EPICS
GPIB Device Support

John Winans

September 24, 1993

Chapter1: GPIB device support

This text describes how to write device support for GPIB devices. It is assumed that the reader is already familiar with the dialogue required to operate a GPIB instrument, EPICS, how to compile device support modules, and how to use makesdr to build an environment such that the new device support modules can be made available to database designers via DCT.

1.1 Purpose

A GPIB device support module is used to provide access to the operating parameters of a GPIB device. GPIB devices may be accessed via National Instruments 1014 cards or via Bitbus Universal Gateways.

1.2 Overview

GPIB devices typically have many parameters, each of which may be thought of in terms of the standard types of database records available in EPICS. It is the job of the device support module designer to decide how the mapping of these parameters will be made to the available record types. Once this mapping is complete, the device support module may be written.

The writing of the device support module consists primarily of the construction of a parameter table. This table is used to associate the database record types with the operating parameters of the GPIB instrument.

Other aspects of module design include the handling of SRQ events and errors. SRQ events are made available to the device support module if so desired. The processing of an SRQ event is completely up to the designer of the module. They may be ignored, tied to event based record processing, or anything else the designer wishes. Error conditions may be handled in a similar fashion.

1.3 Using a GPIB Device Support Module

To make use of a GPIB device support module, You must add the appropriate entries to your application tree's devSup.ascii file. To add entries for GPIB devices, you edit the file and add your entries to the bottom. They will look like this:

"ai" GPIB_IO "devAiModName" "module name/description (GPIB)"

"ai" BBGPIB_IO "devAiModName" "module name/description (BBGPIB)"

The first field "**ai**" specifies that the entry describes an analog input record.

The **GPIB_IO** or **BBGPIB_IO** field is used to specify the type of interface the device is attached to. The **GPIB_IO** interface is the National Instruments 1014 board and the **BBGPIB_IO** is the bitbus universal gateway (BUG.) This field becomes the record's link type field.

"**devAiModName**" represents the DSET name used to identify the entry point(s) to the GPIB device support module. This must be the exact same name specified in the device support module for the same record type, in this case, an analog input record.

The last field is the text field that will appear in the DCT choice menu that pops up when the user edits the DTYP field of a record.

Notice that the same DSET name appears for both the BUG and NI interfaces. This is because the GPIB library code uses the record's link type to differentiate between the two. Also note that

unless explicitly stated otherwise, only GPIB device support modules that are designed using the GPIB device support library can actually support both the NI and BUG interfaces!

When the modifications are complete to the devSup.ascii file and the proper **makesdr** incantation has been made, DCT will provide access to the new devices by allowing them to appear in the choice menu for the DTYP fields for the record types specified in the devSup.ascii file. It will also present a menu box for the output (or input) specification field to query the user for the proper GPIB (or BitBus) link number, which parameter table entry to use when performing I/O, the device address (the actual GPIB address), and in the case of bitbus, the BUG's node number.

For more information on the various ascii and header files, **makesdr**, and how DCT uses them see the document "The Application Developers Guide".

1.4 The GPIB Device Support Module

The basic GPIB device support module is constructed by copying the sample skeleton template and adding the parameters to the parameter table. The template consists of DSET entries, a parameter table, efast tables (optional), name tables (optional), a parm block, a debugging flag, a SRQ handler function (optional), and some custom conversion functions (optional.)

The fast and easy way to create a new GPIB device support module is to copy the template and change the DSET entry names to a new unique value, change the debug flag name, and replace the parameter table entries. Then compile it and you are finished. This assumes that you do not wish to support SRQ processing and do not use the enumerated commands. In those cases, you might have to perform some additional work described in the sections below.

1.4.1 Required DSET Table Entries

DSET tables (Device Support Entry Tables) are what EPICS uses to interact with a device support modules. There must be one for each *type* of record supported by a device support module. The format of a DSET table used in a GPIB device support module is defined the *Epics Application Developers Guide* except that it is followed by three extra pointers. These pointers are only used by the GPIB system. The first extra pointer points to the parm block for the module, the second to the work function associated with the record type the DSET was created for, and the third for the SRQ handling function.

A typical DSET table used in a GPIB device support module looks like this:

```
gDset devAiDevicenameGpib {
6,                              /* number of EPICS elements in the DSET table */
{report,                        /* pointer to report function */
init_dev_sup,                   /* pointer to general init function */
devGpibLib_initAi,              /* pointer to record-specific init function */
NULL,
devGpibLib_readAi,              /* pointer to record-specific I/O function */
NULL,
(DRVSUPFUN) &devSupParms,       /* pointer to GPIB parm block */
(DRVSUPFUN) &devGpibLib_aiGpibWork,     /* pointer to GPIB work function */
(DRVSUPFUN) &devGpibLib_aiGpibSrq}};    /* pointer to GPIB SRQ handler */
```

In general, the above example may be used for any and all analog input GPIB DSET structures. And the DSET tables provided in the generic GPIB device support module template should work unmodified in most cases. The data type gDset is defined in the devCommonGpib.h header file and

should be used by GPIB device support modules when defining DSET tables.

There will be one DSET module for each type of record supported by a GPIB device support module. Only one of them should include the pointers to the general init function and the report function. Other DSET entries should use NULL for those fields. The reason for this is because the init routine is called in each DSET and in the case of a multi-DSET module, it will be called unnecessarily. The report routine works the same way. It is currently called only when a user types the dbior command on an IOC console. And if more than one DSET points to the same report routine, you will see multiple copies of your report... Nothing fatal, just irritating.

All DSETs in the same support module will have the same pointer to the same parm block specified. This is required by the GPIB device support library.

Each DSET will have a different pointer for the work function and SRQ handler. There are record-type specific work functions and SRQ handlers available in the GPIB device support library that may be used for these if they are applicable.

Please see the document "The Epics Application Developers Guide" for more information about DSET tables.

1.4.2 The Parameter Table

This is where the translation to and from the language of the GPIB device is defined. The actual table contains one element for each parameter that is made available to the user via DCT (referred to by DCT as "parm" when editing the input (or output) specification field of a record.)

The format of the parameter table is as follows:

```
static struct gpibCmd gpibCmds[] =
{
    /* Parameter 0 */
    {<f1>, <f2>, <f3>, <f4>,<f5>, <f6>, <f7>, <f8>, <f9>, <f10>, <f11>, <f12>, <f13>},

    /* parameter 1 */
    {&DSET_BO, GPIBCMD, IB_Q_HIGH, "init", NULL, 0, 0, NULL, 0, 0, NULL, NULL, -1}
};
```

This example parameter table contains 2 parameters. They are numbered zero and one. Parameter zero is provided for reference. Parameter one is an actual parameter line that is used in the DC5009 frequency counter’s table. A formal description of the fields is as follows:

<f1>

Specifies the address of the DSET that describes the record type supported by the table entry. In English... what record type this parameter is to be used for. It is specified this way so that there is no actual enumeration of the supported record types. This is used to check a record during the database initialization phase of operation. The GPIB library compares this pointer’s value to that of the record’s DSET field to assure that the user specified parameter is valid for the record type being initialized.

This field must be assigned the address of a DSET.

<f2>

Specifies the type of GPIB I/O operation that is to be performed. The value of the <f2> field is verified to be valid for the record type specified by <f1> when the record is initialized that specifies the parameter. That is to say that only the parameter table entries that are used by a

database are checked. This protects the database designer from an error in the parameter table.

The <f2> field must be set to one of the following enumerated values declared in devCommonGpib.h:

GPIBREAD

1) Send the command string specified in <f4> to the instrument exactly as specified.
2) Data is read from the instrument and placed into the **dpvt.msg** field.
3) If <f8> is NULL, the data from the read operation is scanned via sscanf() as follows:

sscanf(dpvt.msg, <f5>, &(precord->val));

Otherwise, a call is made to the function pointed to by <f8> as follows:

(*(<f8>))(&dpvt, <f9>, <f10>, <f11>);

This has the effect of reading data from the instrument and parsing the desired information out of it. In the case where <f8> is NULL, the value is determined by the sscanf function. These are generally useful, but tricky. Keep in mind the data type of the VAL field for the type of record you are trying to update. For example the Analog Input record type has a double precision floating point data type. So a %lf (percent ell eff) is required, not a %f. For the case when <f8> is non-NULL, see the discussion of <f8> below.

The **GPIBREAD** setting is only valid for input record types.

GPIBWRITE

1) If <f8> is NULL, a command string is built via sprintf() as follows:

sprintf(dpvt.msg, <f5>, precord->val);

Otherwise, a call is made to to the function pointed to by <f8> as follows:

(*(<f8>))(&dpvt, <f9>, <f10>, <f11>);

This allows the module to create a character string that includes the val field of a record. As in the case of the **GPIBREAD** operation, keep the specific data type of the VAL field in mind. An oddity of the sprintf() function that comes with vxWorks is that the %lf (percent ell eff) format command will generate nothing when the VAL field is zero. So you should use a length specifier of at least one when using floating point formatting. For example %.1f (percent dot one eff).

2) dpvt.msg is sent to the instrument.

3) If the responds-to-writes flag in not -1 in the parm block, data is read from the instrument and placed in the **dpvt.rsp** string. If the secondary conversion routine pointer is not NULL in the parm block, it is invoked as follows:

(*(parmblock.wrConversion))(read_status, pdpvt);

For more information on the secondary conversion routine, see "Machines That Respond to Everything."

For more information on the case when <f8> is non-NULL, see the discussion of <f8> below.

GPIBCMD

1) Send the command string specified in <f4> to the instrument exactly as specified.

2) If the responds-to-writes flag is not -1 in the parm block, data is read from the instrument and placed in the **dpvt.rsp** string. And if the secondary conversion routine pointer is not NULL in the parm block, it is invoked as follows:

(*(parmblock.wrConversion))(read_status, pdpvt);

For more information on the secondary conversion routine, see the section "Machines That Respond to Everything."

GPIBCTL

1) Assert the GPIB ATN line and send the command string specified in <f4> to the instrument exactly as specified. Then de-assert the ATN line.

GPIBSOFT

Does no I/O operations and simply calls the custom conversion routine directly. The return value from the conversion routine is passed back to the caller of the device support processing entry point. The custom conversion routine is called as follows:

(*<f8>)(&dpvt, <f9>, <f10>, <f11>);

When GPIBSOFT is specified, <f8> must be set to point to the processing function.

GPIBREADW

In order to use this, there must be an **srqHandler** defined in the parm block.

1) Send the command string specified in <f4> to the instrument exactly as specified.
2) Set **hwpvt.srqCallback** to the record-type specific SRQ handling function.
3) Wait for an SRQ interrupt from the device. (Will be caught by **parmBlock.srqHandler**)
4) The **srqHandler** should call **hwpvt.srqCallback** to deal with it when it arrives.
5) Data is read from the instrument and placed in **dpvt.msg**.
6) If <f8> is NULL, the data from the read operation is scanned via sscanf() as follows:

   sscanf(dpvt.msg, <f5>, &(precord->val));

   Otherwise, a call is made to the function pointed to by <f8> as follows:

   (*(<f8>))(&dpvt, <f9>, <f10>, <f11>);

This has the effect of reading data from the instrument and parsing the desired information out of it. In the case where <f8> is NULL, the value is determined by the sscanf function. These are generally useful, but tricky. Keep in mind the data types of the value fields of the types of records you are trying to update. For example the Analog Input record type has a double precision floating point data type. So a %lf (percent ell eff) is required, not a %f. For the case when <f8> is non-NULL, see the discussion of <f8> below.

GPIBEFASTO

This operation type is only valid on BO and MBBO record types.

1) The string pointed to by <f11>[VAL] is sent to the instrument.

2) If the responds to writes flag is not -1 in the parm block, data is read from the instrument and placed in the **dpvt.rsp** buffer. If the secondary conversion function pointer is not NULL in the parm block, it is invoked as follows:

(*(parmblock.wrConversion))(read_status, pdpvt);

For more information on the secondary conversion function, see the section "Machines That Respond to Everything." For more information of the use of <f11>, see the section "Efast Tables".

Note that setting <f8> to a non-NULL value is invalid for this operation type. Results in that case should be considered catastrophic.

GPIBEFASTI

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

1) Send the command string specified in <f4> to the instrument exactly as specified.
2) Read the data from the instrument and place it into the **dpvt.msg** buffer.
3) Compare **dpvt.msg** against each element of the Efast Table.
4) Set the VAL field to the entry number of the first Efast Table element that matched.

Note that setting <f8> to a non-NULL value is invalid for this operation type. Results in that case should be considered catastrophic.

GPIBEFASTIW

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

In order to use this, there must be an **srqHandler** defined in the parm block.

1) Send the command string specified in <f4> to the instrument exactly as specified.
2) Set **hwpvt.srqCallback** to the record-type specific SRQ handling function.
3) Wait for an SRQ interrupt from the device. (Will be caught by **parmBlock.srqHandler**)
4) The **srqHandler** should call **hwpvt.srqCallback** to deal with it when it arrives.
5) Data is read from the instrument and placed in dpvt.msg.
6) Compare **dpvt.msg** against each element of the Efast Table.
7) Set the VAL field to the entry number of the first Efast Table element that matched.

Note that setting <f8> to a non-NULL value is invalid for this operation type. Results in that case should be considered catastrophic.

<f3>

Specifies the processing priority of the I/O operation when being executed by the actual GPIB driver. This field must be set to either **IB_Q_HIGH** or **IB_Q_LOW.** These values are enumerated in drvGpibInterface.h.

<f4>

Specifies a constant string that is used differently depending on the value of <f2>. See the discussion of <f2> for more information. Set this field to NULL if not used.

<f5>

Specifies a constant string that is used differently depending on the value of <f2>. See the discussion of <f2> for more information. Set this field to NULL if not used.

<f6>

This is used to specify the length of the buffer that is pointed to by **dpvt.rsp**. It is used to hold the message that is read back from a device when performing a responds-to-writes read operation.

Set this field to zero when not used.

See the section "Machines That Respond to Everything" for more information.

<f7>

This is used to specify the length of the buffer that is pointed to by **dpvt.msg**. The buffer is only used when <f2> is set to **GPIBWRITE, GPIBREAD,** or **GPIBREADW**. Set this field to zero when not used.

<f8>

Points to a function that is called to perform the entire I/O operation when <f2> is set to **GPIBSOFT**, or to perform a conversion/parsing operation when <f2> is set to any of the other operation types.

Note that this function is intended to be used to generate and parse strings being sent to and from an instrument, but can be used for anything. Great care should be taken in these functions so as to not overflow the **dpvt.msg** field when it is being used. This function is passed <f9>, <f10>, and <f11> as parameters.

For output type operations, this custom conversion function is called to generate the string (possibly using the records VAL field) that is to be sent to the instrument. For input type operations, it is called to scan the response string from the instrument (and fill in the records VAL field.) It is highly recommended that if a custom conversion routine be used, that the designer of the function be familiar with the record-specific library function that calls it.

The function should be declared as returning an int. This return value is passed back to the caller of the device support module after a processing request when <f2> is set to **GPIBSOFT** and is ignored in other cases.

This field must be set to NULL when no conversion function is present.

<f9>

This is an integer that is passed to any conversion function specified in <f8>. It may be used for any purpose the designer wishes.

<f10>

This is an integer that is passed to any conversion function specified in <f8>. It may be used for any purpose the designer wishes.

<f11>

This field plays a double role. When the parameter table entry has a custom conversion routine, it is passed to the conversion routine specified in <f8>. When <f2> is one of the **EFAST** operations, this field points to the EFAST table. See the EFAST operation descriptions under the <f2> field definitions and the section "Efast Tables" for more on the use of this field.

Set this field to NULL when it is not used.

<f12>

This field points to the Name Table. Name tables are described in the section "Name Tables".

Set this to NULL when no Name Table is used.

<f13>

This field is currently unused and must be set to -1. It is planned that it will be used in the future as part of the initialization process.

1.4.3 Efast (Enumerated Fast I/O) Tables

There are many times when a device's command set includes things like "TERM LO" and "TERM HI" to set the impedance. Or something with an "OFF" or "ON" in the command string. These also just so happen to be the types of commands that are tied to binary and multibit binary record types. And in order to generate a "TERM LO" from a BO record when the value is zero, and a "TERM HI" then the value is one, the sprintf style formatting provided for **GPIBWRITE** operations (see description of **<f2>** above), you have to use a custom conversion routine. And a custom function for every lousy binary and multibit binary supported function is outrageous.

The efast tables are used to get around the formatting problem of these types of situations by removing the formatting altogether. The device support module designer simply types in each of the command or expected response strings for each of the possible states of the **VAL** field of the record.

The format of an efast table is:

```
static char     *(tableName[]) = {
                                    "TERM LO",    /* when VAL = 0 */
                                    "TERM HI",    /* when VAL = 1 */
                                    NULL};
```

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

```
{&DSET_BO, GPIBEFASTO, IB_Q_HIGH, NULL, NULL, 0, 0,
NULL, 0, 0, tableName, NULL, -1},
```

For an input entry, it would look like this:

```
{&DSET_BI, GPIBEFASTI, IB_Q_HIGH, NULL, NULL, 0, 50,
NULL, 0, 0, tableName, NULL, -1},
```

The efast table MUST be null terminated when used for input record types. It is not required for output records, but is a good idea anyway so that they all look the same. Otherwise you might forget one on used for an input operation and spend all day looking for the problem.

The way the the table is used for outputs is that the **VAL** field is used to index into the efast table and select which string to send to the instrument. The string is then sent to the instrument as it appears in the efast table with no formatting.

For input operations, the **<f4>** string is sent to the instrument without formatting, and then the response string is read from the instrument. This response string 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 NULL 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, it 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[]) = {
                                    "OF",         /* set RVAL to 0 */
                                    "ON",         /* set RVAL to 1 */
                                    NULL};
```

Note again that the NULL field is important here. If the instrument gets confused and responds with something that does not start with an "OF" 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 VALID alarm state.

1.4.4 Name Tables

For binary and multibit binary records, an Application Developer must type in all the name fields for each of the possible values the VAL field can be set to. This can be a nuisance if the user has many devices of the same type to make records for. Name tables can be added to binary and multibit binary records that can be used to fill in these name fields for the database designer so they may be left blank when viewed from DCT.

Before continuing, it should be understood that these name tables have absolutely NOTHING to

do with the operation of the GPIB device support library with respect to the way any I/O operations are performed.

To use a name table, the address of the table must be put into <f12> 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  struct  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  struct  devGpibNames   disableEnable = {
                    2,                          /* number of elements */
                    disableEnableList,          /* pointer to strings */
                     NULL,                      /* pointer to value list */
                    1};                         /* number of valid bits */
```

The **devGpibNames** structure is defined in the **devCommonGpib.h** header file. The first thing you need is the list of name strings. This is done by the declaration of an array of pointers to strings. For binary record types, the strings are placed into the name fields in order from lowest to highest as shown above. For multibit binary records, there can be up to sixteen strings defined.

After the table of strings is defined, you define a **devGpibNames** structure that includes the number of strings/name fields to fill in, a pointer to the table of strings, a pointer to the table of values, and the number of bits field (a multi-bit record's **NOBT** field.)

The value list pointer, and NOBT field 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 name table 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 fills them in via DCT.

1.4.5 The Parm Block

The parm block is used as the interface between your module and the GPIB device support library. It contains pointers to your debug flag, parameter table, SRQ handler, secondary conversion routine, and values for time-outs and other parameters. These items are accessed by the library by way of your module's DSET(s).

A sample parm block looks like this:

```
struct  devGpibParmBlock devSupParms = {
  &Dc5009Debug,                 /* debugging flag pointer */
  -1,                           /* set to -1 if device does not respond to writes */
  300,                          /* # of clock ticks to skip after a device times out */
  NULL,                         /* hwpvt list head */
  gpibCmds,                     /* GPIB command array (parameter table) */
  NUMPARAMS,                    /* number of supported parameters */
  -1,                           /* magic SRQ param number (-1 if none) */
  "devXxDc5009Gpib",            /* device support module type name */
  DMA_TIME,                     /* # of clock ticks to wait for DMA completions */
   NULL,                        /* SRQ handler function (NULL if none) */
   NULL                         /* secondary conversion routine (NULL if none) */
};
```

The debugging flag pointer must point to a integer that is set to a non-zero value if you want the GPIB device support library to provide debugging output for you.

The responding to writes flag is a kluge that is used to indicate that all output-type commands (see the section "The parameter table") to this device type will solicit a response from the device. See the section on "Machines that Respond to Everything" for more information about this.

The next field represents the amount of time (in 60ths of a second) that the GPIB system will wait after a device times-out, before trying to contact it again. During this time window, any I/O operations that are directed at 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.) For more about this and other exceptional conditions, see the sections on "SRQ Functions" and "General GPIB Problems."

The hwpvt list head is the head pointer to a singly linked list of structures that are called hardware private blocks. There are one of these hwpvt blocks allocated for each instance of a device type supported by this GPIB device support module. They contain information needed by the GPIB device support library that describes the current state of each device. This includes the time the last time-out happened, total number of time-outs processed by the library (this will not include time-outs that happen in result to I/O operations initiated by the interactive GPIB debugging tool), a user private pointer that may be used by a device support module designer for any reason (it is not referenced by any of the library code), and some information about SRQ interrupt processing (see the section "SRQ Functions" for more on these fields.) The hwpvt structures are built and maintained by the GPIB device support library and unless additional information is needed on a per-device instance basis, you may ignore their existence entirely. The proper initialization of the

hwpvt field is as shown above, NULL. If you should decide that you want to use the user private pointer, you should read the section "Talking to Machines that Don't Fit Into the Required Model."

The command array pointer is a pointer to the parameter table.

The number of supported parameters is next and represents the number of entries in the parameter table. The standard template uses a simple #define to calculate this value. If you use it, you need not concern yourself with it.

The magic SRQ param number is only used if you have an SRQ handler specified. See "The SRQ Handler" and "SRQ Functions" for more information on this.

The device support module type name field is used by the GPIB support library code when it prints debugging information. The string declared here is prepended to any debugging text printed by the library. If you do not wish to have anything printed, you must specify a null string, not the NULL pointer.

The time to wait for DMA completions is passed on to the driver and used to determine if a machine times out on a transaction or not. If the transfer of the data portion of a GPIB message is not complete within the time specified by this field, the transaction is considered timed out, and an appropriate VALID_ALARM is raised for the record being processed. The value specified for this field must be in 60ths of a second.

The pointer to the SRQ handler function should point to a function with the following prototype format: **static int srqHandler(struct hwpvt *phwpvt; int srqStatus;)** It will be called when ever an SRQ is detected from one of the devices supported by the GPIB support module. If SRQs are to be ignored for the supported device type, this must be set to NULL. See the section "The SRQ Handler" for more information.

The secondary conversion routine is used in cases where the responds to writes field is not set to -1. If a machine responds to writes, this field can be used to specify a function to call after the response is read from the device. It is offered here to allow the support module to inspect the response. If no response checking is to be done, this field must be set to NULL. Please see the section on "Machines that Respond to Everything" for more information about this.

1.4.6 The SRQ Handler

This section describes the general operation of the SRQ handling function that may be defined in the parm block for a specific device type.

The overall purpose of an SRQ handler is to determine if a given SRQ is expected, and if so call the required function(s) required to handle it. In the cases where an SRQ is not expected, it is up to the designer of the module designer to decide how to handle them.

The SRQ handler is provided a pointer to the **hwpvt** structure as well as the byte value returned from the serial poll made to the device. Since the device driver has no way of knowing what record (if any) that the SRQ is to be associated with, the SRQ handler has to figure it out. To make things a little easier, the GPIB library code stores the address of the **dpvt** structure as well as the record type specific SRQ processing function into the **hwpvt** structure for any record that is being processed that is expecting an SRQ. The library also informs the driver that no transactions are to be made to the device while waiting for the SRQ by returning a **BUSY** status to the driver when entering the wait state for the SRQ. This assures that the **dpvt** and function pointers in the **hwpvt** structure are valid when the expected SRQ arrives.

See the skeleton GPIB device support module for an example of handling solicited and unsolicited SRQs.

Solicited SRQ Handling

Solicited SRQs are generated by specific commands that are specified in the parameter table. The only types of parameter table entries that are allowed to solicit SRQs are **GPIBREADW** and **GPIBEFASTIW**. In these cases, it is expected that the command string is expected to solicit an SRQ that indicates that an operation has completed.

If an SRQ is expected, the **dpvt** and record specific processing function addresses will be waiting in the hwpvt structure as outlined above. So to handle this type of SRQ, check to see if one was expected (the function pointer being non-NULL) and then invoke the handling function.

1.4.6.1 Unsolicited SRQ Handling

A sane way of dealing with unsolicited SRQs has yet to be determined. The only information available when these occur, is the device address and the poll status. It is completely up to the device support module designer as to what should be done in these cases. At the this time, the only recommended action is to try to associate the SRQs with the processing of a record that is I/O-event scanned.

New work is being done in the area of dealing with event scanned records as this document is being written. So no example code is currently available. However, the following discussion should provide enough information for a developer to create an I/O-event scanned record that can be processed when unsolicited SRQs are recognized by the **srqHandler** function.

When it has been determined that an SRQ does not represent a solicited operation complete, a record may be processed by the use of a **callbackRequest**() or a **scanIoRequest**() function. These can process a record provided that the device support module can remember which one(s) is(are) to be processed.

Currently, the magic SRQ param number specified in the parm block is recognized by the GPIB library code as a parameter that records can specify that require processing when unsolicited SRQs are recognized. The library will only allow one record for each device to be defined that specifies the magic number (in contrast to one per device type.) When these records are initialized, the address of their **dpvt** structures are saved in the **hwpvt** structures associated with the physical devices. So when an unsolicited SRQ comes along, processing the right record is fairly straight forward. Make a **callbackRequest** to the record processing entry point for the record represented by the saved **dpvt** address. (see the skeleton GPIB device support's sample **srqHandler** function for an example of this.)

In the future, it is expected that all record processing for unsolicited SRQs will be done by using the **scanIoRequest** function.

1.4.7 Debugging Flags

The device support library provides some debugging output for exceptional conditions during normal processing if the debug flag in the parm block is set to a non-zero value. It will also prefix its debug statements with the module type name string also defined in the parm block.

1.4.8 Talking to Machines That Don't Fit Into The Required Model

Machines that don't really fit into the simple transaction model defined by the GPIB device support library will require either the addition of some custom code to perform the information conversions, or sometimes a total replacement of some or all of the library code. The more library code that is removed, the more difficult it will be to keep up with changes in EPICS and the GPIB driver(s). However, there is nothing that prevents a developer from doing so when required.

It will be a good idea to start your code design copying the part(s) of the GPIB device support library into your module that require replacement. This way you will have an example of an operational version of your code with all the correct parameters and data types defined on the calls to the actual GPIB driver.

1.4.8.1 Custom Conversion Routines

When a device's response can not be parsed with a simple sscanf or whose commands can not be formatted with an sprintf. A customized conversion routine will be required. These are supplied in the form as a function with a prototype of:

static int specialConvert(struct gpibDpvt *pdpvt; int p1; int p2; char **p3;)

and are specified in the parameter table. See the discussion of the <f8> field above for more information about the parameter table entry.

Given the address of the **dpvt** structure as well as the developer-entered values for **p1**, **p2**, and **p3** (that come from the parameter table), the custom conversion function should have all the information required to perform the needed conversion(s).

For some examples of various conversion routines, see the Dg535 device support module.

1.4.8.2 Machines that Respond to Everything

One of the basic problems with the (current) implementation of the GPIB device support library code, is that there is no way to perform read operations that consist of the formatting of the command sent to the instrument and the parsing of the response returned from it. In general this has not been much of a problem since most vendors of GPIB devices use strictly defined commands that solicit responses from their devices. However, some devices send back replies to everything.

The GPIB device support library currently supports devices like these by providing a flag in the parm block that can be set to a non-negative value indicating that the library should read data from the device on every type of operation defined in the parameter table.

If the responds to writes flag in the parm block is not set to -1, the library will wait for a period of time specified by this flag and then perform a read operation from the device. The data from the read operation will be placed into **dpvt.rsp**. And then a call will be made to the secondary conversion routine (if not specified as NULL in the parm block) and it will be passed the status from the read operation and the address of the dpvt structure. The prototype of a secondary conversion routine function is:

static int secondaryConversion(int status; struct gpibDpvt *pdpvt;)

Where **status** is the number of bytes read from the device or -1 if the read operation failed. And **pdpvt** is the address of the dpvt structure associated with the record being processed.

The return value from the secondary conversion function must either be **OK** or **ERROR**. If **ERROR** is returned the record will be placed into a **VALID_ALARM** state. Otherwise, the processing of the I/O operation will be completed as normal.

In the future, modifications the the GPIB library will be made to correct this problem and secondary conversion routines will no longer be required or supported.

1.5 The GPIB Device Support Library

There is a library of commonly used functions available to the GPIB module designer. It contains enough code such that a device support module can be written that contains as little as two lines of

executable C code.

All GPIB device support library functions have names that are prefixed with "devGpibLib_" and include the type of record they apply to. For example, the devGpibLib_initAi() function is used to initialize analog input records. And the devGpibLib_readAi() function is used to fill in the VAL field on an analog input record.

1.5.1 Initialization Functions

The initialization functions are used to verify the validity of information in the device support module as well as the information in a database record before any database record processing begins.

1.5.1.1 General Initialization

long devGpibLib_initDevSup(int parm; gDset *dset;)

Call with parm=0 before any calls are made to the record-type specific init functions, and again with parm != 0 after all calls have been made to record-type specific init functions. The DSET value must point to any one of the DSET data structures for the GPIB device type that is being initialized.

This function does nothing more than print an initialization time message. It might be used in the future to initialize the value fields of output record types.

1.5.1.2 Record Specific Initialization

The record specific initialization functions are used to allocate and initialize any data structures needed by the library. For all record types, the operations start the same way:

1) Allocate a **dpvt** structure and connect it to the record.
2) Verify that the **DTYPE** field is either **GPIB_IO** or **BBGPIB_IO**.
3) Verify that the parameter number is within valid range.
4) Allocate a **hwpvt** structure if necessary.
5) Inform the GPIB driver of intended use of the link specified in the record.
6) Verify the GPIB device address is within valid range.
7) Verify that the parameter table entry's <f1> points to the same **DSET** as the record's.
8) Register the SRQ handler with the driver if one specified in the parm block.

And then for each record type, operations complete as follows:

long devGpibLib_initAi(struct aiRecord *pai; void (*process)();)

9) Verify that the parameter table entry operation is **GPIBREAD**, **GPIBSOFT**, or **GPIBREADW**.
10) Return a 0 to the calling function.

long devGpibLib_initBi(struct biRecord *pbi; void (*process)();)

9) Verify that the parameter table entry operation is **GPIBREAD**, **GPIBSOFT**, **GPIBEFASTI**, **GPIBE-FASTIW**, or **GPIBREADW**.
10) Return a 0 to the calling function.

long devGpibLib_initLi(struct longinRecord *pli; void (*process)();)

9) Verify that the parameter table entry operation is **GPIBREAD**, **GPIBSOFT**, or **GPIBREADW**.
10) Return a 0 to the calling function.

long devGpibLib_initMbbi(struct mbbiRecord *pmbbi; void (*process)();)

9) Verify that the parameter table entry operation is **GPIBREAD**, **GPIBSOFT**, **GPIBEFASTI**, **GPIBEFASTIW**, or **GPIBREADW**.

10) Return a 0 to the calling function.

long devGpibLib_initSi(struct stringinRecord *psi; void (*process)();)

9) Verify that the parameter table entry operation is **GPIBREAD**, **GPIBSOFT**, or **GPIBREADW**.

10) Return a 0 to the calling function.

long devGpibLib_initAo(struct aoRecord *pao; void (*process)();)

9) Verify that the parameter table entry operation is **GPIBWRITE**, **GPIBSOFT**, **GPIBCMD**, or **GPIBCNTL**.

10) Return a 2 to the calling function.

long devGpibLib_initBo(struct boRecord *pbo; void (*process)();)

9) Verify that the parameter table entry operation is **GPIBWRITE**, **GPIBSOFT**, **GPIBCMD**, **GPIBEFASTO**, or **GPIBCNTL**.

10) Return a 0 to the calling function.

long devGpibLib_initLo(struct longoutRecord *plo; void(*process)();)

9) Verify that the parameter table entry operation is **GPIBWRITE**, **GPIBSOFT**, **GPIBCMD**, or **GPIBCNTL**.

10) Return a 0 to the calling function.

long devGpibLib_initMbbo(struct mbboRecord *pmbbo; void (*process)();)

9) Verify that the parameter table entry operation is **GPIBWRITE**, **GPIBSOFT**, **GPIBCMD**, **GPIBEFASTO**, or **GPIBCNTL**.

10) Return a 0 to the calling function.

long devGpibLib_initSo(struct stringinRecord *psi; void (*process)();)

9) Verify that the parameter table entry operation is **GPIBWRITE**, **GPIBSOFT**, **GPIBCMD**, or **GPIBCNTL**.

10) Return a 0 to the calling function.

1.5.2 Report and Status Functions

long devGpibLibReport(gDset *dset;)

Print a one-liner report of the device name, its addressing information, link type and the total number of observed time-outs. This function is provided so that the **dbior** function can be used to coarsely observe the operation of a device.

1.5.3 Transaction Request Functions

The transaction functions are used to perform the I/O operation associated with an initialized database record. In general, they are called as a result of **dbProcess**() processing a record.

1.5.3.1 Input Functions

This section describes the library functions that can be used to process a record that requires data to be solicited from a GPIB device. If the record being processed specifies a **GPIBSOFT** parameter, the soft-processing takes place immediately with no asynchronous return to the caller. In all other cases, the function queues a request to the GPIB driver and returns asynchronously.

If the queue request to the driver fails, the record is placed in a **VALID_ALARM** state.

The return value from the second call in the asynchronous processing for each function is 2 for each of the following functions except for the MBBI and BI versions... then the return value is 0.

long devGpibLib_readAi(struct aiRecord *pai;)

long devGpibLib_readBi(struct biRecord *pbi;)

long devGpibLib_readLi(struct longinRecord *pli;)

long devGpibLib_readMbbi(struct mbbiRecord *pmbbi;)

long devGpibLib_readSi(struct stringinRecord *psi;)

1.5.3.2 Output Functions

This section describes the library functions that can be used to process a record that requires data to be sent to a GPIB device. If the record being processed specifies a **GPIBSOFT** parameter, the soft-processing takes place immediately with no asynchronous return to the caller. In all other cases, the function queues a request to the GPIB driver and returns asynchronously.

If the queue request to the driver fails, the record is placed in a **VALID_ALARM** state.

The return value from the second call in the asynchronous processing is zero for each of the following functions.

long devGpibLib_writeAo(struct aoRecord *pao;)

long devGpibLib_writeBo(struct boRecord *pbo;)

long devGpibLib_writeLo(struct longoutRecord *plo;)

long devGpibLib_writeMbbo(struct mbboRecord *pmbbo;)

long devGpibLib_writeSo(struct stringoutRecord *pso;)

1.5.4 Transaction Processing Functions

These are the functions that perform the actual I/O operations by making calls to the GPIB driver. These functions are called by the driver's link task then the driver has determined that a transaction request is ready for processing.

1.5.4.1 Work Functions

These functions are divided into two categories, an input group and an output group.

The output group are fairly simple. They format a message as specified in the parameter table and then call the driver to send it. After the message is sent, a **callbackRequest** is made to **dbProcess()** so that the second half of the asynchronous processing may take place. If the output operation fails, the record is placed into a **VALID_ALARM** state before the callback to **dbProcess()** is made.

int devGpibLib_aoGpibWork(struct gpibDpvt *pdpvt;)

int devGpibLib_boGpibWork(struct gpibDpvt *pdpvt;)

int devGpibLib_loGpibWork(struct gpibDpvt *pdpvt;)

int devGpibLib_mbboGpibWork(struct gpibDpvt *pdpvt;)

int devGpibLib_stringoutGpibWork(struct gpibDpvt *pdpvt;)

The input group are a little more complex because a message is not only sent to a device, but a response is read back afterward. If the parameter table entry specifies that it is to be treated as an operation that includes an SRQ to indicate completion, these functions return to the driver before reading the response message back. In the SRQ case, the driver will end up calling the **srqHandler** function defined in the parm block when it arrives. The **srqHandler** is responsible for then calling the record specific SRQ handling function described in the section "SRQ Functions" below. This process is described in the section "The SRQ Handler" above.

In the non-SRQ based style of operation, the message specified in the parameter table is sent to the device, the response read back, the response converted to the required VAL field data type as specified in the parameter table, and a **callbackRequest**() is made to **dbProcess** to initiate the second half of the asynchronous record processing.

If any errors are encountered, the record is placed in a VALID_ALARM state before the **callbackRequest**() is made to **dbProcess**().

int devGpibLib_aiGpibWork(struct gpibDpvt *pdpvt;)

int devGpibLib_biGpibWork(struct gpibDpvt *pdpvt;)

int devGpibLib_liGpibWork(struct gpibDpvt *pdpvt;)

int devGpibLib_mbbiGpibWork(struct gpibDpvt *pdpvt;)

int devGpibLib_stringinGpibWork(struct gpibDpvt *pdpvt;)

1.5.4.2 Finish Functions

The finish functions are used to perform the conversion of the data read back from a work function that processes an input operation. These functions are only used internally by the library's work and SRQ processing functions and are presented here only for the sake of completeness. They are currently not useful except to other functions within the library.

int devGpibLib_aiGpibFinish(struct gpibDpvt *pdpvt;)

int devGpibLib_biGpibFinish(struct gpibDpvt *pdpvt;)

int devGpibLib_liGpibFinish(struct gpibDpvt *pdpvt;)

int devGpibLib_mbbiGpibFinish(struct gpibDpvt *pdpvt;)

int devGpibLib_stringinGpibFinish(struct gpibDpvt *pdpvt;)

1.5.4.3 SRQ Functions

These are used to perform the reading of a message from a device that has already been solicited by one of the input type work functions described above. They are called by the **srqHandler** functions in the GPIB device support modules. They simply perform the read operation and call the finish function associated with the type of record being processed.

For more information on handling SRQs see the section "The SRQ Handler."

int devGpibLib_aiGpibSrq(struct gpibDpvt *pdpvt; int srqStatus;)

int devGpibLib_biGpibSrq(struct gpibDpvt *pdpvt; int srqStatus;)

int devGpibLib_liGpibSrq(struct gpibDpvt *pdpvt; int srqStatus;)

int devGpibLib_mbbiGpibSrq(struct gpibDpvt *pdpvt; int srqStatus;)

int devGpibLib_stringinGpibSrq(struct gpibDpvt *pdpvt; int srqStatus;)

1.6 General GPIB Problems

Every type of communication system has its problems. The problems with the use of GPIB instruments seem to exist in the fact that vendors simply don't test their GPIB interfaces on their products.

More than one device that misses messages or commands that are given one after the other because they are too close together in time has been identified during the testing of the GPIB support library. 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 5 seconds or so (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 actually 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 down 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.