

How to create EPICS device support for a simple serial device

W. Eric Norum
<norume@aps.anl.gov>

May 5, 2003

Contents

1	Introduction	1
2	Determine the required I/O operations	1
3	Create a new application	1
4	Make some changes to the files in configure/	2
5	Create the device support file	2
5.1	Declare the DSET tables provided by the device support	2
5.2	Change the debugging flag name	2
5.3	Select timeout values	3
5.4	Clean up some unused values	3
5.5	Declare the GPIB command array	3
5.6	Write the special conversion functions	5
5.7	Provide the device support initialization	6
6	Add the device support to the application	6
7	Modify the application database definition file	7
8	Create the application database file	7
9	Add the database file to the application	8
10	Modify the application startup script	8
11	Build the application	9
12	Run the application	9
13	Device Support File	12

1 Introduction

This tutorial provides step-by-step instructions on how to create EPICS support for a simple serial device. The steps are presented in a way that should make it possible to apply them in cookbook fashion to create support for other serial devices. For comprehensive description of all the details of the I/O system used here, refer to the GPIB 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 R3.14.2 or higher
- An up-to-date version of `modules/bus/gpib/gpibCore`

Yes, `gpibCore`, since serial devices can now be treated in much the same way as GPIB (IEEE-488) devices. Current versions of the EPICS `gpibCore` module contain a special 'GPIB' driver capable of communicating with devices connected to serial ports on the IOC or with devices connected through Ethernet/Serial converter boxes.

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

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

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 application

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 `makeBaseApp.pl` script supplied with EPICS.

Here are the commands I ran. You'll have to change the `/home/EPICS/R3.14.2` to the path to where your EPICS 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 application as a 'soft' IOC running on the host machine, but the serial 'GPIB' driver also works on RTEMS and vxWorks.

```
norume> mkdir AB300
norume> cd AB300
norume> /home/EPICS/R3.14.2/base/bin/linux-x86/makeBaseApp.pl -t ioc AB300
norume> /home/EPICS/R3.14.2/base/bin/linux-x86/makeBaseApp.pl -i -t ioc AB300
```

The following target architectures are available in base:

```
RTEMS-pc386
linux-x86
solaris-sparc
vxWorks-ppc603
```

What architecture do you want to use? **linux-x86**

4 Make some changes to the files in configure/

Edit the `configure/RELEASE` file which `makeBaseApp.pl` created and add an entry describing the path to where you installed the EPICS GPIB core module:

```
GPIB=/home/EPICS/R3.14.2/modules/bus/gpib/gpibCore
```

Edit the `configure/CONFIG` file which `makeBaseApp.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 =
```

5 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 easiest way to create a device support file is to copy the skeleton device support file from the `gpibCore` module source directory to your application source directory:

```
norume> cd AB300App/src
norume> cp /usr/EPICS/R3.14.2/modules/bus/gpib/gpibCore/gpib/devCommon/devSkeletonGpib.c 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.

5.1 Declare the DSET tables provided by the device support

Since the AB300 provides only login and logout 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      devAB300_ai
#define DSET_LI      devAB300_li
#define DSET_LO      devAB300_lo
```

5.2 Change the debugging flag name

It's useful, but not necessary, to change the name of the debugging flag from `SkeletonDebug` to something more appropriate:

```
static int devAB300Debug = 0;
```

5.3 Select timeout values

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

```
#define TIME_WINDOW 2000      /* wait 2 seconds after device timeout */
#define IO_TIME      5000     /* I/O must complete within 5 seconds */
```

5.4 Clean up some unused values

The skeleton file provides a number of 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.

5.5 Declare the GPIB 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 (f1-f13) from the GPIB documentation.

5.5.1 Command array index 0 – Device Reset

```
{&DSET_LO, GPIBWRITE|GPIBEOS, IB_Q_HIGH, NULL, "\377\377\033", 10, 10,
  NULL, 0, 0, NULL, NULL, '\033'},
```

f1 This command is associated with an longout record.

f2 A WRITE operation is to be performed and that the readback from the device will end when the end-of-string character is received.

f3 This operation should be placed on the high-priority queue of I/O requests.

f4 Because this is a GPIBWRITE operation this element is unused.

f5 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 sprintf % format character in the command string). The AB300 is reset whenever the EPICS record is processed.

f6 The size of the readback buffer. Although only one readback byte is expected I allow for a few extra bytes just in case.

f7 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.

f8 No special conversion function is needed.

f9-f11 There's no special conversion function so no arguments are needed.

f12 There's no name table.

f13 The end-of-string value used to mark the end of the readback operation.

5.5.2 Command array index 1 – Go to new filter position

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

f1 This command is associated with an longout record.

f2 A WRITE operation is to be performed and that the readback from the device will end when the end-of-string character is received.

f3 This operation should be placed on the high-priority queue of I/O requests.

f4 Because this is a GPIBWRITE operation this element is unused.

f5 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.

f6 The size of the readback buffer. Although only two readback bytes are expected I allow for a few extra bytes just in case.

f7 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.

f8 No special conversion function is needed.

f9-f11 There's no special conversion function so no arguments are needed.

f12 There's no name table.

f13 The end-of-string value used to mark the end of the readback operation.

5.5.3 Command array index 2 – Query filter position

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

f1 This command is associated with an longin record.

f2 A READ operation is to be performed and that the read operation will end when the end-of-string character is received.

f3 This operation should be placed on the high-priority queue of I/O requests.

f4 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'`.

f5 Because this is a GPIBREAD operation this element is unused.

f6 There is no command echo to be read.

f7 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.

f8 There's no sscanf format that can convert the reply from the AB300 so a special conversion function must be provided.

f9-f11 The special conversion function requires no arguments.

f12 There's no name table.

f13 The end-of-string value used to mark the end of the read operation.

5.5.4 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|GPIBEOS, IB_Q_LOW, "\035", NULL, 0, 10,
  convertStatusReply, 0, 0, NULL, NULL, '\030'}
```

5.6 Write the special conversion functions

As mentioned above, special conversion functions are needed 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->msg[2] == '\030')
        pli->val = pdpvt->msg[0];
    else
        recGblSetSevr(pli, READ_ALARM, INVALID_ALARM);
    return 0;
}
static int
convertStatusReply(struct gpibDpvt *pdpvt, int P1, int P2, char **P3)
{
    struct longinRecord *pli = ((struct longinRecord *) (pdpvt->precord));

    if( pdpvt->msg[2] == '\030')
        pli->val = pdpvt->msg[1];
    else
        recGblSetSevr(pli, READ_ALARM, INVALID_ALARM);
    return 0;
}
```

Some points of interest:

1. The routine which calls the custom conversion function ignores the return value, so the custom conversion functions must set the record alarm state directly.
2. I put in a sanity check to ensure that the end-of-string character is where it should be.

5.7 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. As you can see, most of the skeleton file values are left unchanged:

```
static long init_ai(int parm)
{
    if (parm == 0) {
        devSupParms.debugFlag = &devAB300Debug;
        devSupParms.respond2Writes = 0;
        devSupParms.timeWindow = TIME_WINDOW;
        devSupParms.hwpvtHead = 0;
        devSupParms.gpibCmds = gpibCmds;
        devSupParms.numparams = NUMPARAMS;
        devSupParms.magicSrq = -1;
        devSupParms.name = "AB300";
        devSupParms.timeout = IO_TIME;
        devSupParms.srqHandler = devGpibLib_srqHandler;
        devSupParms.wrConversion = NULL;
    }
    return (devGpibLib_initDevSup(parm, &DSET_AI));
}
```

Three values have been changed:

1. The debugFlag entry points to the device support debugging flag.
2. The AB300 sends back values in response to commands, but needs no time delay, so the respond2Writes entry is set to 0.
3. The name entry is used for diagnostic purposes only.

The AB300 (like all other serial line 'GPIB' devices) does not generate service requests. The srqHandler field is unchanged from the skeleton file because it doesn't hurt to leave it that way.

6 Add the device support to the application

The makeBaseApp.pl script produces an application Makefile (AB300App/src/Makefile) with a commented-out set of application example source files. Remove the comment character and change the example names to the name of the device support file created in the previous section:

```
AB300_SRCS += devAB300.c
```

You must also link the GPIB support libraries with your application. Add the following line

```
AB300_LIBS += gpib
```

before the

```
AB300_LIBS += $(EPICS_BASE_IOC_LIBS)
```

line in the application Makefile.

7 Modify the application database definition file

Here's where you specify the link between the DSET names defined in the device support file and the DTYP fields in the application database. The `AB300App/src/AB300Include.dbd` file created by `makeBaseApp.pl` needs to be changed to include this information. I used "AB300Gpib" as the device type.

The driver support for serial line 'GPIB' devices must also be included in the application as shown.

```
include "base.dbd"

#
# Define the connection between the DTYP field name and the device DSET tables
#
device(longout, GPIB_IO, devAB300_lo, "AB300Gpib")
device(longin,  GPIB_IO, devAB300_li, "AB300Gpib")
device(ai,      GPIB_IO, devAB300_ai, "AB300Gpib")

#
# Pull in the driver support
#
include "drvTermiosTtyGpib.dbd"
```

8 Create the application database file

Now that the application includes the necessary device and driver support it's possible to create the database describing the actual EPICS process variables associated with the filter wheel.

I created the file `AB300App/Db/AB300.db` with the following contents:

```
record(longout, "$(user):FilterWheel:reset")
{
    field(DESC, "Reset AB300 Controller")
    field(SCAN, "Passive")
    field(DTYP, "AB300Gpib")
    field(OUT,  "#L0 A0 @0")
}
record(longout, "$(user):FilterWheel")
{
    field(DESC, "Set Filter Wheel Position")
    field(SCAN, "Passive")
    field(DTYP, "AB300Gpib")
    field(OUT,  "#L0 A0 @1")
    field(LOPR, 1)
    field(HOPR, 6)
}
```

```

}
record(longin, "$(user):FilterWheel:fbk")
{
    field(DESC, "Filter Wheel Position")
    field(SCAN, "Passive")
    field(DTYP, "AB300Gpib")
    field(INP, "#L0 A0 @2")
    field(LOPR, 1)
    field(HOPR, 6)
}
record(longin, "$(user):FilterWheel:status")
{
    field(DESC, "Filter Wheel Status")
    field(SCAN, "Passive")
    field(DTYP, "AB300Gpib")
    field(INP, "#L0 A0 @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 the device support but must be a valid GPIB address (0-30) since the GPIB address conversion routines check the 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 AB300Include.dbd database definition.

9 Add the database file to the application

The makeBaseApp.pl script put and exampl application database file into AB300App/Db/Makefile as a comment. Replace the example name with the name of the database file created in the previous step, leaving:

```
DB += AB300.db
```

10 Modify the application startup script

The iocBoot/iocAB300/st.cmd application startup script created by the makeBaseApp.pl script needs a few changes to get the application working properly.

1. Ensure that the application database records are loaded. Remove the # and give a reasonable value to the 'user' macro:

```
dbLoadRecords("../db/AB300.db", "user=AB300")
```

2. Add a line to control the debugging level of the serial line 'GPIB' driver. A level of 2 prints out every byte sent to or received from the serial line and is very useful during initial debugging.

```
setDrvTermiosTtyGpibDebug(2)
```

3. Set up the 'link' 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:

```
drvTermiosTtyGpibConfigure(0, "164.54.9.90:4002")
```

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

```
stty("--file=/dev/ttyS0", "cs8", "-parenb", "-crtcts", "clocal")
drvTermiosTtyGpibConfigure(0, "/dev/ttyS0")
```

- If you're using a serial line directly attached to a vxWorks IOC you must first configure the serial port. The following example shows the commands to configure a port on a GreenSprings UART Industry-Pack module. Note that these are vxWorks shell commands and not IOC shell commands.

```
ipacAddCarrier(&vipc616_01, "0x6000, B00000000")
tyGSOctalDrv 1
octalUart0 = tyGSOctalModuleInit("GSIP_OCTAL232", 0x80, 0, 0)
port0 = tyGSOctalDevCreate("/tyGS/0/0", octalUart0, 0, 1000, 1000)
tyGSOctalConfig(port0, 9600, 'N', 1, 8, 'N')
```

Once the serial port has been configured you can invoke the `drvTermiosTtyGpibConfigure` command from either shell as:

```
drvTermiosTtyGpibConfigure(0, "/tyGS/0/0")
```

In all of the above examples the first argument of the `drvTermiosTtyGpibConfigure` command is the link number and must match the L value in the EPICS database record INP and OUT fields.

11 Build the application

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

```
norume> make
```

(**gmake** on solaris).

If all goes well you'll be left with an executable program in `bin/linux-x86/AB300`.

12 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("../db/AB300.dbd", 0, 0)
registerRecordDeviceDriver(pdbbase)
dbLoadRecords("../db/AB300.db", "user=AB300")
setDrvTermiosTtyGpibDebug(2)
drvTermiosTtyGpibConfigure(0, "164.54.9.90:4002")
termiosTtyGpib open connection to 164.54.9.90:4002
termiosTtyGpib opened connection to 164.54.9.90:4002
iocInit()
#####
### EPICS IOC CORE built on Mar 26 2003
### EPICS R3.14.2 $$Name: $$ $$Date: 2003/03/18 22:44:29 $$
```

```
#####  
Starting iocInit  
iocInit: All initialization complete  
epics>
```

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  
termiosTtyGpibWrite 3 \377\377\033  
termiosTtyGpibRead 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  
termiosTtyGpibWrite 1 \035  
termiosTtyGpibRead 2 \001\020  
termiosTtyGpibRead 1 \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
```

```
termiosTtyGpibWrite 2 \017\004
termiosTtyGpibRead 1 \020
termiosTtyGpibRead 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
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
```

```
termiosTtyGpibWrite 1 \035
termiosTtyGpibRead 1 \004
termiosTtyGpibRead 2 \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
```

13 Device Support File

Here is the complete device support file for the AB300 filter wheel (AB300App/src/devAB300.c):

```
/*
 * Copyright (c) 2002 The University of Chicago, as Operator of Argonne
 * National Laboratory.
 * Copyright (c) 2002 The Regents of the University of California, as
 * Operator of Los Alamos National Laboratory.
 * EPICS BASE Versions 3.13.7
 * and higher are distributed subject to a Software License Agreement found
 * in file LICENSE that is included with this distribution.
 */
/* devSkeletonGpib.c */
/* $Id: $ */
/*
 * Based on devXxSkeletonGpib.c:
 * Original Author: John Winans
 * Date: 02-18-92
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <alarm.h>
#include <recGbl.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 devAB300_ai
#define DSET_LI devAB300_li
#define DSET_LO devAB300_lo

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

/*
 * Debugging flag
 */
static int devAB300Debug = 0;
```

```

/*****
 *
 * TIME_WINDOW is the interval (in milliseconds) during which commands should
 * be ignored after the device times out. The ignored commands will appear
 * as errors to device support.
 *
 * IO_TIME is the interval (in milliseconds) in which an I/O operation must
 * complete.
 *
 *****/
#define TIME_WINDOW 2000      /* wait 2 seconds after device timeout */
#define IO_TIME     5000     /* I/O must complete within 5 seconds */

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

    if( pdpvt->msg[2] == '\030')
        pli->val = pdpvt->msg[0];
    else
        recGblSetSevr(pli, READ_ALARM, INVALID_ALARM);
    return 0;
}
static int
convertStatusReply(struct gpibDpvt *pdpvt, int P1, int P2, char **P3)
{
    struct longinRecord *pli = ((struct longinRecord *) (pdpvt->precord));

    if( pdpvt->msg[2] == '\030')
        pli->val = pdpvt->msg[1];
    else
        recGblSetSevr(pli, READ_ALARM, INVALID_ALARM);
    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|GPIBEOS, IB_Q_HIGH, NULL, "\377\377\033", 10, 10,
      NULL, 0, 0, NULL, NULL, '\033'},

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

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

    /* Param 3 -- Query controller status */
    {&DSET_LI, GPIBREAD|GPIBEOS, 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
 *
 * The magic SRQ parm is the parm number that, if specified on an I/O event
 * scanned record, will cause the record to be processed automatically when
 * an unsolicited SRQ interrupt is detected from the device.
 *
 *****/
static long init_ai(int parm)
{
    if(parm==0) {
        devSupParms.debugFlag = &devAB300Debug;
        devSupParms.respond2Writes = 0;
        devSupParms.timeWindow = TIME_WINDOW;
        devSupParms.hwpvtHead = 0;
        devSupParms.gpibCmds = gpibCmds;
        devSupParms.numparams = NUMPARAMS;
        devSupParms.magicSrq = -1;
        devSupParms.name = "AB300";
        devSupParms.timeout = IO_TIME;
        devSupParms.srqHandler = devGpibLib_srqHandler;
        devSupParms.wrConversion = NULL;
    }
    return(devGpibLib_initDevSup(parm,&DSET_AI));
}

```