EPICS: ASYNCHRONOUS DRIVER SUPPORT*

Martin R. Kraimer[1], Mark Rivers[2], Eric Norum[1]

[1] *Argonne National Laboratory, Argonne Illinois 60439 USA*

[2] *The University of Chicago, Chicago Illinois 60637 USA*

ABSTRACT

asynDriver [1] is a general-purpose facility for interfacing device-specific code to low-level drivers. asynDriver supports nonblocking device support that works with both blocking and nonblocking drivers. A primary target for asynDriver is EPICS device support, but much of it is independent of EPICS. This paper presents an overview of asynDriver and the range of devices it supports.

BACKGROUND

The Experimental Physics and Industrial Control System (EPICS) [2] is a set of software tools, libraries, and applications developed collaboratively and used worldwide to create distributed, soft, real-time control systems for instruments such as particle accelerators, telescopes, and other large scientific experiments. EPICS development began around 1990, and over the years, support has been developed for a large variety of hardware devices. This support usually consists of a driver, which communicates with the hardware, and device support for EPICS records. These two components are often linked by a custom-defined private interface. Generic support, in fact several different implementations, has been developed for serial and GPIB devices. The idea of creating asynDriver arose from the desire to provide a more structured environment for developing support for hardware devices and was the result of many years' experience writing EPICS device and driver support. The initial version of asynDriver supported only asynchronous device communication, i.e., communication that causes the requester to wait. When the synApps [3] X-ray beamline control software package was converted to use asynDriver, support for synchronous device communication was added. asynDriver is now a framework for both asynchronous and synchronous communication with hardware devices.

OVERVIEW

A primary target for asynDriver is EPICS device support, but much of it is independent of EPICS.

asynDriver is based upon the following key concepts:

- *Device/Driver communication is made through standard interfaces*. Drivers take care of the details of how to communicate with a device and implement interfaces for use by device support. Interfaces are defined for both message- and register-based devices.
- *A port provides access to device instances.* A port, which has a unique port name, identifies a communication path to one or more device instances.
- *asynManager controls access to a port.* asynManager, a component of asynDriver, provides access to a driver via calls to queueRequest or lockPort/unlockPort. Once device support has access, it can make an arbitrary number of calls to the driver knowing that no other support can call the driver. Device and driver support do not need to implement queues or semaphores since asynManager does this for them. This makes driver development much easier since the driver operates in a single-threaded environment.
- *Connection Services* – Each port and each device on a port can be individually enabled or disabled. A port driver can be commanded to connect to or disconnect from a port or a device. asynManager can be requested to automatically connect when device support attempts to access a port or device that is not connected.
- *asynTrace – General-purpose diagnostic facility.* Rules are defined for providing diagnostic messages. A user can specify trace masks for diagnostic information to be displayed on the console, written to a file, or sent to the EPICS error logging facility.

* Work supported by U.S. Department of Energy, Office of Basic Energy Sciences, under Contract No. W-31-109-ENG-38.

- *asynRecord – Generic access to a device/port.* asynRecord provides an EPICS record and associated MEDM displays that give access to:
  - ♦ A port or a device (address on a port) connected to a port. The port or port and address can be changed dynamically. Thus, with a single asynRecord in an IOC it is possible to talk to any device that has an asyn compatible driver.
  - ♦ asynTrace – All asynTrace options can be specified via asynRecord.
  - ♦ Standard interfaces – These can be used to communicate with devices. For example, if a new instrument arrives that has a serial, GPIB, or Ethernet port, then it is often possible to communicate with it just by attaching an asynRecord to it.
- *Extensive Serial Support* – asynDriver provides many facilities for communicating with RS232, RS485, GPIB, and Ethernet devices
- *synApps Support* – Additional support is available via synApps.

INTERFACES

All communication between software layers is done via interfaces. An interface is a C language structure consisting entirely of function pointers. An asynDriver interface is analogous to a C++ or Java pure virtual interface. Although the implementation is in C, the spirit is object oriented. Thus, this document uses the term “method” rather than “function pointer”.

Interfaces are divided into two classes: Message and Register. Message interfaces are used for devices that use command/response communication, such as serial and GPIB devices. Register interfaces are used for devices such as VME and PCI bus modules that provide access via registers.

The only Message interface in asyn is asynOctet, which uses octet (8-bit bytes) strings for sending and receiving messages.

There are several Register interfaces:

- Int32 – Registers appear as 32-bit integers. Common examples of devices supported by this interface are analog-to-digital converters and digital-to-analog converters.
- UInt32Digital – Registers appear as 32-bit integers and masks select-specific bits. This supports many digital I/O devices.
- Float64 – Registers appear as 64-bit IEEE floating point numbers.
- Int32Array – Registers appear as an array of 32-bit integers.
- Float64Array – Registers appear as an array of 64-bit IEEE floating point numbers.

devEpics, a component of asynDriver, provides device support for many standard EPICS records. devEpics communicates with port drivers via the standard interfaces mentioned above. Thus, if a driver implements one of the standard interfaces, it can be used without writing any EPICS device support.

PORT – PROVIDES ACCESS TO DEVICE

Figure 1 shows how device support communicates with an asynchronous device.

The sequence of operations is:

1. Record processing calls device support.
2. Device support calls queueRequest.
3. queueRequest places the request on the driver work queue. The application thread is now able to go on and perform other operations.
4. The portThread removes the I/O request from the work queue.
5. The portThread calls the processCallback located in device supported.
6. processCallback calls the low-level driver. The low-level driver read/write requests block until I/O completes.
7. processCallback requests that record support complete processing.
8. Record support calls Record Device Support again and record processing completes.

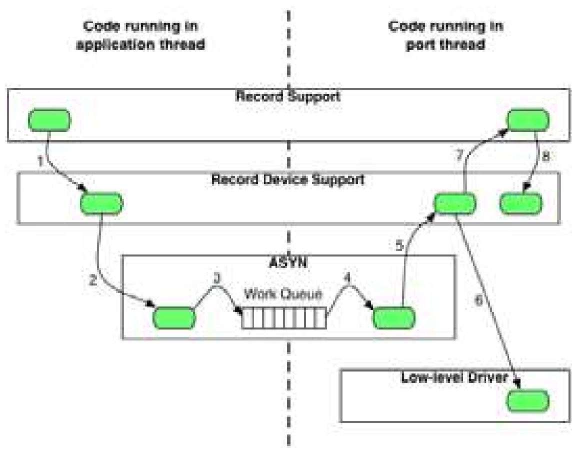


Figure 1. Device support communicating with an asynchronous device.

Figure 2 shows how device support communicates with a synchronous device:

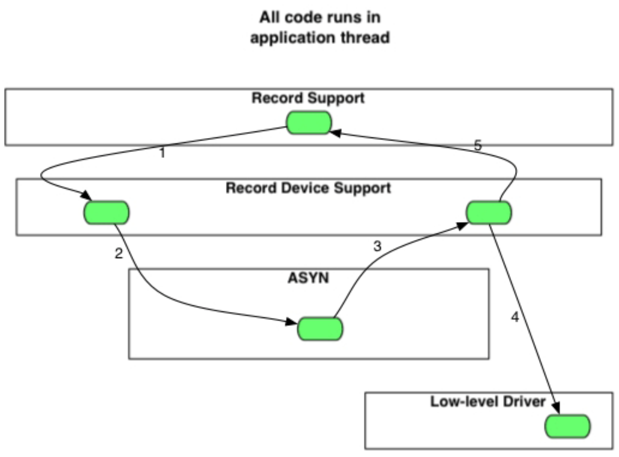


Figure 2. Device support communicating with a synchronous device.

The flow of control is:

1. Record processing calls device support.
2. Device support calls queueRequest.
3. Since the port is synchronous, i.e., can not block, queueRequest calls lockPort and then the processCallback.
4. processCallback calls the low-level driver read or write routine. The low-level driver routine returns the results of the I/O operation to processCallback.
5. processCallback returns to queueRequest, which calls unlockPort and returns to device support, which returns to record support, which completes record processing.

asynManager

asynManager is an interface and associated code. It is the "heart" of asynDriver since it manages the interactions between device support code and drivers. It provides the following services:

- *Reporting:* report
  Generate a report about a port or all ports.
- *asynUser Creation:* createAsynUser, ...
  An asynUser is a "handle" for access to asynManager services and drivers. The call to createAsynUser specifies a processCallback, which is the callback that is called as a result of a queueRequest.

- *Basic asynUser Services:* connectDevice, findInterface, ...
  When device support calls connectDevice, it is connected to the port driver that communicates with the device. findInterface is called for each interface the user requires.

- *Queuing Services:* queueRuest, lockPort, unlockPort, ...
  queueRequest is a request to call processCallback. The caller does not block. queueRequest semantics differ for a port that blocks and a port that does not block.
  When registerPort is called by a driver that blocks, a thread and a set of priority-based queues are created for the port. queueRequest puts the request on one of these queues. The port thread takes the requests from the queue and calls the associated callback. Only one callback is active at a time.
  When registerPort is called by a driver that does not block, a mutex is created for the port. queueRequest takes the mutex, calls the callback, and then releases the mutex. The result in both cases is the same – the driver code appears to be running in a single-threaded environment.
  lockPort is a request to lock all access to low-level drivers until unlockPort is called. If the port blocks, then lockPort and all calls to the port driver may block. lockPort/unlockPort are provided for use by code that is willing to block. They are also called by the thread that calls processCallback.

- *Basic Driver Services:* registerPort, registerInterface
  registerPort is called by a portDriver. registerInterface is called by a portDriver or an interposeInterface. Each port driver provides a configuration command that is executed for each port instance. The configuration command performs port specific initializations and calls registerPort and registerInterface for each interface it supports.

- *Attribute Retrieval:* isMultiDevice, canBlock, getAddr, getPortName, ...
  These methods can be called by any code that has access to the asynUser to introspect the internal settings.

- *Connection Services:* enable, autoConnect
  These methods can be called by any code that has access to the asynUser. These methods can be called to set the enable and auto-connect settings for a port and/or device. queueManager implements autoConnect by calling asynCommon:connect just before it calls processCallback. It does this if autoConnect is true and a port/device is enabled but not connected.

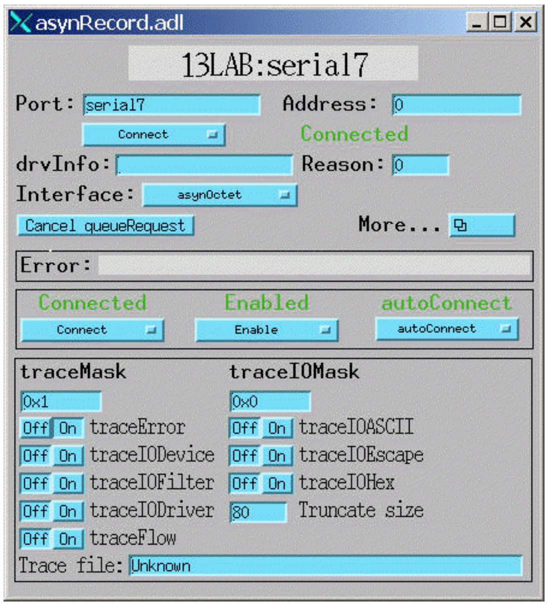
- *Exception Services:* exceptionCallbackAdd, ...
  An exception is raised when any of the following occur: 1) connection state change, 2) enable state change, 3) autoConnect state change, or 4) any change in trace options. Any code that calls exceptionCallbackAdd will be notified when an exception occurs.

- *Interrupt Services:* registerInterruptSource, ...
  Interrupt just means "I have a new value." Interfaces provide interrupt support via method registerInterruptUser. Device support registers to be called whenever an interrupt occurs. Driver support announces interrupts. Driver support together with asynManager calls all the registered device support.

- *Interpose Service:* interposeInterface
  Code that calls interposeInterface implements an interface which is either not supported by a port driver or that is "interposed" between the caller and the port driver. For example, asynInterposeEos can be interposed "above" an asynOctet driver. The asynInterposeEos interface performs end-of-string processing for port drivers that do not support it. This allows multiple port drivers to use the same interposed code. interposeInterface is recursive. An arbitrary number of interpose layers can exist above a single port and address.

AsynRecord

Figure 3 is a display connected to an asynRecord.


asynRecord.adl

13LAB:serial7

Port: serial7 Address: 0

Connect Connected

drvInfo: Reason: 0

Interface: asynOctet

Cancel queueRequest More...

Error:

Connected Enabled autoConnect

Connect Enable autoConnect

traceMask traceIOMask

0x1 0x0

Off On traceError Off On traceIOASCII

Off On traceIODevice Off On traceIOEscape

Off On traceIOFilter Off On traceIOHex

Off On traceIODriver 80 Truncate size

Off On traceFlow

Trace file: Unknown

Figure 3. Display of an asynRecord.

It has the following features:

- It is currently connected to a device with a port name of “serial7” and address 0. The user can dynamically change the Port and Address.
- The More icon provides a list of related displays that allow the user to enter commands or receive responses via standard interfaces.
- The Error box displays any error messages generated by interface calls.
- The Connect, Enable, and autoConnect fields display, and allow the user to change, connection states.
- The “Trace” field displays, and allows the user to change, the trace masks.

MESSAGE-BASED DEVICE SUPPORT

The asynDriver support includes the following components for communicating with message-based devices:

- devGpib – This is the successor to the device support layer of the Winans/Franksen GPIB device support. This support was originally implemented by John Winans (ANL). Benjamin Franksen (BESSY) extended the support to Ethernet GPIB controllers that implement the VXI-11 protocol.
- asynGpib – The successor to the drvGpibCommon layer of the Winans/Franksen support.
- drvAsynSerialPort – Generic serial support. It supports standard serial ports on Linux, RTEMS, and vxWorks
- drvAsynIPPort – Support for devices accessed via simple Ethernet TCP/IP or UDP/IP messages.
- VXI-11 - A replacement for the VXI-11 support of the Franksen support.
- linux-gpib – Support for the Linux GPIB Package library. This is provided by Rok Sanjan (cosyLab).
- gsIP488 – A low-level driver for the Greensprings IP488 Industry Pack module.
- ni1014 – A low-level driver for the National Instruments VME 1014D.
- Message Terminators – asynDriver provides extensive support for dealing with message terminators for RS232/RS485 communication.

SynApps

SynApps is a collection of software tools that help to create a control system for beamlines. The latest version uses asynDriver as the interface to most of its support hardware. It adds support for the following hardware:

- IpUnidig digital I/O (Industry Pack); supports interrupts.
- dac128V digital-to-analog converter(Industry Pack).
- Ip330 analog-to-digital converter (Industry Pack); supports interrupts.
- Canberra AIM multi-channel analyzer and ICB modules (Ethernet).
- XIA DXP DSP spectroscopy system (CAMAC, EPP, PXI).
- APS quad electrometer (VME); supports interrupts.
- epid record fast feedback (float 64 with callbacks for input, float64 for output).
- MCA fast-sweep (Int32Array with callbacks).
- Device support for a large number of serial devices such as multimeters, current amplifiers, vacuum gauges, ion pump controllers, themocouple scanners, etc.
- Device support for CCD detectors, typically communicating with socket servers.

REFERENCES

[1] http://www.aps.anl.gov/epics/modules/soft/asyn asynDriver: Asynchronous Driver Support
[2] http://www.aps.anl.gov/epics Experimental Physics and Industrial Control System
[3] http://www.aps.anl.gov/aod/bcda/synApps/index.php synApps