asynDriver devGpib

Table of Contents

EPICS: asynDriver devGpib Instrument Device Support

Marty Kraimer and Eric Norum

October 19, 2007

NOTE: devGpib is obsolete and should not be used for new applications. StreamDevice should be used instead.

Contents

License Agreement

Introduction

devGpib is the successor to the GPIB support that came with EPICS base 3.13. The 3.13 code was unbundled by Benjamin Franksen, and ultimately became gpibCore which is the 3.14 version of Benjamin's support. devGpib is the successor to the device support portion of gpibCore. The driver part of gpibCore is replaced by asynDriver and asynGpib.

This manual assumes that the reader is familiar with EPICS IOC record and device support and also with asynDriver.

devGpib is one method for implementing EPICS IOC device support for instruments. Other methods are available. If support for an instrument does not exist, consider using STREAMS instead of devGpib.

The 3.13 GPIB support only worked with real GPIB instruments. Since devGpib uses low level asyn drivers, it can also work with other instruments, e.g. serial devices. Thus, this manual uses the terminology *instrument* rather than *GPIB device*.

devGpib requires that a device support module be created for each supported instrument. Facilities are provided to make it relatively easy to write new support.

The following features are provided:

- I/O is performed via database records.
- Instrument device support is written by calling devCommonGpib/devSupportGpib routines.
- devCommonGpib provides support for individual record types.
- devSupportGpib provides record independent support.
- devCommonGpib/devSupportGpib use asynCommon, asynOctet, and asynGpib. Thus, the support will work with any driver that implements these interfaces. If instrument support does not use GPIB specific features, then the support will work with any driver that implements asynCommon and asynOctet. In particular, communication via serial ports is possible.
- devCommonGpib should satisfy most requirements but it is also possible to provide custom support that uses devSupportGpib.

Acknowledgments

John Winans
: John provided the original EPICS GPIB support. Most databases using John's support can be used without modification. With some modification, 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. The low level drivers became separate source modules that implemented the interface. He also created the original support for VXI-11.

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. He also made the conversion to using asyn drivers.

InstallAndBuild

devGpib is bundled with the asynDriver. The code is in library asyn. Thus, once asyn is included as part of an application, devGpib will also be available. It is only necessary to include `devGpib.dbd` in your `<app>Include.dbd` file. In addition, device support for your specific instruments must be installed.

Using Instrument Support

devGpib does not provide support for specific GPIB devices but for implementing such support. Within the EPICS community, support exists for many GPIB devices. The EPICS supported hardware list shows some of the support. If your device is not listed and a message to tech-talk does not provide any help, then you will have to write your own device support. This section assumes that an instrument support is available.

An EPICS record is connected to GPIB by the fields DTYP and INP or OUT.

The DTYP field has the format:

```
field(DTYP,"<device support name>")
```

where

`<device support name>` - Is the name from a device database definition, i.e. a definition of the form:

```
device(<record type>,GPIB_IO,<dsetName>,"<device support name>")
```

The INP or OUT field has the format:

```
field(INP,"#L<link> A<addr> @<number>")
or
field(OUT,"#L<link> A<addr> @<number>")
```

where

`<link>`
: Link number. Low level drivers use portName to provide access to a specific communication interface. In order to keep compatibility with link numbers, the portName MUST be Lxxx where xxx is the value that appears in the #Lxxx portion of the INP or OUT fields of record instances.

`<addr>`
: GPIB address of your device, which can be a primary address or an extended address. A primary address has a value <=30. An extended address is of the form PSS, where P represents the primary address and SS is the secondary address. It is not possible to express an extended address if the primary address is 0. Some examples are:

```
A9    primary address 9
A900  extended address: primary address is 9, secondary address is 0
A906  extended address: primary address is 9, secondary address is 6.
```

`<number>`
: An integer that identifies a gpibCmd definition in a GPIB device support module. If the implementer is nice, documentation like the following is provided:

```
recordType  @<number>   Description
```

If such documentation is not available, look at the device support itself for statements like:

```
/* Param 12 */
{&DSET_LI, GPIBREAD, IB_Q_LOW, "*ESR?",  "%ld", 0, 20, 0, 0, 0, 0, 0, 0},
```

The above states that @12 is a GPIB read command via a longin record. Thus the record definition would be:

```
record(longin,"<name>") {
    field(DTYP,"<device support name>")
    ...
    field(OUT,"#L<link> A<addr> @12")
    ...
}
```

Reports and Timeouts

In order to have these commands available, `devGpib.dbd` must be included as part of the applications `xxxInclude.dbd` file.

A report of all devGpib devices can be generated via the command:

```
dbior("devGpib",level)
```

Three timeouts are defined:

- `timeout` - This is the timeout for individual I/O operations. It is determined by the instrument support.
- `queueTimeout` - Maximum time to wait in a the queue for access to a device instance. The default is 60 seconds and can be changed with the `iocsh` command:

  ```
  devGpibQueueTimeout(interfaceName,gpibAddr,timeout)
  ```

- `srqWaitTimeout` - Maximum time to wait for an SRQ after a GPIBREADW or GPIBEFASTIW has sent a command to the device. The default is 5 seconds and can be changed with the `iocsh` command:

  ```
  devGpibSrqWaitTimeout(interfaceName,gpibAddr,timeout)
  ```

Creating Instrument Device Support

This section describes how to write device support for an instrument. It is assumed that the reader is already familiar with the dialogue required to operate the instrument and EPICS record and device support.

Purpose

An instrument support module provides access to the operating parameters of the instrument.

Overview

Instruments typically have many operating parameters, each of which may be thought of in terms of an EPICS database record type. It is the job of the instrument support designer to map operating parameters to record types. Once this mapping is complete, an instrument support module can be written. For each operating parameter, a gpibCmd must be created.

Device DBD Definition

For each instrument support module, device definitions must be defined:

```
device(<record type>,<link type>,<DSET name>,"<DTYP name>")
```

where

<record type>
: The record type, e.g. ai, ao, ...

<link type>
: Link type. Must be GPIB_IO.

<DSET name>
: Device Support Entry Table name. This is the external name defined in the device support code.

<DTYP name>
: Device Type name. This is what appears in the DTYP field of record instances.

For example, the definitions for the test supplied with devGpib are:

```
device(ai,GPIB_IO,devAiTestGpib,"GPIB Test")
device(ao,GPIB_IO,devAoTestGpib,"GPIB Test")
device(bi,GPIB_IO,devBiTestGpib,"GPIB Test")
device(bo,GPIB_IO,devBoTestGpib,"GPIB Test")
device(longin,GPIB_IO,devLiTestGpib,"GPIB Test")
device(longout,GPIB_IO,devLoTestGpib,"GPIB Test")
device(mbbi,GPIB_IO,devMbbiTestGpib,"GPIB Test")
device(mbbo,GPIB_IO,devMbboTestGpib,"GPIB Test")
device(stringin,GPIB_IO,devSiTestGpib,"GPIB Test")
device(stringout,GPIB_IO,devSoTestGpib,"GPIB Test")
```

For more information about device support, and also how to define INP or OUT links of records, see the EPICS Application Developers Guide.

Instrument Device Support Module

If you are writing a new instrument support module just:

- Create a source directory and 'cd' to it.
- Create a set of instrument support files by running the makeSupport.pl script:

  <*asynTop*>/bin/<*EpicsHostArch*>/makeSupport.pl -t devGpib <*InstName*>
- Modify the src/<*InstName*>.c, src/<*InstName*>.dbd and src/<*InstName*>.db files to provide the functionality needed by your device.

An instrument device support module consists of DSET entries, an array of gpibCmds, efast tables (optional), name tables (optional), a devGpibParmBlock, a debugging flag, an init_ai routine, and custom conversion functions (optional.).

A simplified version of the skeleton file is:

```
/* devSkeletonGpib.c */
...
#include <devCommonGpib.h>
...
/* define all desired DSETs */
...
#define DSET_BO    devBiSkeletonGpib
...
#include <devGpib.h>    /* must be included after DSET defines */
#define TIMEOUT     1.0
#define TIMEWINDOW  2.0
...
/* Strings used by the init routines to fill in the znam,onam,... in BI and BO*/
static  char    *offOnList[] = { "Off", "On" };
static  struct  devGpibNames   offOn = { 2, offOnList, 0, 1 };

static char  *initNamesList[] = { "Init","Init" };
static struct devGpibNames initNames = { 2,initNamesList,0,1 };
/* example EFAST table */
static char *userOffOn[] = {"USER OFF;", "USER ON;", 0};

/* Array of structures that define all GPIB messages */
static struct gpibCmd gpibCmds[] =
```

```
{
  /* Param 0 */
  {&DSET_BO,GPIBCMD,IB_Q_HIGH,"init",0,0,32,NULL,0,0,NULL,&initNames,0},
  /* Param 1 */
  {&DSET_BO, GPIBEFASTO, IB_Q_HIGH,0,0, 0, 32,0, 0, 0, userOffOn, &offOn, 0},
  /* definitions for other parameters follow*/
};
/* The following is the number of elements in the command array above.  */
#define NUMPARAMS sizeof(gpibCmds)/sizeof(struct gpibCmd)
/* User MUST define init_ai */
static long init_ai(int parm)
{
    if(parm==0)  {
        devSupParms.name = "devSkeletonGpib";
        devSupParms.gpibCmds = gpibCmds;
        devSupParms.numparams = NUMPARAMS;
        devSupParms.timeout = TIMEOUT;
        devSupParms.timeWindow = TIMEWINDOW;
        devSupParms.respond2Writes = -1;
    }
}
```

The meaning of each portion of the code should become clear as you read the following sections:

DSET - Device Support Entry Tables

The following statements create the Device Support Entry Tables

```
#define DSET_AI     devAiSkeletonGpib
#define DSET_BI     devBiSkeletonGpib
#define DSET_MBBI    devMbbiSkeletonGpib
...
#include <devGpib.h>    /* must be included after DSET defines */
```

The actual DSETs are created by devGpib.h based on which DSET_xx definitions are defined. A #define must appear for each record type required. DSET_AI must be defined because an init_ai routine must be implemented as described below. If you also define DSET_AIRAW the associated .dbd file must specify different DTYP strings for the DSET_AI and DSET_AIRAW device declarations.

Likewise, if you define both DSET_AO and DSET_AORAW the associated .dbd file must specify different DTYP strings for the DSET_AO and DSET_AORAW device declarations.

devCommonGpib provides device support for many standard record types. It is also possible to create custom support, but this is more difficult.

gpibCmd Definitions

This is where the translation to and from the language of the instrument is defined. The actual table contains one element for each parameter that is made available to the user via the @<number> portion of an INP or OUT field.

In the example above the definitions for the gpibCmds are:

```
static struct gpibCmd gpibCmds[] =
{
  /* Param 0 */
```

```
  {&DSET_BO, GPIBEFASTO, IB_Q_HIGH, 0, 0, 0, 32,0, 0, 0, userOffOn, &offOn, 0},
  /* definitions for other parameters follow*/
};
```

This example defines a single command. A database record using this definition must define field OUT as

```
field(OUT,,"#L<link> A<addr> @0")
```

gpibCmd is

```
typedef struct gpibCmd {
    gDset *dset; /* used to indicate record type supported */
    int type;    /* enum - GPIBREAD...GPIBSRQHANDLER */
    short pri;   /* request priority IB_Q_LOW, IB_G_MEDIUM, or IB_Q_HIGH */
    char *cmd;   /* CONSTANT STRING to send to instrument */
    char *format;/* string used to generate or interpret msg */
    int rspLen;  /* room for response error message */
    int msgLen;  /* room for return data message length */
    /*convert is optional custom routine for conversions */
    int (*convert) (gpibDpvt *pgpibDpvt,int P1, int P2, char **P3);
    int P1;      /* P1 plays a dual role: */
                 /*      For EFAST it is set internally to the
                 /*      number of entries in the EFAST table */
                 /*      For convert it is passed to convert() */
    int P2;      /* user defined parameter passed to convert() */
    char **P3;   /* P3 plays a dual role: */
                 /*      For EFAST it holds the address of the EFAST table */
                 /*      For convert it is passed to convert() */
    devGpibNames *pdevGpibNames; /* pointer to name strings */
    char * eos; /* input end-of-string */
} gpibCmd;
```

where

`dset`
: Address of the Device Support Entry Table (DSET) that describes the record type supported by the table entry.

`type`
: Type of GPIB I/O operation that is to be performed. The `type` field must be set to one of the enumerated values declared in devSupportGpib.h, i.e. GPIBREAD,...,GPIBSRQHANDLER. See next section for the definitions.

`pri`
: Processing priority of the I/O operation. Must be `IB_Q_HIGH`, `IB_Q_MEDIUM`, or `IB_Q_LOW`.

`cmd`
: Constant string that is used differently depending on the value of `type`. See the discussion of `type` below. Set this field to 0 if not used.

`format`
: Printf/scanf format string that is used differently depending on the value of `type`. See the discussion of `type`. Set this field to 0 if not used.

`rspLen`
: Size needed to read back from a device when performing a respond2Writes read operation. If a gpibCmd is a read operation or a write operation that does not respond, then set this field to zero. When devGpib initializes, it finds the maximum rspLen of all commands associated with a port instance and allocates storage of that size.

`msgLen`
: Size needed to read or write from a device. If not needed, set it to zero. When devGpib initializes it finds the maximum msgLen of all commands associated with a port instance and allocates storage of that size.

`convert`

A conversion function that has the prototype:

```
int (*convert) (gpibDpvt *pgpibDpvt,int P1, int P2, char **P3);
```

The use depends on the pgpibCmd->type. See below for details. Set to 0 when no conversion function is present.

Conversion routines should return 0 to signify a successful conversion. If a convert routine finds an error, it should do the following:

◊ Generate an error message as follows:

```
epicsSnprintf(pasynUser->errorMessage,pasynUser->errorMessageSize,
                                                    "<format>",...);
```

◊ return -1 to signify failure

`P1`

This plays a dual role.

For EFAST operations it is set equal to the number of entries in the efast table by `devSupportGpib:init_record`.

For other operations, it is an integer passed to the conversion function specified in `convert`.

`P2`

Integer passed to the conversion function specified in `convert`.

`P3`

This field plays a dual role.

When `type` is one of the EFAST operations, this field points to the EFAST table. See the EFAST operation descriptions under the `type` field definitions and the section "Efast Tables" for more on the use of this field. Set this field to 0 when it is not used.

For other operations, it is passed to the conversion function specified in `convert`. It has a `char**` value.

`pdevGpibNames`

Pointer to a Name Table. Name tables are described in the section "Name Tables". Set this to 0 when no Name Table is used.

`eos`

Input message termination string. It can be:

`0`

A NULL pointer means that this read operation will be terminated by the current asynOctetSetInputEos value, if any.

`""`

An empty string signifies that a single null character ('\0') is the terminator for this operation.

non-empty string (e.g. `"\n"`)

The termination characters (not including the null character terminating the string) for this operation. For example, `"\n"` sets the message terminator to a single ASCII newline (`'\n'`) character. Some drivers, e.g. the VXI-11, allow only a single termination character.

Definitions for pgpibCmd->type

The following describes the semantics for the device support provided by the devGpib support provided with asynDriver. If an application extends this support, it must document its changes.

GPIBREAD

Supports record types: ai, bi, event, longin, mbbi, mbbiDirect, stringin, and waveform. For all of these types the following is done.

- ◊ Send `pgpibCmd->cmd` to the instrument.
- ◊ Read from the instrument into `pgpibDpvt->msg`.
  `pgpibCmd->msgLen` must specify a size large enough for the message.

If `convert` is defined then:

- ◊ `convert` is called. It is expected to give a value to the appropriate field (RVAL for bi, mbbi, mbbiDirect and VAL for all others) of the record or return -1. Note that it is the responsiblity of the custom conversion routine to set or clear the record .UDF field.
- ◊ The field `pgpibDpvt->msgInputLen` contains the number of bytes in the last read msg. This allows messages with null characters to be processed.
- ◊ Record completion occurs.

If `convert` is not defined then what is done depends on the record type.

- ◊ bi, event, longin, mbbi, and mbbiDirect.

  `pgpibDpvt->msg` is converted and the result put into field RVAL (bi, mbbi, mbbiDirect) or VAL (event, longin). If pgpibCmd->format is defined it is used for the conversion, otherwise a format appropriate to the data type of RVAL/VAL is used.
- ◊ ai

  `pgpibDpvt->msg` is converted and the result put into field VAL or RVAL. VAL is used if the DSET does NOT define special_linconv and RVAL is used if special_linconv is defined. If pgpibCmd->format is defined, it is used for the conversion. Otherwise, a format appropriate to the field is used. The DSET generated by devGpib.h does not define special_linconv.
- ◊ stringin

  `pgpibDpvt->msg` is converted and the result put into field VAL. If pgpibCmd->format is defined it is used for the conversion, otherwise "%39c" is used.
- ◊ waveform

  Unless FTVL is menuFtypeCHAR, an error is generated and the record is put into alarm. If FTVL is menuFtypeCHAR,then epicsSnprintf is used to convert `pgpibDpvt->msg` into BPTR. If format is defined it is used, otherwise "%s" is used.

GPIBWRITE

Supports record types: ao, bo, longout, mbbo, mbboDirect, stringout, and waveform.

- ◊ If `convert` is defined, it is called. It must put a command string into pgpibDpvt->msg. It can return:
  - · -1

    Signifies error. The operation is aborted and the record put into alarm.
  - · 0

    Success and call strlen to determine the length of msg.
  - · >0

    Success and the return value is the number of bytes in msg.

◊ If `convert` is not defined, then what happens is determined by the record type.

· ao

If special_linconv is NOT defined in DSET, the RVAL field is converted as a long into msg. If special_linconv is defined, OVAL is converted as a double into msg.

· bo, mbbo, and mbboDirect

VAL is converted as an unsigned long into msg.

· longout

VAL is converted as a long into msg.

· stringout

VAL is converted into msg via epicsSnprintf. If pgpibCmd->format is defined it is used, otherwise "%s" is used.

· waveform

BPTR is converted into msg via epicsSnprintf. If pgpibCmd->format is defined it is used, otherwise "%s" is used.

◊ `pgpibDpvt->msg` is sent to the instrument.

GPIBCVTIO

Supports record types:

ai, ao, bi, bo, event, longin.longout, mbbi, mbbo, mbbiDirect, mmboDirect, stringin, stringout, waveform.

All I/O is done by the `convert` routine, which must be defined. `convert` is called by a callback routine and thus can make an arbitrary number of calls to low level drivers. It is passed the address of `gpibDpvt` which contains the information needed to call the low level drivers: `asynCommon, asynOctet, and asynGpib`. Note that `asynGpib` may not be present, i.e. `pasynGpib` is null. `gpibDpvt` also contains a field `pupvt` which can be used by the convert routine. Is is initialized to null. The macro `gpibCmdGet` can be used to get the address of `gpibCmd` which contains other usefull information.

If the End of String terminator needs to be changed, it must be changed by calling `pdevSupportGpib->setEos` rather than calling `pgpibDpvt->pasynOctet->setEos` Also when the `convert` routine has finished with read operations it must call `pdevSupportGpib->restoreEos`

Subsection "gpibCmd convert example" below provides an example.

GPIBCMD

Supports record types: ao, bo, longout, mbbo, mbboDirect, stringout, and waveform.

Send the command string specified in `pgpibCmd->cmd` to the instrument exactly as specified.

GPIBACMD

This is like GPIBCMD except that ATN is held active. This should rarely be necessary.

GPIBSOFT

No I/O is done. It calls `pgpibCmd->convert`. `pgpibCmd->convert` must be defined If GPIBSOFT fails, it calls `asynPrint` with mask `ASYN_TRACE_ERROR` and also puts the record into alarm.

GPIBREADW

Like GPIBREAD except for that it waits for the device to issue an SRQ before it issues the read request.

GPIBRAWREAD

Like GPIBREAD except that no command is sent to the instrument.

GPIBEFASTO

This operation type is only valid on BO and MBBO record types. `pgpibCmd->P3` must contain the address of an efast table. At init time some checks are made to see that an efast table is defined, but if it is defined incorrectly problems may arise.

The following is done:

◊ Device support sets `pibDpvt.efastVal` equal to the VAL field.
◊ If `pgpibCmd->cmd` is not null, then `msgLen` must also be specified. `msg` is set equal to the concatenation of `cmd` and the efast value specified by `pgpibCmd->P3`. The resulting `msg` is sent.
◊ If `pgpibCmd->cmd` is null, then the string pointed to by `pgpibCmd->P3[efastVal]` is sent.

GPIBEFASTI

This operation type is only valid on BI and MBBI record types.

The following is done:

◊ Send the command string specified in `cmd` to the instrument exactly as specified.
◊ Read the data from the instrument and place it into `pgpibDpvt->msg`.
◊ Compare `msg` with each element of the EFAST table referenced by `pgpibCmd->P3`.
◊ devSupportGpib sets `pgpibDpvt->efastVal` to the index of the EFAST table that matched.
◊ Device support sets RVAL equal to `pgpibDpvt->efastVal`.

GPIBEFASTIW

This operation type is like GPIBEFASTI except that it waits for the device to raise SRQ before the data is read.

GPIBIFC

Valid only for BO records. If rval = (0,1) then (do nothing, pulse IFC). IFC is one of the GPIB Bus Management Lines.

Only define `dset`, `type`, and `pri`. A default pdevGpibNames is provided. For example

```
{&DSET_BO, GPIBIFC, IB_Q_LOW, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

GPIBREN

Valid only for BO records. If rval = (0,1) then (drop,assert) REN. REN is one of the GPIB Bus Management Lines.

If devices are in the LLO state they can be removed from this state by toggling the REN line, i.e. turn it off and then turn it back on.

Only define `dset`, `type`, and `pri`. A default pdevGpibNames is provided.

GPIBDCL

Valid only for BO records. If rval = (0,1) then (do nothing, send DCL). DCL is a Universal GPIB command, i.e. it applys to all devices on the link

Only define `dset`, `type`, and `pri`. A default pdevGpibNames is provided.

GPIBLLO

If rval = (0,1) then (do nothing, send LLO). LLO is a Universal GPIB command, i.e. it applys to all devices on the link

After a LLO, the first time a device is addressed it will disable local control, i.e. the front pannel controls will not respond. To remove devices from this state toggle the REN line. A single device can temporarily be removed from LLO by sending the GPIBCTL command but it will go back to LLO state as soon as it is again addressed.

Only define `dset`, `type`, and `pri`. A default pdevGpibNames is provided.

GPIBSDC

If rval = (0,1) then (do nothing, send SDC). SDC is an addressed GPIB command, i.e. it applys only to the addressed device.

Only define `dset`, `type`, and `pri`. A default pdevGpibNames is provided.

GPIBGTL

If rval = (0,1) then (do nothing, send GTL). GTL is an addressed GPIB command, i.e. it applys only to the addressed device.

If a device has local control locked out, local control can be temporarily granted by issuing this command. However the next time the device is addressed it will again disable local control.

Only define `dset`, `type`, and `pri`. A default pdevGpibNames is provided.

GPIBSRQHANDLER

This is used to handle an unsolicited SRQ from gpibAddr, i.e. a device raises SRQ when a read wait, e.g. GPIBREADW or GPIBEFASTIW, is not active. This request is implemented only for longin records. When the device issues an SRQ, the status byte is put into the val field and the record is processed. It is expected that the record will forward link to records that issue GPIB commands to read the information that caused the SRQ.

Efast (Enumerated Fast I/O) Tables

A device's command set often has things like "OFF" or "ON" in the command string. It is convenient to issue such commands via binary and multibit binary records. Efast tables specify such strings. Simply specify the string value for each of the possible states of the VAL field of the record.

The format of an efast table is:

```
    static char  *tableName[] = {
        "OFF",     /* when VAL = 0 */
        "ON",     /* when VAL = 1 */
        0         /* list terminator */
  };
```

And is referenced in an output parameter table entry like this:

```
    {&DSET_BO, GPIBEFASTO, IB_Q_HIGH, 0, 0, 0, 0,0, 0, 0, tableName, 0, 0},
```

For an input entry, it would look like this:

```
    {&DSET_BI, GPIBEFASTI, IB_Q_HIGH, "<command>", 0, 0, 50, 0, 0, 0,

     tableName, 0, 0},
```

The efast table MUST be 0 terminated.

For outputs the VAL field is used to index into the efast table and select a string to send to the instrument. If `cmd` is 0, the string is sent to the instrument exactly as as it appears in the efast table. If `cmd` is defined, then that string is prepended to the string obtained from the efast table.

For input operations, the `cmd` string is sent to the instrument, and `msg` is read from the instrument. `msg` is compared against each of the entries in the efast table starting at the zeroth entry. The slot number of the first table entry that matches the response string is used as the setting for the RVAL field of the record. When strings are compared, they are compared from left to right until the number of characters in the efast table are checked. When ALL of the characters up to but **NOT** including the 0 of the string in the efast table match the corresponding characters of the response string, it is considered a valid match. This allows the user to check response strings fairly fast. For example, if a device returns something like "ON;XOFF;9600" or "OFF;XOFF;9600" in response to a status check, and you wish to know if the first field is either "OFF" or "ON", your efast table could look like this:

```
    static char *statCheck[] = {
        "OFF",      /* set RVAL to 0 */
        "ON",      /* set RVAL to 1 */
        0};     /* list terminator */
```

The 0 field is *extremely* important. If it is omitted and the instrument gets confused and responds with something that does not start with an "OFF" or "ON", the GPIB support library code will end up running off the end of the table.

In the case when none of the choices in an efast table match for an input operation, the record is placed into a INVALID alarm state.

devGpibNames - Name Table

For binary and multibit binary records, the choice fields of a record can be assigned values at record initialization. If pdevGpibNames has a value, then when a record is initialized, the instrument support uses the associated name table to assign values to choice fields that have not been assigned values. These name tables have **nothing** to do with I/O operations.

`devGpibNames` is:

```
struct devGpibNames {
    int count;            /* CURRENTLY only used for MBBI and MBBO */
    char **item;
    unsigned long *value; /* CURRENTLY only used for MBBI and MBBO */
    short nobt;           /* CURRENTLY only used for MBBI and MBBO */
};
```

To use a name table, the address of the table must be put into `pdevGpibNames` of the parameter table. The table format for a multibit record type looks like this:

```
    static char *tABCDList[] = {
        "T",                /* zrst*/
        "A",                /* onst */
        "B",                /* twst */
        "C",                /* thst */
        "D"};               /* frst */

    static unsigned long tABCDVal[] = {
        1,                 /* zrvl */
        2,                 /* onvl */
        3,                 /* twvl */
        5,                 /* thvl */
        6 };               /* frvl */

    static devGpibNames tABCD = {
```

```
        5,                  /* number of elements in string table */
        tABCDList,          /* pointer to string table */
        tABCDVal,           /* pointer to value table */
        3 };                /* value for the nobt field */
```

The table format for a binary record type looks like this:

```
    static char *disableEnableList[] = {
        "Disable",          /* znam */
        "Enable" };         /* onam */

    static devGpibNames disableEnable = {
        2,                  /* number of elements */
        disableEnableList,  /* pointer to strings */
        0,               /* pointer to value list */
        1};                 /* number of valid bits */
```

`devGpibNames` is defined in `devSupportGpib.h`. For binary records, the strings are placed into the name fields in order from lowest to highest as shown above. For multibit binary records, up to sixteen strings can be defined. A `devGpibNames` structure referencing these strings is then defined.

`value` and `nobt` are not used for binary record types, but should be specified anyway as if the binary record was a multibit binary record with only 2 values.

For multibit record types, the name strings, values, and NOBT fields are filled in from the `devGpibNames` information. For binary record types, only the znam and onam fields are filled in.

Name strings (and their associated values in the multibit cases) are not filled in if the database designer has assigned them values.

Defining the devGpibParmBlock

Each DSET of the instrument support contains the address of a `devGpibParmBlock`. `init_ai` MUST initialize this structure. For example:

```
static long init_ai(int parm)
{
    if(parm==0) {
        devSupParms.name = "devSkeletonGpib";
        devSupParms.gpibCmds = gpibCmds;
        devSupParms.numparams = NUMPARAMS;
        devSupParms.timeout = TIMEOUT;
        devSupParms.timeWindow = TIMEWINDOW;
        devSupParms.respond2Writes = -1;
    }
    return(0);
}
```

`devGpibParmBlock` is:

```
struct devGpibParmBlock {
    char *name;         /* Name of this device support*/
    gpibCmd *gpibCmds;  /* pointer to gpib command list */
    int numparams;  /* number of elements in the command list */
    double timeout; /* seconds to wait for I/O */
    double timeWindow;  /* seconds to stop I/O after a timeout*/
```

```
    double respond2Writes; /* set >= 0 if device responds to writes */
    /*The following are set by devSupportGpib*/
    int  msgLenMax;      /*max msgLen all commands*/
    int  rspLenMax;      /*max rspLen all commands*/
};
```

gpibCmds

`name`
: The device support module type name. Used only to generate diagnostic messages.
: Address of an array of gpibCmd definitions.

`numparams`
: The number of gpibCmds defined. A macro should be used to set its value.

`timeout`
: timeout in seconds for an individual I/O operation.

`timeWindow`
: The number of seconds after a timeout before a new I/O operation will be issued to the device. During this time window, any I/O operations directed to the timed out device will result in an error, and the appropriate alarm status will be raised for the record (either READ_ALARM or WRITE_ALARM depending on the record type and VALID_ALARM in all cases.)

`respond2Writes`
: This field is for devices that echo writes. If respond2Writes >=0 then after an output command the following is done:
    - ◊ If pgpibCmd->rspLen is <=0 no action is taken. This is a way to override respond2Writes for specific commands.
    - ◊ If respond2Writes is >0 then a wait of respond2Writes milliseconds occurs.
    - ◊ A read of up to pgpibCmd->rspLen bytes (terminated earlier by GPIB EOI or by the terminator string, if any) is read into pgpibDpvt->rsp.

`msgLenMax`
: This is set by devGpibSupport. The value is that of the maximum size input message, i.e. the largest msgLen defined in gpibCmds.

`rspLenMax`
: This is set by devGpibSupport. The value is that of the maximum size response message, i.e. the largest rspLen defined in gpibCmds.

SRQ Processing

If the low level device driver implements interface `asynGpib`, then devSupportGpib registers itself to handle SRQs. It defines two types of SRQs: solicited and unsollicited. Solicited SRQs are for GPIBREADW and GPIBEFASTIW commands. If an SRQ is raised for a gpibAddr that does not have an outstanding GPIBREADW or GPIBEFASTIW command, the SRQ is considered unsolicited.

When the devSupportGpib receives an unsolicited SRQ and it has a registered handler for the gpibAddr, it calls the handler. Otherwise it issues a message that it received an unsolicited SRQ.

The device support for the longinRecord supports gpibCmd type GPIBSRQHANDLER. It calls the devSupportGpib registerSrqHandler method. When its interruptCallbackInt32 gets called, it puts the SRQ status byte into the VAL field and then makes a request to process the record. By forward linking this record to other records, the user can handle SRQs from specific devices. Note that the callback is an interruptCallbackInt32 since asynGpib handles SRQ callbacks via the asynInt32 interface and using asynUser.reason = ASYN_REASON_SIGNAL.

To save time during SRQ polling operations it is possible to exclude a device which has become disconnected from the bus for some reason and then to restore the device to the polling list when the device is reattached. A binary output record with the following command table entry is used for this purpose:

/* Add/remove address from SRQ polling list */

{ &DSET_BO,GPIBCVTIO,IB_Q_HIGH,NULL,NULL,0,0,boSRQonOff,0,0,NULL,NULL,NULL }

If the binary-output record rval is (0, 1) then (remove, add) device (from, to) the SRQ polling liist. No commands are sent to the bus. Before exluding a device from being polled stop any communications to it, i.e. set SCAN-field(s) to PASSIVE and turn off the device.

gpibCmd convert example

The asyn distribution includes an example of how to implement the convert parameter for a gpibCmd. The example is in asyn/devGpib/devGpibConvertExample.c. It defines three gpibCmds for stringin and three for stringout records. All three input commands have the same result and all three output commands have the same result. The difference is how much is done by devGpib support and how much is done by the instrument support.

The example show how a convert routine can access fields from the devGpibsupport.

The command tables are:

```
static struct gpibCmd gpibCmds[] =
{
  /* Param 0 */
  {&DSET_SI,GPIBREAD,IB_Q_LOW,"*IDN?", 0,0,200,0,0,0,0,0,0},
  /* Param 1 simple convert   */
  {&DSET_SI,GPIBREAD,IB_Q_LOW,"*IDN?",0,0,200,readString,0,0,0,0,0},
  /* Param 2,example of GPIBCVTIO */
  {&DSET_SI,GPIBCVTIO,IB_Q_LOW,"*IDN?",0,0,200,readCvtio,0,0,0,0,0},
  /* Param 3 */
  {&DSET_SO,GPIBWRITE,IB_Q_LOW,0,0,0,200,0,0,0,0,0,0},
  /* Param 4 simple convert   */
  {&DSET_SO,GPIBWRITE,IB_Q_LOW,0,0,0,200,writeString,0,0,0,0,0},
  /* Param 5,example of GPIBCVTIO */
  {&DSET_SO,GPIBCVTIO,IB_Q_LOW,0,0,0,200,writeCvtio,0,0,0,0,0}
};
```

The command for Param 0 lets the devGpib support do everything as follows:

- The command "*IDN?" is sent to the instrument.
- A responds is read back from the instrument.
- device suipport for the stringin record copies the response to the VAL field of the record.

The command for Param 1 is similar except that, after the response is read from the instrument, readString is called. It copies the response to VAL.

The command for Param 2 causes the devGpib support to call readCvtio without doing any I/O. The convert routine is responsible for all I/O.

The command for Param 3 lets the devGpib support do everything as follows:

- Device support for the stringout record copies the current value of the VAL field to a message buffer.
- The message buffer is sent to the instrument.

The command for Param 4 is similar except that writeString is called to move the value of the VAL field to the message buffer.

The command for Param 5 causes the devGpib support to call writeCvtio without doing any I/O. The convert routine is responsible for all I/O.

The actual code for the convert routines is:

```
static int readString(gpibDpvt *pgpibDpvt,int P1, int P2, char **P3)
{
    stringinRecord *precord = (stringinRecord*)pgpibDpvt->precord;
    strncpy(precord->val,pgpibDpvt->msg,sizeof(precord->val));
    precord->val[sizeof(precord->val) - 1] = 0;
    return(0);
}
static int readCvtio(gpibDpvt *pgpibDpvt,int P1, int P2, char **P3)
{
    stringinRecord *precord = (stringinRecord*)pgpibDpvt->precord;
    asynUser *pasynUser = pgpibDpvt->pasynUser;
    gpibCmd *pgpibCmd = gpibCmdGet(pgpibDpvt);
    asynOctet *pasynOctet = pgpibDpvt->pasynOctet;
    void *asynOctetPvt = pgpibDpvt->asynOctetPvt;
    asynStatus status;
    size_t nchars = 0, lenmsg = 0;
    pgpibDpvt->msgInputLen = 0;

    assert(pgpibCmd->cmd);
    lenmsg = strlen(pgpibCmd->cmd);
    status  = pasynOctet->write(asynOctetPvt,pasynUser,
        pgpibCmd->cmd,lenmsg,&nchars);
    if(nchars==lenmsg) {
        asynPrintIO(pasynUser,ASYN_TRACEIO_DEVICE,pgpibCmd->cmd,nchars,
                                            "%s readCvtio\n",precord->name);
    } else {
        asynPrint(pasynUser,ASYN_TRACE_ERROR,
            "%s write status \"%s\" requested %d but sent %d bytes\n",
                precord->name,pasynUser->errorMessage,lenmsg,nchars);
            return -1;
    }
    if(!pgpibDpvt->msg) {
        asynPrint(pasynUser,ASYN_TRACE_ERROR,
            "%s pgpibDpvt->msg is null\n",precord->name);
        nchars = 0; return -1;
    } else {
        status = pasynOctet->read(asynOctetPvt,pasynUser,
            pgpibDpvt->msg,pgpibCmd->msgLen,&nchars,0);
    }
    asynPrint(pasynUser,ASYN_TRACE_FLOW,"%s readCvtio nchars %d\n",
        precord->name,nchars);
    if(nchars > 0) {
        asynPrintIO(pasynUser,ASYN_TRACEIO_DEVICE,pgpibDpvt->msg,nchars,
            "%s readCvtio\n",precord->name);
    } else {
        asynPrint(pasynUser,ASYN_TRACE_ERROR,
            "%s read status \"%s\" nin %d\n",
            precord->name, pasynUser->errorMessage,nchars);
        pgpibDpvt->msgInputLen = 0;
        return -1;
    }
    pgpibDpvt->msgInputLen = nchars;
```

```
    if(nchars<pgpibcmd->pgpibcmd->msgLen) pgpibDpvt->msg[nchars] = 0;
    readString(pgpibDpvt,P1,P2,P3);
    return 0;
}

static int writeString(gpibDpvt *pgpibDpvt,int P1, int P2, char **P3)
{
    asynUser *pasynUser = pgpibDpvt->pasynUser;
    stringoutRecord *precord = (stringoutRecord*)pgpibDpvt->precord;
    int            nchars;
    gpibCmd *pgpibCmd = gpibCmdGet(pgpibDpvt);
    char *format = (pgpibCmd->format) ? pgpibCmd->format : "%s";

    if(!pgpibDpvt->msg) {
        asynPrint(pasynUser,ASYN_TRACE_ERROR,
            "%s no msg buffer. Must define gpibCmd.msgLen > 0.\n",
            precord->name);
        recGblSetSevr(precord,WRITE_ALARM, INVALID_ALARM);
        return -1;
    }
    nchars = epicsSnprintf(pgpibDpvt->msg,pgpibCmd->msgLen,format,precord->val);
    if(nchars>pgpibCmd->msgLen) {
        asynPrint(pasynUser,ASYN_TRACE_ERROR,
            "%s msg buffer too small. msgLen %d message length %d\n",
            precord->name,pgpibCmd->msgLen,nchars);
        recGblSetSevr(precord,WRITE_ALARM, INVALID_ALARM);
        return -1;
    }
    asynPrint(pasynUser,ASYN_TRACE_FLOW,"%s writeMsgString\n",precord->name);
    return nchars;
}

static int writeCvtio(gpibDpvt *pgpibDpvt,int P1, int P2, char **P3)
{
    stringoutRecord *precord = (stringoutRecord*)pgpibDpvt->precord;
    asynUser *pasynUser = pgpibDpvt->pasynUser;
    asynOctet *pasynOctet = pgpibDpvt->pasynOctet;
    void *asynOctetPvt = pgpibDpvt->asynOctetPvt;
    asynStatus status;
    size_t nsent = 0, lenmsg = 0;
    pgpibDpvt->msgInputLen = 0;

    lenmsg = writeString(pgpibDpvt,P1,P2,P3);
    if(lenmsg <= 0) return -1; status=pasynOctet->write(asynOctetPvt,pasynUser,
        pgpibDpvt->msg,lenmsg,&nsent);
    if(nsent==lenmsg) {
        asynPrintIO(pasynUser,ASYN_TRACEIO_DEVICE,pgpibDpvt->msg,lenmsg,
                                        "%s writeCvtio\n",precord->name);
    } else {
        asynPrint(pasynUser,ASYN_TRACE_ERROR,
            "%s write status \"%s\" requested %d but sent %d bytes\n",
                precord->name,pasynUser->errorMessage,lenmsg,nsent);
            return -1;
    }
    return 0;
}
```

devCommonGpib

The facilities described above should satisfy most gpib requirements. This section and the next explain how to provide additional support. For example, support could be written for additional record types. This section explains devCommonGpib, and the next section explains devSupportGpib.

devCommonGpib provides support for specific record types by making calls to devSupportGpib. As explained above, an instrument support module must define some combination of DSET_AI,...,DSET_WF and then include devGpib.h. devGpib.h defines DSETs (Device Support Entry Tables) that refer to methods implemented in devCommonGpib.c

If you want to write additional support code that can be used for writing instrument specific device support, the easiest way is to start with code in devCommonGpib and modify it.

devGpib provides three header files:

- devCommonGpib.h - This defines the function prototypes for the methods implemented in devCommonGpib.c
- devGpib.h - Included in instrument specific device support. It generates DSETs referring to the functions implemented in devCommonGpib.
- devSupportGpib.h - Describes the structures gpibCmd, devGpibNames, and devGpibParmBlock described above. It also describes additional structures described in the next section.

---

devSupportGpib

devSupportGpib.h describes all the public structures used by devGpib. The structures gpibCmd, devGpibNames, and devGpibParmBlock were described above. They are needed to create an instrument device support module. The remaining structures are used by devCommonGpib, devSupportGpib, or other support. These structures are: gDset, gpibDpvt, devGpibPvt, and devSupportGpib.

gDset

A gDset is an "overloaded" definition of a DSET. A gDset looks to iocCore like a regular DSET, but it has an additional field that is the address of a devGpibParmBlock.

```
struct gDset
{
    long number;
    DEVSUPFUN funPtr[6];
    devGpibParmBlock *pdevGpibParmBlock;
};
```

where

`number`
: This is required for a DSET. It should always be initialized to 6.

`funPtr`
: Pointers to device support functions. Allowing for 6 is sufficient for all the standard types in EPICS base.

`pdevGpibParmBlock`
: The address of the devGpibParmBlock for this instrument.

gpibDpvt

The dpvt field of a record with devGpib device support contains the address of a gpibDpvt

```
struct gpibDpvt
{
    devGpibParmBlock *pdevGpibParmBlock;
    CALLBACK callback;
    dbCommon *precord;
    asynUser *pasynUser;
    asynCommon *pasynCommon;
    void *pasynCommonPvt;
    asynOctet *pasynOctet;
    void *pasynOctetPvt;
    asynGpib *pasynGpib;
    void *pasynGpibPvt;
    int parm;                   /* parameter index into gpib commands */
    char *rsp;                  /* for read/write message error responses */
    char *msg;                  /* for read/write messages */
    int  msgInputLen;           /* number of characters in last READ*/
    int efastVal;               /* For GPIBEFASTxxx */
    void     *pupvt;            /*private pointer for custom code*/
    devGpibPvt *pdevGpibPvt;  /*private for devGpibCommon*/
};
```

where

`pdevGpibParmBlock`
: Address of the devGpibParmBlock. It is the same value as `pgDset->pdevGpibParmBlock`

`callback`
: For use by completeProcess.

`precord`
: Address of the record.

`pasynUser`
: Address of an asynUser which is needed to call asynDriver support.

`pasynCommon,pasynCommonPvt`
: Used to call asynCommon methods. If initialization is successful these will have a value.

`pasynOctet,pasynOctetPvt`
: Used to call asynOctet methods. If initialization is successful these will have a value.

`pasynGpib,pasynGpibPvt`
: Used to call asynGpib methods. These may be 0 even if initialization is successful. If a record that requires asynGpib is processed and pasynGpib is 0, then an error message is generated and the record is put into alarm.

`parm`
: The value of the parm field of INP or OUT. It locates the gpibCmd for its record.

`rsp`
: If respond2Writes is >=0 and rspLen is >0, this is the buffer that holds the read after each write.

`msg`
: A buffer of length msgLen. It is used by several GPIBXXX request types.

`msgInputLen`
: The length of the last input message.

`efastVal`
: Used for GPIBEFASTO, GPIBEFASTI, GPIBEFASTIW. For GPIBEFASTO the device support sets efastVal from RVAL or VAL before queuing an I/O request. For GPIBEFASTI and GPIBEFASTIW, the device support sets RVAL or VAL from efastVal after an I/O request completes.

`pupvt`

This is a private pointer for use by specialized instrument support.

pdevGpibPvt
: This is used by devSupportGpib. Note that the structure is described in devSupportGpib.c, i.e. it is private.

devSupportGpib

describes methods implemented by devSupportGpib.c.

```
/* If a method returns int then (0,-1) => (OK, failure) */
typedef void (*gpibWork)(gpibDpvt *pgpibDpvt,int failure);
typedef int (*gpibStart)(gpibDpvt *pgpibDpvt,int failure);
typedef void (*gpibFinish)(gpibDpvt *pgpibDpvt,int failure);

typedef int (*gpibWork)(gpibDpvt *pgpibDpvt,int failure);
struct devSupportGpib {
    long (*initRecord)(dbCommon *precord, struct link * plink);
    void (*processGPIBSOFT)(gpibDpvt *pgpibDpvt);
    void (*queueReadRequest)(gpibDpvt *pgpibDpvt,gpibStart start,gpibFinish finish);
    void (*queueWriteRequest)(gpibDpvt *pgpibDpvt,gpibStart start, gpibFinish finish);
    /* queueRequest returns (0,1) for (failure,success) */
    int (*queueRequest)(gpibDpvt *pgpibDpvt, gpibWork work);
    void (*registerSrqHandler)(
        gpibDpvt *pgpibDpvt,interruptCallbackInt32 handler,void *userPrivate);
    int (*writeMsgLong)(gpibDpvt *pgpibDpvt,long val);
    int (*writeMsgULong)(gpibDpvt *pgpibDpvt,unsigned long val);
    int (*writeMsgDouble)(gpibDpvt *pgpibDpvt,double val);
    int (*writeMsgString)(gpibDpvt *pgpibDpvt,const char *str);
    int (*readArbitraryBlockProgramData)(gpibDpvt *pgpibDpvt);
    int (*setEos)(gpibDpvt *pgpibDpvt,gpibCmd *pgpibCmd);
    int (*restoreEos)(gpibDpvt *pgpibDpvt,gpibCmd *pgpibCmd);
    void (*completeProcess)(gpibDpvt *pgpibDpvt);
};
epicsShareExtern devSupportGpib *pdevSupportGpib;
```

gpibWork
: Prototype of a function that is called when a request is dequeued by asynManager. This can be either a function passed to queueRequest or an internal function that devGpibSupport uses to implement queueReadRequest and queueWriteRequest.

gpibStart
: If queueReadRequest or queueWriteRequest is called, the internal gpibWork function calls the start routine before it processes the request.

gpibFinish
: If queueReadRequest or queueWriteRequest is called, the internal gpibWork function calls the finish routine when it is done processing.

initRecord
: This initializes all the devGpib structures attached to the record starting with dbCommon.dpvt.

processGPIBSOFT
: If pgpibCmd->type is GPIBSOFT, just call this method.

queueReadRequest
: This handles GPIBREADW, GPIBEFASTIW, GPIBREAD, GPIBEFASTI, and GPIBRAWREAD. The only thing device support needs to provide is work functions to do record specific processing of the received message. The `start` function is called before devGpibSupport issues write/reads and `finish` is called after the read. `start` is optional and not usually required for read operations. The `msg` buffer MUST not be accessed except via the start or finish work functions.

queueWriteRequest

This handles GPIBWRITE, GPIBCMD, GPIBACMD, and GPIBEFASTO. Device support provides a `start` and `finish` work function. The `start` function is called before devGpibSupport issues a write and `finish` is called after the write. The msg buffer MUST not be accessed except via the `start` or `finish` work functions. `start` must set `msg` or `efastVal` depending on the `pgpibCmd->type`.

`queueRequest`
: The device support provides a work function that is called when the request is dequeued. queueRequest returns (0,1) if the request (was not, was) queued.

`report`
: This generates a report or all devGpib devices.

`registerSrqHandler`
: Registers an SRQ handler for gpibAddr.

`writeMsgLong,writeMsgULong,writeMsgDouble,writeMsgString`
: pgpibDpvt->msg is written from pgpibCmd->format and the value passed by the caller.

`readArbitraryBlockProgramData`
: Reads zero or more preamble (non-`#`) characters followed by IEEE-488.2 definite-length arbitrary block program data followed by the end-of-string specified by the active gpibCmd entry, and stores the entire response in pgpibDpvt->msg. This routine is intended for use by custom input functions to read a block of arbitrary data from a serial-line device.

`setEos`
: If the End of String terminator needs to be changed, it must be changed by calling this rather than calling the low level driver.

`restoreEos`
: If code calls setEos it must call restoreEos after all reads are done.

`completeProcess`
: If the port can block, callbackRequestProcessCallback is called. If the port can not block then PACT is set false. Thus before calling queueRequest, the caller should set PACT true.

`pdevSupportGpib`
: An external variable which points to devSupportGpib.

Changes to Previous EPICS GPIB Support

conversionNotes.html describes how to convert GPIB device support modules written for the devGpib support that came with EPICS base 3.13 and earlier. The information also applys to support modules written for Benjamin Franksen's GPIB support or the gpibCore support since both of these evolved from the EPICS base GPIB support.

General GPIB Problems

NOTE: The following comments are from John Winan's original GPIB documentation.

Every type of communication system has its problems. Some instrument vendors don't properly test the GPIB interfaces on their products. Some devices miss messages or commands that are too close together in time. There are handshaking lines that are supposed to throttle the speed, but are apparently improperly implemented by device vendors, or make the (wrong) assumption that the controller in charge is slow in its ability to burst bytes down the bus. The only way that this problem can be worked around is to add delays in the GPIB device device support modules. The current device support library does not provide any means to do this.

Very often, a device will slow down over 800% when a user presses a button on the front panel of the device. This can cause the GPIB message transfer to time out, alarms to be set, and so on. When devices of this type have to be used, operators will have to be instructed to "look, but don't touch."

Some devices like to go out to lunch once every hour, or day or so, and not respond to a command for up to about 5 seconds (the DG 535 has done this on more than one occasion.) This can be more frustrating that anything else. All that can be said about these types of things is BEWARE of machines that don't work as advertised. There is probably something wrong with it that won't surface until it is in use and controlling something very important.

Test, test, and test your devices after writing a new device support module. Many devices can run fine if doing only three or five transactions per second, but crank it up to 50 or more, and watch it go up in flames. Even if all the records in an EPICS database are scanned slowly, they can still get processed in bursts. EPICS can actually process over 20,000 records in one second if they are all ready to go at the same time. And if there are enough records tied to the same device, there is no telling how fast the device will be pushed.

License Agreement