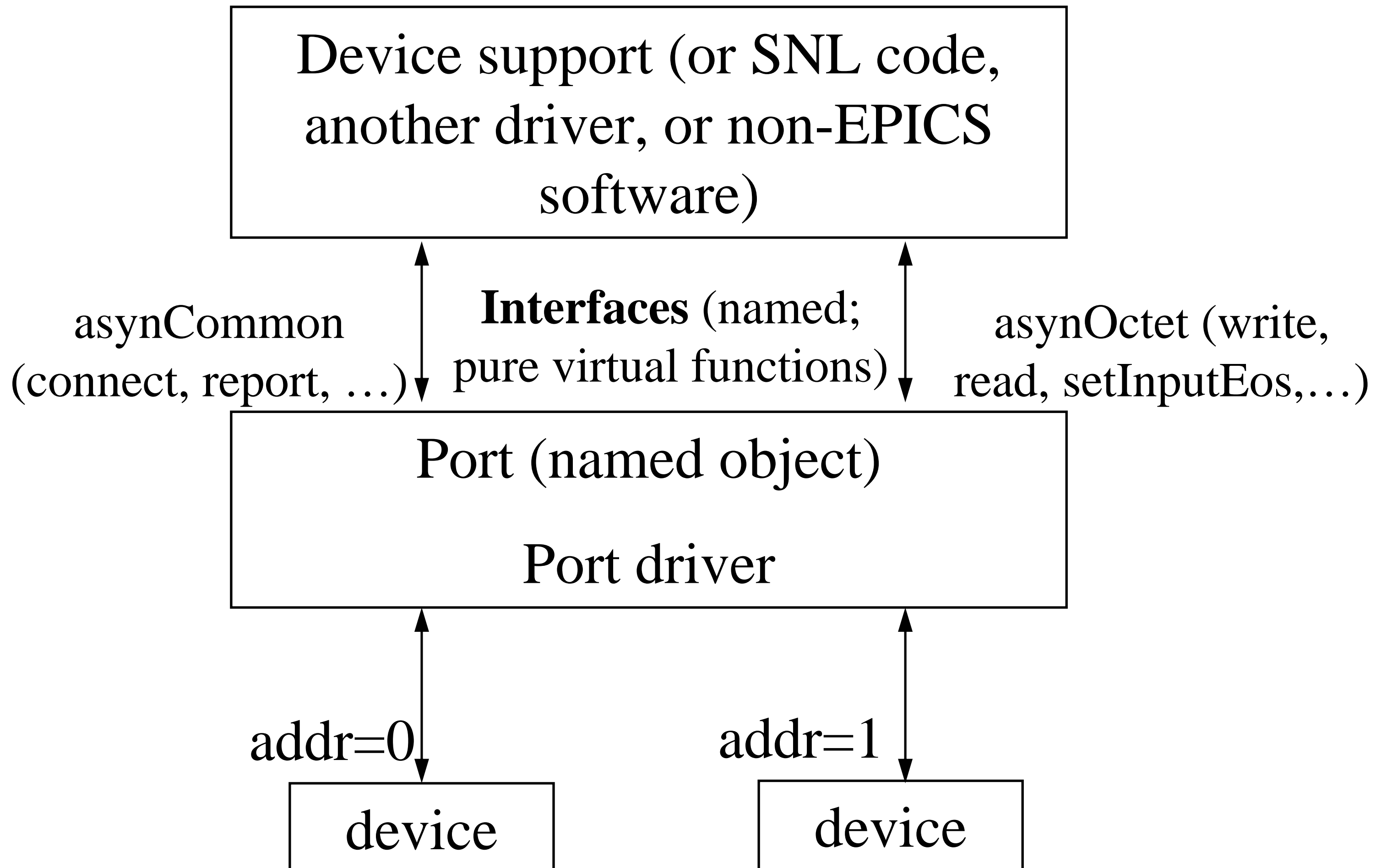
- New devices need to be implemented only once.
  - All device supports can use all drivers.
  - O(n+m) problem instead of O(n*m) problem.
  - Different device supports can share same driver.
- Porting to EPICS 3.14. need to be done only once.
- “Standard” drivers already done.
  - Local serial bus
  - TCP and UDP sockets
  - several GPIB drivers, including LAN/GPIB interfaces

# Current status




- Several device supports converted.
- Many drivers converted.

# Driver architecture



## Vocabulary: Port

- Communication path ("bus") with unique name.
- One or many devices can be connected.
- May have addresses to identify individual devices.
- May be blocking or non-blocking.
- Is configured in startup script.

```
drvAsynSerialPortConfigure "COM2", "/dev/sttyS1"
drvAsynIPPortConfigure "fooServer", "192.168.0.10:40000"
vxi11Configure "LanGpib1", "192.168.0.1", 1, 1000, "hpib"
myDeviceDriverConfigure "portname", parameters
```

## Vocabulary: Interface

- API for a class of ports.
  - common, message based, register based, …
- Defines table of driver functions ("methods")
- Does not implement driver methods.
- Every port has one or many interfaces.
- Clients talk to interfaces, not to drivers.

```
pasynCommon->connect()
pasynOctet->write()
```

# Vocabulary: Driver

- Software to handle one type of ports.
- Implements one or many interfaces.
  - Provides method tables for interfaces.
  - Has internal knowledge about specific port hardware.
- Does not handle any specific device type!
- Examples:
  - serial bus, VXI-11, Green Springs IP488, …
- Configure function in startup script connects driver to port.

# Vocabulary: asynUser

- Identifies the client.
- Each client needs one asynUser.
- From asynDriver's point of view, asynUser *is* the client.
- "Handle" to ports and everything else inside asynDriver.

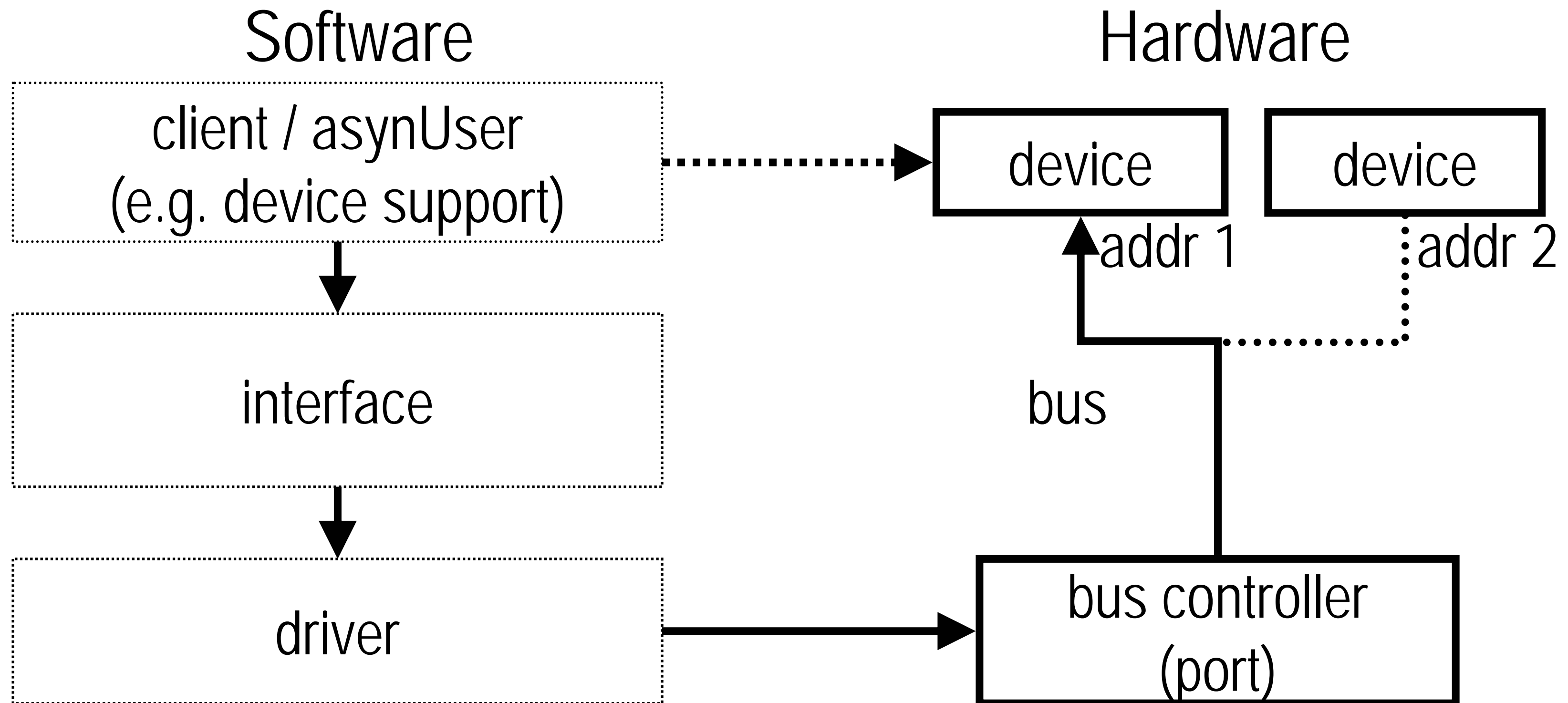
## Vocabulary: asynManager

- Core of asynDriver.
- Creates threads for blocking ports.
- Registers and finds ports and interfaces.
- Schedules access to ports.
- There is exactly one global instance: `pasynManager`
- Clients ask asynManager for services

```
pasynManager->connectDevice(pasynUser , "portname", address)
pasynManager->findInterface(pasynUser, interfaceType, ...)
pasynManager->queueRequest(pasynUser, priority, timeout)
```

- Drivers inform asynManager about any important things.

## Abstraction Layers




- Client knows nothing about port and driver.

# Basic asynDriver interfaces

- asynOctet
  - Message based I/O: serial, GPIB, telnet-like TCP/IP, …
- asynUInt32Digital
  - Bit field registers: status word, switches, …
- asynInt32, asynInt32Array
  - Integer registers: ADC, DAC, encoder, …
  - Integer arrays: spectrum analyzer, oscilloscope, …
- asynFloat64, asynFloat64Array
  - Floating point registers and arrays

# More interfaces

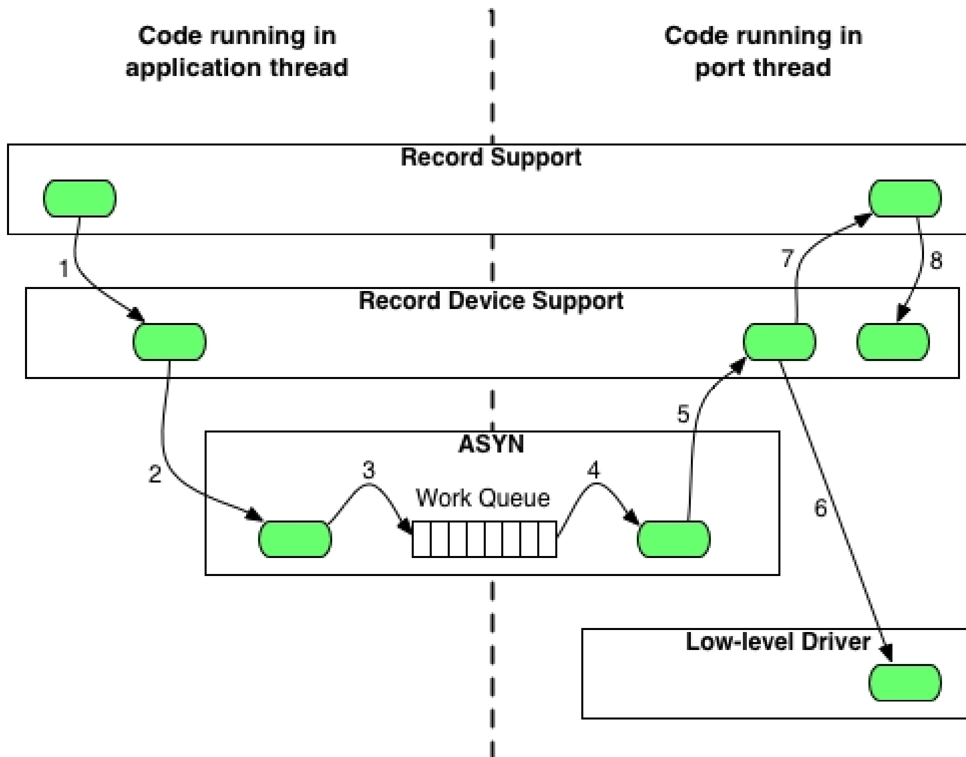
- asynCommon
  - Mandatory for every driver
  - Methods: report, connect, disconnect
- asyn*SyncIO
  - Interfaces for clients which are willing to block
    - Shell commands.
    - SNL and C programs with separate threads.
- asynGpib
  - Additional features which are not included in asynOctet:
    SRQ polling, IFC, REN, addressed and universal commands, …

# Notes about register based interfaces

- Hardware registers may be smaller/larger than Int32 / Float64
  - Driver is responsible for conversion.
  - Higher bits may be ignored / padded.
  - Larger registers may be split or implemented as arrays.
- What does port and address mean here?
  - Device and register number.
- What is an array register?
  - Something that holds a waveform.
  - May be implemented e.g. as many registers or as a fifo.
  - Driver is responsible for conversion to/from array of Int32 / Float64.

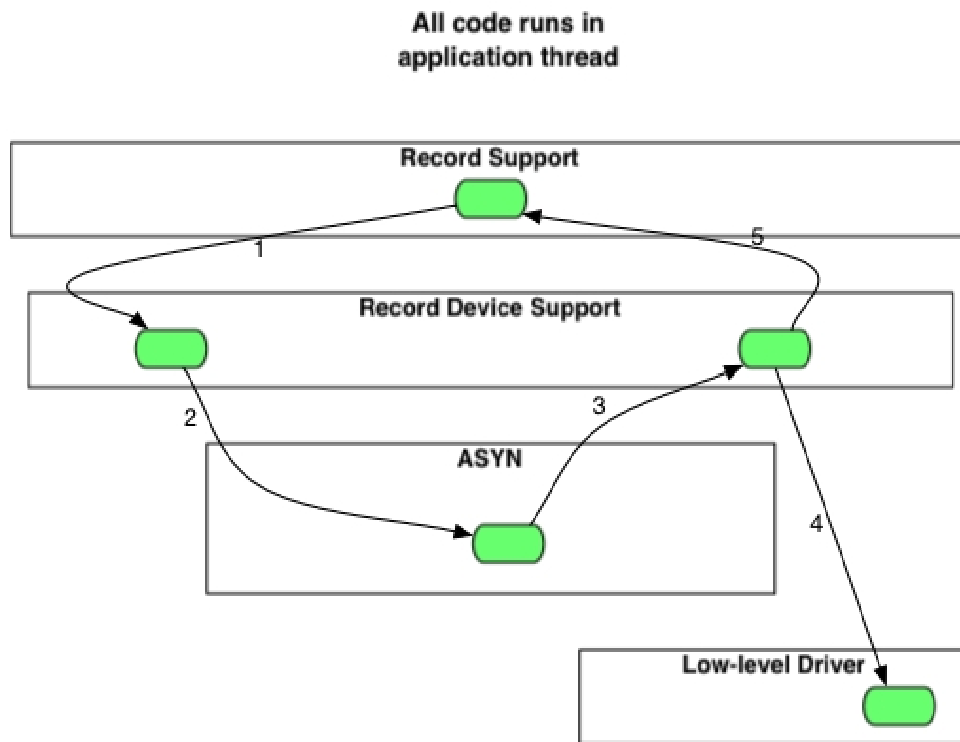

Dirk Zimoch, 2007

# Control flow for blocking port




- Client requests service and provides callback.
- Port thread calls callback when client is scheduled.
- Clients can call (even blocking) driver functions.
- No other client of same port can interfere during callback.

# Control flow for non-blocking port




- Client requests service and provides callback.
- Callback is called immediately.
- Clients can call (non-blocking) driver functions.
- No other client of same port can interfere during callback.

# Blocking and non-blocking ports

- Ports with a field bus attached are usually blocking.
  - Access to hardware may have arbitrary long delays.
  - Client must be willing to block or must use callbacks.
    - Scan tasks are not allowed to block.
    - SNL, shell functions, or other code may block.
  - Driver must have separate port thread to do actual I/O.
  - Device support is asynchronous.
- Ports which access local registers are usually non-blocking.
  - Access to hardware has only very short delays.
  - Device support is synchronous.

# Break

Coming soon: asynDriver clients (device support, etc.)

# Device example

- RS232 and/or TCP/IP device.
- Interface is asynOctet
  - Local serial connection or telnet-style TCP/IP
  - Good news: Drivers already exist.
- Clients
  - Command line functions.
  - General purpose debug record: asynRecord
  - Simple device supports for stringin, waveform, …
  - Complicated device support with string parsing: StreamDevice
  - Good news: All this already exists.

# asynOctet command line functions

- Create / destroy handle

```
asynOctetConnet(handle, port, address=0,
    timeout=1.0, buffersize=80)
asynOctetDisconnect(handle)
```

- Talk to device

```
asynOctetWrite(handle, string)
asynOctetRead(handle)
asynOctetWriteRead(handle, string)
asynOctetFlush(handle)
```

- Set / get terminators

```
asynOctetSetInputEos(port, address, eos)
asynOctetGetInputEos(port, address)
asynOctetSetOutputEos(port, address, eos)
asynOctetGetOutputEos(port, address)
```

# Example: asynOctet command line functions

```
drvAsynSerialPortConfigure "COM1", "/dev/ttyS0"
asynSetOption "COM1", -1, "baud", "9600"
asynSetOption "COM1", -1, "bits", "8"
asynSetOption "COM1", -1, "parity", "none"
asynSetOption "COM1", -1, "stop", "1"
asynOctetSetInputEos "COM1", 0, "\r\n"
asynOctetSetOutputEos "COM1", 0, "\r"
asynOctetConnet "Dirk","COM1"
asynOctetWriteRead "Dirk","value?"
asynOctetDisconnect "Dirk"
```

# More command line functions

- Report

```
asynReport(level, port)
```

- Driver and port options

```
asynSetOption(port, addr, key, value)
asynShowOption(port, addr, key)
asynAutoConnect(port, addr, yesNo)
asynEnable(port, addr, yesNo)
```

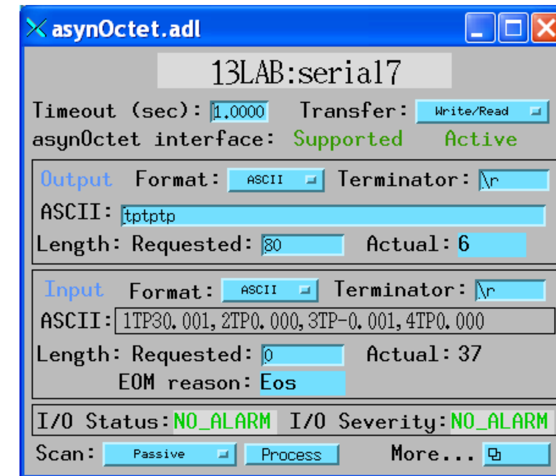
- Tracing (debugging)

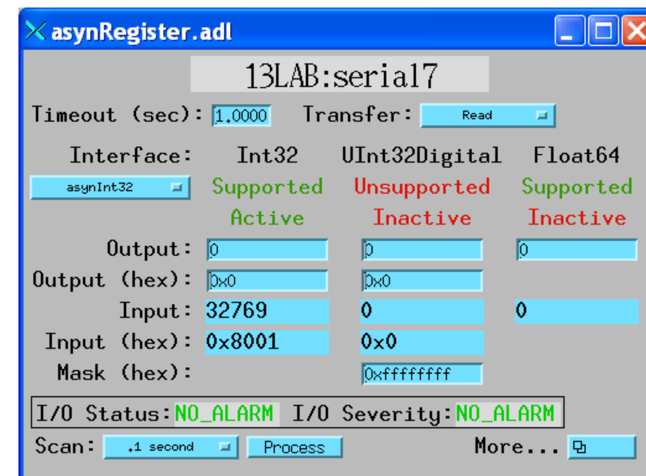
```
asynSetTraceFile(port, addr, filename)
asynSetTraceMask(port, addr, eventmask)
asynSetTraceIOMask(port, addr, formatmask)
```

## asynRecord

- Special record type that can use all asyn interfaces.
- Can connect to different ports at run-time.
- Can change any setting of all interfaces types.
- Is a good debug tool.
- Access to options including tracing.
- Comes with set of medm screens for different interfaces.
- Can only handle simple devices:
  - e.g. asynOctet: write one string, read one string
- Is all you need (more than you want?) for simple devices.

# asynRecord medm screens



## Standard record asyn device supports

- asynOctet support for stringin, and stringout, waveform
  - Can do simple write/read of strings
- Register support for ao, ai, bo, bi, mbboDirect, mbbiDirect, mbbo, mbbi, longout, longin, waveform
  - Can do simple register write, register read.
  - Interrupt can be used for “I/O Intr” scanning.
- Can handle only simple devices
- But for simple devices, that's all you need.

# Example: Records

- Asyn record

```
record (asyn, "$(P):asyn") {
    field (PORT, "TS")
}
```

- String records

```
record (stringout, "$(P):command") {
    field (FLNK, "$(P):reply")
}
record (stringin, "$(P):reply") {
    field (DTYP, "asynOctetWriteRead")
    field (INP,  "@asyn(TS,-1,1000) $(P):command")
}
```

# StreamDevice

- Device support for standard records and asynOctet ports.
- Suitable for medium complicated protocols and string parsing.
- Communication protocol is specified in plain text file
  - Big difference to devGpib: No need to recompile anything to support new device.
- String formatting and parsing similar to printf/scanf, but with much more converters, e.g. bitfield, BCD, enum, raw, …
- Checksum support.
- StreamDevice is not part of the asynDriver package.
  See: epics.web.psi.ch/software/streamdevice/


Dirk Zimoch, 2007

# Example: StreamDevice protocols

```
setValue { out "VALUE %.3f"; }
getValue { out "VALUE?"; in "VALUE=%f"; }
getStatus { out "STAT?"; in "STAT=%B.!"; } # bits: .=0 !=1
setSwitch { out "SWITCH %{OFF|ON}"; # enumeration
   @init {out "SWITCH?"; in "SWITCH=%{OFF|ON}"; } # init record
}
getDataWithEcho {out "DATA?"; in "DATA?"; in "%d"; }
writeCalcoutFieldsWithChecksum {
   out "A=%(A)g B=%(B)g C=%(C)g D=%(D)g %0<CRC32>";
}
read2Values { out "get"; in "%f %(OtherRecord.VAL)f"; }
```

# Exercise (before break)

- TCP device on port 40000
  - First connect with telnet: `telnet localhost 40000`
- Serial device on local port (/dev/ttyS0 or /dev/ttyUSB0)
  - First connect with minicom: `xterm -e minicom &`
- Find out what the device does
  - Try command HELP.
- Try asynRecord and asyn device support.
  - Softioc is in directory ioc
  - medm for asynRecord displays is installed
- Try StreamDevice support.


Dirk Zimoch, 2007

# Break

Coming soon: writing your own device support

# Writing your own device support

- If your device is too complicated, you have to – and you can write your own device support.
- It works smoothly together with other supports, even when talking to the same device!
  - You can write your own support for the complicated stuff only and leave the simple functions to existing supports.
- Also SNL or C-code can directly access the device without disturbing any records using the same port or even the same device.

## Step 1: Connect to the port

- Before doing anything you must become an `asynUser`

```
pasynUser=pasynManager->createAsynUser(processCallback,
   timeoutCallback);
```

  - Provide 1 or 2 callbacks, first is called when you are scheduled to access the port, second is called on timeout.

- Connect to the device (port, address)

```
status=pasynManager->connectDevice(pasynUser, port, addr);
```

- Get the interface you need (e.g. asynOctet)

```
pasynInterface=pasynManager->findInterface(pasynUser,
   asynOctetType, 1);
pasynOctet=(asynOctet *)pasynInterface->pinterface;
pvtOctet=pasynInterface->drvPvt;
```

# Step 2: Request access to the port

- Ask asynManager to put yur request to the queue

```
status=pasynManager->queueRequest(pasynUser, priority,
   timeout);
```

  - Priorities: `asynQueuePriority{Low|Medium|High}`
  - `queueRequest` never blocks.
  - Blocking port: AsynManager will call your `processCallback` when port is free. The callback runs in port thread.
  - Non blocking port: `queueRequest` calls `processCallback`.
  - If port is not free for `timeout` seconds, asynManager calls `timeoutCallback`. This callback runs in timer thread.
  - In `processCallback`, you have exclusive access to the port.

# Step 3: `processCallback` (asynOctet)

- Flush (discard old input)

```
status=pasynOctet->flush(pvtOctet, pasynUser);
```

- Write (with/without output eos appended)

```
status=pasynOctet->write[Raw](pvtOctet, pasynUser, data,
   size, &bytesWritten);
```

  - Actual number of written bytes is returned in `bytesWritten`.

- Read (with/without input eos handling)

```
status=pasynOctet->read[Raw](pvtOctet, pasynUser, buffer,
   maxsize, &bytesReceived, &eomReason);
```

  - Actual number of written bytes is returned in `bytesReceived`.
  - End of message reason is returned in `eomReason`.

## Step 3: `processCallback` (asynInt32)

- Get bounds

```
status=pasynInt32->getBounds(pvtInt32, pasynUser, &low,
   &high);
```

  - Limits for valid register values are returned in `low` and `high`.

- Write

```
status=pasynInt32->write(pvtInt32, pasynUser, value);
```

- Read

```
status=pasynInt32->read(pvtInt32, pasynUser, &value);
```

  - Current register value is returned in `value`.

# Step 3: `processCallback` (asynUInt32Digital)

- Write

```
status=pasynUInt32Digital->write(pvtUInt32Digital,
    pasynUser, value, mask);
```

  - Only bits specified by mask are modified.

- Read

```
status=pasynUInt32Digital->read(pvtUInt32Digital,
    pasynUser, &value, mask);
```

  - Current register value & `mask` is returned in `value`.

## Rules for using driver methods

- Never use I/O methods outside `processCallback`.
- Only talk to the port that has called you back.
- You can do as many I/O as you like.
- You always must use the interface method table `pasyn{Octet|Int32|...}` to access the driver.
- You always need `pvt...` and `pasynUser` as arguments.
- All other clients of the same port (even with other addresses) have to wait until you are finished. This is not nice of you if your device blocks for a long time!

# Allow access to other devices on same port

- Between your I/O calls, other clients can talk to other devices of the same port, if you let them.
- Lock your device.

```
status=pasynManager->blockProcessCallback(pasynUser, 0);
```

- Call only one I/O method at a time in `processCallback`.
- Commit new `queueRequest()` and finish callback.
- When done, release your device.

```
status=pasynManager->unblockProcessCallback(pasynUser, 0);
```

- This only applies to blocking devices with multiple addresses.

# Informational asynManager methods

- Write report to file

```
pasynManager->report(file, detailLevel, port);
```

  - Can be called without asynUser in any context.

- Get information about port.

```
status=pasynManager->isMultiDevice(pasynUser, port, &yesNo);
```

  - Can be called before connected to port.

- Get information about connected port.

```
status=pasynManager->canBlock(pasynUser, &yesNo);
status=pasynManager->isEnabled(pasynUser, &yesNo);
status=pasynManager->isConnected(pasynUser, &yesNo);
status=pasynManager->isAutoConnect(pasynUser, &yesNo);
```

## More asynManager methods

- Cleanup

```
status=pasynManager->disconnect(pasynUser);
```

  - Disconnects asynUser from port.
  - Fails when asynUser is queued or callback is active.

```
status=pasynManager->freeAsynUser(pasynUser);
```

  - `freeAsynUser` automatically calls `disconnect`.

- Cancel queued request

```
status=pasynManager->cancelRequest(pasynUser);
```

  - Blocks when callback is active.

# Interrupts

- Register for asynInt32 interrupts

```
void interruptCallbackInt32(userPvt, pasynUser, value);
status=pasynInt32->registerInterruptUser(pvtInt32,
   pasynUser, interruptCallbackInt32, userPvt,
   &intrruptPvtInt32);
status=pasynInt32->cancelInterruptUser(pvtInt32, pasynUser,
   intrruptPvtInt32);
```

- Similar for other interfaces

```
void interruptCallbackOctet(userPvt, pasynUser, data, size,
   eomReason);
```

- Callbacks do not run in interrupt context!
- Interface has changed in asynDriver version 5.0.


Dirk Zimoch, 2007

# Remarks on device supports

- Always check return value of methods

```
typedef enum {asynSuccess, asynTimeout, asynOverflow,
   asynError} asynStatus;
```

- If port can block you must implement asynchronous support.
  - Set `precord->pact=1` before `queueRequest`.
  - Return after `queueRequest` and wait for callback.
  - In your callback call `callbackRequestProcessCallback`.
  - Update record in second processing run.
- If port cannot block you can implement synchronous support.
  - Update record after `queueRequest` and return.

# Writing blocking clients

- Clients which run in a private thread may use synchonous (i.e. blocking) interfaces.
- Examples: Shell functions, SNL code, custom C code.
- No need to use callbacks.
- No need to know about asynManager.
- **Never use this from scan threads**, i.e. in device supports!
- There is one global interface instance for each synchronous interface type.

# asynOctetSyncIO

- Create asynUser and connect to port

```
status=pasynOctetSyncIO->connect(port, addr, &pasynUser,
   driverInfo);
```

- Blocking I/O methods

```
status=pasynOctetSyncIO->write[Raw](pasynUser, data, size,
   timeout, &bytesTransfered);
status=pasynOctetSyncIO->read[Raw](pasynUser, buffer,
   maxsize, timeout, &bytesReceived, &eomReason);
status=pasynOctetSyncIO->flush(pasynUser);
```

- Disconnect from port and free asynUser

```
status=pasynOctetSyncIO->disconnect(pasynUser);
```

# asynOctetSyncIO convenience methods

- Connect, write, disconnect

```
status=pasynOctetSyncIO->write[Raw]Once(port, addr, data,
   size, timeout, &bytesTransfered, driverInfo);
```

- Connect, read, disconnect

```
status=pasynOctetSyncIO->read[Raw]Once(port, addr, buffer,
   maxsize, timeout, &bytesReceived, &eomReason,
   driverInfo);
```

- Connect, write, read, disconnect

```
status=pasynOctetSyncIO->writeReadOnce(port, addr, data,
   size, buffer, maxsize, timeout, &bytesTransfered,
   &bytesReceived, &eomReason, driverInfo);
```

## Other syncIO interfaces work similar

- Create asynUser and connect to port.
- Blocking I/O methods analogous to asynchonous interface.
- Disconnect and destroy asynUser.
- Convenience methods: Connect, I/O, disconnect.

- For more details see interface description in asynDriver documentation:
  - www.aps.anl.gov/epics/modules/soft/asyn/R4-7/asynDriver.html

# Break

Coming soon: low-level asynDrivers

# Writing asyn drivers

- First look if your port hardware is already supported.
- Remember: This is about ports not devices!
  - A local bus controller card is a port, e.g. CANbus card, GPIB card
  - A network device is a port, e.g. telnet-style TCP, VXI-11
  - An oscilloscope connected via GPIB is not a port!
  - What about VME-bus I/O cards? ADCs, Encoders, …
    - You can write a port driver for that card, but…
    - Better spend the effort to write a general purpose VME-register driver.
    - Put the intelligence into device support, not port driver.

# Which interfaces should be implemented?

- asynCommon: a must

  `report(), connect(), disconnect()`

- asynOctet: if port provides multi-byte messages (text)

  `write(), read(), writeRaw(), readRaw(), flush(), setInputEos(), getinputEos(), setOutputEos(), getOutputEos(), registerInterruptUser(), cancelInterruptUser()`

- asynGpib (in addition to asynOctet): if port is GPIB

  `addressesCmd(), universalCmd(), ifc(), ren(), ...`

- Register interfaces: if port provides "active variables"

  `write(), read(), registerInterruptUser(), cancelInterruptUser(), getBounds(), setInterrupt(), clearInterrupt()`

# Should I define my own interface type?

- No.
- Yes, if your port needs special methods
  - You have do define your own port type with a set of methods.
  - Keep it as generic as possible, not a class with only one member!
  - Is it really not possible to use a combination of standard interfaces?
  - Is asynMotor a candidate?

# Step 1: Define private data structure

- Structure must contain everything you need to operate a port.
- Each port instance has its own structure.
  - There may be more than one instance at a time.
  - Avoid global variables. Put everything into your structure.
  - User will see this structure as `drvPvt`.
  - All your methods get `drvPvt` as first argument. Cast it back to a pointer to your private structure.
- For each interface, put in one asynInterface structure.
- Put in method tables.

## Step 2: Write driver methods

- Implement all methods for all interfaces you want to support.
  - Most interfaces have a "base class" which already provides default implementations for some methods.
  - Your methods can be (should be) static. Nobody will ever access them execpt via the interface function table.
- Write a useful `report()` method.
  - Users want to know: name of your driver, addresses, connection status, interrupts, any internals that may help to identify problems!
  - Use the `detail` argument to filter the amount of information. Report just driver name and summary for level 0.

# Step 2: Write driver methods (cont'd)

- Write `connect()` method
  - Open conenction to actual device, get handle from 3rd party software or similar.
  - For multi-devices, call `pasynManager->getAddr()`.
  - Return `asynError` if device is already connected.
  - Setup connection and/or device.
  - Call `pasynManager->exceptionConnect()`.
  - Every device (port/address) is connected only once at a time, even when many asynUsers use it. The provided asynUser is the first one that uses this device.

# Step 2: Write driver methods (cont'd)

- Write `disconnect()` method
  - Close conenction to actual device, free handle from 3rd party software or similar.
  - For multi-devices, call `pasynManager->getAddr()`.
  - Return `asynError` if device is not connected.
  - Cleanup device and/or connection.
  - Call `pasynManager->exceptionDisconnect()`.

# Step 3: Write configuration function

- This function is called in the startup script to set up the port.
- Give it a useful and specific name
  - Not just `portInit` or `configure`.
  - Examples: `drvAsynSerialPortConfigure`, `drvAsynIPPortConfigure`, `vxi11Configure`
- Export it to iocsh.
- First argument should be port name.
- Give useful default values to as many arguments as possible.
- Check all arguments! People write stupid stuff in startup scripts.

# Configuration function: Fill private structure

- Allocate and fill private structure with everything you need to operate the port.
  - Mutexes, timers, other resources.
- Fill asynInterface structures in your private structure.
  - Fill `interfaceType`: what type of interface is it?
  - Fill `pinterface`: pointer to your method table.
  - Fill `drvPvt`: pointer to your private structure.
- Fill method tables with pointers to your methods.
  - Base interfaces provide `initialize()` method to fill method table with default implementations.

# Configuration function: Register to asynManager

- Call `pasynManager->registerPort()`.
  - This tells asynManager if port has multiple addresses, if port can block and if autoConnect is enabled.
- For each supported interface call `pasynManager->registerInterface()`.
- For each interface that generates interrupts call `pasynManager->registerInterruptSource()`.
  - Interrupt may actually be implemented as poll thread or any type of event handler.
  - It means just: new data has arrived asynchronously

# Step 4: Write interrupt handler (optional)

- Details strongly depends on implementation
  - Connect handler to hardware interrupt.
  - Create thread that polls hardware periodically.
  - Register to event system of 3rd party software.
- Call `pasynManager->interruptStart()`.
  - You get a list of clients which have subscribed for this interrupt.
- For each client, call interrupt callback and provide value.
- Call `pasynManager->interruptEnd()`.

## Advanced concepts

- Exceptions
  - Users can subscribe for special events, e.g. connect/disconnect.
- Interpose interfaces
  - Additional transparent layers can be put between port and user.
  - These layers can pre/post process data.
  - asynOctet terminators (eos) are implemented this way.
- asynOption: Port options (key, value pairs)
  - Example: baud rate, parity, etc for serial port.
- asynDrvUser: Named driver resources

# Examples of port drivers in asyn package

- asynOctet / asynGpib drivers
  - asyn/drvAsynSerial/
  - asyn/vxi11/
  - asyn/ni1014/
  - asyn/gsIP488/
  - asyn/linuxGpib/
- register driver examples
  - testEpicsApp/src/

# More information

- AsynDriver
  - www.aps.anl.gov/epics/modules/soft/asyn/
- StreamDevice
  - epics.web.psi.ch/software/streamdevice/
- linuxGpib
  - linux-gpib.sourceforge.net/
- Drivers/device supports using asynDriver
  - www.aps.anl.gov/aod/bcda/synApps/
- Talks about asynDriver
  - www.aps.anl.gov/aod/bcda/epicsgettingstarted/iocs/ASYN.html
  - www.aps.anl.gov/epics/docs/USPAS2007.php