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

W. Eric Norum
norume@aps.anl.gov

26th July 2004

## 1 Introduction

This tutorial provides step-by-step instructions on how to create EPICS support for a simple serial or GPIB (IEEE-488) device. The steps are presented in a way that should make it possible to apply them in cookbook fashion to create support for other devices. For comprehensive description of all the details of the I/O system used here, refer to the asynDriver and devGpib documentation.

This document isn't for the absolute newcomer though. You must have EPICS installed on a system somewhere and know how to build and run the example application. In particular you must have the following installed:

- EPICS R3.14.6 or higher.
- EPICS modules/soft/asyn version 3.2 or higher.

Serial and GPIB devices can now be treated in much the same way. The EPICS 'asyn' driver devGpib module can use the low-level drivers which communicate with serial devices connected to ports on the IOC or to Ethernet/Serial converters or with GPIB devices connected to local I/O cards or to Ethernet/GPIB converters.

I based this tutorial on the device support I wrote for a CVI Laser Corporation AB300 filter wheel. You're almost certainly interested in controlling some other device so you won't be able to use the information directly. I chose the AB300 as the basis for this tutorial since the AB300 has a very limited command set, which keeps this document small, and yet has commands which raise many of the issues that you'll have to consider when writing support for other devices.

## 2 Determine the required I/O operations

The first order of business is to determine the set of operations the device will have to perform. A look at the AB300 documentation reveals that there are four commands that must be supported. Each command will be associated with an EPICS process variable (PV) whose type must be appropriate to the data transferred by the command. The AB300 commands and process variable record types I choose to associate with them are shown in table 1.

There are lots of other ways that the AB300 could be handled. It might be useful, for example, to treat the filter position as multi-bit binary records instead.

## 3 Create a new device support module

Now that the device operations and EPICS process variable types have been chosen it's time to create a new EPICS application to provide a place to perform subsequent software development. The easiest way to do this is with the makeSupport.pl script supplied with the EPICS ASYN package.

Table 1: AB300 filter wheel commands

| CVI Laser Corporation AB300 filter wheel | |
|---|---|
| Command | EPICS record type |
| Reset | longout |
| Go to new position | longout |
| Query position | longin |
| Query status | longin |

Here are the commands I ran. You'll have to change the `/home/EPICS/modules/soft/asyn` to the path where your EPICS ASYN driver is installed.

```
norume> mkdir ab300
norume> cd ab300
norume> /home/EPICS/modules/soft/asyn/bin/linux-x86/makeSupport.pl -t devGpib AB300
```

## 3.1 Make some changes to the files in configure/

Edit the `configure/RELEASE` file which makeSupport.pl created and confirm that the entries describing the paths to your EPICS base and ASYN support are correct. For example these might be:

```
ASYN=/home/EPICS/modules/soft/asyn
EPICS_BASE=/home/EPICS/base
```

Edit the `configure/CONFIG` file which makeSupport.pl created and specify the IOC architectures on which the application is to run. I wanted the application to run as a soft IOC, so I uncommented the `CROSS_COMPILER_TARGET_ARCHS` definition and set the definition to be empty:

```
CROSS_COMPILER_TARGET_ARCHS =
```

## 3.2 Create the device support file

The contents of the device support file provide all the details of the communication between the device and EPICS. The makeSupport.pl command created a skeleton device support file in `AB300Sup/devAB300.c`. Of course, device support for a device similar to the one you're working with provides an even easier starting point.

The remainder this section describes the changes that I made to the skeleton file in order to support the AB300 filter wheel. You'll have to modify the steps as appropriate for your device.

### 3.2.1 Declare the DSET tables provided by the device support

Since the AB300 provides only longin and longout records most of the `DSET_`*xxx* define statements can be removed. Because of the way that the device initialization is performed you must define an analog-in DSET even if the device provides no analog-in records (as is the case for the AB300).

```
#define DSET_AI    devAiAB300
#define DSET_LI    devLiAB300
#define DSET_LO    devLoAB300
```

### 3.2.2 Select timeout values

The default value of `TIMEWINDOW` (2 seconds) is reasonable for the AB300, but I increased the value of `TIMEOUT` to 5 seconds since the filter wheel can be slow in responding.

```
#define TIMEOUT     5.0    /* I/O must complete within this time */
#define TIMEWINDOW  2.0    /* Wait this long after device timeout */
```

### 3.2.3 Clean up some unused values

The skeleton file provides a number of example character string arrays. None are needed for the AB300 so I just removed them. Not much space would be wasted by just leaving them in place however.

### 3.2.4 Declare the command array

This is the hardest part of the job. Here's where you have to figure how to produce the command strings required to control the device and how to convert the device responses into EPICS process variable values.

Each command array entry describes the details of a single I/O operation type. The application database uses the index of the entry in the command array to provide the link between the process variable and the I/O operation to read or write that value.

The command array entries I created for the AB300 are shown below. The elements of each entry are described using the names from the GPIB documentation.

**Command array index 0 – Device Reset**

```
{&DSET_LO, GPIBWRITE, IB_Q_LOW, NULL, "\377\377\033", 10, 10,
       NULL, 0, 0, NULL, NULL, "\033"},
```

**dset** This command is associated with an longout record.

**type** A WRITE operation is to be performed.

**pri** This operation will be placed on the low-priority queue of I/O requests.

**cmd** Because this is a GPIBWRITE operation this element is unused.

**format** The format string to generate the command to be sent to the device. The first two bytes are the RESET command, the third byte is the ECHO command. The AB300 sends no response to a reset command so I send the 'ECHO' to verify that the device is responding. The AB300 resets itself fast enough that it can see an echo command immediately following the reset command.

Note that the process variable value is not used (there's no printf % format character in the command string). The AB300 is reset whenever the EPICS record is processed.

**rspLen** The size of the readback buffer. Although only one readback byte is expected I allow for a few extra bytes just in case.

**msgLen** The size of the buffer into which the command string is placed. I allowed a little extra space in case a longer command is used some day.

**convert** No special conversion function is needed.

**P1,P2,P3** There's no special conversion function so no arguments are needed.

**pdevGpibNames** There's no name table.

**eos** The end-of-string value used to mark the end of the readback operation. GPIB devices can usually leave this entry NULL since they use the End-Or-Identify (EOI) line to delimit messages.

**Command array index 1 – Go to new filter position**

```
{&DSET_LO, GPIBWRITE, IB_Q_LOW, NULL, "\017%c", 10, 10,
        NULL, 0, 0, NULL, NULL, "\030"},
```

**dset** This command is associated with an longout record.

**type** A WRITE operation is to be performed.

**pri** This operation will be placed on the low-priority queue of I/O requests.

**cmd** Because this is a GPIBWRITE operation this element is unused.

**format** The format string to generate the command to be sent to the device. The filter position (1-6) can be converted to the required command byte with the printf `%c` format.

**rspLen** The size of the readback buffer. Although only two readback bytes are expected I allow for a few extra bytes just in case.

**msgLen** The size of the buffer into which the command string is placed. I allowed a little extra space in case a longer command is used some day.

**convert** No special conversion function is needed.

**P1,P2,P3** There's no special conversion function so no arguments are needed.

**pdevGpibNames** There's no name table.

**eos** The end-of-string value used to mark the end of the readback operation.

**Command array index 2 – Query filter position**

```
{&DSET_LI, GPIBREAD, IB_Q_LOW, "\035", NULL, 0, 10,
        convertPositionReply, 0, 0, NULL, NULL, "\030"},
```

**dset** This command is associated with an longin record.

**type** A READ operation is to be performed.

**pri** This operation will be placed on the low-priority queue of I/O requests.

**cmd** The command string to be sent to the device. The AB300 responds to this command by sending back three bytes: the current position, the controller status, and a terminating `'\030'`.

**format** Because this operation has its own conversion function this element is unused.

**rspLen** There is no command echo to be read.

**msgLen** The size of the buffer into which the reply string is placed. Although only three reply bytes are expected I allow for a few extra bytes just in case.

**convert** There's no sscanf format that can convert the reply from the AB300 so a special conversion function must be provided.

**P1,P2,P3** The special conversion function requires no arguments.

**pdevGpibNames** There's no name table.

**eos** The end-of-string value used to mark the end of the read operation.

**Command array index 3 – Query controller status** This command array entry is almost identical to the previous entry. The only change is that a different custom conversion function is used.

```
{&DSET_LI, GPIBREAD, IB_Q_LOW, "\035", NULL, 0, 10,
        convertStatusReply, 0, 0, NULL, NULL, "\030"},
```

### 3.2.5 Write the special conversion functions

As mentioned above, special conversion functions are need to convert reply messages from the AB300 into EPICS PV values. The easiest place to put these functions is just before the `gpibCmds` table. The conversion functions are passed a pointer to the `gpibDpvt` structure and three values from the command table entry. The `gpibDpvt` structure contains a pointer to the EPICS record. The custom conversion function uses this pointer to set the record's value field.

Here are the custom conversion functions I wrote for the AB300.

```
/*
 * Custom conversion routines
 */
static int
convertPositionReply(struct gpibDpvt *pdpvt, int P1, int P2, char **P3)
{
    struct longinRecord *pli = ((struct longinRecord *)(pdpvt->precord));

    if (pdpvt->msgInputLen != 3) {
        epicsSnprintf(pdpvt->pasynUser->errorMessage,
                      pdpvt->pasynUser->errorMessageSize,
                      "Invalid reply");
        return -1;
    }
    pli->val = pdpvt->msg[0];
    return 0;
}
static int
convertStatusReply(struct gpibDpvt *pdpvt, int P1, int P2, char **P3)
{
    struct longinRecord *pli = ((struct longinRecord *)(pdpvt->precord));

    if (pdpvt->msgInputLen != 3) {
        epicsSnprintf(pdpvt->pasynUser->errorMessage,
                      pdpvt->pasynUser->errorMessageSize,
                      "Invalid reply");
        return -1;
    }
    pli->val = pdpvt->msg[1];
    return 0;
}
```

Some points of interest:

1. Custom conversion functions indicate an error by returning -1.
2. If an error status is returned an explanation should be left in the `errorMessage` buffer.
3. I put in a sanity check to ensure that the end-of-string character is where it should be.

### 3.2.6 Provide the device support initialization

Because of way code is stored in object libraries on different systems the device support parameter table must be initialized at run-time. The analog-in initializer is used to perform this operation. This is why all device support files must declare an analog-in DSET.

Here's the initialization for the AB300 device support. The AB300 immediately echos the command characters sent to it so the respond2Writes value must be set to 0. All the other values are left as created by the makeSupport.pl script:

```
static long init_ai(int parm)
{
    if(parm==0) {
        devSupParms.name = "devAB300";
        devSupParms.gpibCmds = gpibCmds;
        devSupParms.numparams = NUMPARAMS;
        devSupParms.timeout = TIMEOUT;
        devSupParms.timeWindow = TIMEWINDOW;
        devSupParms.respond2Writes = 0;
    }
    return(0);
}
```

## 3.3 Modify the device support database definition file

This file specifies the link between the DSET names defined in the device support file and the DTYP fields in the application database. The makeSupport.pl command created an example file in `AB300Sup/devAB300.dbd`. If you removed any of the `DSET_`*xxx* definitions from the device support file you must remove the corresponding lines from this file.

```
device(ai,        GPIB_IO, devAiAB300,    "AB300")
device(longin,    GPIB_IO, devLiAB300,    "AB300")
device(longout,   GPIB_IO, devLoAB300,    "AB300")

include "asyn.dbd"
```

## 3.4 Create the device support database file

This is the database describing the actual EPICS process variables associated with the filter wheel.

I modified the file `AB300Sup/devAB300.db` to have the following contents:

```
record(longout, "$(P)$(R)FilterWheel:reset")
{
    field(DESC, "Reset AB300 Controller")
    field(SCAN, "Passive")
    field(DTYP, "AB300")
    field(OUT,  "#L$(L) A$(A) @0")
}
record(longout, "$(P)$(R)FilterWheel")
{
    field(DESC, "Set Filter Wheel Position")
    field(SCAN, "Passive")
    field(DTYP, "AB300")
```

```
    field(OUT,  "#L$(L) A$(A) @1")
    field(LOPR, 1)
    field(HOPR, 6)
}
record(longin, "$(P)$(R)FilterWheel:fbk")
{
    field(DESC, "Filter Wheel Position")
    field(SCAN, "Passive")
    field(DTYP, "AB300")
    field(INP,  "#L$(L) A$(A) @2")
    field(LOPR, 1)
    field(HOPR, 6)
}
record(longin, "$(P)$(R)FilterWheel:status")
{
    field(DESC, "Filter Wheel Status")
    field(SCAN, "Passive")
    field(DTYP, "AB300")
    field(INP,  "#L$(L) A$(A) @3")
}
```

Notes:

1. The numbers following the L in the INP and OUT fields are the number of the 'link' used to communicate with the filter wheel. This link is set up at run time by commands in the application startup script.
2. The numbers following the A in the INP and OUT fields are unused by serial devices but must be a valid GPIB address (0-30) since the GPIB address conversion routines check the value and the diagnostic display routines require a matching value.
3. The numbers following the @ in the INP and OUT fields are the indices into the GPIB command array.
4. The DTYP fields must match the names specified in the devAB300.dbd database definition.
5. The device support database follows the ASYN convention that the macros $(P), $(R), $(L) and $(A) are used to specify the record name prefixes, link number and GPIB address, respectively.

### 3.5 Build the device support

Change directories to the top-level directory of your device support and:

```
norume> make
```

(**gnumake** on Solaris).
If all goes well you'll be left with a device support library in lib/<*EPICS_HOST_ARCH*>/, a device support database definition in dbd/ and a device support database in db/.

## 4 Create a test application

Now that the device support has been completed it's time to create a new EPICS application to confirm that the device support is operating correctly. The easiest way to do this is with the makeBaseApp.pl script supplied with EPICS.

Here are the commands I ran. You'll have to change the `/home/EPICS/base` to the path to where your EPICS base is installed. If you're not running on Linux you'll also have to change all the `linux-x86` to reflect the architecture you're using (`solaris-sparc`, `darwin-ppc`, etc.). I built the test application in the same <top> as the device support, but the application could be built anywhere. As well, I built the application as a 'soft' IOC running on the host machine, but the serial/GPIB driver also works on RTEMS and vxWorks.

```
norume> cd ab300
norume> /home/EPICS/base/bin/linux-x86/makeBaseApp.pl -t ioc AB300
norume> /home/EPICS/base/bin/linux-x86/makeBaseApp.pl -i -t ioc AB300
The following target architectures are available in base:
    RTEMS-pc386
    linux-x86
    solaris-sparc
    win32-x86-cygwin
    vxWorks-ppc603
What architecture do you want to use? linux-x86
```

# 5 Using the device support in an application

Several files need minor modifications to use the device support in the test, or any other, application.

## 5.1 Make some changes to configure/RELEASE

Edit the `configure/RELEASE` file which makeBaseApp.pl created and confirm that the EPICS_BASE path is correct. Add entries for your ASYN and device support. For example these might be:

```
AB300=/home/EPICS/modules/instrument/ab300/1-2
ASYN=/home/EPICS/modules/soft/asyn/3-2
EPICS_BASE=/home/EPICS/base
```

## 5.2 Modify the application database definition file

Your application database definition file must include the database definition files for your instrument and for the ASYN drivers. There are two ways that this can be done:

1. If you are building your application database definition from an *xxx*`Include.dbd` file you include the additional database definitions in that file. For example, to add support for the AB300 instrument and local and remote serial line drivers:

```
include "base.dbd"

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

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

*xxx*_DBD += base.dbd
*xxx*_DBD += devAB300.dbd
*xxx*_DBD += drvAsynIPPort.dbd
*xxx*_DBD += drvAsynSerialPort.dbd
.
.

## 5.3 Add the device support libraries to the application

You must link your device support library and the ASYN support library with the application. Add the following lines

```
xxx_LIBS += devAB300
xxx_LIBS += asyn
```

before the

```
xxx_LIBS += $(EPICS_BASE_IOC_LIBS)
```

line in the application `Makefile`.

## 5.4 Modify the application startup script

The `st.cmd` application startup script created by the makeBaseApp.pl script needs a few changes to get the application working properly.

1. Load the device support database records:

   ```
   cd $(AB300)            (cd AB300 if using the vxWorks shell)
   dbLoadRecords("db/devAB300.db","P=AB300:,R=,L=0,A=0")
   ```

2. Set up the 'port' between the IOC and the filter wheel.

   - If you're using an Ethernet/RS-232 converter or a device which communicates over a telnet-style socket connection you need to specify the Internet host and port number like:

     ```
     drvAsynIPPortConfigure("L0","164.54.9.91:4002",0,0,0)
     ```

   - If you're using a serial line directly attached to the IOC you need something like:

     ```
     drvAsynSerialPortConfigure("L0","/dev/ttyS0",0,0,0)
     asynSetOption("L0", -1, "baud", "9600")
     asynSetOption("L0", -1, "bits", "8")
     asynSetOption("L0", -1, "parity", "none")
     asynSetOption("L0", -1, "stop", "1")
     asynSetOption("L0", -1, "clocal", "Y")
     asynSetOption("L0", -1, "crtscts", "N")
     ```

   - If you're using a serial line directly attached to a vxWorks IOC you must first configure the serial port interface hardware. The following example shows the commands to configure a port on a GreenSprings UART Industry-Pack module.

     ```
     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",-1,"baud","9600")
     ```

In all of the above examples the first argument of the configure and set port option commands is the link identifier and must match the L value in the EPICS database record INP and OUT fields. The second argument of the configure command identifies the port to which the connection is to be made. The third argument sets the priority of the worker thread which performs the I/O operations. A value of zero directs the command to choose a reasonable default value. The fourth argument is zero to direct the device support layer to automatically connect to the serial port on startup and whenever the serial port becomes disconnected. The final argument is zero to direct the device support layer to use standard end-of-string processing on input messages.

3. (Optional) Add lines to control the debugging level of the serial/GPIB driver. The following enables error messages and a description of every I/O operation.

```
asynSetTraceMask("L0",-1,0x9)
asynSetTraceIOMask("L0",-1,0x2)
```

A better way to control the amount and type of diagnostic output is to add an asynRecord to your application.

## 5.5 Build the application

Change directories to the top-level directory of your application and:

```
norume> make
```

(**gnumake** on Solaris).
If all goes well you'll be left with an executable program in bin/linux-x86/AB300.

## 5.6 Run the application

Change directories to where makeBaseApp.pl put the application startup script and run the application:

```
norume> cd iocBoot/iocAB300
norume> ../../bin/linux-x86/AB300 st.cmd
dbLoadDatabase("../../dbd/AB300.dbd",0,0)
AB300_registerRecordDeviceDriver(pdbbase)
cd ${AB300}
dbLoadRecords("db/devAB300.db","P=AB300:,R=,L=0,A=0")
drvAsynIPPortConfigure("L0","164.54.3.137:4001",0,0,0)
asynSetTraceMask("L0",-1,0x9)
asynSetTraceIOMask("L0",-1,0x2)
iocInit()
############################################################################
###  EPICS IOC CORE built on Apr 23 2004
###  EPICS R3.14.6 $$Name:  $$ $$Date: 2004/06/23 13:50:20 $$
############################################################################
Starting iocInit
iocInit: All initialization complete
```

Check the process variable names:

```
epics> dbl
AB300:FilterWheel:fbk
AB300:FilterWheel:status
AB300:FilterWheel
AB300:FilterWheel:reset
```

Reset the filter wheel. The values sent between the IOC and the filter wheel are shown:

```
epics> dbpf AB300:FilterWheel:reset 0
DBR_LONG:           0         0x0
2004/04/21 12:05:14.386 164.54.3.137:4001 write 3 \377\377\033
2004/04/21 12:05:16.174 164.54.3.137:4001 read 1 \033
```

Read back the filter wheel position. The dbtr command prints the record before the I/O has a chance to occur:

```
epics> dbtr AB300:FilterWheel:fbk
ACKS: NO_ALARM      ACKT: YES           ADEL: 0             ALST: 0
ASG:                BKPT: 0x00          DESC: Filter Wheel Position
DISA: 0             DISP: 0             DISS: NO_ALARM      DISV: 1
DTYP: AB300Gpib     EGU:                EVNT: 0             FLNK:CONSTANT 0
HHSV: NO_ALARM      HIGH: 0             HIHI: 0             HOPR: 6
HSV: NO_ALARM       HYST: 0             INP:GPIB_IO #L0 A0 @2
LALM: 0             LCNT: 0             LLSV: NO_ALARM      LOLO: 0
LOPR: 1             LOW: 0              LSV: NO_ALARM       MDEL: 0
MLST: 0             NAME: AB300:FilterWheel:fbk             NSEV: NO_ALARM
NSTA: NO_ALARM      PACT: 1             PHAS: 0             PINI: NO
PRIO: LOW           PROC: 0             PUTF: 0             RPRO: 0
SCAN: Passive       SDIS:CONSTANT       SEVR: INVALID       SIML:CONSTANT
SIMM: NO            SIMS: NO_ALARM      SIOL:CONSTANT       STAT: UDF
SVAL: 0             TPRO: 0             TSE: 0              TSEL:CONSTANT
UDF: 1              VAL: 0
2004/04/21 12:08:01.971 164.54.3.137:4001 write 1 \035
2004/04/21 12:08:01.994 164.54.3.137:4001 read 3 \001\020\030
```

Now the process variable should have that value:

```
epics> dbpr AB300:FilterWheel:fbk
ASG:                DESC: Filter Wheel Position             DISA: 0
DISP: 0             DISV: 1             NAME: AB300:FilterWheel:fbk
SEVR: NO_ALARM      STAT: NO_ALARM      SVAL: 0             TPRO: 0
VAL: 1
```

Move the wheel to position 4:

```
epics> dbpf AB300:FilterWheel 4
DBR_LONG:           4         0x4
2004/04/21 12:10:51.542 164.54.3.137:4001 write 2 \017\004
2004/04/21 12:10:51.562 164.54.3.137:4001 read 1 \020
2004/04/21 12:10:52.902 164.54.3.137:4001 read 1 \030
```

Read back the position:

```
epics> dbtr AB300:FilterWheel:fbk
ACKS: NO_ALARM      ACKT: YES           ADEL: 0             ALST: 1
ASG:                BKPT: 0x00          DESC: Filter Wheel Position
DISA: 0             DISP: 0             DISS: NO_ALARM      DISV: 1
DTYP: AB300Gpib     EGU:                EVNT: 0             FLNK:CONSTANT 0
HHSV: NO_ALARM      HIGH: 0             HIHI: 0             HOPR: 6
```

```
    HSV: NO_ALARM       HYST: 0             INP:GPIB_IO #L0 A0 @2
    LALM: 1             LCNT: 0             LLSV: NO_ALARM      LOLO: 0
    LOPR: 1             LOW: 0              LSV: NO_ALARM       MDEL: 0
    MLST: 1             NAME: AB300:FilterWheel:fbk             NSEV: NO_ALARM
    NSTA: NO_ALARM      PACT: 1             PHAS: 0             PINI: NO
    PRIO: LOW           PROC: 0             PUTF: 0             RPRO: 0
    SCAN: Passive       SDIS:CONSTANT       SEVR: NO_ALARM      SIML:CONSTANT
    SIMM: NO            SIMS: NO_ALARM      SIOL:CONSTANT       STAT: NO_ALARM
    SVAL: 0             TPRO: 0             TSE: 0              TSEL:CONSTANT
    UDF: 0              VAL: 1
    2004/04/21 12:11:43.372 164.54.3.137:4001 write 1 \035
    2004/04/21 12:11:43.391 164.54.3.137:4001 read 3 \004\020\030
```

And it really is 4:

```
    epics> dbpr AB300:FilterWheel:fbk
    ASG:                DESC: Filter Wheel Position             DISA: 0
    DISP: 0             DISV: 1             NAME: AB300:FilterWheel:fbk
    SEVR: NO_ALARM      STAT: NO_ALARM      SVAL: 0             TPRO: 0
    VAL: 4
```

# 6 Device Support File

Here is the complete device support file for the AB300 filter wheel (`AB300Sup/devAB300.c`):

```
/*
 * AB300 device support
 */

#include <epicsStdio.h>
#include <devCommonGpib.h>

/*****************************************************************************
 *
 * The following define statements are used to declare the names to be used
 * for the dset tables.
 *
 * A DSET_AI entry must be declared here and referenced in an application
 * database description file even if the device provides no AI records.
 *
 *****************************************************************************/
#define DSET_AI     devAiAB300
#define DSET_LI     devLiAB300
#define DSET_LO     devLoAB300

#include <devGpib.h> /* must be included after DSET defines */

#define TIMEOUT     5.0    /* I/O must complete within this time */
#define TIMEWINDOW  2.0    /* Wait this long after device timeout */

/*
```

```
 * Custom conversion routines
 */
static int
convertPositionReply(struct gpibDpvt *pdpvt, int P1, int P2, char **P3)
{
    struct longinRecord *pli = ((struct longinRecord *)(pdpvt->precord));

    if (pdpvt->msgInputLen != 3) {
        epicsSnprintf(pdpvt->pasynUser->errorMessage,
                      pdpvt->pasynUser->errorMessageSize,
                      "Invalid reply");
        return -1;
    }
    pli->val = pdpvt->msg[0];
    return 0;
}
static int
convertStatusReply(struct gpibDpvt *pdpvt, int P1, int P2, char **P3)
{
    struct longinRecord *pli = ((struct longinRecord *)(pdpvt->precord));

    if (pdpvt->msgInputLen != 3) {
        epicsSnprintf(pdpvt->pasynUser->errorMessage,
                      pdpvt->pasynUser->errorMessageSize,
                      "Invalid reply");
        return -1;
    }
    pli->val = pdpvt->msg[1];
    return 0;
}

/*****************************************************************************
 *
 * Array of structures that define all GPIB messages
 * supported for this type of instrument.
 *
 *****************************************************************************/

static struct gpibCmd gpibCmds[] = {
    /* Param 0 -- Device Reset */
    {&DSET_LO, GPIBWRITE, IB_Q_LOW, NULL, "\377\377\033", 10, 10,
        NULL, 0, 0, NULL, NULL, "\033"},

    /* Param 1 -- Go to new filter position */
    {&DSET_LO, GPIBWRITE, IB_Q_LOW, NULL, "\017%c", 10, 10,
        NULL, 0, 0, NULL, NULL, "\030"},

    /* Param 2 -- Query filter position */
    {&DSET_LI, GPIBREAD, IB_Q_LOW, "\035", NULL, 0, 10,
        convertPositionReply, 0, 0, NULL, NULL, "\030"},
```

```
    /* Param 3 -- Query controller status */
    {&DSET_LI, GPIBREAD, IB_Q_LOW, "\035", NULL, 0, 10,
        convertStatusReply, 0, 0, NULL, NULL, "\030"}
};

/* The following is the number of elements in the command array above.  */
#define NUMPARAMS sizeof(gpibCmds)/sizeof(struct gpibCmd)

/*****************************************************************************
 *
 * Initialize device support parameters
 *
 *****************************************************************************/
static long init_ai(int parm)
{
    if(parm==0) {
        devSupParms.name = "devAB300";
        devSupParms.gpibCmds = gpibCmds;
        devSupParms.numparams = NUMPARAMS;
        devSupParms.timeout = TIMEOUT;
        devSupParms.timeWindow = TIMEWINDOW;
        devSupParms.respond2Writes = 0;
    }
    return(0);
}
```

# 7 asynRecord support

The asynRecord provides a convenient mechanism for controlling the diagnostic messages produced by asyn drivers. To use an asynRecord in your application:

1. Add the line

   ```
   DB_INSTALLS += $(ASYN)/db/asynRecord.db
   ```

   to an application `Makefile`.

2. Create the diagnostic record by adding a line like

   cd $(ASYN) (cd ASYN if using the vxWorks shell)
   `dbLoadRecords("db/asynRecord.db","P=AB300,R=Test,PORT=L0,ADDR=0,IMAX=0,OMAX=0")`

   to the application startup (`st.cmd`) script. The `PORT` value must match the the value in the corresponding `drvAsynIPPortConfigure` or `drvAsynSerialPortConfigure` command. The `addr` value should be that of the instrument whose I/O you wish to monitor. The `P` and `R` values are arbitrary and are concatenated together to form the record name. Choose values which are unique among all IOCs on your network.

To run the asynRecord screen, add *<asynTop>*`/medm` to your `EPICS_DISPLAY_PATH` environment variable and start medm with `P`, `R`, `PORT` and `ADDR` values matching those given in the `dbLoadRecords` command:

```
medm -x -macro "P=AB300,R=Test,PORT=L0,ADDR=0" asynRecord.adl
```