ASYN/StreamDevice
Support Frameworks

Eric Norum
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.
asyn Architecture

Device support (or SNL code, another driver, or non-EPICS software)

Port (named object)

Port driver

Interfaces (named; pure virtual functions)

asynCommon (connect, report, …)

asynOctet (write, read, setInputEos, …)

addr=0

device

device

addr=1
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, USB-TMC.
- 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 or 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
- If your IOC uses ASYN it should provide at least one asynRecord to give clients control of diagnostic messages!
asynRecord – asynOctet devices

Interactive I/O to serial device

Configure serial port parameters

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
    - ASYN_TRACE_WARNING
- Tracing is enabled/disabled for (port/addr)
- Trace messages can be turned on/off from iocsh, vxWorks shell, and from CA clients such as EDM via asynRecord
- asynOctet I/O from shell
Typical source file arrangement

- Instrument support is placed in
  
  .../modules/instrument/<instrumentname>/Rx.y/

- Each <instrumentname>/Rx.y/ directory contains at least

  Makefile
  configure/
  <InstrumentName>Sup/
  documentation/
  License
Script to make this a little easier

- `mkdir /.../modules/instrument/myinst/head`
- `cd /.../modules/instrument/myinst/head`
- `/<path to ASYN support module>/bin/<arch>/makeSupport.pl` 
  `-t streamSCPI myinst`

  Makefile
  configure/...
  myinstSup/
    Makefile devmyinst.db devmyinst.proto
  documentation/
    devmyinst.html

- A few changes to the latter 3 files and you’re done!
- Notice that there are no C or C++ files.
  - Running `make` just copies the `.db` and `.proto` files to the support module top-level `db/` directory.
Introduction to Stream Device

• Generic EPICS device support for devices with “byte stream” communication.
  – RS-232 (Local serial port or LAN/Serial adapter)
  – TCP/IP
  – VXI-11
  – GPIB (Local interface or LAN/GPIB adapter)
  – USB-TMC (Test and Measurement Class)

• A single stream device module can serve to communicate using any of the above communication mechanisms.
Introduction to Stream Device

- Command/reply messages:
  - *IDN?
  - SET:VOLT 1.2
  - Non-ASCII ‘strings’ too

- Command generation and reply parsing configured by protocols

- Formatting and interpretation handled with format converters
  - Similar to C printf/scanf format strings
  - Custom converters too, but not easy
Stream Device *Protocols*

- Defined in *protocol files*
- Plain ASCII text file
- No compiling – IOC reads and interprets protocol file(s) at startup
- Protocols are linear
  - No looping
  - No conditionals
  - Rudimentary exception handlers
- A single entry can read/write multiple fields in one or many records
- Output records can be initialized from instrument at IOC startup
  - With one big caveat – instrument must be on and communicating at IOC startup
StreamDevice EPICS Database

record(bo, "$(P)$(R)CLS") {
    field(DESC, "SCPI Clear status")
    field(DTYP, "stream")
    field(OUT, "@devmyInst.proto cmd(*CLS) $(PORT) $(A)")
}
record(longin, "$(P)$(R)GetSTB") {
    field(DESC, "SCPI get status byte")
    field(DTYP, "stream")
    field(INP, "@devmyInst.proto getD(*STB) $(PORT) $(A)")
}

• DTYP=stream
• INP/OUT fields specify protocol file name, protocol entry (with optional arguments), ASYN port and address.
• Address can be any value (typically 0) for single-address interfaces.
StreamDevice Protocol File

```cpp
cmd {
    out "\$1";
}
getD {
    out "\$1?";
    in "%d";
}
```

- Protocol entries contain statements to produce output and request input
- C-style escape sequence can be used (\r, \n, \033, \e)
- Format converters are similar to those used by C printf/scanf
  - By default the VAL or RVAL field is used as the data source/destination
- Can refer to any field, even in another record
StreamDevice Additional Records

DTYP ≠ stream for protocol entry additional records:

record(stringin, "$(P)$(R)Serial")
{
    field(DESC, "Serial number")
    field(DTYP, "Soft Channel")
}
record(ai, "$(P)$(R)VP5")
{
    field(DESC, "+5V supply")
    field(DTYP, "Raw Soft Channel")
    field(EGU, "V")
    field(PREC, "3")
    field(LINR, "SLOPE")
    field(ESLO, "1e-3")
    ...
record(longin, "$(P)$(R)Temp1")
{
    field(DESC, "Sensor 1 temperature")
    field(DTYP, "Soft Channel")
}
StreamDevice Protocol File

Protocol entries can be long – Use multiple lines and string concatenation to improve readability

```java
query {
    out "Q";
    in " :
        "SN=%($1Serial.VAL)39[^,],",
        "UN=%($1Name.VAL)39[^,],",
        "IP=%[^,],",
        "V3=%d,",
        "V5=%($1VP5.RVAL)d,",
        "V+12=%($1VP12.RVAL)d,",
        "V-12=%($1VM12.RVAL)d,",
        "T1=%($1Temp1.VAL)d,",
        ...
        "POH=%($1HoursOn.VAL)g,",
        "MAXTMP=%($1MaxTemp.VAL)g;"
}

Notice the use of the width field – guard against buffer overruns!
StreamDevice Protocol File – Terminators

- Terminators can be set globally or per entry.
- Some interfaces can handle only a single character. If device replies with '\r\n' then specify `In Terminator='\n'` and ignore the '\r' in the reply.

```cpp
In Terminator = "\n";
Out Terminator = "\r";
```
StreamDevice Protocol File – Initial Readback

- Useful to set initial value of output records to match the value presently in the instrument.
- `@init` ‘exception handler’
- Often the same as the corresponding readback prototype entry

```plaintext
getF {
    out "\$1?";
    in "%f";
}

setF {
    @init { out "\$1?"; in "%f"; } 
    out "\$1 %f";
}

record(ao, "$(P)$(R)IntegrationTime")
{
    field(DESC,"Reading integration time")
    field(DTYP,"stream")
    field(OUT, "@devKeithley6487.proto setF(NPLC) $(PORT) $(A)")
}
Adding StreamDevice/ASYN instrument support to an application

- This is easy because the instrument support developers always follow all the guidelines – right?
- Most of these steps apply to pretty much any support module, not just StreamDevice/ASYN instruments.
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 the instruments and ASYN:

```
ASYN     = /usr/local/epics/R3.14.12/modules/soft/asyn/asynR4-21
EPICS_BASE= /home/EPICS/base
```
Modify the application Makefile

```makefile
xxx_DBD += base.dbd
xxx_DBD += stream.dbd
xxx_DBD += drvAsynIPPort.dbd
    (and/or drvAsynSerialPort.dbd, drvAsynUSBTMC.dbd, etc.)
xxx_DBD += asyn.dbd

xxx_LIBS += stream asyn
```
Modify the application database Makefile

Copy the instrument support database and prototype files to the application `<top>/db/` directory:

```
DB_INSTALLS  +=  $(DAWN_RUSH)/db/devDawnRuSH.db
DB_INSTALLS  +=  $(DAWN_RUSH)/db/devDawnRuSH.proto
```
Modify the application startup script

epicsEnvSet("CRATE_ADDRESS", "\$(CRATE_ADDRESS=crateapex01:23) \")

(above line is optional, but makes it easy to override for testing)

epicsEnvSet("STREAM_PROTOCOL_PATH", "\${TOP}/db")

;

drvAsynIPPortConfigure("CR0", "\$(CRATE_ADDRESS) TCP", 0, 0, 0)

;

dbLoadRecords("db/devDawnRuSH.db", "P=apexCrate:, R=1:, PORT=CR0")

• P,R – PV name prefixes – PV names are \$(P)$(R)name
• PORT– ASYN port name from corresponding devxxxConfigure command
Lab Session
Control ‘network-attached device’

- Host www.xxx.yyy.zzz – TCP Port 24742
- ‘\n’ command terminator, ‘\r\n’ reply terminator
- *IDN?*
  - Returns device identification string (up to 100 characters)
- LOAD?
  - Returns three floating-point numbers separated by spaces (1, 5, 15 minute load average)
- ON?
  - Returns OFF/ON (0/1) status
- VOLTS?
  - Returns most recent voltage setting
- CURR?
  - Returns current readback (±11A)
Lab Session
Control ‘network-attached device’

- **ON** [0, 1]
  - Turns supply OFF/ON (0/1)
- **VOLTS** x.xxxx
  - Sets voltage (±10V range)