E488S
GPIB-Ethernet Gateway

and

NTO-GPIB
GPIB-Manager for QNX Neutrino

V1.3

Preface

This manual contains the description of the GPIB/Ethernet gateway E488S from Andre Koppel Software and the standalone version of the GPIB manager for QNX Neutrino that is used within the gateway. The actual version of this manual and updates of the GPIB manager can be found at www.akso.de.

The GPIB manager used within the gateway is identical to the standalone version with one difference. The gateway uses a special kind of license key and using the GPIB manager with a E488S license key only works within the E488S GPIB/Ethernet gateway hardware.

To run the GPIB manager a valid license key must do exist within the configuration file. Without a license key, the GPIB manager runs in demo mode for a short time. An advanced demo license key can be requested at any time from Andre Koppel Software or SW Datentechnik GmbH.

To get the full functionality a standard license key is needed. It can be purchased from SW Datentechnik GmbH. For pricing and additional information pleae get in contact with SWD at www.swd.de.

Andre Koppel Software GmbH
Köhlerstr. 23
12205 Berlin/Germany
Tel: (+4930) 81009190
FAX: (+4930) 32601046
Email: info@akso.de

Contents

1 Unpacking and Basic Setup

While unpacking the E488S-system, you will find the following parts within the box:

- The E488S GPIB/Ethernet Gateway itself
- A power supply
- A power cable
- A CD that contains example programs for use on the client side

The power supply supports a wide voltage and frequency range so it should work in every environment.

There are several connectors on the rear side that are identical to the connectors found on a PC, but only the GPIB-, the power- and the Ethernet connectors are used to get a working system.

To get the system running the power supply must be connected to the box. The power switch can be found on the rear side near the power connector. According to IEC 61000-4-11 the system must be switched of for a minimum of 5 seconds before it is switched on again to guarantee a stable state for the power supply. Connecting and removing the GPIB cable should only be done while every device is switched off.

Three network connectors can be found at the rear of the box. They are named ETH0, ETH1 and ETH2. The network must be connected to one of the ports. For easy access the ports are preconfigured:

- ETH0: Address 192.168.1.84, Netmask 255.255.255.0, Gateway 192.168.1.1
- ETH1: dhcp (configuration is retrieved from a dhcp server)
- ETH2: Setup is done via front panel

The user may login to the system using telnet (this is possibly removed in the future) or ssh. The default password for the root account is “e488s”. It is also possible to connect a keyboard and a monitor to the rear side of the gateway. The user may also use ftp to transfer files to or from the system.

1.1 Front panel configuration

If the front panel is not used for input, the status of the system is shown. The screen changes every five seconds. Currently the status display consists of seven pages.

The four buttons on the left side of the display are used to navigate through the setup. The buttons are unnamed but navigation can easy be understood. There is an up-, a down-. a left and a right-button. The left and right buttons are used to cycle through the setup pages and if a page was selected, they are used to switch from one parameter to the next one. The up and down buttons are used to select a setup page and to modify numeric parameters. Each setup page contains additional help text for easy configuration.

1.2 Starting the manager and optional startup parameters

The GPIB manager build into the E488S system is started automatically while the system is booted. There is no need to modify the startup parameters. This chapter describes the options and parameters that may be entered if the manager is used as a standalone product.

If a configuration `/etc/config/aksgpib.cfg` file do exist, the manager could simply be started by entering the name without any additional parameters:

```
devb-gpib &
```

A great number of optional parameters may be used to modify the behaviour of the manager. Like any other QNX command, the user can get a short description of the command line arguments with the use command:

```
use devb-gpib
```

Several command line arguments can be found within the configuration file also. A command line argument overwrites an entry within a configuration file.

The following table contains a description of all the optional parameters

Switch	Parameter	Description
-u	`name`	The parameter defines the basename of a configuration file. If only one gpib-interface is installed in your system, there is no need to change the basename (normally `/etc/config/aksgpib.cfg`). If there is more than one interface installed, it is necessary to define a configuration file for the second interface because the information contained in the standard configuration file does not match the second hardware. Do not append "`.cfg`" or "`.NODE`" to the basename, this is done automatically by the software.
-v	`Bitmask`	This switch defines a new verbose or debugging level. The details of the bitmask are described within chapter 4.6 (see verbose level). The parameter may either be given decimal or hex.
-n	`Name`	This switch defines a device name of the manager. The standard name used by the manager is "`/dev/gpib`". If only one interface is installed into your system, it is not necessary to define a new name. If the manager is started twice to run a second interface, a new and unique name must be given.

-a	Nnn	ISA-Version: The parameter defines the I/O address used by the software to communicate with the gpib hardware. The address must be given in hex without a leading 0x. The default address used by the software is 308. PCI-Version: The parameter defines the PCI base that should be used. If more than one PCI card is build into the system, this parameter must be used to define which one should be used.
-q	nn	The parameter defines the IRQ used by the software for interrupt-driven communication. Currently there is only need for an IRQ if automatic service request recognition by the gpib software is needed. The default IRQ line used by the software is 7. If 0 is given, the IRQ support is disabled. This parameter is ignored by the pci-version of the manager, because the pci version directly gets the information from the system.
-f	n	The software is able to autodetect the TNT488.2 Chipset by National Instruments®. The following manufacturer codes are defined: Code 0 means INES® IEEE488-Interface and also means autodetection of National Instruments® Interfaces which are equipped with a TNT488.2 Chipset. Code 1 means AKS GPIB-Interface Code 2 means National Instruments® GPIB-Interface equipped with a TNT488.2 Chipset (if autodetection does not work). Code 6 means National Instruments® GPIB-Interface equipped with a TNT488.2 Chipset that is build after 1994. This chipset is normally always autodetected. Code 14 selects One-Chip-Mode in combination with a TNT488.2-Chipset. This Mode must be used if High-Speed-Mode is selected. The high-speed-FIFOs used within the TNT488.2-ASIC are only supported if One-Chip-Mode is selected. Transfer-rates more than one MB are possible if One-Chip-Mode is selected.
-x		The switch must be used to start the driver in non-CIC-mode. Because only one computer connected to the GPIB must be the CIC, all other computers or possibly a QNX-Computer acting as a device must be defined as non-CIC.
-z		If the driver runs in client-mode and an error or timeout occurs, the chipset is reset and initialised to a stable state if this switch is used. Normally this is not necessary.

-h		The switch defines the transmission mode. If set to yes, high-speed mode is enabled in combination with a TNT4882-Card. Because high-speed mode results in communication errors sometimes, it is switched off by default. One-Chip-Mode must be selected in combination with a TNT4882-Card to make high-speed mode available.
-l		The switch selects additional circuitry within the TNT4882 that increases the time NRFD or NDAC must be unasserted before the TNT4882 responds to the unassertion. This effectively slows down the TNT4882 handshake for a few devices that do not meet the IEEE4882.1 standard. The switch is independent to the `-h`-switch and both may be combined.
-b	`nnnn`	The parameter defines the length of the communication buffers used by the software to receive and transmit data. The default value for the buffer length is 2048 bytes. The buffers are automatically adjusted if more data is requested during a transmission. Using this parameter the default can be modified.
-m	`nn`	The parameter defines the own address used by the driver to communicate with the bus. The default value for the own address is `0`.
-p	`nn`	The parameter defines the priority used to run the driver. The standard process priority, which is used to run user applications, is 10. A lower priority level means lower process priority. If a priority level is used that is much too low, the manager runs very slowly. If the level is much too high, the driver consumes a great amount of system time. Level 10 is a good choice.
-d	`nnnn`	The parameter defines a minimum delay between two command sequences transmitted over the gpib-bus. The delay must be given in milliseconds. The overall performance is reduced by such a delay but sometimes the units connected to the bus are unable to communicate at high speed (as defined by the standard). If the equipment you are using contains such devices and if in such a case these devices produce overrun-conditions, it is a good choice to define a general delay by using this parameter.

-r		Some gpib-chips are a little bit buggy. Such a chip receives an additional byte after the end sequence has terminated the transmission. If such a byte was not processed by the driver, the driver receives the byte in the first position of the following transmission. Normally the software contains some code to solve the problem. It checks for an additional byte after the termination of a message. To disable the advanced-read functionality this switch must be used.
-c		If this command line switch is used, abort I/O was sent after every transmission. Sometimes this feature becomes very useful if you try to connect such buggy devices, which continue to listen even if they are unaddressed with UNT. Often these kinds of devices are set to unlisten if they get the abort I/O message.
-s	`nnn`	The option is used to insert an additional delay between two status checks after a command was send to the bus. This may reduce timeout problems with older hardware. The default is set to `0`. The value must be given in milliseconds. The switch corresponds to the setup-parameter `CSRC_delay`.
-t		The tlog is switched on.

2 Configuration

During start-up the GPIB manager build into E488S reads in a configuration file that can be modified either by using the socket interface or with an editor (vi for example) while logged in to the system. This chapter describes the details of the configuration file. Within the file-system the file is stored at /etc/config/aksgpib.cfg. A backup of the configuration can be found at /etc/config/aksgpib.default. If the administrator destroys the configuration file or any strange destructive event occurs, the backup may be copied manually to the standard file.

There is a function build into the GPIB manager to regenerate the configuration file based on the current settings. It is also possible to instruct the manager to write a new configuration file during every system shutdown. If the file is generated automatically by the GPIB manager it a great amount of comments are created also that contain detailed descriptions of the settings. Of course, user comments about why the things are defined can't be written. If the user has added own ideas encoded into comments it is recommended to not switch on the automatic regeneration of the file to avoid loosing the ideas.

The syntax of the configuration file is even simple:

- Comments may appear everywhere, they do begin with a hash "#" and they end at the end of the line where they do appear.
- Global settings may appear everywhere outside individual device configurations.
- 31 Devices are created automatically, there is no need for a definition of these devices, but it is allowed to define them to overwrite the default settings.
- Up to 100 custom named devices can be defined, it is allowed to define custom devices that contain the same target address. It is possible to define different settings for the same device using this feature.
- It is possible to define devices with no target address for generic use.

2.1 Global settings within the configuration file

Parameters defined globally within the configuration file are used either to modify the general behaviour of the GPIB manager or to define general device defaults that are applied to the default devices during their first creation. The following table contains the details for the global setup parameters. Commands are not case sensitive.

The following listing shows a typical global configuration section:

```
License               "long string"  # The license key given in hex
Manufactorer          TNT4882_ONEC   # if autodetection did not work
debug                 0x00008e       # not to high
delta_timing          Yes            # Verbose messages are shown with
                                     # delta timing
Admin_name            "/dev/gpib"    # Device Name for this Manager
I_am_CIC              Yes            # I am the controller
priority              10             # this may change during runtime
scheduler             FIFO           # FIFO, ROUND_ROBIN or ADAPTIVE
Advanced_read         No             # only for compatibility with
                                     # older cards
IB_delay              0              # delay between commands send out
My_Address            0              # this is my talk and my listen address
Recover_Timeout       5000           # currently not used
CSRC_delay            0              # msec sleep time while waiting until
                                     # device ready
My_Delays             BUSY_LOOP      # we are waiting this way (BUSY_LOOP
                                     # or TIMER)
Take_Control          TCA            # synchron (TCS) or asynchron (TCA)
Take_Control_Delay    0              # wait after transmission before
                                     # raising ATN [msec]
Store_Setup           No             # don't regenerate this file on exit
Socket_Gedit_allowed  Yes            # Global modifications are allowed
                                     # while connected via Socket
TCPIP_Passwd          No             # request passwd with IP comm
TLOG_verbose          0x000fff       # level to be used in tlogs
TLOG_before           100            # number of lines to save before trigger
TLOG_after            5              # number of lines to save after trigger
TLOG_trigger          0x000000ff     # event that do the trigger
TLOG_device_mask      0x0000         # devices that are monitored
TLOG_line_bits        0x00000000
TLOG_fmask            "/tmp/tlog"
TLOG                  Yes
```

Table 1: Global configuration Parameters

The details of the global Parameters are explained on the following pages. If there are several options for the parameter, they are separated by a pipe sign. Hexvalues must be written with a leading 0x.

```
License "String with 64 bytes"
```

The license consists of a 64 string with a length of 64 bytes. It is hexadecimal encoded. A valid license must do exist within the configuration file If there is no valid license key found, the gpib manager runs in demo mode only for a short period.

```
Manufactorer TNT4882_ONEC | TNT4882 | NEC7210
```

If there is no manufacturer defined, the GPIB manager tries to autodetect the GPIB board installed. E488S is equipped with a high performance chipset from National Instruments that is autodetected, but is may also be defined by name “TNT4882_ONEC”. If TNT4882 is used instead, the manager uses other methods to access the chipset. The NEC7210 is an supported but obsolete chipset.

```
Debug Hexvalue
```

The command defines a bitmask for debugging output generated by the driver on a connected console. See Appendix A: Verbose levels of the GPIB Manager for further information. Because the output generated by debug can only be seen on the console, it is better to use the TLOG to create output of the program and GPIB flow

```
Delta_Timing Yes | No
```

Verbose and TLOG output may either be shown with absolute timing (event time since start of system) or with delta timing (time elapsed since last event).

```
Admin_name "/etc/gpib"
```

The command defines the name where the GPIB manager creates an entry within the filesystem. This should never be changed.

```
I_am_CIC
```

The default behaviour is to start the system as the controller in charge (CIC), but it is possible to start the system as a standby controller.

```
Priority Value
```

The command is used to fine-tune the priority of the driver. The default priority for user tasks is defined to be 10. If you raise the priority of the GPIB, the GPIB requests are processed faster, but possibly this slows down your system. It is not possible to lower the priority below 10. If a priority lower than 10 is defined, the priority is not changed.

```
scheduler FIFO | ROUND_ROBIN | ADAPTIVE
```

The default scheduling policy of the drivers threads is FIFO. This means a GPIB command is completed without pre-emption of another task at the same priority level. You may change this to ADAPTIVE or to ROUND_ROBIN.

```
Advanced read Yes | No
```

Within E488S this parameter should never be changed and left off (or No). It is used for compatibility with older GPIB cards.

```
IB_delay Milliseconds
```

If strange equipment is connected to E488S, it is possible to overrun the target devices if the whole bus access is to fast (even this should never happen because of the management by the bus). The command can then be used to instruct the manager to wait for the number of given Milliseconds between two accesses to the bus.

```
Recover_Timeout
```

The command is reserved for future use.

```
CSRC_delay Milliseconds
```

If the manager is waiting for any GPIB command send out to be completely processed it makes still nothing, only waiting. Waiting and checking the status is done within a busy loop. For general using busy loops within a multitasking OS is not a good idea. It slows down the whole system. Within the embedded system E488S this isn't a real problem because there is nothing else to do while the GPIB is blocked, but if additional jobs are implemented, this could become a problem. To prevent running into trouble using this command the Manager can be instructed to yield to any other jobs currently waiting. This is done by defining a waittime (one millisecond should be enough). But the resulting problem is a possible slowdown of the GPIB throughput.

Leave this parameter untouched for now (default is zero) to get highest GPIB speed.

```
My_Delays BUSY_LOOP | TIMER
```

Often very small delays are needed while processing data to not overrun the I/O chipsets (shorter than 200 nanoseconds). No realtime operating system (even QNX) is able to provide such small delays. If such a delay is requested, you will get a much more bigger delay (in the range of 100 microseconds, which is 100000 times bigger) which results is a drastic slowdown of the whole GPIB throughput. To avoid this, the manager didn't use the system delays; instead it generates such delays by running busy wait loops while checking CPU counter registers (there you will get a nanoseconds resolution). This guarantees the maximum throughput. But there isn't light everywhere. Busy-waiting slows down the host system (E488S in this case) itself. Because within the manager busy waiting is only done with a maximum of 200 nanoseconds, we do ignore the negative effects. If there are any reasons why we don't want this, we can instruct the manager to use the system timer instead.

```
Take_Control TCA | TCS
```

If E488S tries to raise the attention management line on the bus, it not simply modifies the line without looking what's going on currently on the bus. Instead it sends the internal message Take Control to the Chipset. There are two ways of doing this, either synchrony (TCS) or asynchrony (TCA). The default is TCA.

```
Take_Control_Delay Milliseconds
```

Some devices (mostly very old ones) do not like it, if ATN was raised immediately after finishing a data transfer. There is a chance to crash such devices this way (we have seen this in the past). If you do have such kind of devices within you system, you may define a millisecond delay that instructs the manager to wait after every data transmission before raising attention. The default is switched off (zero).

```
Store_Setup Yes | No
```

The Manager is able to regenerate the configuration file on exit. A newly created configuration file contains all the changes made while the manager is running. It is up to the user to switch this feature on or off, the default is off (no).

```
Socket_Gedit_Allowed Yes | No
```

While connected to the system via the socket interface it is possible to change global system settings and device configuration parameters. Using this command it is possible to switch off this option. If a user tries to modify settings while the option is switched off, he gets an error. If switched off, this option can't be modified while the manager is running. A complete system restart must be done.

```
TCPIP_Passwd Yes | No
```

The command is currently not implemented. Future version do contain optional password checking for the socket interface.

```
TLOG_verbose          0x000fff
TLOG_before           100
TLOG_after            5
TLOG_trigger          0x000000ff
TLOG_device_mask      0x0000
TLOG_line_bits        0x00000000
TLOG_fmask            "/tmp/tlog"
TLOG                  Yes
```

The TLOG family of setup commands are described in detail in Advanced transmission analyses.

2.2 Device specific settings within the configuration file

The device specific section of the configuration file is optional. It may contain definitions for the 31 unnamed standard devices and definitions for up to 100 custom devices.

To redefine a standard device simply an entry is created where the name given on the Define_Device command line is identical to the GPIB address of the corresponding device. These kinds of devices are addressed by numbers during data transfer. The standard devices do exist in every case, there is no need to define them; definitions for these devices should only be made if the configuration of the device differs from the defaults in a non-compatible way.

Custom devices are defined by giving a unique name on the Define_Device line. The name given here is case sensitive. It is allowed to define several custom devices that do access the same GPIB target. This would make sense for example to create different setups. It is also possible to define custom devices, that don't contain any address definition. These devices may be used for generic GPIB commands or for cooked mode access.

The following listing shows an example for the redefinition of a standard device:

```
Define_Device 1
    Name                "Controller 1" # for identification only
    Addr                1              # the device uses this GPIB address
    SAddr               -1             # no default subaddress used
    Com_mode            RAW            # transmission automatically done
    EOS8                Yes            # we use eight bit EOS
    EOSBYTE             10             # this is our EOS-byte
    EOT_OUT             EOIEOS         # send EOS with EOI set as last char
    EOT_IN              EOIOEOS        # incoming stream termination either
                                       # EOI or EOS
    ITimeout            10000          # our timeout during receive [msec]
    OTimeout            10000          # our timeout during send [msec]
    ABORTIO_after_send  No             # no ABORT IO after transmission
    Timeout_CRST        No             # chip reset on timeout
    TCA_after_send      Yes            # take control async after send
    wait_priority       0              # priority not changed during wait
    twc                 0              # don't wait between I/O operations
    SPOLL_Type          ADR_SPE        # either SPE_ADR or ADR_SPE
    Buffer_mode         OPENSPECIFIC   # either shared or openspecific
Define_End
```

The example shown above contains no special settings. The settings used here are mostly useful. Defining the name could be helpful to identify the device later. The socket command `?devices` shows the name. The communication mode for the default devices should be left `RAW`. It makes sense to define a useful value for the timeouts. The default of 1000 milliseconds should be big enough for most devices; possibly to big. Timeouts should be defined big enough to guarantee a transmission to the device could be finished regardless of the data transferred. Bigger

timeouts did not help; in the case of a transmission error, a big timeout possibly only retards the error detection.

The following example shows the definition of a custom device. The device defined here provides a gateway to the low-level command set. The device did not contain any address definition because the complete addressing must be done with the commands send to the device. It is defined to be of communication mode IEEE488, which means data send to the device must contain low-level commands.

```
Define_Device "low_level"
    Name            "low_level"      # for identification only
    Addr            -1               # this is an unspecific device
    SAddr           -1               # no default subaddress used
    Com_mode        IEEE488          # low level IEEE488 command
                                     # interpretation
Define_End
```

At least a small look into the definition of a port device. The following example shows the definition of a device that can be accessed via telnet on port 4880:

```
Define_Device "unspec_portdev"
    Name            "unspec_portdev" # for identification only
    Addr            -1               # this is an unspecific device,
                                     # address must be given each time
    SAddr           -1               # no default subaddress used
    Com_mode        COOKED           # high level interpretation of data
    TCPIP_Port      4880             # device can be accessed via IP using
                                     # this port
    TCPIP_Welcome   ENABLE           # a welcome message is shown if a
                                     # user logs in
    TCPIP_Editmode  ON               # command line history and editing
                                     # enabled
    IO_Settings     TARGET           # use settings of addressed target
Define_End
```

The device is defined to be of the type “cooked”, which means every data send to the device is interpreted by the GPIB manager. A command language is defined, that must be used to do anything useful. The language is described in detail in Using the Socket Interface. To be more user friendly the device provides command line editing if a user logs in and it shows a welcome message. The device is defined to use the settings of an other device while talking to a target.

There are thousands of possible configurations. The Appendix B: Example configuration file contains a complete configuration file, which should be read to become familiar with the things. On the following pages detailed description of every possible device setup entry is given.

```
Define_device Number
Define_End
Define_device "Name"
Define_End
```

Every device definition starts with Define_device and ends with Define_End. If the parameter given of the Define_device line is numeric, the definition is used to overwrite an existing definition. If there is more than one definition with the same number, the latest definition overwrites any previous. If a name was given, it must be enclosed in double quotes. It is not allowed to define a name more than once. Names are case sensitive.

The numbers or names defined with Define_device are later be used to access the device either via local access, NFS or the socket interface.

```
Name "description"
```

The command can be used to assign a name to a device. The default name for a device is identical to the parameter given with the Define_device command. The name given here is for information purposes only. The name is not used by the system.

```
Addr Address
SAddr Address
```

The command assigns a GPIB target to the device. If a numeric device is defined, the number may be omitted or must be the same as the one given with the Define_device command. If a symbolic name is used, the number may also be omitted or defined as –1, in which case an anonymous device is defined that must be given an address for every access. Anonymous devices should also be used for socket type devices combined with device forewarding.

The SAddr command is used to define a secondary address for a device. If a secondary address is defined for a device entry, secondary addressing is done for every access to the GPIB target. If no secondary address is defined, primary addressing only is used while accessing the GPIB target.

```
Com_mode RAW | COOKED | IEEE488
```

The definition of the communication mode deeply modifies the behaviour of a device. An error made here may result in very strange effect while communicating with the device. It is suggested to leave the standard devices (0-31) defined to be RAW; socket devices should be defined as COOKED. All other devices can be defined accordingly to the usage of the device. The meaning of the definition is as follows:

- RAW devices don't modify data sent or received. If data is written to such a device, the GPIB manager directly transfers the data to the GPIB target device. If data was read from such a device, the manager initiates a read request to get data from the GPIB target.

A read transfer is done at the moment a client requests data from a target device. A new read request was initiated if the complete data of the last read request was delivered and a client request additional data. Transferring data from a GPIB target is done automatically without any user invention). If data was written to the device, the manager waits until the write request was completed (watching for the occurrence of an EOS character within the data stream or using a counter). Using RAW devices via NFS may result in several problems. There is a special configuration parameter for NFS accesss.

- COOKED devices do provide a command language that is interpreted by the manager. The command language is described in detail in Commands accepted by the interpreter. With RAW devices data is never transferred automatically to or from a GPIB target. Transfers are done due to the commands send to the device. If a receive command is initiated, the manager transfers data from the GPIB target into the receive buffer. The data can then be retrieved with a read command.

- IEEE488 devices do provide a command language to do low level GPIB programming. Everything must be done by the programmer. The language is powerful enough to do even very strange things. It's one of the best ways to shoot into the own knees.

```
EOS8
```

The command enables eight-bit EOS recognition. The default is to look only for the low seven bits to detect an end of stream condition. The EOS8 definition is only used if EOS detection is enabled for incoming bytes.

```
EOSBYTE Value
```

If EOS detection is enabled, the manager looks for the EOS character to detect the end of a transmission. The default EOS character is ASCII 10 (newline). The newline character is a convention that is accepted by most GPIB devices as EOS character per default, but sometimes there is a need to change this (ASCII 13 for example).

```
EOT_OUT NEOINEOS | EOINEOS | NEOIEOS | EOIEOS
```

The command defines the behaviour of the driver to mark the end of a transmission for the receiver. Most GPIB devices do detect the transmission end based on the EOI line or the EOS character or both. Only rare devices do work with byte counting. The command provides the following options:

`NEOINEOS`	No EOS character is send, no EOI line is raised. The receiving target must know the number of bytes to receive.
`EOINEOS`	The EOI line is raised while the last byte is transferred, no EOS character is transmitted.

NEOIEOS	The EOI line is leaved untouched, but EOS is transmitted as the last character
EOIEOS	The EOS character is transmitted as the last byte combined with EOI raised.

```
EOT_IN EOI | EOS | EOIAEOS | EOIOEOS | NONE
```

The command defines how the manager detects the end of an incoming stream while receiving data from a device. For most cases, EOIOEOS would be the best setting. The following options are allowed

EOI	The driver only looks for EOI raised to detect the end of the incoming data.
EOS	The driver only looks for an EOS character. The status of the EOI line is simply ignored.
EOIAEOS	The end of an incoming stream must be the EOS character with EOI line raised additionally.
EOIOEOS	The end of an incoming stream must be either the EOS character or any other character combined with EOI raised.

```
ITimeout Milliseconds
OTimeout Milliseconds
```

The command defines the timeout while data is received (incoming timeout) or data is send (outgoing timeout). The timeout counter is started at the moment the manager begins addressing the target and runs until unaddressing of the target is done after a successful transfer of the data. If the data could not be transferred until the timeout time has been elapsed, the manager generates a timeout error condition. The value should be defined big enough to ensure complete transmission in every case, but it should not be defined to big to avoid unnecessary delays if an error occurs. A timeout value of 500 to 1000 milliseconds should be ok for most devices.

```
ABORTIO_after_send Yes | No
```

If the command is switched on, the GPIB manager sends an interface clear after every transmission. This should normally be switched off. It should only be switched on, if the device is known to be problematic.

```
Timeout_CRST Yes | No
```

If the option is switched on, the GPIB manager completely initialises the GPIB chipset if a timeout occurs. This option should be switched on if a device is known to be problematic in the case of a transmission failure. We have found in the past that there are some devices, that do work mostly all the time, but if there is an error with the device, the error is very coarse, locking the whole GPIB. For these devices, we have switched on the feature described here. Mostly this recovers the device and brings the GPIB back to life. The feature should normally left off, because “normal”

timeouts did not need such a brute force recover and because the chip reset (that also initiates an IFC) needs a great amount of time.

```
TCA_after_send Yes | No
```

If the GPIB driver acts as an controller, this command defines if the controller asynchronously takes control after a successful transmission. If the parameter is switched of (No), control is taken synchronously. The default is asynchronous.

```
wait_priority Level
```

If `twc` is defined to be non zero, there is a chance that the driver runs into a timeout because the scheduler did not reschedule the driver fast enough (under heavy load). This means a timeout occurs even if the transmission was successful. To overcome this behaviour, the driver is able to raise its own priority before waiting, lowering priority to the old value after waiting was done. This results in coming back soon enough, not running into a timeout. The level should be defined higher than the default drivers level.

```
twc Milliseconds
```

Some very old devices that do not conform to the IEEE 488 are not able to drive the handshaking protocol at full speed. For these devices, twc provides a method to slowdown the communication. Twc can then be used to define a time given in milliseconds, the GPIB manager waits between sending two commands. This option should be left off (set to 0) for most devices.

```
SPOLL_Type ADR_SPE | SPE_ADR
```

The command is used to define the ordering of commands to make a serial poll. The standard ordering is to first address the device, initiating the spoll thereafter. Some devices are not able to respond, if the command ordering is done this way. For these devices the command order can be switched, that means the spoll command is send first and the addressing Is done second.

```
NFI_mode yes | no
```

If this switch is defined for a device, the GPIB manager makes some additional assumptions about transmissions. If a device is accessed via NFS, sometime the client requests a second chunk of data if the first read was not fulfilled completely, which results in a second read. Lets look on an example:

```
n=read(fd,buffer,100);  // GPIB manager gets data from the bus
                        // (10 bytes for example) and responds with
                        // the 10 bytes
```

Internally the NFS thinks “uh, oh, I have requested 100 bytes, but only got 10 bytes, let us request addition 90 bytes”, which results in a hidden read done by the NFS itself

```
cnt=read(fd,buffer,90); // GPIB manager only sees a second request
                        // getting new data from the GPIB
```

This is not what we want. If the client system behaves this way, the manager should detect the second request as part of the first read. It then should not get more data from the gpib, instead it should respond with zero bytes, which means “there is no more data” for the client.

If NFI mode is switched on, the GPIB manager behaves this way. This works quite well, but there is a problem. If a client really reads makes a second request with a number of bytes that is identical to the number of bytes not delivered by the GPIB manager for the first request, the GPIB manager did not generate a new GPIB transfer. It only generates a new transfer (GPIB read), if the read combination described did not happen. You may see, if NFI mode is switched on, there is a rare chance (with some strange programming on the client side) to not get whats really requested. But if access to the GPIB manager is done with NFS, most problems produced by the NFS subsystem on the client side are solved with this switch set to on.

```
Buffer_mode OPENSPECIFIC | SHARED
```

This is a switch made for RAW mode access to a device. If a client opens a device, the default behaviour is to create an own communication buffer for this client. If a read request was initiated by a client, a chunk of data was transferred from the GPIB to the buffer. Further read request done by the client are answered by the GPIB manager with data from the internal buffer until the buffer is empty. Then, on the next read request, new data is transferred from the GPIB to the buffer. The content of the buffer gets lost, if a client closes a connection. Sometimes this is not what we want. If there is more than one client connected to the same device, sometimes we want to use a buffer that is shared by all devices. In this case, data is transferred from the GPIB into the buffer with the first read request, but read requests from different clients (different open calls) are responded from this one buffer thereafter. If the buffer gets empty, new data is transferred into from the GPIB.

It makes sense to create a custom device with such a feature, which can be accessed by several clients. It is not suggested to change the behaviour of the standard devices this way (even it works).

```
TCPIP_Port Portno
```

If a number is given here, the device is glued to the socket interface. Devices with a port number defined can be accessed via TCP/IP. There are two such devices defined within the default configuration (ports 4880 and 4881).

```
TCPIP_Welcome Yes | No
```

If the option is switched on, a welcome message is send to the socket if a client connects to the port. This option should be switched on for ports that are accessed manually to inform the user (telnet interface).

```
TCPIP_Editmode On | Off
```

If the edit mode is switched on, the socket interface provides some kind of command line editing on a socket. Several different terminal types are automatically detected. Cursor keys, delete, insert and even a command line history (up down keys) are provided. If edit mode is switched on, the socket interface also changes the Carriage Return character (send by the return key) to Newline (which is used to separate commands).

```
IO_Settings OWN | TARGET
```

This also is a socket option. Normally if a client connects to a socket device, any further GPIB-transfers are done with the settings of the socket device, but sometime this is not what we want. Instead, the socket interface should use the settings of the device that is addressed during a data transmission. To instruct the socket interface to use the client parameters instead of the socket parameters, the IO_Settings should be set to TARGET.

3 Using the File Interface

The GPIB driver provides a file interface for easy access by any application program. The E488S GPIB/Ethernet gateway additionally exports the file interface via NFS to the network. If access to the GPIB driver is done with QNX internal QNET networking or local to the system, the things are simple and nothing special must be done. If NFS is used to access GPIB, possibly the things become strange because the NFS implementation on the client side possibly contains caching algorithms that must be either switched of special programming is needed to outwit the caching. Sometimes it becomes nearly impossible to access the GPIB via NFS because of the caching, we suggest using the socket interface to the GPIB in these cases.

Everything written within this chapter (set up, programming, access and so one) is valid for both the gateway and the standalone GPIB.

The GPIB creates an own directory within the file system named /dev/gpib. Within the directory, the manager creates the default devices (0-31) and custom devices as defined within the configuration file. The device entries created by the manager are of type “file” with a length of 2 GB. We have found creating a file type entry works much better with most applications than creating an entry of type “device”. File type entries can be accessed with NFS from nearly any OS, device type entries may only be accessed local or if the remote machine uses QNX also.

The features, interface and functionality provided by each file within /dev/gpib depends on the definition within the configuration file. Every file may support one of three different interfaces:

- RAW: data send to such a file is send to the corresponding GPIB device without any translation. The device entries /dev/gpib/0 to /dev/gpib/31 are of such a type.
- COOKED: data send to such a file is processed by the interpreter build into the manager. A command language that is described in detail in Chapter 4.6 is used to process all the data written to such a file.
- IEEE488: data send to such a file is processed by a low-level interpreter build into the manager. The low level interpreter directly accepts GPIB low-level commands encoded with standard GPIB mnemonics. This language is described in detail in chapter 5.3

If access to the manager is done using NFS, the client must mount /dev/gpib using the mount syntax of the client used. For example:

```
mount –t nfs 192.168.1.84:/dev/gpib /gpibgw
```

In theory it is even simple to access a device using the file interface and if the GPIB device connected to the system is a well behaving newer one the theory gets real and access to the device using the file system is a simple walk. But the things possibly may become strange if the device did not really conform to IEEE488 or there is no end of stream definition while receiving data. Finally, yet importantly, the things possibly become tricky if NFS is used to access the GPIB manager with a caching client. The following chapters do contain the details to access GPIB devices using the file system in simple and strange configurations.

3.1 Access to the standard devices in raw mode

The standard devices are numbered from 0-31. These devices may be accessed by reading or writing from /dev/gpib/0 to /dev/gpib/31. The devices are created by the manager even without a configuration file entry. If a read is done from such a file (device), the manager completely manages devices addressing, transfers data until EOI or EOS was seen, de-addresses of the device and sending back the data received to the process that is reading the file entry. Sounds pretty simple and really it is. If data is written to a device-file, the manger again makes all the addressing, transfers the data to the device (including EOI/EOS) and then de-addresses the device. The same, it's as simple as it sounds.

Transmitting data possibly only gets complicated if the device did not accept or send end of stream definitions (like EOI or EOS). This means counting only is used to transfer data to or from the device. The manager provides several techniques to get informed about the length of data that must be transmitted. Of course, the easiest way to talk to such devices is by using cooked device entries.

The manager uses lseek commands to get informed about the number of bytes that should be transmitted. The include file `gpib_devctl.h` contains definitions for special lseek commands. To define the number of bytes the driver should send, the macro GPIB_LSEEK_WLEN is defined. To inform the gpib manager the following sequence must be send:

```
lseek(fp,GPIB_LSEEK_WLEN | (number_of_bytes<<8),SEEK_SET)
```

To define the number of bytes that should be received. The macro GPIB_LSEEK_RLEN is used. In both cases the length parameter must be shifted by 8 bits to the left, which is identical with multiplying by 8192. This is done to outwit a possible caching.

The include file `gpib_devctl.h` defines a great amount of lseek macros to perform several tasks, for example to send a trigger, to initiate device clear and so on. Additional examples on how to use the file interface can be found be looking into the example program `rwgpib.c` that can be found at Appendix D.

If a device is defined to be of type counting (no end of stream definition) or of type EOI only (no EOS character at the end of a data stream) and no lseek command is used to define the length of bytes that should be transmitted, the manager sends out the data stream on each write.

If data was read from a device that has no end of stream definition and no lseek command is used to define the number of bytes to read, the manager reads as many bytes from the device as are requested by the read operation.

3.2 Reading and Writing Data using NFS

If access to the GPIB manager is done using NFS the things possibly become complicated due to caching on the client side. If a client reads data via NFS, the underlying NFS manager did not really receives or transmits the amount of data that was requested by the client. Instead, all data was transferred in blocks of 8 KB. Because of this behaviour, we do shift the parameter part of every lseek command by

8 bits to the left. Unfortunately, this did not solve all caching problems. If we do use the same lseek command twice, the NFS layer on the client side possibly ignores the second request. An additional trick was implemented into the GPIB manager to solve this problem. The manager provides a split mode for lseek commands. While using split lseek commands, a random like value (or a counter) may be added to the lseek value that guarantees flushing of the caches. If split lseek commands are used, one lseek command must be divided into several lseek commands with a random part contained within the lower bits. An example on how to use this technique is shown in appendix d.

For general it should be avoided accessing the GPIB via NFS using the standard devices (0-31). The better way is to define named custom devices for every device that is used. For these custom devices, NFI_mode should be switched on (see configuration file example in appendix A or definition of configuration parameters).

Studying the example shown in appendix D should clarify the different ways that could be used to get a stable NFS access running. We have tried this with QNX, Linux and Solaris. In every case, there is a way for a stable data transmission, even if caching of the client side complicates the things.

3.3 Accessing cooked and IEEE488 type devices

Data send to devices that are defined to be of type cooked or IEEE488 are not directly transmitted to the target. For both device types, the GPIB manager contains a special interpreter, and what's really done is sending commands to the manager. The command language is described in detail within the socket chapter. In every case a user (or program) can not simply read from such a device to get some data. A command must first be send to the device, that is interpreted and processed by the manager and the result can then be read back.

The two interpreters for both device types are line oriented. This means every command send to such a device must be terminated by a newline character.

4 Using the Socket Interface

Within the configuration chapter, you will see several ways to set up a socket interface for easy communication either manual using Telnet for example or using a program. Within this chapter, you will see the details of the commands provided by E488S using the socket interface.

The basic set up of E488S defines two socket interfaces; one of them may be used for manual GPIB control (port 4880) the other one is defined to be used by program control (port 4881). Additional socket ports may be added by modifying the configuration file. There are no fundamental differences between these two interfaces. The port defined for manual control (port 4880) only provides a user-friendlier interface than the port 4881.

4.1 The User Friendly Socket Interface

The port defined for manual control (port 4880) adds some functionality to make it more convenient to use the socket interface. All the additional support options may be toggled within the configuration file. The standard settings do provide the following:

- A welcome message
- A command prompt “>” is shown if the interpreter waits for new commands to enter
- A limited form of command line editing (Backspace and/or delete keys are accepted to modify the entries made) that works in every case, even if the interface was unable to detect the Terminal Escape Sequences used by the telnet program.
- Automatic support for several Terminal Escape Sequences (auto detects ANSI and VT100) to provide enhanced command line editing (cursor up, down, left right, insert and delete)
- Acceptance of CR (this means the return key) instead of Newline (0x0a)
- A command line history with a length of 100 entries that can be used by the cursor up/down keys

A user may login to the port 4880 using any kind of telnet program from any kind of operating system and send commands to the GPIB bus or try anything to figure out how to use the equipment. The idea behind using this port is try the necessary steps for automatic communication using port 4881. Anything entered here may be directly ported to port 4881 communication. Port 4880 with its user-friendly interface is also very use full to set up the things, defining communication settings and to debug communication sessions (see the tlog family of commands).

There is a restriction to the interactive connection. It is not allowed to switch to binmode using interactive mode. However, this isn’t a real

problem because binary data can't be really transmitted or shown on a telnet screen.

4.2 The Socket Interface for automated Control

The default configuration defines port 4881 to be used for automated control for example to be included into programs or batch files. The following list describes the characteristics of port 4881:

- No welcome message is shown
- No command prompt is shown
- There is no command line editing, this means all control characters received are interpreted as part of the command or communication
- There is no support for Telnet Escape Sequences, all these characters are also interpreted as part of the communication
- Every command line must be confirmed with a Newline Character (0x0a). This character instructs the interpreter to parse and process the command

As you can see, port 4881 is not very user friendly but it is very machine friendly. So, try it first using port 4880 and code it into programs using port 4881.

While using port 4881 don't forget to terminate each command with a newline character (0x0a or "\n"). We have found that often programmers forget to add the newline character. There is an option (default switched off) to define binary transmission mode. If this option is switched on, the programmer must include a transmission count to instruct the driver how many bytes do follow. If binary mode is switched on, there is no need for a newline at the end of the command or data.

This characters instructs the receive loop of the communication system to begin processing the data. Why do we have used newline instead of the common character ";"? There is a very simple answer: Most GPIB communication devices do also use newline as an end message (EOS). So there is no need to additionally include a newline into data to send. The newline both defines the end of one command and it defines the end message for outgoing data. If there is any need to include a newline itself into a data stream, there are two ways. It may either be encoded using escape sequences or you may give a byte count to the interpreter so it is able to know the number of bytes that must be received from the socket before a command is processed (see further chapter). Removing any IP-transmission timing problems is the main advantage in doing the things this way.

4.3 Escape sequences and character recognition while using the socket interface

Each command send to the socket interface is processed by the interpreter build into the GPIB manager. A command consists of a

command token and optionally parameters. After successfully receiving a stream of data the interpreter first tries to find out the command. This is done by comparing the data received with the internal interpreter tables. All **commands are case insensitive**, even within this manual and the online help commands are shown with upper and lower characters for better readability. Commands may be shortened by the user as long as the interpreter is able to clearly identify the command. For example, the user may enter TLOG_tr instead of TLOG_trigger, but it is not allowed to use TLOG_, because the interpreter is not able to clearly identify the meaning of this command.

If binary transmission mode is switched off (this is the normal case), the interpreter is not able to include a newline character as part of the data transmitted (it defines the end message in this case). While using interactive mode, the interpreter is also not able to accept most of possible escape sequences as part of the data (control characters and escape sequences are interpreted directly for command line editing). However to include binary data into a transmission stream the user may encode such bytes using standard escape sequences. Every escape sequence begins with the character “\”. The interpreter accepts the following sequences:

Name	Sequence	Meaning
Backslash	\\	Sequence to encode the backslash itself
Newline	\n	A newline character 0x0a was inserted
Return	\r	A return character 0x0d was included
Backspace	\b	A backspace character was included
Tab	\h	A horizontal Tab 0x09 was included
Hexcode	\0xnn	The hexadecimal encoded binary value nn was included
Octalcode	\0nnn	The octal encoded binary value nnn was included

Table 2: Character Encoding

Binary transmission may be switched on with the binmode command. Usage of the binmode command is only allowed while not in interactive mode and a transmission count must be given in every case in binmode because the interpreter can’t know the length of the data to process.

If binary mode is switched of, the data transmitted back as a result is encoded also. The interpreter encodes all bytes less than 32 or greater than 127 with octal representation. It encodes the Backslash with a double Backslash.

4.3.1 Character counting while sending data and commands in non interactive mode

If the user wants to transmit data that contains binary bytes (less than 32 or greater than 127), the interpreter offers the option to include a counting

value at the beginning of the data stream. Binmode must be switched on to use this feature in non interactive mode. The command itself and any optional addressing values are not part of the counting. At the writing of this manual, this is only used with the “send” command. Behind the length parameter a “:” must follow, followed by the data itself. Any control characters within the data stream are ignored if the counting value is give. An example C-encoded :

```
write(sock,”send 4;7:DhelloX”,16);
write(sock,”send 4:DhelloX\n”,15);
```

While using interactive mode entering the data on a command line, the interpreter automatically adds a newline “\n” if the user hits the return key.

Both commands do produce the same result. With the first command the interpreter sees the counting value 7 and then after it has read in 7 bytes (not including the “:”) it begins processing the input without waiting for any newline and without processing any escape ore encoded sequences within the data stream. The second example uses the simple format. The interpreter gets the data and if any kind of escape sequence was found while reading, these sequences are transformed into binary form. Using the second format the interpreter begins processing the command after it has read in the newline character.

4.4 Results returned by the interpreter

For every command or statement send to the system you will get back a result code, that describes if the command could be processes successfully or if there are any errors. The result code contains several parts.

The first part is either `00:` or `01:`. You will get the `00:` response if the command could be processed successful. You will get `01:` if there is any error, found by the parser itself (misspelled command) or occurred during execution of the command.

The following resulting bytes depend on the `rcount` setting. If `rcount` is switched on, the number of bytes that do follow are send out encoded with five digits and the character “:” that follows. If `rcount` is switched off, no byte count was send.

If an error was found during parsing or has occurred during processing of the command, the error code, a comma and a suberror code or the error position (in the case of a misspelled command) is send out.

If the command could be processed successfully and the command produces any kind of result, the result is appended.

The last two bytes send out always (even in binary transmission mode) is a carriage/linefeed combination (“\r\n” or 0x0d, 0x0a).

Let us take a look on some examples. The following table assumes `rcount` to be switched on:

Command	Result	Meaning
Gulp	01:00004:20,0	Simply nonsense. Error code 20 means command not found
Tlog_	01:00004:21,0	Command abbreviation given to interpreter is not unique
Spoll 4	00:00003:24	Command executed successfully. Result returned was decimal 24 (spoll value). The length 3 in this case means there was a “\n” appended at the end by the spoll command itself.
Spoll 5,4	01:00003:7,6	I/O operation aborted (Error code 7), a timeout has occurred while waiting for data to receive (Suberror code 6)
Spoll 34	01:00004:22,6	Invalid parameter (Error code 22) at position 6 (address is invalid)

Table 3: Examples of results returned by the interpreter

4.5 Device configurations used by the socket interface

A socket interface and the corresponding port number are defined within the configuration file. Additionally to the port definition, such an entry also describes the communication settings that are used while talking to or receiving data from a device. Sometimes this is not what we want; instead, we like to use the settings of the device we are communicating with. If we do connect to port 4880 for example, the driver selects the device “unspec_portdev”, which is defined to contain the settings if anyone connects to the port 4880. The device entry for example contains definitions for the end of stream termination for outgoing and incoming data. The settings are defined to be useful defaults, but if anything special is needed what should we do? There are two options for a special kind of redirection. Within the configuration file the command `IO_Settings TARGET` may be used for the device. The other way is to switch the I/O definitions while connected to the port using the command `settings_copy on`. Both commands do produce the same result, while talking to a device (or receiving data from a device) the settings of the device that is addressed are used instead of the device that is defined for socket communication. The difference between writing the setting into the setup file and using the online command is the default behaviour. It’s up to the system engineer to use the appropriate setting.

4.6 Commands accepted by the interpreter

The number of commands accepted by the interpreter is growing, and because of this, it is possible that a brand new command will not be found within this manual. In every case, there is an online help build into E488S, which contains a short description of all commands accepted by the interpreter. To get the help text, the user simply sends “?” to the system.

This chapter describes all the commands currently available. The commands are grouped into chapters with different targets. Common to all chapters is the syntax used by the commands:

- Optional parameters are shown in square brackets.
- Each command begins with a statement and ends with a newline character.
- While in interactive mode the interpreter maps return (0x0d) to newline (0x0a).
- If a length parameter is given to the interpreter, no newline is needed.
- Addresses may be given either decimal or by device name (if a corresponding device was defined previously)

4.6.1 Commands to get help and to query health status

The interpreter provides several commands to show some help and to query the current help status. Each such command begins with a Question mark:

```
?
```

The command shows a short list of the other help commands available.

```
?hlp_errors
```

The command shows a detailed description of the error codes and sub error codes that are generated by the driver.

```
?hlp_dsyntax
```

The command shows some help about the encoding of data send to the driver or received by the driver if `binmode` is switched off.

```
?hlp_gpib
```

The command shows a list of all GPIB commands accepted by a cooked device (the socket interface for example).

```
?hlp_gsetup
```

The command shows a list of all commands that are used to modify global parameters and to add, edit and modify devices.

```
?hlp_lwl
```

The command shows a list of all low-level commands that are accepted by the low level device.

```
?hlp_results
```

The command shows the differences between all the success, error and counting values returned by the driver for all the combinations of result control switches.

```
?hlp_tlog
```

The command shows a list of all command that provide advanced debugging and analyses.

```
?hlp_vlevels
```

The command shows a list off all the supported binary encoded verbose levels. A binary or of the values is given to either the verbose or `TLOG_verbose` command. At the writing of this manual, the following output was generated. See Appendix A: Verbose levels of the GPIB Manager for further information.

```
Name            Value    Meaning
-----------------------------------------------------------------
DBG_HIGHRES     0x00001  output is done with a high res timer
DBG_GENERAL     0x00002  output general info
DBG_ERROR       0x00004  output general error conditions
DBG_IOERROR     0x00008  output I/O error conditions
DBG_TIMING      0x00008  output I/O timing info
DBG_IBCMDS      0x00010  output GPIB commands
DBG_FUNCTIONS   0x00020  output function calls
DBG_SOCKET      0x00040  output socket connections
DBG_FILEIO      0x00080  output resource manager file access
DBG_CHIPREG     0x00100  output register values of chipset
DBG_TRANSFER    0x00200  output transmitted data and commands
DBG_SETUP       0x00400  output configuration and setup changes
```

Table 4: Verbose levels

```
?devices
```

The command shows a list of all the devices that are defined. You will always see devices 0 to 31 (because these ones are defined by default). If additional devices are defined, they are also shown. Let us look on a typical list returned by the command:

```
00:00350: 0: 0 (Controller 0)
 1: 1 (Controller 1)
2, 3
 4: 4 (Keithley 196)
5, 6
 7: 7 (Simulator)
8, 9
10: 10 (HP300)
11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
23
24: 24 (HP7495A)
25, 26, 27, 28, 29, 30, 31
 4: Keithley_196
10: HP3488A
 4: Keithley_196_raw
 4: OS_Keithley_196
-1: unspecific
-1: unspec_portdev
-1: unspec_portdev_nw
```

The meaning of this list is straight simple to understand. 350 bytes are returned. The standard devices 0, 1, 4, 7, 10, 24 do contain own settings within the configuration and they are given a name for easier detection. These devices can't be addressed by the name because they are standard devices (to show this, the names are enclosed into braces). The other standard devices have no special definitions, they do use the system defaults. Additionally some named devices are defined. These devices can only be addressed by the name instead of the address. For example there are three different entries for a Keithley multimeter, each of them do address the same target, but the communication settings for each entry do defer. To get the differences the device entries may be queried using the command `Query_device`. At least you can see three different devices that do not contain an address. These entries are anonymous. This means while talking to such a device a address must be given additionally. Such anonymous device entries are used for NFS and for socket communication. The default for example is to define device `unspec_portdev` for socket communication with port 4880 and `unspec_portdev_nw` for socket communication with port 4881.

```
?connections
```

The query generates a list of all currently open connections. There are two different types of connections shown. File type connections, which are generated for example by connections via NFS and socket type connections. For a file type connection the system shows the target that is addressed using this file descriptor and the access mode. For a socket type connection the system shows the client IP address and the connection port. For both modes of access, the system also shows the complete transmission time for command and for data transfer (time needed by the GPIB). Lets look on an output:

```
1: file device 4, mode: RAW Times: 0-00:00:00.000026538 :0-00:00:00.053681562
2: connection from 192.168.1.39:51660 Times: 0-00:00:00.000000000 :0-00:00:00.000000000
3: connection from 192.168.1.43: 3482 Times: 0-00:00:00.000597026 :0-00:00:00.173053006
```

The first entry shows a program or script that is accessing the GPIB using the file interface either via NFS or local to the system. Only a very small amount of GPIB transfer time was needed (26 microseconds for commands and 53 milliseconds for data transfer). The second connection is of type socket and it was open only, but no data was transferred until now. The third connection, type socket also has transferred some data to the GPIB.

```
Health
```

The command sends back some health information:

```
00:00068:1057469331: online: 36941 sec, Session 2, 30
deg.C,Core Voltage OK
```

The first value behind the counting value defines the uptime of the E488S systems in seconds. The second value defines the connection time of this session. The third value shows the current session number (there is one other connection pending currently). The fourth value shows the temperature within the system. And the last value should always be OK, because if the voltage isn't ok, the system didn't work. Possibly the last value shows some details in the future if voltage transient recording was implemented.

```
?myadr
```

The command shows the address used by E488S on controller mode.

4.6.2 Modifying global parameters for current connection

There are several parameters that do act globally or that do modify the global settings for the actual running socket connection and that are of interest for the socket communication.

```
Interactive on|off
```

The command toggles command line editing and echoing of characters received on or off. This switch corresponds with the device configuration switch `TCPIP_Editmode`. The normal case is to not modify this setting. Any modification done here is local to the actual running connection. If the connection was closed, any modification will be reset.

While in interactive mode, the system provides some simple command line editing and echoing of data received. See Chapter 4.1 for further details.

```
Rprefix on|off
```

If the “receive prefix” is switched off, the user did not get a response about successful completion of commands send or if there are any errors. This means the values `00:` or `01:` are not send back as a response. It may be useful switching off the “receive prefix” if you are running in batch mode, sending out data only (not receiving anything). But be aware, there is no chance to detect any kind of errors if you have switched off the `rprefix`.

```
Rcount on|off
```

If E488s was accessed true a large switched network, it is possible that packets send out from E488S are split into smaller parts on the road. If there are any delays on the net it could become impossible for the receiver to detect the end of the resulting data. If this happens, the user may switch on the receive count. If switched on, E488S appends the number of bytes that do follow directly behind the success value (this means beginning at the third byte position). The receive count is always five bytes in length zero padded. See chapter 4.4 for further details. If the connection was closed, any modification will be reset.

```
Binmode on|off
```

Binary transmission mode can only be switched on if interactive mode is switched off (this is the default for connections to port 4881). If binary mode is switched on, data transmitted to a device using `send` or received from a device using `receive` did not need any kind of escape sequences to encode characters below decimal 32 or above 127, but be aware even if binmode is switched on, the newline character defines the end of a data stream.

To get information if a newline found within a data stream received using the receive command defines the end of the stream, the `rcount` switch

should also be defined to be `on`. To instruct the driver how many bytes to send using `send` the programmer should always include the optional counting parameter into the `send` statement. If the counting value is given, the interpreter did not misinterpret newline characters that are embedded into the datastream.

```
settings_copy on|off
```

If the user connects to either port 4880 or port 4881, the default behaviour is to use the corresponding device entry for every transmission. But sometimes the user wants to use the device settings of the device that is addressed within a command. If `settings_copy` is switched `on`, the driver uses the settings of the target that is addressed within a command. See chapter 4.5 for further details.

```
sstore
```

The gpib manager not only reads in a configuration while during startup, it is able to write a fully documented configuration file, that contains all the settings that differ from the defaults. Using the command `sstore` there is no need for a user to login to E488S to modify the standard configuration. The user can make all the needed changes to the driver, add and remove custom devices and if everything is done, instruct the manager to write a configuration file. Next time the system is powered up; it makes the entire configuration based on the newly generated setup file.

```
bye
quit
```

Both commands may be used to exit from the port connection. If the connection is simply closed by the client, the server (the GPIB manager) automatically closes the connection with a maximum delay of 10 seconds.

```
Reboot
```

The command reboots the whole system. Be careful using this command! Before rebooting the system the user should take a look on the currently open connections using the command `?connections` to ensure it is save to reboot. If the setup configuration parameter `Socket_Gedit_allowed` is switched off, the user can't reboot the system, you will get an error in such case.

```
verbose value
```

The command defines a global new verbose level. The global verbose messages are printed on the screen. This means that for the normal user it makes no sense to switch the verbose level because a monitor must be connected to the system to see the output. The value is binary coded; there is one bit defined for every verbose level. It may be given either decimal or hexadecimal (leading 0x). Look into the appendix for a detailed listing of the encoding of the values or get a list of the currently supported values with the query command `?vlevels`.

The command `TLOG_verbose` is of more interest for the user, because the output can be retrieved directly either via the socket connection or via ftp.

4.6.3 Modifying and querying global parameters and devices

A socket connection provides several commands to modify and query the global setup the system itself. Querying devices and parameters is allowed always. Making modifications to the global parameters and modifying devices is allowed only if the configuration parameter `Socket_Gedit_allowed` (in `aksgpib.cfg`) is set to `on` or `yes`. This is for security reasons to prevent modifications done by notion less users. Modifications made to the system may be saved to flash using the command `sstore`.

```
Query_globals
```

The command may always be used, even if `Socket_Gedit_allowed` is switched off. It shows the current global settings of the system. The output produced by the command follows a syntax that is identical to the syntax processed by the parser during startup of the system. The output may be copied directly into a text file to create a custom configuration file that can be downloaded to the system. The following Listing shows an example output:

```
>query_globals
00:Manufactorer         TNT4882_ASIC # if autodetection did not work
debug                   0x00008e     # not to high
delta_timing            Yes          # Verbose messages are shown with delta timing
Admin_name              "/dev/gpib"  # Device Name for this Manager
I_am_CIC                Yes          # I am the controller
priority                10           # this changes during runtime
Advanced_read           No           # only for compatibility with older cards
IB_delay                0            # delay between commands send out
My_Address              1            # this is my talk and my listen address
Recover_Timeout         5000         # currently not used
CSRC_delay              0            # msec sleep time while waiting until device ready
My_Delays               BUSY_LOOP    # we are waiting this way (BUSY_LOOP or TIMER)
Take_Control            TCA          # synchron (TCS) or asynchron (TCA)
Take_Control_Delay      0            # wait after transmissin before raising ATN [msec]
Store_Setup             No           # don't regenerate this file on exit
Socket_Gedit_allowed    Yes          # Global modifications are allowed while connected via Socket
TCPIP_Passwd            No           # request passwd with IP comm
TLOG_verbose            0x000fff     # level to be used in tlogs
TLOG_before             100          # number of lines to save before trigger
TLOG_after              5            # number of lines to save after trigger
TLOG_trigger            0x000000ff   # event that do the trigger
TLOG_device_mask        0x0000       # devices that are monitored
TLOG_line_bits          0x00000000
TLOG_fmask              "/tmp/tlog"
TLOG                    Yes
```

```
Globals command [parameter]
```

The `Globals` command acts as a prefix command. Following the `Globals` command a global system setting must be entered like those shown with `query_globals`. For example to modify the timing shown in the tlog or verbose output the user may enter:

```
Globals delta_timing off
```

This command switches from delta timing to absolute timing.

Be aware, some settings are not allowed while the system is running. These settings are simply ignored, or, if the setup is stored with the `sstore` command, the settings are activated during next startup of the system.

```
Query_Device Address | Device_name
```

The command shows all settings of the given Device. Device Numbers 0 to 31 so represent default devices that are defined within every system. Devices that are identified by name are custom devices. A complete list of all devices is shown with `?Devices`. If the command `sstore` is used to create a configuration file, only parameters that differ from the defaults are written. The query command shows all parameters, even the default ones. A query may look like the following example:

```
Define_Device "Keithley_196"
    Name                "Keithley_196"       # for identification only
    Addr                4        # the device uses this GPIB address
    SAddr               -1       # no default subaddress used
    Com_mode            COOKED
    EOS8                Yes      # we use eight bit EOS
    EOSBYTE             10       # this is our EOS-byte
    EOT_OUT             EOIEOS   # (NEOINEOS, EOINEOS, NEOIEOS or EOIEOS)
    EOT_IN              EOIAEOS  # (EOI, EOS, EOIAEOS, EOIOEOS or NONE)
    SCPI_Mode           No       # some things are done automatically
    ITimeout            10000    # our timeout during receive [msec]
    OTimeout            10000    # our timeout during send [msec]
    ABORTIO_after_send  No       # for general ABORT IO after transmission
    Timeout_CRST        No       # chip reset on timeout
    TCA_after_send      Yes      # take control async after send
    twc                 0        # wait this number of msec between two I/O operations
    SPOLL_Type          SPE_ADR  # either SPE_ADR or ADR_SPE
    NFI_Mode            No       # NFI means intelligent (no)cache management for NFS access
    Buffer_mode         SHARED   # either shared or openspecific
    TCPIP_Port          0        # devices may be accessed via IP using this port
    TCPIP_Welcome       ENABLE   # either ENABLE or DISABLE
    TCPIP_Editmode      OFF      # no command history, no editing
    IO_Settings         OWN      # use settings of IP device (OWN) or target settings (TARGET)
Define_End
```

For a detailed description of all the definitions made for a device please look into the configuration chapter.

```
Define_Device Address | Device_name
Define_End
Define_Abort
```

The commands are used to modify the settings of an existing device. The device may be given by address (0-31) or by name (device with name given must exist). If the device did not exist, you will get back an error. If the device exists, the interpreter switches to device definition mode. Any further entries are identical to the device definition within the configuration file. It's not necessary to define all settings. You may only define the settings you like to modify.

For the options allowed within the definition take a look onto the output of the command `Query_Device`. A device definition was finished by using the command `Define_End`. Until `Define_End` was entered, no modifications are done to the real device, this means everzthing is

buffered and copied to the real settings if the definition was finished using `Define_End`.

The command `Define_Abort` may be used to reject any changes made. The command also finishes the definition mode.

```
Unlink_Device Device_name
```

The command removes a custom device from the current definition. After removing the device, it will not be available for further access. If there are connections pending on the device, the process of deletion get preferred until the last connection was closed.

4.6.4 Timing a connection

The system provides some commands for timing measurement. These commands are useful to get detailed information about the throughput of commands (Attention high) and data, send to and received from a device. The timing measurement is made in the background automatically for every connection. These means accumulations of transmission times are not on a device specific base but on an open specific base because the default connection would be unspecific. The timing measurement is only made for the GPIB transfers itself. Such measurements can't be made by a custom program on the users side, because if the user makes timing measurement, delays produced by the IP traffic do falsify the results. It could be of high interest if the user also makes timing measurement to get detailed information about delays and bottlenecks within the whole system.

```
?timings
```

The command shows the sum of the transmission times of all commands (transmissions with ATN high) and all data since the beginning of the connection or the last `timings_reset`. The resolution of the values shown is one nanosecond, but be aware that the system can't really make measurements with such a resolution. The typical accuracy is 10 nanoseconds.

```
?timing
```

The command shows the transmission time of the last command sequence executed (either a send, receive, spoll etc. command).

```
timings_reset
```

The command resets the timing values of the current connection.

4.6.5 Event processing

The socket interface contains the option to query and wait for events that may occur on the GPIB.

```
SRQ_wait [milliseconds]
```

The command checks for a service request. The optional parameter defines the number of milliseconds to wait for the event. If the parameter is not given the commands waits unconditionally (possibly for godot). If the parameter is defined to be zero, the command polls the current SRQ status. The command may be aborted by the user by sending the binary value 3 (ctrl C).

If SRQ was raised before the command was called, the command immediately returns the number of milliseconds since the SRQ was seen by E488S (the system automatically detects an SRQ event).

If SRQ gets raised while the command is waiting, the command returns success.

If a timeout has occurred (number of milliseconds are elapsed) or the command was aborted by the user, the command returns an error.

```
CIC_wait [milliseconds]
```

The command checks if the driver (the system) becomes (or is) the CIC. The optional parameter defines the number of milliseconds to wait for the event. If the parameter is not given the commands waits unconditionally (possibly for godot). If the parameter is defined to be zero, the command polls the current CIC status. The command may be aborted by the user by sending the binary value 3 (ctrl C).

If the system currently is the CIC, the command immediately returns the number of milliseconds since E488S has become the CIC (the system automatically detects a CIC status change).

If the system becomes the CIC while the command is waiting, the command returns success.

If a timeout has occurred (number of milliseconds are elapsed) or the command was aborted by the user, the command returns an error.

4.6.6 Advanced transmission analyses

The system contains very advanced GPIB debugging, timing and analyses tools. The complete toolset can be managed with the TLOG family of commands. Nearly every processing command internally to the system and GPIB transmission can be written to an internal ring buffer. The logging is coded to be highly effective. Only a very small amount of time is needed to do the logging. While transferring data via the GPIB there is no time lost for the small additional overhead because the GPIB is slow enough and the ordering of the code is done such way that the logs are written during idle time of the system.

We have checked the timing generated by the TLOG family of commands with a 100 MHz digital memory oscilloscope. In every case, the timing is as exact as 100 nanoseconds. GPIB transmission timing is as exact as 10 nanoseconds. We where able to write such code because we are using a high performance operating system (QNX) and we have glued this with more than ten years of experience.

The ring buffer used by the TLOG is big enough to hold 10000 lines. If the ring buffer gets full, newer entries do overwrite the oldest ones. No copying is done internally if the buffer gets full, instead read and write pointers are moved (this is typical for a ring buffer). This kind of ring buffer management did not produce any kind of time overhead due to copying.

Like a logic analyser the TLOG support trigger points. If a previously defined trigger hits, a range of the current TLOG is written to the internal flash for further analyses. The TLOG system always holds up to twenty files. The last file may simply be viewed with `?TLOG`.

```
Tlog on | off
```

The command switches the TLOG management on or off. For most cases, it should be best to leave the TLOG switched on, which is primarily done within the configuration file. If the TLOG is on, the user may analyse the system at any time going into the depth of the system and the GPIB, analysing strange events that have occurred while there was no operator at the system. The TLOG should only be switched off to get the last percent of performance enhancement (if ever, because GPIB transfers are mostly slower than the TLOG generation done in the background).

```
Tlog_verbose value
```

The command defines the logging level of the TLOG. To get a list of the currently supported levels you may enter `?vlevels`. The appendix also contains such a list. The value may be given either decimal (not very useful) or hexadecimal with a leading 0x. The value that generates an output with a maximum degree of information is 0x007ff.

```
Tlog_before nnnn
```

The command defines the number of entries written to the flash before a trigger point. If the trigger hits the TLOG thread writes entries from the ring buffer beginning with the trigger position minus TLOG_before until TLOG_after to flash. The default value is 100 lines. It is up to you to define a useful value here. The maximum value is 10000,

```
Tlog_after nnnn
```

The command defines the number of entries written to flash after a trigger point. The whole entry is written from the trigger point minus Tlog_before until Tlog_after. The TLOG thread waits writing data to flash until the number of entries defined with Tlog_after is accumulated if a trigger hits. Because of this, the time needed to write the file depends on the actions done by the whole system. It is a good idea to make this value not to great (the default is 5) to not get a big delay between the event itself (the trigger point) and the time the resulting file is written to flash.

```
Tlog_tmask value
```

The command defines a binary encoded mask of trigger points that are watched by the system. The bit mask defines typical error conditions that may occur while doing GPIB transmissions. At the writing of this manual the following conditions are defined:

Value	Meaning
0x0001	A timeout has occurred after a device was addressed to talk. Possible reasons: • Device has crashed • Device has been switched off • Cable broken • Other device with same GPIB address
0x0002	A timeout has occurred after a device was addressed to listen. The possible reasons are the same as for the talk timeout.
0x0004	A timeout has occurred after a device was addressed for serial polling. The possible reasons are the same as for the talk timeout. Additionally it is possible that the device did not support serial polling.
0x0008	A timeout has occurred while sending data. There is an uncountable amount of possible reasons for this. Additionally to the reasons given for the talk timeout the following may be possible: • Buffer overflow within the device • Configuration error for the end message It would be of interest looking to the byte position where the error has occurred. If no byte could be sent to the device for sure there would be one of the first reasons. If the error occurs with the last byte send to the device there

	is a configuration error with the end message. You have sent all the data but the device was unable to see the end message (either EOI, EOS or both).
0x0010	Abort-I/O currently not used
0x0020	Internal Abort-I/O was called. If the system is not the CIC, only ib_init (chipset initialisation) was called. If the system is the CIC, first attention was raised, then IFC was set for about 100 microseconds. Internal Abort-I/O is called because of the following conditions: • A transmission can't be completed due to a problem • A device is defined such way that Abort-I/O should be called after every transmission
0x0040	A timeout has occurred while receiving data. Like the possible reasons for the same condition while sending data, there is also a great amount of reasons while this can happen. To solve the problem it would be very interesting to see if any data was transferred before the error has occurred: • If no data was received, probably the device did not like to send data. An SCPI device for example must first be instructed to send data. • If all the data was transferred, but the end message was not detected by E488S in most cases the reason is a configuration error for the end of stream. • If the timeout occurs in the middle of a transmission, you really have a problem. There must be a problem with the device itself.
0x0100	This is a very special form of an event trigger. E488S looks for empty messages received. Within a standard system this normally didn't happen because a standard system don't send empty messages. The normal case is either to send nothing (timeout) or to send something, but not to send only an end of stream termination without any data. We have found it useful sometimes to trigger for such an event, because if this occurs often there is a problem with the device addressed to talk. For example, a device is able to send data but the data itself isn't stable and because of this the device didn't send anything.

Table 5: TLOG trigger events

The small table shown above only gives some hint's for debugging of GPIB connections. Within the real world, the number of possible problems that may occur is uncountable. Detailed analyses of the TLOG output should normally help isolating the problem.

```
Tlog_trigger
```

The command forces a TLOG trigger event. This can be done every time without any problem. Like every other trigger the currently processes

event is marked with an asterisk. What you will get is a snapshot of the system activities at the moment.

```
Tlog_list
```

The command shows the number range of the TLOG files that are currently stored within the system. Every TLOG file may be reviewed with the command `?tlog`.

```
?Tlog [nnn]
```

The command shows the contents of the TLOG file with the given number. If no number is given, the last TLOG is shown. Because the TLOGs are stored within the systems file system, the user may also download the TLOGs via ftp.

4.6.7 Doing GPIB transfers

Processing GPIB commands is not part of the socket interface itself. The GPIB commands are processed by a device that has the communication mode set to COOKED. Indeed the socket device is defined to be of such a type. This means the processing of GPIB commands isn't done by the socket interface itself like all the other commands from the previous sections, but it is redirected to the device management internally. From the users point of view there is no difference. The GPIB commands are entered the same way as any other commands, but there is a difference in writing this manual. For a detailed description about using the GPIB command interface go to section cooked devices or to the section IEEE488 devices.

Some additional hints should be given in this section to help avoiding misinterpretations. Until now every time you have accessed the socket interface, the corresponding device entry was used to define the environment to use. The device entry not only contains definitions for echo and welcome messages send thru the socket. Indeed it also contains all the definitions for a GPIB device. These definitions are generally usable for most devices (end of stream termination and so on), but sometime a device needs a very special handling. To overcome the needs of a complete reconfiguration of the socket interface it is recommended to define a custom device that exactly contains the settings needed. Additionally the user must instruct the socket interface to use the settings of the device entry that corresponds with the address that is given with a command instead of the settings of the socket interface. This is done with the command `settings_copy on`.

If the user likes to send low level GPIB commands directly to the bus, the device "low_level" should be used as the target address given with the send command.

5 Different modes of stream interpretation

One of the biggest advantages of E488S is the intelligent transfer management. The system not only puts data received via Ethernet to the GPIB, instead it contains an intelligent management system that is highly customisable and which provides advanced buffering in both directions (sending and receiving data to and from the GPIB), intelligent command language driven data transfer and low level GPIB command processing.

One of the biggest advantages of E488S is the ability to buffer requests in both directions completely under software control by the customer. Any timing problems that may occur either on the GPIB or the Ethernet are isolated and not moved to the other medium. This means Ethernet delays did not have any impact on the GPIB transfer and GPIB transfers did not have any impact on Ethernet packet sizes and transmission times.

While configuring a device, the user can define the access mode for the device. Currently three different modes are defined:

- RAW
- COOKED
- IEEE488

If a device is defined to be of type raw, data sent to the device is not touched by the GPIB manager. The manager makes all the necessary addressing, data transfer and deaddressing.

If a device is defined to be of type cooked, all data sent to the device is interpreted by the GPIB manager. The manager provides a stream oriented command language thru such devices. The user did not send any data directly to such a device, instead commands to the manager are send thru the device.

If a device is defined to be of type IEEE488, it provides a low level interface to the GPIB itself. Such a device also interprets the stream sent. But the stream did not contains high level commands like the one used for the cooked devices. Commands sent to a IEEE488 device type entry must contains commands like UNL, UNT, GTL and so on.

This section of the manual describes the usage of the different kind of devices.

5.1 RAW devices

If data transfers are done using raw devices, the data isn't interpreted by the GPIB manager. This means the raw data stream is send directly to the device. This did not mean that no other action than putting the data out to the bus is done. If E488S currently is the CIC (this is the normal case), all addressing and deaddressing is done automatically. While writing data to such a device the following steps are performed:

- set local message remote (REM)
- send unlisten (UNL), untalk (UNT)
- send listen address of receiver (LAD), send my talk address (MTA)
- send data with end of stream termination defined by device entry
- send unlisten, untalk

If there is a read from such a device, the following steps are performed:

- set local message remote (REM)
- send unlisten (UNL), untalk (UNT)
- send my listen address (MLA), send address of talker (TAD)
- receive data until end of stream or counter hits (depends on device entry)
- send unlisten, untalk

It's not even simple to detect how many data should be send or received, because while sending data the GPIB manager can't automatically detect the position where it should stop collecting data from a software client, beginning sending data do a device. While receiving data the manager is only able to automatically detect the length if there is an end of stream defined for the device.

While sending data the number of bytes transmitted e.g. the byte count where the transmission starts depends on a given counter and the end of stream definition:

End of Stream Definition	Counter given	Action taken
None or EOI	Yes	Data sent if "count" number of bytes received from client
None or EOI	No	Data sent if one write request was fulfilled (e.g. data gets sent on every write)
EOS or EOS and EOI	Yes	Data sent if "count" number of bytes received, EOS character automatically appended to outgoing data
EOS or EOS and EOI	No	Data sent if EOS character was seen within the outgoing data stream

Table 6: RAW devices and moment where data is sent out

If NFS is used to access the GPIB via E488S the things become more tricky. There is an own section within this manual that deals with additional NFS options.

While receiving data, the number of bytes transferred is defined by the sending device. For the listener there is no way to defined the length of the message transferred. It's only possible for the listener to not get all the data, which the talker likes to send. The following table shows the actions performed by the receiver engine depending on the end of stream definition and a given bite count:

End of Stream Definition	Counter given	Action taken
None	Yes	E488S gets "count" bytes from the device
None	No	E488S gets up to 2048 bytes from device
EOI	Yes	E488S gets bytes from the device until "count" number of bytes are received or EOI was raised by the talker
EOI	No	E488S gets bytes from the device until EOI was received
EOS	Yes	E488S gets bytes from the device until "count" number of bytes are received or the EOS character was seen within the incoming stream.
EOS	No	E488S gets bytes from the device until EOS character was seen within the incoming stream.
EOI and EOS	Yes	E488S gets bytes from the device until "count" number of bytes are received or the EOS character with EOI raised was seen within the incoming stream.
EOI and EOS	No	E488S gets bytes from the device until EOS character with EOI raised was seen within the incoming stream.
EOI or EOS	Yes	E488S gets bytes from the device until "count" number of bytes are received or the EOS character or any other character with EOI raised was seen within the incoming stream.
EOI or EOS	No	E488S gets bytes from the device until EOS character or any other character with EOI raised was seen within the incoming stream.

Table 7: RAW devices and moment where data received

Like sending data, possibly the things become very tricky if data is received via NFS. Additional NFS problems may arise because of caching made by the client. There is also an own section, which gives detailed instruction how to outwit possible limitations of NFS while receiving data.

As you can see, the biggest problem in using raw devices is the detection of the length of an outgoing or incoming data stream. Basically GPIB is a block oriented system, but with variable block length (even for one device). The system needs any kind of information about the length of data that should be transmitted. If such information is not given by a

clearly defined end of stream definition, things possibly become ugly if raw file system access only is used.

The things become very simple if the socket interface is used, because socket send and receive commands optionally define a byte count to give detailed information to E488S about the amount of data that must be transferred. Also while using the socket interface the manager always knows the number of bytes to transfer with the only exception “receiving data with none end of stream definition”. It is not the normal case to set up a socket interface directly for a raw device. Instead raw devices “stay silent”, they are defined within the system, but they are accessed from a cooked device that provides the socket interface to the outer world.

Using the raw devices is recommended for all devices that have a clearly defined end of stream definition.

5.2 COOKED Devices

These devices are normally used for the socket interface, but they may be accessed also via NFS or like any other device. Data send to a cooked device is not send directly to a GPIB device. Instead, the cooked device provides a command language and the data stream send to such a device is interpreted and processed. This chapter contains the details of the command language.

A cooked device acts stream oriented with a newline character at the end of each command. No data conversion is done by the cooked device itself. Because of this while using the send command with cooked devices combined with binary data, the optional length parameter for the send command must be given to avoid misinterpretation of a newline character possibly embedded within the send data stream.

Cooked devices are used by the socket interface, but the socket interface adds the option for escape encoding control characters within the data stream.

Reading from a cooked device did not automatically retrieves data from a target device (talker). A send command must previously be send to the cooked device. This also means data was put into the receive buffer (the buffer that was read by your client) at the moment where the data that was send to the cooked device is processed by the E488S interpreter.

Cooked devices are typically used for the socket Interface. The standard configuration of E488S provides two such interfaces. If commands are send to the socket interface the socket thread first tries to interpret the data received and if it can't it forwards the input the underlying cooked device. This section describes the commands provided by the cooked devices.

The command names are not case sensitive. A command name may be shortened until it can be clearly identified by the parser (for example rece instead of RECEIVE).

If a command needs an address, the address may be given either decimal (0-31) or by name. If a named address is given, there must be a device entry, which corresponds to such address. To query the list of defined devices the user may enter `?devices` while connected to the socket interface.

5.2.1 GPIB commands provided with cooked devices

The command separator used by E488S is a newline character “\n” that must be send with every command. For every command processed E488S sends back a response. It is possible to concatenate several commands, each of them separated by a newline.

```
IBINIT
```

The command initialises the GPIB chipset itself. If E488S is currently the CIC, the IFC line was also raised for about 100 microseconds.

```
CLEAR
```

The command sends the global message DCL to all devices connected to the bus. There is no addressing be performed. The result and reaction of a device after it has received the DCL message depends on the particular device. Normally a full reset is done.

```
DCLEAR Address[,Subaddress]
```

The command sends the one-byte message SDC to the selected device. The device address must be given as an argument. The subaddress is optional. The action performed by a device that receives an SDC message depends on this device. Normally a partial reset is done.

```
MYADR Address
```

The command defines the address of E488S itself. The default address is 0. The default address should be defined within the configuration file. The address is used for every GPIB command where data is transferred between E488S and a device (SPOLL, RECEIVE, SEND).

```
?MYADR
```

The function returns the currently defined own address of E488S.

```
IFC
```

The command sends Interface Clear by pulling the IFC line for a period of 100 microseconds, as defined by IEEE488. No addressing is done for this command.

```
ERRSTAT
```

The command returns the error and the suberror code of the last command. For each command processed by E488S the error codes are reset. This means if you did not get “0,0”, then the last command has generated an error. For further details about the error codes look into the Appendix.

```
PASSCTL Address
```

If E488S is currently the Controller In Charge (CIC) the command passed control to the device with the given address.

```
SPOLL Address[,Subaddress]
```

The command requests the serial poll status of the addressed device. The serial poll status byte contains a bit-encoded message containing the actual state of the device. The definition of the bits is device-dependent. By definition, only bit 6 (0x40) is reserved for the request of service status (SRQ). Please consult the reference manual of your device for further information.

```
LOCKOUT Address
```

The command sends a local lockout (LLO) to the addressed device. If a device supports local lockout, the result is a locking of the front panel. If a device did not support LLO, the command should simply be ignored by the device. Often the command is used to prevent a user from modifying settings of a device via a front panels keyboard.

```
LOCAL Address
```

The command sends the go to local message (GTL) to the addressed device. This unlocks a previously LLO.

```
REMOTE Address
```

The behaviour of the command depends on the given address. IF the address is not the own address of E488S and not –1, the behaviour is identical to the lockout command, else the command simply raises the IEEE488 remote line.

```
TRIGGER Address
```

The command send the GPIB message group execute trigger (GET) to the addressed device. The action performed by a device after receiving a GET message is not generally defined. Normally a measurement cycle begins in such a case.

```
SEND Address[,Subaddress][;Numbytes]:Databytes
```

The command is used to send data to the given device. If the device address is identical to the address of E488S, no addressing is performed. Because of this, the command may also be used to send data if E488S is not the CIC but addressed to talk.

The numbytes parameter is optional, but it should be given if the data bytes do contains binary data. Because most communication with GPIB devices did not contain any binary data, there is no need to enter the

numbytes value. While using the interactive socket interface (port 4880) the user may simply enter:

```
send 4:DhelloX
```

If the return key is pressed, the interactive socket interface translates the value 0x0d (return) to 0x0a (newline) and forwards “send 4:DhelloX\n” to the cooked device responsible for processing port 4880 connections.

If the non-interactive socket interface (port 4881) is used the software developer must append the newline to define the end of the command:

```
write(sock_fd,”send 4:DhelloX\n”,15);
```

There is one thing special with the newline character while sending data. The newline character that is used to separate the commands is also used by the GPIB as a standard EOS character. If the end of stream definition of a device defines newline as the standard EOS character, the low-level transmission part of the driver automatically checks if the outgoing stream contains the needed newline at the end. If newline was not found, it is appended automatically.

If the numbytes parameter is given, there is no need for the “\n” separator parameter because the length of the whole command is defined by numbytes:

```
write(sock_fd,”send 4;7:DhelloX”,16);
```

As said above, the newline needed to define EOS at the end of the data stream is automatically added by the driver if the device configuration defines end of stream for the target device this way.

```
RECEIVE Address[,Subaddress][;Numbytes]
```

The command instructs E488S to receive data from the device with the given address. If the device address is identical to the address of E488S, no addressing is performed. Because of this, the command may also be used to receive data if E488S is not the CIC but addressed to listen.

The numbytes parameter is optional for most devices, but it must be given if there is no end of stream definition for the addressed device. If numbytes is not given and there is an end of stream definition for the device (either EOS or EOI or a combination of both), the command receives all data until it receives end of stream. If numbytes is given for a device that also contains an end of stream definition, only numbytes bytes are transferred. A holdoff if done after numbytes bytes are transferred to stop the data transmission. If numbytes is not given for a device that has no end of stream definition, the maximum number of bytes (default 2048) would be transferred (which possibly ends in a timeout).

5.3 IEEE488 Devices

Devices defined to be of type IEEE488 do provide the lowest level of access to the GPIB. Advanced knowledge of the GPIB is necessary to get useful results if the low level command set provided here is used. But the user may manually create sequences not provided by the cooked mode commands.

The default configuration of E488S contains a device named low_level. This device can be used for low-level access, for example by accessing the device from the socket interface.

Not every command deals with the GPIB directly, some of them are used for reprogramming the GPIB chipset and some others are provided to control the command sequence itself (timeout for example). The commands are not case sensitive.

The following table contains a list of all low-level GPIB commands.

Command	Parameter	Meaning
CRST		Chip reset
GTS		Goto standby: attention was lowered Must be send out before any data transmission starts
LTN		Local Listen: switches E488S to the listener state Must be send out before data is received
LUN		Local Unlisten: leaves listen state If Attention was raised, LUN was automatically set
SIFC		Set Interface Clear (raising the management line IFC)
RIFC		Remove Interface Clear GPIB defines that IFC must be active for at least 100 microseconds to be detected by all devices
SREN		Set Remote (raising the management line REN) Most devices do not accept GPIB commands if REN is not active. It's a good idea to set REN each time without removing REN ever.
RREN		Remove Remote
TCA		Take Control Asynchronous
TCS		Take Control Synchronous
AUXM	nnn[,nnn [,…]]	Sends bytes to the Auxiliary mode register
CMD	nnn [,nnn [,…]]	Sends command bytes to the bus. The command can be used to send bytes to the bus that do not have a named equivalent within this list.
RHDF		Remove holdoff: After data transmission is done while receiving data, the holdoff must be cleared using rhdf
LAD	Address [,Address[,…]]	Listen Address: sends listener address
TAD	Address [,Address[,…]]	Talk Address: sends talker address
SAD	Subaddress	Secondary Address: Sends Secondary Listener/Talker Address
MTA		My Talk Address: Sends Talk Address of myself
MLA		My Listen Address: Sends Listen Address of myself

UNL		Unlisten
UNT		Untalk
GTL		Goto Local: clears remote line
SDC		Selected Device Clear
GET		Group Execute Trigger
SPE		Serial Poll Enable
SPD		Serial Poll Disable

Table 8: low-level commands provided by IEEE488 devices

The following table contains a list of all commands accepted by the low level interface, that are not directly send to the chipset or the bus, but that are needed to do transfers. This means there is some kind of intelligence build into the commands.

Command	Parameter	Meaning
EOSBYTE	Value	The command defines the byte that is used to detect the end of stream while receiving data using ENTER_EOS, ENTER_EOE or ENTER_EAE
EOS7		The system looks for 7 bit encoded EOS bytes
EOS8		The system looks for 8 bit encoded EOS bytes
TIMEOUT	Milliseconds	The command defines a timeout that begins counting at the moment of the definition until the currently executes command is processed.
NDELAY	Nanoseconds	The command waits the given number of nanoseconds. Be aware, the commands really waits nanoseconds (2 nsec resolution), but this results in a heavy system load. A maximum of 2 seconds may be given for the value.
DATA	nn[,nn[,…] EOI | END	The command sends data bytes out to the GPIB. The command should be used after a transmission was initiated. If the data stream ends with EOI, EOI is raised while the last data byte is transferred. If the command ends with END, no EOI is raised.
DATA	"stream" EOI | END	Instead of giving single data bytes, the data may be put into double quotes. This can be mixed with single data byte values.
ENTER_CNT	nnn	The system retrieves nnn bytes of data from the GPIB after is was addressed to listen
ENTER_EOS		The system retrieves data bytes from the GPIB until it reaches the EOS byte.
ENTER_EOI		The system retrieves data bytes from the GPIB until it reaches a byte with EOI set.
ENTER_EAE		The system retrieves data bytes from the GPIB until it reaches the EOS byte with EOI set.
ENTER_EOE		The system retrieves data bytes from the GPIB until it reaches a byte with EOI set or the EOS byte was seen.

Table 9: low-level combined commands provided by IEEE488 devices

5.3.1 Examples of low-level GPIB commands

If the customer likes to do own low-level GPIB programming, it is suggested to read some additional literature. Detailed information about using the GPIB chipsets used within E488S can be found at the National Instruments homepage.

To get some data for example, the following sequence may be used:

```
send low_level:sren tca unl unt mla tad 4 ltn gts enter_eoi tca
               rhdf unl unt
```

The sequence is identical to the high-level command "`receive 4`". The command sequence shown above contains a mixture of GPIB commands, local chipset instructions and instructions to the driver to do some processing:

- `sren` and `tca` are commands that do change the status of GPIB management lines (remote and attention).
- `unl`, `unt`, `mla` and `tad 4` are commands send to the bus (commands are data bytes send out with attention active).
- `rhdf` is a command that changes the status of the handshake lines (the holdoff is removed).
- `ltn` and `gts` are instructions to the chipset itself, `ltn` switches the chipset to the listener state while `gts` (go to standby) initiates the data transfer itself.
- At least `enter_eoi` is a high level instruction to E488S to begin collecting data looking for the EOI management line to detect the end of the data stream.

To send some data, the following two sequences do produce the same result:

```
send low_level:sren tca lad 4 mta gts data 68,72,69,76,76,79,
               88,10,EOI tca unl unt

send low_level:sren tca lad 4 mta gts data "DHELLOX",10,EOI tca
               unl unt
```

The sequence is identical to the command "`send 4:DHELLOX`". The data command accepts a mixture of single data bytes and strings.

It is not really a good idea to use the low-level interface to do programming that is better done with the internal commands of E488S (for example manually doing GPIB transfers), but the low-level command set should be used (must be used) if the user likes to create sequences where there is no high-level equivalent. Such an example is triggering several devices at the same time. The high-level command `TRIGGER` only accepts one address (like all the other high-level commands), but the GPIB allows triggering of several devices. This can be done with the following low-level commands:

```
send low_level:sren tca unl lad 4,5,7,10 get unl
```

The example shows how to address several devices once. After the devices have been addressed, the `get` command (Group Execute Trigger) initiates triggering of all the devices.

As you can see, some (some more) knowledge is needed to create a set of commands that do produce any useful output. There are a great number of mistakes, which can be made while doing GPIB communication. The low-level command set provided here can be compared with a big gun, which may possibly be used to shot into our knees.

Appendix A: Verbose levels of the GPIB Manager

The system generates verbose messages and TLOG entries based on a binary encoded value. The following table shows the definition of the bits:

Name	Value	Meaning
DBG_HIGHRES	0x00001	Output is done with a high res timer (nanoseconds resolution)
DBG_GENERAL	0x00002	Output general info
DBG_ERROR	0x00004	Output general error conditions
DBG_IOERROR	0x00008	Output I/O error conditions
DBG_TIMING	0x00008	Output I/O timing info
DBG_IBCMDS	0x00010	Output GPIB commands
DBG_FUNCTIONS	0x00020	Output internal function calls. This is only useful in the case of program errors for debugging
DBG_SOCKET	0x00040	Output socket connections
DBG_FILEIO	0x00080	Output resource manager file access. This could be useful to trace client access
DBG_CHIPREG	0x00100	Output register values of chipset
DBG_TRANSFER	0x00200	Output transmitted data and commands
DBG_SETUP	0x00400	Output configuration and setup changes

Appendix B: Example configuration file

The following listing shows the configuration file that is installed for the GPIB manager. The config file can be used as is or it may be custom tailored by the user. In the case the user destroys the file, there is a backup names “aksgpib.default” that is never touched by the system.

```
#####################################################################
# Name:        aksgpib.cfg
# Author:      Andre Koppel
# Version:     3.2
#
# Contens:
#       This configuration file describes the characteristics
#       and the behaviour of an gpib-driver connected to one
#       of the supported gpib-cards. During startup every driver
#       searches /etc/config for a configuration file with
#       the name "aksgpib.NODE" where node is the actual nodenumber
#       of the system and if this file was not found an attempt
#       to open the file aksgpib.cfg was made. The name of this
#   config file can be given on the command line during startup
#   of the driver with the -u switch.
#
# Manufactorer Codes:
#   NEC7210         PCIIA or compatible (INES IEEE488, IBM GPIB)
#                   Autodetect of TNT4882_ASIC
#   NEC7210_AKS     AKS GPIB and CEC-Interface
#   TNT4882         NAT4882/Turbo488 National Instruments TNT488-
#                   and GPIB.2-Interfaces
#   TNT4882_ASIC    TNT4882C-ASIC National Instruments AT-GPIB/TNT
#                   or PCMCIA-GPIB/TNT manufactored after 1994.
#   TNT4882_ONEC    Same as TNT4882_ASIC, but the One-Chip Mode is
#                   used (High-Speed FIFOs), reaching transmission
#                   speeds of more than one Megabyte/sec.
#
# If a PCI GPIB-interface is used, use the QNX utility
# "pci -v | less" to find out the resources of the interface.
# The PCI vendor-ID of National Instruments is 1093, the device-ID
# is c801. If you don't use the "-v" option, the GPIB device is not
# shown.
#####################################################################

iobase              0xec00       # ignored with PCI boards
IRQ                 11           # ignored with PCI boards

Manufactorer        TNT4882_ONEC # if autodetection did not work
debug               0x0008e      # show infos: General, Errors, Setup
Admin_name          "/dev/gpib"
priority            10           # this changes during runtime
Advanced_read       no           # only for compatibility with older cards
IB_delay            0            # no delay between commands send out
My_Address          0            # this is my talk and my listen address
Recover_Timeout     5000         #
#CSRC_delay         1
My_Delays           Busy_Loop
#Take_Control       TCS
Take_Control_Delay  0

#########################################################################
# let us define the event triggered logging
#########################################################################

TLOG_verbose        0x00fff
TLOG_before         100
TLOG_after          5
TLOG_trigger        0x00ff
TLOG                on

#########################################################################
# First of all we are doing some raw definitions. We are redefining
# some of the devices 0 to 31. We do not use a symbolic name with the
# Define_Device command, we are using a number instead. The driver
# automatically allocates 32 devices (0-31) and puts them into
```

```
# /dev/gpib. So what we are doing is to redefine the behaviour of
# the devices. It is not allowed to use the Addr directive within
# such a definition because the address is given with the Define_Device
# command.
# The main difference between the raw definitions and the device specific
# definition within the next section is the handling of the handling
# of the buffering. The raw devices within this section are handled
# on a per open basis (each open gets own buffers). This makes it
# impossible to share data between different open calls, but the advantage
# is that different processes accessing the same device can't be get
# in conflict.
########################################################################

Define_Device 0
  Name                  "Controller 0"        # this is a comment only
  twc                   0
  EOT_OUT               EOIEOS
  EOT_IN                EOIAEOS
  EOS8                  Yes
  EOSBYTE               10
  ITimeout              1000
  OTimeout              1000
  ABORTIO_after_send    No
  TCA_after_send        Yes
  SPOLL_Type            ADR_SPE
Define_End

Define_Device 1
  Name                  "Controller 1"
  twc                   0
  EOT_OUT               EOIEOS
  EOT_IN                EOIAEOS
  EOS8                  Yes
  EOSBYTE               10
  ITimeout              10000
  OTimeout              10000
  ABORTIO_after_send    No
  TCA_after_send        Yes
  SPOLL_Type            ADR_SPE
Define_End

# For the following device we switch on nfi mode (network file intelligence)
# using nfi-mode the driver looks at the requests that follow each other in
# a sequence of requests (only in raw mode).
# Consider the following:
# In Raw mode the driver gets new data automatically from a device if a
# clients reads data and the input buffer currently did not contain any data.
# IF local file access is used, everything is quite well. If a client
# requests 20 bytes and the buffer only contains 10 bytes, then the driver
# sends out 10 bytes to the client indicating that's all for now. The client
# gets these 10 bytes and he knows "if I read again some data the driver
# newly requests data from the device". Looks even simple and works well.
# But now the problem:
# If a gpib-device is distributed via network (NFS for example) the
# network file subsystem did not behave this way. If a client requests 20
# bytes and the client side of nfs only gets back 10 bytes, then the nfs
# thinks "oh, not as much received as requested, let us look if there are
# some additional data" and the client nfs makes a second request of 10
# bytes. This produces a problem. For the drivers side of view this is
# a brand new request and it gets new data from the device.
# Here is the solution:
# If nfi mode is switched on, the driver remenbers how many bytes it
# can not provide (20 bytes requested,10 bytes 10, it remembers 10).
# If the same client now requests 10 bytes, the the driver sees this
# as an consecutive request and the drivers answers with no bytes: EOF
# this works quite well.

Define_Device 4
  Name                  "Keithley 196"
  twc                   0
  Com_mode              RAW
  NFI_Mode              Yes
  EOT_OUT               EOIEOS
  EOT_IN                EOIAEOS
  EOS8                  Yes
  EOSBYTE               10
  ITimeout              10000
  OTimeout              10000
```

```
  ABORTIO_after_send    No
  TCA_after_send        Yes
  SPOLL_Type            SPE_ADR
Define_End

Define_Device 24
  Name                  "HP7495A"
  twc                   1
  NFI_Mode              Yes
  EOT_OUT               EOIEOS
  EOT_IN                EOI
  EOS8                  No
  EOSBYTE               10
  ITimeout              4000
  OTimeout              50
  ABORTIO_after_send    No
  TCA_after_send        Yes
  SPOLL_Type            ADR_SPE
Define_End

Define_Device 7
  Name                  "Simulator"
  twc                   0
  EOT_OUT               EOIEOS
  EOT_IN                EOIAEOS
  EOS8                  No
  EOSBYTE               10
  ITimeout              1000
  OTimeout              50
  ABORTIO_after_send    No
  TCA_after_send        Yes
  SPOLL_Type            SPE_ADR
Define_End


Define_Device 10
  Name                  "HP300"
  twc                   0
  EOT_OUT               EOIEOS
  EOT_IN                EOI
  EOS8                  No
  EOSBYTE               10
  ITimeout              1000
  OTimeout              50
  ABORTIO_after_send    No
  TCA_after_send        Yes
  SPOLL_Type            ADR_SPE
Define_End


######################################################################
# Now we are defining individual devices by entering the name behind
# the Define_Device command.
# Opening such a devices results in using a global buffer allocated
# for the device. All processes and threads using such a device
# do share the same buffer for writing and reading.
# If you are using command line tools (such as echo or cat) it's the
# only way to use such a device because the buffer contens is not
# thrown away after the command has finished.
######################################################################

# We are driving the Keithley Multimeter in double cooked mode.
# This means that all the data send to the driver is interpreted
# by the driver before it is send to the device. The results of
# the driver and the data received from the device are put into
# the receive buffer (remember, it's a global buffer available for
# all programs that use this device)

Define_Device "Keithley_196"
  Addr                  4
  Com_mode              COOKED
  twc                   0
  EOT_OUT               EOIEOS
  EOT_IN                EOIAEOS
  EOS8                  Yes
  EOSBYTE               10
  ITimeout              10000
  OTimeout              10000
```

```
  ABORTIO_after_send    No
  TCA_after_send        Yes
  SPOLL_Type            SPE_ADR
Define_End


Define_Device "HP3488A"
  Addr                  10
  Com_mode              COOKED
  twc                   0
  EOT_OUT               EOIEOS
  EOT_IN                EOIAEOS
  EOS8                  Yes
  EOSBYTE               10
  ITimeout              10000
  OTimeout              10000
  ABORTIO_after_send    No
  TCA_after_send        Yes
  SPOLL_Type            SPE_ADR
Define_End

# The following device uses a shared buffer, but is accessed using raw mode. This
# means that commands send to the device are not interpreted by the driver, all
# data is send to the device as soon as an end indicator is detected or lseek or
# pwrite is used and the number of bytes written to the driver is the same as
# the length parameter. If a receive is initiated, the driver imidiately gets
# the data from the device.

Define_Device "Keithley_196_raw"
  Addr                  4
  Com_mode              RAW
  twc                   0
  EOT_OUT               EOIEOS
  EOT_IN                EOIAEOS
  EOS8                  Yes
  EOSBYTE               10
  ITimeout              10000
  OTimeout              10000
  ABORTIO_after_send    No
  TCA_after_send        Yes
  SPOLL_Type            SPE_ADR
Define_End

# Now we define a named device with an open specific buffering
# (not very usefull for shell-programming). Remember: named devices do use
# a shared buffer by default and this can be changed to openspecific by
# using the Buffer_mode parameter (unnamed devices do use openspecific
# buffers by default).

Define_Device "OS_Keithley_196"
  Addr                  4
  Com_mode              COOKED
  Buffer_mode           OPENSPECIFIC
  twc                   0
  EOT_OUT               EOIEOS
  EOT_IN                EOIAEOS
  EOS8                  Yes
  EOSBYTE               10
  ITimeout              10000
  OTimeout              10000
  ABORTIO_after_send    No
  TCA_after_send        Yes
  SPOLL_Type            SPE_ADR
Define_End

# Now let us define an unspecific device. This only makes sense in cooked
# mode because there is no address defined. This device is shared by default
# If we like it to be openspecific, we must do specify it. Look at Buffer_mode
# While in cooked mode the data isn't directly send to the device. Instead
# the data send to the device is interpreted by the GPIB Manager. In other
# words the data stream must contain instructions like send, receive, trigger
# and so on

Define_Device "unspecific"
  Com_mode              COOKED
  twc                   0
  EOT_OUT               EOIEOS
  EOT_IN                EOIAEOS
```

```
  EOS8                  Yes
  EOSBYTE               10
  ITimeout              10000
  OTimeout              10000
  ABORTIO_after_send    No
  TCA_after_send        Yes
  SPOLL_Type            SPE_ADR
Define_End

# Now we do define an additional interesting device. We do define a low
# level GPIB device entry. Data send to such a device must contains
# GPIB instructions like TCA, RHDF, UNL, MTA, ENTER_EOS and so on.
# This devices gives the lowest interface to the GPIB chipsets.
# Even hardware registers of the chipset may me modified.
# Be carefull!!!!!!


Define_Device "low_level"
  Com_mode              IEEE488
  twc                   0
  EOT_OUT               EOIEOS
  EOT_IN                EOIAEOS
  EOS8                  Yes
  EOSBYTE               10
  ITimeout              10000
  OTimeout              10000
  ABORTIO_after_send    No
  TCA_after_send        No
Define_End

#########################################################################
# At least we define some unspecific devices that are bind to a port.
# The driver automatically provides access to such a device via
# TCP/IP if the TCPIP_Port command is used.
# Remark: The devices defined here are not only accessible via TCP/IP
# they are also usable like any other GPIB device.
#
# Because it makes no sense to define a shared IP adressable device,
# we include the statement Buffer_mode OPENSPECIFIC
#########################################################################

# First we define a convenient Device. Here we want to set the editmode
# to echo so we can see what we enter and we get a command prompt in this
# mode.
# Setting a port up this way is usefull for manual TCP/IP communication.
# We get a welcome message, we get an echo and the End of line was
# terminated by a "\r" (carriage Return) instead of "\n".

Define_Device "unspec_portdev"
  Com_mode              COOKED
  TCPIP_Port            4880
  TCPIP_Editmode        ON
  Buffer_mode           OPENSPECIFIC
  twc                   0
  EOT_OUT               EOIEOS
  EOT_IN                EOIAEOS
  EOS8                  Yes
  EOSBYTE               10
  ITimeout              10000
  OTimeout              10000
  ABORTIO_after_send    No
  TCA_after_send        Yes
  SPOLL_Type            SPE_ADR
  IO_Settings           TARGET
Define_End


# Let us define a Communication Port that has the Welcome message
# disabled by default. Because the Editmode statement is not
# included, it's switched off.
# Setting a port up this way is usefull for automatic communication.

Define_Device "unspec_portdev_nw"
  Com_mode              COOKED
  TCPIP_Port            4881
  TCPIP_Welcome         DISABLED
  Buffer_mode           OPENSPECIFIC
  twc                   0
```

```
  EOT_OUT               EOIEOS
  EOT_IN                EOIAEOS
  EOS8                  Yes
  EOSBYTE               10
  ITimeout              10000
  OTimeout              10000
  ABORTIO_after_send    No
  TCA_after_send        Yes
  SPOLL_Type            SPE_ADR
  IO_Settings           TARGET
Define_End
```

Appendix C: Macros used by the File and Socket Interface

This appendix contains all the macros used to define debugging bitmaps, error codes and advanced lseek commands. The actual version of the file can be found within the distribution package of the E488S gateway or the NTO GPIB driver.

```
/*
 * Device control file for the QNX 6.x Neutrino GPIB Resource Manager
 * written by Andre Koppel (c) Andre Koppel Software
 *
 * This file must either be copied to a projects directory or to
 * /usr/include
 *
 * Version 1.6
 *
 * History:
 * 2003.10.21 added numerous lseek-commands with feedback. Now IFC
 *            and all the other one way commands do return a result
 *            in the result stream if the corresponding "R" command
 *            is used (GPIB_LSEEK_RIFC instead of GPIB_LSEEK_IFC)
 */

/*************************************************************
 * First of all we do define verbose levels used by the driver
 * The verbose levels are binary combinations. Or in all the
 * values needed to get a big or lesser output
 *************************************************************/

#define DBG_HIGHRES    0x00001 /* output is done with a high res timer */
#define DBG_GENERAL    0x00002 /* output general info */
#define DBG_ERROR      0x00004 /* output general error conditions */
#define DBG_IOERROR    0x00008 /* output I/O error conditions */
#define DBG_TIMING     0x00010 /* output I/O timing info */
#define DBG_IBCMDS     0x00020 /* output GPIB commands */
#define DBG_FUNCTIONS  0x00040 /* output function calls */
#define DBG_SOCKET     0x00080 /* output socket connections */
#define DBG_FILEIO     0x00100 /* output resource manager file access */
#define DBG_CHIPREG    0x00200 /* output register values of chipset */
#define DBG_TRANSFER   0x00400 /* output transmitted data and commands */
#define DBG_SETUP      0x00800 /* output configuration and setup changes */
#define DBG_MASK       0xfffff

/*********************************************
 * Error-Codes used by the driver
 *********************************************/

#define GPIB_EDVR     1  /* undefined System Error */
#define GPIB_ECIC     2  /* Function defines GPIB board to be CIC */
#define GPIB_ENOL     3  /* No Listeners on GPIB */
#define GPIB_EADR     4  /* GPIB board not addressed correctly */
#define GPIB_EARG     5  /* invalid Argument to function call */
#define GPIB_ESAC     6  /* GPIB board not System Controller as required */
#define GPIB_EABO     7  /* I/O operation aborted (timeout) */
#define GPIB_ENEB     8  /* Non-existend GPIB board */
#define GPIB_EDMA     9  /* DMA error */
#define GPIB_EOIP    10  /* Asynchronous I/O in progress */
#define GPIB_ECAP    11  /* No capability for operation */
#define GPIB_EFSO    12  /* File system error */
#define GPIB_EBUS    13  /* GPIB bus error */
#define GPIB_ESTB    14  /* Serial poll status byte queue overflow */
#define GPIB_ESRQ    15  /* SRQ stuck in ON position */
#define GPIB_ETAB    16  /* Table Problem */
#define GPIB_EBUSY   17  /* Resource busy (other process uses resource) */
#define GPIB_ENCIC   18  /* GPIB board not non-Controller as required */
#define GPIB_SCPI_NQ 19  /* SCPI-Device: receive from device without previous query */
#define GPIB_CMDNF   20  /* command not found (telnet-Interface) */
#define GPIB_CMDNU   21  /* command abbreviation given to interpreter not unique */

#define GPIB_MAXERR 21

/*********************************************
 * Suberrorr codes provided by some functions
```

```
 ********************************************/

#define GPIB_SCOT    1  /* Timeout during command out */
#define GPIB_SCLT    2  /* Timeout during Check-List */
#define GPIB_SCTT    3  /* Timeout during Check-Talk */
#define GPIB_SCST    4  /* Timeout during Check-Spoll */
#define GPIB_SDOT    5  /* Timeout while sending data */
#define GPIB_SDIT    6  /* Timeout during data in check */
#define GPIB_SDOC    7  /* Data-Out-Check-Error */
#define GPIB_SADR    8  /* invalid device address */
#define GPIB_SSRQ    9  /* no SRQ-support enabled */
#define GPIB_SRBY   10  /* Resource busy */
#define GPIB_STLSL  11  /* Talker or Listener state lost during send/receive */

#define GPIB_MAXSUBERR 11

#ifdef __QNXNTO__
#include <devctl.h>
#endif

/*********************************************************
 * It's essential for GPIB to not only send and receive data
 * but to transmit out of band information (special setups
 * and commands). Because there is no special way to transmit
 * such information via standard read/write requests on a raw
 * device, the driver provides some special ways by the use
 * of the seek command. We assume that there is no need to
 * transmit more than 16MB with one read/write command
 * (normally GPIB-Data sequences are even short) so we can
 * use the 8 high bits of the seek command to redefine the
 * meaning of seek.This gives us a consistent way to transmit
 * out of band data even via NFS (if a GPIB-device entry
 * is exported via NFS, we can not use the standard way
 * "devctl" because devctl commands are not transmitted via
 * NFS.
 * So let us define some special meanings for lseek using
 * the 8 high bits.
 *********************************************************/

/*********************************************************
 * We are using the lseek command to define the number of
 * bytes written to or received from a device if the device
 * communication is done without a well defined end marker.
 * To define the length of the requested bytes (aka read, aka
 * receive) make left-shift the length-parameter by 8 and do
 * a logical or with GPIB_LSEEK_RLEN.
 * It is needed if the transmitted data is not terminated with
 * an EOS character or with the EOI Line
 * The length for data transmission is either given by a
 * pwrite (or then number of bytes written as a bunch if
 * write is used instead of pwrite) or for use with the lseek-command
 * the length-parameter is left-shift by 8 bits (like the length
 * of the read-instruction lseek) and or'd with GPIB_LSEEK_WLEN.
 * The length must be given, if the transmission is done without
 * an EOS character.
 *
 * remember: lseek is one of several possible ways. It's also
 *           possible to use pread/pwrite or the cooked mode
 *********************************************************/
#define GPIB_LSEEK_CMASK 0xf0000000 /* commands are identified by a number within */
                                    /* this bit position */

#define GPIB_LSEEK_IGN   0x00000000 /* every such lseek is simply ignored, this is */
                                    /* only to clear the NFS caches */

#define GPIB_LSEEK_WLEN  0x10000000 /* use this to define the write length */
                                    /* of a sequence not terminated with EOS */
#define GPIB_LSEEK_RLEN  0x30000000 /* use this :-) for the read-length */

/*********************************************************
 * Some NFS-implementations are limited:
 * we can't use 64-Bits in lseek (not each OS suports off_t=64 Bit, for example QNX4)
 * some implementations do a client side caching, that can't be switched off
 * What we are doing to solve this is splitting the lseek-command into two parts
 * using the following formular:
 * lseek(..., (cmd & 0xfff00000),SEEK_SET)
 * lseek(...,((cmd & 0x000fff00)<<11),SEEK_SET)
 * this results in two seeks, where both seeks contain only 13 most significant
```

```
 * bits, the  19 least significant bits are simply ignored.
 * Because nfs uses 8 KB blocks to get the data, the programmer may seek to a great
 * amount of different positions (to outwit caching of NFS).
 * To do this add a value of 8192 (incremented by 8192 for each access) to the lseek
 * After the first lseek is done, a read must be performed (if only lseek is used
 * without a read, client side NFS simply ignored the lseek, waiting for the next
 * command. You will get back "split cmd p1"
 * An example:
 * We wan't to transmit GPIB_LSEEK_RMGMNT & 0x8000
 * long our_cmd = GPIB_LSEEK_RMGMNT | 0x8000
 * lseek(fd,((our_cmd | GPIB_LSEEK_SPLIT) & 0xfff00000) | 8192,SEEK_SET)
 * read(fd,buffer,12);
 * lseek(fd,((our_cmd & 0x000fff00)<<11) | 16384,SEEK_SET)
 * read(fd,result_buffer,8)
 *
 * Next time we do the same command we add the 8192 to outwit caching:
 * lseek(fd,((our_cmd | GPIB_LSEEK_SPLIT) & 0xfff00000) | 24576,SEEK_SET)
 * read(fd,buffer,12);
 * lseek(fd,((our_cmd & 0x000fff00)<<11) | 32768,SEEK_SET)
 * read(fd,result_buffer,8)
 *
 * Now we want to receive 9 bytes because our device did not send any
 * EOS/EOS using the split-command:
 * long len_cmd = GPIB_LSEEK_RLEN | (9<<8) // 9 is the number of bytes requested
 * lseek(fd,((len_cmd | GPIB_LSEEK_SPLIT) & 0xfff00000) | 40960,SEEK_SET)
 * read(fd,buffer,12);
 * lseek(fd,((len_cmd & 0x000fff00)<<11) | 49152,SEEK_SET)
 * read(fd,result_buffer,9)
 *
 *
 * Our 8192-counter begins at 0x2000 and ends at 0x7e000, means counting
 * from 8192 to 516096, results in 63 blocks of 8192 byte steps.
 */

#define GPIB_LSEEK_SPLIT 0x40000000

/*********************************************************
 * Now we define the lseek definitions for out of band data.
 * Settings are defined with a special control bit that marks
 * the lseek command as a control command:
 *********************************************************/

#define GPIB_LSEEK_CTRL 0x20000000

/*********************************************************
 * Let us define the commands. A command is defined by the
 * second high byte of the lseek position.
 * Often commands also need some data (settings). These
 * data are transmitted with the two low bytes of the lseek.
 * The lseek command return the result of the command
 * (we don't return the actual file-position, we have redefined
 * the result).
 * Some NFS implementations are broken, they don't return the
 * lseek result to the client. For these implementations
 * we have added the commands beginning with "R". These
 * commands additionally return the result within the receive
 * stream.
 *********************************************************/

#define GPIB_LSEEK_RVALRCV   0x01000000 /* result value is put into receive stream */
                                        /* instead of returning them directly only */

#define GPIB_LSEEK_SPOLL     0x20010000
#define GPIB_LSEEK_RSPOLL    0x21010000 /* Result is put into receive stream */
#define GPIB_LSEEK_TRIGGER   0x20020000
#define GPIB_LSEEK_RTRIGGER  0x21020000 /* Result is put into receive stream */
#define GPIB_LSEEK_CLEAR     0x20030000
#define GPIB_LSEEK_RCLEAR    0x21030000 /* Result is put into receive stream */
#define GPIB_LSEEK_DCLEAR    0x20040000
#define GPIB_LSEEK_RDCLEAR   0x21040000 /* Result is put into receive stream */
#define GPIB_LSEEK_LLOUT     0x20050000
#define GPIB_LSEEK_RLLOUT    0x21050000 /* Result is put into receive stream */
#define GPIB_LSEEK_GTLOCAL   0x20060000
#define GPIB_LSEEK_RGTLOCAL  0x21060000 /* Result is put into receive stream */
#define GPIB_LSEEK_REMOTE    0x20070000
#define GPIB_LSEEK_RREMOTE   0x21070000 /* Result is put into receive stream */
#define GPIB_LSEEK_LOCAL     0x20080000
#define GPIB_LSEEK_RLOCAL    0x21080000 /* Result is put into receive stream */
```

```
#define GPIB_LSEEK_IFC       0x20090000
#define GPIB_LSEEK_RIFC      0x21090000 /* Result is put into receive stream */
#define GPIB_LSEEK_PASSCTL   0x200A0000
#define GPIB_LSEEK_RPASSCTL  0x210A0000 /* Result is put into receive stream */
#define GPIB_LSEEK_CHECKCIC  0x200B0000
#define GPIB_LSEEK_RCHECKCIC 0x210B0000 /* Result is put in receive stream */
/* for the following look at the corresponding devctl-command */
#define GPIB_LSEEK_SADR      0x200C0000 /* or in the secondary addr, use 0xff */
                                        /* to query only */
#define GPIB_LSEEK_RSADR     0x210C0000 /* or in secondary address, previous */
                                        /* address is returned in rcv buffer */
#define GPIB_LSEEK_GSADR     0x200D0000 /* or in the secondary addr, use 0xff */
                                        /* to query only. */
#define GPIB_LSEEK_RGSADR    0x210D0000 /* like GSADR, but previous address returned */
                                        /* in rcv buffer. */
#define GPIB_LSEEK_MODE      0x200E0000 /* or in (mode<<16), 0xff00 to query only */
#define GPIB_LSEEK_RMODE     0x210E0000 /* return previous val in rcv buffer */
#define GPIB_LSEEK_TMODE     0x200F0000 /* or in (tmode<<16), 0xff00 to query only */
#define GPIB_LSEEK_RTMODE    0x210F0000 /* return previous val in rcv buffer */
#define GPIB_LSEEK_MGMNT     0x20100000 /* send Information on Management, or in: */
                                        /* 0x8000: SRQ received (CIC mode) */
                                        /* 0x8100: I become the CIC (non CIC mode) */
                                        /* 0x8200: addressed to listen (non CIC mode) */
                                        /* 0x8400: addressed to talk (non CIC mode) */
                                        /* 0x8800: SPOLL-Sequence initiated (non CIC mode) */
#define GPIB_LSEEK_RMGMNT    0x21100000 /* like MGMNT, but result is returned on next read */
                                        /* this mode is switched off by not setting the */
                                        /* low byte, until then it stays on for every read */
                                        /* depending on the type requested you will find */
                                        /* the following within the receive buffer (allways */
                                        /* \n-terminated 8 Bytes): */
                                        /* SRQ0000: service request */
                                        /* CICnnnn: I become the CIC from previous CIC nnnn */
                                        /* LADnnss: my listen address nn received with */
                                        /*          subaddress ss */
                                        /* TADnnss: my talk address nn received with */
                                        /*          subaddress ss */
                                        /* SPE0000: Serial poll was initiated */
                                        /* IDL0000: driver was set to idle */

/*******************************************************
 * Now the convenience definitions
 *******************************************************/

#define SDCMD_GPIB_SPOLL(fd)    lseek(fd,GPIB_LSEEK_SPOLL,0)
#define SDCMD_GPIB_TRIGGER(fd)  lseek(fd,GPIB_LSEEK_TRIGGER,0)
#define SDCMD_GPIB_CLEAR(fd)    lseek(fd,GPIB_LSEEK_CLEAR,0)
#define SDCMD_GPIB_DCLEAR(fd)   lseek(fd,GPIB_DCLEAR_TRIGGER,0)
#define SDCMD_GPIB_LLOUT(fd)    lseek(fd,GPIB_LSEEK_LLOUT,0)
#define SDCMD_GPIB_GTLOCAL(fd)  lseek(fd,GPIB_LSEEK_GTLOCAL,0)
#define SDCMD_GPIB_REMOTE(fd)   lseek(fd,GPIB_LSEEK_REMOTE,0)
#define SDCMD_GPIB_LOCAL(fd)    lseek(fd,GPIB_LSEEK_LOCAL,0)
#define SDCMD_GPIB_IFC(fd)      lseek(fd,GPIB_LSEEK_IFC,0)
#define SDCMD_GPIB_PASSCTL(fd)  lseek(fd,GPIB_LSEEK_PASSCTL,0)
#define SDCMD_GPIB_CHECKCIC(fd) lseek(fd,GPIB_LSEEK_CHECKCIC,0)

/*********************************************************************
 *********************************************************************
 * Now the devctl interface. Using devcl we can provide a well
 * defined interface to transmit out of band data to and from
 * the driver. There is only one disadvantage: devctl is not
 * transmitted via NFS and SMB, so this interface can only be used
 * if a process is tocal on on a qnet. If it is not, you must
 * use the lseek interface above.
 *********************************************************************
 *********************************************************************/
/* the definitions of the transmission mode */
/* first the termination for the outgoing data */
#define GPIB_DEVCTL_TMODE_ODCH  4   /* don't set new value, ignore parameter   */
#define GPIB_DEVCTL_TMODE_ONO   0   /* no termination, counting only is used   */
#define GPIB_DEVCTL_TMODE_OEOI  1   /* EOI line is pulled with last char     */
#define GPIB_DEVCTL_TMODE_OEOS  2   /* Data Stream is terminated with EOS char */
#define GPIB_DEVCTL_TMODE_OEAE  3   /* like OEOS, but EOI is pulled with EOS   */
/* the definitions for data received from the GPIB */
#define GPIB_DEVCTL_TMODE_IDCH 80   /* don't set new value, ignore parameter   */
#define GPIB_DEVCTL_TMODE_IEOI  0   /* pulled EOI-line defines end of stream   */
#define GPIB_DEVCTL_TMODE_IEOS 16   /* received EOS char defines end of stream */
#define GPIB_DEVCTL_TMODE_IEOE 32   /* pulled EOI-line or received EOS char    */
```

```
#define GPIB_DEVCTL_TMODE_IEAE 48   /* pulled EOI-line and received EOS char   */
#define GPIB_DEVCTL_TMODE_INO  64   /* no termination, counting is used        */

/* spoll gives back the result as a char value */
#define DCMD_DEVGPIB_SPOLL    set_device_direction(0x5001,_POSIX_DEVDIR_FROM)
#define DCMD_DEVGPIB_TRIGGER  set_device_direction(0x5002,_POSIX_DEVDIR_NONE)
#define DCMD_DEVGPIB_CLEAR    set_device_direction(0x5003,_POSIX_DEVDIR_NONE)
#define DCMD_DEVGPIB_DCLEAR   set_device_direction(0x5004,_POSIX_DEVDIR_NONE)
#define DCMD_DEVGPIB_LLOUT    set_device_direction(0x5005,_POSIX_DEVDIR_NONE)
#define DCMD_DEVGPIB_GTLOCAL  set_device_direction(0x5006,_POSIX_DEVDIR_NONE)
#define DCMD_DEVGPIB_REMOTE   set_device_direction(0x5007,_POSIX_DEVDIR_NONE)
#define DCMD_DEVGPIB_LOCAL    set_device_direction(0x5008,_POSIX_DEVDIR_NONE)
#define DCMD_DEVGPIB_IFC      set_device_direction(0x5009,_POSIX_DEVDIR_NONE)
#define DCMD_DEVGPIB_PASSCTL  set_device_direction(0x500A,_POSIX_DEVDIR_NONE)
#define DCMD_DEVGPIB_CHECKCIC set_device_direction(0x500B,_POSIX_DEVDIR_FROM)

/*************************************************************************
 * now are some commands to query the actual value and optionally set a
 * new one. All parameters passed are given as signed char values. If the
 * actual setting should only be queried without defining a new value, the
 * parameter send out must be -1 (not for SADR and GSADR where it must be -2)
 *************************************************************************/

/* The secondary address on a per open basis (for the actual fd)            */
#define DCMD_DEVGPIB_SADR     set_device_direction(0x500C,_POSIX_DEVDIR_TOFROM)
/* The secondary address on a device basis. This means that on the next open,*/
/* the new value is used. Such a change is persistent over open calls (good  */
/* for command line programming)                                             */
#define DCMD_DEVGPIB_GSADR    set_device_direction(0x500D,_POSIX_DEVDIR_TOFROM)
/* The following command changes the access mode of the data stream          */
/* The parameter defines 0 for raw mode, 1 for cooked mode                   */
/* The parameter is either on a per open basis (if fd points to /dev/gpib/0  */
/* to /dev/gpib/31) or device specific (if fd points to a named device       */
#define DCMD_DEVGPIB_MODE     set_device_direction(0x500E,_POSIX_DEVDIR_TOFROM)
/* the following command defines the transmission termination for incoming   */
/* and outgoing data. See GPIB_DEVCTL_TMODE_... for the end mode.            */
/* This parameter is device specific, this means the mode is changed for all */
/* transmissions to and from the device, it's not on a per open basis        */
#define DCMD_DEVGPIB_TMODE    set_device_direction(0x500F,_POSIX_DEVDIR_TOFROM)

/*************************************************************************
 * some advanced features: We are able to inform our client if a management
 * line was raised (SRQ) or a management function was initiated (talk/listen)
 * so we define some devctl commands that only return if such an action occurs
 * Like all the other commands these devctl commands are also implemented as
 * lseek commands
 *************************************************************************/

#define DCMD_DEVGPIB_WAITSRQ set_device_direction(0x5080,_POSIX_DEVDIR_TOFROM)
#define DCMD_DEVGPIB_WAITCIC set_device_direction(0x5081,_POSIX_DEVDIR_TOFROM)
#define DCMD_DEVGPIB_WAITLAD set_device_direction(0x5082,_POSIX_DEVDIR_TOFROM)
#define DCMD_DEVGPIB_WAITTAD set_device_direction(0x5084,_POSIX_DEVDIR_TOFROM)
#define DCMD_DEVGPIB_WAITSPE set_device_direction(0x5088,_POSIX_DEVDIR_TOFROM)
```

Appendix D: Generic read/write program

```
/*************************************************************
 * Name:       rwgpib.c
 * Author:     Andre Koppel
 * Contens:    The file contains code to send Data to a gpib-device
 *          or to receive data from a device. There is nothing
 *          special with this code. The command line tools echo
 *          and cat do the same.
 * usage:      rwgpib device r|w|s [options] [data]
 *           r       means send data to the device
 *           w       means retrieve data from device
 *           s       processing only seek command
 *        options:
 *           -lnnn   define number of bytes to
 *                   transmit using pread/pwrite
 *           -Lnnn   define number of bytes to transmit using
 *                   lseek and read/write
 *           -rnnn   Number of bytes to receive default=1024
 *           -cnnn   Do Repeat instruction nnn times
 *           -x      split-mode used for lseek
 *           -v      be verbose
 *           -n      put newline at end of data stream to send
 *
 * History:    1.1 2003.07.29 first version
 *          1.2 2003.09.17 Switch -x implemented
 *          1.3 2003.10.01 modifications for split-lseek
 *          1.4 2003.10.05 verbose-mode
 *          1.5 2003.11.12 added -n switch -n
 *************************************************************/

#include <sys/types.h>
#include <unistd.h>
#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <stdlib.h>
#include "../gpib_devctl.h"

int verbose  = 0;     /* if the user likes to get more information */
int is_split = 0;   /* In split-mode we do a command using two lseeks */


void usage(void)
{
puts("invalid argument");
puts("usage:   rwgpib device r|w|s [options] [data]");
puts("           r       means send data to the device");
puts("           w       means retrieve data from device");
puts("           s       processing only seek command");
puts("options:");
puts("           -lnnn   define number of bytes to transmit using");
puts("                   pread/pwrite (not suported with QNX4)");
puts("           -Lnnn   define number of bytes to transmit using ");
puts("                   lseek and read/write");
puts("           -rnnn   Number of bytes to receive default=1024");
puts("           -cnnn   Do Repeat instruction nnn times");
puts("           -x      split-mode used for lseek");
puts("           -n      attach newline at end of data to send");
puts("           -v      be verbose");
exit(EINVAL);
}

/*************************************************************
 * We are defining an own lseek, it's named gpib_lseek.
 * Special here is the option to split one seek into two seeks.
 * Using NFS lseek is is a little bit special because the NFS
 * client makes only seeks with a block length of 8192 bytes
 * and then it gets a complete block of 8192 bytes (making the
 * doing the remaining seek with the cached data.
 * Because of this we can not use the low 13 Bits of the seek
 * parameter for data (it's simply not transmitted via NFS
 * What we are doing is using the split-mode of the driver.
 * If split-mode is used, two consecutive seeks are combined
```

```
 * into one complete parameter. The driver simply ignored
 * the low 13 bits. They must be zero. Additionally using the
 * split-mode a counting value is used (counting from 0 to 64)
 * to outwit the client NFS-cache. This means each seek is done
 * to a different position (this is also discarded by the driver).
 ************************************************************/

static long gpib_lseek(int fd,long pos,int is_split)
{
static long count=0;
unsigned long sp;
long res;
char buffer[20];

if(!is_split)           // if split mode isn't used there is nothing special
   return(lseek(fd,pos,SEEK_SET));

count+=8192;             // we move the position each time to outwit the cache
if(count>=0x80000)       // we must reset our counter to not modify
   count=8192;           // the definition bits
sp=((pos | GPIB_LSEEK_SPLIT) & 0xfff00000) | count;
if(verbose)
   {
   printf("sending split-lseek(0x%lx) part one: 0x%lx\n",pos,sp);
   fsync(fileno(stdout));
   }
lseek(fd,sp,SEEK_SET);
read(fd,buffer,12);     // we must do this, because the driver has send
                        // back a response
if(verbose)
   {
   printf("received answer \"%12.12s\"\n",buffer);
   fsync(fileno(stdout));
   }
sp=((pos & 0x000fff00)<<11) | count;
if(verbose)
   {
   printf("sending split-lseek part two: 0x%lx\n",sp);
   fsync(fileno(stdout));
   }
res=lseek(fd,sp,SEEK_SET);
return(res);
}

/*********************************************************
 * there is no support for pread/pwrite with qnx4, we do
 * fake it with seek
 *********************************************************/

#ifdef __QNX__
#ifdef __WATCOMC__

int pwrite(int fd,void *buffer,int len,int io_len)
{
gpib_lseek(fd,(io_len<<8) | GPIB_LSEEK_WLEN,is_split);
return(write(fd,buffer,len));
}

int pread(int fd,void *buffer,int rlen,int io_len)
{
gpib_lseek(fd,(io_len<<8) | GPIB_LSEEK_RLEN,is_split);
return(read(fd,buffer,rlen));
}

#endif
#endif


int main(int argc,const char *argv[])
{
char *buffer;
int fd,status,len,cmd,openmode;
const char *devname = NULL;
int   argp = 3;
int   rlen   = 1024;  /* Number of bytes to receive */
long  io_len = 0;     /* Number of bytes to transmit */
int   is_lseek = 0;   /* 0: use pwrite/pread, 1 use lseek */
int   repeat_cnt = 0; /* We do not repeat command */
```

```
int   is_nl = 0;

if(argc<2)
   usage();
devname=argv[1];
while(argp<argc && *argv[argp]=='-')
      {
      switch(argv[argp][1])
             {
             case 'L' : is_lseek=1;
             case 'l' : io_len=strtol(argv[argp]+2,NULL,0);
                        break;
             case 'r' : rlen=atoi(argv[argp]+2);
                        break;
             case 'c' : repeat_cnt=atoi(argv[argp]+2);
                        break;
             case 'x' : is_split=1;
                        is_lseek=1;
                        break;
             case 'v' : verbose=1;
                        break;
             case 'n' : is_nl=1;
                        break;
             default  : fprintf(stderr,"unknown arg \"%s\"\n",argv[argp]);
                        exit(0);
             }
      ++argp;
      }
buffer=malloc(rlen+10);
if(buffer==NULL)
   {
   fprintf(stderr,"Bad: not enough memory to allocate buffer\n");
   exit(ENOMEM);
   }

cmd=*argv[2];
// printf("cmd: %c\n",cmd);
if(cmd!='r' && cmd!='w' && cmd!='s')
   cmd='r';                              /* if command line did not define if */
                                         /* the user want's to send or receive */
                                         /* we assume receive */
openmode=is_split ? O_RDWR : cmd=='s' ? O_WRONLY : O_RDONLY;

#ifdef CHOSE_OPEN_MODE
fd=open(devname,openmode|O_SYNC);
f(fd==-1)
   fd=open(devname,openmode);
if(fd==-1)
   fd=open(devname,O_RDWR);
#else
fd=open(devname,O_RDWR|O_SYNC);
#endif

if(fd==-1)
   {
   fprintf(stderr,"unable to open \"%s\" for %s, errno=%d(%s)\n",devname,
           openmode==O_WRONLY ? "writing" : openmode==O_RDONLY ? "reading" : "undefined",
           errno,strerror(errno));
   exit(EINVAL);
   }

switch(cmd)
       {
       case 's' :          /* processing only seek-command */
                  do {
                     if(repeat_cnt)
                        {
                        printf("number of turns remaining: %d\n",repeat_cnt);
                        fsync(fileno(stdout));
                        }
                     if(io_len)
                        {
                        status=gpib_lseek(fd,io_len,is_split);
                        fdatasync(fd);
                        if(verbose)
                           {
                           printf("result of gpib_lseek(0x%0x): %d\n",io_len,status);
                           fsync(fileno(stdout));
```

```
                    }
                 // if we have send a seek-command that sends back
                 // a response (GPIB_LSEEK_RSPOLL for example) we do
                 // read the result
                 if(io_len & GPIB_LSEEK_RVALRCV)
                    {
                    len=read(fd,buffer,3);
                    buffer[3]=0;
                    printf("seek-response %d: \"%s\"\n",len,buffer);
                    }
                 }
              else
                 fprintf(stderr,"seek-position not given\n");
              } while(--repeat_cnt>0);
           break;
case 'r' :            /* receive data */
           do {
              if(repeat_cnt)
                 {
                 printf("numer of turns remaining: %d\n",repeat_cnt);
                 fsync(fileno(stdout));
                 }
              if(is_lseek)
                 {
                 // length parameter must be shifted by 8 bits to the left
                 //
                 status=gpib_lseek(fd,(io_len<<8) |GPIB_LSEEK_RLEN,is_split);
                 if(verbose)
                    {
                    printf("result of gpib_lseek(0x%0x): %d\n",io_len,status);
                    fsync(fileno(stdout));
                    }
                 }
              if(io_len && !is_lseek)
                 {
                 if(verbose)
                    {
                    printf("doing pread(,,%d,%d)\n",rlen,io_len);
                    fsync(fileno(stdout));
                    }
                 len=pread(fd,buffer,rlen,io_len);
                 }
              else
                 {
                 if(verbose)
                    {
                    printf("doing read(,,%d)\n",rlen);
                    fsync(fileno(stdout));
                    }
                 len=read(fd,buffer,rlen);
                 }
              if(len==-1)
                 {
                 fprintf(stderr,"Error %d while reading data: %s\n",
                         errno,strerror(errno));
                 fsync(fileno(stderr));
                 }
              else
                 {
                 buffer[len]=0;
                 printf("%d Bytes: \"%s\"\n",len,buffer);
                 fsync(fileno(stdout));
                 }
              } while(--repeat_cnt>0);
           break;
case 'w' :     /* send data */
           if(argp<argc)
              strcpy(buffer,argv[argp]);
           else
             *buffer=0;
           if(is_nl) // we add a newline if requested
              strcat(buffer,"\n");
           len=strlen(buffer);
           do {
              if(is_lseek)
                 gpib_lseek(fd,(io_len<<8) | GPIB_LSEEK_WLEN,is_split);
              if(io_len && !is_lseek)
                 status=pwrite(fd,buffer,len,io_len);
```

```
                    else
                       status=write(fd,buffer,len);
                    if(status==-1)
                       fprintf(stderr,"Error %d while writing data (%s): %s\n",
                               errno, buffer, strerror(errno));
                    else
                       printf("%d Bytes written\n",status);
                    } while(--repeat_cnt>0);
                 break;
       }
close(fd);
return(0);
}
```

Appendix E: Index