Linux-GPIB User's Guide

 Claus Schroeter (clausi@chemie.fu-berlin.de)

Version 1.0

The Linux-GPIB Package is a Driver and User-Interface Package for usual GPIB cards. This Document describes the basic steps for installation and usage of the package.

Contents

Copyrights

✘ There are special Copyrights for the esp488 package in ./driver:

Introduction

1.1 Short Overview

Let's understand this package as the begin of a larger project to make linux more useable to people dealing with automation, process control and engineering in laboratory and industrial environment. Realize that there is much work todo (with this and other projects dealing with this stuff).

The IEEE488 or GPIB Standard, first developed by HP (1965) (called HPIB) provides a wide spreaded construction kit that makes measurement, data aquisition and control as simple as possible.

Til now this bus-system wasn't availiable for linux users because the Hardware wasn't supported yet. I want to fill this gap step by step. My second Idea is to make it as simple as possible for the (perhaps unexperienced) user to write applications. This can be done step by step too.

Under DOS a wide Variety of nice (colorful) Software exists to simplify the GPIB Handling. That makes a lot of people afraid to going to LINUX. I want to fill this gap too. The first step i did was to implement a interface to TCL/TK, a powerful Toolkit Language by John Ousterhout, to the GPIB library. With this simple a-few-hundred-lines C interface users could write very poverful Programs with comfortable graphical User Interfaces under X11 within some Minutes (depending on the knowledge of tcl/tk).

At this time a lot of extensions exists for Tcl/Tk so that it were possible to provide GPIB functions over a network or write client/server Measurement systems. There are no limits to the phantasies what to do with this tool.

As i mentioned this should be the first step. So i made a driver for the mostly used Card (AT-GPIB from NI) and i hope the support for other cards follows soon.

1.2 Features

The Package consists of four Parts:

① **The driver Module**
This piece of code does all the dirty work, write magic numbers in magic places and gives the results back.

The Driver has been designed as loadable Module and is full Runtime configurable. You can change the functionality, base-address etc at runtime without rebuilding the kernel. To Install the driver, minor changes to the Kernel are required.

Normally once the module has been build you could use it for many purposes without recompiling.

② **The GPIB Library**.
This Library provides higher level calls to C-Programs, that makes the work easier. Tere are simple to understand functions that provide Device-Management, configuration and read/write commands to the GPIB the bus.

The function names of the most Library functions are similar to the National Instruments NI488.2™Library. Porting of your programs should be simple.

The library does the whole driver runtime configuration that could be specified in a special configuration file.

A simple Errorlogging Mechanism is provided so that every Operation on the driver could be backtraced on /dev/console or whereever.

③ **The TCL/TK extension**
I think this is the most popular thing of this Package.

With the TCL/tk Interface you can test your hardware, play around and write simple applications with graphical user interfaces within some minutes.

I added the XY-Graph Plot extension to this Package (created by Sani Nassif and George Howlett AT&T) that will be useful to GPIB-Programmers: You can read out a device and Plot it directly to screen. (have a look to the demo files)

A Software Diagnostic Program that checks your Setup for installation errors is also provided.

1.2.1 Additional Documentation

For finding routines and driver conventions quick and easy a handy Quick-Reference-Guide is included in the documentation (refcard.ps).

1.3 Requirements

1.3.1 Hardware

The following GPIB Hardware is supported by this Package:

- National Instruments AT-GPIB Boards
  There are three types supported - one with NEC7210 TLC and TURBO488 chip, one with NAT4882/TURBO488 chips and the newest one with TNT4882 ASIC.
- National Instruments PCII Boards
- National Instruments PCIIa Boards
- IBM GPIB Adapter (PCIIa compatible)

1.3.2 Additional Software

You will need Linux-1.1.61 or higher to work with this Package.

To install the driver to your kernel you will need the modutils package (insmod/rmmod).

If you want to use dynamic DMA Memory allocation you have to patch your kernel with the dma-manager patch in ./patches.

If you want (and i bet you will) the TCL/tk Interface you need a working tcl/tk library (my version was tcl7.3 and tk3.6 but i think you can use newer versions as well). You also can get a shared binary version from sunsite or tsx-11. The installation should be very simple. It's recommended to have the Tcl/Tk sources availiable if you want to write own Widgets for Tk or if you Want to add other TCL/Tk extensions to ibsh.

The original National Instruments sample files could be helpful for beginners to understand how to write GPIB Programs.

Installing the Software

2.1 Preparing

There are some things you must do before you can slip into any Linux GPIB stuff.

❶ Unpack the gpib-linux<version>.tar.gz File to your Source Directory (say `/usr/local/src`). (Oh what stupid, if you read this you will have done this yet :-))

❷ Now install the additional Software (mentioned above) first. On newer distributions as Slackware-2.0 you will not need to install the modutils Package. Check out if insmod,rmmod,lsmod exists in `/sbin`. It's recommended to make a Directory (or symbolic link) tcldev in your Source Directory where the whole Tcl Stuff lives. If you install the Tcl Sources compile them and make install to put the binaries to the right places. Edit `ibsh/Makefile` for your requirements.

❸ You have to prepare the Kernel for exporting the needed Symbols to the driver module. Edit the File /linux/kernel/ksysms.c (or ksyms.c for newer kernels). See the sample.ksyms.c File in ./driver for the required symbols With newer Kernels (> 1.2.0) you should only need to define `high_memory`

❹ Now recompile your kernel and install it as default.

❺ Configure your board-jumpers to the desired base-address,IRQ, and DMA

❻ Reboot your system

2.2 Compile-Time Options

There are some Options that could be set for the driver. You can decide if you want continous Debugging or if the driver should run in DMA transfer mode. For some pathologic configurations you can slow down the Bus-I/O.

Compile time options are set with the configuration script provided with the package. First time you're building the package the configuration will be done automatically. Later you can reconfigure your driver with `make config`

2.3 Compiling the Sources

Change to the toplevel Directory (say `/usr/local/src/atgpib-linux<version>`).

Now type make and answer the Questions from the Configuration.

You will get an object file `gpib<id>.o` (id is the id number given in `driver/Makefile`) If the driver has sucessfully been build try to load it (directly via `insmod gpib<id>.o` or by typing make load)

If no error message occurs you have loaded the driver succesfully into the kernel. Look at the File `/usr/adm/messages`, you should see some messages related to GPIB.

2.4 Configuring Driver/library Characteristics

2.4.1 Configuring the Driver

Everything the driver Module needs to know from the User is the Base Address, the IRQ Level and DMA channel, it's Major Number and optionally the Debug Level.

The Base, IRQ and DMA settings depends on your board jumper settings. Refer to your Hardware installation Guide for details.

The Major Number is used to decide wich driver inode file is suitable to which module. This Number has to be unique for your System. To find out which Majors are used by other Drivers check out `/proc/devices`

The Debug Level determines wich actions of the Module should be logged by syslogd to `/usr/adm/messages`.

The Driver Module is Configurable with three Methods:

- Via the compile-time options.
- At Loading time you could specify options:

```
/sbin/insmod  gpib0.o [[option=value] ....]
```

`option` could be one of

ibbase specifies the Hardware address of the GPIB board (default is 0x2c0)

ibdma specifies the DMA Channel for the board (default is 5)

ibirq specifies the IRQ Level for the board (default is 11)

ibmajor specifies the Major Number for the module (default is 31) Realize that this Number must match your driver inode's Major Number

dbgMask sets your debugging level if you selected DEBUG Mode in the configuration. If you want to use the module with debugging enabled you should set the default debug level to NONE (Configuration).

- With Library routines and the library configuration script.

2.4.1.1 Specifying Debug Levels

If you enabled the Debugging Option in your Configuration Script you could back-trace every action that your driver is doing. One of the consequences of this Method is your `/usr/adm/messages` logging file will get very large. To prevent a messy logfile you could select what actions should be logged. The 8 bits of the Debug-Level have the following Meanings:

0	All actions will be logged
1	Log every Function Entry
2	Log every Function Exits
3	Branches (as DMA start etc)
4	Data given to functions
5	Interrupts and Timeouts
6	Register i/o
7	Reserved

Table 2.1: Debugging level Bits

Example: Load the Module with

```
/sbin/insmod gpib0.o dbgMask=0x06
```

and start the ibchk program. Your Kernel Logfile (`/usr/adm/messages`) will show up messages like:

Now you can backtrace any function that is called during program operation.

Example 2.4.1 Driver Logging

```
Nov 20 13:37:56 clausi kernel: AT-GPIB Driver -- Major=31
Base=0x2c0 Irq=11 DMA=5 enabled
Nov 20 13:37:56 clausi kernel: -- DebugMask = 0x106
Nov 20 13:37:56 clausi kernel: -- Kernel Release 1.1.51
Nov 20 13:37:56 clausi kernel: gpib - ibinstall:in
Nov 20 13:37:56 clausi kernel: gpib - ibinstall:out
Nov 20 13:38:16 clausi kernel: gpib - ibopen:in
                               .
```

2.4.1.2 Using DMA Transfers

If you want to use DMA Transfer Mode enable the DMA Mode in the Configuration Script. Unfortunately there are some caveats that should be mentioned.

If your System Memory is greater than 16M you should use the Dynamic DMA Manager Patch to the Kernel provided in `./patches/dma2909.diff.gz` (If it isn't applied yet).

With this Patch a fixed DMA Buffer in the lower 16M is allocated at boot-time that could be used by Modules similar to `kmalloc()/kfree()` .

To apply this Patch to the Kernel, copy the `./patches/dma2909.diff.gz` to `/usr/src`, cd to `/usr/src/linux` and type

```
zcat ../dma2909.diff.gz |patch -p0
```

now your kernel should have been patched.

Now reconfigure your kernel with `'make config'` and rebuild it.

2.4.2 Configuring the Library

The GPIB Library can be configured via the `/etc/gpib.conf` file.

✘ Realize that the driver does not know anything about this configuration if the Library is not used. If you want to use the driver without the library you must pay attention that your base address, IRQ and DMA has been set at compiling- or loading-time.

You're able to define mnemonics for your devices like "dvm" or "scope". The default configuration file is /etc/gpib.conf but you can override this by setting the `IB_CONFIG` environment variable to the desired path. Look at the default config file to understand the mechanism. You can also set the base address and the IRQ level of the card. Edit the configuration for your requirements.

You can set the board characteristics as base adress and irq within the config file for several boards.

The syntax is as follows:

Example 2.4.2 The Config-File Syntax

```
config { options }
   device { options }
   .
   .
   .

config { options }
   device { options }
   .
   .
   .
```

You can have `MAX_BOARDS` GPIB-cards in your PC[1]. Each config section begins the definition for devices connected to this board. The first config will refer to `/dev/gpib0`, the second to `/dev/gpib1` and so on.

Device Names consists of Letters and numbers and must begin with a letter.

Example: You have two gpib-boards in your computer one at 2c0h (default) and one at 3c0 with interrupt 10 and 11. A scope has been connected to the first board with GPIB adress 7 and a DVM has been connected to the second board. Your config file will look like

Example 2.4.3 A sample Config File

```
config { base=0x2c0 irq=11 }
 device { name=scope pad=7 }

config { base=0x3c0 irq=10 }
 device { name=dvm  pad=3 }
```

☛ The 'name' and the 'pad' option are required.

[1] The default of 4 GPIB boards should be sufficient

Since the last version of the library the EOS modes can be specified for each device seperately. This will be useful for situations where different devices with different fixed-eos handling are used on the same bus. The configuration provides an initialization string per device this string will be send to the device if ibfind() is called.

There are some other useful options look at ./lib/gpib.conf.dist for examples.

You can have one master device per GPIB-bus (Usually the GPIB-Adapter) To perform low level board commands to the whole bus this device must be configured as

controller. You can configure this special device with one `device{}` entry while setting the `master` flag on this configuration. Note that only one device on the bus can be the controller at the same time.

The Driver Module

3.1 Special Configurations

The users of GPIB will mostly be satisfied if strings can be transfered over the bus. This strings are usually not very large (a few bytes). One exeption are devices with large data buffers (as scopes or scanners etc). Reading or writing this buffers will consume some more time as single byte transfer does. Use DMA Transfer Mode will speed up this. The problem with DMA Transfer under Linux is the continous Memory block that you need for this transfer, it has to be aligned to a page boundary and must be in the lower 16M of System Memory. For drivers compiled into the kernel this is not a big problem as this requirements could be satisfied at boot time. A Module runs after the kernel has been loaded into the memory so there is no guarantee that memory pages you can get from kernel are located below 16M.

To fix this Problem The Driver uses the dma-manager from David Niemi (i added the Patches to the Package). If you want to use DMA and your Memory is greater than 16M you should apply this Patch to your Kernel.

3.2 Testing the Configuration with `ibchk`

For a simple error check run `util/ibchk`. ibchk will test your Configuration against the Hardware preset board characteristics and reports useful error messages if something is going wrong. Ibchk looks if the device file has been created with the correct major number or if a gpib board is present at the desired base adress, it looks for reused interrupts or dma channels and reports any errors with suggestions to fix them. Since ibchk doesn't know anything about your GPIB Bus Configuration, your device setups can't be tested with ibchk.

Ibchk is **not** a program that should be used as programming example as it looks

MSB								LSB							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
r	r	r	S	S	S	S	S	r	r	r	P	P	P	P	P

Table 3.1: Minor Number Bits (P=Sad,S=Sad,r=reserved)

very deep into the driver specialities to do its job.

3.3 Testing the Library

This is your task because I don't know anything about your devices connected to your gpib. I added a sample lib/ibtest.c you can change it for your devices. If you build ibsh you can do this via tcl. Look at ibsh/demos/ibtest.tcl for example.

Note: Realize that the GPIB driver used with the GPIB library serves not as a usual character driver since all operations are done via ioctl calls. A User Program need to be linked to the GPIB Library if you want to use the config file Mechanism.

3.4 Using the VFS Interface

The driver module provides a simple Mechanism to send or receive bytes to or from the bus via normal read/write operations. In this case where simple i/o is sufficient the driver can be used as normal character driver. To use the full functionality of the driver it is recommended to use the GPIB-Library for your application programs.

In VFS-mode the driver automatically initializes the Bus for transfer and implicitely set up the device addresses decoding the MINOR number of the device special file.

If you want to use the VFS mode you have to create one device special for each device connected to the bus using the `mknod` command. It would be useful to create an own subdirectory within `/dev`, for example `/dev/GPIB`.

Example 3.4 shows how to create the inode files for three imaginary devices one called "tektronix604" with `PAD=7`, one called "counter" with `PAD=2` and one called "dvm" with `PAD=5`. To access this devices you can simply use shell commands as shown in the Example or you can use the `read()` or `write()` C-Library calls.

Example 3.4.1 Creating device specials for VFS Mode

```
mknod /dev/GPIB/tektronix604 c 31 7
mknod /dev/GPIB/counter c 31 2
mknod /dev/GPIB/dvm  c 31 5

echo "D1Ready" >/dev/GPIB/dvm
```

Currently the VFS Mode is under construction so features as setting EOS modes and issueing commands to the bus via VFS are not possible.

3.5 Multiprocessing

While sending or recieving commands, the bus system runs through different states to distinguish between the different phases of the transfer. A running transfer can't be interrupted until the bus reaches it's initial state because the usual TLC's cannot save the bus state on a exeptional condition. So it's the software's task to pay attention in which state the bus transfer can be interrupted to give other processes a chance to make their transfers.

The linux-GPIB driver is capable to share the bus between different processes on a per-command mechanism. That means that each request is queued but one command can be transferred at the same time and other processes must wait for its completion. For the linux-GPIB programmer that means that he has to keep the transfers as short as possible if he want to have the Multiprocessing feeling.

Using the GPIB Library

4.1 What is goin' on behind the scenes?

From User's Side of View GPIB is a simple thing: You have some devices as Scopes, Digital Multimeters, Counters and so on. Each of this device has a Plug on the rear side of it's case and is labeled to understand ieee488 commands. You buy one of this expensive Cables and plug it from device to computer – and ready to go (You think).

Yes from Hardware side (outside the devices) this seems to be simple but to write Programs using GPIB you have to know more about what is going on between the computer and the devices.

I think we should not explain all the Lines of the cables in detail but it's not wrong to know a little about what's going on behind this plugs.

As mentioned in each device's Manual (i bet) you can select a GPIB-Adress for each device. This 'device adress' has to be unique for the bus. With this adress the bus-controller can find the device on the bus.

The GPIB bus can connenct many devices that could communicate independently over the bus. A device that has the controller capability could take control over the bus and send commands to the devices. Only one device can be the controller at the same time.

The dataflow direction is controlled by the Talker and the Listener States of the Devices. More than one device could be in listener or in talker state.

If the Controller want to write out some bytes to a device it sets the device in listener state and itself to talker state, then the desired bytes are written over the bus (See Figure 4.1).

For read operations the controller will become the listener and the device will be adressed as talker, then the bytes will be read (See Figure 4.2).

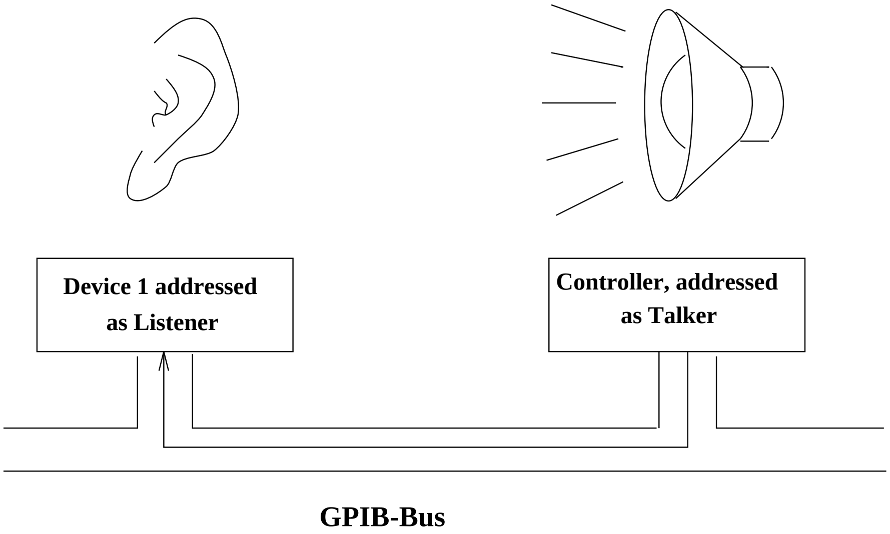

Figure 4.1: States During Write Operation

Similar to SCSI, each device (that means the interface of the device) can be in different states during the transmission Process. Some states are controlled by some Lines on the Bus (ATN,REN,IFC,SRQ,RQS) and some states could be reached with commands on the Data-Bus ($D_0 ... D_7$).

IFC (Interface Clear) resets all device-controllers to its initial State (Similar to a Chip Reset), this message will be send on initialization process of the system-controller.

ATN (Attention) and REN (Remote Enable) are used to signal a bus action to the device-controllers, this line is driven by the system-controller whenever you apply a command on the bus.

SRQ (Service Request) and RQS (Request Service) are used from device-controller to signal a special state to the system-controller, this message usualy will be send if the device has done the requested operation.

The so called multiline Messages (that means commands over the Data Lines while ATN is active) are grouped in unadressed Commands (Comands for all Devices on the Bus) and adressed Commands (for one or more specified Device).

Unadressed Commands are DCL (Device Clear), LLO (Local Lockout), PPU (parallel Unconfigure), SPD (Serial Poll Disable) and SPE (Serial Poll Enable).

Adressed commands are GET (Group Execute Trigger), GTL (go to Local), PPC (Parallel Poll Configure), SDC (Selected Device Clear) and TCT (take Control).

The Adresses itself will be send as special Multiline Messages. All devices on

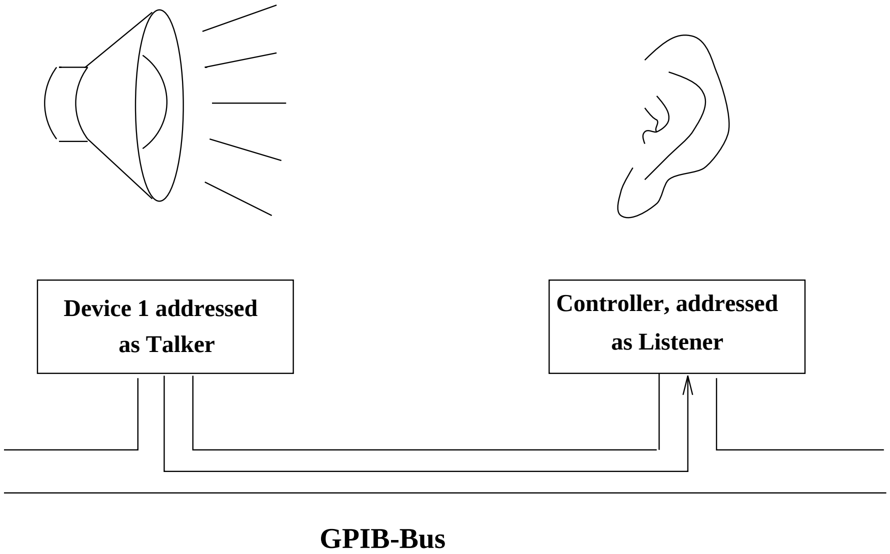

Figure 4.2: States During Read Operation

the bus can be adressed as Talker and as Listener. Each Transfer on the bus in adressed mode (usual with computer controlled GPIB) needs to set the Talker and Listener Adresses properly depending on which Direction the Transfer will be made. As soon as ATN is inactive every Byte on the bus is interpreted as valid Data. The additional Messages UNL (Unlisten) and UNT (Untalk) reset the desired Listener or Talker to its normal state.

Imagine you want to send a simple string "ABC" to your device, how is it done by the controller ?

The controller will activate ATN and send the Listener Adress of the desired device(s) then it sends its own Talker Adress. Now ATN will be deactivated and the data Bytes "ABC" will go over the bus. To finish the transmission the EOI Line will be activated. Now the controller can send the next string to the device or send UNL and UNT to unlisten the device.

As i concealed the controller has to setup the command Phase with IFC and REN first.

Lets show this Transfer on a nice Diagram:

The following Diagram shows how multiline Messages are coded on the eight Data Lines (a X means any Bit 0 or 1).

You will not need this for Programming with the GPIB Library but it will be helpful for advanced GPIB-Programmers or for finding bugs in the driver sourcecode.

ATN	command	EOI	
1		0	ATN starts the 'Command Phase'
1	UNL	0	Send Unlisten
1	LAD_1	0	Adress first Device as Listener
1	LAD_2	0	Adress second Device as Listener
	...		
1	LAD_n	0	Adress Device n as Listener
1	TAD	0	Adress One Device as Talker
0	DAB_1	0	Deactivate ATN, send first Data Byte
0	DAB_2	0	Send 2nd Data Byte
	...		
0	DAB_n	1	Send last Data Byte together with EOI
0	EOS		Send EOS Message

Table 4.1: Bus Commands during Transfer

4.2 How The Library is organized

4.2.1 Compatibility with NI-488.2™Library

The essential GPIB-Bus Handling functions should be compatible to NI-488.2™Library calls. Porting of User Programs from DOS to LINUXshould be simple. A short overview shows which functions are present in both librarys.

Message	Mnemonic	ATN Bit	DIO-Lines 7654 3210
Attention	ATN	1	XXXX XXXX
Data Byte	DAB	0	DDDD DDDD
Device Clear	DCL	1	X001 0100
End of String	EOS	0	EEEE EEEE
Group Execute Trigger	GET	1	X000 1000
Go To Local	GTL	1	X000 0001
Local Lock out	LLO	1	X001 0001
Talker Adress	TAD	1	X10T TTTT
Listener Adress	LAD	1	X01L LLLL
Par. Poll Configure	PPC	1	X000 0101
Par. Poll Enable	PPE	1	X110 SPPP
Par. Poll Disable	PPD	1	X111 DDDD
Par. Poll Unconfigure	PPU	1	X001 0101
Selected Dev. Clear	SDC	1	X000 0100
Ser. Poll Disable	SPD	1	X001 1001
Ser. Poll Enable	SPE	1	X001 1000
Take Control	TCT	1	X000 1001
Unlisten	UNL	1	X011 1111
Untalk	UNT	1	X101 1111

Table 4.2: Multiline Messages

Function	NI-488.2 Library	LINUXGPIB Library
ibbna	•	
ibcac	•	•
ibclr	•	•
ibcmd	•	•
ibcmda	•	
ibconfig	not identical	•
ibdev	•	
ibdma	•	
ibeos	•	•
ibeot	•	•
ibevent	•	
ibfind	•	•
ibFindDevIndex		•
ibgts	•	•
ibist	•	
iblines	•	•
ibln	•	
ibloc	•	
ibonl	•	•
ibpad	•	•
ibParseConfigFile		•
ibPutErrlog		•
ibPutErrMsg		•
ibpct	•	
ibppc	•	
ibrd	•	•
ibrda	•	
ibrdf	•	

Function	NI-488.2 Library	LINUXGPIB Library
ibrpp	•	•
ibrsc	•	
ibrsp	•	•
ibrsv	•	•
ibsad	•	•
ibsic	•	•
ibsre	•	•
ibsrq	•	
ibstop	•	
ibtmo	•	•
ibtrap	•	
ibtrg	•	•
ibwait	•	•
ibwrt	•	•
ibwrta	•	
ibwrtf	•	

Due to the fact that You're using a Unix compatible operating system with all its advantages and due to the different architecture of the driver there are some incompatibilites in the functionality of the Driver/Library that should be mentioned[1].

- ✗ NI-488™uses `decl.h` as header file to its applications. Linux-GPIB uses `ib.h`
- ✗ NI-488™automatically performs IFC, REN, and LLO to the bus when the first device function is called. Linux-GPIB performs this commands only for the first call to `ibfind()` and only for the first process using the bus (If you don't want any of this commands at init time you can configure this in `/etc/gpib.conf`).
- ✗ NI-488™can be configured so thet the board will request to be the Controller-In-Charge whenever a device function is called. Linux-GPIB always assumes the board is currently Controller-In-Charge.
- ✗ `if(ibfind,"gpib0") < 0) {...}` will work for NI-488 (since MS-DOS has 16 bit integers), but will not work with Linux-GPIB. You should use `if(ibfind,"gpib0") & ERR ) {...}` instead.

[1]Thanks to Glenn!

4.2.2 Device Management

Each device connected to the bus must have a unique GPIB-Adress set. This can be done via the Front Panel Buttons or rear panel DIP-Switch depending on the Device. Refer to the 'Selecting GPIB Adress' Section of Your Device's User Manual.

The Configuration File for the GPIB Library assigns your Devices with the Interface board. You must give each Device an unique name wich identifies the Device and set The Primary Adress for this Device.

If you make a call to ibfind first time the following actions will be done by ibfind automagically:

- ibfind looks if the IB_CONFIG Environment is set and parses the configuration file. If is not set it parses the default config file (/etc/gpib.conf).
- Each device found in the config file will be added to the device descriptor table.
- The Error Logging File will be opened (see Error Logging)
- The device driver (e.g. /dev/gpib0) will be opened and the configuration of the driver will be made (set irq,dma and base-adress).
- The driver will be set in Active State via ibonl.

4.2.3 Error Logging

Every time a library call results to an error, a Message is logged to the Errorloging file specified in the errlog variable. If no errlog variable has been specified in the config file Error Logging will be done to stderr.

If you work with X11 you could use xconsole to Log your Errors set the errlog variable to /dev/console for this.

Each Message has the following format:

```
routine : [time](device) < state > error <explanation>
```

If the error occurs while no device is active `device` is empty.

`state` is a combination of states as described below. A verbose explanation of the error follows the message. With this feature a error can be easyly backtraced.

For example means that ibwrt failed while performing the ioctl to the device (in ibBoardFunc) the reason of the error is a not responding device (no listener).

Example 4.2.1 Error Messages

```
dvwrt          :[Sun Sep 11 15:41:02 1994](-)<  ERR CMPL CIC TACS>  ENOL <No Listene
ibBoardFunc    :[Sun Sep 11 15:41:02 1994](-)<  ERR CMPL CIC TACS>  ENOL <No Listene
```

4.2.4 Device Status and Error Codes

As the Driver and the Library should be NI-488.2™compatible, two global variables exists for driver state and error reporting. An additional variable carries information about the number of bytes transferred on a I/O operation.

```
int ibsta;
int iberr;
int ibcnt;
```

4.2.4.1 Status Word

The Meaning of the Status Bits shows the following table:

State	Description
ERR	Function Terminated on error
TIMO	Time Limit on I/O or wait function exceeded
END	EOI Terminated ibrd
SRQI	SRQ is asserted
CMPL	I/O is complete
CIC	Controller in Charge
ATN	Attention is asserted
TACS	Board is adressed as TALKER
LACS	Board is adressed as LISTENER

For detailed Information on these codes See Ni-488.2™Software Reference Manual for MS-DOS (Part Number 320282-01)

4.2.4.2 Error Codes

Error Code	Description
EDVR	system error
ECIC	not CIC
ENOL	no listeners
EADR	CIC and not addressed before I/O
EARG	bad argument to function call
ESAC	not SAC
EABO	I/O operation was aborted
ENEB	non-existent board (GPIB interface offline)
EDMA	DMA hardware error detected
EBTO	DMA hardware uP bus timeout
EOIP	new I/O attempted with old I/O in progress
ECAP	no capability for intended opeation
EFSO	file system operation error
EOWN	shareable board exclusively owned
EBUS	bus error
ESTB	lost serial poll bytes
ESRQ	SRQ stuck on
ECFG*	Config file not found
EPAR*	Parse Error in Config
ETAB*	Table Overflow
ENSD*	Device not found in Configuration
*=	Linux GPIB Library only

If `iberr` is EDVR ibcnt is set to errno and further Explanation is written to Error Logging file.

For detailed Information on these codes See Ni-488.2™Software Reference Manual for MS-DOS (Part Number 320282-01)

4.3 Getting Started

The simplest task using a GPIB device is sending or receiving strings over the bus for example to setup device characteristics or receiving measurements from the device. For easy understanding let us imagine a virtual Digital Voltmeter (DVM) that understands commands to set the Voltage Range and returns a measurement result on readout.

Example 4.3.1 A Simple C-Program

```
#include <stdio.h>
#include <ib.h>

main(){

int  dvm;     /* The Unit descriptor of the device */
char cmd[80]; /* A buffer for the string           */

/* get device from configuration */

if( (dvm=ibfind("dvm")) & ERR ){
printf("Can't find DVM!\n");
exit(1);
}

/* Clear the Device              */

if( ibclr(dvm) & ERR ){
printf("Clear Error!\n");
exit(1);
}

/* Now Setup a few things */

strcpy(cmd,"D0 R1 S0");
if( ibwrt(dvm,cmd,strlen(cmd)) & ERR ){
printf("Write Error!\n");
exit(1);
}

/* now Read back the result */

if( ibrd(dvm,cmd,79) & ERR ){
printf("Read Error!\n");
exit(1);
}

        /* print out result */
printf("\n Result=%s",cmd);

}
```

Program 4.3.1 shows how such task could be solved using Device-Level Functions.