LanGpib Driver Support for HP E2050A - Release 2.4

Benjamin Franksen
BESSY GmbH
Albert-Einstein-Straße 15
12489 Berlin
Germany

Table of Contents

1 Introduction

Scope of this document

This document describes the EPICS driver support for the LAN/GPIB gateway HP E2050A. It has been completely revised for release 2.4.

The goal of this document is

- to give users a simple introduction and a guide for installation and usage,

- to give device support developers a detailed interface description,
- to explain design concepts and structure to anyone interested.

The purpose of the LanGpib driver support module is to provide a set of functions to allow other EPICS modules (especially device support modules) to interface GPIB-connected devices.

GPIP

GPIB (General Purpose Interface Bus, also called HP-IB, IEC bus or IEEE-488) is a parallel bus which can connect up to 16 devices. It is often used for connecting peripheral test and measurement devices to a control computer. The bus can handle an arbitrary amount of data (byte streams) and uses 3 hand-shake lines to adapt to the different timing capabilities of the individual devices. Every device on the bus has a unique number between (including) 0 and 30.

At one moment exactly one of the devices acts as the bus controller, sending out commands and data to the other devices (there are some commands that are sent to all devices), or receiving data from them. The controller is the only device on the bus that can initiate data transfer (with the exception of SRQs (Service ReQuests)).

More detailed information on GPIB can be found in almost every manual for GPIB devices such as - for example - the ‘Keithley, Model 617 Programmable Electrometer, Instruction Manual’. The standard reference is ‘488.1-1987 (R1994) IEEE Standard Digital Interface for Programmable Instrumentation (ANSI)’.

HP E2050A

The Hewlett-Packard HP E2050A is a gateway between a local area network (LAN) and a GPIB network. On the GPIB side it acts as the bus controller, whereas on the LAN side it acts as an RPC (Remote Procedure Call) server. Besides its GPIB connector it has an interface to RS232. On the LAN side it has a BNC and a 10 Base-T connection. It needs (and is shipped with) an external power supply.

VXI-11

The software interface to the control computer is called VXI-11, a standard provided by the VXIbus Consortium. It is based on Sun’s RPC protocol which in turn relies on TCP/IP as the network/transport layer. See ‘VXI-11: TCP/IP Instrument Protocol and Interface Mapping Specifications’ and ‘How To Use VXI-11 and HP E2050A for Programming GPIB Devices’ for more information on VXI-11.

EPICS

EPICS demands that a driver support module exports a so called Driver Support Entry Table, a structure that contains at least

- the number of entries,
- a report function, and
- an init function,

but possibly more functions. Entry points to the driver should be limited to this table and all functions should be declared as ‘static’ in order to hide the names from other modules in the system. There is one exception to this rule: if a function is intended to be called from the command line either as a debugging tool or as a configuration tool (to be called before iocInit) then it is necessary that this function is not static. A similar reasoning applies to global variables.

EPICS device support for specific record types demand further that each device belongs to a certain class (the hardware-link type). This class determines the type and amount of information the person who configures a database has to specify in the hardware-links that symbolically ‘connect’ a record with the hardware. A method has to be determined, how a specific device on a specific gateway/bus is to be addressed (from EPICS).

VxWorks

VxWorks is currently the only real-time operating system that the EPICS real-time core supports. It is a preemptive multitasking system , which implies that concurrent access to a driver must be viewed as normal. Any function that is called during normal processing (that is: after initialization) must be reentrant. Also concurrent accesses to the same device (that is: the same gateway instance) must be sequentialized.

Since the software interface of the HP E2050A is realized by building on Sun's ONC/RPC protocol, it is necessary to observe the limitations of the specific RPC implementation under VxWorks. One important limitation is that VxWorks allows only one task to operate on a certain RPC channel. Therefore (and also for other reasons) access to the driver must be queued and only one dedicated task per gateway may call RPCs directly.

General Requirements

Based on and according to the restrictions stated so far, the driver is supposed to

- handle multiple gateways and the maximal number of allowed devices per gateway (16),
- reserve exclusive access to devices to block other IOCs from manipulating them, and
- serialize concurrent access from within a single IOC.

A Remark

This will probably be the last major revision of LanGpib, and won't be further supported appart from bug-fixes, of course. By now, it should work stable under most circumstances. A completely restructured version of the EPICS GPIB support in which LanGpib co-exists with other low-level drivers (for NI card, Bitbus, etc..) will appear in the not-too-distant future, if all goes well.

2 Design Concepts

2.1 History

I did not write driver support for GPIB from scratch because there was already a working driver that supported interfaces like the NI1014 VME board. The author of this driver (John Winans) also wrote a library, containing a lot of functions to make writing specific device support for new devices (or changing existing ones) much easier. I decided it was a good idea to keep as much of the concepts and structure and only change the functions that depended directly on the hardware. As it turned out this was possible but not to the extent I originally planned (partly because of the limitations mentioned in the Introduction).

What was finally kept was mainly the overall design structure, the functional interface to the device support and almost all of the library.

Many of the concepts which are discussed in greater detail in the next sections are not my own, but are part of the original driver. No attempt was made to clearly separate new concepts from old ones.

Terminology: 'link'

Also, for historical reasons the word 'link' is used for a number of only partially related things:

- EPICS hardware link, symbolic way to 'connect' a record to a specific hardware device.
- VXI-11 calls a connection from a network instrument client (such as one of our linkTasks, see below) to a (e.g. GPIB) device a link.
- John Winans calls the connection to a GPIB controller (such as one of our gateways) a link.

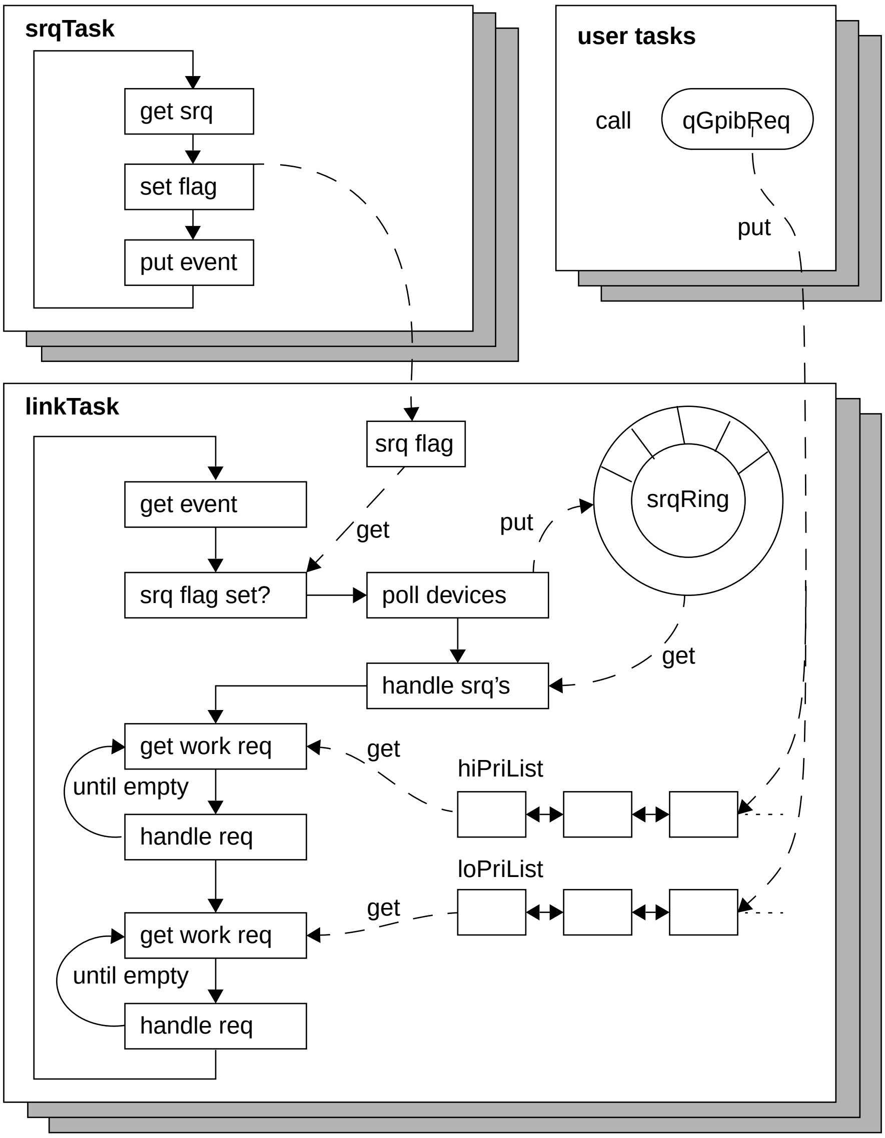
These definitions are in descending granularity: One VXI device link controls multiple EPICS hardware links, and one J. Winans link controls multiple VXI device links.

I apologize to the reader for this confusing terminology and hope that it is clear from the context which type of link is meant. In this document when the word 'link' or 'link number' is used without further specification, the J. Winans link is meant. VXI links are called 'device links'.

2.2 Task Structure

Every gateway controlled by the IOC under consideration has associated a structure hpLink and a two tasks. The driver maintains a list of the link structures in which all information belonging to a certain link (meaning GPIB controller connection) is stored.

Figure 1: Task Structure



linkTask

The first task is the linkTask and maintains two lists in which work requests from other tasks are queued, one for high and one for low priority requests. The linkTask also maintains a ring buffer for SRQ events. The working loop of the linkTask starts to process when some other task has given the link event semaphore (linkEventSem, also part of the link structure). The task first checks for SRQ events in the ring buffer, then for normal high priority requests and then for low priority requests.

srqTask

The other task, the srqTask, acts as RPC serv er for incoming SRQs. It e xecutes RPCs from the GPIB controller which come over the intr channel. The work done here is not very complicated: the call is checked for validity and if it is valid, the SRQ flag on the link is set and the linkTask is woken up by giving the link event semaphore.

Requests for reading and writing data to devices as well as for executing special commands are mapped onto the corresponding RPCs. SRQs are handled by polling all registered devices on the bus (reading the status byte in serial poll mode). If a device has the corresponding bit set, the information is stored in the ring buffer. After polling, the ring buffer is processed: if there is an SRQ handler function installed for this link and this device, it is called, otherwise only the device status of the device is cleared (so that it releases the SRQ line).

2.3 Initialization

Driver initialization is a bit complicated. The main reason for this is that VXI-11 demands that for every device on the bus a separate communication channel (a so called *device link*, see the notes on terminology) is to be created using a dedicated RPC. But this RPC can only be called by the linkTask that controls the particular bus segment in question. This is because the VxWorks implementation of ONC/RPC allows only one task to communicate on a given RPC channel (if another task tries to perform an RPC on the same RPC channel the result is a system crash), and there are exactly two RPC channels per GPIB gateway: one for SRQs (served by the srqTask) and one for everything else (served by the linkTask).

Also, at the time iocInit calls the driver init function, i.e. *before* record instances are initialized, it might be unknown how many LAN/GPIB gateways the IOC has to control and even more what devices are on the corresponding bus segments and what their GPIB addresses are. This is in contrast to drivers for VMEbus cards where the driver can simply look for responses from specific VMEbus addresses to detect how many cards of a certain type are present in the system.

The inititialization function that is called by iocInit for every driver does only the most basic things like initialization of global variables. It creates the mutex structure for the list of link structures and determines the IOC's IP address and network address.

The 'real' initialization takes place as soon as ioctl is called with parameter cmd == IBGENLINK. If this is the first call for the given link number, a lot of things happen. First the list of link structures is locked. Then the link structure is created, and the link task is started. The task calling ioctl blocks on the init semaphore for this link.

The link task will perform all further initialization which are mainly:

1. Reset the gateway by opening a telnet connection and sending 'reset<nl>y<nl>' (yes, it's ugly, but there is NO other way to do that).
2. Create the RPC channels, one for normal operation, the other (in backward direction) for SRQs.
3. Create a device link for every possible GPIB address.
4. Signal end-of-init by giving the init semaphore and start processing events as described in Section 2.2 on page 4.

On reception of the semaphore, the calling task is unblocked and the list of link structures is unlocked.

Thus, in the usual setting, i.e. the driver as a low-level module called by (record type dependent) device supports, the actual construction of a link to a GPIB device takes place during the device supports init_record method. It should be noted, though, that it is equally possible to call the driver from any other task in the system, e.g. another driver. In fact, this was the reason why the initialization procedure of older versions was changed to the one described here. A lot of the work that led from the old complicated solution to the current one, which is much simpler but also more flexible, was done by Till Straumann (then member of the PTB, Physikalisch Technische Bundesanstalt).

2.4 Hardware Link Type

The hardware link type was chosen as GPIB_IO mainly because this type already existed. A second reason was to make replacing the old driver (and interface device) with the new one as easy as possible. With this link type, the record's OUT resp. INP field has to be specified with a string of the form "#L<link> A<addr> @<other params>". The parameters <link> and <addr> are integral positive numbers that specify:

<link> the local part of the IP address of the gateway (e.g. in a class C network this is the number after the last point of the IP address),

<addr> the GPIB address of the device.

This assumes that the gateway and the controlling IOC are on the same local network, i.e. have the same network address. The remaining parameter <other params> is for use by device supports.

WARNING: The subnet mask (which can be configured in the IOC's NVRAM) is currently NOT used when the driver determines the gateway's IP address from the link number. It is therefore necessary to specify the *complete* local address inside the *network*, not the address inside the local *subnet*.

When record instances are loaded (dbLoadRecords, called before iocInit) this string is parsed and the numbers are assigned to a certain record field (the INP or OUT link). The value of a link with type GPIB_IO has the definition (see link.h):

```
struct gpibio {
        short link;
        short addr; /* device address */
        char *parm;
};
```

This implies that this driver can only be used with class C and class B networks because the link number is restricted to a short.[1]

2.5 Device Support

The driver support discussed in this document is independent from any special hardware that may be connected to the IOC via GPIB. It only transports raw data from and to such devices via the gateway. In order to make use of a specific device, it must be known which commands the device understands and in which format it answers. Unfortunately there is no common set of commands accepted by all GPIB devices. [Since IEEE-488.2 some common commands should be understood by most devices. But this is a very small set and not sufficient for even the most simple tasks.]

This implies that for every new device to be supported a new device support must be written. To simplify this task, John Winans wrote the GPIB Device Support Library[2]. It resides in the files

devCommonGpib.h Interface to the GPIB Device Support Library

devCommonGpib.c Implementation of the GPIB Device Support Library

There is an extensive documentation on how the library functions can be used to create device support for new devices, see

http://www-csr.bessy.de/asd/controls/epics/EpicsDocumentation/HardwareManuals/GPIB/gpib.960325.html.

[1] The decision to use GPIB_IO further implies, that no other GPIB driver can be used in parallel with this one on the same IOC.

[2] Not to be confused with the EPICS Device Support Library that supports registration of VMEbus card addresses and interrupt vectors.

Changes to the old version

Although in the beginning I planned to write the new driver in such a way that the GPIB Device Support Library could remain untouched, this turned out to be impossible. Anyway, the interface to device support modules is such that it should be possible to re-compile old device supports without changes to the source code.

Any device support module written *without* using the library (in the manner described in the above document) will only work with the new driver if at least the following change is done: In the init_record function of the device support there should be a call to drvGpib.ioctl, where the 4th parameter (command) has value IBGENLINK. The 5th parameter was previously unused and has probably a value like 0 or -1. This must be changed to the GPIB address of the device the record belongs to.

WARNING: I never personnaly tested the driver without the library. If you find any incompatibilities or other problems that should be noted here, please report them to me.[3]

The library exports a standard get_ioint_info function (the same for all record types) that is used for SRQ initiated processing, if the record's SCAN field is set to 'I/O Intr'. The corresponding IOSCANPVT is part of the hwpvt structure for the device.

There is also a simple srqHandler function in the library. Since it cannot generically analyze the status byte (so the latter is 'lost' to the program) that was received from the device, its use is limited. You may use it as a starting point for developing your own, device specific, handler. The antique method for I/O interrupt scanned records as described in J.W.'s manual, using a so called 'magic param number' is completely obsolete and should no longer be used. Any number of records (with different parm numbers) may be I/O interrupt scanned.

Generic header file

I also wrote a header file called

devGpib.h (Default DSETs and entry functions.)

to be included in device support modules. This further simplifies writing a new device support. If any of the macros

DSET_AI	/* ai */
DSET_AO	/* ao */
DSET_LI	/* longin */
DSET_LO	/* longout */
DSET_BI	/* bi */
DSET_BO	/* bo */
DSET_MBBI	/* mbbi */
DSET_MBBO	/* mbbo */
DSET_MBBID	/* mbbiDirect */
DSET_MBBOD	/* mbboDirect */
DSET_SI	/* stringin */
DSET_SO	/* stringout */
DSET_EV	/* event */
DSET_WF	/* waveform */

are defined before including devGpib.h, a standard DSET is generated automatically for the corresponding record types, as well as standard init and report functions for the DSETs which call the GPIB Device Support Library.

The file devNewSkeletonGpib.c is provided as an example device support module using the devGpib.h header file.

[3] That is, send an email to franksen@mail.bessy.de.

2.6 Module Structure

The following files belong to the driver support module:

Header files:

drvGpibInterface.h	Interface to the driver support module
drvLanGpib.h	VXI-11 specific constants
vxi11core.h	Interface to core/abort channel RPC protocol (generated but changed)
vxi11intr.h	Interface to intr channel RPC protocol (generated but changed)

Program files:

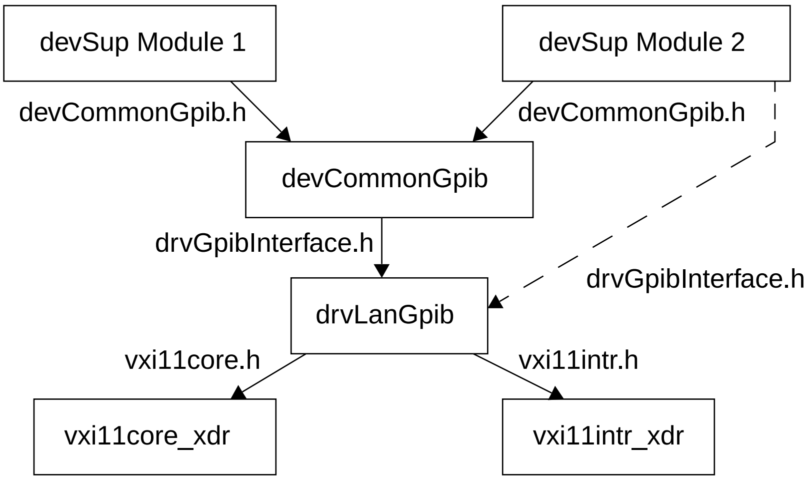
drvLanGpib.c	Driver implementation
vxi11core_xdr.c	XDR functions for core/abort channel (generated)
vxi11intr_xdr.c	XDR functions for intr channel (generated)

The interface to the driver is hardware independent in the sense, that it can be used not only for the HP E2050A but also for other hardware (like NI1014 VME cards). In fact, it is (almost) exactly the same as the one used for the older versions of the driver.

The files beginning with ‘vxi11’ implement the VXI-11 standard interface for LAN driven GPIB devices. Some of them are generated from protocol descriptions and some are manually changed to fit the special application. For details concerning VXI-11 and how the protocol is used see the document ‘VXI-11 and HP E2050A, A Programmer’s Guide’.

Figure 2 on page 8 shows the modular decomposition of device/driver support for GPIB records. An arrow means ‘calls/uses’; the text besides the arrows denotes the header file that specifies the interface. The dotted line from ‘devSup Module 2’ to ‘drvLanGpib’ means that a device support module may call the driver directly. For most devices this will not be necessary.

Figure 2: Module Structure



3 Interface Description

The (public) declarations and definitions discussed in this section can be used by including drvGpibInterface.h.

3.1 Structures

The most important data structure in the driver is hpLink. It contains all information about a single GPIB link. It is split into a public part (ibLink, declared in devGpibInterface.h) and a private hardware specific part (the actual hpLink). Originally, the ibLink part was defined public, and the separation was done to allow different hardware to be operated with a single driver (NI1014 and bitbus links and now LanGpib) while maintaining a common interface. At some time I decided that to make ibLink opaque. The fields are therefore no longer documented here.

ibLink

```
struct ibLink;
typedef struct ibLink ibLink;
```

dpvtGpibHead

Every device support that uses the gpib driver must supply the records' device private field with a pointer to a structure that begins with dpvtGpibHead. This structure is given as parameter to qGpibReq (see also next section).

```
typedef struct dpvtGpibHead {
        ELLNODE      list;
        CALLBACK     callback;
        int          (*workStart)();
        int          link;
        int          device;
        ibLink       *pibLink;
        struct       dpvtBitBusHead *bitBusDpvt;
        int          dmaTimeout;
} dpvtGpibHead;
```

list	List node used to put this structure into the loPriList or hiPriList of an ibLink struct.
callback	Callback structure used in the completion phase of asynchronous record processing. Not used directly by the driver.
workStart	A pointer to the work function. It should return IDLE if the request has been completed and BUSY if not (ie because it w aits for SRQ completion). As parameter it should accept a pointer to the request it is part of (the dpvtGpibHead).
link	The link number.
device	The GPIB address.
pibLink	A pointer to the corresponding ibLink structure.
bitBusDpvt	Obsolete.
dmaTimeout	Obsolete.

srqStatus

This structure represents the items that are placed on the ibLink->srqRing. Its fields are:

```
typedef struct srqStatus {
    unsigned char device;
    unsigned char status;
} srqStatus;
```

device	GPIB address of device.
status	Result of the srq poll (status byte).

drvGpibSet

The driver exports a set of entry points via its driver support entry table. The entry table has the following entries:

```
typedef struct drvGpibSet {
        long         number;
        DRVSUPFUN    report;
        DRVSUPFUN    init;
        int             (*qGpibReq)();
        int             (*registerSrqCallback)();
        int             (*writeIb)();
        int             (*readIb)();
        int             (*readIbEos)();
        int             (*writeIbCmd)();
        int             (*ioctl)();
        int             (*srqPollInhibit)();
} drvGpibSet;
```

number	Number of entries in this table.
report	Print report on the console.
init	Initialize driver support
qGpibReq	Queue a GPIB work request for future execution.
registerSrqCallback	Register an SRQ event handler.
writeIb	Write data to GPIB devices.
readIb	Read data from GPIB devices.
readIbEos	Read data from GPIB devices using end-of-string char.
writeIbCmd	Write raw data out the bus, while keeping the ATN line high.
ioctl	Provides access to low-level GPIB protocol operations. Also used to create and retrieve links.
srqPollInhibit	Mark a given device as non-pollable.

For a detailed descriptions including parameters and return values see Section 3.3 on page 11.

drvGpib

extern drvGpibSet **drvGpib**;

The driver entry table variable.

3.2 Constants

These are also defined in drvGpibInterface.h.

IOCTL commands

The possible values for parameter cmd in function ioctl (see Section on page 13).

IBNILNK	Return the max allowable NI links. Obsolete.
IBTMO	One time timeout setting for next GPIB command. Obsolete.
IBIFC	Send an interface clear pulse.
IBREN	Turn on or off the REN line.
IBGTS	Go to controller standby (ATN off...).
IBGTA	Go to active state.
IBGENLINK	Ask the driver to start a link running.
IBGETLINK	Request address of the ibLink structure.
IBGTL	Go to local.
IBLLO	Local lockout.
IBDEVCLEAR	If the parameter v is the GPIB address of the controller, then send a DCL (device clear, all devices). If it is a device address then send a SDC (selective device clear).
IBRESETLNK	Reset this link, reinitialize all communication.

Remarks:

- The commands IBNILNK and IBTMO are now obsolete and should not be used.
- The commands IBGENLINK and IBGETLINK should only be used during device support initialization. The parameter v here means GPIB address.
- All other commands may ONLY be used in the same way as the writeIb, readIb, ... functions, i.e. not directly but only from a work function, that is given to qGpibReq.

Work Request Priorities

IB_Q_LOW	Request has low priority.
IB_Q_HIGH	Request has high priority.

Device Status values

IDLE	Device is currently idle.
BUSY	Device is currently busy.

These are return values for request callbacks and SRQ handler functions. A return value of BUSY tells the driver not to initiate further requests to this device until IDLE is returned. Typically, a request that sends a command and expects an asynchronous answer (via SRQ) will return BUSY on sending the command and let the srqHandler return IDLE on completion.

Other Constants

IBAPERLINK	Max number of devices per link.

3.3 Functions

Some general remarks:

All functions in the driver except srqPollInhibit and resetIb are static and can only be accessed via the driver support entry table (see Section on page 9). Note that the names of the fields are not necessarily the names of the functions. A user call looks like drvGpib.*fieldname*(*parameters*...).

Return type for all functions is int (or long). They return ERROR if an error happened, otherwise OK. Exceptions: return ERROR or, if successful, the number of bytes actually read or written.

WARNING!

The user programs MUST NEVER call any of the functions readIb, readIbEos, writeIb, or writeIbCmd directly. This applies also to ioctl, if the cmd parameter is an ything but IBGENLINK or IBGETLINK. Instead the y must be called by a user supplied w ork function which is given indirectly as a parameter in the call to qGpibReq. This is check ed by an assertion![4]

report

long **report**(int *level*);

Print report on the console.

level	Interest level (0 and 1 supported).
Return value	OK.

init

long **init**();

First step of driver initialization. Automatically called by iocInit.

Return value	OK or ERROR.

qGpibReq

int **qGpibReq**(dpvtGpibHead **pdpvt*, int *prio*);

Queue a GPIB work request for future execution. The ONLY way a user function can initiate a GPIB message transaction.

[4] Assertions also check for valid parameters like the pointer to the link structure (must be !=0) and the GPIB address (must be between 0 and 30). To disable assertions, compile the driver with the symbol ‘NDEBUG’ defined.

A work request represents a function that the ibLinkTask is to call (when ready). A pointer to this wrok function is contained in the dpvtGpibHead structure. This work function is allowed to call the readIb, readIbEos, writeIb, and writeIbCmd functions.

pdpvt	Head of device private structure.
prio	Priority of the request. Can be IB_Q_LOW or IB_Q_HIGH.
Return value	OK or ERROR.

registerSrqCallback

int **registerSrqCallback**(ibLink *_pibLink_, int _gpibAddr_, int (*_handler_)(), void *_parm_);

Register an SRQ event handler. When the SRQ handler is called, it is passed the requested parm and the poll-status from the gpib device. Similar to the workStart function, a registered handler should return either IDLE or BUSY, indicating wether the device is now ready to accept new commands and queries or not.

pibLink	Pointer to the link structure.
gpibAddr	GPIB address of device.
handler	SRQ handling function.
parm	Parameter to be passed to handler.
Return value	OK or ERROR.

writeIb

int **writeIb**(ibLink *_pibLink_, int _gpibAddr_, char *_data_, int _length_, int _time_);

Write data to GPIB devices.

pibLink	Pointer to the link structure.
gpibAddr	GPIB address of device.
data	Pointer to the data to be written.
length	Number of bytes to be written.
time	Timeout (in ticks) for the operation.
Return value	Number of bytes written or ERROR.

readIb

int **readIb**(ibLink *_pibLink_, int _gpibAddr_, char *_data_, int _length_, int _time_);

Read data from GPIB devices. Equivalent to readIbEos(...,-1).

pibLink	Pointer to the link structure.
gpibAddr	GPIB address of device.
data	Pointer to a buffer where to put the data.
length	Length of data buffer.
time	Timeout (in ticks) for the operation.
Return value	Number of bytes read or ERROR.

readIbEos

int **readIbEos**(ibLink *_pibLink_, int _gpibAddr_, char *_data_, int _length_, int _time_, int _eos_);

Read data from GPIB devices; extra parameter specifies end-of-string character.

pibLink	Pointer to the link structure.
gpibAddr	GPIB address of device.
data	Pointer to a buffer where to put the data.
length	Length of data buffer.
time	Timeout (in ticks) for the operation.
eos	A character that signals end-of-string.
Return value	Number of bytes read or ERROR.

writeIbCmd

int **writeIbCmd**(ibLink *_pibLink_, char *_data_, int _length_);

Write raw data out the bus, while keeping the ATN line high. This is almost obsolete.

pibLink Pointer to the link structure.

data Pointer to the data.

length Number of bytes to write.

Return value OK or ERROR.

ioctl

int **ioctl**(int *link*, int *link*, int *bug*, int *cmd*, int *v*, void **p*);

Provides access to low-level GPIB protocol operations. Also used to create and retrieve links. Parameter cmd, tells it what to do exactly. Most of these commands are obsolete now. See Section 3.3.1 on page 8 for a list of supported values for cmd.

linkType Must be GPIB_IO.

link Link number. If p points to a valid link structure then link is checked against the link number inside the link structure. Otherwise the link structure is searched using the link number.

bug Obsolete. Set to -1.

cmd The command to execute.

v Depends on cmd: If -1, ignored. If 0 or 1 and cmd=IBREN then means OFF resp. ON. Otherwise means GPIB address if that makes sense.

p Depends on cmd: If cmd=IBGETLINK then pointer to pointer to the hpLink structure (result). Else pointer to a valid hpLink structure (see description of item *link*).

Return value OK or ERROR.

srqPollInhibit

int **srqPollInhibit**(int *linkType*, int *link*, int *bug*, int *gpibAddr*);

Mark a given device as non-pollable. This is only necessary for some older devices who are too dumb to deal with beeing polled.

This function is may be called from the VxWorks command line or from a startup script before iocInit.

linkType Must be GPIB_IO.

link Link number.

bug Obsolete. Set to -1.

gpibAddr GPIB address of device.

Return value OK or ERROR.

resetIb

void **resetIb**(int link);

This function is an exception, in that it is not part of the drvGpibSet. It is meant to be called from the VxWorks command line, but *during* operations: it is a convenient way to re-initialize a link, for example if the gateway has been shut down or disconnected. It is called with the link number as argument.

3.4 Variables

There are some global variables that may be changed from the startup file or from the command line. They adapt the behavior of the driver to handle different situations. The flags all default to zero (off).

Debug Flags

int ibDebug Set to 1 to turn on debug messages. This slows down the driver considerably.

int ibSrqDebug Set to 1 to turn on only debug messages related to SRQ handling.

Other Flags

int ibSrqLock Set to 1 to disable all SRQ checking and polling.

int ibRecoverWithIFC Set to 1 to fire out an IFC pulse after device timeouts.

int ibDeviceLLO Set to 1 if devices should be locked on initialization.

Timeouts

int ibSrqTimeout Number of seconds to wait for SRQ completion. Default is 2.

int ibSrqRingZize Maximum number of events stored in the SRQ event ring. Default is 2.

int hpLanTimeout Number of seconds after which gateway is assumed dead. Default is 10.

struct timeval rpcTimeout May be changed if device timouts are greater than 10 seconds. Default is {10, 0}, meaning 10 seconds, 0 microseconds.

4 Installation

This section explains how the lanGpib driver is installed in an existing EPICS environment. By far the easiest way to install the package is to create a seperate <top> area (named, for example, 'GPIB') with makeBaseAp.pl (this requires that you use EPICS3.13.1 or later). Copy the tar file into the new <top> directory and unpack it. You may need to adapt the config/RELEASE resp. config/RELEASE.<arch> to your local EPICS configuration. Then 'make' everything from the <top> directory. Be sure that any application that uses GPIB has an entry for this <top> area in its local config/ RELEASE file. THIS ENTRY MUST COME BEFORE THE ENTRY FOR THE EPICS BASE. Otherwise the application will include the old header files from base, which will lead to unpredictable results.

If you use an EPICS release before 3.13.1, or you don't want to use makeBaseApp, or you cannot for any reason make a separate <top> area or use the Makefiles provided with the package, you will be interested in the following information. The term 'application' refers to ANY C-code, that includes the above header files. This includes device support modules.

Files

You must compile and link at least the following files to your application:

- drvLanGpib.c
- devCommonGpib.c
- vxi11core_xdr.c
- vxi11intr_xdr.c

Further more, you must replace the following header files

- drvGpibInterface.h
- devCommonGpib.h

by the new version. The new versions are software compatible with the old ones (i.e. old device support modules will compile without change). The best way to do that is to add them to the INC macro in your local Makefile.Vx like

```
INC += drvGpibInterface.h
INC += devCommonGpib.h
```

in the same place where you usually put LIBOBJS and similar stuff. This installes them into <top>/include. If you use an EPICS relase before 3.13.1, it may be necessary to add the line USR_INCLUDES += -I$(INSTALL_LOCATION)/include to the Makefile.Vx of your application.
The header files

- vxi11core_xdr.h
- vxi11intr_xdr.h
- drvLanGpib.h

are only used internally by the driver.

A special thing is the header file

- devGpib.h

because it is not necessarily needed in order to write new device support modules, but *very* useful if you don't like typing. So it should also be installed in your <top>/include directory. You can use

- devNewSkeletonGpib.c

as a kind of template for GPIB device supports. This is an example file that shows how to implement GPIB device support modules (see also J.W.'s doc on the GPIB device support library). Use your favorite editor to search-and-replace 'Skeleton' by the name of your device. The resulting file should at least compile without errors. devNewSkeletonGpib.c is contained in the Makefiles for devCommonGpib, to check compliance. It can be commented out, there.

VxWorks Startup File

In principle this driver needs no special call from your startup file. Nevertheless there is one case in which you must make such a call. This is when you are working with some older device that doesn't like to be polled (as a result of an SRQ). Then you have to make the call srqPollInhibit(<linkId>,<gpibAddr>) which prevents this device from being polled when an SRQ is encountered. You can make as many of these calls as you like.

You can set the global variables ibDebug and ibSrqDebug at any time to values other than zero in order to generate debug messages, although I doubt that they are really useful to someone who has not studied the source code carefully (or written it). WARNING: this slows down the driver remarkably!

Other variables that may be set in the startup file are listed in Section 3.4 on page 13. The rpcTimeout is a special thing: you can treat it like an integer value (specifying only seconds) from the command line or startup file.

Old Device Support

If you want to use old device support modules written for the NI1014/BitBus version of the GPIB driver, you have to remember:

1. The link numbers now have a slightly different meaning: Although in both versions a link number identifies a GPIB bus segment, the difference is that in the old version it was the equal to card number in the IOC (probably a value between 0 and 3) whereas in the new version it is the local part of the IP address of the gateway, and so can be any number between 0 and 0xFFFF. You must probably change your database definitions accordingly.
2. Be careful if device support calls the driver directly or if it uses the GPIBCNTL command (see devCommonGpib.h). This can lead to desaster, if not done with extraordinary care and knowledge - of GPIB as well as the driver.

In contrast to an earlier version of this driver, old device support modules in general do not need to be changed.

5 Release Notes

This chapter exists solely in order to keep track of the main development path. The documented changes are neither complete nor particularly reliable because they have mostly been written in retrospection. You can ignore them completely without loosing any vital information, especially if you use this package for the first time. It *may* be of interest for people who upgrade from earlier versions.

5.1 Release 2.4.a

This is a bug-fixed version of 2.4.

The startup message of the driver now correctly reports ‘release 2.4.a’.

The two Makefile.Vx have been cleaned up. The y no longer contain the USR_INCLUDE += xy, because this is not necessary for <top> en vironments created with the ne west version of makeBaseApp.pl.

I copied the fixes in devCommonGpib (detected and corrected by Marty Kraimer in the EPICS base version): the results of the sscanf calls are no w tested ag ainst ==1 instead of !=0.

The claimed compatibility to old device supports was violated in two places:

- Several #include statements were missing.
- The structure tag dmaTimeout in struct devSupParms was changed to timeout.

Both have been fixed, the first by including all necessary header files inside drvGpibInterface.h and devCommonGpib.h, the second by adding #define dmaTimeout timeout to devCommonGpib.h.

The re-init procedure had a severe bug that caused most records to hang with PACT==TRUE after a re-initialization. This was fixed by issuing any outstanding callbacks with a dummy return value before re-initialization (in addition to emptying the work queues; see function ibLinkTask).

5.2 Release 2.4

Most of the changes concern SRQ handling. A number of deeply hidden bugs have been found by Peter Müller from the PTB. He also helped a great deal in testing, analyzing the problems and finding solutions.

The general problem has been to use the VXI-11 function read_stb for serial polling. This call is seriously flawed and should not be used. Instead, I went back to the good old method already present in J. Winan’s old driver (i.e. do the ‘send SPE, read devices, send SPD’ cycle by simply using writeIbCmd and readIb). After correcting this, other errors (actually errors in the VXI implementation of the RPC calls) that were masked before, appeared. These were corrected (i.e. worked around) by inserting additional UNL and UNT commands into the read and write functions, wherever they were missing.

Maybe support for additional record types (mbb[io]Direct) has already been added in an earlier release. Can’t remember that exactly.

Complete revision of the documentation (phew!), including purge of all the obsolete stuff (the old remnants caused more confusion than providing any help). I kept the old release notes, though, but I rearranged them a bit.

5.3 Release 2.3

Many things have changed from release 2.0 to release 2.3. I’ll try to summarize them:

Initialization again

First, the initialization has been changed again. A device link is now created for every possible device address on a link on the first call to drvGpib.ioctl(IBGENLINK). This has been done in order to prepare for a unification of all the GPIB drivers that is overdue and should be done by the end of the year 1999. As a result, the dynamic initialization feature of release 2.0 became to a large part - but not completely - obsolete.

On the other hand, a problem that had already been observed in earlier versions was now no longer tolerable: If a lot of links have been created and then the IOC was rebooted, the gateway sometimes failed to create some of the the new links because it erroneously thought that the old ones were still valid (error VXI_NORES = no resources left). No possible timeout configuration of the gateway seemed to solve the problem. Furthermore, due to the idiotic way in which VxWorks makes a reboot (the netTask is always shut down before any other task) it is not possible to install a rebootHook to clean up things.

The only way out of the trouble was to reset the gateway before initialization. This has been done by opening a telnet socket (port 23) to the gateway and automatically inserting ‘reboot\ny\n’. After that, we wait 5 seconds and then proceed. Mark that this is also done on re-initialization (see “Re-initialization” on page 18).

Mind that it is still necessary to register every single device with the driver, since unregistered devices are never polled (to avoid never ending timeout storms during poll).

Other Driver Changes

First, the code has been thoroughly tested under EPICS Release 3.13.1.

The srqAcknowledge function is now obsolete and has been removed from the header file drvGpibInterface.h as well as the finish function. Pending SRQs are now acounted for in a much simpler fashion (once again Till Straumann pointed that out to me).

The SRQ polling procedure has been fixed. There are severe bugs in the read_stb RPC call (no SPD after timeout!!). This call is no longer used. Instead, (serial) polling is done in the good old way: SPE, read all registered devices, SPD. After that was done, it became clear that a number of devices completely stopped working, after one polling round. A lot of GPIB analyzing revealed that the RPC call device_read and device_write never did sent UNT messages. After adding UNT at the beginning of a read/write operation and UNT+UNL at the end, everything worked fine.

Timeouts are no longer reported on the command line.

The symbol DEBUG must be defined in order for the driver and device support library to generate debug messages.

The ibLink structure is no longer public. There has never been a need for this.

Device Support Library

The GPIB device support library has been overhauled quite a bit. It does not look like the original one any longer.

Some structure definitions in devCommonGpib.h have changed slightly. The member dmaTimeout in struct devGpibParmBlock has been changed to timeout; the members bug and linkType have been removed from struct hwpvt; the members process and processPri in struct gpibDpvt have been replaced by the member callback; a new member timeout was added and linkType was removed from struct gpibDpvt. All these changes should not interfere with existing device supports since they are used only by the driver and the library.

A new GPIB command GPIBIOCTL has been added, see the comment in the header file.

Read with terminating EOS character is now fully supported, see remarks in the header file.

A default srq_handler as well as a default get_io_int DSET function are exported by the GPIB device support library and may be used by device supports.

Peter Müller from the PTB helped a great deal in debugging and error detection. Most of the improvements would not have been possible without that.

5.4 Release 2.0

Lazy Initialization

Due to the work of Till Straumann from the PTB (Physikalisch-Technische Bundesanstalt) the driver supports lazy initialization. New devices and even new gateways can now be added at runtime (and not only during iocInit).

For the EPICS Application Developer the main difference is that the initHooks call is now completely obsolete. For the EPICS system programmer it means mainly that other drivers (such as motor controllers) may call drvGpib.ioctl(IBGENLINK,...)), even before drvGpib.init has been called. Semantics of this command now includes creating only the link but no device connection.

Initialization procedes as follows:

If iocInit calls drvGpib.init before any other module, the first step of initialization is the same as before. Additionally, this stage can be jumped over by immediately calling drvGpib.ioctl with command parameter IBGENLINK. Any further call to drvGpib.init will be ignored.

The driver function drvGpib.ioctl, when called with command IBGENLINK, first checks if there is already a linkTask for the given link number. If not, the task is created and ioctl blocks until the task gives the initSem. If the parameter v is NONE[5] nothing further happens. If it is a valid GPIB address, the device link to this address is created by sending a certain work request to the linkTask and taking initSem which is used here for a second synchronization. This request contains as workStart procedure a special callback that first generates the device link and afterwards gives the initSem, thereby signalling drvGpib.ioctl that it is finished.

A whole bunch of new semaphores guarantee data integrity in case a link is generated at runtime. Mind that *link deletion is not implemented* (see also next subsection).

Re-initialization

A common problem with previous releases was that when due to external interferences the gateway had to be reset (ie turned off and on again) or was disabled for other reasons, the IOC had to be rebooted also. There is now a mechanism inside the driver that recognizes a failure of the gateway by probing the bus status every 10 seconds. If there are severe timeouts the linkTask is shut down and no more requests are queued. The link structures are, however, not deleted. A special command IBRESETLNK has been added to be given to drvGpib.ioctl:

IBRESETLNK — Reset this link, reinitialize all communication.

It can be (but does not have to be) called when the driver detected a broken gateway connection. It can also be called at any other time.

Support for EPICS R3.13.0.beta11

Three new files are included to support beta11: lanGpib.dbd and lanGpib.LIBOBJS can be used to include the driver into an application, devCommonGpib.LIBOBJS to include the GPIB device support library. All three are installed into <top>/dbd; the makefiles have been changed accordingly. It is recommended to install the package under the ‘share’ directory.

Miscaleanous Changes

In previous releases, every registered device was cleared on initialization (with an SDC). This could lead to problems with some strange devices that weren’t happy at all about that. I threw it out. Instead there is a new command for drvGpib.ioctl, called

IBDEVCLEAR — If the parameter v is the GPIB address of the controller, then send a DCL (device clear, all devices). If it is a device address then send a SDC (selective device clear).

With this command the application can do an SDC at init time or whenever necessary.

[5] NONE is defined by VxWorks to be -1.