Gpib Programming Tutorial

Programming a GPIB based instrument designed at the Electronics Group

Free University *Amsterdam*, The Netherlands
Faculty of Sciences
Phone : +31-20-4447973
Fax : +31-20-4447899
E-mail : rob@nat.vu.nl (Rob Limburg)
Web : http://www.few.vu.nl/~elec

GpibProgTut.wpd (rev. 1.2)
Project-id [projectnummer]

Contents

1 Introduction

If you want to get started quickly and ignore most of the (background) information, go to **appendix A Quick start.**

1.1 Gpib bus

In 1965, Hewlett-Packard designed the Hewlett-Packard Interface Bus (HP-IB) to connect their line of programmable instruments to their computers. Today, the name General Purpose Interface Bus (GPIB) is more widely used than HP-IB. Despite its long established history, the bus is still very popular. Today, thousands of different instruments are equipped with this interface.

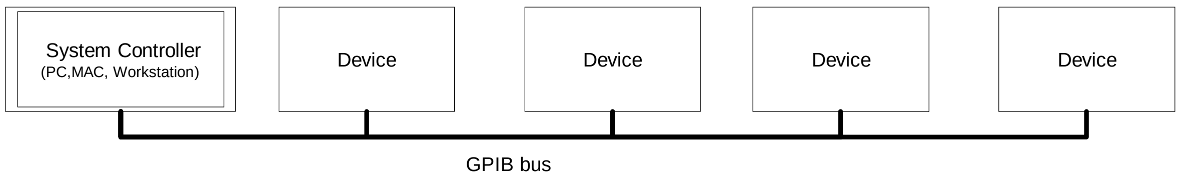
1.1.1 Gpib bus standard

The GPIB bus is standardised by the IEEE (Institute of Electrical and Electronics Engineers). The IEEE Standard 488-1975 defined the electrical and mechanical specifications. The ANSI/IEEE 488.2-1987 standard defined precisely how controllers and instruments should communicate.

Standard Commands for Programmable Instruments (SCPI) took the command structures defined in IEEE 488.2 and created a single, comprehensive programming command set that is used with any SCPI instrument.

1.2 Measurement setup

Typically, a measurement setup comprises a system controller (usually a PC or workstation) and at



least one device (instrument). The system controller has to have a gpib controller card. These cards can be bought from different vendors such as Hewlett-Packard, National Instruments, Computerboards, Ines and others)
The physical connection is a cable with standard connectors on both sides. The standard connector is the Amphenol or Cinch Series 57 MICRORIBBON or AMP CHAMP type. The connectors can be stacked to connect all the devices.

1.2.1 Restrictions

The following restrictions are typical for normal operation:

1. A maximum separation of 4 m between any two devices and an average separation of 2 m over the entire bus
2. A maximum total cable length of 20 m
3. No more than 15 device loads connected to each bus, with no less than two-thirds powered on

1.2 Gpib Instruments from our group

NeoNil

Most gpib instruments from our group are constructed around the NeoNil modular computer board. This computer features a G96 bus which can be used to add plug in cards. These plug in cards carry the special functionality needed to interact with the ‘outside world’ (e.g. experiment). With these plug

in cards an intelligent instrument can be build.
The NeoNil has an gpib interface which conforms to the IEEE488.2 standard.

Plug in cards
The G96 plug in cards can be ‘off the shelf’ developed by our group or can be bought from an plug in card vendor. Also, custom plug in cards can be developed on demand. These custom plug in cards come in two favours:

- Prototype quality. These cards are developed usually for incidental/special purposes. The cards are build using Wire-Wrap techniques on a prototype card. There is usually only one (1) card build.
- Production quality A PCB (printed circuit board) is designed. Large amount of these cards can be build.

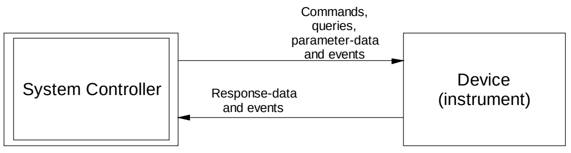
There is usually no difference in functionality or performance between the two types of cards. Only the production method differs.
Check the web pages of the Electronics Group for the available ‘off the shelf’ plug in boards or visit us at T012/T016.

The Software
The software is equally important as the hardware. All the software can be ‘custom’ developed, but we also maintain standard software components that we can assemble into a new applications.
Custom hardware needs custom software, so it is clear that for those parts new code has to be developed.
One major software component is the SCPI Engine. This software component allows us to build modular SCPI based software. All the command, query and data parsing and formatting is done by the SCPI Engine. This allows us to concentrate on the design of new modules rather than to be bothered by the syntax intricacies of SCPI and IEEE488.2.

The SCPI Engine supports the separation of independent software parts. In SCPI naming convention these are called ‘subsystems’. More on this later.

2 The communication principle

A GPIB instrument receives its information from another device. Usually this is the system controller. The instrument distinguishes between events, commands, queries and data. The figure on the right illustrates the flow of the various information types.



- **Events**. The GPIB bus has a number of special control lines which allow fast and simple passing of primitive information, called events. This primitive information is also called ‘Interface Events’ or ‘Interface Messages’ because they are initiated and processed by the interface hardware. An example of an event is the IFC (InterFace Clear) event, Which forces the instrument to be able to listen to the system controller. Events are discussed last in this chapter.
- **Commands**. Commands are strings which cause the instrument to perform an action, such as taking measurements or activate a motor.
- **Queries**. Queries are strings which cause the instrument to generate response. Queries usually do not cause an instrument to perform an action. Queries always end with a question mark (?).
- **Data units**. Data units are used to pass information. Data can be in the form of parameters which come with commands or queries *to* the instrument. Data can also be in the form of response *from* an instrument.

The GPIB bus operates block oriented, multiple commands, queries and data strings units can be combined into one line of text. An instrument will start executing commands and queries if a complete line of text has been received. To know when a complete line is transferred, the EOI (End Or Identify) event has to be generated simultaneously when the last byte is send.

Important note : Software drivers of some gpib cards default have EOI disabled! Configure the driver that EOI is enabled otherwise the instrument will not work!

2.1 Commands and queries

Commands and queries follow the same syntax rules, except that queries always end with a question mark. All commands and queries are build up using headers:

Header : A header is a string of maximum 12 characters. There is no distinction between upper and lowercase. The header may contain digits, but will always starts with a character or an underscore (_).

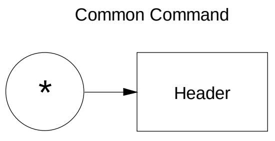
There are two types of commands and queries; the common and the compound types. If a header is longer than 4 characters, it may be abbreviated. See 3.3 Abbreviation on page 9.

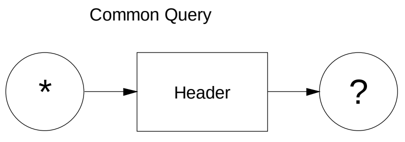
2.1.1 Common commands and queries

Common commands and queries are considered to be part of the IEEE488.2 and/or SCPI standard and thus are instrument independent. Common commands and queries are used for general ‘housekeeping’ and for the control of the status register structure.

Common commands and queries always start with an asterisk (*). They are simple, non hierarchical and can be used anywhere at any time.

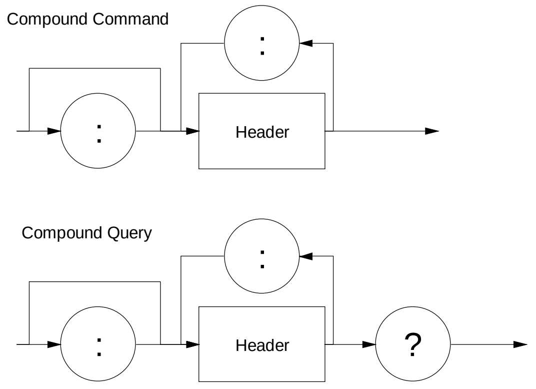
There are certain common commands and queries mandatory, others are optional. The standard common commands supported in our instruments can be found in chapter ‘Instrument standards’.





2.1.2 Compound commands and queries

The instrument specific functionality is controlled with compound commands and queries. Compound commands and queries represent a hierarchical structure also called a header tree.
Compound commands and queries are build up from one or more headers separated by colons (:). Each header represent a node in the tree, the leftmost header represents the highest level. The rightmost header should represent the 'leaf' node, which is the lowest level in the tree.
The compound query follows the same rules but has an question mark appended to its last header.



For example:
`:ad16_:trig:reset`
Resets the trigger mechanism of the AD16 plug in card.

Compound commands are further explained in 'Construction rules', from page 8.
As with common commands there are also standard compound commands and queries. These are all contained in so called *subsystems*.
Sub-systems are groups of commands and queries which are strongly related. For example, the AD16 plug in card has a corresponding SCPI subsystem called (suprise!) **`ad16_`** . The standard subsystems that are present in every programmable instrument from our group are listed in 'Instrument standards' at page 29.

2.2 Data

Data is used to pass information. Data is separated into parameters and responses.
Parameters are used to pass information along with commands and queries. This information is directed *towards* the instrument.
Responses are used to return information as a reply to queries. This information is directed *from* the instrument.
There are different types of data, they can be grouped into numerical, text and block data types. The purpose is to transfer information in a flexible, human oriented way. Almost all data types are ASCII encoded, only the binary block data type contains a data field with arbitrary encoding.

2.2.1 Numerical data

Numerical data is used to transfer decimal and non-decimal values.

Decimal numeric data

Decimal numeric data can be represented in three different formats:

- Integer numbers. This form is denoted as NR1 (Numeric Representation 1). Positive or negative whole numbers. Examples: 4 -23 90
- Real numbers. This form is denoted as NR2 (Numeric Representation 2). Fixed point numbers. Examples: 23.45 1.22 -4.55
- Exponential numbers. This form is denoted as NR3 (Numeric Representation 3). Floating point numbers. Examples: 4.3E-3 -8.9456E8 123E-5

Decimal numeric data may be received with greater precision than the capability of the device, such values are first rounded to the greatest precision possible.
For example: An power source can adjust its output voltage up to 0.1Volts, having a 100 Volt span. This power source receives a parameter to set the output voltage with the value of 41.46. The power source will round the value to 41.5 Volts and set the output voltage accordingly.

Non decimal numerical data

Positive integer numbers can be represented in non decimal numerical data. Three different formats are allowed for the bases 16 (hexadecimal), 8 (octal) and 2 (binary).

- Hexadecimal. Start with '#H' or '#h', followed by digits (0..9) and/or hex characters (A..F, a..f). Example the decimal value 63 is represented in the hexadecimal format as **`#H3F`** or as **`#h3f`**
- Octal. Start with '#O' or '#o', followed by digits (0..7). Example the decimal value 63 is represented in the octal format as **`#OH77`** or as **`#o77`**
- Binary. Start with '#B' or #'b', followed by digits (0..1). Example the decimal value 63 is represented in the binary format as **`#B111111`** or as **`#b111111`**

Additional information about the appliance of numerical data can be found in chapter 3 from page 8.

2.2.2 Text data

Text data can be separated into character data and string data.

Character data

Character data is used for information which can be best expressed as mnemonics. Character data follows the same rules as the headers described in Chapter 2.1:

"A header is a string of maximum 12 characters. There is no distinction between upper and lower-case. The header may contain digits, but will always starts with a character or an underscore (_). There are two types of commands and queries; the common and the compound types. If a header is longer than 4 characters, it may be abbreviated. See 3.3 Abbreviation on page 9."

Example:
A data acquisition instrument has 3 types of trigger sources available. To select one of these, the command **`TRIGger:SELect`** can be used. It accepts the following character data parameter values; **`BUS`** ,**`TIMer`** and **`EXTernal`**.

From this example it is clear that these character data values need almost no explanation.

String data

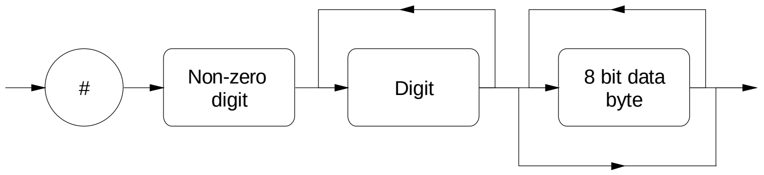
String data is used (in contrast to character data) for more free form textual data. String data may contain all 7 bit ASCII characters, including the non-printable ones. This data type is very useful when some form of formatting is required. For instance when text is to be displayed on a terminal or on a printer.
String data is always placed between single quotes (') or double quotes (").
When string data is delimited by single quotes, double quotes can be freely used. Also, when string data is delimited by double quotes, single quotes can be freely used.
Special care should be taken when single quotes should be used within string data delimited with single quotes. The rule here is to insert an extra single quote before the existing single quote. The two single quotes will be interpreted as one single quote character and not as an delimiter. The same rule also applies for double quote delimiters.

ASCII response data

ASCII response data is only used in query response directed *from* the instrument. It is not valid to use this data format as a parameter. In contrast to string data, ASCII response data is not delimited. It may contain any 7 bit ASCII characters.

2.2.3 Arbitrary Block data

Arbitrary block data can be used to transfer data fast without any encoding. There is no way to recover the encoding other than by some formal convention



(documentation).
The only encoding available is the block length in bytes. This is provided by the header which is ASCII encoded.
The first character in the header is the ‘#’ followed by a non-zero digit indicating the amount of digits to follow. The digits that follow represent the amount of data bytes in the arbitrary block data.
Example: (<b> represents a single byte in this example)

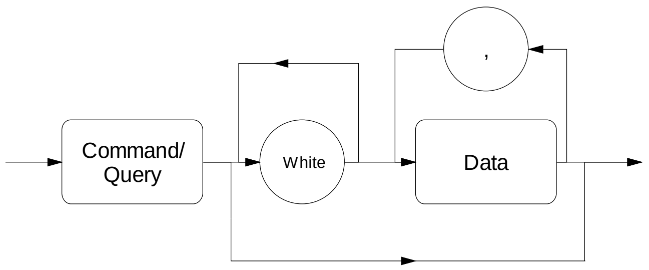
```
        #213<b><b><b><b><b><b><b><b><b><b><b><b><b>
```

the digit ‘2' means that the next 2 digits will form the block length. The block length is 13 bytes.

2.3 Combining commands and queries with data

In the previous sections the basic format of the commands, queries and data is defined. There was no mention *how* these elements can be combined into one line.

Now it is time to see how to combine commands, queries and data into one line. Only the basic format is given here, more information can be found in chapter 3 Construction rules.
A Command or a query can be combined with data. If there is data, the separation between the command or query and the data is done with one or more white spaces[1]. Multiple data elements are to be separated with comma’s (,).



Example:
A software timer can be programmed with a period and a repeat counter. The command to program a 100ms period and a repeat count of 5 could look like:

```
        timer 100,5         ok
        timer     100,5     ok, more spaces between header and data
        timer100,5          wrong! no space between header and data
        timer 100   , 5     ok, extra spaces between data elements
```

When a query returns multiple response data elements, these elements are also separated with comma’s (,). The above timer command example can also be given in the query form:
Controller sends:

```
        timer?
```

The instrument could reply:

```
        100,5
```

2.4 Events

The GPIB bus has a number of special control lines which allow fast and simple passing of predefined primitive information, called events. This primitive information is also called ‘Interface Events’ or ‘Interface Messages’ because they are initiated and processed by the interface hardware.
Events contain no data, they are either single control line signal levels(“single line events”) or certain data bus values combined with the ATN control line asserted (“multiline events”).
This section gives a short introduction of the most important events. In order to create a perfect measurement setup with optimum performance some knowledge of these events is inevitable. Check chapter 9 on page 30 for more information on further reading.

Some single line events are:

- IFC. InterFace Clear, forces the instrument to be able to listen to the system controller.
- REN. Remote ENable, signal line to claim control over an instrument which can also be

[1] A White space character is defined as a single ASCII character in the range 0..9, 0B..20 (hex).

operated by a local front panel.
- EOI. End Or Identify. When ATN = 0, used to indicate the end of a data transfer. When ATN = 1, used by the controller to perform a parallel poll.
- SRQ. Service ReQuest. This allows the instrument to get the attention of the controller. A very important feature of the GPIB. See 4.5 Service request on page 14.

Some multiline events are:
- DCL. Device CLear. Clears the input and output buffers in *all* instruments on the bus. See chapter 5 Reset and Clear on page 17.
- SDC. *Selected* Device Clear. Clears the input and output buffers in *one selected* instrument on the bus. See chapter 5 Reset and Clear on page 17.
- GET. Group Execute Trigger. All instruments on the bus receive a trigger pulse. See chapter 6 Trigger Mechanism on page 18.

3 Construction rules

In “2.3 Combining commands and queries with data” the basic construction rule for creating a command or query with data is defined.
The Construction rules defined here take it one step further. They are primary intended to make life easier. Some enhance readability, some enhance performance and others reduce programming effort.

3.1 More in one line

It is allowed to combine multiple commands and/or queries with their respective data into one single[2] line. This has several advantages, some of which become clear in a later stage of this chapter.
The first advantage is speed, the driver software in the controller and in the instrument have over-head to set up and process a data transfer. The bigger the blocks, the less the overhead per byte.
The second advantage can be clarity. When grouping related commands and queries into one line, the application program source can become much more readable.
To combine multiple commands and queries with their respective data, a separator must be used. The semicolon (;) is used as a separator between commands and or queries. This separator is officially called a “program message unit separator”. This separator is also used in the response to separate the query response data elements (if any).
Example:
Controller sends: **`:ad16_:version?;:system:version?`**
The instrument replies: **`1,5,0;1996.0`**

The first three data fields are from the first query, the result from the last query is separated by a semicolon (;).

3.2 Query form

All commands have a query counterpart, unless stated otherwise in the documentation. The query counterpart usually has no effect other than to return the current parameter value. See 3.8 Default, minimum and maximum for more possibilities with the query counterpart of a command.

It is also possible that only the query exists. In the documentation only the query form is printed (the header with the question mark). In that case the query can have other effects. For instance reading a value out of an FIFO (First In, First Out) buffer. This causes the value read to be disposed from the buffer.

3.3 White spaces

The usage of at least one white space is mandatory in some areas, for instance as a separator between commands/queries and data.
One or more white spaces may however be used in other areas too:

- Before a command or query.
- Surrounding the data separator (,). Example: **`trig:level 5   ,   10`**
- Surrounding the command/query separator (;). Example: **`*cls ;   *idn?`**
- Before the last byte send

It is **not** allowed: (to avoid common mistakes)

- Between compound command or query headers. Example: **`:ad16_: anout?`**
- Inside headers. Example: **`:ad 16_:anout?`**
- After the exponent character (‘E’ or ‘e’) in NR3 numeric data. Example: **`124.6E 8`**

[2] The length of a single line may never exceed the size of the instrument’s input buffer. An GPIB instrument designed at our group always has a query (SYSTem:INBuf?) which returns the actual size of the input buffer.

3.4 Omitting the header path

When using compound commands and/or queries it is possible in some cases to omit the leading header path. This is a very powerful feature because it can shorten the commands and queries considerable and reduces the search time within the header tree in the instrument.

The queries:

```
:system:version?;:system:err?
```

Omitting the header path for the second query:

```
:system:version?;err?
```

The header part (**`:system`**) of the second query may be omitted here. The second query does not specify it's absolute position (it does not start with (:)). Because the instrument's parser saves the last position in the header tree, it can find the **`err?`** query when searching from this saved position.

The queries:

```
:system:version?;:ad16_:anout?
```

Omitting the header path for the second query:

```
:system:version?;anout?
```

Will generate an error as the last saved position is in the **`:system`** branch, not in the **`ad16_`** branch. The **`anout?`** query will not be found at the **`:system`** level in the header tree.

Omitting the header path of a command or query is only possible within one single line. The first command or query on a line is assumed to be specified from the root. This also means that the colon leading the first header on a new line is optional.

Common commands and queries (beginning with a *) do not affect the saved position.

Consider the following:

```
:ad16_:trigger:count 100;:ad16_:trigger:delay 50;:ad16_:trigger:arm;*trg
```

Omitting the header paths:

```
ad16_:trigger:count 100;delay 50;arm;*trg
```

The first colon and the header paths for the **`delay`** and **`arm`** commands are omitted resulting in better performance and good readability.

3.5 Abbreviation

Compound commands or queries are build up using headers. These headers have a short form if their length exceeds 4 characters. The short form is always 4 characters or less.

The short form should be noted in the documentation. To distinct the short form from the long form the short form is printed in capitals, the remainder in non-capitals.

Example when the documentation defines:

```
:SYSTem
      :ERRor?
```

The following is:

`:syst:error?`	valid
`:system:err?`	valid
`:syst:err?`	valid
`:syste:err?`	invalid, **`:syste`** is not a short nor a long form

3.6 Default headers

A header tree can contain default headers. Default headers are headers which *may* be omitted. Default headers can be used to expand the header tree without losing software compatibility. Very useful if commands and/or queries need to be added to an existing system. Also, default headers can be used to shorten certain often used commands or queries.

Default commands or queries are printed between square brackets in the documentation.

Example from the docs:

```
:AD16_
      :ANIN
            [:READ]?     <numeric value>       (query only)
            :BLOCk?      <binary block data>   (query only)
            :POS
```

The following is:

```
      :ad16_:anin?          valid
      :ad16_:anin:read?     valid
```

Consider an instrument with the following header branch:

```
:SYSTem
      :CONFig
            :VIEW?
```

An updated version of this instrument has the same header branch expanded to:

```
:SYSTem
      :CONFig
            :VIEW
                  [:ALL]?      (new default header but same functionality)
                  :REGion      (new added header)
```

The **:SYSTem:CONFig:VIEW?** query will work on the old *and* on the new instrument.
The **:SYSTem:CONFig:VIEW:ALL?** will work on the new instrument only.

3.7 Forgiving listening and precise talking

Forgiving listening allows users to loosely specify what they want, whereas the instrument replies in a consistent manner. An instrument will round its received parameters to the nearest acceptable value. Consider the AD16 card which will accept positive integer values for its trigger counter.

Example 1:

```
ad16_:trig:count 3.45;count?
```

will return:

```
3
```

Example 2:

```
ad16_:trig:count 3.51;count?
```

will return:

```
4
```

Example 3:

```
ad16_:trig:count 3.51E01;count?
```

will return:

```
35
```

3.8 Default, minimum and maximum

It is possible to specify a maximum, minimum or default value for a most of parameters *without* knowing these values.
All numeric and character data types have a specified minimum, maximum and default value. The designer of the instrument defines the actual values. For instance the instrument will reset all its values to the defined default values when the instrument is powered up or when it receives a ***RST** common command.
The user can also specify these values, the instrument accepts the following for all its numeric and character data parameters:

MINimum	Will select the instrument’s predefined minimum value
MAXimum	Will select the instrument’s predefined maximum value
DEFault	Will select the instrument’s predefined default value

Example:
```
        ad16_:trig:count max;count?
```
will return:
```
        8192
```

Also the user can query these values:

Example:
```
        ad16_:trig:count? max
```
will return:
```
        8192
```

Note that in the first example count is set to its maximum value (8192) and then queried, in the second example the maximum value is queried, count is not modified.

3.9 Oops, typo!

Even with forgiving listening, mistakes are easy to make. To help determine what you did wrong, extensive error checking and error reporting is provided. When you start to work with a new instrument it is very convenient to use some interactive tool where you can type the commands and queries and see the response from the instrument directly. National Instruments has such a tool called simplegpib, for HP-VEE the 'GeneralTest' utility can be obtained from our group.

3.8.1 Error checking

Error checking is crucial for reliable operation of an instrument and it helps users to track down programming mistakes.
Error checking is performed on:

I. Syntax of headers, separators and parameters (generates command error)
II. Unknown headers (generates command error)
III. Illegal operation (generates execution error)
    1. Parameters out of range
    2. Trying to execute a (temporary) disabled command
    3. Illegal trigger operation
IV. Query processing (generates query error)
    1. Attempt to read response when nothing available
    2. Sending a new command or query without reading previous response
V. Internal problems (generates execution error or device dependant error)
    1. Hardware problems
    2. Software problems (eg. no free memory available)

Errors are reported as an error code with a short description in the error queue (see next section). Also, summary bits are available for each category (command, execution, device dependant, and query error) in the status system (see chapter 4 Instrument status).

3.8.2 The error queue

Each time an error occurs the corresponding error (or status) code and a short description (if possible) is stored in the error queue.
The error queue can be read by the user with the `:SYSTem:ERRor?` query. This query returns the first error code with description:

```
        <numeric value>,<string>
```

Error and status messages are queued using a FIFO (First-In, First-Out) strategy.
The numeric value is in the range [-32768, 32767]. An error/event value of zero indicates that no

error or event has occurred. If the queue overflows, the last error/event in the queue is replaced with the error:

```
        -350,”Queue overflow”
```

Reading an error/event from the queue removes that error/event from the queue. When all errors/events have been read from the queue, further error/event queries shall return:

```
        0,”No error”
```

Example:

Controller sends:

```
        syst:err?
```

The instrument could reply:

```
        -500,”Power up”
```

Controller sends:

```
        syst:err?
```

The instrument could reply:

```
        0,”No error”
```

Command errors specify the character position of the header which caused the error, so it should be fairly easy to find:
Example:
Controller sends:

```
        ad16_:triz:count 4
```

The instrument could reply:

```
        -113,”Undefined header; At position 7"
```

Position 7 is the start of the erroneous header `:triz` (which should be `:trig` or `:trigger`).

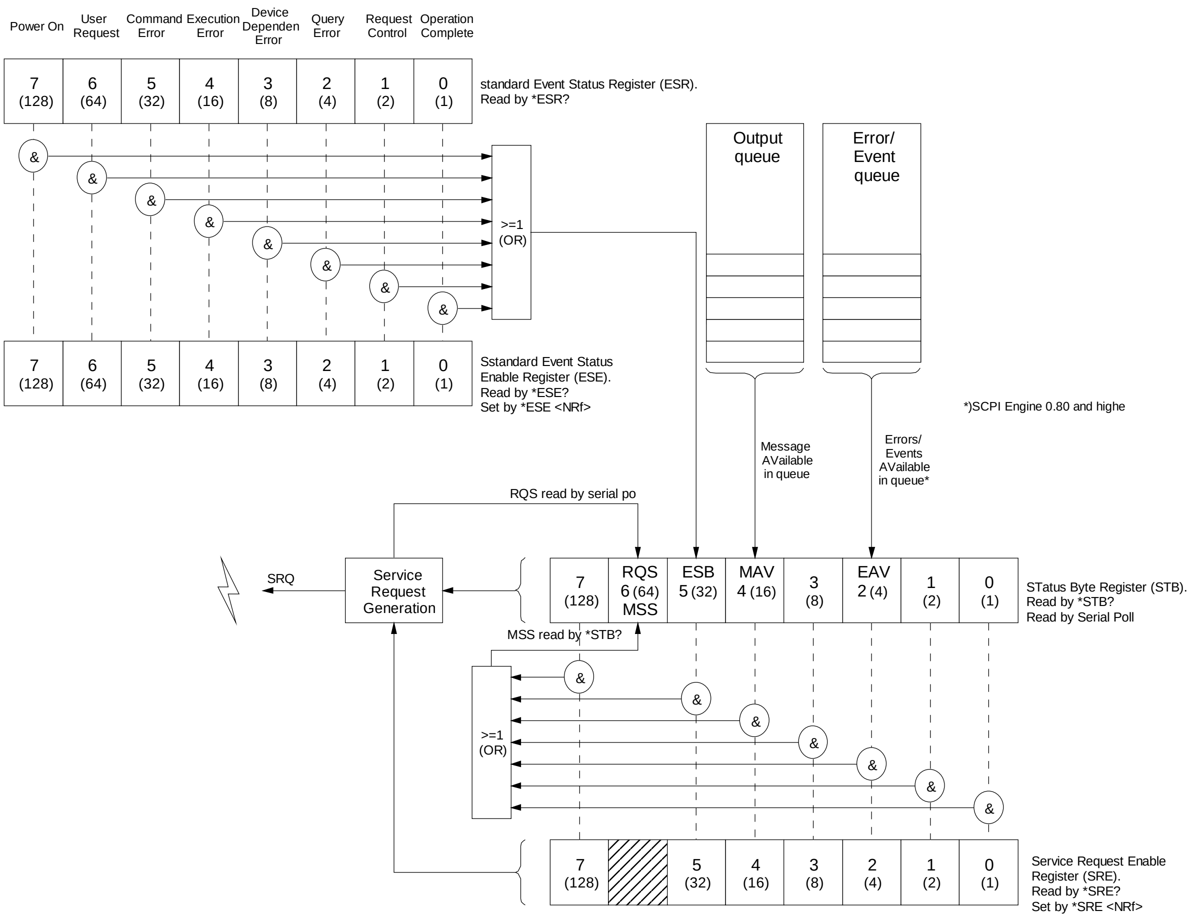
3.8.3 It hangs!
A requirement of an IEEE488.2 compatible instrument is that it will not talk when it has nothing to say. This means when an invalid query is received, the instrument will append an error in the error queue, but will **not** send any reply. The controller however, expects a response and will wait for it until some defined timeout value expires. After that it will probably issue an error to the user.
This is often referred to as a bug, but it isn’t.
After the controller’s timeout period the instrument can be programmed as usual. One can read the error queue and find out what caused the problem.

It is possible to get an IEEE488.2 instrument to hang, better speaking, to get it into a deadlock situation. Of course it is also possible that there is fault. To (try to) end such an situation it is best to first send an IFC (InterFace Clear) interface event and then a DCL (Device CLear). If this doesn’t help, pushing the reset button or cycling power is the only solution (sigh).

If you did not get the instrument to react in the first place check out if the controller card is configured as system controller, the gpib-id (default 15) is set correctly and insure that EOI (End Or Identify) is enabled in your program or software driver.

4 Instrument status

IEEE488.2 compatible instruments have to be equipped with a status structure, as shown on the figure below. The purpose of this structure is to supply information to the controller about the status of the instrument. Since this structure is part of the IEEE488.2 standard, all instruments will supply this information in the same consistent manner. Also, the status structure can be programmed in a way that a certain event will generate a SRQ (Service ReQuest, see 4.5).
The status can be modified and queried by a serial poll and several common commands and queries. Some parts of the status structure are controlled by the instrument itself, others can be manipulated by the common commands and queries.


Power On
User Request
Command Error
Execution Error
Device Dependen Error
Query Error
Request Control
Operation Complete
7 (128) 6 (64) 5 (32) 4 (16) 3 (8) 2 (4) 1 (2) 0 (1)
standard Event Status Register (ESR). Read by *ESR?
&
>=1 (OR)
Sstandard Event Status Enable Register (ESE). Read by *ESE? Set by *ESE <NRf>
Output queue
Error/ Event queue
*)SCPI Engine 0.80 and highe
Message AVailable in queue
Errors/ Events AVailable in queue*
RQS read by serial po
SRQ
Service Request Generation
7 (128) RQS 6 (64) MSS ESB 5 (32) MAV 4 (16) 3 (8) EAV 2 (4) 1 (2) 0 (1)
STatus Byte Register (STB). Read by *STB? Read by Serial Poll
MSS read by *STB?
7 (128) 5 (32) 4 (16) 3 (8) 2 (4) 1 (2) 0 (1)
Service Request Enable Register (SRE). Read by *SRE? Set by *SRE <NRf>

4.1 Status Byte registers (STB and SRE)

The status byte register (STB) is the centre of the status structure. The status register is the register which is polled in case of a serial poll (see 4.3). Also, the contents of the status register can cause the generation of a service request (see 4.2).
The status byte has an accompanying enable register, the Service Request Enable register (SRE). This register controls *which* bits of the STB could cause generation of a service request.
When the RQS bit in STB is set, a service request will be generated, see 4.2. The controller usually reacts by performing a serial poll, see 4.3.
Bits 0,1,2,3 and 7 are undefined in the IEEE488.2 standard, the other bits are:

Bit pos. Ordinal value   Meaning

6	64	RQS, ReQuested Service. This bit is set when the instrument has requested service by means of the SeRvice Request (SRQ). When the controller reacts by performing a serial poll, the STatus Byte register (STB) is transmitted with this bit set. Afand cleared afterwards. It is only set again when a new event occurs that requires service. MSS, Master Summary Status. This bit is a summary of the STB and the SRE register bits 1..5 and 7. Thus it is not cleared when a serial poll occurs. It is cleared when the event which caused the setting of MSS is cleared or when the corresponding bits in the SRE register are cleared.
5	32	ESB, Event Summary Bit. This is a summary bit of the standard status registers ESR and ESE.
4	16	MAV, Message AVailable. This bit is set when there is data in the output queue waiting to be read.

The STatus Byte register (STB) forms a basis in which more registers can have their summary bit. The ESB bit (5) is an example of that.

4.2 Serial poll

The controller has two ways of obtaining the STatus Byte register (STB) from an instrument, the *STB? query and a serial poll.
Serial poll is a special GPIB bus sequence to obtain the status byte register. The status byte register can also be obtained by the **`*STB?`** query, so why a special bus sequence for that?
One of the reasons is efficiency, the status register is usually incorporated inside the GPIB interface chip. This interface chip handles the serial poll completely. This means that the device specific operations are not interrupted by serial poll processing.
When a *STB? query is used, the instrument has to interrupt current processing, store the query, parse it, obtain the register, format it and send it to the controller. This difference can be crucial for reliable instrument operation.

4.3 Service Request

Service request is a powerful mechanism that enables an instrument to get the attention of the host computer (controller). The processing of the service request is simple but efficient. The instrument user controls by means of the status structure which events may cause a service request.
The user programmes the Service Request Enable register (SRE) with the bits from the Status Byte Register (STB) that may cause a service request. This can be data available in the output queue, some event(s) from the standard Event Status registers (ESR and ESE) or some instrument specific (bits 0..3,7).
When the controller detects a service request, it does not know *which* instrument caused the service request. So the first thing is to inquire who caused it. The controller does this by performing a serial poll on *all* instruments. The instrument which caused the service request has it’s RQS bit (6) set. The controller can further determine the cause of the instrument’s service request by examine the other bits in the status byte register. See 4.5 for examples.

4.4 Standard Event Status Registers (ESR and ESE)

The standard Event Status Register (ESR) contains the actual status information directly derived from the instrument. This register is read only. It can be read using the *ESR? common query. This query will return the contents of the register in NR1 (Numerical Representation #1) format.

The contents of this register will be cleared when read!

Bit pos.	Ordinal value	Meaning
7	128	Power-on. The power has cycled.
6	64	User request. The instrument operator has issued a request, for instance

		turning a knob on the front panel.
5	32	Command Error. A command error has occurred.
4	16	Execution error. The instrument was not able to execute a command for some reason. The reason can be that the supplied data is out of range but can also be an external event like a safety switch/knob or some hardware / software error.
3	8	Device Specific Error.
2	4	Query Error. Error occurred during query processing.
1	2	Request Control. The instrument is requesting to become active controller.
0	1	Operation Complete. The instrument has completed all operations. This bit is used for synchronisation purposes, see chapter 7 for more information.

As can be seen in the figure on page 13,the standard Event Status Enable register (ESE) is used to control which bits from the ESR are summarized in the ESB bit (5) in the Status Byte register (STB). By setting bit 5 in the Service Request Enable register (SRE), this can result in a service request. This ESE register is read/write. It can be controlled by the **`*ESE`** command and **`*ESE?`** query. The register is represented in NR1 (Numerical Representation #1) format.
Examples will follow in 4.5.

4.5 Using the status system in your application

The status system can help your application become more aware of the instrument's current state. A few examples illustrate some possibilities:

Example to detect command errors:

`*ESE 32`	Summarize command error bit (5) into ESB bit (5) of the status byte register
`blabla`	Generates a command error
`*STB?`	Query the Status Byte register

Returns:

`32`	The Event Status Bit (ESB) (5) is set.
`*esr?`	Get the contents of the ESR and clear it.

Returns:

`160`	160 = 128+32 thus the Power on bit and the command error bit are set
`*esr?`	Get the contents of the ESR and clear it.

Returns:

`0`	No error.

Example to detect command and query errors in a smarter way:

`*ESE 36`	Summarize command error bit (5) and the query error bit (2) into ESB bit (5) of the status byte register
`blabla?`	Just send the query, do not try to read the response as this could lead to a timeout when the query does not exist. This example will generate a query error.
SPOLL	Serial poll the instrument.

Returns:

`32`	The Event Status Bit (ESB) (5) is set, the MAV bit (4) is not set thus the query did not generate any response.

A stepper motor controller instrument uses bit 0 in the Status byte to indicate a jammed motor. The application on the host computer should be notified as soon as possible:

`*SRE 1`	Generate a service request when bit 0 in the status byte gets set.

Motor gets jammed:

SRQ	Instrument generates a service request.

Host computer serial polls all instruments, at a certain moment this one:

SPOLL	Serial poll the instrument.

Returns:

65	65= 64 + 1. This instrument caused the SRQ and has bit 0 set.

The host computer interprets bit 0, and takes the appropriate actions (turning off power, notify users ect).

5 Reset and Clear

A GPIB instrument can be initialized via commands and via interface events. Reset and clear is separated into several layers. The purpose is to control exact the part of the instrument that you want. For instance if the communication is stalled due to some query error, it is not needed to reset the whole instrument thereby losing all device specific settings.
The following list the possible ways to reset or clear (parts of) the instrument:

- Power on. This brings the whole instrument into the initial state
- InterFace Clear (IFC interface event). Resets the GPIB bus to its initial state.
- Device Clear (DCL or SDC interface event). Resets the information interchange between controller and instrument; clears input and output buffer, aborts operations that prevent processing of new commands/queries. After a device clear the instrument is ready to accept new commands and/or queries.
- Clear Status command (**`*CLS`**). Clears the whole status structure.
- Reset command (**`*RST`**). Resets the instrument specific functionality.

In order to get comparable results as cycling power, the controller must send an IFC, DCL or SDC, **`*CLS`** command and a **`*RST`** command.

6 Trigger mechanism

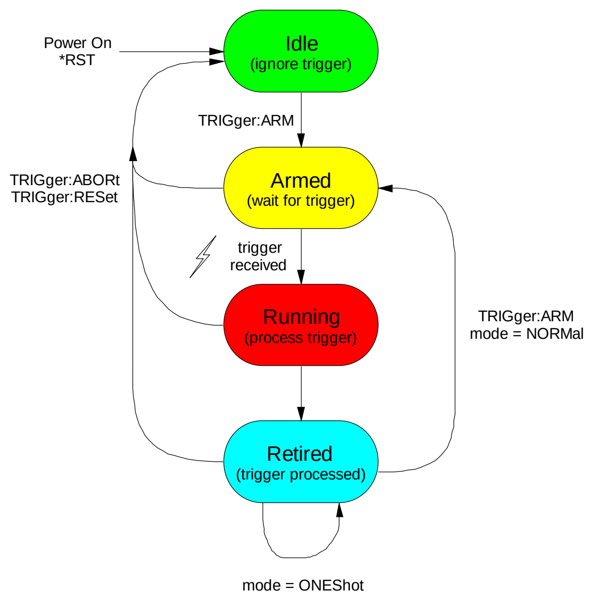
A GPIB instrument can be **BUS** triggered by means of a GET interface event, a **`*TRG`** common command. Triggering from any other (internal or external) signal is completely instrument specific. For an instrument GET and the *TRG command have the same effect but for the measurement setup there can be differences:

- The *TRG command has to be interpreted, which introduces software delays and jitter. The GET interface event needs less processing.
- GET applies to all instruments on the same GPIB bus, the **`*TRG`** command applies only to the addressed instrument.

Most of our instruments are modular. To allow flexible operation of our instruments a trigger mechanism had to be selected. This mechanism should allow each independent part of our instrument (subsystems, see xx) to be programmed in the way it reacts to a trigger.
The next section presents the default mechanism used in our instruments. If an instrument uses a different approach, refer to it's documentation.

6.1 Standard Mechanism

The triggering of most of our instruments follow or extent the state diagram below.



At power-on, system reset or **`*RST`** the trigger mechanism comes into the 'Idle' state. In this state all triggers are ignored.

When a **`TRIGger:ARM`** command is received, the trigger mechanism comes into the 'Armed' state. The subsystem is now ready to react on a trigger. In this state the trigger mechanism forbids modification of the trigger settings because it needs to react quickly to a trigger event.

When a trigger is received the mechanism comes into the 'Running' state. In this state the system is processing the trigger. Triggers received during that process are ignored and reported as errors.

When the trigger is processed, the mechanism comes into the 'Retired' state.
In the **`NORMal`** mode the mechanism falls directly to the 'Armed' state waiting for the next trigger to arrive.
In the **`ONEShot`** mode, it stays in the 'Retired' state until it is forced out. It can be forced into the 'Idle' state by the **`TRIGger:ABORt`** or the **`TRIGger:RESet`** command. It can also be forced back into the 'Armed' state by the **`TRIGger:ARM`** command.

The trigger mechanism is reset by the **`TRIGger:RESet`** command, it forces the mechanism to the 'Idle' state and resets the mode to **`NORMal`** and the source to **`BUS`**. The **`TRIGger:ABORt`** command just forces the 'Idle' state but leaves the mode and source settings unaffected.

7 Overlapped commands and synchronisation

Normally commands and queries are processed sequentially, they are processed in the order as they are received and the next one is started only when the current one has finished.
Sometimes the designed of an instrument decides to implement some commands to operate in background. When the associated command is received, a background task is started which will continue to run while the instrument can process new commands and queries send by the user. This increases flexibility but unfortunately also increases complexity. Therefore the designer will only implement this feature when there is real advantage to the user.

In the IEEE488.2 standard these background tasks are called overlapped commands, so this will be used here also.

An example of a good overlapped command implementation:

> Consider a (stepper) motor controller which implements a **`:HOMe`** command to bring the motor in the homing position . This operation can take a number of seconds. Normally the motor controller would be unable to handle commands and queries during that operation. By implementing this command 'overlapped' the controller can perform other requests while the motor is moving towards home.

To consider a potential problem:

> The same instrument also implements the **`:FORWard`** command which moves the motor forward over a specified distance. Imagine that the motor controller is moving the motor backwards towards the home position after receiving the **`:HOMe`** command. At that time the motor controller receives the **`:FORWard`** command. It would be undesirable to abort the homing process and move the motor forwards. There has to be some synchronisation between commands to ensure that this kind of problems do not occur.

Hopefully the designer of an instrument will implement the necessary synchronisation to correctly operate the instrument. In the previous example there was mention about synchronisation between commands however there can also be need for synchronisation between host computer and instrument.
For implementing synchronisation between host computer (controller) and instrument the IEEE488.2 standard has defined two commands and a query.

Cmnd/Query	Meaning	When to use
`*WAI`	Wait command. Wait until all pending commands and queries are processed. The following **`:HOME;*WAI;:FORWARD`** ensures that **`:FORWARD`** is not started until **`:HOME`** has finished.	To force synchronisation between commands and/or queries. (the controller will not know the length of the waiting period **`*WAI`** introduces).
`*OPC?`	OPeration Complete query. This query returns '1' when all pending commands and/or queries are finished. During the period that there are pending commands and/or queries this query gives *NO* response! While the controller is waiting for the response the whole bus is stalled until the instrument is idle (or the controller times out).	Synchronisation between controller and instrument for *short* waiting periods. When the delay is too long (several seconds) a controller timeout can occur resulting in an error situation. So it is not advisable for unpredictable or long delays.

***OPC**	OPeration Complete command. Set the Operation Complete bit in the Event Status Register to '1' when all pending commands and/or queries are finished. The controller can read this bit with the *ESR? query. The controller can also manipulate the status system in a way that, for instance, the instrument will notify the controller with a SRQ.	Synchronisation between controller and instrument. Suitable for short, long and unpredictable delays. Knowledge of the status system mandatory.

8 Instrument standards

8.1 Standard common commands and queries

The following table lists the available common commands and queries available in all GPIB based instruments from our group.

General

***RST**	Reset command. Abort all activities and initialize the device. See page 17,Reset and clear.
***IDN?**	Identification query. Returns an identification string in the following format: 'Manufacturer, Model, Serial number, Firmware level'
***TST?**	Self test query. Perform a self-test. Returns '0' if self test completed without errors, all other values determine an error cause.
***TRG**	Trigger command. Execute trigger function(s). This command has the same effect as the IEEE488.1 Group Execute Trigger (GET) interface-message. See page 18 Trigger Mechanism.

Synchronisation (See page 19, Overlapped commands and synchronisation)

***OPC**	Operation Complete command. Set the Operation Complete bit in the Standard Event Status Register if all commands and queries are finished.
***OPC?**	Operation Complete query. Return ASCII '1' as soon as all commands and queries are finished.
***WAI**	Wait To Continue command. Wait until all commands and queries are finished.

Instrument Status (See page 13,Instrument status)

***CLS**	Clear Status command. Clears the whole status structure.
***ESE**	Standard Event Status Enable command. Modify the contents of the Event Status Enable Register.
***ESE?**	Standard Event Status Enable query. Return the contents of the Event Status Enable Register.
***ESR?**	Standard Event Status Register query. Return the contents of the Event Status Register.
***SRE**	Service Request Enable command. Modify the contents of the Service Request Enable Register.
***SRE?**	Service Request Enable query. Return the contents of the Service Request Enable Register.
***STB?**	Status Byte query. Return the contents of the Status Byte Register.

8.2 Standard subsystems

8.2.1 Introduction

As mentioned before, the header tree is a hierarchical structure that can comfortably reflect the modular structure of our instruments. The header tree is divided into *subsystems*.
Sub-systems are groups of commands and queries which are strongly related. For example, the AD16 plug in card has a corresponding SCPI subsystem called **ad16_** . The standard subsystems that are present in every programmable instrument from our group are listed here.

8.2.2 System sub-system

:SYSTem (Node)
The SYSTem subsystem provides the user with commands and queries that are not related to instrument performance.

:ERRor? (Query only)
Get the next entry from the instrument's error/event queue. Returns <error/status value>, an integer, followed by <description> which is a string.

Error and status messages are queued using a FIFO (First-In, First-Out) strategy.
Items in this queue contain an integer in the range [-32768, 32767] denoting an error/event number and associated descriptive text. An error/event value of zero indicates that no error or event has occurred. If the queue overflows, the last error/event in the queue is replaced with the error:

```
-350,”Queue overflow”
```

Reading an error/event from the queue removes that error/event from the queue. When all errors/events have been read from the queue, further error/event queries shall return:

```
0,”No error”
```

Example:

Controller sends:	**:syst:err?**
Instrument could reply:	**-500,”Power up”**

Controller sends:	**:syst:err?**
Instrument could reply:	**0,”No error”**

See also ‘Oops typo!’ from page 11.

:DATE **<numeric_value>,<numeric_value>,<numeric_value>**
Queries or sets the internal calender. The first parameter is the year, followed by the month and the day.

Example:

Controller sends:	**:syst:date?**
instrument could reply:	**99,12,31**

Example:

Controller sends:	**:syst:date 99,1,1**

Date will now be set to januari 1st 1999.

:TIME **<numeric_value>,<numeric_value>,<numeric_value>**
Queries or sets the internal clock. The first parameter is the hour, followed by the minute and the second.

Example:

Controller sends:	**:syst:time?**
Instrument could reply:	**16,48,14**

Example:
Controller sends: **`:syst:time 23,59,1`**
Time will now be set to 23:59:1.

`:VERSion?` (Query only)
Returns the SCPI version number for which this instrument complies.

Example:
Controller sends: **`:syst:vers?`**
Instrument could reply: **`1996.0`**

8.2.3 Micros sub-system

`:MICRos` (Node)
The Micros subsystem provides the user with some support for the underlying Micros operating system.

`:INFo` (Node)
INFo commands provide some information about the underlying hardware and operating system.

`[:ID?]` (Default, Query only)
Returns a string containing operating system identification information.
Example
Controller sends: **`:micros:info:id?`**
Instrument could reply: **`Micros 3.2.3, (c) Rob Limburg Vrije Universiteit Amsterdam`**

`:CPU?` (Query only)
Returns a string containing microprocessor identification information.
Example:
Controller sends: **`:micros:info:cpu?`**
Instrument could reply: **`Scc68070`**

`:RELease?` (Query only)
Returns an unsigned integer (NR1) indicating operating system release.
Example:
Controller sends: **`:micros:info:rel?`**
Instrument could reply: **`323`**

`:SYSTem?` (Query only)
Returns a string containing hardware platform identification information.
Example:
Controller sends: **`:micros:info:syst?`**
Instrument could reply: **`NeoNil`**

`:MEM` (Node)
MEM commands provide information about the amount of available memory (RAM) in the system.

`:CONTigous?` (Query only)
Returns an unsigned integer (NR1) indicating the size (in bytes) of the biggest contiguous block of

free memory.
Example:
Controller sends: **`:micros:mem:cont?`**
Instrument could reply: **`1558206`**

`[:TOTal?]` (Default, Query only)
Returns an unsigned integer (NR1) indicating the total amount (in bytes) of free memory.
Example:
Controller sends: **`:micros:mem:tot?`**
Instrument could reply: **`1844708`**

`:STReam` (Node)
STReam commands give the user the ability to control the (other) communication channels available on the system. Streams are flexible data channels provided by the Micros operating system. A stream is opened by calling the OPEN query. This query returns an unsigned integer. This value has to be passed with all subsequent STReam commands and queries.

`:OPEN` **`<string>,<string>`** (Query only)
Open a stream with the given stream and protocol-name. The first string has to contain the stream name and the second string the protocol name. This query returns a non decimal unsigned integer which is the *stream-handle*.
If the stream could be opened, the stream-handle is unequal to 0 and FFFFFFFF$_{hex}$. If the stream could not be opened, an error is appended to the error/status queue.
Example:
Controller sends: **`:micros:stream:open “SERIAL1”,”RS232"`**
instrument could reply: **`#H1DFD4`**
Some of the available streams are listed at the end of this section.

`:CLOSe` **`<numeric_value>`**
Close the stream. If the stream could not be closed, an error is appended to the error/status queue.
Example:
Controller sends: **`:micros:stream:close #H1DFD4`**
Now the stream is closed, output buffer will be flushed, input buffer cleared.

`:READ?` **`<numeric_value>`** (Query only)
Reads data from the stream referenced by the <numeric_value>. This query returns the data in the arbitrary block form.
Note that this query is non-blocking, only data resident in the input buffer is returned.
Example:
Controller sends: **`:micros:stream:read? #H1DFD4`**
Instrument could reply: **`#211Hello dude!`**

`:WRITe` **<numeric_value>,<arbitrary_block>**
Sends data to the stream referenced by <numeric_value>. The data is in arbitrary block form.
Example:
Controller sends: **`:micros:stream:write #H1DFD4, #212Hello world!`**
The data will now be sent to the stream.

Some of the available streams on the NeoNil computerboard:

SERIAL1 (rs232 port 1 on front panel NeoNil):

Stream	'SERIAL1'
Protocol	'RS232[:OPTION1=X][:OPTIONn=Y]'

Protocol options:

B=X[,Y]	Buffer size in bytes. B=X bytes voor input en output buffer, B=X/Y X bytes voor input en Y bytes voor output.
I=X	Interrupt priority level 0..32767
R=X	BaudRate = X bits/sec, possible values: 50,75,110,134,150,200,300,600,1200,1050,1800, 2000,2400,4800,7200,9600,19200 of 38400
P=NO ODD EVEN	Parity off, odd or even.
C=X	Character length = 5,6,7 or 8 bits.
S=X	Stopbit length = 1 or 2 bits
F=NO HARD SOFT	Flow control = none, hardware(RTS-CTS) or software (Xon-Xoff).

Default configuration:

buffer size	1Kbyte
interrupt prio.	30000
baud rate	9600 baud
parity	No
char. length	8
stop bit length	1
flow control	XonXoff

SERIAL2 (rs232 port 2 on front panel NeoNil):
Stream 'SERIAL2'
Protocol 'RS232[:OPTION1=X][:OPTIONn=Y]'

Protocol options:

B=X[,Y]	Buffer size in bytes. B=X bytes voor input en output buffer, B=X/Y X bytes voor input en Y bytes voor output.
L=X	Interrupt Level 0..7
I=X	Interrupt priority level 0..32767
R=X	baudRate = X baud: 75,150,300,1200,2400,4800,9600 or 19200 baud.
P=NO ODD EVEN	Parity off,odd or even.
C=X	Character length = 5,6,7 or 8 bits.
S=X	Stopbit length = 1 or 2 bits.
F=NO HARD SOFT	Flow control = none, hardware(RTS-CTS) or software (Xon-Xoff).

Default configuration:

buffer size	1Kbyte
interrupt level	3
interrupt prio.	30000
baud rate	9600 baud
parity	No
char. length	8
stop bit length	1
flow control	XonXoff

9 Error codes

Errors are divided into command errors, execution errors, query errors and instrument specific errors. Refer to section 3.9 for more information on errors. Here only the codes and the meanings are defined. Not all codes listed here can be generated by our instruments.
Negative error codes are defined by SCPI, positive numbers are instrument specific.

0	No error

Command errors. The command error bit in the standard Event Status Register (ESR) is set to ‘1' when such an error occurs.

-100	Command error (one of the following is used if approriate)
-101	Invalid character. An element contains a character which is invalid for that type of element. For example a command containing am ampersand: **`:SETUP&`**
-102	Syntax error. An unrecognized command or data type was encountered.
-103	Invalid separator. The instrument’s parser was expecting a separator and encountered an illegal character. For instance the semicolon was omitted after a program message unit.
-104	Data type error. The instrument’s parser detected a data element of different type than expected. For example, a numerical data element was expected but a arbitrary block data element was detected.
-105	GET not allowed. A group execute trigger interface event was detected within a line of commands and/or queries.
-108	Parameter not allowed.
-109	Missing parameter.
-110	Command header error.
-111	Header separator error
-112	Program header too long.
-113	Undefined header.
-114	Header suffix out of range.
-120	Numeric data error.
-121	Invalid character in number.
-123	Exponent too large.
-124	Too many digits.
-128	Numeric data not allowed.
-130	Suffix error.
-131	Invalid suffix.
-134	Suffix too long.
-138	Suffix not allowed.
-140	Character data error
-141	Invalid character.
-144	Character data too long.
-148	Character data not allowed
-150	String data error.
-151	Invalid string data
-158	String data not allowed
-160	Block data error
-161	Invalid block data
-168	Block data not allowed
-170	Expression error
-171	Invalid expression
-178	Expression data not allowed
-180	Macro error.
-181	Invalid outside macro definition.
-183	Invalid inside macro definition
-184	Macro parameter error

Execution errors. The execution error bit in the standard Event Status Register (ESR) is set to ‘1' when such an error occurs.

-200	Execution error.
-201	Invalid while in local
-202	Settings lost due to rtl
-203	Command protected
-210	Trigger error
-211	Trigger ignored
-212	Arm ignored
-213	Init ignored
-214	Trigger deadlock
-215	Arm deadlock
-220	Parameter error
-221	Settings conflict
-222	Data out of range
-223	Too much data
-224	Illegal parameter
-225	Out of memory
-226	Lists not same length
-230	Data corrupt or stale
-231	Data questionable
-232	Invalid format
-233	Invalid version
-240	Hardware error
-241	Hardware missing
-250	Mass storage error
-251	Missing mass storage
-252	Missing media
-253	Corrupt media
-254	Media full
-255	Directory full
-256	Filename not found
-257	File name error
-258	Media protected
-260	Expression execution error
-261	Math error in expression
-270	Macro exec error
-271	Macro syntax error
-272	Macro execution error
-273	Illegal macro label
-274	Macro parameter execution error
-275	Macro definition too long
-276	Macro recursion error
-277	Macro redefinition not allowed
-278	Macro header not found
-280	Program error
-281	Cannot create program
-282	Illegal program name
-283	Illegal variable name
-284	Program currently running
-285	Program syntax error
-286	Program runtime error
-290	Memory use error
-291	Out of program memory
-292	Referenced name does not exist
-293	Referenced name already exist

-294	Incompatible type

Device specific errors. The device dependant error bit in the standard Event Status Register (ESR) is set to ‘1' when such an error occurs.

-300	Device specific error
-310	System error
-311	Memory error
-312	Pud memory lost
-313	Calibration memory lost
-314	Save recall memory lost
-315	Configuration memory lost
-320	Storage fault
-321	Out of internal memory
-330	Self test failed
-340	Calibration failed
-350	Queue overflow
-360	Communication error
-361	Parity error
-362	Framing error
-363	Input buffer overrun

Query errors. The query error bit in the standard Event Status Register (ESR) is set to ‘1' when such an error occurs.

-400	Query error
-410	Query interrupted
-420	Query unterminated
-430	Query deadlocked
-440	Query unterminated

Other SCPI defined error values. The corresponding bit in the standard Event Status Register (ESR) is set to ‘1' when such an event occurs.

-500	Power on
-600	User request
-700	Request control
-800	Operation complete

10 Software revisions

A programmable instrument from our group is a modular design. There is no such thing as 'the' instrument software revision. Instead, a few software revisions are mentioned here which could be of interest of the user.

SCPI Engine revision
The SCPI Engine is the 'engine' of the programmable instrument's command and query processing. The revision level is contained in the **`*IDN?`** query response.

GPIB stream revision
This software revision is only used internally. The GPIB stream handles all interaction with the GPIB bus (GPIB bus driver).

SCPI standard version
The SCPI standard itself is changing over time. The instruments developed in our group are (at this time) mostly SCPI 1996 complient.
The SCPI standard version is returned by the **`:SYSTem:VERSion?`** query.

Sub-system version(s)
Important (long lasting with more or less continuous development) subsystems have version levels. This software revision is returned by the

```
:<subsys-name>:VERSion?
```

query. It returns three unsigned numeric values representing the version level of this subsystem:

```
<major>,<minor>,<patch>
```

The ***major*** version indicates the level on overall behaviour of the subsystem. A major level 2 subsystem is likely to give problems when expecting a major level 1.
The ***minor*** version level indicates the syntax level. A minor level 2 subsystem is likely to have more commands or queries than a minor level 1 subsystem. It could also have some minor modifications to existing commands. Your application software (LabView, HP-VEE, etc) should work with little or no modifications.
The ***patch*** version level indicates the changes to the subsystem which do not affect the behaviour, and do not add or modify the existing commands.
Example:

Controller sends:	**`ad16_:vers?`**
Instrument could reply:	**`1,3,3`**

11 Further reading

The IEEE488.2 and SCPI standard documents:

- ANSI/IEEE Std 488.1-1987, IEEE Standard Digital Interface for Programmable Instrumentation.
  Publisher: The Institute of Electrical and Electronics Engineers, New York
  ISBN : 471-62222-2
- ANSI/IEEE Std 488.2-1987, IEEE Standard Digital Interface for Programmable Instrumentation.
  Publisher: The Institute of Electrical and Electronics Engineers, New York
  ISBN : -
- SCPI 1996, Standard Commands for Programmable Instruments.
  Publisher: SCPI Consortium, La Mesa
  ISBN: -

More readable:

- Automatic Measurement Control, a tutorial on SCPI and IEEE488.2 by John M.Pieper.
  Publisher: ACEA / Rohde & Schwarz
  ISBN : -

From our group (download at our web site):

The AD16 data acquisition board.
- AD16 SCPI Subsystem.
- AD16 SCPI Quick Reference Card

The STEPPER, RUNNER and TWISTER motion control boards.
t.b.a.

Appendix A Quick Start

This appendix shows you how to get started with an programmable instrument from the Electronics Group.

A1 What you need

- A computer with a gpib controller card (we can advise you which card to purchase)
- A gpib cable (we can provide this)
- The instrument with a power cord
- 240 Volts outlets with protective ground

A2 What you need to know before you begin

- Never connect two computer (peripherals) while power is on.
- Always use instruments in a 240Volts protective ground connection
- Make sure the protective grounds are from the same group (tie them as close to each other as possibly can)

A2 What you need to do

I. If it is not done already, install the gpib controller card in the computer. Follow the directions of the manufacturer. Make sure it is configured as **system-controller**.
II. Place the instrument at the desired location, connect the cable.
III. Boot the computer.
IV. Select the **gpib-id** on the front panel of the instrument (default 15)
V. Turn on the instrument
VI. Make sure the software driver in your computer has **EOI** (End Or Identify) **enabled**!
VII. Use a software tool to communicate with the instrument.

Some notes:

Most gpib card manufactures have some simple tool to do troubleshooting. You can simply enter a command or query and see the results.
The Electronics Group also has a simple utility written in HP-VEE. This can be downloaded from our web site.