asynDriver

Table of Contents

Table of Contents

asynDriver: Asynchronous Driver Support

Release 4.0

Marty Kraimer, Eric Norum and Mark Rivers

November 17, 2004

License Agreement

Contents

Purpose

asynDriver is a general purpose facility for interfacing device specific code to low level drivers, asynDriver supports non−blocking device support that works with both blocking and non−blocking drivers.

A primary target for asynDriver is EPICS IOC device support but, other than using libCom, much of it is independent of EPICS.

The following are some of the existing EPICS general purpose device support systems that have been converted to use asynDriver.

- gpibCore is the operating−system−independent version of the Winans/Franksen GPIB support.
- MPFOSI (Message Passing Facility). It is no longer needed or supported since asynDriver can completely replace MPF.
- synApps (The APS BCDA synchrotron applications). The mca, dxp, serial, GPIB, Ip330, IpUnidig, DAC128V and quadEM applications in this package have all been converted to asyn.

The following are some of the existing EPICS general purpose device support systems that could be converted to use asynDriver.

- STREAMS is the protocol file−based support for serial/GPIB/CAN from Dirk Zimoch.
- devAscii/drvAscii is serial support from KECK Observatory.

Each of these systems is used at EPICS facilities for accessing GPIB and/or serial devices. Because device support has been written for many instruments and thousands of database records use the device support, users will not be easily persuaded to switch from their existing solution. Thus, asynDriver implements a framework below device support that can be used by all of the above systems so that all can share the same drivers.

Each system needs to be modified so that the device support component is compatible with existing use, but replace the driver part with asynDriver. The benefit is that all could share the same set of low level drivers.

gpibCore and mpfSerial have already been converted and are included with asynDriver.

Hopefully Dirk Zimoch will get time soon to convert STREAMS, and Allen Honey time to convert devAscii.

In the future, other protocols will be supported, especially for Ethernet based devices.

Status

This version provides

- asynManager: the software layer between device support and drivers.
- asynRecord: EPICS record support that provides a generic interface to asynManager, asynCommon, asynOctet, asynGpib, and other interfaces.
- standard interfaces: Standard message and register based interfaces are defined. Low Level Drivers implement standard interfaces. Device support communicates with low level drivers via standard interfaces.
- devEpics: Generic device support for EPICS records.
- devGpib: EPICS device support that replaces the device support layer of the Winans/Franksen gpibCore support.
- asynGpib: a replacement for the drvGpibCommon layer of the Franksen gpibCore support.
- drvAsynSerialPort: Support for devices connected to serial ports.
- drvAsynIPPort: Support for devices connected to devices connected through Ethernet/Serial converter boxes, TCP/IP sockets or UDP/IP sockets.
- VXI−11: A replacement for the VXI−11 support of the Franksen gpibCore support.
- Linux−gpib: Support for the Linux GPIB Package library.
- gsIP488: A low level driver for the Greensprings IP488 Industry Pack module.
- ni1014: A low level driver for the National Instruments VME 1014D.

Acknowledgments

The idea of creating asynDriver resulted from many years of experience with writing device support for serial and GPIB devices. The following individuals have been most influential.

John Winans
: John provided the original EPICS GPIB support. Databases using John's support can be used without modification with devGpib. With small modifications, device support modules written for John's support can be used.

Benjamin Franksen
: John's support only worked on vxWorks. In addition, the driver support was implemented as a single source file. Benjamin defined an interface between drvCommon and low level controllers and split the code into drvGpib and the low level drivers. He also created the support for drvVxi11.

Eric Norum
: Eric started with Benjamin's code and converted it to use the Operating System Independent features of EPICS 3.14.

Marty Kraimer
: Marty started with Eric's version and made changes to support secondary addressing; and to replace ioctl with code to support general bus management, universal commands, and addressed commands.

Pete Owens
: Pete, for the Diamond Light Source, did a survey of several types of device/driver support packages for serial devices. Diamond decided to use the STREAMS support developed by Dirk Zimoch.

Dirk Zimoch
: Dirk developed STREAMS, which has a single device support model, but supports arbitrary low level message based drivers, i.e. GPIB, serial, etc.

Jun−ichi Odagare
: Jun−ichi developed NetDev, a system that provides EPICS device support for network based devices. It has a single device support model, but provides a general framework for communicating with network based devices.

Mark Rivers
: Mark became an active developer of asynDriver soon after he started converting SYNAPPS to use asynDriver. He soon pushed to have asynDriver support synchronous drivers, support register based drivers, and support interrupts. With these additions asynDriver is a framework for interfacing to a large class of devices instead of just message based asynchronous devices.

Yevgeny A. Gusev
: Yevgeny has found bugs and suggested improvements in the way asynManager handles queue timeouts and cancels. He provides an expert and welcome set of eyes to look at difficult code!!!

Overview of asynDriver

Definitions

asynDriver is a software layer between device specific code and drivers that communicate with devices. It supports both blocking and non−blocking communication and can be used with both register and message based devices. asynDriver uses the following terminology:

- interface

All communication between software layers is done via interfaces. An interface definition 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".

- port

  A physical or logical entity which provides access to a device. A port provides access to one or more devices.
- portDriver

  Code that communicates with a port.
- portThread

  If a portDriver can block, a thread is created for each port, and all I/O to the portDriver is done via this thread.
- device

  A device (instrument) connected to a port. For example a GPIB interface can have up to 15 devices connected to it. Other ports, e.g. EIA232 serial ports, only support a single device. Whenever this document uses the word device without a qualifier, it means something that is connected to a port.
- device support

  Code that interacts with a device.
- synchronous

  Support that does not voluntarily give up control of the CPU.
- asynchronous

  Support that is not synchronous. Some examples of asynchronous operations are epicsThreadSleep, epicsEventWait, and stdio operations. Calls to epicsMutexTake are considered to be synchronous operations, i.e. they are permitted in synchronous support.
- asynDriver

  The name for the support described in this manual. It is also the name of the header file that describes the core interfaces.
- asynManager

  An interface and the code which implements the methods for interfaces asynManager and asynTrace.
- asynchronous Driver

  A driver that may block while communicating with a device. Typical examples are serial, gpib, and network based drivers.
- synchronous Driver

  A driver that does not block while communicating with a device. Typical examples are VME register based devices.
- Message Based Interfaces

  Interfaces that use octet arrays for read/write operations.
- Register Based Interfaces

Interfaces that use integers or floats for read/write operations.

- interrupt

  As implemented by asynManager, interrupt just means "I have a new value for port, address".

Synchronous/asynchronous and message/register are orthogonal concepts. For example a register based driver can be either synchronous or asynchronous. The terminology register vs message is adapted from VXI.

Standard interfaces are defined so that device specific code can communicate with multiple port drivers. For example if device support does all its communication via reads and writes consisting of 8 bit bytes (octets), then it should work with all port drivers that support octet messages. If device support requires more complicated support, then the types of ports will be more limited. Standard interfaces are also defined for drivers that accept 32 bit integers or 64 bit floats. Additional interfaces can be defined, and it is expected that additional standard interfaces will be defined.

One or more devices can be attached to a port. For example, only one device can be attached to an RS−232 port, but up to 15 devices can be attached to a GPIB port.

Multiple layers can exist between device specific code and a port driver. A software layer calls interposeInterface in order to be placed between device specific code and drivers. For more complicated protocols, additional layers can be created. For example, GPIB support is implemented as an asynGpib interface which is called by user code, and an asynGpibPort interface which is called by asynGpib.

A driver normally implements multiple interfaces. For example asynGpib implements asynCommon, asynOctet, and asynGpib.

asynManager uses the Operating System Independent features of EPICS base. It is, however, independent of record/device support. Thus, it can be used by other code, e.g. a sequence program.

Standard Interfaces

This section briefly describes the interfaces provided by asynManager and standard interfaces implemented by port drivers. asynManager methods are called by application threads.

The interfaces are:

asynManager provides services for communicating with a device connected to a port.

asynCommon is an interface that must be implemented by all low level drivers. The methods are:

- report − Report status of port.
- connect − Connect to the port or device.
- disconnect − Disconnect from the port or device.

asynTrace is an interface for generating diagnostic messages.

asynDrvUser is an interface for communicating information from a user to a driver

In addition to these standard interfaces port drivers will implement one or more of the message and/or register based interfaces described in later sections.

Theory of Operation

Flow of Control

During initialization, port drivers register each communication port as well as all supported interfaces.

User code creates an asynUser, which a "handle" for accessing asynDriver facilities, by calling

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

An asynUser has the following features:

- An asynUser is the means by which asynManager manages multiple requests for accessing a port.
- processCallback is the addresss of a user supplied callback routine.
- timeoutCallback is the address of caller supplied callback that will be called if a queueRequest remains on the queue too long.
- Device support code should create an asynUser for each "atomic" access to low level drivers, i.e. a set of calls that must not be interlaced with other calls to the low level drivers. For example device support for EPICS record support should create an asynUser for each record instance.
- Device support code should NOT try to share an asynUser between multiple sources of requests for access to a port. If this is done then device support must itself handle contention issues that are already handled by asynManager.

User code connects to a low level driver via a call to

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

This call must specify the name of the port and the address of the device. It then calls findInterface to locate the interfaces with which it calls the driver. For example:

```
 pasynInterface = pasynManager->findInterface(pasynUser,asynOctetType,1);
```

User code requests access to a port by calling:

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

This results in either processCallback or timeoutCallback being called. Most requests to a port must be made from processCallback. queueRequest does not block (assuming that processCallback does not block). If queueRequest is called for a port that can block the request is queued to a thread dedicated to the port. If queueRequest is called for a port does not block it just calls processCallback. In either case multiple threads do not simultaneously call a low level driver. This guarantee is valid only if low level drivers are only accessed by calling queueRequest.

The following examples are based on EPICS IOC record/device support.

The first example shows access to a port that can block.

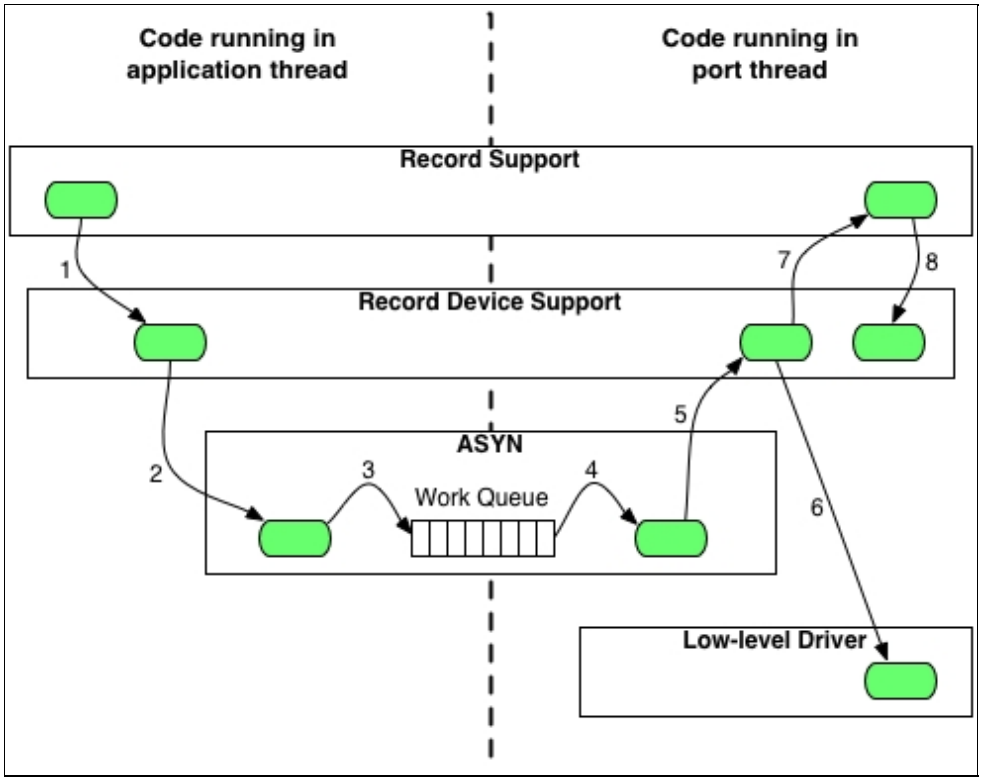


Figure 1: Asynchronous Control Flow

The sequence of record device support events that occurs starting with an application thread is pictured above in Figure 1, and explained below in the following steps:

1. Record processing calls Record device support with record.PACT=0.
2. Record 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. All subsequent operations for this I/O request are handled in the port driver thread.
4. The portThread removes the I/O request from the work queue.
5. The portThread calls the processCallback located in Record device support.
6. processCallback calls the low−level driver. The low−level driver read or write routine blocks until the I/O completes or until a timeout occurs. The low−level driver routine returns the results of the I/O operation to processCallback.
7. processCallback calls the record support process method.
8. Record support calls Record device support again, this time with record.PACT=1. Record device support updates fields in the record and returns to record support which completes record processing.

The second example shows access to a port that cannot block.

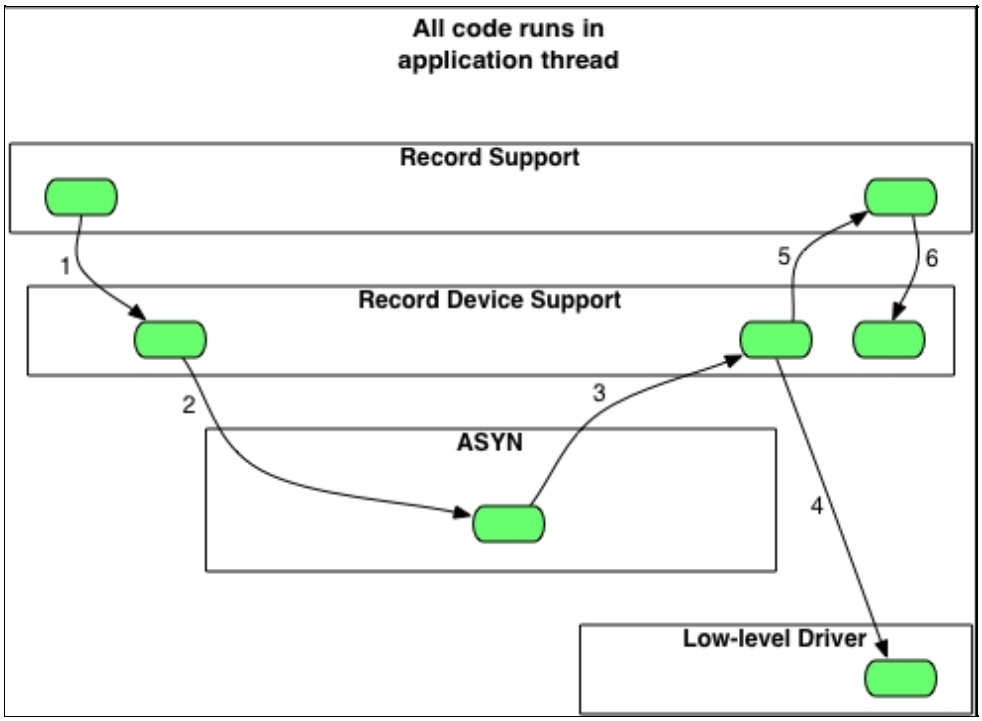


Figure 2: Synchronous Control Flow

The sequence of record device support events that occurs starting with an application thread is pictured above in Figure 2, and explained below in the following steps:

1. Record processing calls Record device support.
2. Record device support calls queueRequest.
3. Since the port is synchronous, i.e. can not block, queueRequest calls 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 calls the record support process method.
6. Record support calls Record device support again, this time with record.PACT=1. Record device support updates fields in the record and 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

  Method: report
- asynUser creation

  Methods: createAsynUser, duplicateAsynUser, freeAsynUser

  An asynUser is a "handle" for accessing other asynManager services and for calling interfaces implemented by drivers. An asynUser must only be created via a call to createAsynUser or duplicateAsynUser since asynManager keeps private information for each asynUser. freeAsynUser puts

the asynUser on a free list rather than calling free. Clients can continually create and free asynUsers quickly and without fragmenting memory.

The call to createAsynUser specifies a processCallback and a timeoutCallback. These are the callbacks that will be called as a result of a queueRequest.

An asynUser should not be shared between parts of code that can simultaneously access a driver. For example device support for standard EPICS records should create an asynUser for each record instance.

- Basic asynUser services

  Methods: connectDevice, disconnect, findInterface

  These methods should only be called by the code that created the asynUser.

  After an asynUser is created the user calls connectDevice. The user is connected to a port driver that can communicate with a device. findInterface is called for each interface the user requires. disconnect is called when the user is done with the device.
- Queuing services

  Methods: queueRequest, cancelRequest, lock, unlock

  These methods should only be called by the code that created the asynUser.

  queueRequest is a request to call either the processCallback or timeoutCallback specified in the call to createAsynUser. Most interface methods must only be called from processCallback via a call to queueRequest. Exceptions to this rule must be clearly documented (a common exception are methods registerInterruptUser/cancelInterruptUser).

  What queueRequest does depends on if the port driver can block.

  When registerPort is called by a driver that can block, a thread is created for the port. A set of queues, based on priority, is created for the thread. queueRequest puts the request on one of the queues. The port thread takes the requests from the queues 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 releases the mutex. The mutex guarantees that two callbacks to a port are not active at the same time.

  Lock is a request to prevent acccess to a port by other asynUsers between queueRequests. After lock is called, nothing happens until the next time processCallback is ready to be called. Starting at that time no other asynUser's processCallback will be called until unlock is called. It should be noted that lock/unlock works with non−blocking drivers. The implementation depends on the fact that epicsMutex supports recursive mutexs.
- Basic Driver services

  Methods: registerPort,registerInterface

  registerPort should only be 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 then calls registerPort and then registerInterface for each interface it supports.

- Attribute Retrieval

  Methods: isMultiDevice, canBlock, getAddr, getPortName, isConnected, isEnabled, isAutoConnect

  These methods can be called by any code that has access to the asynUser
- Connection services

  These methods can be called by any code that has access to the asynUser.

  Methods: enable,autoConnect

  These methods can be called to set the enable and autoConnect 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

  Methods: exceptionCallbackAdd, exceptionCallbackRemove, exceptionConnect, exceptionDisconnect

  Device support code calls exceptionCallbackAdd and exceptionCallbackRemove. The complete list of exceptions is defined in asynDriver.h as "enum asynException".

  Whenever a port driver connects or disconnects, normally as a result of a call to asynCommon:connect or asynCommon:disconnect, it must also call exceptionConnect or exceptionDisconnect.
- Interrupt services

  Methods: registerInterruptSource, getInterruptPvt, createInterruptNode, freeInterruptNode, addInterruptUser, removeInterruptUser, interruptStart, interruptEnd

  Interrupt just means: "I have a new value for ." Many asyn interfaces, e.g. asynInt32, provide interrupt support. These interfaces provide methods registerInterruptUser and cancelInterruptUser. Device support calls registerInterruptUser if it wants to be called whenever an interrupt occurs. Drivers or other code that implements the interface calls the registers users when it has new data. asynManager provides services that help drivers implement thread−safe support for interrupts.

  A driver that wants to support interrupts must call registerInterruptSource for each interface it supports. After it has done this it can call interruptStart to obtain a list of all registered users. It call interruptEnd when it is done. If any calls are made to addInterruptUser or removeInterruptUser between the calls to interruptStart/interruptEnd, asynManager puts the request on a list. When interruptEnd is called it processes the request.

  The code that implements registerInterruptSource must call addInterruptUser and removeInterruptUser.

  On operating systems like vxWorks or RTEMS interruptStart,interruptEnd MUST NOT be called from interupt level.

  Code that has called connectDevice can call registerInterruptUser and cancelInterruptUser. The callbacks must not block.

Many standard interfaces, e.g. asynInt32, provide methods registerInterruptUser, cancelInterruptUser. These interfaces also provide an auxilliary interface, e.g. asynInt32Base, and code which implements registerInterruptUser and cancelInterruptUser. The implementation calls the addInterruptUser and removeInterruptUsersynManager methods.

- General purpose freelist service

  Methods: memMalloc, memFree

  These methods do not require an asynUser. They are provided for code that must continually allocate and free memory. Since memFree puts the memory on a free list instead of calling free, they are more efficient that calloc/free and also help prevent memory fragmentation.
- Interpose service

  Method: interposeInterface

  Code the calls interposeInterface implements an interface that is either not supported by a port driver or that is "interposed" between the caller and the port driver. For example asynInterposeEos interposes asynOctet. It performs end of string processing for port drivers that do not support it.

  interposeInterface is recursive, i.e. an arbitrary number of interpose layers can exist above a single port,addr.

Multiple Device vs Single Device Port Drivers

When a low level driver calls registerPort, it declares if it handles multiple devices. This determines how the addr argument to connectDevice is handled and what getAddr returns.

- multiDevice false

  The addr argument to connectDevice is ignored and getAddr always returns −1
- multiDevice true

  If connectDevice is called with addr<0, the connection is to the port and getAddr always returns −1. If addr>=0, then the caller is connected to the device at the specified address. getAddr will return this address. An asynUser connected to the port can issue requests that affect all address on the port. For example disabling access to the port prevents access to all addresses on the port.

Connection Management

asynManager keeps track of the following states:

- connection

  Is the port or device connected? This state is initialized to disconnected.
- enabled

  Is the port or device enabled? This state is initialized to enabled.
- autoConnect

Does asynManager call connect if it finds the port or device disconnected? This is initialized to the state specified in the call to registerPort.

If the port does not support multiple devices, then port and device status are the same. If the port does support multiple devices, then asynManager keeps track of the states for the port and for every device connected to the port.

Whenever any of the states change for a port or device, then all users that previously called exceptionCallbackAdd for that port or device are called.

Low level drivers must call pasynManager:exceptionConnect whenever they connect to a port or port,addr and must call exceptionDisconnect whenever they disconnect.

Protecting a Thread from Blocking

The methods asynManager:report and asynCommon:report can be called by any thread, but the caller is blocked until the report finishes. The other asynManager methods can be called by any thread including portThread. None of these methods (except report) block.

Unless stated otherwise the methods for other interfaces must only be called by the queue callback specified in the call to createAsynUser, i.e. queueRequest must be called.

Methods registerInterruptUser and cancelInterruptUser of any interface do not have to be called by processCallback. They never block. The registerInterruptUser callback must not block because it could be called by a non blocking driver.

portThread

If a driver calls asynManager:registerPort with the ASYN_CANBLOCK bit of attributes set then asynManager creates a thread for the port. Each portThread has its own set of queues for the calls to queueRequest. Four queues are maintained. One queue is used only for asynCommon:connect and asynCommon:disconnect requests. The other queues provide different priorities: low, medium, and high. portThread runs forever implementing the following algorithm:

1. Wait for work by calling epicsEventMustWait. Other code such as queueRequest call epicsEventSignal.
2. If the port is disabled, go back to 1.
3. For every element in queue, asynQueuePriorityConnect:
    - Removes the element from the queue.
    - Calls the user's callback
4. If the port is not connected and autoConnect is true for the port, then attempt to connect to the port.
5. If the port is still not connected, go back to 1.
6. For each element of the queues asynQueuePriorityHigh, ...,asynQueuePriorityLow.
    - If disabled, skip this element.
    - If not connected and autoConnect is true for the device, then attempt to connect to the device.
    - If not connected, skip this element.
    - If locked by another thread, skip this element.
    - If not locked and user has requested lock, then lock.
    - Remove from queue and call user callback.

The actual code is more complicated because it unlocks before it calls code outside asynManager. This means that the queues can be modified and exceptions may occur.

asynDriver Structures and Interfaces

asynDriver.h describes the following:

- asynStatus − An enum that describes the status returned by many methods.
- asynException − An enum that describes exceptions.
- asynQueuePriority − An enum that describes the queue priorities.
- asynUser − A struture that contains generic information and is the "handle" for calling most methods.
- asynInterface − a structure that describes an interface.
- asynManager − An interface for communicating with asynDriver.
- asynCommon − An interface providing methods that must be implemented by all low level drivers.
- asynTrace − An interface plus associated functions and definitions that implement the trace facility.

asynStatus

Defines the status returned by most methods. If a method returns a status other than asynSuccess, and one of the arguments to the method is pasynUser, then the method is expected to write a message into pasynUser−>errorMessage.

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

asynStatus

asynSuccess	The request was successful.
asynTimeout	The request failed with a timeout.
asynOverflow	The driver has lost input data. This can happen if an internal buffer or the user supplied buffer is too small. Whenever possible, low level drivers must be written so that the user can read input in small pieces.
asynError	Some other error occured.

asynException

Defines the exceptions for method exceptionOccurred

```
typedef enum {
    asynExceptionConnect,asynExceptionEnable,asynExceptionAutoConnect,
    asynExceptionTraceMask,asynExceptionTraceIOMask,
    asynExceptionTraceFile,asynExceptionTraceIOTruncateSize
} asynException;
```

asynException

asynExceptionConnect	The connection state of the port or device has changed.
asynExceptionEnable	The enable state of the port or device has changed.
asynExceptionAutoConnect	The autoConnect state of the port or device has changed.

asynExceptionTraceMask	The traceMask for the port or device has changed.
asynExceptionTraceIOMask	The traceIOMask for the port or device has changed.
asynExceptionTraceFile	The trace file for the port or device has changed.
asynExceptionTraceIOTruncateSize	The traceIOTruncateSize for the port or device has changed.

asynQueuePriority

This defines the priority passed to queueRequest.

```
typedef enum {
    asynQueuePriorityLow,asynQueuePriorityMedium,asynQueuePriorityHigh,
    asynQueuePriorityConnect
}asynQueuePriority;
```

asynQueuePriority

asynQueuePriorityLow	Lowest queue priority.
asynQueuePriorityMedium	Medium queue priority.
asynQueuePriorityHigh	High queue priority.
asynQueuePriorityConnect	Queue a connect or disconnect request. This priority must be used for and only for connect/disconnect requests.

asynUser

Describes a structure that user code passes to most asynManager and driver methods. Code must allocate and free an asynUser by calling asynManager:createAsynUser (or asynManager:dupliateAsynUser) and asynManager:freeAsynUser.

```
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;
```

asynUser

errorMessage	When a method returns asynError it should put an error message into errorMessage via a call to:
	`epicsSnprintf(pasynUser->errorMessage,pasynUser->errorMessageSize,`
	`"<format>",...)`

errorMessageSize	The size of errorMessage. The user can not change this value.
timeout	The number of seconds before timeout for I/O requests. This is set by the user and can be

	changed between calls to drivers. If a call to a low level driver results in the driver making many I/O requests this is the time for each I/O request. The user must provide a non zero value or many low level drivers will timeout.
userPvt	For use by the user. The user should set this immediately after the call to pasynManager−>createAsynUser. If this is changed while asynUser is queued, the results are undefined, e.g. it could cause a crash.
userData	Also for use by the user.
drvUser	A driver can use this to hold asynUser specific data. The asynDrvUser interface is used for communication between asynUser and the driver.
reason	Drivers and asynUsers can use this as a general purpose field. By convention it is used for asynManager interrupt support. A driver that is calling an interrupt users often uses reason to decide if the users callback should be called. Values of reason less than 0 are reserved for standard meanings. For example ASYN_REASON_SIGNAL is used to mean "out of band" request. The devGpib support uses this to report SRQs.
auxStatus	Any method can provide additional return information in auxStatus. The meaning is determined by the method.

asynInterface

This defines an interface registered with asynPortManager:registerPort or asynManager:interposeInterface.

```
typedef struct asynInterface{
    const char *interfaceType; /*For example, asynCommonType */
    void *pinterface;          /*For example, pasynCommon */
    void *drvPvt;
}asynInterface;
```

asynInterface

interfaceType	A character string describing the interface.
pinterface	A pointer to the interface. The user must cast this to the correct type.
drvPvt	For the exclusive use of the code that called registerPort or interposeInterface.

asynManager

This is the main interface for communicating with asynDriver.

```
/*registerPort attributes*/
#define ASYN_MULTIDEVICE  0x0001
#define ASYN_CANBLOCK     0x0002

/*standard values for asynUser.reason*/
#define ASYN_REASON_SIGNAL −1

typedef struct interruptNode{
    ELLNODE node;
    void    *drvPvt;
}interruptNode;

typedef struct asynManager {
    void      (*report)(FILE *fp,int details,const char*portName);
```

```
    asynUser  *(*createAsynUser)(userCallback process,userCallback timeout);
    asynUser  *(*duplicateAsynUser)(asynUser *pasynUser,
                                userCallback queue,userCallback timeout);
    asynStatus (*freeAsynUser)(asynUser *pasynUser);
    void       *(*memMalloc)(size_t size);
    void       (*memFree)(void *pmem,size_t size);
    asynStatus (*isMultiDevice)(asynUser *pasynUser,
                               const char *portName,int *yesNo);
    /* addr = (-1,>=0) => connect to (port,device) */
    asynStatus (*connectDevice)(asynUser *pasynUser,
                               const char *portName,int addr);
    asynStatus (*disconnect)(asynUser *pasynUser);
    asynStatus (*exceptionCallbackAdd)(asynUser *pasynUser,
                                       exceptionCallback callback);
    asynStatus (*exceptionCallbackRemove)(asynUser *pasynUser);
    asynInterface *(*findInterface)(asynUser *pasynUser,
                            const char *interfaceType,int interposeInterfaceOK);
    asynStatus (*queueRequest)(asynUser *pasynUser,
                               asynQueuePriority priority,double timeout);
    asynStatus (*cancelRequest)(asynUser *pasynUser,int *wasQueued);
    asynStatus (*canBlock)(asynUser *pasynUser,int *yesNo);
    asynStatus (*lock)(asynUser *pasynUser);   /*lock portName,addr */
    asynStatus (*unlock)(asynUser *pasynUser);
    asynStatus (*getAddr)(asynUser *pasynUser,int *addr);
    asynStatus (*getPortName)(asynUser *pasynUser,const char **pportName);
    /* drivers call the following*/
    asynStatus (*registerPort)(const char *portName,
                               int attributes,int autoConnect,
                               unsigned int priority,unsigned int stackSize);
    asynStatus (*registerInterface)(const char *portName,
                               asynInterface *pasynInterface);
    asynStatus (*exceptionConnect)(asynUser *pasynUser);
    asynStatus (*exceptionDisconnect)(asynUser *pasynUser);
    /*any code can call the following*/
    asynStatus (*interposeInterface)(const char *portName, int addr,
                               asynInterface *pasynInterface,
                               asynInterface **ppPrev);
    asynStatus (*enable)(asynUser *pasynUser,int yesNo);
    asynStatus (*autoConnect)(asynUser *pasynUser,int yesNo);
    asynStatus (*isConnected)(asynUser *pasynUser,int *yesNo);
    asynStatus (*isEnabled)(asynUser *pasynUser,int *yesNo);
    asynStatus (*isAutoConnect)(asynUser *pasynUser,int *yesNo);
    /*The following are methods for interrupts*/
    asynStatus (*registerInterruptSource)(const char *portName,
                               asynInterface *pasynInterface, void **pasynPvt);
    asynStatus (*getInterruptPvt)(asynUser *pasynUser,
                               const char *interfaceType, void **pasynPvt);
    interruptNode *(*createInterruptNode)(void *pasynPvt);
    asynStatus (*freeInterruptNode)(asynUser *pasynUser,interruptNode *pnode);
    asynStatus (*addInterruptUser)(asynUser *pasynUser,
                                 interruptNode*pinterruptNode);
    asynStatus (*removeInterruptUser)(asynUser *pasynUser,
                                 interruptNode*pinterruptNode);
    asynStatus (*interruptStart)(void *pasynPvt,ELLLIST **plist);
    asynStatus (*interruptEnd)(void *pasynPvt);
}asynManager;
epicsShareExtern asynManager *pasynManager;
```

asynManager

report	

	Reports status about the asynPortManager. If portName is non−NULL it reports for a specific port. If portName is NULL then it reports for each registered port. It also calls asynCommon:report for each port being reported.
createAsynUser	Creates an asynUser. The caller specifies two callbacks, process and timeout. These callback are only called as a result of a queueRequest. The timeout callback is optional. errorMessageSize characters are allocated for errorMessage. The amount of storage can not be changed. This method doesn't return if it is unable to allocate the storage.
duplicateAsynUser	Creates an asynUser by calling createAsynUser. It then initializes the new asynUser as follows: The fields timeout, userPvt, userData, and drvUser are initialized with values taken from pasynUser. Its connectDevice state is the same as that for pasynUser.
freeAsynUser	Free an asynUser. The user must free an asynUser only via this call. If the asynUser is connected to a port, disconnect is called. If the disconnect fails, this call will also fail. The storage for the asynUser is saved on a free list and will be reused in later calls to createAsynUser or duplicateAsynUser. Thus continually calling createAsynUser (or duplicateAsynUser) and freeAsynUser is efficient.
memMalloc/memFree	Allocate/Free memory. memMalloc/memFree maintain a set of freelists of different sizes. Thus any application that needs storage for a short time can use memMalloc/memFree to allocate and free the storage without causing memory fragmentation. The size passed to memFree MUST be the same as the value specified in the call to memMalloc.
isMultiDevice	Answers the question "Does the port support multiple devices?" This method can be called before calling connectDevice.
connectDevice	Device code calls this to connect to a device. It passes the name of the communication port and the address of the device. The port Name is the same as that specified in a call to registerPort. The call will fail if the asynUser is already connected. If the port does not support multiple devices, than addr is ignored. The call will fail if the asynUser is already connected to a device. connectDevice only connects a user to the port driver for the portName,addr. The port driver may or may not be connected to the actual device. Thus, connectDevice and asynCommon:connect are completely different. See the Theory of Operation section for a description of the difference between single and multi−device port drivers.
disconnect	Disconnect from the port,addr to which connectDevice is connected. The call will fail if the asynUser is queued or locked, or has a callback registered via exceptionCallbackAdd. Note that asynManager:disconnect and asynCommon:disconnect are completely different.
exceptionCallbackAdd	Callback will be called whenever one of the exceptions defined by asynException occurs. The callback can call isConnected, isEnabled, or isAutoConnect to find the new state.
exceptionCallbackRemove	Callback is removed. This must be called before disconnect.
findInterface	Find a driver interface. If interposeInterfaceOK is true, then findInterface returns the last interface registered or interposed. Otherwise, the interface

	registered by registerPort is returned. It returns 0 if the interfaceType is not supported.
	
	The user needs the address of the driver's interface and of pdrvPvt so that calls can be made to the driver. For example:
	
	`asynInterface *pasynInterface;`
	`asynOctet *pasynOctet;`
	`void *pasynOctetPvt;`
	`...`
	`pasynInterface = pasynManager->findInterface(`
	`  pasynUser,asynOctetType,1);`
	`if(!pasynInterface) { /*error do something*/}`
	`pasynOctet = (asynOctet *)pasynInterface->pinterface;`
	`pasynOctetPvt = pasynInterface->pdrvPvt;`
	`...`
	`/* The following call must be made from a callback */`
	`pasynOctet->read(pasynOctetPvt,pasynUser,...`

queueRequest	When registerPort is called, the caller must specify if it can block, i.e. attribute bit ASYN_CANBLOCK is set or cleared. If the port has been registered with ASYN_CANBLOCK true then the request is put on a queue for the thread associated with the queue. If the port has been registered with ASYN_CANBLOCK false then queueRequest locks the port and calls the process callback. In either case the process callback specified in the call to createAsynUser is called. If the asynUser is already on a queue, asynError is returned. The timeout starts when the request is queued. A value less than or equal to 0.0 means no timeout. The request is removed from the queue before the callback is called. Thus callbacks are allowed to unlock and issue new queue requests. The priority asynQueuePriorityConnect must be used for asynCommon:connect and asynCommon:disconnect calls, and must NOT be used for any other calls. If a timeout callback was not passed to createAsynUser and a queueRequest with a non−zero timeout is requested, an error message is issued and no timeout will occur.
cancelRequest	If a asynUser is queued, remove it from the queue. If either the process or timeout callback is active when cancelRequest is called than cancelRequest will not return until the callback completes.
canBlock	yesNo is set to (0,1), i.e. (false,true) if calls to the low level driver can block. The value is determined by the attributes passed to registerPort.
lock/unlock	lock/unlock are used to block other users from accessing a device while a user is making a series of queueRequests. Only the addr specified in the connectDevice request is locked. asynManager locks when a queueRequest is taken from the queue. At that point all other entries in the queue must wait until unlock is called by the same pasynUser that locked. lock/unlock fail if a request is currently queued. The addr argument passed to connectDevice determines if the port or only a device is locked.
getAddr	

	*addr is set equal to the address which the user specified in the call to connectDevice or −1 if the port does not support multiple devices. See the Theory of Operation section for a description of the difference between single and multi−device port drivers.
getPortName	*pportName is set equal to the name of the port to which the user is connected.
registerPort	This method is called by drivers. A call is made for each port instance. Attributes is a set of bits. Currently two bits are defined: ASYN_MULTIDEVICE and ASYN_CANBLOCK. The driver must specify these properly. autoConnect, which is (0,1) for (no,yes), provides the initial value for the port and all devices connected to the port. If priority is 0, then a default will be assigned. If stackSize is 0, a default is assigned. The portName argument specifies the name by which the upper levels of the asyn code will refer to this communication interface instance.
registerInterface	This is called by port drivers for each supported interface.
exceptionConnect	This method must be called by the driver when and only when it connects to a port or device.
exceptionDisconnect	This method must be called by the driver when and only when it disconnects from a port or device.
interposeInterface	This is called by a software layer between client code and the port driver. For example, if a device echos writes then a software module that issues a read after each write could be created and call interposeInterface for interface asynOctet. Multiple interposeInterface calls for a port/addr/interface can be issued. *ppPrev is set to the address of the previous asynInterface. Thus the software module that last called interposeInterface is called by user code. It in turn can call the software module that was the second to last to call interposeInterface. This continues until the actual port driver is called. interposeInterface can also be called with an asynInterface that has not been previously registered or replaced. In this case *ppPrev will be null. Thus, new interfaces that are unknown to the low level driver can be implemented.
enable	If enable is set yes, then queueRequests are not dequeued unless their queue timeout occurs.
autoConnect	If autoConnect is true and the port or device is not connected when a user callback is scheduled to be called, asynManager calls pasynCommon−>connect. See the discussion of Flow of Control below for details.
isConnected	*yesNo is set to (0,1) if the port or device (is not, is) connected.
isEnabled	*yesNo is set to (0,1) if the port or device (is not, is) enabled.
isAutoConnect	*yesNo is set to (0,1) if the portThread (will not, will) autoConnect for the port or device.
registerInterruptSource	If a low level driver supports interrupts it must call this for each interface that supports interrupts.

getInterruptPvt	Any code that wants to call createInterruptNode must call this because it provides the argument required by createInterruptNode. The caller must be connected to a device, i.e. must have called connectDevice, or getInterruptPvt returns asynError.
createInterruptNode/freeInterruptNode	These methods are the only way a user can allocate and free an interruptNode. pasynPvt is the value obtained from getInterruptPvt. createInterruptNode/freeInterruptNode are separate methods rather than being done automatically by addInterruptUser/removeInterruptUser so that addInterruptUser/removeInterruptUser can be efficient.
addInterruptUser/removeInterruptUser	Code that implements registerInterruptUser/cancelInterruptUser must call addInterruptUser/removeInterruptUser to add and remove users from the list or else calls to interruptStart/interruptEnd will no work. This is an efficient operation so that a user can repeatedly call registerInterruptUser/cancelInterruptUser. If either of these is called while a interrupt is being processed, i.e. between calls to interruptStart/interruptEnd, the call will block until interruptEnd is called. If a registerInterruptUser callback calls cancelInterruptUser it will block forever.
interruptStart/interruptEnd	The code that implements interrupts is interface dependent. The only service asynManager provides is a thread−safe implemention of the user list. When the code wants to call the callback specified in the calls to registerInterruptUser, it calls interruptStart to obtain the list of callbacks. When it is done it calls interruptEnd. If any requests are made to addInterruptUser/removeInterruptUser between the calls to interruptStart and interruptEnd, asynManager delays the requests until interruptEnd is called.

asynCommon

asynCommon describes the methods that must be implemented by drivers.

```
/* Device Interface supported by ALL asyn drivers*/
#define asynCommonType "asynCommon"
typedef struct  asynCommon {
    void       (*report)(void *drvPvt,FILE *fp,int details);
    /*following are to connect/disconnect to/from hardware*/
    asynStatus (*connect)(void *drvPvt,asynUser *pasynUser);
    asynStatus (*disconnect)(void *drvPvt,asynUser *pasynUser);
}asynCommon;
```

asynCommon

report	Generates a report about the hardware device. This is the only asynCommon method that does not have to be called by the queueRequest callback.
connect	Connect to the hardware device or communication path. The queueRequest must specify priority asynQueuePriorityConnect.
disconnect	Disconnect from the hardware device or communication path. The queueRequest must specify priority asynQueuePriorityConnect.

asynDrvUser

asynDrvUser provides methods that allow an asynUser to communicate user specific information to/from a port driver

```
#define asynDrvUserType "asynDrvUser"
typedef struct  asynDrvUser {
    /*The following do not have to be called via queueRequest callback*/
    asynStatus (*create)(void *drvPvt,asynUser *pasynUser,
        const char *drvInfo, const char **pptypeName,size_t *psize);
    asynStatus (*getType)(void *drvPvt,asynUser *pasynUser,
        const char **pptypeName,size_t *psize);
    asynStatus (*destroy)(void *drvPvt,asynUser *pasynUser);
}asynDrvUser;
```

asynDrvUser

create	The driver can create any resources it needs. It can use asynUser.drvUser to provide access to the resources. If the asynUser and the driver both know about how to access the resources they must agree and a name for the resource and a size. If pptypeName is not null the driver can give a value to *pptypeName. If psize is not null the driver can give a value to *psize. Unless asynUser receives a typeName and size that it recognizes it must not access asynUser.drvUser.
getType	If other code, e.g. an interposeInterface wants to access asynUser.drvUser it must call this and verify that typeName and size are what it expects.
destroy	Destroy the resources created by create and set asynUser.drvUser null.

asynOption

asynOption provides a generic way of setting driver specific options. For example the serial port driver uses this to specify baud rate, stop bits, etc.

```
#define asynOptionType "asynOption"
/*The following are generic methods to set/get device options*/
typedef struct asynOption {
    asynStatus (*setOption)(void *drvPvt, asynUser *pasynUser,
                            const char *key, const char *val);
    asynStatus (*getOption)(void *drvPvt, asynUser *pasynUser,
                            const char *key, char *val, int sizeval);
}asynOption;
```

asynOption

setOption	Set value associated with key.
getOption	Get value associated with key.

Trace Interface

```
/*asynTrace is implemented by asynManager*/
/*All asynTrace methods can be called from any thread*/
/* traceMask definitions*/
#define ASYN_TRACE_ERROR     0x0001
#define ASYN_TRACEIO_DEVICE  0x0002
#define ASYN_TRACEIO_FILTER  0x0004
#define ASYN_TRACEIO_DRIVER  0x0008
#define ASYN_TRACE_FLOW      0x0010
```

```
/* traceIO mask definitions*/
#define ASYN_TRACEIO_NODATA 0x0000
#define ASYN_TRACEIO_ASCII  0x0001
#define ASYN_TRACEIO_ESCAPE 0x0002
#define ASYN_TRACEIO_HEX    0x0004

/* asynPrint and asynPrintIO are macros that act like
   int asynPrint(asynUser *pasynUser,int reason, const char *format, ... );
   int asynPrintIO(asynUser *pasynUser,int reason,
        const char *buffer, int len, const char *format, ... );
*/
typedef struct asynTrace {
    /* lock/unlock are only necessary if caller performs I/O other than*/
    /* by calling asynTrace methods                                    */
    asynStatus (*lock)(asynUser *pasynUser);
    asynStatus (*unlock)(asynUser *pasynUser);
    asynStatus (*setTraceMask)(asynUser *pasynUser,int mask);
    int        (*getTraceMask)(asynUser *pasynUser);
    asynStatus (*setTraceIOMask)(asynUser *pasynUser,int mask);
    int        (*getTraceIOMask)(asynUser *pasynUser);
    asynStatus (*setTraceFILE)(asynUser *pasynUser,FILE *fp);
    FILE       *(*getTraceFILE)(asynUser *pasynUser);
    asynStatus (*setTraceIOTruncateSize)(asynUser *pasynUser,int size);
    int        (*getTraceIOTruncateSize)(asynUser *pasynUser);
    int        (*print)(asynUser *pasynUser,int reason, const char *pformat, ...);
    int        (*printIO)(asynUser *pasynUser,int reason,
               const char *buffer, int len,const char *pformat, ...);
}asynTrace;
epicsShareExtern asynTrace *pasynTrace;
```

asynTrace

asynDriver provides a trace facility with the following attributes:

- Tracing is turned on/off for individual devices, i.e. a portName, addr.
- Trace has a global trace mask for asynUsers not connected to a port or port, addr.
- The output is sent to a file or to stdout.
- A mask determines the type of information that can be displayed. The various choices can be ORed together.
    - ASYN_TRACE_ERROR Run time errors are reported, e.g. timeouts.
    - ASYN_TRACEIO_DEVICE High level device support reports I/O activity.
    - ASYN_TRACEIO_FILTER Any layer between device support and the low level driver reports any filtering it does on I/O.
    - ASYN_TRACEIO_DRIVER Low level driver reports I/O activity.
    - ASYN_TRACE_FLOW Report logic flow. Device support should report all queue requests, callbacks entered, and all calls to drivers. Layers between device support and low level drivers should report all calls they make to lower level drivers. Low level drivers report calls they make to other support.
- Another mask determines how message buffers are printed. The various choices can be ORed together.
    - ASYN_TRACEIO_NODATA Don't print any data from the message buffers.
    - ASYN_TRACEIO_ASCII Print with a "%s" style format.
    - ASYN_TRACEIO_ESCAPE Call epicsStrPrintEscaped.
    - ASYN_TRACEIO_HEX Print each byte with " %2.2x".

In order for the trace facility to perform properly; device support and all drivers must use the trace facility. Device and driver support can directly call the asynTrace methods. The asynPrint and asynPrintIO macros are provided

so that it is easier for device/driver support. Support can have calls like:

```
asynPrint(pasynUser,ASYN_TRACE_FLOW,"%s Calling queueRequest\n",
    someName);
```

The asynPrintIO call is designed for device support or drivers that issue read or write requests. They make calls like:

```
asynPrintIO(pasynUser,ASYN_TRACEIO_DRIVER,data,nchars,"%s nchars %d",

            someName,nchars);
```

The asynTrace methods are implemented by asynManager. These methods can be used by any code that has created an asynUser and is connected to a device. All methods can be called by any thread. That is, an application thread and/or a portThread. If a thread performs all I/O via calls to print or printIO, then it does not have to call lock or unlock. If it does want to do its own I/O, it must lock before any I/O and unlock after. For example:

```
pasynTrace->lock(pasynUser);
fd = pasynTrace->getTraceFILE(pasynUser);
/*perform I/O of fd */
pasynTrace->unlock(pasynUser);
```

If the asynUser is not connected to a port, i.e. pasynManager->connectDevice has not been called, then a "global" device is assumed. This is useful when asynPrint is called before connectDevice.

asynTrace

lock/unlock	These are only needed if some code wants to do its own I/O instead of using print and printIO. The set methods, print, and printIO all lock while performing their operations. The get routines do not lock (except for getTraceFILE) and they are safe. The worst that happens is that the user gets a little more or a little less output.
setTraceMask	Set the trace mask. Normally set by the user requesting it via a shell command or the devTrace device support.
getTraceMask	Get the trace mask. Device support that wants to issue trace messages calls this to see what trace options have been requested.
setTraceIOMask	Set the traceIO mask. Normally set by the user requesting it via a shell command or the devTrace device support.
getTraceIOMask	Get the traceIO mask. Support that wants to issue its own IO messages instead of calling asynPrintIO should call this and honor the mask settings. Most code will not need it.
setTraceFILE	Set the stream to use for output. A NULL argument means use errlog. Normally set by the user requesting it via a shell command or by the devTrace device support. If the current output stream is none of (NULL, stdout, stderr) then the current output stream is closed before the new stream is used.
getTraceFILE	Get the file descriptor to use for output. Device support that wants to issue its own IO messages instead of calling asynPrintIO should call this and honor the mask settings. In this case, lock must have been called first. Most code will not need it. If the return value is 0, then ouput should be directed to errlog.
setTraceIOTruncateSize	Determines how much data is printed by printIO. In all cases it determines how many bytes of the buffer are displayed. The actual number of characters printed depends on

	the traceIO mask. For example ASYN_TRACEIO_HEX results in 3 characters being printed for each byte. Normally set by the user requesting it via a shell command or the devTrace device support.
getTraceIOTruncateSize	Get the current truncate size. Called by asynPrintIO. Code that does its own I/O should also support the traceIO mask.
print	If reason ORed with the current traceMask is not zero, then the message is printed. Most code should call asynPrint instead of calling this method.
printIO	If reason ORed with the current traceMask is not zero then the message is printed. If len is >0, then the buffer is printed using the traceIO mask and getTraceIOTruncateSize. Most code should call asynPrintIO instead of calling this method.

Standard Message Based Interfaces

These are interfaces for communicating with message based devices, where message based means that the device communicates via octet strings, i.e. arrays of 8 bit bytes. Three interfaces are provided: asynOctet, asynOctetSyncIO, and asynOctetTrapReadWrite. asynOctetSyncIO provides a synchronous inteface to asynOctet and can be used by code that is willing to block. asynOctetTrapReadWrite is an interposeInterface that can be used to intercept calls to asynOctet.

asynOctet

asynOctet describes the methods implemented by drivers that use octet strings for sending commands and receiving responses from a device.

NOTE: The name octet is used instead of ASCII because it implies that communication is done via 8−bit bytes.

```
#define ASYN_EOM_CNT 0x0001 /*Request count reached*/
#define ASYN_EOM_EOS 0x0002 /*End of String detected*/
#define ASYN_EOM_END 0x0004 /*End indicator detected*/

typedef void (*interruptCallbackOctet)(void *userPvt, asynUser *pasynUser,
                      char *data,size_t numchars, int eomReason);

typedef struct asynOctetInterrupt {
    asynUser *pasynUser;
    int      addr;
    interruptCallbackOctet callback;
    void *userPvt;
}asynOctetInterrupt;

#define asynOctetType "asynOctet"
typedef struct asynOctet{
    asynStatus (*write)(void *drvPvt,asynUser *pasynUser,
                    const char *data,size_t numchars,size_t *nbytesTransfered);
    asynStatus (*writeRaw)(void *drvPvt,asynUser *pasynUser,
                    const char *data,size_t numchars,size_t *nbytesTransfered);
    asynStatus (*read)(void *drvPvt,asynUser *pasynUser,
                    char *data,size_t maxchars,size_t *nbytesTransfered,
                    int *eomReason);
    asynStatus (*readRaw)(void *drvPvt,asynUser *pasynUser,
                    char *data,size_t maxchars,size_t *nbytesTransfered,
                    int *eomReason);
    asynStatus (*flush)(void *drvPvt,asynUser *pasynUser);
    asynStatus (*registerInterruptUser)(void *drvPvt,asynUser *pasynUser,
```

```
                    interruptCallbackOctet callback, void *userPvt,
                    void **registrarPvt);
    asynStatus (*cancelInterruptUser)(void *registrarPvt, asynUser *pasynUser);
    asynStatus (*setInputEos)(void *drvPvt,asynUser *pasynUser,
                    const char *eos,int eoslen);
    asynStatus (*getInputEos)(void *drvPvt,asynUser *pasynUser,
                    char *eos, int eossize, int *eoslen);
    asynStatus (*setOutputEos)(void *drvPvt,asynUser *pasynUser,
                    const char *eos,int eoslen);
    asynStatus (*getOutputEos)(void *drvPvt,asynUser *pasynUser,
                    char *eos, int eossize, int *eoslen);
}asynOctet;

#define asynOctetBaseType "asynOctetBase"
typedef struct asynOctetBase {
    asynStatus (*initialize)(const char *portName,
        asynInterface *pint32Interface,
        int processEosIn,int processEosOut,int interruptProcess);
    void       (*callInterruptUsers)(asynUser *pasynUser,void *pasynPvt,
        char *data,size_t maxchars,size_t *nbytesTransfered,int *eomReason);
} asynOctetBase;
epicsShareExtern asynOctetBase *pasynOctetBase;
```

asynOctet

write	Send a message to the device. *nbytesTransfered is the number of 8−bit bytes sent to the device. Interpose or driver code may add end of string terminators to the message but the extra characters are not included in *nbytesTransfered.
writeRaw	Send a message to the device. *nbytesTransfered is the number of 8−bit bytes sent to the device. Interpose or driver code must not add end of string terminators to the message.
read	Read a message from the device. *nbytesTransfered is the number of 8−bit bytes read from the device. If read returns asynSuccess than eomReason tells why the read completed. Interpose or driver code may strip end of string terminators from the message. If it does the first eos character will be replaced by null and the eos characters will not be included in nbytesTransfered.
readRaw	Read a message from the device. *nbytesTransfered is the number of 8−bit bytes read from the device. If read returns asynSuccess than eomReason tells why the read completed. Interpose or driver code must not strip end of string terminators from the message.
flush	Flush the input buffer.
registerInterruptUser	Register a user that will be called whenever a new message is received. NOTE: The callback must not block and must not call registerInterruptUser or cancelInterruptUser.
cancelInterruptUser	Cancel a registered user.
setInputEos	Set End Of String for input. For example "\n". Note that gpib drivers usually accept at most a one character string.
getInputEos	Get the current End of String.
setOutputEos	Set End Of String for output.
getOutputEos	Get the current End of String.

asynOctetBase is an interface and associated code that is used by drivers that implement interface asynOctet. asynOctetBase provides code to handle callbacks to users that call registerInterruptUser. It can optionally implement interrupt support. For single device support it can optionally call asynInterposeEosConfig to handle end of string processing for input and/or output.

asynInt32Base

initialize	After a driver calls registerPort is can call: `pasynInt32Base->initialize(...` Any null methods in the asynInterface are replaced by default implementations. asynInt32Base always implements registerInterruptUser and cancelInterruptUser. asynInt32Base can not implement processEosIn, processEosOut, and interruptProcess if the port is a multi−device port. If the port is not multi−device and either processEosIn or processEosOut is specified, asynInterposeEosConfig is called. Since this method are called only during initialization it can be called directly rather than via queueRequest.
callInterruptUsers	Calls the callbacks registered via registerInterruptUser.

asynOctetSyncIO

asynOctetSyncIO provides a convenient interface for software that needs to perform "synchronous" I/O to an asyn device, i.e. that starts an I/O operation and then blocks while waiting for the response. The code does not need to handle callbacks or understand the details of the asynManager and asynOctet interfaces. Examples include motor drivers running in their own threads, SNL programs, and the shell commands described later in this document.

```
typedef struct asynOctetSyncIO {
   asynStatus (*connect)(const char *port, int addr,
                         asynUser **ppasynUser, const char *drvInfo);
   asynStatus (*disconnect)(asynUser *pasynUser);
   asynStatus (*openSocket)(const char *server, int port, char **portName);
   asynStatus (*write)(asynUser *pasynUser, char const *buffer, int buffer_len,
                   double timeout,int *nbytesTransfered);
   asynStatus (*writeRaw)(asynUser *pasynUser,char const *buffer,int buffer_len,
                   double timeout,int *nbytesTransfered);
   asynStatus (*read)(asynUser *pasynUser, char *buffer, int buffer_len,
                   double timeout, int *nbytesTransfered,int *eomReason);
   asynStatus (*readRaw)(asynUser *pasynUser, char *buffer, int buffer_len,
                   double timeout, int *nbytesTransfered,int *eomReason);
   asynStatus (*writeRead)(asynUser *pasynUser,
                   const char *write_buffer, int write_buffer_len,
                   char *read_buffer, int read_buffer_len,
                   double timeout,
                   int *nbytesOut, int *nbytesIn, int *eomReason);
   asynStatus (*flush)(asynUser *pasynUser);
   asynStatus (*setInputEos)(asynUser *pasynUser,
                   const char *eos,int eoslen);
   asynStatus (*getInputEos)(asynUser *pasynUser,
                   char *eos, int eossize, int *eoslen);
   asynStatus (*setOutputEos)(asynUser *pasynUser,
                   const char *eos,int eoslen);
   asynStatus (*getOutputEos)(asynUser *pasynUser,
                   char *eos, int eossize, int *eoslen);
   asynStatus (*writeOnce)(const char *port, int addr,
                   char const *buffer, int buffer_len, double timeout,
                   int *nbytesTransfered, const char *drvInfo);
   asynStatus (*writeRawOnce)(const char *port, int addr,
                   char const *buffer, int buffer_len, double timeout,
                   int *nbytesTransfered, const char *drvInfo);
   asynStatus (*readOnce)(const char *port, int addr,
                   char *buffer, int buffer_len, double timeout,
                   int *nbytesTransfered,int *eomReason, const char *drvInfo);
```

```
   asynStatus (*readRawOnce)(const char *port, int addr,
                  char *buffer, int buffer_len, double timeout,
                  int *nbytesTransfered,int *eomReason, const char *drvInfo);
   asynStatus (*writeReadOnce)(const char *port, int addr,
                  const char *write_buffer, int write_buffer_len,
                  char *read_buffer, int read_buffer_len,
                  double timeout,
                  int *nbytesOut, int *nbytesIn, int *eomReason,
                  const char *drvInfo);
   asynStatus (*flushOnce)(const char *port, int addr,const char *drvInfo);
   asynStatus (*setInputEosOnce)(const char *port, int addr,
                  const char *eos,int eoslen,const char *drvInfo);
   asynStatus (*getInputEosOnce)(const char *port, int addr,
                  char *eos, int eossize, int *eoslen,const char *drvInfo);
   asynStatus (*setOutputEosOnce)(const char *port, int addr,
                  const char *eos,int eoslen,const char *drvInfo);
   asynStatus (*getOutputEosOnce)(const char *port, int addr,
                  char *eos, int eossize, int *eoslen,const char *drvInfo);
} asynOctetSyncIO;
epicsShareExtern asynOctetSyncIO *pasynOctetSyncIO;
```

asynOctetSyncIO

connect	Connects to an asyn port and address, returns a pointer to an asynUser structure.
disconnect	Disconnect. This frees all resources allocated by create.
openSocket	Opens a new connection to a TCP/IP or UDP/IP socket, returning the name of a newly created asyn port. The name of the port created is of the form "server:port [protocol]", i.e. "corvette:21" or "164.54.160.50:21" or "corvette:21 UDP".
write	Calls asynOctet−>write and waits for the operation to complete or time out.
writeRaw	Calls asynOctet−>writeRaw and waits for the operation to complete or time out.
read	Calls asynOctet−>read. Waits for the operation to complete or time out.
readRaw	Calls asynOctet−>readRaw. Waits for the operation to complete or time out.
writeRead	Calls pasynOctet−>flush, pasynOctet−>write, and asynOctet−>read. Waits for the operations to complete or time out.
flush	Calls pasynOctet−>flush
setInputEos	Calls pasynOctet−>setInputEos
getInputEos	Calls pasynOctet−>getInputEos
setOutputEos	Calls pasynOctet−>setOutputEos
getOutputEos	Calls pasynOctet−>getOutputEos
writeOnce	This does a connect, write, and disconnect.
writeRawOnce	This does a connect, writeRaw, and disconnect.
readOnce	This does a connect, read, and disconnect.
readOnce	This does a connect, read, and disconnect.
writeReadOnce	This does a connect, writeRead, and disconnect.

End of String Support

asynOctet provides methods for handling end of string (message) processing. It does not specify policy. Device support code, interpose layers, or low level drivers can all handle EOS processing. An application developer must decide what policy will be followed for individual devices. The policy will be determined by the device, the device support, and the driver.

Standard Register Based Interfaces

Introduction

This section descibes interfaces for register based devices. Support is provided for:

- Int32 − registers appear as 32 integers
- UInt32Digital − registers appear a 32 bit unsigned integers and masks can be used to address specific bits.
- Float64 − registers appear as double precision floats.
- Int32Array − Arrays of 32 bit integers.
- Float64Array − Arrays of double precision floats.

For Int32, UInt32Digital, and Float64 three interfaces are provided. In addition a default implementation and a synchronous inplementation are provided. Lets use Int32 as an example.

- asynInt32 − An interface with methods: read, write, getBounds, registerInterruptUser, and cancelInterruptUser.
- asynInt32Base − An interface used by drivers that implement asynInt32. It also has an implementation that:
    - ♦ registers the asynInt32 interface
    - ♦ has default methods for read, write, and getBounds
    - ♦ implements registerInterruptUser and cancelInterruptUser

  Drivers that implement asynInt32 normally call asynInt32Base:initialize and let it handle registerInterruptUser and cancelInterruptUser.
- asynInt32SyncIO − A synchronous interface to asynInt32

addr − What does it mean for register based interfaces?

Low level register based drivers are normally multi−device. The meaning of addr is:

- Int32 − The driver supports an array of Int32 values. addr selects an array element. For example a 16 channel ADC would support addr 0 through 15.
- Int32Array − Each addr is an array of Int32 values.
- Float64 − The driver supports an array of Float64 values. addr selects an array element.
- Float64Array − Each addr is an array of Float64 values.
- UInt32Digital − The driver supports an array of UInt32 values. addr selects an array element. For example a 128 bit digital I/O module appears as an array of four UInt32 registers.

Example Drivers

Two examples of drivers that might implement/use the interfaces are:

- Analog to Digital Convertor.

  An example is a 16 channel ADC. The driver implements interfaces asynCommon and asynInt32. It uses interface asynInt32Base. It can call asynManager:interruptStart and asynManager:interruptEnd to support interrupts. It can use reason and addr to decide which callbacks to call.
- Digital I/O module

An example is a 64 bit combination digital input and digital output module. The driver implements interfaces asynCommon and asynUInt32Digital. It uses interface asynUInt32DigitalBase. It can call asynManager:interruptStart and asynManager:interruptEnd to support interrupts. It can use reason, mask, and addr to decide which callbacks to call.

asynInt32

asynInt32 describes the methods implemented by drivers that use integers for communicating with a device.

```
typedef void (*interruptCallbackInt32)(void *userPvt, asynUser *pasynUser,
                                       epicsInt32 data);
typedef struct asynInt32Interrupt {
    int addr;
    asynUser *pasynUser;
    interruptCallbackInt32 callback;
    void *userPvt;
} asynInt32Interrupt;
#define asynInt32Type "asynInt32"
typedef struct asynInt32 {
    asynStatus (*write)(void *drvPvt, asynUser *pasynUser, epicsInt32 value);
    asynStatus (*read)(void *drvPvt, asynUser *pasynUser, epicsInt32 *value);
    asynStatus (*getBounds)(void *drvPvt, asynUser *pasynUser,
                            epicsInt32 *low, epicsInt32 *high);
    asynStatus (*registerInterruptUser)(void *drvPvt,asynUser *pasynUser,
                            interruptCallbackInt32 callback, void *userPvt,
                            void **registrarPvt);
    asynStatus (*cancelInterruptUser)(void *registrarPvt, asynUser *pasynUser);
} asynInt32;

/* asynInt32Base does the following:
   calls  registerInterface for asynInt32.
   Implements registerInterruptUser and cancelInterruptUser
   Provides default implementations of all methods.
   registerInterruptUser and cancelInterruptUser can be called
   directly rather than via queueRequest.
*/

#define asynInt32BaseType "asynInt32Base"
typedef struct asynInt32Base {
    asynStatus (*initialize)(const char *portName,
                             asynInterface *pint32Interface);
} asynInt32Base;
epicsShareExtern asynInt32Base *pasynInt32Base;
```

asynInt32

write	Write an integer value to the device.
read	Read an integer value from the device.
getBounds	Get the bounds. For example a 16 bit ADC might set low=−32768 and high = 32767.
registerInterruptUser	Registers a callback that will be called whenever new data is available. Since it can be called directly rather than via a queueRequest this method must not block.
cancelInterruptUser	Cancels the callback. Since it can be called directly rather than via a queueRequest this method must not block.

asynInt32Base is an interface and associated code that is used by drivers that implement interface asynInt32. asynInt32Base provides code to handle registerInterruptUser/cancelInterruptUser. The driver must itself call the

callbacks via calls to asynManager:interruptStart and asynManager:interruptEnd.

asynInt32Base

initialize	After a driver calls registerPort is can call: `pasynInt32Base->initialize(...` Any null methods in the asynInterface are replaced by default implementations. It is expected that most drivers use the default implementations of registerInterruptUser and cancelInterruptUser. Since these methods are called only during initialization they do not need to be called via a queueRequest callback.

The default implementation of each method does the following:

asynInt32

write	Reports an error "write is not supported" and returns asynError
read	Reports an error "read is not supported" and returns asynError
getBounds	Reports an error "getBounds is not supported" and returns asynError
registerInterruptUser	registers an interruot callback.
cancelInterruptUser	Cancels the callback

asynInt32SyncIO

asynInt32SyncIO describes a synchronous interface to asynInt32. The code that calls it must be willing to block.

```
#define asynInt32SyncIOType "asynInt32SyncIO"
typedef struct asynInt32SyncIO {
    asynStatus (*connect)(const char *port, int addr,
                          asynUser **ppasynUser, const char *drvInfo);
    asynStatus (*disconnect)(asynUser *pasynUser);
    asynStatus (*write)(asynUser *pasynUser, epicsInt32 value,double timeout);
    asynStatus (*read)(asynUser *pasynUser, epicsInt32 *pvalue,double timeout);
    asynStatus (*getBounds)(asynUser *pasynUser,
                    epicsInt32 *plow, epicsInt32 *phigh);
    asynStatus (*writeOnce)(const char *port, int addr,
                    epicsInt32 value,double timeout, const char *drvInfo);
    asynStatus (*readOnce)(const char *port, int addr,
                    epicsInt32 *pvalue,double timeout, const char *drvInfo);
    asynStatus (*getBoundsOnce)(const char *port, int addr,
                    epicsInt32 *plow, epicsInt32 *phigh,const char *drvInfo);
} asynInt32SyncIO;
epicsShareExtern asynInt32SyncIO *pasynInt32SyncIO;
```

asynInt32SyncIO

connect	Connects to a port and address, returns a pointer to an asynUser structure.
disconnect	Disconnect. This frees all resources allocated by create.
write	Calls pasynInt32->write and waits for the operation to complete or time out.

read	Calls pasynInt32->read and waits for the operation to complete or time out.
getBounds	Calls pasynInt32->getBounds and waits for the operation to complete or time out.
writeOnce	This does a connect, write, and disconnect.
readOnce	This does a connect, read, and disconnect.
getBoundsOnce	This does a connect, getBounds, and disconnect.

asynInt32Array

asynInt32Array describes the methods implemented by drivers that use arrays of integers for communicating with a device.

```
typedef void (*interruptCallbackInt32Array)(
              void *userPvt, asynUser *pasynUser, epicsInt32 *value,
              size_t nelements);
typedef struct asynInt32ArrayInterrupt {
    asynUser *pasynUser;
    int addr;
    interruptCallbackInt32Array callback;
    void *userPvt;
} asynInt32ArrayInterrupt;
#define asynInt32ArrayType "asynInt32Array"
typedef struct asynInt32Array {
    asynStatus (*write)(void *drvPvt, asynUser *pasynUser,
                        epicsInt32 *value, size_t nelements);
    asynStatus (*read)(void *drvPvt, asynUser *pasynUser,
                        epicsInt32 *value, size_t nelements, size_t *nIn);
    asynStatus (*registerInterruptUser)(void *drvPvt, asynUser *pasynUser,
              interruptCallbackInt32Array callback, void *userPvt,
              void **registrarPvt);
    asynStatus (*cancelInterruptUser)(void *drvPvt, asynUser *pasynUser);
} asynInt32Array;

#define asynInt32ArrayBaseType "asynInt32ArrayBase"
typedef struct asynInt32ArrayBase {
    asynStatus (*initialize)(const char *portName,
                             asynInterface *pint32ArrayInterface);
} asynInt32ArrayBase;
epicsShareExtern asynInt32ArrayBase *pasynInt32ArrayBase;
```

asynInt32Array

write	Write an array of integers to a device.
read	Read an array of integers from a device.
registerInterruptUser	Register a callback that is called whenever new data is available. Since it can be called directly rather than via a queueRequest this method must not block.
cancelInterruptUser	Cancel the callback. Callback and userPvt must match the values passed to registerInterruptUser. Since it can be called directly rather than via a queueRequest this method must not block.

asynUInt32Digital

asynUInt32Digital describes the methods for communicating via bits of an Int32 register.

```
typedef enum {
    interruptOnZeroToOne, interruptOnOneToZero, interruptOnBoth
} interruptReason;
```

```
typedef void (*interruptCallbackUInt32Digital)(void *userPvt,
                 asynUser *pasynUser, epicsUInt32 data);
typedef struct asynUInt32DigitalInterrupt {
    epicsUInt32 mask;
    int addr;
    asynUser *pasynUser;
    interruptCallbackUInt32Digital callback;
    void *userPvt;
} asynUInt32DigitalInterrupt;
#define asynUInt32DigitalType "asynUInt32Digital"
typedef struct asynUInt32Digital {
    asynStatus (*write)(void *drvPvt, asynUser *pasynUser,
         epicsUInt32 value, epicsUInt32 mask);
    asynStatus (*read)(void *drvPvt, asynUser *pasynUser,
        epicsUInt32 *value, epicsUInt32 mask);
    asynStatus (*setInterrupt)(void *drvPvt, asynUser *pasynUser,
        epicsUInt32 mask, interruptReason reason);
    asynStatus (*clearInterrupt)(void *drvPvt, asynUser *pasynUser,
        epicsUInt32 mask);
    asynStatus (*getInterrupt)(void *drvPvt, asynUser *pasynUser,
        epicsUInt32 *mask, interruptReason reason);
    asynStatus (*registerInterruptUser)(void *drvPvt, asynUser *pasynUser,
        interruptCallbackUInt32Digital callback,void *userPvt,epicsUInt32 mask,
        void **registrarPvt);
    asynStatus (*cancelInterruptUser)(void *registrarPvt, asynUser *pasynUser);
} asynUInt32Digital;

/* asynUInt32DigitalBase does the following:
   calls  registerInterface for asynUInt32Digital.
   Implements registerInterruptUser and cancelInterruptUser
   Provides default implementations of all methods.
   registerInterruptUser and cancelInterruptUser can be called
   directly rather than via queueRequest.
*/

#define asynUInt32DigitalBaseType "asynUInt32DigitalBase"
typedef struct asynUInt32DigitalBase {
    asynStatus (*initialize)(const char *portName,
                             asynInterface *pasynUInt32DigitalInterface);
} asynUInt32DigitalBase;
epicsShareExtern asynUInt32DigitalBase *pasynUInt32DigitalBase;
```

asynUInt32Digital

write	Modify the bits specified by mask with the corresponding bits in value.
read	Read the bits specified by mask into value. The other bits of value will be set to 0.
setInterrupt	Set the bits specified by mask to interrupt for reason.
clearInterrupt	Clear the interrupt bits specified by mask.
getInterrupt	Set each bit of mask that is enabled for reason.
registerInterruptUser	Register a callback that will be called whenever the driver detects a change in any of the bits specified by mask. Since it can be called directly rather than via a queueRequest this method must not block.
cancelInterruptUser	Cancels the registered callback. Since it can be called directly rather than via a queueRequest this method must not block.

asynUInt32DigitalBase is an interface and associated code that is used by drivers that implement interface asynUInt32Digital. asynUInt32DigitalBase provides code to implement registerInterruptUser and

cancelInterruptUser.

asynUInt32DigitalBase

initialize	After a driver calls registerPort is can call: `pasynUInt32DigitalBase->initialize(...` Any null methods in the asynInterface are replaced by default implementations. It is expected that most drivers use the default implementations of registerInterruptUser and cancelInterruptUser. Since these methods are called only during initialization they do not need to be called via a queueRequest callback.

The default implementation of each method does the following:

asynUInt32Digital

write	Reports an error "write is not supported" and returns asynError
read	Reports an error "read is not supported" and returns asynError
setInterrupt	Reports an error "setInterrupt is not supported" and returns asynError
clearInterrupt	Reports an error "clearInterrupt is not supported" and returns asynError
getInterrupt	Reports an error "getInterrupt is not supported" and returns asynError
registerInterruptUser	registers the interrupt user. The low level driver must call the registered callbacks via calls to asynManager:interruptStart and asynManager:interruptEnd.
cancelInterruptUser	Cancels the callback

asynUInt32DigitalSyncIO

asynUInt32DigitalSyncIO describes a synchronous interrace to asynUInt32Digital. The code that calls it must be willing to block.

```
#define asynUInt32DigitalSyncIOType "asynUInt32DigitalSyncIO"
typedef struct asynUInt32DigitalSyncIO {
    asynStatus (*connect)(const char *port, int addr,
                          asynUser **ppasynUser, const char *drvInfo);
    asynStatus (*disconnect)(asynUser *pasynUser);
    asynStatus (*write)(asynUser *pasynUser,
                          epicsUInt32 value,epicsUInt32 mask,double timeout);
    asynStatus (*read)(asynUser *pasynUser,
                          epicsUInt32 *pvalue,epicsUInt32 mask,double timeout);
    asynStatus (*setInterrupt)(asynUser *pasynUser,
                          epicsUInt32 mask, interruptReason reason,double timeout);
    asynStatus (*clearInterrupt)(asynUser *pasynUser,
                          epicsUInt32 mask,double timeout);
    asynStatus (*getInterrupt)(asynUser *pasynUser,
                          epicsUInt32 *mask, interruptReason reason,double timeout);
    asynStatus (*writeOnce)(const char *port, int addr,
                          epicsUInt32 value,epicsUInt32 mask,double timeout,
                          const char *drvInfo);
    asynStatus (*readOnce)(const char *port, int addr,
                          epicsUInt32 *pvalue,epicsUInt32 mask,double timeout,
                          const char *drvInfo);
    asynStatus (*setInterruptOnce)(const char *port, int addr,
                          epicsUInt32 mask, interruptReason reason,double timeout,
```

```
                        const char *drvInfo);
    asynStatus (*clearInterruptOnce)(const char *port, int addr,
                        epicsUInt32 mask,double timeout,const char *drvInfo);
    asynStatus (*getInterruptOnce)(const char *port, int addr,
                        epicsUInt32 *mask, interruptReason reason,double timeout,
                        const char *drvInfo);
} asynUInt32DigitalSyncIO;
epicsShareExtern asynUInt32DigitalSyncIO *pasynUInt32DigitalSyncIO;
```

asynUInt32DigitalSyncIO

connect	Connects to a port and address, returns a pointer to an asynUser structure.
disconnect	Disconnect. This frees all resources allocated by create.
write	Calls pasynUInt32Digital−>write and waits for the operation to complete or time out.
read	Calls pasynUInt32Digital−>read and waits for the operation to complete or time out.
setInterrupt	Calls pasynUInt32Digital−>setInterrupt and waits for the operation to complete or time out.
clearInterrupt	Calls pasynUInt32Digital−>clearInterrupt and waits for the operation to complete or time out.
getInterrupt	Calls pasynUInt32Digital−>getInterrupt and waits for the operation to complete or time out.
writeOnce,...,getInterruptOnce	Does a connect, (write,...,getInterrupt), and disconnect.

asynFloat64

asynFloat64 describes the methods for communicating via IEEE double precision float values.

```
typedef void (*interruptCallbackFloat64)(void *userPvt, asynUser *pasynUser,
            epicsFloat64 data);
typedef struct asynFloat64Interrupt {
    asynUser *pasynUser;
    int addr;
    interruptCallbackFloat64 callback;
    void *userPvt;
} asynFloat64Interrupt;
#define asynFloat64Type "asynFloat64"
typedef struct asynFloat64 {
    asynStatus (*write)(void *drvPvt, asynUser *pasynUser, epicsFloat64 value);
    asynStatus (*read)(void *drvPvt, asynUser *pasynUser, epicsFloat64 *value);
    asynStatus (*registerInterruptUser)(void *drvPvt, asynUser *pasynUser,
        interruptCallbackFloat64 callback, void *userPvt,void **registrarPvt);
    asynStatus (*cancelInterruptUser)(void *registrarPvt, asynUser *pasynUser);
} asynFloat64;

/* asynFloat64Base does the following:
   calls  registerInterface for asynFloat64.
   Implements registerInterruptUser and cancelInterruptUser
   Provides default implementations of all methods.
   registerInterruptUser and cancelInterruptUser can be called
   directly rather than via queueRequest.
*/

#define asynFloat64BaseType "asynFloat64Base"
typedef struct asynFloat64Base {
```

```
    asynStatus (*initialize)(const char *portName,
                             asynInterface *pasynFloat64Interface);
} asynFloat64Base;
epicsShareExtern asynFloat64Base *pasynFloat64Base;
```

asynFloat64

write	Write a value.
read	Read a value.
registerInterruptUser	Register a callback that is called whenever new data is available. Since it can be called directly rather than via a queueRequest this method must not block.
cancelInterruptUser	Cancel the callback. Since it can be called directly rather than via a queueRequest this method must not block.

asynFloat64Base

asynFloat64Base is an interface and associated code that is used by drivers that implement interface asynFloat64. asynFloat64Base provides code to implement registerInterruptUser and cancelInterruptUser.

```
#define asynFloat64BaseType "asynFloat64Base"
typedef struct asynFloat64Base {
    asynStatus (*initialize)(const char *portName,
                             asynInterface *pasynFloat64Interface);
} asynFloat64Base;
epicsShareExtern asynFloat64Base *pasynFloat64Base;
```

asynFloat64Base

initialize	After a driver calls registerPort is can call: `pasynFloat64Base->initialize(...` Any null methods in the asynInterface are replaced by default implementations. It is expected that most drivers use the default implementations of registerInterruptUser and cancelInterruptUser. Since these methods are called only during initialization they do not need to be called via a queueRequest callback.

The default implementation of each method does the following:

asynFloat64

write	Reports an error "write is not supported" and returns asynError
read	Reports an error "read is not supported" and returns asynError
registerInterruptUser	registers the interrupt user. The low level driver must call the registered callbacks via calls to asynManager:interruptStart and asynManager:interruptEnd.
cancelInterruptUser	Cancels the callback

asynFloat64SyncIO

asynFloat64SyncIO describes a synchronous interrace to asynFloat64. The code that calls it must be willing to block.

```
#define asynFloat64SyncIOType "asynFloat64SyncIO"
typedef struct asynFloat64SyncIO {
```

```
    asynStatus (*connect)(const char *port, int addr,
                        asynUser **ppasynUser, const char *drvInfo);
    asynStatus (*disconnect)(asynUser *pasynUser);
    asynStatus (*write)(asynUser *pasynUser,epicsFloat64 value,double timeout);
    asynStatus (*read)(asynUser *pasynUser,epicsFloat64 *pvalue,double timeout);
    asynStatus (*writeOnce)(const char *port, int addr,
                        epicsFloat64 value,double timeout,const char *drvInfo);
    asynStatus (*readOnce)(const char *port, int addr,
                        epicsFloat64 *pvalue,double timeout,const char *drvInfo);
} asynFloat64SyncIO;
epicsShareExtern asynFloat64SyncIO *pasynFloat64SyncIO;
```

asynFloat64SyncIO

connect	Connects to a port and address, returns a pointer to an asynUser structure.
disconnect	Disconnect. This frees all resources allocated by create.
write	Calls pasynFloat64−>write and waits for the operation to complete or time out.
read	Calls pasynFloat64−>read and waits for the operation to complete or time out.
writeOnce	This does a connect, write, and disconnect.
readOnce	This does a connect, read, and disconnect.

asynFloat64Array

asynFloat64Array describes the methods for communicating via IEEE double precision float values.

```
typedef void (*interruptCallbackFloat64Array)(
              void *userPvt, asynUser *pasynUser, epicsFloat64 *data,
              size_t nelements);
typedef struct asynFloat64ArrayInterrupt {
    asynUser *pasynUser;
    int addr;
    interruptCallbackFloat64Array callback;
    void *userPvt;
} asynFloat64ArrayInterrupt;
#define asynFloat64ArrayType "asynFloat64Array"
typedef struct asynFloat64Array {
    asynStatus (*write)(void *drvPvt, asynUser *pasynUser,
                        epicsFloat64 *value, size_t nelements);
    asynStatus (*read)(void *drvPvt, asynUser *pasynUser,
                        epicsFloat64 *value, size_t nelements, size_t *nIn);
    asynStatus (*registerInterruptUser)(void *drvPvt, asynUser *pasynUser,
         interruptCallbackFloat64Array callback,
         void *userPvt,void **registrarPvt);
    asynStatus (*cancelInterruptUser)(void *drvPvt, asynUser *pasynUser);
} asynFloat64Array;

/* asynFloat64ArrayBase does the following:
   calls  registerInterface for asynFloat64Array.
   Implements registerInterruptUser and cancelInterruptUser
   Provides default implementations of all methods.
   registerInterruptUser and cancelInterruptUser can be called
   directly rather than via queueRequest.
*/

#define asynFloat64ArrayBaseType "asynFloat64ArrayBase"
typedef struct asynFloat64ArrayBase {
    asynStatus (*initialize)(const char *portName,
                             asynInterface *pfloat64ArrayInterface);
```

```
} asynFloat64ArrayBase;
epicsShareExtern asynFloat64ArrayBase *pasynFloat64ArrayBase;
```

asynFloat64Array

write	Write an array of values.
read	Read an array of values.
registerInterruptUser	Register a callback that is called whenever new data is available.
cancelInterruptUser	Cancel the callback

Standard Interpose Interfaces

asynInterposeEos

This can be used to simulate EOS processing for asynOctet if the port driver doesn't provide EOS support. If an EOS is specified it looks for the eos on each read. It is started by the shell command:

```
asynInterposeEosConfig port addr processEosIn processEosOut
```

where

- port is the name of the port.
- addr is the address
- processEosIn (0,1) means (do not, do) implement eosIn commands.
- processEosOut (0,1) means (do not, do) implement eosOut commands.

This command should appear immediately after the command that initializes a port Some drivers provide configuration options to call this automatically.

asynInterposeFlush

This can be used to simulate flush processing for asynOctet if the port driver doesn't provide support for flush. It just reads and discards characters until no more characters arive before timeout seconds have occured. It is started by the shell command:

```
asynInterposeFlushConfig port addr timeout
```

where

- port is the name of the port.
- addr is the address
- timeout is the time to wait for more characters

this command should appear immediately after the command that initializes a port

Generic Device Support for EPICS records

NOTE: The generic device support for epics records is still being developed.

Generic device support is provided for standard EPICS records. This support should be usable for a large class of low level register based drivers. For truly complicated devices other support is required. This release provides the following:

- devAsynInt32 − support for drivers that implement interface asynInt32
- devAsynUInt32Digital − support for drivers that implement interface asynUInt32Digital
- devAsynFloat64 − support for drivers that implement interface asynFloat64
- devAsynOctet − support for drivers that implement interface asynOctet
- devEpics − This is just a single file devEpics.dbd that includes the dbd files for the above support.

The support uses the following conventions for DTYP and INP. OUT fields are the same as INP.

```
field(DTYP,"asynXXX")
field(INP,"@asyn(portName,addr,timeout) drvParams")
 or
field(INP,"@asynMask(portName,addr,mask,timeout) drvParams")
```

where

- XXX − The name of the type of interface supported.
- portName − The name of the port.
- addr − The address. If addr is not specified the default is 0.
- mask − This is for devAsynUInt32Digital.
- timeout − The timeout value for asynUser.timeout. If not specified the default is 1.0.
- drvParams − This is passed to the low level driver via the asynDrvUser interface.

For example:

```
field(DTYP,"asynInt32")
field(INP,"@asyn(portA,0,.1) thisIsForDriver")
```

asynManager interrupts and EPICS device support

devAsynInt32, devAsynFloat64, devAsynUInt32Digital, and devAsynPctet call registerInterruptUser for input record. The callback is used in one of two ways:

- Input Records except Average

  It is used to support SCAN = "I/O Intr".
- Input records that are averaged, i.e. asynInt32Average or asynFloat64Average.

  These records are normally scanned periodically. The registerInterruptUser callback is used to calculate an average value between record processes.

asynInt32 device support

The following support is available:

```
device(ai,INST_IO,asynAiInt32,"asynInt32")
device(ai,INST_IO,asynAiInt32Average,"asynInt32Average")
device(ao,INST_IO,asynAoInt32,"asynInt32")
device(mbbi,INST_IO,asynMbbiInt32,"asynInt32")
device(mbbo,INST_IO,asynMbboInt32,"asynInt32")
device(longin,INST_IO,asynLiInt32,"asynInt32")
device(longout,INST_IO,asynLoInt32,"asynInt32")
```

devAsynInt32.c provides EPICS device support for drivers that implement interface asynInt32.

- aiRecord

  A value is given to rval. Linear conversions are supported if the driver implements getBounds.

  - asynInt32 − SCAN "I/O Intr" is supported. If the record is "I/O Intr" scanned then when the registerInterruptUser callback is called, it saves the value and calls scanIoRequest. When the record is processed the saved value is put into rval. If the record is not "I/O Intr" scanned then each time the record is processed, a new value is read via a call to pasynInt32−>read.
  - asynInt32Average − The registerInterruptUser callback adds the new value to a sum and also increments the number of samples. When the record is processed the average is computed and the sum and number of set to zero.
- aoRecord

  rval is written. Linear conversions are supported if the driver properly implements getBounds.
- longinRecord

  A value is given to val. Each time the record is processed a new value is read. SCAN "I/O Intr" is supported similar to aiRecord.
- longoutRecord

  val is written.
- mbbiRecord

  A value is given to rval. mask is computed from nobt and shft. Each time the record is processed a new value is read. SCAN "I/O Intr" is supported similar to aiRecord.
- mbboRecord

  rval is written. mask is computed from nobt and shft.

Analog Input Example Records

```
record(ai,"aiInt32") {
     field(SCAN,"I/O Intr")
     field(DTYP,"asynInt32")
     field(INP,"@asyn($(port),$(addr))")
     field(EGUF,"10.0")
     field(EGUL,"−10.0")
     field(PREC,"3")
}
```

```
record(ai,"aiInt32Average") {
     field(SCAN,"10 second")
     field(DTYP,"asynInt32Average")
     field(INP,"@asyn($(port),$(addr))")
     field(EGUF,"10.0")
     field(EGUL,"−10.0")
     field(PREC,"3")
}
```

Analog Output Example Record

```
 record(ao,"aoInt32") {
     field(DTYP,"asynInt32")
     field(OUT,"@asyn($(port),$(addr))")
     field(EGUF,"10.0")
     field(EGUL,"−10.0")
     field(PREC,"3")
}
```

Long Input Example Records

```
record(longin,"liInt32") {
     field(SCAN,"I/O Intr")
     field(DTYP,"asynInt32")
     field(INP,"@asyn($(port),$(addr))")
}
```

Long Output Example Record

```
 record(longout,"loInt32") {
     field(DTYP,"asynInt32")
     field(OUT,"@asyn($(port),$(addr))")
}
```

Multibit Binary Input Example Records

```
record(mbbi,"mbbiInt32") {
     field(SCAN,"I/O Intr")
     field(DTYP,"asynInt32")
     field(INP,"@asyn($(port),$(addr))")
     field(NOBT,"2")
     field(SHFT,"2")
     field(ZRST,"zeroVal")
     field(ONST,"oneVal")
     field(TWST,"twoVal")
     field(THST,"threeVal")
}
```

Multibit Binary Output Example Record

```
 record(mbbo,"mbboInt32") {
     field(DTYP,"asynInt32")
     field(OUT,"@asyn($(port),$(addr))")
     field(NOBT,"2")
     field(SHFT,"16")
     field(ZRST,"zeroVal")
     field(ONST,"oneVal")
     field(TWST,"twoVal")
     field(THST,"threeVal")
```

```
}
```

devAsynUInt32Digital

The following support is available:

```
device(bi,INST_IO,asynBiUInt32Digital,"asynUInt32Digital")
device(bo,INST_IO,asynBoUInt32Digital,"asynUInt32Digital")
device(longin,INST_IO,asynLiUInt32Digital,"asynUInt32Digital")
device(longout,INST_IO,asynLoUInt32Digital,"asynUInt32Digital")
device(mbbi,INST_IO,asynMbbiUInt32Digital,"asynUInt32Digital")
device(mbbo,INST_IO,asynMbboUInt32Digital,"asynUInt32Digital")
device(mbbiDirect,INST_IO,asynMbbiDirectUInt32Digital,"asynUInt32Digital")
device(mbboDirect,INST_IO,asynMbboDirectUInt32Digital,"asynUInt32Digital")
```

devAsynUInt32Digital.c provides EPICS device support for drivers that implement interface asynUInt32Digital. The INP or OUT field must define asynMask. The mask specified in the argument to asynMask is used in the calls to asymUInt32Digital methods. In addition it is used to set the mask fields in bi and bo records and the mask and shft fields in mbbi, mbbo, mbbiDirect, and mbboDirect records.

mask

- biRecord

  A value is given to rval. asynInt32 − SCAN "I/O Intr" is supported. If the record is "I/O Intr" scanned then when the registerInterruptUser callback is called, it saves the value and calls scanIoRequest. When the record is processed the saved value is put into rval. If the record is not "I/O Intr" scanned then each time the record is processed, a new value is read via a call to pasynUInt32Digital−>read.

- boRecord

  rval is written.
- longinRecord

  A value is given to val. Each time the record is processed a new value is read. SCAN "I/O Intr" is supported similar to aiRecord.

- longoutRecord

  val is written.
- mbbiRecord

  A value is given to rval. Each time the record is processed a new value is read. SCAN "I/O Intr" is supported similar to aiRecord.
- mbboRecord

  rval is written.
- mbbiDirectRecord

  A value is given to rval. Each time the record is processed a new value is read. SCAN "I/O Intr" is supported similar to aiRecord.
- mbboDirectRecord

rval is written.

Binary Input Example Record

```
record(bi,"biUInt32Bit0") {
    field(SCAN,"I/O Intr")
    field(DTYP,"asynUInt32Digital")
    field(INP,"@asynMask( $(port) , 0, 0x1 , 1.0) ")
    field(ZNAM,"zero")
    field(ONAM,"one")
}
```

Binary Output Example Record

```
record(bi,"boUInt32Bit2") {
    field(DTYP,"asynUInt32Digital")
    field(INP,"@asynMask( $(port) , 0, 0x4 , 1.0) ")
    field(ZNAM,"zero")
    field(ONAM,"one")
}
```

Long Input Example Record

```
record(longin,"liUInt32") {
    field(SCAN,"I/O Intr")
    field(DTYP,"asynUInt32Digital")
    field(INP,"@asynMask( $(port) , 0, 0xffffffff , 1.0) ")
}
```

Long Output Example Record

```
record(longout,"loUInt32") {
    field(DTYP,"asynUInt32Digital")
    field(INP,"@asynMask( $(port) , 0, 0xffffffff , 1.0) ")
}
```

Multibit Input Example Record

```
record(mbbi,"mbbiUInt32") {
    field(SCAN,"I/O Intr")
    field(DTYP,"asynUInt32Digital")
    field(INP,"@asynMask( digital , 0, 0x3 , 1.0) ")
    field(ZRST,"zero")
    field(ONST,"one")
    field(TWST,"two")
    field(THST,"three")
    field(ZRVL,"0x0")
    field(ONVL,"0x1")
    field(TWVL,"0x2")
    field(THVL,"0x3")
}
```

Multibit Output Example Record

```
record(mbbo,"mbboUInt32") {
    field(DTYP,"asynUInt32Digital")
    field(OUT,"@asynMask( digital , 0, 0x7 , 1.0) ")
    field(ZRST,"zero")
```

```
    field(ONST,"one")
    field(TWST,"two")
    field(THST,"three")
    field(FRST,"four")
    field(FVST,"five")
    field(SXST,"six")
    field(SVST,"seven")
    field(ZRVL,"0x0")
    field(ONVL,"0x1")
    field(TWVL,"0x2")
    field(THVL,"0x3")
    field(FRVL,"0x4")
    field(FVVL,"0x5")
    field(SXVL,"0x6")
    field(SVVL,"0x7")
}
```

devAsynFloat64 device support

The following support is available:

```
device(ai,INST_IO,asynAiFloat64,"asynFloat64")
device(ai,INST_IO,asynAiFloat64Average,"asynFloat64Average")
device(ao,INST_IO,asynAoFloat64,"asynFloat64")
```

devAsynFloat64.c provides EPICS device support for drivers that implement interface asynFloat64.

- aiRecord

  A value is given to rval. Linear conversions are supported if the driver properly implements getBounds.

  - asynFloat64 − SCAN "I/O Intr" is supported. If the record is "I/O Intr" scanned then when the registerInterruptUser callback is called, it saves the value and calls scanIoRequest. When the record is processed the saved value is put into rval. If the record is not "I/O Intr" scanned then each time the record is processed, a new value is read via a call to pasynFloat64−>read.
  - asynFloat64Average − The registerInterruptUser callback adds the new value to a sum and also increments the number of samples. When the record is processed the average is computed and the sum and number of set to zero.
- aoRecord

  rval is written. Linear conversions are supported if the driver properly implements getBounds.

octet device support

The following support is available:

```
device(stringin,INST_IO,asynSiOctetCmdResponse,"asynOctetCmdResponse")
device(stringin,INST_IO,asynSiOctetWriteRead,"asynOctetWriteRead")
device(stringin,INST_IO,asynSiOctetRead,"asynOctetRead")
device(stringout,INST_IO,asynSoOctetWrite,"asynOctetWrite")
device(waveform,INST_IO,asynWfOctetCmdResponse,"asynOctetCmdResponse")
device(waveform,INST_IO,asynWfOctetWriteRead,"asynOctetWriteRead")
device(waveform,INST_IO,asynWfOctetRead,"asynOctetRead")
device(waveform,INST_IO,asynWfOctetWrite,"asynOctetWrite")
```

Support for drivers that implement interface asynOctet. The support is for stringin/stringout and waveform records. The waveform support is similar to the string support. The waveform records must define FTVL to be CHAR or UCHAR, i.e. it must be an array of octets. The waveform provides the following features not provided by the string support:

- unlimited size − string records hold a maximum of 40 characters.
- non ascii − Thus arbitrary octet arrays are supported.

Four types of support are provided:

- CmdResponse The INP field is of the form:

  ```
  field(INP,"@asyn(portName,addr,timeout) cmd")
  ```

  During record initialization, cmd is converted by dbTranslateEscape. The resultant string is the command to send to the device. When the record is processed the command is sent to the device and the response read into the record.
- WriteRead The INP field is of the form:

  ```
  field(INP,"@asyn(portName,addr,timeout) pvname")
  ```

  pvname must refer to a field in a record in the same IOC. During record initialization the pvname is locatedd. When the record is processed dbGet is called to read the current value of pvname. This value is sent to the device. A read is then issued and the result stored in the record. For asynSiOctetWriteRead, the value obtained from pvname is passed through dbTranslateEscape before sending it. For asynWfOctetWriteRead it is not passed through dbTranslateEscape.
- Write The INP(OUT) field is of the form:

  ```
  field(INP,"@asyn(portName,addr,timeout) drvUser")
  ```

  drvUser is information that is passed to the portDriver if it implements interface asynDrvUser. When the record is processed the value stored in the record is sent to the device.
- Read The INP field is of the form:

  ```
  field(INP,"@asyn(portName,addr,timeout) drvUser")
  ```

  drvUser is information that is passed to the portDriver if it implements interface asynDrvUser. When the record is processed a read request is made. The result is read into the record.

asynRecord: Generic EPICS Record Support

A special record type asynRecord is provided. Details are described in asynRecord. This section provides a brief description of how to use it.

Each IOC can load one or more instances of asynRecord. An example is:

```
cd $(ASYN)
dbLoadRecords("db/asynRecord.db","P=asyn,R=Test,PORT=L0,ADDR=15,IMAX=0,OMAX=0")
```

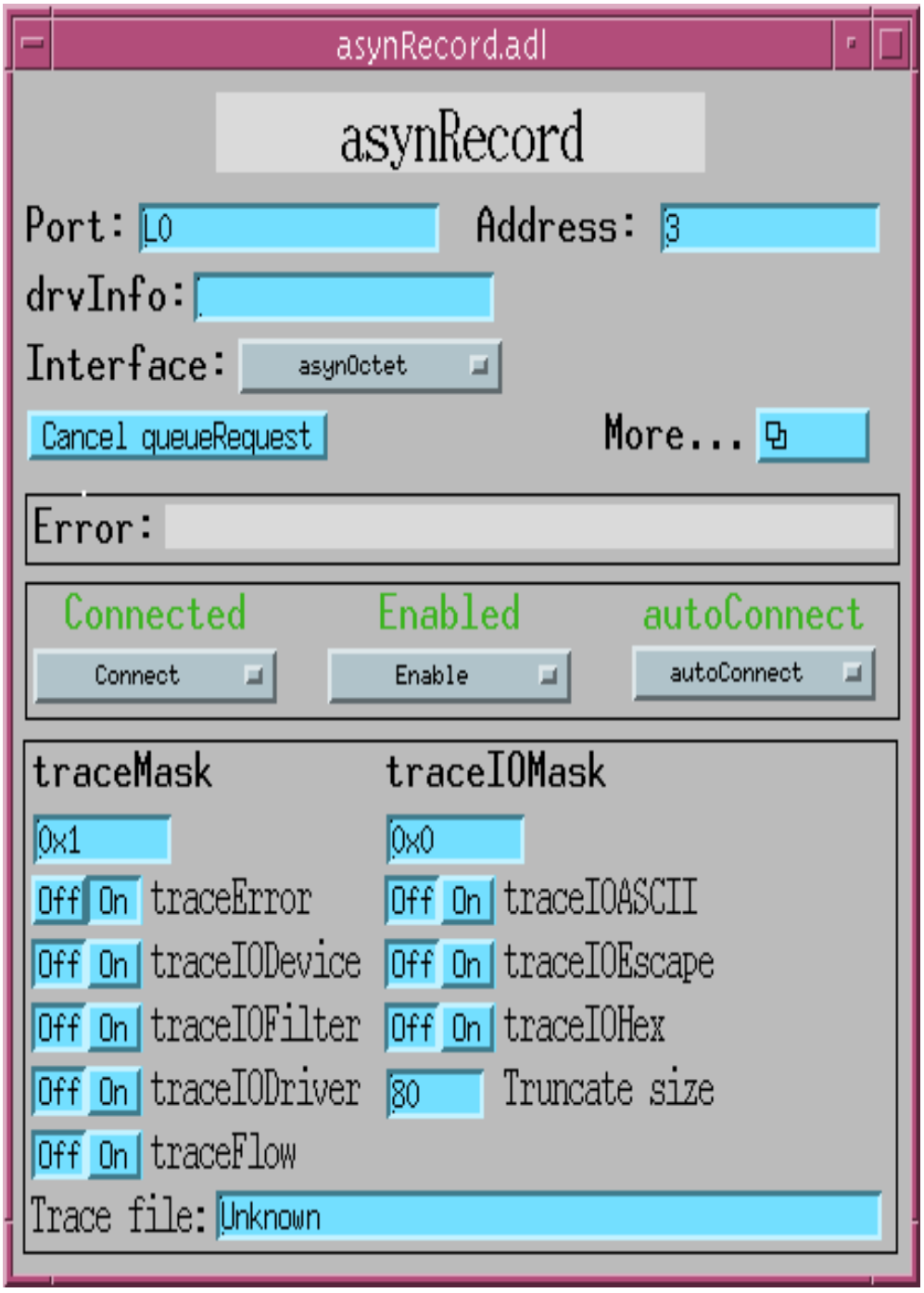
The example creates a record with name "asynTest" (formed from the concatenation of the P and R macros) that will connect to port "L0" and addr 15. After the ioc is started, it is possible to change PORT and/or ADDR. Thus,

a single record can be used to access all asyn devices connected to the IOC. Multiple records are only needed if one or more devices need a dedicated record.

An medm display is available for accessing an asynRecord. It is started as follows:

```
cd <asyn>/medm
medm -x -macro "P=asyn,R=Test" asynRecord.adl
```

The following medm display appears.



Example

The following reads from a device via octet messages:

```
#include <asynDriver.h>
...
#define BUFFER_SIZE 80
typedef struct myData {
    epicsEventId done;
    asynOctet    *pasynOctet;
    void         *drvPvt;
    char         buffer[BUFFER_SIZE];
}myData;

static void queueCallback(asynUser *pasynUser) {
    myData     *pmydata = (myData *)pasynUser->userPvt;
    asynOctet  *pasynOctet = pmydata->pasynOctet;
    void       *drvPvt = pmydata->drvPvt;
    asynStatus status;
    int        writeBytes,readBytes;
    int        eomReason;

    asynPrint(pasynUser,ASYN_TRACE_FLOW,"queueCallback entered\n");
    status = pasynOctet->write(drvPvt,pasynUser,pmydata->buffer,
              strlen(pmydata->buffer),&writeBytes);
    if(status!=asynSuccess) {
        asynPrint(pasynUser,ASYN_TRACE_ERROR,
            "queueCallback write failed %s\n",pasynUser->errorMessage);
    } else {
        asynPrintIO(pasynUser,ASYN_TRACEIO_DEVICE,
            pmydata->buffer,strlen(pmydata->buffer),
            "queueCallback write sent %d bytes\n",writeBytes);
    }
    status = pasynOctet->read(drvPvt,pasynUser,pmydata->buffer,
         BUFFER_SIZE,&readBytes,&eomReason);
    if(status!=asynSuccess) {
        asynPrint(pasynUser,ASYN_TRACE_ERROR,
            "queueCallback read failed %s\n",pasynUser->errorMessage);
    } else {
        asynPrintIO(pasynUser,ASYN_TRACEIO_DEVICE,
            pmydata->buffer,BUFFER_SIZE,
            "queueCallback read returned: retlen %d eomReason 0x%x data %s\n",
            readBytes,eomReason,pmydata->buffer);
    }
    if(pmydata->done) epicsEventSignal(pmydata->done);
}

static void asynExample(const char *port,int addr,const char *message)
{
    myData        *pmyData;
    asynUser      *pasynUser;
    asynStatus    status;
    asynInterface *pasynInterface;
    int           canBlock;

    pmyData = (myData *)pasynManager->memMalloc(sizeof(myData));
    memset(pmyData,0,sizeof(myData));
    strcpy(pmyData->buffer,message);
    pasynUser = pasynManager->createAsynUser(queueCallback,0);
    pasynUser->userPvt = pmyData;
    status = pasynManager->connectDevice(pasynUser,port,addr);
```

```
    if(status!=asynSuccess) {
        printf("can't connect to serialPort1 %s\n",pasynUser->errorMessage);
        exit(1);
    }
    pasynInterface = pasynManager->findInterface(
        pasynUser,asynOctetType,1);
    if(!pasynInterface) {
        printf("%s driver not supported\n",asynOctetType);
        exit(-1);
    }
    pmyData->pasynOctet = (asynOctet *)pasynInterface->pinterface;
    pmyData->drvPvt = pasynInterface->drvPvt;
    canBlock = 0;
    pasynManager->canBlock(pasynUser,&canBlock);
    if(canBlock) pmyData->done = epicsEventCreate(epicsEventEmpty);
    status = pasynManager->queueRequest(pasynUser,asynQueuePriorityLow, 0.0);
    if(status) {
        asynPrint(pasynUser,ASYN_TRACE_ERROR,
            "queueRequest failed %s\n",pasynUser->errorMessage);
    }
    if(canBlock) epicsEventWait(pmyData->done);
    status = pasynManager->freeAsynUser(pasynUser);
    if(status) {
        asynPrint(pasynUser,ASYN_TRACE_ERROR,
            "freeAsynUser failed %s\n",pasynUser->errorMessage);
    }
    epicsEventDestroy(pmyData->done);
    pasynManager->memFree(pasynUser->userPvt,sizeof(myData));
}
```

The flow of control is as follows:

1. A port driver calls registerPort. This step is not shown in the above example.
2. asynExample allocates myData and an asynUser.
3. asynExample connects to a device and to the asynOctet interface for the port driver.
4. When it is ready to communicate with the driver it calls queueRequest.
5. queueCallback is called. It calls the port driver's write and read methods.

Test Application

The asynDriver distribution includes code to test asynDriver. It is also an example of how to interface to asynManager. The example resides in <top>/testApp and contains the following components:

```
Db/
   test.db
adl/
   test.adl
src/
   devAsynTest.c
   devAsynTest.dbd
   echoDriver.c
   interposeInterface.c
```

echoDriver is a port driver that echos messages it receives. It implements asynCommon and asynOctet. When asynOctet:write is called it saves the message. When asynOctet:read is called, the saved message is returned and the message is flushed. echoDriverInit has an argument that determines if it acts like a multiDevice or a single device port driver.

An instance of echoDriver is started via the iocsh command:

```
echoDriverInit(portName,delay,noAutoConnect,multiDevice)
```

where

- portName − the port name for this instance.
- delay − The time to delay after a read or write. If delay is 0 then echoDriver registers as a synchronous port driver, i.e. bit ASYN_CANBLOCK of attributes is not set. If delay>0 then ASYN_CANBLOCK is set.
- noAutoConnect − Determines initial setting for port.
- multiDevice − If true then it supports two devices with addresses 0 and 1. If false it does not set ASYN_MULTIDEVICE, i.e. it only supports a single device.

test.db is a template containing three records: a calc record, which forward links to a stringout record which forward links to a stringin record. The string records attach to the device support supplied by devAsynTest.c. The stringout and stringin records share the same asynUser. When the stringout record processes it:

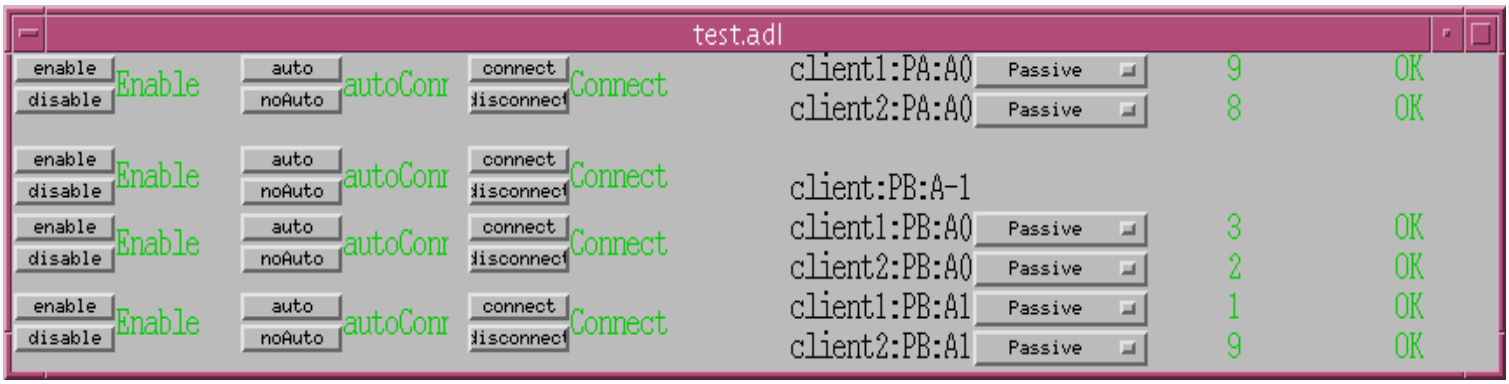
- Fetches the current value from the calc record (converted to ascii).
- Calls pasynManager−>lock.
- Calls pasynManager−>queueRequest.
- The callback calls pasynOctet−>write and then asks for the record to complete processing.
- The stringout record forward links to the stringin record.

The stringin records does the following:

- Calls pasynManager−>queueRequest.
- The callback routine:
  - ♦ calls pasynOctet−>read
  - ♦ Checks what it received vs what the stringout record wrote. If the values match, it sets its VAL field to "OK", otherwise it writes an error message into VAL.
  - ♦ Asks for the record to complete processing.
- The stringin record calls pasynManager−>unlock.

devAsynTest also does additional checking for connect state and enable/disable.

Executing "medm −x test.adl" produces the display:



It assumes that an ioc has been started via:

```
cd <top>/iocBoot/ioctest
../../bin/solaris-sparc/test st.cmd
```

This starts two versions of echoDriver as port "A" and "B". port A acts as single device port. port B acts as a multiDevice port that has two devices. For each of the three possible devices, the st.cmd file starts up two sets of records from test.db The st.cmd file also loads a set of records from asynTest.db for port A and for port B and for each of the two devices attached to port B. It also loads a set of records from asynGeneric.db.

asynGpib

GPIB has additional features that are not supported by asynCommon and asynOctet. asynGpib defines two interfaces.

- asynGpib − This is the interface that device support calls. It provides the following:
  - A set of GPIB specific methods that device support can call.
  - Code that handles generic GPIB functions like SRQ polling.
  - A registerPort method which is called by GPIB port drivers.
- asynGpibPort − A set of methods implemented by GPIB drivers

asynGpibDriver.h

asynGpibDriver.h contains the following definitions:

```
/* GPIB Addressed Commands*/
#define IBGTL "\x01"    /* Go to local */
#define IBSDC "\x04"    /* Selective Device Clear */
#define IBGET "\x08"    /* Group Execute Trigger */
#define IBTCT "\x09"    /* Take Control */

/* GPIB Universal Commands*/
#define IBDCL 0x14      /* Device Clear */
#define IBLLO 0x11      /* Local Lockout */
#define IBSPE 0x18      /* Serial Poll Enable */
#define IBSPD 0x19      /* Serial Poll Disable */
#define IBUNT 0x5f      /* Untalk */
#define IBUNL 0x3f      /* Unlisten */

/* Talk, Listen, Secondary base addresses */
#define TADBASE    0x40   /* offset to GPIB listen address 0 */
#define LADBASE    0x20   /* offset to GPIB talk address 0 */
#define SADBASE    0x60   /* offset to GPIB secondary address 0 */

#define NUM_GPIB_ADDRESSES    32
#include "asynDriver.h"
#include "asynInt32.h"
#define asynGpibType "asynGpib"
/* GPIB drivers */
typedef struct asynGpib asynGpib;
typedef struct asynGpibPort asynGpibPort;
/*asynGpib defines methods called by gpib aware users*/
struct asynGpib{
    /*addressedCmd,...,ren are just passed to device handler*/
    asynStatus (*addressedCmd) (void *drvPvt,asynUser *pasynUser,
        const char *data, int length);
    asynStatus (*universalCmd) (void *drvPvt,asynUser *pasynUser,int cmd);
    asynStatus (*ifc) (void *drvPvt,asynUser *pasynUser);
```

```
    asynStatus (*ren) (void *drvPvt,asynUser *pasynUser, int onOff);
    /* The following are implemented by asynGpib */
    asynStatus (*pollAddr)(void *drvPvt,asynUser *pasynUser, int onOff);
    /* The following are called by low level gpib drivers */
    /*srqHappened is passed the pointer returned by registerPort*/
    void *(*registerPort)(
        const char *portName,
        int attributes,int autoConnect,
        asynGpibPort *pasynGpibPort, void *asynGpibPortPvt,
        unsigned int priority, unsigned int stackSize);
    void (*srqHappened)(void *asynGpibPvt);
};
epicsShareExtern asynGpib *pasynGpib;

struct asynGpibPort {
    /*asynCommon methods */
    void (*report)(void *drvPvt,FILE *fd,int details);
    asynStatus (*connect)(void *drvPvt,asynUser *pasynUser);
    asynStatus (*disconnect)(void *drvPvt,asynUser *pasynUser);
    /*asynOctet methods passed through from asynGpib*/
    asynStatus (*read)(void *drvPvt,asynUser *pasynUser,
                       char *data,int maxchars,int *nbytesTransfered,
                       int *eomReason);
    asynStatus (*write)(void *drvPvt,asynUser *pasynUser,
                       const char *data,int numchars,int *nbytesTransfered);
    asynStatus (*flush)(void *drvPvt,asynUser *pasynUser);
    asynStatus (*setEos)(void *drvPvt,asynUser *pasynUser,
                 const char *eos,int eoslen);
    asynStatus (*getEos)(void *drvPvt,asynUser *pasynUser,
                 char *eos, int eossize, int *eoslen);
    /*asynGpib methods passed thrtough from asynGpib*/
    asynStatus (*addressedCmd) (void *drvPvt,asynUser *pasynUser,
                 const char *data, int length);
    asynStatus (*universalCmd) (void *drvPvt, asynUser *pasynUser, int cmd);
    asynStatus (*ifc) (void *drvPvt,asynUser *pasynUser);
    asynStatus (*ren) (void *drvPvt,asynUser *pasynUser, int onOff);
    /*asynGpibPort specific methods */
    asynStatus (*srqStatus) (void *drvPvt,int *isSet);
    asynStatus (*srqEnable) (void *drvPvt, int onOff);
    asynStatus (*serialPollBegin) (void *drvPvt);
    asynStatus (*serialPoll) (void *drvPvt, int addr, double timeout,int *status);
    asynStatus (*serialPollEnd) (void *drvPvt);
};
```

asynGpib

asynGpib describes the interface for device support code. It provides gpib specific functions like SRQ handling. It makes calls to asynGpibPort. asynGpib.c implements asynCommon and asynOctet. It supports asynInt32 by using the methods from asynInt32Base. asynInt32 is the way asynGpib reports SRQs to asynUsers. An asynUser that wishes to receive SRQs calls pasynInt32−>registerInterruptUser and must set asynUser.reason = ASYN_REASON_SIGNAL.

asynGpib

addressedCmd	The request is passed to the low level driver.
universalCmd	The request is passed to the low level driver.
ifc	The request is passed to the low level driver.

ren	The request is passed to the low level driver.
pollAddr	Set SRQ polling on or off. onOff = (0,1) means (disable, enable) SRQ polling of specified address.
registerPort	Register a port. When asynGpib receives this request, it calls asynManager.registerPort.
srqHappened	Called by low level driver when it detects that a GPIB device issues an SRQ.

asynGpibPort

asynGpibPort is the interface that is implemented by gpib drivers, e.g. the VXI−11. It provides:

asynGpibPort

asynCommon methods	All the methods of asynCommon
asynOctet methods	All the methods of asynOctet
addressedCmd	Issue a GPIB addressed command.
universalCmd	Issue a GPIB universal command.
ifc	Issue a GPIB Interface Clear command.
ren	Issue a GPIB Remote Enable command
srqStatus	If return is asynSuccess then isSet is (0,1) if SRQ (is not, is) active. Normally only called by asynGpib.
srqEnable	Enable or disable SRQs. Normally only called by asynGpib.
serialPollBegin	Start of serial poll. Normally only called by asynGpib.
serialPoll	Poll the specified address and set status to the response. Normally only called by asynGpib.
serialPollEnd	End of serial poll. Normally only called by asynGpib.

Port Drivers

Local Serial Port

The drvAsynSerialPort driver supports devices connected to serial ports on the IOC.

Serial ports are configured with the `drvAsynSerialPortConfigure` and `asynSetOption` commands:

```
drvAsynSerialPortConfigure("portName","ttyName",priority,noAutoConnect,
     processEosIn,processEosOut)
asynSetOption("portName",addr,"key","value")
```

where the arguments are:

- portName − The portName that is registered with asynGpib.
- ttyName − The name of the local serial port (e.g. "/dev/ttyS0").
- priority − Priority at which the asyn I/O thread will run. If this is zero or missing,then epicsThreadPriorityMedium is used.
- addr − This argument is ignored since serial devices are configured with multiDevice=0.

- noAutoConnect − Zero or missing indicates that portThread should automatically connect. Non−zero if explicit connect command must be issued.
- processEosIn, processEosOut − If either is set asynOctetBase will call asynInterposeEosConfig.

The setEos and getEos methods have no effect and return asynError. The read method blocks until at least one character has been received or until a timeout occurs. The read method transfers as many characters as possible, limited by the specified count. asynInterposeEos can be used to support EOS.

The following table summarizes the drvAsynSerialPort driver asynSetOption keys and values. Default values are enclosed in square brackets.

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
ctrscts	[N] Y

The clocal and crtscts parameter names are taken from the POSIX termios serial interface definition. The clocal parameter controls whether the modem control lines (Data Terminal Ready, Carrier Detect/Received Line Signal Detect) are used (clocal=N) or ignored (clocal=Y). The crtscts parameter controls whether the hardware handshaking lines (Request To Send, Clear To Send) are used (crtscts=Y) or ignored (crtscts=N). The default parameter values (clocal=Y, crtscts=N) are those of a 'data−leads−only' serial interface.

The vxWorks serial driver does not provide independent control of the hardware handshaking and modem control lines, thus clocal=Y implies crtscts=N and clocal=N implies crtscts=Y.

vxWorks IOC serial ports may need to be set up using hardware−specific commands. Once this is done, the standard drvAsynSerialPortConfigure and asynSetOption commands can be issued. For example, the following example shows the configuration procedure for a port on a GreenSprings octal UART Industry−Pack module on a GreenSprings VIP616−01 carrier.

```
ipacAddVIPC616_01("0x6000,B0000000")
tyGSOctalDrv(1)
tyGSOctalModuleInit("RS232", 0x80, 0, 0)
tyGSOctalDevCreate("/tyGS/0/0",0,0,1000,1000)
drvAsynSerialPortConfigure("L0","/tyGS/0/0",0,0,0)
asynSetOption("L0",0,"baud","9600")
```

TCP/IP or UDP/IP Port

The drvAsynIPPort driver supports devices which communicate over a TCP/IP or UDP/IP connection. A typical example is a device connected through an Ethernet/Serial converter box.

TCP/IP or UDP/IP connections are configured with the `drvAsynIPPortConfigure` command:

```
drvAsynIPPortConfigure("portName","hostInfo",priority,noAutoConnect,
     processEosIn,processEosOut)
```

where the arguments are:

- portName − The portName that is registered with asynGpib.
- hostInfo − The Internet host name, port number and optional IP protocol of the device (e.g. "164.54.9.90:4002", "serials8n3:4002", "serials8n3:4002 TCP" or "164.54.17.43:5186 udp"). If no protocol is specified, TCP will be used.
- priority − Priority at which the asyn I/O thread will run. If this is zero or missing, then epicsThreadPriorityMedium is used.
- noAutoConnect − Zero or missing indicates that portThread should automatically connect. Non−zero if explicit connect command must be issued.
- processEosIn, processEosOut − If either is set asynOctetBase will call asynInterposeEosConfig.

The setEos and getEos methods have no effect and return asynError. The read method blocks until at least one character has been received or until a timeout occurs. The read method transfers as many characters as possible, limited by the specified count.

There are no asynSetOption key/value pairs associated with drvAsynIPPort connections.

asynInterposeEos and asynInterposeFlush can be used to provide additional functionality.

VXI−11

VXI−11 is a TCP/IP protocol for communicating with IEEE 488.2 devices. It is an RPC based protocol. In addition to the VXI−11 standard, three additional standards are defined.

- VXI−11.1 − A standard for communicating with VXIbus devices. These devices have a vxiName that starts with "vxi".
- VXI−11.2 − A standard for communicating with a IEEE 488.1 device. This means that the TCP/IP connection is talking to a GPIB controller that is talking to a GPIB bus. These devices have an vxiName that starts with "gpib".
- VXI−11.3 − A standard for communicating with IEEE 488.2 devices. This means that the TCP/IP connection is talking directly with an device. These devices have an vxiName that starts with "inst".

Consult the following documents (available on−line) for details.

```
VMEbus Extensions for Instrumentation
    VXI−11   TCP/IP Instrument Protocol Specification
    VXI−11.1 TCP/IP−VXIbus Interface Specification
    VXI−11.2 TCP/IP−IEEE 488.1 Interface Specification
    VXI−11.3 TCP/IP−IEEE 488.2 Instrument Interface Specification
```

The following commands may be specified in the st.cmd file

```
    E2050Reboot("inet_addr")
    E5810Reboot("inet_addr","password")
    vxi11Configure("portName","inet_addr",recoverWithIFC,timeout,
        "vxiName",priority,noAutoConnect)
```

where

- inet_addr − Internet Address
- password − password. If given as 0 the default E5810 is used.

- portName − The portName that is registered with asynGib.
- inet_addr − Internet address.
- recoverWithIFC − (0,1) => (don't, do) issue IFC when error occurs.
- timeout − I/O operation timeout in seconds as datatype double. If 0.0, then a default is assigned.
- vxiName − Must be chosen as specified above.
- priority − Priority at which the asyn I/O thread will run. If this is zero or missing, then epicsThreadPriorityMedium is used.
- noAutoConnect − Zero or missing indicates that portThread should automatically connect. Non−zero if explicit connect command must be issued.

The vxi11 driver implements two timeouts: ioTimeout and rpcTimeout (Remote Procedure Call timeout). The ioTimeout is taken from asynUser:timeout. The rpcTimeout is handled internally for each port. It has a default of 4 seconds but can be changed by calling setOptions. For example:

```
asynSetOption L0 −1 rpctimeout .1
```

Will change the rpcTimeout for port L0 to .1 seconds.

Linux−Gpib

The linux−gpib port driver was written to support The Linux GPIB Package library.

Configuration command is:

```
GpibBoardDriverConfig(portName,autoConnect,BoardIndex,timeout,priority)
```

where

- portName − An ascii string specifying the port name that will be registered with asynDriver.
- noAutoConnect − Zero indicates that portThread should automatically connect. Non−zero means explicit connect command must be issued.
- boardIndex −Integer containing index of board (0 means /dev/gpib0). Normaly it is 0. This must be the same as in gpib.conf file (minor number − board index) of driver configuration.
- timeout − Time in seconds in which an i/o operation must complete. Zero means disabled. This is "general" timeout for every call to low level drivers. For actual read/write operations timeout must be defined in device support. Both timeouts are converted into integers 0−17 which represents disabled to 1000 seconds.
- priority − An integer specifying the priority of the port thread. A value of 0 will result in a default value being assigned.

An example is:

```
GpibBoardDriverConfig("L0",1,0,3,0)
```

NOTES:

- AsynOption Interface is supported. Key (hexadecimal) and val (integer) arguments to setPortOptions function must be appropriate values represented as charachter arrays as stated in GPIB library documentation.
- pgpibCmd type GPIBREADW nad GPIBEFASTIW were not tested.
- The linux−port driver was tested with PC104−GPIB board from Measurement Computing.

Green Springs IP488

This is support for the Green Springs Industry Pack GPIB carrier. The configuration command is:

```
gsIP488Configure(portName,carrier,module,intVec,priority,noAutoConnect)
```

where

- portName − An ascii string specifying the port name that will be registered with asynDriver.
- carrier − An integer identifying the Industry Pack Carrier
- module − An integer identifying the module on the carrier
- intVec − An integer specifying the interrupt vector
- priority − An integer specifying the priority of the portThread. A value of 0 will result in a defalt value being assigned
- noAutoConnect − Zero or missing indicates that portThread should automatically connect. Non−zero if explicit connect command must be issued.

An example is:

```
#The following is for the Greensprings IP488 on an MV162
ipacAddMVME162("A:l=3,3 m=0xe0000000,64")
gsIP488Configure("L0",0,0,0x61,0,0)
```

National Instruments GPIB−1014D

This is support for a National Instruments VME GPIB interface. The configuration command is:

```
ni1014Config(portNameA,portNameB,base,vector,level,priority,noAutoConnect)
```

where

- portNameA − An ascii string specifying the port name that will be registered with asynDriver for portA.
- portNameB − An ascii string specifying the port name that will be registered with asynDriver for portB. If only one port should be registered, then leave this as a null string. The support should also work for a single port NI1014 but has not been tested.
- base − VME A16 base address.
- vector − VME interrupt vector.
- level − An integer specifying the interrupt level.
- priority − In integer specifying the priority of the portThread. A value of 0 will result in a defalt value being assigned
- noAutoConnect − Zero or missing indicates that portThread should automatically connect. Non−zero if explicit connect command must be issued.

An example is:

```
ni1014Config("L0","L1",0x5000,0x64,5,0,0)
```

NOTES:

- Ports A and B are almost but not quite the same. Thus the code for connecting to port A is slightly different than the code for portB.

- In order to disconnect and reconnect either port, BOTH ports must be disconnected and reconnected.
- When the ports are connected, portA MUST be connected before port B.
- Programmed I/O, via interrupts, rather than DMA is implemented. Thus no A24 address space is required.

Diagnostic Aids

iocsh Commands

```
asynReport(filename,level,portName)
asynInterposeFlushConfig(portName,addr,timeout)
asynInterposeEosConfig(portName,addr)
asynSetTraceMask(portName,addr,mask)
asynSetTraceIOMask(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)
```

`asynReport` calls `asynCommon:report` for a specific port if portName is specified, or for all registered drivers and interposeInterface if portName is not specified.

`asynInterposeFlushConfig` is a generic interposeInterface that implements flush for low level drivers that don't implement flush. It just issues read requests until no bytes are left to read. The timeout is used for the read requests.

`asynInterposeEosConfig` is a generic interposeInterface that implements End of String processing for low level drivers that don't.

`asynSetTraceMask` calls `asynTrace:setTraceMask` for the specified port and address.

`asynSetTraceIOMask` calls `asynTrace:setTraceIOMask` for the specified port and address.

`asynSetTraceFile` calls `asynTrace:setTraceFile`. The filename is handled as follows:

- Not specified − A NULL pointer is passed to setTraceFile. Subsequent messages are sent to errlog.
- An empty string ("") or "stdout" − stdout is passed to setTraceFile.
- Any other string − The specified file is opened with an option of "w" and the file pointer is passed to setTraceFile.

`asynSetTraceIOTruncateSize` calls `asynTrace:setTraceIOTruncateSize`

`asynSetOption` calls `asynCommon:setOption`. `asynShowOption` calls `asynCommon:getOption`.

asynOctetConnect, ...asynOctetFlush provide shell access to asynOctetSyncIO methods. The entry is a character string constant that identifys the port,addr.

where

- filename − An ascii string naming a file. If null or a null string, then the output is sent to `stdout.`
- level − The report level.
- portName − An ascii string specifying the portName of the driver.
- addr − In integer specifying the address of the device. For multiDevice ports a value of −1 means the port itself. For ports that support a single device, addr is ignored.
- mask − The mask value to set. See the mask bit definitions in asynDriver.h
- key − The key for the option desired.
- val − The value for the option.
- yesNo − The value (0,1) means (no,yes).
- entry − A character string that identifies the asynOctetConnect request.
- oeos,ieos − The output and input End of String terminator.
- timeout − timeout as an integer in milliseconds. The default is 1.
- buffer_len − length of buffer for I/O. Default=80.
- nread − max number of bytes to read. Default=buffer_len.
- flush − (0,1) means (don't, do) flush before reading. Default=0.
- output − output string.

The commands asynOctetConnect, asynOctetRead, asynOctetWrite, asynOctetWriteRead, asynOctetFlush allow I/O to a device from the ioc shell. Examples are:

```
asynOctetConnect("myid","A",0,"\n","\n",1,20)
asynOctetWrite("myid","testnew")
asynOctetRead("myid")
testnew\n
asynOctetWriteRead("myid","this is test")
this is test\n
```

Install and Build

Install and Build asynDriver

After obtaining a copy of the distribution, it must be installed and built for use at your site. These steps only need to be performed once for the site (unless versions of the module running under different releases of EPICS and/or the other required modules are needed).

1. Create an installation directory for the module, usually this will end with

   ```
   .../support/asyn/
   ```

2. Place the distribution file in this directory. Then issue the commands (Unix style)

   ```
   gunzip <file>.tar.gz
   tar xvf <file>.tar
   ```

3. This creates a support <top>.

   ```
   .../support/asyn/X-Y
   ```

where X−Y is the release number. For example:

```
.../support/asyn/3-1
```

4. Edit the `config/RELEASE` file and set the paths to your installation of EPICS_BASE and IPAC. IPAC is only needed if you are building for vxWorks.
5. Run `make` in the top level directory and check for any compilation errors.

Using asynDriver Components with an EPICS iocCore Application

Since asynDriver does NOT provide support for specific devices an application must obtain device specific support elsewhere. This section only explains how to include asynDriver components.

In the `configure/RELEASE` file add definitions for `IPAC`, `ASYN`, and `EPICS_BASE`.

In the src directory where the application is built:

- Add the following to `Makefile`

```
<app>_LIBS += asyn
```

- Add the following to `<app>Include.dbd` and uncomment the line or lines appropriate for your application:

```
include "asyn.dbd"
#include "drvAsynSerialPort.dbd"
#include "drvAsynIPPort.dbd"
#include "drvVxi11.dbd"
#include "drvGsIP488.dbd"
#include "drvIpac.dbd"
#registrar(mv162ipRegistrar)
```

In the `st.cmd` file add:

dbLoadRecords("db/asynRecord.db", "P=<ioc>, R=<record>, PORT=<port>, ADDR=<addr>, OMAX=<omax>, IMAX=<imax>")

You must provide values for <ioc>, <record>, <port>, <addr>, <omax>, and <imax>.

Once the application is running, medm displays for an ioc can be started by: medm −x −macro "P=<ioc>,R=<record>" <asyntop>/medm/asynRecord.adl &

You must provide correct values for <ioc> and <record>. Once asynRecord is started, it can be connected to different devices.

License Agreement

1. The "Software", below, refers to asynDriver (in either source code, or binary form and accompanying documentation). Each licensee is addressed as "you" or "Licensee."

2. The copyright holders shown above and their third−party licensors hereby grant Licensee a royalty−free nonexclusive license, subject to the limitations stated herein and U.S. Government license rights.

3. You may modify and make a copy or copies of the Software for use within your organization, if you meet the following conditions:
   a. Copies in source code must include the copyright notice and this Software License Agreement.
   b. Copies in binary form must include the copyright notice and this Software License Agreement in the documentation and/or other materials provided with the copy.

4. You may modify a copy or copies of the Software or any portion of it, thus forming a work based on the Software, and distribute copies of such work outside your organization, if you meet all of the following conditions:
   a. Copies in source code must include the copyright notice and this Software License Agreement;
   b. Copies in binary form must include the copyright notice and this Software License Agreement in the documentation and/or other materials provided with the copy;
   c. Modified copies and works based on the Software must carry prominent notices stating that you changed specified portions of the Software.

5. Portions of the Software resulted from work developed under a U.S. Government contract and are subject to the following license: the Government is granted for itself and others acting on its behalf a paid−up, nonexclusive, irrevocable worldwide license in this computer software to reproduce, prepare derivative works, and perform publicly and display publicly.

6. WARRANTY DISCLAIMER. THE SOFTWARE IS SUPPLIED "AS IS" WITHOUT WARRANTY OF ANY KIND. THE COPYRIGHT HOLDERS, THEIR THIRD PARTY LICENSORS, THE UNITED STATES, THE UNITED STATES DEPARTMENT OF ENERGY, AND THEIR EMPLOYEES: (1) DISCLAIM ANY WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE OR NON−INFRINGEMENT, (2) DO NOT ASSUME ANY LEGAL LIABILITY OR RESPONSIBILITY FOR THE ACCURACY, COMPLETENESS, OR USEFULNESS OF THE SOFTWARE, (3) DO NOT REPRESENT THAT USE OF THE SOFTWARE WOULD NOT INFRINGE PRIVATELY OWNED RIGHTS, (4) DO NOT WARRANT THAT THE SOFTWARE WILL FUNCTION UNINTERRUPTED, THAT IT IS ERROR−FREE OR THAT ANY ERRORS WILL BE CORRECTED.

7. LIMITATION OF LIABILITY. IN NO EVENT WILL THE COPYRIGHT HOLDERS, THEIR THIRD PARTY LICENSORS, THE UNITED STATES, THE UNITED STATES DEPARTMENT OF ENERGY, OR THEIR EMPLOYEES: BE LIABLE FOR ANY INDIRECT, INCIDENTAL, CONSEQUENTIAL, SPECIAL OR PUNITIVE DAMAGES OF ANY KIND OR NATURE, INCLUDING BUT NOT LIMITED TO LOSS OF PROFITS OR LOSS OF DATA, FOR ANY REASON WHATSOEVER, WHETHER SUCH LIABILITY IS ASSERTED ON THE BASIS OF CONTRACT, TORT (INCLUDING NEGLIGENCE OR STRICT LIABILITY), OR OTHERWISE, EVEN IF ANY OF SAID PARTIES HAS BEEN WARNED OF THE POSSIBILITY OF SUCH LOSS OR DAMAGES.