FCA3000, FCA3100, MCA3000 Series Timer/Counter/Analyzers

Programmer Manual

077-0494-00

Tektronix

FCA3000, FCA3100, MCA3000 Series Timer/Counter/Analyzers

Programmer Manual

www.tektronix.com

077-0494-00

Tektronix

Contacting Tektronix

Tektronix, Inc.
14150 SW Karl Braun Drive
P.O. Box 500
Beaverton, OR 97077
USA

For product information, sales, service, and technical support:

- In North America, call 1-800-833-9200.
- Worldwide, visit www.tektronix.com to find contacts in your area.

Table of Contents

Preface iii

Getting Started

Setting Up the Instrument 1-1
Interface Functions 1-2
Using the USB Interface 1-3

Syntax and Commands

Command Syntax 2-1
- Command and Query Structure 2-1
- Clearing the Instrument 2-2
- Command Entry 2-3
- Argument Types 2-4
- Macros 2-6

Command Groups 2-11
- Arming Subsystem 2-11
- Calculate Subsystem 2-11
- Calibration Subsystem 2-13
- Configure Function 2-13
- Display Subsystem 2-14
- Fetch Function 2-14
- Format Subsystem 2-15
- Hard Copy 2-15
- Initiate Subsystem 2-16
- Input Subsystem 2-16
- Measurement Subsystem 2-17
- Memory Subsystem 2-19
- Output Subsystem 2-20
- Read Function 2-20
- Sense Command Subsystem 2-20
- Status Subsystem 2-21
- System Subsystem 2-22
- Test Subsystem 2-23
- Trigger Subsystem 2-23
- Common Commands 2-27

Command Descriptions 2-29

Status and Events

Status and Events 3-1
Registers 3-1
Queues 3-4
Event Handling Sequence 3-4
Synchronization Methods 3-6
Error Messages 3-10

Programming Examples

Programming Examples 4-1
Introduction 4-1
Individual Measurements (Example #1) 4-2
Block Measurements (Example #2) 4-4
Fast Measurements (Example #3) 4-6
USB Communication (Example #4) 4-9
Continuous Measurements (Example #5) 4-11

Appendices

Appendix A: Character Set A-1
Appendix B: Default Command Settings B-1
Appendix C: Instrument Settings After *RST C-1
Appendix D: Reserved Words D-1

Preface

This programmer manual covers the Tektronix FCA3000, FCA3100, and MCA3000 Series Timer/Counter/Analyzer instruments. It provides information on operating your instrument using the General Purpose Interface Bus (GPIB) or USB interface.

The programmer manual contains the following sections:

- Getting Started. This section introduces you to the programming information and provides basic information about setting up your instrument for remote control.
- Syntax and Commands. This section describes the command syntax structure, provides tables that list all the commands by functional groups, and describes all commands in alphabetical order.
- Status and Events. This section discusses the status and event reporting system for the GPIB interfaces. This system informs you of certain significant events that occur within the instrument. Topics that are discussed include registers, queues, event handling sequences, synchronization methods, and messages that the instrument may return, including error messages.
- Programming examples. This section provides examples of C code used to take measurements with the instruments.
- Appendices. The appendices contain miscellaneous information such as a list of reserved words, a table of the factory initialization (default) settings, and interface specifications.

Getting Started

Setting Up the Instrument

Setting the GPIB Address

The default GPIB address of the instrument is 10. Push **USER OPT > Interface** to see the active address above the soft key button labeled **GPIB address**.

To change the instrument GPIB address, push **GPIB address** and enter a new address value between 0 and 30. The GPIB address is stored in nonvolatile memory and remains until you change it.

You can also set the GPIB address remotely by using a GPIB command.

Standby Power and Remote Access

When the instrument is in REMOTE mode, you cannot power it off from the power button. You must first push the **Esc** button to enter Local mode, and then push the **Power** button.

Testing the Bus

To test that the instrument is operational over the bus, send the *IDN? command to identify the instrument and the *OPT? command to identify which features are available.

Interface Functions

Table 1-1: Interface function summary

Code	Description
SH1	Source handshake: The instrument can exchange data with other instruments or a controller using the bus handshake lines DAV, NRFD, and NADC.
AH1	Acceptor handshake: The instrument can exchange data with other instruments or a controller using the bus handshake lines DAV, NRFD, and NADC.
C0	Control function: The instrument does not function as a controller.
T6	Talker function: The instrument can send responses and the results of its measurements to other devices or to the controller. T6 has the following functions: ■ Basic talker ■ No talker only ■ Send out a status byte as response to a serial poll from the controller ■ Automatic unaddressing as a talker when it is addressed as a listener
L4	Listener function: The instrument can receive programming instructions from the controller. L4 has the following functions: ■ Basic listener ■ No listen only ■ Automatic unaddressing as listener when addressed as a talker
SR1	Service request: The instrument can call for attention from the controller, such as when a measurement is completed and a result is available.
RL1	Remote/local function: You can control the instrument manually (locally) from the front panel or remotely from the controller. The LLO, local-lock-out function, can disable the LOCAL button on the front panel.
PP0	Parallel poll: The instrument does not have any parallel poll facility.
DC1	Device clear function: The controller can reset the instrument by sending the interface message DCL (Device clear) or SDC (Selective Device Clear).
DT1	Device trigger function: You can start a new measurement from the controller by sending the interface message GET (Group Execute Trigger).
E2	Bus drivers: The GPIB interface has tri-state bus drivers.

Using the USB Interface

The instrument is equipped with a USB full speed interface, which supports the same command set as the GPIB interface.

The USB interface is a full speed interface (12 Mbit/s), supporting the industry standard USBTMC (Universal Serial Bus Test and Measurement Class) revision 1.0, with the subclass USB488, revision 1.0. The full specification for this protocol is at www.usb.org.

A valid driver for this protocol must be installed to be able to communicate over USB. We recommend NI-VISA version 3.2 or above, which is available from National Instruments (www.ni.com) for several operating systems. The Windows version is supplied on the product CD.

In order to test the communication and send single commands, use the National Instruments utility supplied with the NI-VISA drivers to open a VISA session to send and receive data from the instrument, and also set control signals such as Remote or Local.

Third party application programs, such as LabView, normally support USB communication directly, for example through the Instrument I/O Assistant.

Custom specific programs using USB communication can be written in C/C++, supported by libraries and lib-files supplied with the NI-VISA driver (default location C:\VXIPNP\WinNT\). A sample program is in included in the Examples section. (See page 4-9, *USB Communication (Example #4)*.)

Instruments connected to the USB bus are identified by a unique vendor identifier, the instrument model number and the instrument serial number. The structure of the instrument identifier string is:

"`USB0::0x0699::0x3003::######::INSTR`"

Where:

- `0x0699` is the vendor identifier code for Tektronix instruments
- `0x3003` is the instrument model (based on the last four digits of the model number)
- `######` is the instrument serial number

Use this string to identify the instrument vendor, model, or serial number when searching for or connecting to a specific instrument.

Syntax and Commands

Command Syntax

You can control the operations and functions of the instrument through the GPIB port or the USB 2.0 device port using commands and queries. The related topics listed below describe the syntax of these commands and queries. The topics also describe the conventions that the instrument uses to process them. See the *Command Groups* topic in the table of contents for a listing of the commands by command group, or use the index to locate a specific command.

Backus-Naur Form Notation

This documentation describes the commands and queries using Backus-Naur Form (BNF) notation. The following table lists the BNF notation symbols.

Table 2-1: Symbols for Backus-Naur form

Symbol	Meaning
< >	Defined element
=	Is defined as
|	Exclusive OR
{ }	Group; one element is required
[ ]	Optional; can be omitted
. . .	Previous element(s) may be repeated
( )	Comment

Command and Query Structure

Commands consist of set commands and query commands (usually called commands and queries). Commands modify instrument settings or tell the instrument to perform a specific action. Queries cause the instrument to return data and status information.

Most commands have both a set form and a query form. The query form of the command differs from the set form by its question mark at the end. For example, the set command `ACQuisition:HOFF` has a query form `ACQuisition:HOFF?`. Not all commands have both a set and a query form. Some commands have set only and some have query only.

Messages

A command message is a command or query name followed by any information the instrument needs to execute the command or query. Command messages may contain five element types, defined in the following table.

Table 2-2: Command message elements

Symbol	Meaning
<Header>	This is the basic command name. If the header ends with a question mark, the command is a query. The header may begin with a colon (:) character. If the command is concatenated with other commands, the beginning colon is required. Never use the beginning colon with command headers beginning with a star (*).
<Mnemonic>	This is a header subfunction. Some command headers have only one mnemonic. If a command header has multiple mnemonics, a colon (:) character always separates them from each other.
<Argument>	This is a quantity, quality, restriction, or limit associated with the header. Some commands have no arguments while others have multiple arguments. A <space> separates arguments from the header. A <comma> separates arguments from each other.
<Comma>	A single comma is used between arguments of multiple-argument commands. Optionally, there may be white space characters before and after the comma.
<Space>	A white space character is used between a command header and the related argument. Optionally, a white space may consist of multiple white space characters.

Commands

Commands cause the instrument to perform a specific function or change one of the settings. Commands have the structure:

```
[:]<Header>[<Space><Argument>[<Comma> <Argument>]...]
```

A command header consists of one or more mnemonics arranged in a hierarchical or tree structure. The first mnemonic is the base or root of the tree and each subsequent mnemonic is a level or branch off the previous one. Commands at a higher level in the tree may affect those at a lower level. The leading colon (:) always returns you to the base of the command tree.

Queries

Queries cause the instrument to return status or setting information. Queries have the structure:

- `[:]<Header>`
- `[:]<Header>[<Space><Argument> [<Comma><Argument>]...]`

You can specify a query command at any level within the command tree unless otherwise noted. These branch queries return information about all the mnemonics below the specified branch or level.

Clearing the Instrument

You can clear the Output Queue and reset the instrument to accept a new command or query by using the selected Device Clear (DCL) function.

Command Entry

The following rules apply when entering commands:

- You can enter commands in upper or lower case.
- You can precede any command with white space characters. White space characters include any combination of the ASCII control characters 00 through 09 and 0B through 20 hexadecimal (0 through 9 and 11 through 32 decimal).
- The instrument ignores commands consisting of any combination of white space characters and line feeds.

Abbreviating

You can abbreviate many instrument commands. The syntax of each command shows the minimum acceptable abbreviations in capitals. For example, you can enter the command `CALCulate:AVERage:COUNt` as `CALC:AVER:COUN` or `calc:aver:coun`.

Abbreviation rules may change over time as new instrument models are introduced. Thus, for the most robust code, use the full spelling.

Concatenating

You can concatenate any combination of set commands and queries using a semicolon (;). The instrument executes concatenated commands in the order received.

When concatenating commands and queries, you must follow these rules:

1. Separate completely different headers by a semicolon and by the beginning colon on all commands except the first one. For example, you can concatenate the commands `CALCULATE:AVERAGE:COUNT 20` and `INPUT:ATTENUATION 10` into the following single command:

   ```
   CALCULATE:AVERAGE:COUNT 20;INPUT:ATTENUATION 10
   ```

2. If concatenated commands have headers that differ by only the last mnemonic, you can abbreviate the second command and eliminate the beginning colon. For example, you can concatenate the commands `INPUT:ATTENUATION 10` and `INPUT:COUPLING DC` into a single command:

   ```
   INPUT:ATTENUATION 10; COUPLING DC
   ```

   The longer version works equally well:

   ```
   INPUT:ATTENUATION 10;INPUT:COUPLING DC
   ```

3. Never precede a star (*) command with a colon:

   ```
   INPUT:ATTENUATION 10;*OPC
   ```

   Any commands that follow are processed as if the star command was not there. For example, the commands `INPUT:ATTENUATION 10;*OPC;INPUT:COUPLING DC` set the input attenuation to 10X and set the input coupling to DC.

Message Terminator

This documentation uses `<EOM>` (End of Message) to represent a message terminator. An incoming end of message terminator can be one of the following:

- END message (EOI asserted concurrently with the last data byte). The last data byte may be an ASCII line feed (LF) character.
- Combining LF and EOI.

The instrument always terminates outgoing messages with LF and EOI.

Argument Types

Commands use arguments such as enumeration, numeric, quoted string and block. Each of these arguments are listed in detail below.

Enumeration

Enter these arguments as unquoted text words. Like key words, enumeration arguments follow the same convention where the portion indicated in uppercase is required and that in lowercase is optional.

For example: `INPUT:COUPLING DC`

Numeric

Many instrument commands require numeric arguments. The syntax shows the format that the instrument returns in response to a query. This is also the preferred format when sending the command to the instrument though any of the formats will be accepted. This documentation represents these arguments as described below.

Table 2-3: Numeric arguments

Symbol	Meaning
<Integer>	Signed integer value
<Decimal data>	Floating point value with or without an exponent

Most numeric arguments are automatically forced to a valid setting, by either rounding or truncating, when an invalid number is input, unless otherwise noted in the command description.

Quoted String

Some commands accept or return data in the form of a quoted string, which is simply a group of ASCII characters enclosed by a single quote (') or double quote ("). The following is an example of a quoted string: `"This is a quoted string"`.

A quoted string can include any character defined in the 7-bit ASCII character set. Follow these rules when you use quoted strings:

1. Use the same type of quote character to open and close the string. For example: `"this is a valid string"`.
2. You can mix quotation marks within a string if you follow the previous rule. For example: `"this is an 'acceptable' string"`.
3. You can include a quote character within a string by repeating the quote. For example: `"here is a "" mark"`.
4. Strings can have upper or lower case characters.
5. If you use a GPIB network, you cannot terminate a quoted string with the END message before the closing delimiter.
6. A carriage return or line feed embedded in a quoted string does not terminate the string. The return is treated as another character in the string.
7. The maximum length of a quoted string returned from a query is 1000 characters.

Here are some invalid strings:

- `"Invalid string argument'` (quotes are not of the same type)
- `"test<EOI>"` (termination character is embedded in the string)

Block

Several instrument commands use a block argument form, as defined in the following table:

Table 2-4: Block argument

Symbol	Meaning
<NZDig>	A nonzero digit character in the range of 1-9
<Dig>	A digit character, in the range of 0-9
<DChar>	A character with the hexadecimal equivalent of 00 through FF (0 through 255 decimal)
<Block>	A block of data bytes defined as: <Block>::= {#<NZDig><Dig>[<Dig>...][<DChar>...] |#0[<DChar>...]<terminator>}

<NZDig> specifies the number of <Dig> elements that follow. Taken together, the <NZDig> and <Dig> elements form a decimal integer that specifies how many <DChar> elements follow.

Macros

A macro is a single command, that represents one or several other commands, depending on your definition. You can define 25 macros of 40 characters in the instrument. One macro can address other macros, but you cannot call a macro from within itself (recursion). You can use variable parameters that modify the macro.

Use macros to do the following:

- Provide a shorthand for complex commands.
- Cut down on bus traffic.

Macro Names

You can use both commands and queries as macro labels. The label cannot be the same as common commands or queries. If a macro label is the same as an instrument command, the instrument will execute the macro when macros are enabled (*EMC 1), and it will execute the instrument command when macros are disabled (*EMC 0).

Data Types Within Macros

The commands to be performed by the macro can be sent both as block and string data.

String data is the easiest to use since you don’t have to count the number of characters in the macro. However, there are some things you must keep in mind:

Both double quote (“) and single quote (‘) can be used to identify the string data. If you use a controller language that uses double quotation marks to define strings

within the language (like BASIC) we recommend that you use block data instead, and use single quotes as string identifiers within the macro.

- *When using string data for the commands in a macro, remember to use a different type of string data identifiers for strings within the macro. If the macro should for instance set the input slope to positive and select the period function, you must type:*

":Inp:slope pos; :Func 'PER 1'"

or

':Inp:slope pos; :Func "PER 1"'

Define Macro Command

*DMC assigns a sequence of commands to a macro label. Later when you use the macro label as a command, the instrument will execute the sequence of commands.

Use the following syntax:

*DMC <macro-label>, <commands>

Simple macro example. *DMC 'FREQUENCY?',":FUNC 'FREQ 1';:INP:LEV:AUTO ON ;:ARM:START:LAY2:SOURCE BUS;:INIT:CONT ON;*TRG"

This example defines a macro "FREQUENCY?" that takes a single frequency measurement with an automatic trigger level setting and places the result in the output queue.

Macros with arguments. You can pass arguments (variable parameters) with the macro. Insert a dollar sign ($) followed by a single digit in the range 1 to 9 where you want to insert the parameter. See the example below.

When a macro with defined arguments is used, the first argument sent will replace any occurrence of $1 in the definition; the second argument will replace $2, and so on.

Example. *DMC 'AUTOFILT',":INP:LEV:AUTO $1;:INP:FILT $1;:INP2:LEV:AUTO $1;:INP2:FILT $1"

This example defines a macro called AUTOFILT that takes one Boolean argument such as ON or OFF for ($1).

AUTOFILT OFF

Turns off both the auto function and the analog lowpass filter on both input channels.

Deleting Macros

Use the *PMC (purge macro) command to delete all macros defined with the*DMC command. This removes all macro labels and sequences from the memory. To delete only one macro in the memory, use the:MEMory:DELete:MACRo command.

NOTE. *You cannot overwrite a macro; you must delete it before you can use the same name for a new macro.*

Enabling and Disabling Macros

***EMC Enable Macro Command.** When you want to execute an instrument command or query with the same name as a defined macro, you need to disable macro execution. Disabling macros does not delete stored macros; it just hides them from execution.

Disabling: *EMC0 disables all macros.

Enabling: *EMC1.

***EMC? Enable Macro Query.** Use this query to determine if macros are enabled.

Possible response: 1 = macros are enabled, 0 = macros are disabled

How to Execute a Macro

Macros are disabled after *RST, so to be sure, start by enabling macros with *EMC 1. Now macros can be executed by using the macro labels as commands.

Example:

*DMC ‘LIMITMON’,’ :CALC:STAT ON; :CALC:LIM:STAT ON; :CALC:LIM:LOW:DATA $1;STAT ON; :CALC:LIM:UPP:DATA $2;STAT ON’

*EMC 1

Now sending the command

LIMITMON 1E6,1.1E6

will switch on the limit monitoring to alarm between the limits 1MHz and 1.1MHz.

Retrieve a Macro

GMC? Get Macro Contents query. This query sets a response containing the definition of the macro you specified when sending the query.

Example using the above defined macro:

*GMC? ‘LIMITMON’

#292:CALC:STAT ON;:CALC:LIM:STAT ON; :CALC:LIM:LOW:DATA $1;STAT ON; :CALC:LIM:UPP:DATA $2;STAT ON’

LMC? Learn Macro query. This query returns a response containing the labels of all the macros stored in the Timer/instrument.

Example:

*LMC? might return "MYINPSETTING","LIMITMON"

Now there are two macros in memory, and they have the following labels: “MYINPSETTING” and “LIMITMON”.

Command Groups

Arming Subsystem

Table 2-5: Arming commands

Command	Description
ARM:COUNt	Sets or returns the upward exit of the wait-for-bus-arm state.
ARM:DELay	Sets or returns a delay between the pulse on the selected arming input and the time when the instrument starts measuring.
ARM:LAYer2	Overrides the waiting for bus arm, provided the source is set to bus.
ARM:LAYer2:SOURce	Sets or returns the mode for the wait-for-bus-arm function,
ARM:SLOPe	Sets or returns the slope for the start arming condition.
ARM:SOURce	Selects START arming input or switches off the start arming function.
ARM:STOP:SLOPe	Sets or returns the slope for the stop arming condition.
ARM:STOP:SOURce	Selects STOP arming input or switches off the STOP arming function.
ARM:STOP:TIMer	Sets or returns a delay between a pulse on the selected start arming input and the point of time when totalizing stops (FCA3000 Series only, Totalize mode only).

Calculate Subsystem

The calculate subsystem processes the measuring results. Here you can recalculate the result using mathematics, make statistics and set upper and lower limits for the measurement result. The instrument itself monitors the result and alerts you when the limits are exceeded.

Limit monitoring makes it is possible to get a service request when the measurement value falls below a lower limit or rises above an upper limit. Two status bits are defined to support limit monitoring. One is set when the results are greater than the UPPer limit, the other is set when the result is less than the LOWer limit. Enable the bits by using the standard *SRE command and :STAT:DREG0:ENAB. Using both these bits, it is possible to get a service request when a value passes out of a band (UPPer is set at the upper band border and LOWer at the lower border) OR when a measurement value enters a band (LOWer set at the upper band border and UPPer set at the lower border). Turning the limit monitoring calculations on or off will not influence the status register mask bits which determine whether or not to generate a service request when a limit is reached. Note that the calculate subsystem is automatically enabled when limit

monitoring is switched on. This means that other enabled calculate sub-blocks are indirectly switched on.

Table 2-6: Calculate commands

Command	Description
CALCulate:AVERage:ALL?	Returns mean value, standard deviation, min and max value from the current statistics sampling.
CALCulate:AVERage:COUNt	Sets or returns the number of samples to use in statistics sampling.
CALCulate:AVERage:COUNt:CURRent?	Returns the number of samples in the current statistics sampling.
CALCulate:AVERage:STATe	Switches the statistical function on and off or returns the state.
CALCulate:AVERage:TYPE	Sets or returns the statistical function to be performed.
CALCulate:DATA?	Fetches data calculated in the post processing block.
CALCulate:IMMediate	Causes the calculate subsystem to reprocess the statistical function on the sense data without reacquiring the data. Query returns this reprocessed data.
CALCulate:LIMit	Turns On/Off the limit-monitoring calculations.
CALCulate:LIMit:CLEar	Resets the instrument that reports its result using the CALCulate:LIMit:FCOunt? query.
CALCulate:LIMit:CLEar:AUTO	Activates or deactivates automatic reset by INIT of the instrument that reports its result using the CALCulate:LIMit:FCOunt? query.
CALCulate:LIMit:FAIL?	Returns the result of limit testing.
CALCulate:LIMit:FCOunt?	Returns the total number of times the set lower and upper limits have been passed since the instrument was last reset.
CALCulate:LIMit:FCOunt:LOWer?	Returns the number of times the set lower limit was passed since the instrument was last reset.
CALCulate:LIMit:FCOunt:UPPer?	Returns the number of times the set upper limit was passed since the instrument was last reset
CALCulate:LIMit:LOWer	Sets or returns the value of the lower limit.
CALCulate:LIMit:LOWer:STATe	Sets whether the measured value should be checked against the lower limit.
CALCulate:LIMit:PCOunt?	Returns the number of measurement results between the set lower and upper limits since the instrument was last reset
CALCulate:LIMit:UPPer	Sets or returns the value of the upper limit.
CALCulate:LIMit:UPPer:STATe	Sets whether the measured value should be checked against the upper limit.
CALCulate:MATH	Defines the mathematical expression used for mathematical operations.

Table 2-6: Calculate commands (cont.)

Command	Description
CALCulate:MATH:STATe	Switches on/off the mathematical function.
CALCulate:STATe	Switches on/off the complete post-processing block.
CALCulate:TOTalize:TYPE	Selects postprocessing for totalize.

Calibration Subsystem

This subsystem controls the calibration of the interpolators used to increase the resolution of the instrument.

Table 2-7: Calibration commands

Command	Description
CALibration:INTerpolator:AUTO	Sets or returns whether the instrument calibrates the time interpolators for every measurement.

Configure Function

The CONFigure command sets up the instrument to make the same measurements as the MEASure query, but without initiating the measurement and fetching the result. Use configure when you want to change any parameters before making the measurement.

CONFigure; READ?

The CONFigure command causes the instrument to choose an optimal setting for the specified measurement. CONFigure may cause any device setting to change. READ? starts the acquisition and returns the result.

This sequence operates in the same way as the MEASure command, but now it is possible to insert commands between CONFigure and READ? to fine-tune the setting of a particular function. For example, you can change the input impedance from 1M Ω to 50Ω.

Start with the command CONFigure:FREQ 2E6,1, where 2E6 is the expected value 1 is the required resolution (1Hz).

Then send INPut:IMPedance 50 to set input impedance to 50 Ω.

Then send READ? to start the measurement and returns the result.

CONFigure;INITiate;FETCh?

The READ? command can be divided into the INITiate command, which starts the measurement, and the FETCh? command, which requests the instrument to return the measuring results to the controller.

Start with the command CONFigure:FREQ 20E6,1, where 20E6 is the expected signal value 1 is the required resolution.

Then send INPut:IMPedance 1E6 to set input impedance to 1 MΩ.

Then send INITiate to start the measurement.

Then send FETCh? to fetch the result.

Table 2-8: Comparison of ways to take a measurement

Command	Advantage
MEASure?	Simple to use, few additional possibilities.
CONFigure READ?	Somewhat more difficult, but some extra possibilities.
CONFigure INITiate FETCh?	Most difficult to use, but many extra features.

Table 2-9: Configure commands

Command	Description
CONFigure:ARRay:<MeasuringFunction>	Sets up the instrument to perform the number of measurements you choose.
CONFigure:<MeasuringFunction>	Sets up the instrument to perform one measurement.
CONFigure:TOTalize[:CONTinuous]	Set up the instrument to take repeated measurements.

Display Subsystem

Commands in this subsystem control what data is to be present on the display and whether the display is on or off.

Table 2-10: Display command

Command	Description
DISPlay:ENABle	Turns On/Off the updating of the screen.

Fetch Function

Table 2-11: Fetch commands

Command	Description
FETCh:ARRay?	Fetches multiple measurements.
FETCh[:SCALar]?	Fetches a single measurement.

Format Subsystem

The Format subsystem converts the internal data representation to the data transferred over the external GPIB interface. Commands in this block control the data type to be sent over the external interface.

Time Stamp Readout Format

When FORMat:TINFormation is set to ON, the readout contains two values instead of one for FETCh:SCALar?, READ:SCALar? and MEASure:SCALar?.

The first is the measured value, expressed in the basic unit of the measurement function, and the second value is the timestamp value in seconds.

In FORMat ASCII mode, the result is given as a floating-point number, followed by a floating point timestamp value.

In FORMat REAL mode, the result is given as an eight-byte block containing the floating-point measured value, followed by an eight-byte block containing the floating-point timestamp value.

When doing readouts in array form, with FETCh:ARRay?, READ:ARRay? or MEASure:ARRay?, the response consists of alternating measurement values and timestamp values, formatted in a similar way as for scalar readout. All values are separated by commas.

Table 2-12: Format commands

Command	Description
FORMat	Sets or returns the format in which the result is sent on the bus.
FORMat:BORDer	Sets or returns the order in which response data bytes formatted.
FORMat:SMAX	Sets or queries the upper limit for FETCh:ARRay?
FORMat:TINFormation	Turns on/off the time stamping of measurements.

Hard Copy

Table 2-13: Hard copy command

Command	Description
HCOPy:SDUMp:DATA?	Returns block data containing a screen image in Windows BMP format.

Initiate Subsystem

Table 2-14: Initiate commands

Command	Description
INITiate	Initiate the trigger system to take a measurement.
INITiate:CONTinuous	Initiate the trigger system to take continuous measurements.

Input Subsystem

The Input subsystem performs all the signal conditioning of the input signal before it is converted into data by the Sense subsystem. The Input subsystem includes coupling, impedance, filtering, and so forth.

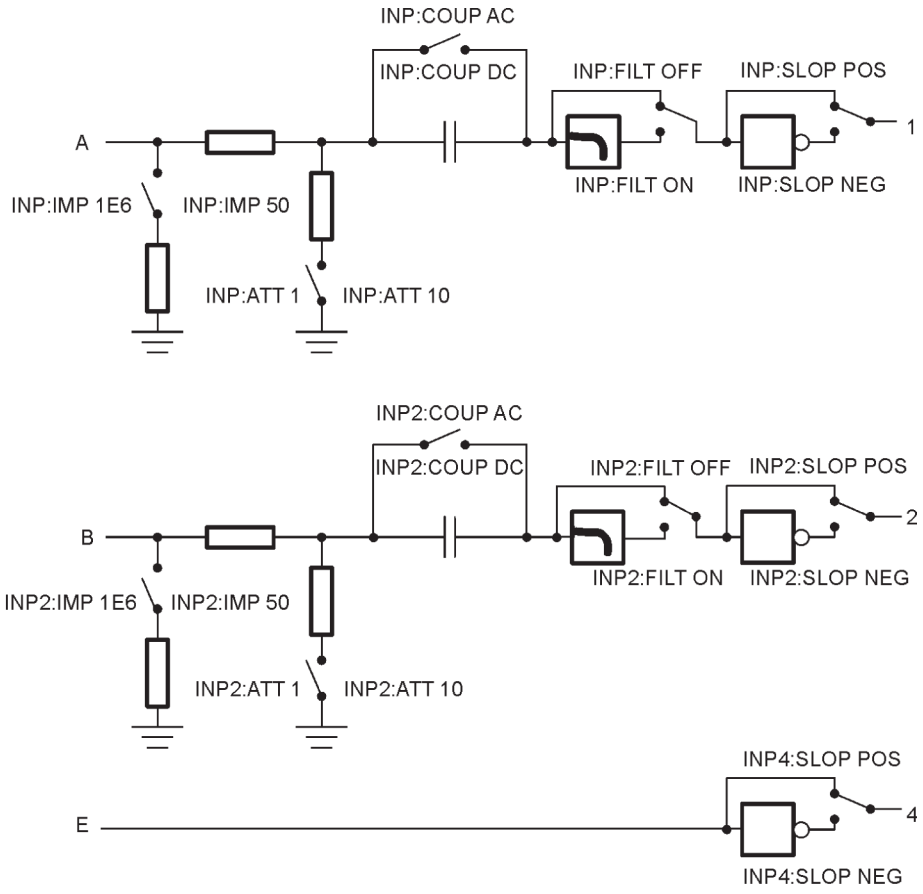


Table 2-15: Input commands

Command	Description
AUTO	Performs the same task as the front panel button AUTO SET.
INPut{[1]|2}:ATTenuation	Sets or returns the input attenuation.
INPut{[1]|2}:COUPling	Sets or returns the input coupling.
INPut{[1]|2}:FILTer	Switches on or off the analog low pass filter.
INPut{[1]|2}:FILTer:DIGital	Switches on or off the digital low pass filter.
INPut{[1]|2}:FILTer:DIGital:FREQuency	Sets or returns the digital filter cutoff frequency.
INPut{[1]|2}:IMPedance	Sets or returns the input impedance.
INPut{[1]|2}:LEVel	Sets or returns the input threshold level.
INPut{[1]|2}:LEVel:AUTO	Switches autotrigger level on or off.
INPut{[1]|2}:LEVel:RELative	Sets or returns specific trigger levels for different measurements.
INPut{[1]|2}:SLOPe	Sets or returns the slope for certain measurements.

Measurement Subsystem

The Measure function group has a different level of compatibility and flexibility than other commands. The parameters used with commands from the Measure group describe the signal you are going to measure. This means that the Measure functions give compatibility between instruments, since you don't need to know anything about the instrument you are using.

MEASure?

This is the most simple query to use, but it does not offer much flexibility. The MEASure? query lets the instrument configure itself for an optimal measurement, starts the data acquisition, and returns the result.

MEASure:FREQ? example. This will execute a frequency measurement and the result is sent to the controller. The instrument will select a setting for this purpose by itself, and will carry out the required measurement as "well" as possible; moreover, it will automatically start the measurement and send the result to the controller. You may add parameters to give more details about the signal you are going to measure, for example:

Send the query MEASure:FREQ? 20 MHz,1, where: 20 MHz is the expected value, which can, of course, also be sent as 20E6, and 1 is the required resolution. (1Hz)

Also the channel numbers can be specified, for example: MEASure:FREQ? (@3) or MEASure:FREQ? 20E6, 1,(@1)

Table 2-16: Comparison of ways to take a measurement

Command	Advantage
MEASure?	Simple to use, few additional possibilities.
CONFigure READ?	Somewhat more difficult, but some extra possibilities.
CONFigure INITiate FETCh?	Most difficult to use, but many extra features.

Table 2-17: Measurement commands

Command	Description
ABORt	Terminates a measurement.
MEASure:ARRay:FREQuency:BTBack?	Takes a series of back-to-back frequency measurements.
MEASure:ARRay:<MeasuringFunction>?	Sets up a series of measurements with the results returned in a single string.
MEASure:ARRay:PERiod:BTBack?	Takes a series of back-to-back period measurements.
MEASure:ARRay:STSTamp?	Takes a series of back-to-back time-stamp measurements.
MEASure:ARRay:TIError?	Takes a series of back-to-back relative frequency measurements.
MEASure:ARRay:TSTAmp?	Takes a series of back-to-back time-stamp measurements taken at all positive and negative trigger level crossings.
MEASure{:FALL:TIME|:FTIM}?	Takes a fall time measurement.
MEASure:FREQuency?	Takes a frequency measurement.
MEASure:FREQuency:BURSt?	Takes a measurement of the carrier frequency of a burst.
MEASure:FREQuency:POWer[:AC]?	Takes a power measurement.
MEASure:FREQuency:PRF?	Takes a pulse-repetition frequency measurement.
MEASure:FREQuency:RATio?	Takes a frequency ratio measurement.
MEASure:<MeasuringFunction>?	Sets up a single measurement with the result returned in a string.
MEASure:MEMory?	Recalls an instrument setting stored in memory and returns a measurement value.
MEASure:MEMory<N>?	Recalls an instrument setting stored in memory and returns a measurement value.
MEASure:NDUTycycle?	Takes a negative duty cycle measurement.
MEASure:NWIDth?	Takes a negative pulse width measurement.
MEASure{:PDUTycycle|:DCYCle}?	Takes a positive duty cycle measurement.
MEASure:PERiod?	Takes a period measurement.
MEASure:PERiod:AVERage?	Returns an average of multiple period measurements.

Table 2-17: Measurement commands (cont.)

Command	Description
MEASure:PHASe?	Takes a phase measurement.
MEASure:PWIDth?	Takes a positive pulse width measurement.
MEASure{:RISE:TIME|:RTIM}?	Takes a rise time measurement.
MEASure:TINTerval?	Takes a time interval measurement.
MEASure[:VOLT]:MAXimum?	Takes a positive peak voltage measurement.
MEASure[:VOLT]:MINimum?	Takes a negative peak voltage measurement.
MEASure[:VOLT]:NCYCles?	Measures the number of cycles in a burst.
MEASure[:VOLT]:NSLEwrate?	Takes a negative slew rate measurement.
MEASure[:VOLT]:PSLEwrate?	Takes a positive slew rate measurement.
MEASure[:VOLT]:PTPeak?	Takes a peak-to-peak voltage measurement.
MEASure[:VOLT]:RATio?	Takes a peak-to-peak voltage ratio measurement.
TOTalize:GATE	Opens and closes the gate for continuous measurements.

Memory Subsystem

The Memory subsystem holds macro and instrument state data inside the instrument.

Table 2-18: Memory commands

Command	Description
MEMory:DATA:RECord:COUNt?	Returns the number of samples in a given memory location.
MEMory:DATA:RECord:DELete	Erases a given memory location.
MEMory:DATA:RECord:FETCh?	Returns one sample from a given memory location.
MEMory:DATA:RECord:FETCh:ARRay?	Returns multiple samples from a given memory location.
MEMory:DATA:RECord:FETCh:STARt	Sets the pointer to the first sample in a given memory location.
MEMory:DATA:RECord:NAME?	Returns the name of a given memory location.
MEMory:DATA:RECord:SAVE	Saves samples in a given memory location.
MEMory:DATA:RECord:SETTings?	Returns the instrument settings used when the specified <Dataset> was saved.
MEMory:DELete:MACRo	Deletes an individual macro.
MEMory:FREE:MACRo?	Returns the bytes used and available for macros.
MEMory:NSTates?	Returns (one greater than) the number of available memory locations for instrument settings.

Output Subsystem

Table 2-19: Output commands

Command	Description
OUTPut:POLarity	Sets or returns the polarity of the pulse output.
OUTPut:TYPE	Sets or returns the function of the pulse output.
SOURce:PULSe:PERiod	Sets the period for the pulse output.
SOURce:PULSe:WIDTh	Sets the pulse width for the pulse output.

Read Function

Table 2-20: Read commands

Command	Description
READ?	Performs a new measurement and reads out a measuring result.
READ:ARRay?	Performs multiple measurements and reads out the measuring results.

Sense Command Subsystem

The Sense subsystem converts the signals into internal data that can be processed by the Calculate subsystem. The SENSe commands control various characteristics of the measurement and acquisition process. These include gate time, measurement function, resolution, and so on.

Depending on application, you can select different input channels and input characteristics.

Switchbox. In automatic test systems, it is difficult to swap BNC cables when you need to measure on several measuring points. The FCA3000 series lets you switch between input A and B to take measurements directly without the need for external switching devices.

Prescaling. For all measuring functions except *time interval* , *rise/fall time* , *phase* and *time stamping* , the maximum input A or B frequency is 300 MHz.

For the measuring functions explicitly mentioned above, the instrument has a max repetition rate of 160MHz.

For the measuring functions *Frequency* and *Period Average* , the signal to Input A or Input B is prescaled by a factor of 2. For *Frequency in Burst* , *PRF* and *Number of Cycles in Burst* , the signal is prescaled by a factor of 2 if the command :SENSe:FREQuency:BURSt:PREScaleris set to ON. This is also the default condition.

Table 2-21: Sense commands

Command	Description
ACQuisition:APERture	Sets or returns the gate time for a measurement.
ACQuisition:HOFF	Switches the holdoff function on or off.
ACQuisition:HOFF:TIME	Sets or returns the holfoff time value.
FREQuency:BURSt:APERture	Sets the time length within a burst during which the burst frequency is measured.
FREQuency:BURSt:PRESCaler[:STATe]	Switches the frequency burst prescaler on and off.
FREQuency:BURSt:STARt:DELay	Sets or returns the time length between the burst start and the actual start of the burst measuring time.
FREQuency:BURSt:SYNC:PERiod	Sets the synchronization delay time used in burst measurements.
FREQuency:POWer:UNIT	Sets or returns the measurement unit for power measurements.
FREQuency:RANGe:LOWer	Sets a lower-limit frequency for certain voltage and autotrigger function.
FREQuency:REGRession	Switches the linear regression function on and off.
FUNCtion	Sets the measuring function to be performed and input channel.
HF:ACQuisition[:STATe]	Switches the automatic acquisition system on or off.
HF:FREQuency:CENTer	Sets the center frequency value for the RF input.
ROSCillator:SOURce	Selects the source for the time base.
TIError:FREQuency	Sets a reference frequency for relative frequency measurements.
TIError:FREQuency:AUTO	Sets the instrument to use a relative frequency that is listed for automatic recognition.
TINTerval:AUTO	Sets the instrument to automatically detect the start channel in a time interval measurement.

Status Subsystem

This subsystem can be used to get information about what is happening in the instrument at the moment.

Table 2-22: Status commands

Command	Description
STATus:DREGister0?	Returns the contents of the Device Event Register.
STATus:DREGister0:ENABle	Sets the enable bit of the Device Register 0.
STATus:OPERation?	Returns the contents of the operation event status register.

Table 2-22: Status commands (cont.)

Command	Description
STATus:OPERation:CONDition?	Returns the contents of the operation status condition register.
STATus:OPERation:ENABle	Sets the enable bits of the operation status enable register.
STATus:PRESet	Sets or clears all other enable registers other than the IEEE-488.2 enable registers.
STATus:QUEStionable?	Returns the contents of the status questionable event register.
STATus:QUEStionable:CONDition?	Returns the contents of the status questionable condition register.
STATus:QUEStionable:ENABle	Sets the enable bits of the status questionable enable register.

System Subsystem

This subsystem controls some system parameters like timeout.

Table 2-23: System commands

Command	Description
SYSTem:COMMunicate:GPIB:ADDRess	Sets or returns the GPIB address.
SYSTem:ERRor?	Queries for an ASCII text description of an error that occurred.
SYSTem:LANGuage	Selects one of two command sets.
SYSTem:PRESet	Recalls the default settings for the instrument.
SYSTem:SET	Returns the complete current state of the instrument.
SYSTem:TALKonly	Sets the instrument to talk-only mode.
SYSTem:TEMPerature?	Returns the temperature in degrees C at the fan control sensor inside the instrument housing.
SYSTem:TOUT	Switches the time-out on or off.
SYSTem:TOUT:AUTO	Sets an automatic time out after the first start trigger.
SYSTem:TOUT:TIME	Sets or returns the time-out time.
SYSTem:UNPRotect	Unprotects the user data set or read by the *PUD command.

Test Subsystem

This subsystem tests the hardware and software of the instrument and reports errors.

Table 2-24: Test command

Command	Description
TEST:SELect	Selects which internal self-tests shall be used when self-test is requested by the *TST? command.

Trigger Subsystem

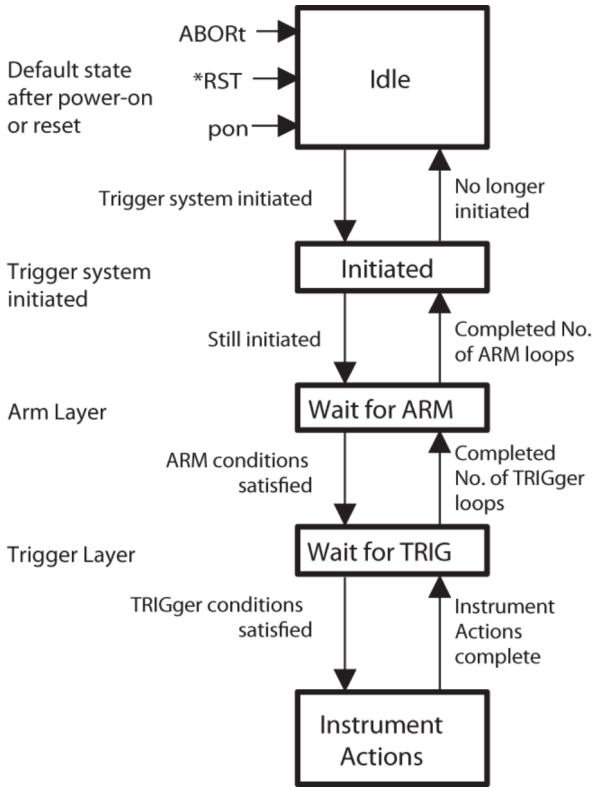
The Trigger subsystem enables synchronization of instrument actions with specified internal or external events.

Instrument Action

Some examples of events to synchronize with are as follows:

- Measurement
- Bus trigger
- External signal level or pulse
- Ten occurrences of a pulse on the external trigger input
- Other instrument ready
- Signal switching
- Input signal present
- One second after input signal is present
- Sourcing output signal
- Switching system ready

The ARM-TRIG Trigger Configuration gives a typical trigger configuration, the ARM-TRIG model. The configuration contains two event-detection layers: the ‘Wait for ARM’ and ‘Wait for TRIG’ states.



This trigger configuration is sufficient for most instruments. More complex instruments, such as the FCA3000 and MCA3000 Series, have more ARM layers.

The 'Wait for TRIG' event-detection layer is always the last to be crossed before instrument actions can take place.

Structure of the IDLE and INITIATED States

When you turn on the power or send *RST or ABORT to the instrument, it sets the trigger system in the IDLE state.

The trigger system will exit from the IDLE state when the instrument receives an INITiate:IMMediate. The instrument will pass directly through the INITIATED state downward to the next event-detection layers (if the instrument contains any more layers).

The trigger system will return to the INITIATED state when all events required by the detection layers have occurred and the instrument has made the intended measurement. When you program the trigger system to INITiate:CONTinuous ON, the instrument will directly exit the INITIATED state moving downward and will repeat the whole flow described above. When INITiate:CONTinuous is OFF, the trigger system will return to the IDLE state.

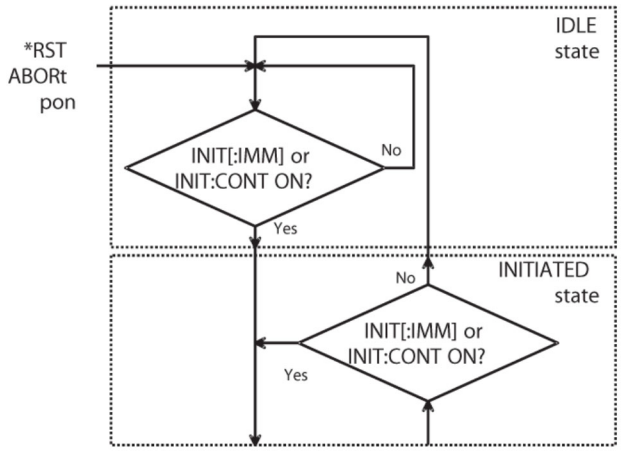


Figure 2-1: Flow diagram of IDLE and INITIATED layers.

Structure of an event-detection layer. The general structure of all event-detection layers is identical. (See Figure 2-1.)

In each layer there are several programmable conditions, which must be satisfied to pass by the layer in a downward direction:

Forward traversing an event-detection layer. After initiating the loop instruments, the instrument waits for the event to be detected. You can select the event to be detected by using the <layer>:SOURce command. For example: ARM:LAYer2:SOURce BUS

You can specify a more precise characteristic of the event to occur. For example: ARM:LAYer:DELay 0.1

You may program a certain delay between the occurrence of the event and entering into the next layer (or starting the device actions when in the TRIGger layer). This delay can be programmed by using the <layer>:DELay command.

Backward traversing an event-detection layer. The number of times a layer event has to initiate a device action can be programmed by using the <layer>:COUNt command. For example: :TRIGger:COUNt 3 causes the instrument to measure three times, each measurement being triggered by the specified events.

Triggering

***TRG trigger command.** The trigger command has the same function as the Group Execute Trigger command GET, defined by IEEE488.1.

*When to use * TRG and GET*

The *TRGand the GET commands have the same effect on the instrument. If the instrument is in idle (not parsing or executing any commands), GET will execute much faster than *TRG since the instrument must always parse *TRG.

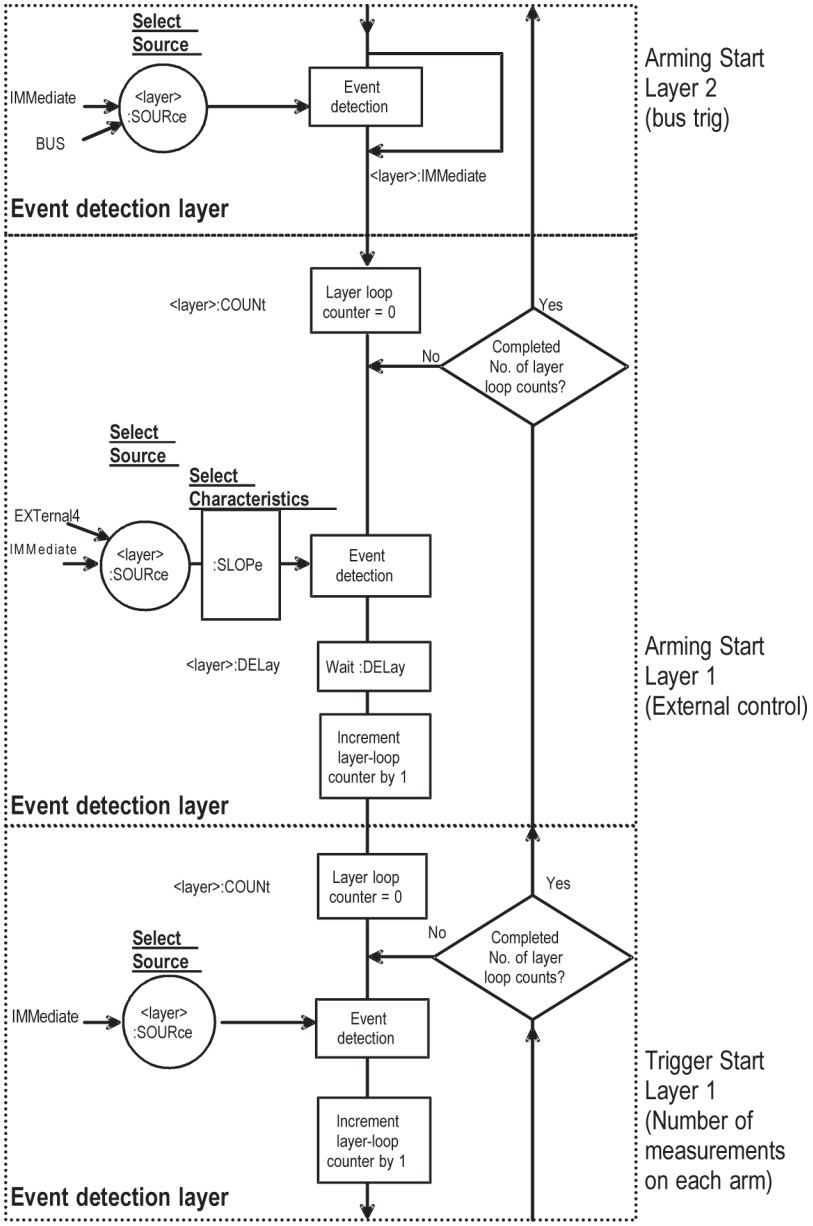


Figure 2-2: Structure of event detection layers.

Table 2-25: Trigger commands

Command	Description
TRIGger:COUNt	Sets or returns how many measurements the instrument should make for each arm condition.
TRIGger:SOURce	Enables or disables the sample rate control.
TRIGger:TIMer	Sets the sample rate fur use with the statistics functions.

Common Commands

Table 2-26: IEEE common commands

Command	Description
*CLS	Clears the status data structures by clearing all event registers and the error queue.
*DDT	Sets or queries the command that the device will execute on receiving the GET interface message or the *TRG common command.
*DMC	Defines a new macro.
*EMC	Enables macros.
*ESE	Sets the enable bits of the standard event enable register.
*ESR?	Returns the contents of the standard event status register.
*GMC?	Returns the definition of a existing macro.
*IDN?	Reads out the manufacturer, model, serial number, and firmware level in an ASCII response data element.
*LMC?	Returns the labels of all defined macros.
*LRN?	Returns a message that can be sent to the instrument to return it to the state it was in when the *LRN? query was made.
*OPC	Generates the operation complete message in the Standard Event Status Register.
*OPT?	Return all detectable features present in the instrument.
*PMC	Deletes all macro definitions.
*PSC	Enables/disables automatic power-on clearing.
*PUD	Sets or returns protected user data.
*RCL	Recalls one of the previously stored complete instrument settings from the internal nonvolatile memory of the instrument.
*RMC	Deletes an individual macro.
*RST	Resets the instrument.
*SAV	Saves the settings of the instrument in an internal nonvolatile memory.
*SRE	Sets or returns the service request enable register bits.
*STB?	Returns the value of the Status Byte.
*TRG	Starts the measurement and places the result in the output queue.

Table 2-26: IEEE common commands (cont.)

Command	Description
*TST?	Starts an internal self-test and generates a response indicating whether or not the instrument completed the self-test without any detected errors.
*WAI	Prevents the instrument from executing any further commands or queries until execution of all previous commands or queries is completed.

Command Descriptions

ABORt (No Query Form)

The ABORt command terminates a measurement. The trigger subsystem state is set to idle-state. The command does not invalidate already finished results when breaking an array measurement. This means that you can fetch a partial result after an abort.

Aborts all previous measurements if *WAI is not used.

Group Measurement

Syntax

```
ABORt
```

ACQuisition:APERture

Sets the gate time for one measurement.

Group Sense

Syntax

```
ACQuisition:APERture {<Decimal value > | MIN | MAX }
ACQuisition:APERture?
```

Arguments <DECIMAL VALUE> is 20 ns to 1000s. MIN sets 20 ns and MAX sets 1000 s.

Returns <Decimal value >

200 ms after SYST:PRES

10 ms after *RST

ACQuisition:HOFF

Sets the Hold Off function On or Off.

Group Sense

Syntax

```
ACQuisition:HOFF <boolean>
ACQuisition:HOFF?
```

Arguments <BOOLEAN> = 1 | ON | 0 | OFF

Returns 1 | 0

ACQuisition:HOFF:TIME

Sets the Hold Off time value.

Group Sense

Syntax
```
ACQuisition:HOFF:TIME {<Decimal value> | MIN | MAX}
ACQuisition:HOFF:TIME?
```

Arguments <DECIMAL DATA>= a number between 20E-9 and 2.0

Returns <Decimal value>

ARM:COUNt

This count variable controls the upward exit of the wait-for-bus-arm state. The instrument loops the trigger subsystem downwards COUNt number of times before it exits to the idle state.

This means that a COUNt number of measurements can be done for each Bus arming or INITiate.

NOTE. *The actual number of measurements made on each INIT is equal to (ARM:COUNT)*(TRIG:START:COUNT).*

Group Arming

Syntax
```
ARM:COUNt <Numeric value>| MIN | MAX | INFinity
ARM:COUNt?
```

Arguments <Numeric value> is an integer between 1 and 2,147,483,647 (2^{31} -1). The integer 1 switches the function OFF.

MIN sets 1.

MAX sets 2147483647.

INFinity makes the arm loop continue indefinitely, or until other device-dependent parameters set limits.

ARM:DELay

This command sets a delay between the pulse on the selected arming input and the time when the instrument starts measuring.

Range: 20ns to 2s, with 10 ns resolution.

Group Arming

Syntax

```
ARM:DELay <Numeric value> | MIN | MAX
ARM:DELay?
```

Arguments <Numeric value> is a number between $20*10^{-9}$ and 2.

MIN sets 0 which switches the delay OFF.

MAX sets 2 s.

Returns <Numeric value>

Examples

```
ARM:DELAY 0.1
```

ARM:LAYer2 (No Query Form)

This command overrides the waiting for bus arm, provided the source is set to bus. When this command is issued, the instrument will immediately exit the wait-for-bus-arm state.

The instrument generates an error if it receives this command when the trigger subsystem is not in the wait-for-bus-arm state.

If the Arming source is set to Immediate, this command is ignored.

Group Arming

Syntax

```
ARM:LAYer2
```

Examples

```
ARM:LAYER2
```

ARM:LAYer2:SOURce

Switches between Bus and Immediate mode for the wait-for-bus-arm function, (layer 2). GETand *TRG triggers the instrument if Bus is selected as source.

If the instrument receives GET/*TRG when not in wait-for-bus-arm state, it ignores the trigger and generates an error.

It also generates an error if it receives GET/*TRG and bus arming is switched off (set to IMMediate).

Group Arming

Syntax
```
ARM:LAYer2:SOURce {BUS | IMMediate}
ARM:LAYer2:SOURce?
```

Arguments

BUS

IMMediate

Examples
```
ARM:LAYER2:SOURCE BUS
```

ARM:SLOPe

Sets the slope for the start arming condition.

Group Arming

Syntax
```
ARM:SLOPe {POSitive | NEGative}
ARM:SLOPe?
```

Arguments

POS

NEG

Examples
```
ARM:SLOPE NEG
```

ARM:SOURce

Selects START arming input or switches off the start arming function. When switched off the DELay is inactive.

Group Arming

Syntax
```
ARM:SOURce {EXTernal1 | EXTernal2 | EXTernal4 | IMMediate}
ARM:SOURce?
```

Arguments

`EXTernal1` is input A

`EXTernal2` is input B

`EXTernal4` is input E

`IMMediate` is Start arming OFF

NOTE. *For the Totalize function in the FCA3100 Series, IMM means manual start-stop using the commands TOT:GATEON|OFF.*

Returns EXT1 | EXT2 | EXT4 | IMM

Examples
```
ARM:SOURCE EXT4
```

ARM:STOP:SLOPe

Sets the slope for the stop arming condition.

Group Arming

Syntax
```
ARM:STOP:SLOPe {POSitive | NEGative}
ARM:STOP:SLOPe?
```

Returns POS|NEG

Examples
```
ARM:STOP:SLOPE NEG
```

ARM:STOP:SOURce

Selects STOP arming input or switches off the STOP arming function. The FCA3100 Series has also a programmable timer that is accessible in Totalize mode.

Group Arming

Syntax
```
ARM:STOP:SOURce {EXTernal1 | EXTernal2 | EXTernal4 | TIMer
| IMMediate}
ARM:STOP:SOURce?
```

Arguments

`EXTernal1` is input A

`EXTernal2` is input B

`EXTernal4` is input E

`TIMer` is timed STOP in Totalize measurements (FCA3100 Series only). The time is set with the command ARM:STOP:TIMer.

`IMMediate` sets Stop arming OFF

Returns EXT1 | EXT2 | EXT4 | TIM | IMM

Examples
```
ARM:STOP:SOURCE EXT4
```

ARM:STOP:TIMer

This command sets a delay between a pulse on the selected start arming input (when totalizing starts) and the point of time when totalizing stops.

Range: 20 ns to 2 s, with 10 ns resolution.

Group Arming

Syntax
```
ARM:STOP:TIMer <Numeric value> | MIN | MAX
ARM:STOP:TIMer?
```

Arguments

`<Numeric value>` is a number between $20 * 10^{-9}$ and 2 s.

`MIN` sets $20 * 10^{-9}$ s.

`MAX` sets 2 s.

Returns <Numeric value>

Examples
```
ARM:STOP:TIMER 0.1
```

AUTO (No Query Form)

Performs the same task as the front panel button AUTO SET.

Group Input

Syntax
```
AUTO ONCE | PRESet
```

Arguments ONCE corresponds to pressing AUTO SET once.

PRESet corresponds to double-clicking AUTO SET.

CALCulate:AVERage:ALL? (Query Only)

Returns mean value, standard deviation, min and max value from the current statistics sampling.

Group Calculate

Syntax
```
CALCulate:AVERage:ALL?
```

Returns <mean value>, <standard deviation>, <min value>, <max value>

CALCulate:AVERage:COUNt

Sets the number of samples to use in statistics sampling.

Group Calculate

Syntax
```
CALCulate:AVERage:COUNt <number of samples>
CALCulate:AVERage:COUNt?
```

Arguments `<number of samples>` is an integer in the range 2 to $2*10^9$.

Returns < number of samples>

CALCulate:AVERage:COUNt:CURRent? (Query Only)

Returns the number of samples in the current statistics sampling.

Group Calculate

Syntax
```
CALCulate:AVERage:COUNt:CURRent?
```

Returns <number of samples>

CALCulate:AVERage:STATe

This command switches the statistical function on and off.

The CALCulate subsystem is automatically enabled when the statistical functions are switched on. This means that other enabled calculate sub-blocks are indirectly switched on. The statistics must be enabled before the measurements are performed. When the statistical function is enabled, the instrument will keep the trigger subsystem initiated until the CALCulate:AVERage:COUNt variable is reached. This is done without any change in the trigger subsystem settings. Consider that the trigger subsystem is programmed to perform 1000 measurements when initiated. In such a case, the instrument must make 10000 measurements if the statistical function requires 9500 measurements because the number of measurements must be a multiple of the number of measurements programmed in trigger subsystem (1000 in this example).

Group Calculate

Syntax
```
CALCulate:AVERage:STATe < Boolean >
CALCulate:AVERage:STATe?
```

Arguments
```
<BOOLEAN> = ( 1| ON | 0 | OFF )
```

Returns 1|0

NOTE. *Statistics with array readouts cannot be combined. To store and fetch individual values in a block measurement, use the default command* `CALCulate:AVERage:STATe` *is OFF.*

CALCulate:AVERage:TYPE

Selects the statistical function to be performed.

NOTE. *Use CALCulate:DATA? to read the result of statistical operations. READ? and FETCh[:SCALar]? will only send the results that the statistical operation is based on.*

Group Calculate

Syntax

```
CALCulate:AVERage:TYPE { MAX | MIN | MEAN | SDEViation |
ADEViation}
CALCulate:AVERage:TYPE?
```

Arguments **MAX** returns the maximum value of all samples taken under CALC:AVERcontrol.

MIN returns the minimum value of all samples taken under CALC:AVERcontrol.

MEAN returns the mean value of the samples taken:

$$\overline{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

SDEV returns the standard deviation of the samples taken:

$$s = \sqrt{\frac{\sum_{i=1}^{N} (x_i - \overline{x})^2}{N - 1}}$$

ADEV returns the Allan deviation of the samples taken:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N-1} (x_{i+1} - x_i)^2}{2(N - 1)}}$$

Returns MAX | MIN | MEAN | SDEV | ADEV

CALCulate:DATA? (Query Only)

Returns data calculated in the post processing block.

NOTE. *Use this command to return the calculated result without making a new measurement.*

Group Calculate

Syntax `CALCulate:DATA?`

Returns <Decimal data>

Examples `CALCULATE:DATA?` might return `CALC:MATH:STAT ON;:CALC:MATH(((1*X)-10.7E6)/1) ;:INIT; *OPC`

Wait for operation complete

```
CALCULATE:DATA?
<Measurement result> - 10.7E6
```

CALCulate:IMMediate

This event causes the calculate subsystem to reprocess the statistical function on the sense data without reacquiring the data. Query returns this reprocessed data.

Group Calculate

Syntax

```
CALCulate:IMMediate
CALCulate:IMMediate?
```

Returns <Decimal data>

Where: <Decimal data> is the recalculated data.

Examples `CALCULATE:IMMEDIATE` CALC:AVER:STAT ON;TYPES DEV;:INIT;*OPC

Wait for operation complete

```
CALC:DATA?
<VALUE OF STANDARD DEVIATION>
CALC:AVER:TYPE MEAN
CALC:IMM?
<MEAN VALUE>
```

CALCulate:LIMit

Turns On/Off the limit-monitoring calculations. Limit monitoring generates a service request when the measurement value falls below a lower limit, or rises above an upper limit.

Two status bits are defined to support limit-monitoring. One is set when the results are greater than the UPPer limit, the other is set when the result is less than the LOWer limit. The bits are enabled using the standard * SREcommand and:STAT:DREG0:ENAB. Using both these bits, it is possible to get a service request when a value passes out of a band (UPPer is set at the upper band border and LOWer at the lower border) OR when a measurement value enters a band (LOWer set at the upper band border and UPPer set at the lower border). Turning the limit-monitoring calculations On/Off will not influence the status register mask bits, which determine whether or not a service request is generated when a limit is reached.

NOTE. *The calculate subsystem is automatically enabled when limit-monitoring is switched on. This means that other enabled calculate sub-blocks are indirectly switched on.*

Group Calculate

Syntax

```
CALCulate:LIMit <Boolean>
CALCulate:LIMit?
```

Related Commands Example 1 in Chapter 4 deals with limit-monitoring.

Arguments <BOOLEAN> = (1 | ON | 0 | OFF)

Returns 1 | 0

CALCulate:LIMit:CLEar (No Query Form)

The command resets the instrument that reports its result using the CALCulate:LIMit:FCOunt? query.

Group Calculate

Syntax

```
CALCulate:LIMit:CLEar
```

CALCulate:LIMit:CLEar:AUTO

The command activates (ON) or deactivates (OFF) automatic reset by INIT of the instrument that reports its result using the CALCulate:LIMit:FCOunt? query.

Group Calculate

Syntax
```
CALCulate:LIMit:CLEar:AUTO <Boolean>
CALCulate:LIMit:CLEar:AUTO?
```

Arguments <Boolean> = (1 | ON | 0 | OFF)

CALCulate:LIMit:FAIL? (Query Only)

Returns a 1 if the limit testing has failed (the measurement result has passed the limit), and a 0 if the limit testing has passed.

The following events reset the fail flag:

- Power-on
- * RST
- A:CALC:LIM:STATOFF:CALC:LIM:STATONtransition
- Reading a 1 with this command

Group Calculate

Syntax
```
CALCulate:LIMit:FAIL?
```

Returns 1| 0

Examples
```
SENS:FUNC 'FREQ';:CALC:LIM:STATON;:CALC:LIM:UPPER
1E3;READ?;*WAI;:CALC:LIM:FAIL?
```
might return 1 if frequency is above 1 kHz, and 0 otherwise.

CALCulate:LIMit:FCOunt? (Query Only)

The command returns the total number of times the set lower and upper limits have been passed since the instrument was last reset by CALCulate:LIMit:CLEar or automatically by INITiate if CALCulate:LIMit:CLEar:AUTO is set to ON.

In other words, the returned value is the sum of the values returned by CALCulate:LIMit:FCOunt:LOWer? and CALCulate:LIMit:FCOunt:UPPer?.

Group Calculate

Syntax `CALCulate:LIMit:FCOunt?`

Returns <Number of counts>

CALCulate:LIMit:FCOunt:LOWer? (Query Only)

The command returns the number of times the set lower limit was passed since the instrument was last reset by CALCulate:LIMit:CLEar or automatically by INITiate if CALCulate:LIMit:CLEar:AUTO is set to ON.

Group Calculate

Syntax `CALCulate:LIMit:FCOunt:LOWer?`

Returns <Number of counts>

CALCulate:LIMit:FCOunt:UPPer? (Query Only)

The command returns the number of times the set upper limit was passed since the instrument was last reset by CALCulate:LIMit:CLEar or automatically by INITiate if CALCulate:LIMit:CLEar:AUTO is set to ON.

Group Calculate

Syntax `CALCulate:LIMit:FCOunt:UPPer?`

Returns <Number of counts>

CALCulate:LIMit:LOWer

Sets the value of the lower limit, that is, the lowest measurement result allowed before the instrument generates a 1 that can be read with CALCulate:LIMit:FAIL?, or by reading the corresponding status byte.

Group Calculate

Syntax
```
CALCulate:LIMit:LOWer {<Decimal data>| MAX | MIN }
CALCulate:LIMit:LOWer?
```

Arguments PARAMETER RANGE: $-9.9*10^{+37}$ to $+9.9*10^{+37}$.

Returns <Decimal data>

CALCulate:LIMit:LOWer:STATe

Selects if the measured value should be checked against the lower limit.

Group Calculate

Syntax
```
CALCulate:LIMit:LOWer:STATe <Boolean>
CALCulate:LIMit:LOWer:STATe?
```

Arguments <BOOLEAN> = (1/ON | 0/OFF)

Returns 1| 0

CALCulate:LIMit:PCOunt?

The command returns the number of measurement results between the set lower and upper limits since the instrument was last reset by CALCulate:LIMit:CLEar or automatically by INITiate if CALCulate:LIMit:CLEar:AUTO is set to ON.

Group Calculate

Syntax
```
CALCulate:LIMit:PCOunt?
```

Returns < number of counts>

CALCulate:LIMit:UPPer

Sets the value of the upper limit (the highest measurement result allowed) before the instrument generates a 1 that can be read with CALCulate:LIMit:FAIL?, or by reading the corresponding status byte.

Group Calculate

Syntax
```
CALCulate:LIMit:UPPer {<Decimal data>| MAX | MIN }
CALCulate:LIMit:UPPer?
```

Arguments RANGE: $-9.9*10^{+37}$ to $+9.9*10^{+37}$

Returns <Decimal data>

CALCulate:LIMit:UPPer:STATe

Selects if the measured value should be checked against the upper limit.

Group Calculate

Syntax
```
CALCulate:LIMit:UPPer:STATe <Boolean>
CALCulate:LIMit:UPPer:STATe?
```

Arguments <BOOLEAN> = (1/ON | 0/OFF)

Returns 1| 0

CALCulate:MATH

Defines the mathematical expression used for mathematical operations.

NOTE. *The data type <expression data> must be enclosed within parentheses.*

Group Calculate

Syntax

```
CALCulate:MATH (<expression>)
CALCulate:MATH?
```

Arguments

<EXPRESSION> is one of the following five mathematical expressions:

- ((K * X) + L)
- (((K * X) + L) / M)
- (((K / X) + L) / M)
- ((X / M) - 1)

NOTE. *No deviations are allowed. K, L and M can be any positive or negative numerical constant. Each operator must be surrounded by space characters.*

Returns

<expression>

Examples

```
CALCULATE:MATH (((64 * X) + -1.07E7) / 1E6)
```

CALCulate:MATH:STATe

Switches on/off the mathematical function.

NOTE. *The CALCulate subsystem is automatically enabled when MATH operations are switched on. This means that other enabled calculate sub-blocks are indirectly switched on. Switching off mathematics, however, does not switch off the CALCulate subsystem.*

Group

Calculate

Syntax

```
CALCulate:MATH:STATe <Boolean>
CALCulate:MATH:STATe?
```

Arguments

<BOOLEAN> = (1/ON | 0/OFF)

Returns

1|0

Examples

```
CALCULATE:MATH:STATE 1
```

This example switches on mathematics.

CALCulate:STATe

Switches on/off the complete post-processing block. If disabled, neither mathematics or limit-monitoring can be done.

Group Calculate

Syntax
```
CALCulate:STATe <Boolean>
CALCulate:STATe?
```

Arguments <BOOLEAN> = (1/ON | 0/OFF)

Returns 1|0

OFF

Examples
```
CALCULATE:STATE
```

```
CALC:STAT 1
```

Switches on Post Processing.

CALCulate:TOTalize:TYPE

Selects postprocessing for totalize.

NOTE. *If both counting registers (primary and secondary channel) are being used, you can manipulate the measurement results before presentation by selecting one of three postprocessing formulas that operate directly on the raw data.*

Group Calculate

Syntax
```
CALCulate:TOTalize:TYPE APLUSB|AMINUSB|ADIVB
CALCulate:TOTalize:TYPE?
```

Arguments APLUSB selects the expression A + B to add the results in the two registers.

AMINUSB selects the expression A - B to subtract the value in register B from the value in register A.

ADIVB selects the expression A / B to calculate the ratio of the contents in registers A and B.

Returns APLUSB|AMINUSB|ADIVB

Examples `CALCULATE:TOTALIZE:TYPE ADIVB` selects the formula A / B.

CALibration:INTerpolator:AUTO

The FCA3000, FCA3100, and MCA3000 Series are reciprocal instruments that use an interpolating technique to increase the measurement resolution. In time measurements, for example, interpolation increases the resolution from 10 ns to 0.1 ns.

The instrument calibrates the interpolators automatically once for every measurement when this command is ON. When this command is OFF, the instrument does no calibrations but uses the values from the last preceding calibration. The intention of this command is to turn off the auto calibration for applications that dump measurements into the internal memory. This will increase the measurement speed.

Group Calibration

Syntax
```
CALibration:INTerpolator:AUTO <Boolean>
CALibration:INTerpolator:AUTO?
```

Arguments `<BOOLEAN>` = (1 | ON / 0 | OFF)

Returns 1|0

*CLS (No Query Form)

The *CLS common command clears the status data structures by clearing all event registers and the error queue. It does not clear enable registers and transition filters. It clears any pending *WAI, *OPC, and *OPC?.

Group Common

Syntax `*CLS`

Examples *CLS

CONFigure:ARRay:<MeasuringFunction>

The CONFigure:ARRay command differs from the CONFigure command in that it sets up the instrument to perform the number of measurements you choose in the <array size>.

To perform the selected function, you must trigger the instrument with the READ:ARRay? or INITiate;:FETCh:ARRay? queries.

Group Configure

Syntax

```
CONFigure:ARRay:<MeasuringFunction> (<array
size>)[,<parameters> [,(<channels>)]]
CONFigure:ARRay:<MeasuringFunction>?
```

Arguments <ARRAY SIZE> sets the number of measurements in the array.

<MEASURING FUNCTION>, <PARAMETERS> and <CHANNELS> are defined for each measuring function in the following table.

Table 2-27: Measuring functions and parameters

Measuring functions	Parameters
FREQuency	[<expected value>[,<resolution>],][(@1|@2|@3|@4|@6)]
FREQuency:BURSt	[<expected value>[,<resolution>],][(@1|@2|@3)]
FREQuency:POWer	(@3)
FREQuency:PRF	[<expected value>[,<resolution>],][(@1|@2|@3)]
FREQuency:RATio	[<expected value>[,<resolution>],][(@1|@2|@3),(@1|@2|@3)]
NCYCles	(@1|@2|@3)
PDUTycycle|DCYCle	[<reference>],[(@1|@2)]
NDUTycycle	[<reference>],[(@1|@2)]
PERiod	[<expected value>[,<resolution>],][(@1|@2|@3)]
PERiod:AVERage	[<expected value>[,<resolution>],][(@1|@2|@3)]
PHASe	[<expected value>[,<resolution>],][(@1|@2),(@1|@2)]
PSLEwrate	(@1|@2)
NSLEwrate	(@1|@2)
RISE:TIME|RTIM	[<lo threshold>[,<hi threshold>[,<expected value[,<resolution>]]],][(@1|@2)]
FALL:TIME|FTIM	[<lo threshold>[,<hi threshold>[,<expected value[,<resolution>]]],][(@1|@2)]
PWIDth	[<reference>],[(@1|@2)]
NWIDth	[<reference>],[(@1|@2)]
TINTerval	[<expected value>[,<resolution>],][(@1|@2),(@1|@2)]
TSTAmp	(@1|@2)

Table 2-27: Measuring functions and parameters (cont.)

Measuring functions	Parameters
[VOLTage:]MAXimum	(@1|@2)
[VOLTage:]MINimum	(@1|@2)
[VOLTage:]PTPeak	(@1|@2)
[VOLTage:]RATIO	(@1|@2),(@1|@2)

<(@{1|2|3|4|6})> is the channel to measure on, where:

(@1) means input A [1]

(@2) means input B [1]

(@3) means input C (RF input option),

(@4) means input E (Rear panel arming input)

(@6) means the internal reference

NOTE. *The channel is expression data and it must be in parentheses ( ).*

[1] These channels are prescaled by 2 when measuring frequency, and prescaled by 1 for all other functions. An exception is burst frequency measurements, where you can choose between the two factors. See the MEASure:FREQuency:BURSt? command and the command FREQuency:BURSt:PREScaler[:STATe]. There is a tradeoff between the minimum number of pulses in a burst and the frequency range.

Examples

```
CONF:ARR:PER (7),5E3,1E6,(@4)
```

This example sets up the instrument to make seven period measurements. The expected result is 5 ms, and the required resolution is 1 ms. The EXT ARM input is the measuring input.

To make the measurements and fetch the seven measurement results:

`READ:ARRAY? 7` might return `5.23421E-3,5.12311E-3,5.87526E-3, 5.50345E-3,5.33901E-3,5.25501E-3,5.03571E-3`

CONFigure:<MeasuringFunction>

Use the configure command instead of the measure query when you want to change other settings, for instance, the input settings before making the measurement and fetching the result.

The CONFigure command controls the settings of the Input, Sense and Trigger subsystems in the instrument in order to make the best possible measurement. It also switches off any calculations with CALC:STATE OFF.

READ? or INITiate:FETCh? will make the measurement and read the resulting measured value.

Since you may not know exactly what settings the instrument has chosen to configure itself for the measurement, send an * RST before doing other manual set up measurements.

Group Configure

Syntax

```
CONFigure:<MeasuringFunction>[ <parameters>[,(<channels>)]]
CONFigure:<MeasuringFunction>?
```

Arguments <MEASURING FUNCTION>, <PARAMETERS> and <CHANNELS> are defined for each measuring function in the following table.

Table 2-28: Measuring functions and parameters

Measuring functions	Parameters
FREQuency	[<expected value>[,<resolution>],][(@1|@2|@3|@4|@6)]
FREQuency:BURSt	[<expected value>[,<resolution>],][(@1|@2|@3)]
FREQuency:POWer	(@3)
FREQuency:PRF	[<expected value>[,<resolution>],][(@1|@2|@3)]
FREQuency:RATio	[<expected value>[,<resolution>],][(@1|@2|@3),(@1|@2|@3)]
NCYCles	(@1|@2|@3)
PDUTycycle|DCYCle	[<reference>],[(@1|@2)]
NDUTycycle	[<reference>],[(@1|@2)]
PERiod	[<expected value>[,<resolution>],][(@1|@2|@3)]
PERiod:AVERage	[<expected value>[,<resolution>],][(@1|@2|@3)]
PHASe	[<expected value>[,<resolution>],][(@1|@2),(@1|@2)]
PSLEwrate	(@1|@2)
NSLEwrate	(@1|@2)
RISE:TIME|RTIM	[<lo threshold>[,<hi threshold>[,<expected value[,<resolution>]]],][(@1|@2)]
FALL:TIME|FTIM	[<lo threshold>[,<hi threshold>[,<expected value[,<resolution>]]],][(@1|@2)]
PWIDth	[<reference>],[(@1|@2)]
NWIDth	[<reference>],[(@1|@2)]
TINTerval	[<expected value>[,<resolution>],][(@1|@2),(@1|@2)]
TSTAmp	(@1|@2)
[VOLTage:]MAXimum	(@1|@2)
[VOLTage:]MINimum	(@1|@2)
[VOLTage:]PTPeak	(@1|@2)
[VOLTage:]RATIO	(@1|@2),(@1|@2)

Returns <String> contains the current measuring function and channel. The response is a <String data element> containing the same answer as for [:SENSe]:FUNCtion?.

CONFigure:TOTalize[:CONTinuous]

Postprocessing of two-channel results is done with the CALCulate command. Arming is used for realizing non-manual functions like TOTalize:GATE or ARM:STOP:TIMer.

This is a count/totalize function controlled from the GPIB interface using the command TOTalize:GATE ON|OFF.

The instrument counts up for each event on the primary input channel. The same applies to the secondary channel if it is activated. The result is one or two values depending on the presence of the secondary channel. In addition to selecting totalizing, the CONFigure:TOTalize[:CONTinuous] command also selects positive trigger slope. If you want to count negative slopes on input A, send INPut{[1]|2}:SLOPe NEG after the CONFigure:TOTalize[:CONTinuous] command. The results of successive ON-OFF periods are accumulated.

Group Configure

Syntax

```
CONFigure:TOTalize[:CONTinuous][ (@{1|2})][,(@{1|2})]
CONFigure:TOTalize[:CONTinuous]?
```

Arguments `(@{1|2})` is the primary channel: `(@{1|2})` is the secondary channel:

`(@1)` stands for input A `(@2)` stands for input B

This measurement cannot be made as a MEASure, it must be made as a CONFigure followed by INIT:CONT ON, gate control with SENS:TOT:GATE {ON|OFF} and completed with a FETCh:ARR? <array size>.

Examples

```
CONF:TOT;:INP:SLOPE NEG
```

This example sets up the instrument to totalize the negative slopes on Input A and disable the secondary channel. (Same as (@1))

Normal Program Sequence for Totalizing on A

`CONFIGURE:TOTALIZE[:CONTINUOUS]` (@1)	Set up the instrument for totalize on A, reset registers
INIT:CONT ON	Initiate the instrument continuously
TOT:GATE ON	Start totalizing
FETC:ARR? -1	Read the most recent intermediate result without stopping the totalizing

Normal Program Sequence for Totalizing on A	
TOT:GATE OFF	Stop totalizing
FETC:ARR? -1	Fetch the final result from the totalizing

NOTE. *When totalizing you often want to read intermediate results without stopping the totalizing process. FETC:ARR? -1 always outputs the current register value.*

*DDT

Sets or queries the command that the device will execute on receiving the GET interface message or the *TRG common command.

Group Common

Syntax

```
*DDT <arbitrary block>
*DDT?
```

Arguments <arbitrary block> is one of six accepted blocks:

#14INIT
#15FETC?
#15READ?
#18ARM:LAY2
#19INIT;*OPC
#215ARM:LAY2;:FETC?

Examples

```
*DDT #19INIT; *OPC?
```

DISPlay:ENABle

Turns On/Off the updating of the screen. This can be used for security reasons or to improve the GPIB speed when the screen does not need to be updated.

Group Display

Syntax

```
DISPlay:ENABle < Boolean >
DISPlay:ENABle?
```

Arguments <BOOLEAN> = (1 / ON | 0 / OFF)

Returns 1|0

*DMC

Allows you to assign a sequence of one or more program message units to a macro label. The sequence is executed when the macro label is received as a command or query. Twenty-five macros can be defined at the same time, and each macro can contain an average of 40 characters.

If a macro has the same name as a command, it masks out the real command with the same name when macros are enabled. If macros are disabled, the original command is executed.

If you define macros when macro execution is disabled, the instrument executes the * DMC command fast, but if macros are enabled, the execution time for this command is longer.

Group Common

Syntax

```
*DMC <Macro label>, <Program messages>
```

Arguments <MACRO LABEL> is a 1- to 12-character macro label. Enclose string data in quotes (“ ”or ’ ’), as shown in the example.

<PROGRAM MESSAGES> the commands to be executed when the macro label is received, both block data and string data formats can be used.

Examples

```
*DMC 'FREQUENCY?', "FUNC 'FREQ 1'; INP:LEV:AUTO ON;
ARM:SOURCE BUS; INIT:CONT ON; *TRG"
```

*EMC

This command enables and disables expansion and execution of macros. If macros are disabled, the instrument will not recognize a macro although it is defined in the instrument. (The Enable Macro command takes a long time to execute.)

Group Common

Syntax

```
*EMC <Decimal data>
*EMC?
```

Arguments

<DECIMAL DATA> = is 0 or 1. A value which rounds to 0 turns off macro execution. Any other value turns macro execution on.

NOTE. *1 or 0 is <Decimal data>, not <Boolean>! ON and OFF are not valid arguments for this command.*

Returns

{1 | 0}

1 means that macro expansion is enabled. 0 means that macro expansion is disabled.

Examples

```
*EMC 1
```

Enables macro expansion and execution.

*ESE (No Query Form)

Sets the enable bits of the standard event enable register. This enable register contains a mask value for the bits to be enabled in the standard event status register. A bit that is set true in the enable register enables the corresponding bit in the status register. An enabled bit will set the ESB (Event Status Bit) in the Status Byte Register if the enabled event occurs. (See page 3-3, *The Event Status Enable Register (ESER)*.)

Group

Common

Syntax

```
*ESE <Decimal data>
```

Arguments

<DEC.DATA> = the sum (between 0 and 255) of all bits that are true.

Table 2-29: Event status enable register (1 = enable)

Bit	Weight	Enables
7	128	PON, Power-on occurred
6	64	URQ, User Request
5	32	CME, Command Error
4	16	EXE, Execution Error

Table 2-29: Event status enable register (1 = enable) (cont.)

Bit	Weight	Enables
3	8	DDE, Device Dependent Error
2	4	QYE, Query Error
1	2	RQC, Request Control (not used)
0	1	Operation Complete

Returns <Decimal data>

Examples `*ESE 36`

In this example, command error bit 5, and query error bit 2, will set the ESB-bit of the Status Byte if these errors occur.

*ESR?

Reads out the contents of the standard event status register. Reading the Standard Event Status Register clears the register.

Group Common

Syntax `*ESR?`

Returns <dec.data> = the sum (between 0 and 255) of all bits that are true. (See Table 2-29 on page 2-53.)

FETCh:ARRay? (Query Only)

`FETCh:ARRay?` query differs from the FETCh[:SCALar]? query by fetching several measuring results at once.

An array of measurements must first be made by the commands: INITiate, MEASure:ARRay:<MeasuringFunction>? or CONFigure:ARRay:<MeasuringFunction>; READ:ARRay?.

If the array size is set to a positive value, the first measurement made is the first result to be fetched.

When the instrument has made an array of measurements, FETCh:ARRay? 10 fetches the first 10 measuring results from the output queue. The second FETCh:ARRay? 10 fetches the result 11 to 20, and so on. When the last measuring result is fetched, FETCh:ARRay? starts over again with the first result.

In totalizing for instance, you may want to read the last measurement result instead of the first one. This is possible if you set the array size to a negative number. Example: FETCh:ARRay? -5 fetches the last five results. The output queue pointer is not altered when the array size is negative. That is, the example above always returns the last five results every time the command is sent.

FETCh:ARRay? -1 is useful to fetch intermediate results in free-running or array measurements without interrupting the measurement.

Group Fetch

Syntax FETCh:ARRay? <fetch array size>|MAX

FETCh[:SCALar]? (Query Only)

The fetch query retrieves one measuring result from the measurement result buffer of the instrument without making new measurements. Fetch does not work unless a measurement was taken by the INITiate, MEASure:<MeasuringFunction>?, or READ? commands.

If the instrument has made an array of measurements, FETCh[:SCALar]? fetches the first measuring results first. The second FETCh[:SCALar]? fetches the second result and so on. When the last measuring result is fetched, fetch starts over again with the first result.

The same measuring result can be fetched again and again if the result is valid, until the following occurs:

- *RST is received.
- an INITiate, MEASure or READ command is executed
- any reconfiguration is done.
- an acquisition of a new reading is started.

If the measuring result in the output buffer is invalid, but a new measurement was started, the fetch query completes when a new measuring result becomes valid. If no new measurement was started, an error is returned.

The optional SCALar means that one result is retrieved.

Group Fetch

Syntax `FETCh[:SCALar]?`

FORMat

Sets the format in which the result is sent on the bus.

Group Format

Syntax
```
FORMat ASCii|REAL|PACKed
FORMat?
```

Related Commands FORMat:TINFormation, FETCh[:SCALar]?

Arguments `ASCII`: The length is automatically controlled by the resolution of each measurement result.

`REAL`: The length parameter is ignored; the output is always in 8-byte format.

`PACKED`: See REAL.

Returns ASC|REAL|PACK

FORMat:BORDer

Sets the order in which response data bytes formatted as REAL or PACKED are sent on the bus.

Group Format

Syntax
```
FORMat:BORDer NORMal|SWAPped
FORMat:BORDer?
```

Related Commands FORMat

Arguments `NORMAL`: Response data is sent with the MSB first and the LSB last (big-endian order)

`SWAPPED`: Response data is sent with the LSB first and the MSB last (little-endian order)

Returns NORM or SWAP

FORMat:SMAX

Sets or queries the upper limit for FETCh:ARRay? MAX command in number of samples. The command is intended to set an upper limit for use with any controllers or application programs that cannot read large amounts of data.

Group Format

Syntax

```
FORMat:SMAX <Numeric value>
FORMat:SMAX?
```

Arguments Integer N, where 4 ≤ N ≤ 10000

Returns <Numeric value>

FORMat:TINFormation

This command turns on/off the time stamping of measurements. Time stamping is always done at the start of a measurement with full measurement resolution, and is saved in the measurement buffer together with the measurement result.

The setting of this command will affect the output format of the MEASure, READ and FETCh queries. See the FETCh[:SCALar]? query.

For FETCh:SCALar?, READ:SCALar? and MEASure:SCALar? the readout consists of two values instead of one. The first value is the measured value and the second value is the timestamp value.

In FORMat ASCII mode, both the measured value and the timestamp value are given as floating-point numbers expressed in the basic units (Hz or s).

In FORMat REAL mode, the result is given as an eight-byte block containing the floating-point measured value, followed by an eight-byte block containing the floating point timestamp value.

In FORMat PACKed mode, the result is given as an eight-byte block containing the floating-point measured value followed by an eight-byte block containing the timestamp value expressed as a 64-bit integer (int64), the implicit unit being ps.

When doing readouts in array form, with FETCh:ARRay?, READ:ARRay?, or MEASure:ARRay?, the response will consist of alternating measurement values.

Group Format

Syntax
```
FORMat:TINFormation Boolean
FORMat:TINFormation?
```

FREQuency:BURSt:APERture

Sets the time length within a burst during which the burst frequency is measured.

Group Sense

Syntax
```
FREQuency:BURSt:APERture {<Numeric value>|MIN|MAX}
FREQuency:BURSt:APERture?
```

Arguments <NUMERIC VALUE> is a number between 2E-8 (20 ns) and 2 s.

Returns <Numeric value>

FREQuency:BURSt:PREScaler[:STATe]

The burst frequency limit is 300 MHz if the prescaler is ON and 160 MHz if it is OFF.

Group Sense

Syntax
```
FREQuency:BURSt:PREScaler[:STATe] <Boolean>
FREQuency:BURSt:PREScaler[:STATe]?
```

Arguments <BOOLEAN> = (1 | ON | 0 | OFF)

Returns 1 | 0

FREQuency:BURSt:STARt:DELay

Sets the burst start delay (the time length between the burst start and the actual start of the burst measuring time). This parameter controls the point of time when a measurement sample is taken.

Group Sense

Syntax
```
FREQuency:BURSt:STARt:DELay {<Numeric value>|MIN|MAX}
FREQuency:BURSt:STARt:DELay?
```

Arguments <NUMERIC VALUE> is a number between 2E-8 (20 ns) and 2 s.

Returns <Numeric value>

FREQuency:BURSt:SYNC:PERiod

Sets the synchronization delay time used in burst measurements. A correct value should be longer than the burst time and shorter than 1/PRF (the inverse of the pulse repetition frequency).

Group Sense

Syntax
```
FREQuency:BURSt:SYNC:PERiod {<Numeric value>|MIN|MAX}
FREQuency:BURSt:SYNC:PERiod?
```

Arguments <NUMERIC VALUE> is a number between 1E-6 (1 µs) and 2 s.

Returns <Numeric value>

FREQuency:POWer:UNIT

Selects dBm or W as the basic measurement unit to be displayed or read out.

Group Sense

Syntax
```
FREQuency:POWer:UNIT DBM|W
FREQuency:POWer:UNIT?
```

Arguments DBM | W

The reference level 0 dBm is 1 mW in 50 Ω. Increasing the level by 3 dB means doubling the power. Decreasing the level by 3 dB means halving the power.

Returns DBM | W

FREQuency:RANGe:LOWer

Use this command to speed up voltage measurements and Autotrigger functions when you do not need to measure on low frequencies.

Table 2-30: Time to determine trigger levels (typical)

Min. frequency limit (1 Hz)	Default (100 Hz)
8 s	80 ms

Group Sense

Syntax

```
FREQuency:RANGe:LOWer {<Numeric value>|MIN|MAX}
FREQuency:RANGe:LOWer?
```

Arguments <NUMERIC VALUE> between 1 and 50000 (Hz).

MIN sets 1 Hz.

MAX sets 50 kHz.

Returns <Numeric value>

FREQuency:REGRession

Despite its name, this command also applies to Period Average.

By continuous time stamping and linear regression analysis, the resolution compared to a normal reciprocal instrument is improved by one or two digits for measuring times between 200 ms and 100 s.

Not all combinations of settings will work:

In local mode (front panel control), this function may be overridden by the firmware:

- Measurement time < 16 us: On is changed to Auto(Off)
- Measurement time > 2.5 s: Off is changed to Auto(On)
- External arming: On is changed to Auto(Off)

An info box pops up explaining this.

In remote mode (bus control), no consistency checks are made until you try to issue an INITiate command. If, at that time, the settings are inconsistent, you get a "Settings conflict" error, and the measurement will not start.

Group Sense

Syntax

```
FREQuency:REGRession ON|OFF|AUTO
FREQuency:REGRession?
```

FUNCtion

Selects which measuring function is to be performed and on which channel(s) the instrument should measure.

Group Sense

Syntax

```
FUNCtion '<Measuring function> [<Primary channel>
[,<Secondary channel>]]'
FUNCtion?
```

Arguments <MEASURING FUNCTION> is the function you want to select. Choose a function from the following table.

<PRIMARY CHANNEL> is the channel used in all single-channel measurements and the main channel in dual-channel measurements.

<SECONDARY CHANNEL> is the other channel in dual-channel measurements. Only the primary channel may be programmed for all single channel measurements.

NOTE. *The measuring function and the channels together form one <String> that must be placed within quotation marks.*

Table 2-31: Measuring functions and channels

Measuring functions	Available channels
FREQuency	1|2|3|4|6
FREQuency:RATio	1|2|3,1|2|3
FREQuency:BURSt	1|2|3
FREQuency:PRF	1|2|3
NCYCles	1|2|3

Table 2-31: Measuring functions and channels (cont.)

Measuring functions	Available channels
PDUTycycle|DCYCle	1|2
NDUTycycle	1|2
PERiod	1|2|3
PERiod:AVERage	1|2|3
PHASe	1|2,1|2
PSLEwrate	1|2
NSLEwrate	1|2
RISE:TIME|RTIM	1|2
FALL:TIME|FTIM	1|2
PWIDth	1|2
NWIDth	1|2
TINTerval	1|2,1|2
TSTAmp	1|2
[VOLTage:]MAXimum	1|2
[VOLTage:]MINimum	1|2
[VOLTage:]PTPeak	1|2
[VOLTage:]RATIO	1|2,1|2

Returns "<Measuring function>,<Primary channel>[,<Secondary channel>]"

Examples Select a pulse period measurement on input A (channel 1):

```
FUNCTION'PERIOD 1'
```

*GMC? (Query Only)

This command returns the definition of the specified macro label.

Group Common

Syntax

```
*GMC? < macro label>
```

Arguments <Macro label> = the label of the macro for which you want to see the definition. (String data must be surrounded by or " "or ' ' as in the example below.)

Returns <Block data>

Examples

```
*GMC? 'AUTOTRGLVL?'
```

Returns a block data response, for example:

```
#242:FUNC 'FREQ 1'; INP:LEV:AUTO ONCE; INP:LEV?
```

HCOPy:SDUMp:DATA? (Query Only)

Returns block data containing screen dump in Windows BMP format.

Group Hard Copy

Syntax

```
HCOPy:SDUMp:DATA?
```

Returns #43942<Binary BMP Data>

The '4' means that the following four digits (3942) tell how many data bytes will succeed. The proper screen data is preceded by a 62-byte header, which means that 3942 - 62 = 3880 bytes carry the pixel information. The number of pixels is 3880 x 8 = 31040. The display geometry is 320 x 97 = 31040.

HF:ACQuisition[:STATe]

Switches the automatic acquisition system on or off. ON means Automatic Acquisition, OFF means Manual Acquisition. When the instrument is switched from remote to local operation, Automatic Acquisition mode is entered, irrespective of the previous remote setting.

Group Sense

Syntax

```
HF:ACQuisition[:STATe] <Boolean>
HF:ACQuisition[:STATe]?
```

Arguments

```
<BOOLEAN> = {1 | ON} | {0 | OFF}
```

Returns 1 | 0

HF:FREQuency:CENTer

Sets the center frequency value for the RF input. Used when Manual Acquisition is selected.

Group Sense

Syntax
```
HF:FREQuency:CENTer <Numeric value>
HF:FREQuency:CENTer?
```

Arguments <Numeric value> = a number between $3*10^8$ (Hz) and $27*10^9$, $40*10^9$, $46*10^9$ or $60*10^9$ (Hz), depending on the model number -27G, -40G, -46G or -60G respectively.

Returns <Numeric value>

*IDN? (Query Only)

Returns the manufacturer, model, serial number, and firmware level in an ASCII response data element. The query must be the last query in a program message.

Response is <Manufacturer>, <Model>, <Serial Number>, <Firmware Level>.

Group Common

Syntax
```
*IDN?
```

Examples `*IDN?` might return `<MANUFACTURER>, <MODEL>, 1234567, V1.01 28 Jun 2004`

INITiate (No Query Form)

The INITiate command initiates a measurement. Executing an INITiate command changes the instrument trigger subsystem state from idle-state to wait-for-bus-arm-state. The trigger subsystem will continue to the other states, depending on programming. With the *RST setting, the trigger subsystem will bypass all its states and make a measurement, then return to idle state. (See page 2-23, *Trigger Subsystem*.)

Group Initiate

Syntax `INITiate`

INITiate:CONTinuous

The trigger system could continuously be initiated with this command. When Continuous is OFF, the trigger system remains in the idle-state until Continuous is set to ON or the INITiate is received. When Continuous is set to ON, the completion of a measurement cycle immediately starts a new trigger cycle without entering the idle-state. In other words, the instrument is continuously measuring and storing response data.

Group Initiate

Syntax
```
INITiate:CONTinuous <Boolean>
INITiate:CONTinuous?
```

Returns 1|0

INPut{[1]|2}:ATTenuation

Attenuates the specified input channel signal by 1 or 10. The attenuation is automatically set if the input level is set to AUTO.

Group Input

Syntax
```
INPut{[1]|2}:ATTenuation {<Numeric value>| MAX | MIN }
INPut{[1]|2}:ATTenuation?
```

Arguments `INPut{[1]|2}` specifies the input channel to set (1 = A, 2 = B). If no value is entered for this argument, the command sets the attenuation for Input A.

Numeric values <5 or `MIN` set the attenuation to 1.

Numeric values ≥5 or `MAX` set the attenuation to 10.

Returns 1.00000000000E+000|1.00000000000E+001

Examples `INPUT:ATTENUATION 10` sets the Input A attenuation to x10.

`INPUT2:ATTENUATION MIN` sets the Input B attenuation to x1.

INPut{[1]|2}:COUPling

Selects AC coupling (normally used for frequency measurements), or DC coupling (normally used for time measurements).

Group Input

Syntax
```
INPut{[1]|2}:COUPling {AC|DC}
INPut{[1]|2}:COUPling?
```

Returns AC|DC

Examples
```
INPUT{[1]|2}:COUPLING DC
INPUT{[1]|2}:COUPLING AC
```

INPut{[1]|2}:FILTer

Switches on or off the analog low pass filter on input 1 (A) and/or input 2 (B). It has a cutoff frequency of 100 kHz.

Group Input

Syntax
```
INPut{[1]|2}:FILTer <Boolean>
INPut{[1]|2}:FILTer?
```

Arguments `INPut{[1]|2}` specifies the input channel to set (1 = A, 2 = B). If no value is entered for this argument, the command sets the attenuation for Input A.

`<BOOLEAN>` = {1 | ON} | {0 | OFF}

Returns 1|0

INPut{[1]|2}:FILTer:DIGital

Switches on or off the digital low pass filter on input 1 (A) and/or input 2 (B). The cutoff frequency is set by the command: INPut{[1]|2}:FILTer:DIGital:FREQuency

Group Input

Syntax

```
INPut{[1]|2}:FILTer:DIGital <Boolean>
INPut{[1]|2}:FILTer:DIGital?
```

Arguments

`INPut{[1]|2}` specifies the input channel to set (1 = A, 2 = B). If no value is entered for this argument, the command sets the attenuation for Input A.

`<BOOLEAN>` = {1 | ON} | {0 | OFF}

Returns

1|0

INPut{[1]|2}:FILTer:DIGital:FREQuency

Any frequency between 1 Hz and 50 MHz can be entered. The filter is activated by the command: INPut{[1]|2}:FILTer:DIGital

Group

Input

Syntax

```
INPut{[1]|2}:FILTer:DIGital:FREQuency {<Numeric value>| MIN
| MAX}
INPut{[1]|2}:FILTer:DIGital:FREQuency?
```

Arguments

`INPut{[1]|2}` specifies the input channel to set (1 = A, 2 = B). If no value is entered for this argument, the command sets the attenuation for Input A.

`<NUMERIC VALUE>` is a value between 1 and 50000000.

`MIN` sets the filter to 1 Hz.

`MAX` sets the filter to 50 MHz.

Returns

<Numeric value>

INPut{[1]|2}:IMPedance

The impedance can be set to 50 Ω or 1 MΩ.

Group

Input

Syntax

```
INPut{[1]|2}:IMPedance {<Decimal data>| MAX | MIN }
INPut{[1]|2}:IMPedance?
```

Arguments

`INPut{[1]|2}` specifies the input channel to set (1 = A, 2 = B). If no value is entered for this argument, the command sets the attenuation for Input A.

`MIN` or `<DECIMAL DATA>` that rounds off to 50 or less, sets the input impedance to 50 Ω.

`MAX` or `<DECIMAL DATA>` that rounds off to 1001 or more, sets the impedance to 1 MΩ.

Returns

5.00000000000E+001|1.00000000000E+6

Examples

`INPUT:IMPEDANCE 50` sets the input A impedance to 50 Ω.

`INPUT2:IMPEDANCE 1000000` sets the input B impedance to 1 MΩ.

INPut{[1]|2}:LEVel

Input A and input B can be individually set to autotrigger or to fixed trigger levels of between -5V and +5V in steps of 2.5mV. If the attenuator is set to 10X, the range is -50V and +50V in 25 mV steps. Setting an absolute trigger level turns off autotrigger for the selected channel.

For autotrigger, see INPut{[1]|2}:LEVel:AUTO.

Group

Input

Syntax

```
INPut{[1]|2}:LEVel {<Decimal data>| MAX | MIN }
INPut{[1]|2}:LEVel?
```

Arguments

`INPut{[1]|2}` specifies the input channel to set (1 = A, 2 = B). If no value is entered for this argument, the command sets the attenuation for Input A.

`<DECIMAL DATA>` is a number between -5V and +5V if att = 1X, and between -50V and +50V if att = 10X.

`MAX` sets +5 V or +50 V and `MIN` sets -5 V or -50 V, depending on the attenuator setting.

Returns

<Decimal data>

Examples

```
INPUT:LEVEL 0.01
INPUT2:LEVEL 2.0
```

INPut{[1]|2}:LEVel:AUTO

If set to AUTO, the instrument automatically controls the trigger level.

The autotrigger function normally sets the trigger levels to 50 % of the signal amplitude, except for the following measurements or modes:

- Rise/Fall time measurements: Here the Input 1 (A) trigger level is set to 10% resp. 90% and the Input 2 (B) trigger level is set to 90% respectively. 10% of the amplitude.
- Frequency and Period Average mode: The input trigger levels are set to 70% and 30% of the signal amplitude.
- Functions for which AUTO does not work are Frequency or Period Back-to-Back, Time Interval Error (TIE) and Totalize. If one of these is selected, an AUTO ONCE is performed instead.

Group Input

Syntax

```
INPut{[1]|2}:LEVel:AUTO {<Boolean> | ONCE}
INPut{[1]|2}:LEVel:AUTO?
```

Arguments

`INPut{[1]|2}` specifies the input channel to set (1 = A, 2 = B). If no value is entered for this argument, the command sets the attenuation for Input A.

`<BOOLEAN>` = {1 | ON} | {0 | OFF}.

`ONCE` sets the instrument to make one automatic calculation of the trigger level at the beginning of a measurement. This value is then used until another level-setting command is sent to the instrument, or until a new measurement is initiated.

INPut{[1]|2}:LEVel:RELative

When autotrigger is active, the relative trigger levels are normally fixed at values that depend on the selected function, for instance 10% (Input A) and 90% (Input B) for Rise Time, 50% (Input A & Input B) for Time Interval, 70% (Input A) and 30% (Input B) for Frequency. At times you may want to change these values. Since the default values are restored automatically after changing function, this command may have to be sent repeatedly. The two input channels are programmed separately and are not interdependent.

The command itself does not switch on autotrigger, so if you want to set relative levels after having used absolute levels, you must also send the command INPut{[1]|2}:LEVel:AUTO, unless you have changed measurement function.

Group Input

Syntax

```
INPut{[1]|2}:LEVel:RELative <Numeric value>
INPut{[1]|2}:LEVel:RELative?
```

Arguments

`INPut{[1]|2}` specifies the input channel to set (1 = A, 2 = B). If no value is entered for this argument, the command sets the attenuation for Input A.

`<NUMERIC VALUE>` is a positive number between 0 and 100 (%).

Returns

<Numeric value>

Examples

`INPUT:LEVEL:RELATIVE 20` (Input A set to 20% to measure ECL rise time)

`INPUT2:RELATIVE 80` (Input B set to 80% to measure ECL rise time)

INPut{[1]|2}:SLOPe

Selects if the instrument should trigger on a positive or a negative transition. Selecting negative slope is useful for Time Interval measurements.

The slope is fixed for Pos/Neg Pulse Width/Duty Factor and Rise/Fall Time.

Arming slope is not affected by this command. Use ARM:STARt:SLOPe and ARM:STOP:SLOPe instead.

Group

Input

Syntax

```
INPut{[1]|2}:SLOPe {POS | NEG}
INPut{[1]|2}:SLOPe?
```

Arguments

`INPut{[1]|2}` specifies the input channel to set (1 = A, 2 = B). If no value is entered for this argument, the command sets the attenuation for Input A.

`POS` sets the instrument to trigger on a positive signal transition.

`NEG` sets the instrument to trigger on a negative signal transition.

Returns

POS | NEG

Examples

```
INPUT:SLOPE POS
```

```
INPUT2:SLOPE NEG
```

*LMC? (Query Only)

Makes the instrument send a list of string data elements, containing all macro labels defined in the instrument.

Group Common

Syntax `*LMC?`

Returns <String> { ,<String> }

<String> = a Macro label. (String data is surrounded by quotes as in the example below.)

Examples *LMC? might return `AUTOFILT, "AMPLITUDE?"`

*LRN?

Learn Device Setup Query. Causes a response message that can be sent to the instrument to return it to the state it was in when the *LRN? query was made.

Group Common

Syntax `*LRN?`

Returns :SYST:SET_<Block data>

Where: <Block data> is #3104<104 data bytes>

Examples *LRN?

MEASure:ARRay:FREQuency:BTBack? (Query Only)

This is the inverse function of Period Back-to-Back. See MEASure:ARRay:PERiod:BTBack?. If CALCulate:AVERage:STATe is `ON`, measurement time is used for pacing the time stamps. The pacing parameter is not used in this case. Thus a series of consecutive frequency average measurements without dead time can be made in order to fulfil the requirements for correct calculation of Allan variance or deviation.

Group Measurement

Syntax `MEASure:ARRay:FREQuency:BTBack? (<array size>)[,(@1)|(@2)]`

Arguments `<ARRAY SIZE>` sets the number of samples. `(@1)|(@2)` is the measurement channel: `(@1)` means input A

`(@2)` means input B

MEASure:ARRay:<MeasuringFunction>? (Query Only)

The `MEASure:ARRay:<MeasuringFunction>?` query differs from the MEASure:<MeasuringFunction>? query in that it performs the number of measurements you decide in the <array size> and sends all the measuring results in one string to the controller.

NOTE. *The array size for MEASure and CONFigure, and the channels, are expression data that must be in parentheses ( ).*

The `MEASure:ARRay:<MeasuringFunction>?` query is a compound query identical to: :ABORt; CONFigure:ARRay:<Meas-func>(<array-size>); READ:ARRay?(<array-size>)

Group Measurement

Syntax
```
MEASure:ARRay:<MeasuringFunction>?
(<arraysize>)[,[<parameters>] [,(<channels>)]]
```

Arguments `<ARRAY SIZE>` sets the number of measurements in the array. The maximum number is limited to 10000 due to the physical size of the output buffer. See also FETCH:ARR? and READ:ARR?

`<MEASURING FUNCTION>, <PARAMETERS>` and `<CHANNELS>` are defined for each measuring function in the following table.

Table 2-32: Measuring functions and parameters

Measuring functions	Parameters
FREQuency	[<expected value>[,<resolution>],][(@1|@2|@3|@4|@6)]
FREQuency:BURSt	[<expected value>[,<resolution>],][(@1|@2|@3)]
FREQuency:POWer	(@3)

Table 2-32: Measuring functions and parameters (cont.)

Measuring functions	Parameters
FREQuency:PRF	[<expected value>[,<resolution>],][(@1|@2|@3)]
FREQuency:RATio	[<expected value>[,<resolution>],][(@1|@2|@3),(@1|@2|@3)]
NCYCles	(@1|@2|@3)
PDUTycycle|DCYCle	[<reference>],[(@1|@2)]
NDUTycycle	[<reference>],[(@1|@2)]
PERiod	[<expected value>[,<resolution>],][(@1|@2|@3)]
PERiod:AVERage	[<expected value>[,<resolution>],][(@1|@2|@3)]
PHASe	[<expected value>[,<resolution>],][(@1|@2),(@1|@2)]
PSLEwrate	(@1|@2)
NSLEwrate	(@1|@2)
RISE:TIME|RTIM	[<lo threshold>[,<hi threshold>[,<expected value[,<resolution>]]],][(@1|@2)]
FALL:TIME|FTIM	[<lo threshold>[,<hi threshold>[,<expected value[,<resolution>]]],][(@1|@2)]
PWIDth	[<reference>],[(@1|@2)]
NWIDth	[<reference>],[(@1|@2)]
TINTerval	[<expected value>[,<resolution>],][(@1|@2),(@1|@2)]
TSTAmp	(@1|@2)
[VOLTage:]MAXimum	(@1|@2)
[VOLTage:]MINimum	(@1|@2)
[VOLTage:]PTPeak	(@1|@2)
[VOLTage:]RATIO	(@1|@2),(@1|@2)

Returns <Measuring result>{[,<measuring result>]}

Examples `MEASURE:ARRAY:FREQUENCY? (10)` returns ten measurement results.

MEASure:ARRay:PERiod:BTBack? (Query Only)

Every positive or negative zero crossing (depending on the selected slope) up to the maximum frequency (125 kHz with interpolator calibration ON or 250 kHz with interpolator calibration OFF) is time-stamped. For every new time stamp the previous value is subtracted from the current value, and the result is stored.

If CALCulate:AVERage:STATe is ON, the array contains all periods up to the maximum input frequency. For higher frequencies the average period time during the 4 µs or 8 µs observation time is stored. So, for higher frequencies the actual function is rather Period Average Back-to-Back.

The main purpose of this function is to make continuous measurements of relatively long period times without losing single periods due to result processing. A typical example is the 1-pps timebase output from GPS receivers.

Group Measurement

Syntax `MEASure:ARRay:PERiod:BTBack? (<array size>)[,(@1)|(@2)]`

Arguments `<ARRAY SIZE>` sets the number of samples. `(@1)|(@2)` is the measurement channel: `(@1)` means input A

`(@2)` means input B

MEASure:ARRay:STSTamp? (Query Only)

A time stamp (TS) is taken of the trigger level crossing on the selected input channel. The commands MEASure:ARRay:<MeasuringFunction>? and CONFigure:ARRay:<MeasuringFunction> automatically invoke FORMat:TINFormation `ON` to get the time stamp data, but when FUNCtion is used instead, you should normally let it be preceded by the FORMat:TINFormation `ON` command explicitly. Otherwise the TS [1] values are omitted. See Returned format below.

The deadtime to the next TS is due to pacing and interpolator calibration and can go down to 4 µs. The X register/counter records the number of trigger level crossings.

Depending on the state of the command FORMat:TINFormation, one or two values are output for each TS. If `OFF`, only the content of the X register/counter at the timestamp is output. If `ON`, both the X register/counter and the TS value are read and output as two values, separated by a comma in ASCII and REAL mode.

[1] **TS is the time stamp value in seconds since a certain start event that is not available for external control. Therefore the TS values can only be used for relative time measurements.**

Group Measurement

Syntax `MEASure:ARRay:STSTamp? (<array size>)[,(@1)|(@2)]`

Arguments Array size is the number of TS. One TS can contain 1 or 2 numeric values depending on the state of the FORMat:TINFormation command.

Returns <number of trg lvl crossings>,(<TS for trg lvl crossing>,)...deadtime...<number of trg lvl crossings>(<TS for trg lvl crossing>,)...deadtime...and so forth.

The format is set by the FORMat command, and the data in parentheses is sent if FORMat:TINFormation ON is active.

MEASure:ARRay:TIError? (Query Only)

This command automatically performs TIE measurements on clock signals from a predefined collection of system frequencies: 4, 8, 15.75, 64 kHz or 1.544, 2.048, 5, 10, 27, 34, 45, 52 MHz

TIE is defined as positive and increasing if the measured frequency exceeds the reference frequency.

Group Measurement

Syntax

```
MEASure:ARRay:TIError?  (array size)[,[<exp
value>[,<resol>],][(@1|(@2)]]
```

MEASure:ARRay:TSTAmp? (Query Only)

Time stamps are taken of all positive and negative trigger level crossings of the selected input channel. The commands MEASure:ARRay:<MeasuringFunction>? and CONFigure:ARRay:<MeasuringFunction> automatically invoke FORMat:TINFormation ON to get the time stamp data, but when FUNCtion is used instead, you should normally let it be preceded by the FORMat:TINFormation ON command explicitly. Otherwise the TS [2] values are omitted. See Returned format below.

Measurements are performed in groups of four TS results, two positive and two negative, with no deadtime between the values. Deadtime between groups is affected by pacing and interpolator calibration, down to 4 µs.

Measurement results of 0 indicate negative trigger level crossings, whereas positive values indicate the number of positive trigger level crossings since the last reset.

[2] **TS is the time stamp value in seconds since a certain start event that is not available for external control. So the TS values can only be used for relative time measurements.**

Group Measurement

Syntax

```
MEASure:ARRay:TSTAmp?  (<array size>)[,(@1)|(@2)]
```

Arguments <array size> sets the number of samples. One complete group requires an array size of 4. It can contain 4 or 8 numeric values depending on whether FORMat:TINFormation is `OFF` or `ON`.

MEASure{:FALL:TIME|:FTIM}? (Query Only)

The transition time from 90% to 10% of the signal amplitude is measured.

The measurement is always a single measurement and the Auto-trigger is always on, setting the trigger levels to 90% and 10% of the amplitude. If you need an average transition time measurement, or other trigger levels, use the SENSe subsystem and manually set trigger levels instead.

Group Measurement

Syntax
```
MEASure{:FALL:TIME|:FTIM}?[ [<lower threshold> [,<upper
threshold>[,<expected value>[,<resolution>]]]] [,(@1|@2)] ]
```

Arguments `<LOWER THRESHOLD>`, `<UPPER THRESHOLD>`, `<EXPECTED VALUE>` and `<RESOLUTION>` are all ignored by the instrument.

`<(@1)>` or `<(@2)>` is the measurement channel (input A or input B).

MEASure:FREQuency? (Query Only)

Traditional frequency measurements. The instrument uses the <expected value> and <resolution> to calculate the Measurement Time (ACQuisition:APERture).

Group Measurement

Syntax
```
MEASure:FREQuency?[ [<expected value>[,<resolution>]]
[,<(@{1|2|3|4|6})>]]
```

Arguments <expected value> is the expected frequency,

<resolution> is the required resolution.

<(@{1|2|3|4|6})> is the channel to measure on, where:

`(@1)` means input A [3]

`(@2)` means input B [3]

`(@3)` means input C (RF input option),

(@4) means input E (Rear panel arming input)

(@6) means the internal reference

NOTE. The channel is expression data and it must be in parentheses ( ).

[3] These channels are prescaled by 2 when measuring frequency, and prescaled by 1 for all other functions. An exception is burst frequency measurements, where you can choose between the two factors. See the MEASure:FREQuency:BURSt? command and the command FREQuency:BURSt:PREScaler[:STATe]. There is a tradeoff between the minimum number of pulses in a burst and the frequency range.

Examples MEASURE:FREQUENCY? (@3) might return 1.78112526833E+009, which measures the frequency at input C.

MEASure:FREQuency:BURSt? (Query Only)

Measures the carrier frequency of a burst. The burst duration must be less than 50% of the pulse repetition frequency (PRF).

How to measure bursts is described in detail in the Operators Manual.

The instrument uses <expected value> and <resolution> to select a Measurement Time. See ACQuisition:APERture. See FREQuency:BURSt:SYNC:PERiod.

Group Measurement

Syntax

```
MEASure:FREQuency:BURSt?[ [<expected value>[,<resolution>]]
[,<(@{1|2|3|4})>]]
```

Arguments <EXPECTED VALUE> is the expected carrier frequency, <RESOLUTION> is the required resolution; for example, 1 sets 1 Hz resolution.

<(@{1|2|3|4})> is the measurement channel:

(@1) means input A [4]

(@2) means input B [4]

(@3) means input C (RF input on FCA3003, FCA3020, FCA3103, FCA3120, MCA3027, and MCA3040)

(@4) means input E (Rear panel arming input)

If you omit the channel, the instrument measures on input A (@1).

[4] The prescaling factor for these channels can be set to 1 or 2 with the command FREQuency:BURSt:PREScaler[:STATe].

MEASure:FREQuency:POWer[:AC]? (Query Only)

Measures the power of the signal on input C in dBm or W. Use the command FREQuency:POWer:UNIT to select measurement unit.

Group Measurement

Syntax `MEASure:FREQuency:POWer[:AC]?[ (@3)]`

Arguments `(@3)` is the measurement channel number of the RF input C. It is redundant in this case, as there is no other RF channel available.

MEASure:FREQuency:PRF? (Query Only)

Measures the PRF (Pulse Repetition Frequency) of a burst signal. The burst duration must be less than 50% of the pulse repetition frequency (PRF).

NOTE. *It is better to set up the measurement with the FUNCtion ":FREQ:PRF" command when measuring pulse repetition frequency. This command will allow you to set a suitable sync delay with the FREQuency:BURSt:SYNC:PERiod command.*

How to measure bursts is described in detail in the Operators Manual.

Group Measurement

Syntax `MEASure:FREQuency:PRF?[[<exp. val.>[,<res.>]][,<(@{1|2|3|4})>]]`

Arguments `<EXP. VAL.>` is the expected PRF, `<RES.>` is the required resolution.

`<(@{1|2|3|4})>` is the measurement channel:

`(@1)` means input A

`(@2)` means input B

`(@3)` means input C (RF-input option)

`(@4)` means input E (Rear panel arming input)

If you omit the channel, the instrument measures on input A`(@1)`.

The <EXPECTED VALUE> and <RESOLUTION> are used to calculate the Measurement Time (ACQuisition:APERture). The Sync. Delay is always 10ms (default value).

MEASure:FREQuency:RATio? (Query Only)

Frequency ratio measurements between two inputs.

Group Measurement

Syntax

```
MEASure:FREQuency:RATio?[ [<expected value>
[,<resolution>]][,<(@{1|2|3})>,<(@{1|2|3})>]]
```

Arguments <expected value> and <resolution> are ignored

<(@{1|2|3})>,<(@{1|2|3})> are the measurement channels: (@1) means input A,(@2) means input B, (@3) means input C (RF input option)

If you omit the channels, the instrument measures between input A and input B.

NOTE. *The channel is expression data and must be within parentheses ( ).*

Examples MEASURE:FREQUENCY:RATIO? (@1),(@3) might return 2.345625764333E+000. This example measures the ratio between input A and input C.

MEASure:<MeasuringFunction>? (Query Only)

The measure query makes a complete measurement, including configuration and readout of data. Use measure when you can accept the generic measurement without fine tuning.

NOTE. *When a CONFigure command or* MEASure:<MeasuringFunction>? *query is issued, all instrument settings are set to the *RST settings, except those specified as <parameters> and <channels> in the CONFigure command or* MEASure:<MeasuringFunction>? *query.*

You cannot use the MEASure:<MeasuringFunction>? query with CONFigure:TOTalize[:CONTinuous], since this function measures without stopping (continuously forever).

The MEASure:<MeasuringFunction>? query is a compound query identical to ABORt; CONFigure:<Meas_func>; READ?

NOTE. *Aborts all previous measurement commands if *WAI is not used.*

Group Measurement

Syntax

```
MEASure:<MeasuringFunction>?[ [<parameters>][
,(<channels>)]]
```

Arguments <MEASURING FUNCTION>, <PARAMETERS> and <CHANNELS> are defined for each measuring function in the following table.

Table 2-33: Measuring functions and parameters

Measuring functions	Parameters
FREQuency	[<expected value>[,<resolution>],][(@1|@2|@3|@4|@6)]
FREQuency:BURSt	[<expected value>[,<resolution>],][(@1|@2|@3)]
FREQuency:POWer	(@3)
FREQuency:PRF	[<expected value>[,<resolution>],][(@1|@2|@3)]
FREQuency:RATio	[<expected value>[,<resolution>],][(@1|@2|@3),(@1|@2|@3)]
NCYCles	(@1|@2|@3)
PDUTycycle|DCYCle	[<reference>],[(@1|@2)]
NDUTycycle	[<reference>],[(@1|@2)]
PERiod	[<expected value>[,<resolution>],][(@1|@2|@3)]
PERiod:AVERage	[<expected value>[,<resolution>],][(@1|@2|@3)]
PHASe	[<expected value>[,<resolution>],][(@1|@2),(@1|@2)]
PSLEwrate	(@1|@2)
NSLEwrate	(@1|@2)
RISE:TIME|RTIM	[<lo threshold>[,<hi threshold>[,<expected value[,<resolution>]]],][(@1|@2)]
FALL:TIME|FTIM	[<lo threshold>[,<hi threshold>[,<expected value[,<resolution>]]],][(@1|@2)]
PWIDth	[<reference>],[(@1|@2)]
NWIDth	[<reference>],[(@1|@2)]
TINTerval	[<expected value>[,<resolution>],][(@1|@2),(@1|@2)]
TSTAmp	(@1|@2)
[VOLTage:]MAXimum	(@1|@2)
[VOLTage:]MINimum	(@1|@2)
[VOLTage:]PTPeak	(@1|@2)
[VOLTage:]RATIO	(@1|@2),(@1|@2)

MEASure:ARRay:<MeasuringFunction>?

Returns <data> Where the format of the returned data is determined by the format commands FORMat.

Examples `MEASURE:FREQUENCY? (@3)` might return `1.78112526833E+009`

MEASure:MEMory? (Query Only)

Same as MEASure:MEMory<N>? ommand but somewhat slower. Allows use of all memories from 1 through 19.

Group Measurement

Syntax `MEASure:MEMory? <N>`

Examples `MEASURE:MEMORY? 13` recalls the instrument setting in memory number 13, takes a measurement, and fetches the result.

MEASure:MEMory<N>? (Query Only)

Use this command when you want to measure several parameters fast.

MEAS:MEM1? recalls the contents of memory one and reads out the result, MEAS:MEM2? recalls the contents of memory two and reads out the result, and so forth.

The equivalent command sequence is *RCL 1; READ?.

The allowed range for <N> is 1 to 9. Use the somewhat slower MEASure:MEMory? command if you use memories 10 to 19.

Group Measurement

Syntax `MEASure:MEMory<N>?`

Returns <measurement result>

MEASure:NDUTycycle? (Query Only)

Traditional negative duty cycle measurement is performed. That is, the ratio between the on time and the off time of the input pulse is measured.

Group Measurement

Syntax `MEASure:NDUTycycle?[ [<threshold>] [,(@{1|2})]]`

Arguments `<THRESHOLD>` parameter sets the trigger levels in volts. If omitted, the auto trigger level is set to 50 percent of the signal.

`(@{1|2})` is the measurement channel: `(@1)` means input A, `(@2)` means input B.

If you omit the channel, the instrument measures on input A (@1).

Examples `MEASURE:NDUTYCYCLE?` might return +5.097555E-001. In this example, the duty cycle is 50.97%.

MEASure:NWIDth? (Query Only)

A negative pulse width measurement is performed.

This is always a single measurement. If you need an average pulse width measurement, use the SENSe subsystem instead.

Group Measurement

Syntax `MEASure:NWIDth?[ [<threshold>] [,<(@{1|2})>] ]`

Arguments `<THRESHOLD>` parameter sets the trigger levels in volts. If omitted, the auto trigger level is set to 50 percent of the signal.

`<(@{1|2})>` is the measurement channel:

`(@1)` means input A

`(@2)` means input B.

If you omit the channel, the instrument measures on input A.

MEASure{:PDUTycycle|:DCYCle}? (Query Only)

Traditional positive duty cycle measurement is performed. That is, the ratio between the on time and the off time of the input pulse is measured.

Group Measurement

Syntax `MEASure{:PDUTycycle|:DCYCle}?[ [<threshold>] [,(@{1|2})]]`

Arguments `<THRESHOLD>` parameter sets the trigger levels in volts. If omitted, the auto trigger level is set to 50 percent of the signal.

`(@{1|2})` is the measurement channel: `(@1)` means input A, `(@2)` means input B.

If you omit the channel, the instrument measures on input `A (@1)`.

Examples `MEASURE:PDUTYCYCLE?` might return `+5.097555E-001`. In this example, the duty cycle is 50.97%

MEASure:PERiod? (Query Only)

A period time measurement is taken on a single period. Measuring time set by the ACQuisition:APERture command does not affect the measurement.

The <expected value> and <resolution> are used to calculate the Measurement Time (ACQuisition:APERture).

Group Measurement

Syntax
```
MEASure:PERiod?[ [<expected value>
[,<resolution>]][,<(@{1|2|3})>]]
```

Arguments `<EXPECTED VALUE>` is the expected Period,

`<RESOLUTION>` is the required resolution,

`<(@{1|2|3})>` is the measurement channel:

`(@1)` means input A

`(@2)` means input B

`(@3)` means input C (RF input option).

If you omit the channel, the instrument measures on input A (@1).

MEASure:PERiod:AVERage? (Query Only)

A traditional period time measurement is performed on multiple periods. Measuring time set by the ACQuisition:APERture command determines the resolution.

The <expected value> and <resolution> are used to calculate the Measurement Time (ACQuisition:APERture).

Group Measurement

Syntax

```
MEASure:PERiod:AVERage?[ [<expected value>
[,<resolution>]][,<(@{1|2|3})>]]
```

Arguments

<EXPECTED VALUE> is the expected Period,

<RESOLUTION> is the required resolution,

<(@{1|2|3})> is the measurement channel:

(@1) means input A

(@2) means input B

(@3) means input C (RF input option).

If you omit the channel, the instrument measures on input A (@1).

MEASure:PHASe? (Query Only)

A traditional PHASe measurement is performed.

Group Measurement

Syntax

```
MEASure:PHASe?[ [<expected value>[,<resolution>]]
[,(@{1|2}),(@{1|2})]]
```

Arguments

<EXPECTED VALUE> and <RESOLUTION> are ignored by the instrument.

The first (@{1|2}) is the start channel and the second (@{1|2}) is the stop channel, (@1) means input A, (@2) means input B.

If you omit the channel, the instrument measures between input A and input B.

MEASure:PWIDth? (Query Only)

A positive pulse width measurement is performed.

This is always a single measurement. If you need an average pulse width measurement, use the SENSe subsystem instead.

Group Measurement

Syntax `MEASure:PWIDth?[ [<threshold>] [,<(@{1|2})>] ]`

Arguments `<THRESHOLD>` parameter sets the trigger levels in volts. If omitted, the auto trigger level is set to 50 percent of the signal.

`<(@{1|2})>` is the measurement channel:

`(@1)` means input A

`(@2)` means input B.

If you omit the channel, the instrument measures on input A.

MEASure{:RISE:TIME|:RTIM}? (Query Only)

The transition time from 10% to 90% of the signal amplitude is measured. The measurement is always a single measurement and the Auto-trigger is always on, setting the trigger levels to 10% and 90% of the amplitude. If you need an average transition time measurement or other trigger levels, use the SENSe subsystem and manually set trigger levels instead.

Group Measurement

Syntax `MEASure{:RISE:TIME|:RTIM}?[ [<lower threshold> [,<upper threshold>[,<expected value>[,<resolution>]]]] [,(@1|@2)]`

Arguments `<LOWER THRESHOLD>`, `<UPPER THRESHOLD>`, `<EXPECTED VALUE>` and `<RESOLUTION>` are all ignored by the instrument.

`<(@1)>` or `<(@2)` is the measurement channel (input A or input B).

MEASure:TINTerval? (Query Only)

Traditional time-interval measurements are performed. The trigger levels are set automatically, and positive slope is used. The first channel in the channel list is the start channel, and the second is the stop channel.

Group Measurement

Syntax `MEASure:TINTerval? (@{1|2}),(@{1|2})]`

Arguments The first `(@{1|2|4})` is the start channel and the second `(@{1|2|4})` is the stop channel. `(@1)` means input A

`(@2)` means input B.

If you omit the channel, input A is the start channel, and input B is the stop channel.

MEASure[:VOLT]:MAXimum? (Query Only)

This command measures the positive peak voltage with the input DC coupled.

Group Measurement

Syntax `MEASure[:VOLT]:MAXimum?[ ({@1|@2})]`

Arguments `({@1|@2})` is the measurement channel. `(@1)` means input A, `(@2)` means input B.

MEASure[:VOLT]:MINimum? (Query Only)

This command measures the negative peak voltage with the input DC coupled.

Group Measurement

Syntax `MEASure[:VOLT]:MINimum?[ ({@1|@2})]`

Arguments `({@1|@2})` is the measurement channel. `(@1)` means input A, `(@2)` means input B.

MEASure[:VOLT]:NCYCles? (Query Only)

If FREQ:BURSt is active, this function measures the number of cycles in each burst.

<(@{1|2|3})>,<(@{1|2|3})> are the measurement channels: (@1) means input A, (@2) means input B, and (@3) means input C (RF input option).

NOTE. *The channel is expression data and must be within parentheses ( ).*

Group Measurement

Syntax `MEASure[:VOLT]:NCYCles?[ [(@1|@2|@3)]`

Returns <Numeric value (integer)>

Examples `MEASURE:VOLT:NCYCLES?` (@3) might return `2356`. This example shows a measurement on the RF channel.

MEASure[:VOLT]:NSLEwrate? (Query Only)

This command measures the negative slew rate in V/s on either main input channel.

Group Measurement

Syntax `MEASure[:VOLT]:NSLEwrate?[(@1|@2)]`

Arguments `(@{1|2})` is the measurement channel. `(@1)` means input A, `(@2)` means input B.

MEASure[:VOLT]:PSLEwrate? (Query Only)

This command measures the positive slew rate in V/s on either main input channel.

Group Measurement

Syntax `MEASure[:VOLT]:PSLEwrate?[ (@1|@2)]`

Arguments `(@{1|2})` is the measurement channel. `(@1)` means input A, `(@2)` means input B.

MEASure[:VOLT]:PTPeak? (Query Only)

This command measures the peak-to-peak voltage on either main input channel.

Group Measurement

Syntax `MEASure[:VOLT]:PTPeak?[ (@{1|2})].`

Arguments `(@{1|2})` is the measurement channel. `(@1)` means input A, `(@2)` means input B.

MEASure[:VOLT]:RATio? (Query Only)

This command measures the peak-to-peak voltage ratio in dB between the selected channels.

Group Measurement

Syntax `MEASure[:VOLT]:RATio?[ (@1|@2),(@1|@2)]`

Arguments `(@{1|2})` is the measurement channel. `(@1)` means input A, `(@2)` means input B.

MEMory:DATA:RECord:COUNt? (Query Only)

If the optional <Dataset Number> parameter is specified, the command returns the number of samples in the corresponding FLASH memory position 0-7.

If no parameter is specified, a comma-separated list is returned, containing the number of samples in each of the eight FLASH memory positions 0-7.

Group Memory

Syntax `MEMory:DATA:RECord:COUNt?[<Dataset Number>]`

MEMory:DATA:RECord:DELete (No Query Form)

The command erases the measurement data array in the FLASH memory position with the number (0-7) given in the command parameter <Dataset Number>.

Group Memory

Syntax `MEMory:DATA:RECord:DELete <Dataset Number>`

MEMory:DATA:RECord:FETCh? (Query Only)

The command fetches one sample from the FLASH memory position with the number (0-7) given in the command parameter <Dataset Number>.

Set the start position with the command MEMory:DATA:RECord:FETCh:STARt.

Group Memory

Syntax `MEMory:DATA:RECord:FETCh? <Dataset Number>`

MEMory:DATA:RECord:FETCh:ARRay? (Query Only)

The command fetches an array of samples from the FLASH memory position with the number (0-7) given in the command parameter <Dataset Number>.

You can either specify the number of samples to be fetched or get all samples (up to 32000) by using the MAXimum parameter.

Group Memory

Syntax `MEMory:DATA:RECord:FETCh:ARRay? <Dataset Number>,<Number of Samples>|MAXimum`

MEMory:DATA:RECord:FETCh:STARt (No Query Form)

The data pointer is set to the first sample in the Dataset entered as a number (0-7) in the command parameter <Dataset Number>.

Group Memory

Syntax `MEMory:DATA:RECord:FETCh:STARt <Dataset Number>`

MEMory:DATA:RECord:NAME? (Query Only)

If the optional <Dataset Number> parameter is specified, the command returns the name assigned to the Dataset.

If no parameter is given, the command returns a comma-separated list of all Dataset Names.

Group Memory

Syntax `MEMory:DATA:RECord:NAME?[<Dataset Number>]`

MEMory:DATA:RECord:SAVE (No Query Form)

One of the eight (0-7) memory positions must be entered, but you can also enter an optional name (max 6 characters) for easier recognition.

A default name is assigned automatically if you omit the <Label> parameter. It represents the abbreviated measurement function and the channel. For example: Period Single A will read PerA.

If the instrument is not in Hold when this command is sent, then Execution Error (-200) is placed in the error queue.

If the instrument is not in Statistics Mode when this command is sent, then Settings Conflict Error (-221) is placed in the error queue.

If specified <Dataset> already contains data, then Directory Full Error (-255) is placed in the error queue.

If there are more than 32000 samples to save, only the last 32000 is saved without notification to the operator.

Group Memory

Syntax `MEMory:DATA:RECord:SAVE <Dataset Number>[,<Label>]`

MEMory:DATA:RECord:SETTings? (Query Only)

The command returns the instrument settings used when the specified <Dataset> was saved. The format is the same as for SYSTem:SET.

Group Memory

Syntax `MEMory:DATA:RECord:SETTings? <Dataset Number>`

MEMory:DELete:MACRo (No Query Form)

This command removes an individual macro.

Group Memory

Syntax `MEMory:DELete:MACRo '<Macro name>'`

Related Commands *PMC, if you want to delete all macros.

The IEEE488.2 command *RMC (Remove Macro command) will also work. It performs exactly the same action as `MEMory:DELete:MACRo`.

Arguments `<MACRO NAME>` is the name of the macro you want to delete. `<MACRO NAME>` is a String data that must be surrounded by quotation marks.

MEMory:FREE:MACRo? (Query Only)

This command returns information of the free memory available for macros in the instrument. If no macros are specified, 1160 bytes are available.

Group Memory

Syntax `MEMory:FREE:MACRo?`

Returns <Bytes available>, <Bytes used>

MEMory:NSTates? (Query Only)

The Number of States query (only) requests the number of *SAV/*RCL instrument setting memory states available in the instrument. The instrument responds with a value that is one greater than the maximum that can be sent as a parameter to the *SAV and *RCL commands. (States are numbered from 0 to max –1.)

Group Memory

Syntax `MEMory:NSTates?`

Returns <the number of states available>

*OPC

Generates the operation complete message in the Standard Event Status Register (SESR) when all pending commands that generate an OPC message are complete. The *OPC? query places the ASCII character "1" into the output queue when all such OPC commands are complete. The *OPC? response is not available to read until all pending operations finish. (See page 3-118, *Status and Events*.)

The *OPC command allows you to synchronize the operation of the instrument with your application program. (See page 3-6, *Synchronization Methods*.)

Certain instrument operations can affect the `*OPC` response. (See Table 3-3 on page 3-6.)

Group Common

Syntax
```
*OPC
*OPC?
```

Related Commands *WAI

Examples `*OPC` generates the operation complete message in the SESR at the completion of all pending OPC operations.

`*OPC`? might return 1 to indicate that all pending OPC operations are finished.

*OPT? (Query Only)

Returns a list of all detectable options present in the instrument, with absent options represented by an ASCII ’0’.

Group Common

Syntax `*OPT?`

Returns <Timebase>,<Prescaler | Microwave converter>, <Reserved>

Where:

<Timebase> = Standard | Option 19 | Option 30 | Option 40

<Prescaler> = 0 | Option 10 | Option 14B

<Microwave converter> = 27GHz | 40GHz

<Reserved> = 0 until further notice.

Options definition:

Option 10 = 3 GHz prescaler (Input C)

Option 14B = 20 GHz prescaler (Input C)

Opton 19 = Medium timebase (Option MS)

Option 30 = High timebase (Option HS)

Option 40 = Ultra High timebase (Option US)

OUTPut:POLarity

The command controls the polarity of the pulse output, but only if it is configured as an alarm circuit. See also the command OUTPut:TYPE.

Group Output

Syntax

```
OUTPut:POLarity NORMal | INVerted
```

Arguments `NORMAL` means that the output level is high when the alarm is activated. `INVERTED` means that the output level is low when the alarm is activated.

The output amplitude is fixed at TTL levels into 50 Ω.

OUTPut:TYPE

The command controls the rear panel pulse output configuration.

Group Output

Syntax

```
OUTPut:TYPE PULSe | GATE | ALARm | OFF
OUTPut:TYPE?
```

Arguments

PULSE means that the output serves as a fixed TTL level pulse generator.

NOTE. *See SOURce:PULSe:PERiod and SOURce:PULSe:WIDTh for time parameter setting commands.*

GATE (low level) means that the output signals a pending measurement. ALARM (low or high level) means that the output has an alarm condition.

NOTE. *See command OUTPut:POLarity to change the active polarity.*

OFF (low level) means no activity.

*PMC (No Query Form)

Removes all macro definitions.

Group

Common

Syntax

*PMC

Related Commands

MEMory:DELete:MACRo '<Macro-name>' if you want to remove a single macro.

Examples

*PMC

*PSC

Enables/disables automatic power-on status register clearing. The status registers listed below are cleared when the power-on status clear flag is 1. Power-on does not affect the registers when the flag is 0.

- Service request enable register (*SRE)
- Event status enable register (*ESE)
- Operation status enable register (:STAT:OPER:ENAB)
- Questionable data/signal enable register (:STAT:QUES:ENAB)
- Device enable registers (:STAT:DREG0:ENAB)

NOTE. **RST does not affect this power-on status clear flag.*

Group Common

Syntax

```
*PSC <Decimal data>
*PSC?
```

Arguments <DECIMAL DATA> = a number that rounds to 0 turns off automatic power-on clearing. Any other value turns it on.

Returns {1 | 0}

1 is enabled and 0 is disabled.

Examples

```
*PSC1
```

This example enables automatic power-on clearing.

*PUD

Protected user data. This is a data area in which the user may write any data up to 64 characters. The data can always be read, but you can only write data after unprotecting the data area. Typical uses include storing calibration information, instrument usage time, or inventory control numbers.

The content at delivery is: #234 FACTORY CALIBRATED ON: 20YY-MM-DD, where YY = year, MM = month, DD = day.

Group Common

Syntax

```
*PUD <Arbitrary block program data>
*PUD?
```

Returns <Arbitrary block response data>, where: <arbitrary block program data> is the data last programmed with *PUD.

Examples

```
SYSTEM:UNPROTECT; *PUD #240Calibrated 1993-07-16, inventory
No.1234
```

means that <arbitrary block program data> will follow. 2 means that the two following digits will specify the length of the data block. 40 is the number of characters in this example.

*RCL (No Query Form)

Recalls one of the up to 20 previously stored complete instrument settings from the internal nonvolatile memory of the instrument.

Memory number 0 contains the power-off settings.

Group Common

Syntax `*RCL <Decimal data>`

Arguments `<DECIMAL DATA>` = a number between 0 and 19.

Examples `*RCL` 10

READ? (Query Only)

The read function performs new measurements and reads out a measuring result without reprogramming the instrument. Using the `READ?` query in conjunction with the CONFigure:<MeasuringFunction> command gives you the capability to fine tune the measurement.

If the instrument is set up to do an array of measurements, `READ?` makes all the measurements in the array, stores the results in the output buffer, and fetches the first measuring result. Use FETCh[:SCALar]? or FETCh:ARRay? to fetch other measuring results from the output buffer. The `READ?` query is identical to ABORt; INITiate; FETCh[:SCALar]?.

Aborts all previous measurement commands if *WAI is not used.

Group Read

Syntax `READ?`

Returns <data>

The format of the returned data is determined by the format commands FORMat and FORMat:FIXed.

Examples CONF:FREQ; INP:FILT ON; READ? configures the instrument to make a standard frequency measurement with the 100kHz filter on. The instrument is triggered, and data from the measurement are read out with the READ? query.

READ? takes a new measurement and fetches the result without changing the programming of the instrument.

READ:ARRay? (Query Only)

The READ:ARRay? query is identical to ABORt; INITiate; FETCh:ARRay?.

Aborts all previous measurement commands if *WAI is not used.

NOTE. *The Statistics with array readouts cannot be combined and if the individual values in a block measurement have to be stored, make sure the default command CALCulate:AVERage:STATe is OFF.*

The READ:ARRay? query differs from the READ? query by reading out several results at once after making the number of measurements previously set up by CONFigure:ARRay:<MeasuringFunction> or MEASure:ARRay:<MeasuringFunction>?.

Group Read

Syntax READ:ARRay? {<array size for FETCh>|MAX}

Arguments <ARRAY SIZE FOR FETCH> sets the number of measurement results in the array. The size must be equal to or less than the number of measurements in the output buffer. The maximum limit is 10000 due to the physical size of the output buffer.

MAX means that all the results in the output buffer are fetched.

Returns <data>[,<data>]

The format of the returned data is determined by the format commands FORMat.

Examples ARM:COUN 10; READ:ARRAY?: 5 configures the instrument to make an array of 10 standard measurements. The instrument is triggered and data from the first five measurements are read out with the READ? query.

*RMC (No Query Form)

This command removes an individual macro.

Group Common

Syntax `*RMC '<Macro name>'`

Related Commands *PMC, if you want to delete all macros.

Arguments <MACRO NAME> is the name of the macro you want to delete.

NOTE. *<MACRO NAME> is String data that must be surrounded by quotation marks.*

ROSCillator:SOURce

Selects the signal from the external reference input as timebase instead of the internal timebase oscillator. If the parameter is set to the default value AUTO, the instrument uses the external reference signal if present.

Group Sense

Syntax
```
ROSCillator:SOURce {INT|EXT|AUTO}
ROSCillator:SOURce?
```

Returns <INT|EXT|AUTO>

*RST (No Query Form)

The Reset command resets the instrument. It is the third level of reset in a 3-level reset strategy, and it primarily affects the instrument functions, not the IEEE 488 bus. (See page 4-1, *Instrument Settings After *RST*.)

All previous commands are discarded, macros are disabled, and the instrument is prepared to start new operations.

Group Common

Syntax *RST

Examples *RST

*SAV (No Query Form)

Saves the instrument settings in an internal nonvolatile memory location. Nineteen memory locations are available. Switching the power off and on does not change the settings stored in the registers.

NOTE. *The memory positions 1 to 10 can be protected from the front panel USER OPT menu. Use the SYSTem:UNPRotect command to change protected memory positions to unprotected.*

Group Common

Syntax *SAV

Arguments <DECIMAL DATA> = a number between 1 and 19.

Examples *SAV 11

SOURce:PULSe:PERiod

The pulse generator time parameters are activated when the output type is configured to pulse using the OUTPut:TYPE PULSe command.

Group Output

Syntax
```
SOURce:PULSe:PERiod <Numeric value>
SOURce:PULSe:PERiod?
```

Arguments <NUMERIC VALUE> = a number between 2E-8 (20 ns) and 2 s, in 10 ns increments.

SOURce:PULSe:WIDTh

The pulse generator time parameters are activated when the output type is configured to pulse using the OUTPut:TYPE PULSe command.

Group Output

Syntax
```
SOURce:PULSe:WIDTh <Numeric value>
SOURce:PULSe:WIDTh?
```

Arguments <NUMERIC VALUE> = a number between 2E-8 (20 ns) and 2 s, in 10 ns increments.

*SRE

The SRE (Service Request Enable) command sets or queries the service request enable register bits. This enable register contains a mask value for the bits to be enabled in the status byte register. A bit that is set true in the enable register enables the corresponding bit in the status byte register to generate a Service Request.

Group Common

Syntax
```
*SRE <Decimal data>
*SRE?
```

Arguments <Decimal data> = the sum (between 0 and 255) of all bits that are true. See the following table.

Table 2-34: Service Request Enable register (1 = enable)

Bit	Weight	Enables
7	128	OPR, Operation Status
6	64	RQS, Request Service
5	32	ESB, Event Status Bit
4	16	MAV, Message Available
3	8	QUE, Questionable Data/Signal Status
2	4	EAV, Error Available
1	2	Not used
0	1	Device Status

Returns <Integer>, the sum of all bits that are sent.

Examples *SRE? might return 16 to indicate that a message is available in the output queue.

STATus:DREGister0? (Query Only)

This query reads out the contents of the Device Event Register. Reading the Device Event Register clears the register.

Group Status

Syntax
```
STATus:DREGister0?
```

Returns <Decimal data> = the sum (between 0 and 30) of all bits that are true. See the following table.

Bit number	Weight	Condition
2	4	Last measurement below low limit.
1	2	Last measurement above high limit.

STATus:DREGister0:ENABle

This command sets the enable bit of the Device Register 0.

Group Status

Syntax
```
STATus:DREGister0:ENABle <bit mask>
STATus:DREGister0:ENABle?
```

Arguments <Decimal data> = the sum (between 0 and 6) of all bits that are true. See the following table.

Bit number	Weight	Condition
2	4	Enable monitoring of low limit
1	2	Enable monitoring of high limit

Returns <bit mask>

STATus:OPERation? (Query Only)

Reads out the contents of the operation event status register. Reading the Operation Event Register clears the register. (See Figure 2-3 on page 2-103.)

Group Status

Syntax `STATus:OPERation?`

Returns <Decimal data> = the sum (between 0 and 368) of all bits that are true. See the following table.

Bit number	Weight	Condition
8	256	No measurement
6	64	Waiting for bus arming
5	32	Waiting for triggering and/or external arming
4	16	Measurement

STATus:OPERation:CONDition? (Query Only)

Reads out the contents of the operation status condition register. This register reflects the state of the measurement process. See table below.

Group Status

Syntax `STATus:OPERation:CONDition?`

Returns <Decimal data> = the sum (between 0 and 3953) of all bits that are true. See the following table.

Bit number	Weight	Condition
11	2048	Computing statistics
10	1024	In limit
9	512	Using internal reference

Bit number	Weight	Condition
8	256	Meas. stopped / Computing statistics (in compatibility mode)
6	64	Waiting for bus arming
5	32	Waiting for triggering and / or external arming
4	16	Measurement started
0	1	Calibrating

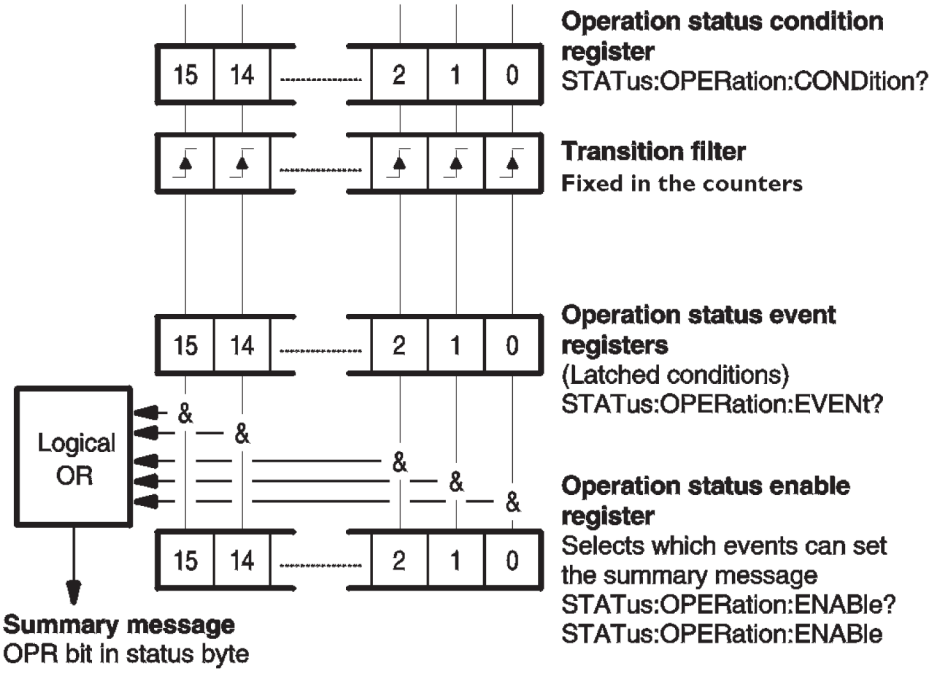


Figure 2-3: Device status continuously monitored

STATus:OPERation:ENABle

Sets the enable bits of the operation status enable register. This enable register contains a mask value for the bits to be enabled in the operation status event register. A bit that is set true in the enable register enables the corresponding bit in the status register. (See Figure 2-3 on page 2-103.)

An enabled bit will set bit #7, OPR (Operation Status Bit), in the Status Byte Register if the enabled event occurs. (See page 3-2, *The Status Byte Register (SBR)*.)

Power-on will clear this register if power-on clearing is enabled via *PSC.

Group Status

Syntax

```
STATus:OPERation:ENABle <Decimal data>
STATus:OPERation:ENABle?
```

Arguments <Decimal data> = the sum (between 0 and 368) of all bits that are true. See the following table.

Bit number	Weight	Condition
8	256	No measurement
6	64	Waiting for bus arming
5	32	Waiting for triggering and/or external arming
4	16	Measurement

Returns <Decimal data>

STATus:PRESet (No Query Form)

This command has an SCPI standardized effect on the status data structures. The purpose is to precondition these toward reporting only device-dependent status data.

- It only affects enable registers. It does not change event and condition registers.
- The IEEE-488.2 enable registers, which are handled with the common commands *SRE and *ESE remain unchanged.
- The command sets or clears all other enable registers. Those relevant for this instrument are as follows:
  - It sets all bits of the Device status Enable Registers to 1.
  - It sets all bits of the Questionable Data Status Enable Registers and the Operation Status Enable Registers to 0.
  - The following registers never change in the instrument, but they do conform to the standard STATus:PRESet values.
  - All bits in the positive transition filters of Questionable Data and Operation status registers are 1.
  - All bits in the negative transition filters of Questionable Data and Operation status registers are 0.

Group Status

Syntax `STATus:PRESet`

STATus:QUEStionable? (Query Only)

Reads out the contents of the status questionable event register. Reading the Status Questionable Event Register clears the register. (See Figure 2-3 on page 2-103.)

Group Status

Syntax `STATus:QUEStionable?`

Returns <Decimal data> = the sum (between 0 and 20324) of all bits that are true. See the following table.

Bit number	Weight	Condition
14	16384	Unexpected parameter
11	2048	Out of limit
10	1024	Measurement timeout / Out of limit (in compatibility mode)
9	512	Overflow
8	256	Calibration error
6	64	Phase interpolation calibration off
5	32	Frequency interpolation calibration off
2	4	Time interpolation calibration off

STATus:QUEStionable:CONDition? (Query Only)

Reads out the contents of the status questionable condition register.

Group Status

Syntax `STATus:QUEStionable:CONDition?`

Returns <Decimal data> = the sum (between 0 and 20324) of all bits that are true. See the following table.

Bit number	Weight	Condition
14	16384	Unexpected parameter
11	2048	Out of limit
10	1024	Measurement timeout / Out of limit (in compatibility mode)
9	512	Overflow
8	256	Calibration error
6	64	Phase interpolation calibration off
5	32	Frequency interpolation calibration off
2	4	Time interpolation calibration off

STATus:QUEStionable:ENABle (No Query Form)

Sets the enable bits of the status questionable enable register. This enable register contains a mask value for the bits to be enabled in the status questionable event register. A bit that is set true in the enable register enables the corresponding bit in the status register. (See Figure 2-3 on page 2-103.)

An enabled bit will set bit #3, QUE (Questionable Status Bit), in the Status Byte Register if the enabled event occurs. (See page 3-2, *The Status Byte Register (SBR)*.)

Power-on will clear this register if power-on clearing is enabled via *PSC.

Group Status

Syntax `STATus:QUEStionable:ENABle <Decimal data>`

Arguments <Decimal data> = the sum (between 0 and 20324) of all bits that are true. See the following table.

Bit number	Weight	Condition
14	16384	Unexpected parameter
11	2048	Out of limit

Bit number	Weight	Condition
10	1024	Measurement timeout / Out of limit (in compatibility mode)
9	512	Overflow
8	256	Calibration error
6	64	Phase interpolation calibration off
5	32	Frequency interpolation calibration off
2	4	Time interpolation calibration off

Returns <Decimal data>

Examples

```
STATUS:QUESTIONABLE:ENABLE 16896
```

In this example, both 'unexpected parameter' bit 14, and 'overflow' bit 8, will set the QUE-bit of the Status Byte when a questionable status occurs.

*STB? (Query Only)

Queries the value of the Status Byte. Bit 6 reports the Master Summary Status bit (MSS), not the Request Service (RQS). The MSS is set if the instrument has one or more reasons for requesting service.

Group Common

Syntax

```
*STB?
```

Related Commands Use a serial poll to read the status byte with the RQS bit.

Returns <Integer> = the sum (between 0 and 255) of all bits that are true. See the following table.

Table 2-35: Status Byte register (1 = true)

Bit	Weight	Name	Condition
7	128	OPR	Enabled operation status has occurred.

Table 2-35: Status Byte register (1 = true) (cont.)

Bit	Weight	Name	Condition
6	64	MSS	Reason for requesting service
5	32	ESB	Enabled status event condition has occurred
4	16	MAV	An output message is ready
3	8	QUE	The quality of the output signal is questionable
2	4	EAV	Error available
1	2		Not used
0	1	DREG0	Enabled status device event conditions have occurred

SYSTem:COMMunicate:GPIB:ADDRess

This command sets the GPIB address. It is valid until a new address is set, either by sending a new bus command or via the front panel USER OPT menu.

Group System

Syntax

```
SYSTem:COMMunicate:GPIB:ADDRess {<Numeric value>| MAX | MIN
} [,{<Numeric value>| MAX | MIN }]
SYSTem:COMMunicate:GPIB:ADDRess?
```

Arguments <NUMERIC VALUE> is a number between 0 and 30.

MIN sets address 0.

MAX sets address 30. [,<Numeric value> | MAX | MIN] sets a secondary address. This is accepted but not used in the FCA3000 Series.

Returns <Numeric value>

Examples

```
SYSTEM:COMMUNICATE:GPIB:ADDRESS 12
```

This example sets the bus address to 12.

SYSTem:ERRor? (Query Only)

Queries for an ASCII text description of an error that occurred. The error messages are placed in an error queue, with a FIFO (First In-First Out) Structure. This queue is summarized in the Error AVailable (EAV) bit in the status byte.

Group System

Syntax
```
SYSTem:ERRor?
```

Returns <error number>,"<Error Description String>", where: <Error Description String> = an error description as ASCII text. (See page 3-10, *Error Messages*.)

SYSTem:LANGuage

The user can select between two command sets, where native exploits the full capability of the instrument, and compatible facilitates portability to test systems using the Agilent instruments 53131 and 53132.

The command set described in this manual refers to the native mode only.

Group System

Syntax
```
SYSTem:LANGuage NATive | COMPatible
SYSTem:LANGuage?
```

SYSTem:PRESet (No Query Form)

This command recalls the same default settings that are entered when you push **USER OPT > Save/Recall > Recall Setup > Default**.

Table 2-36: Differences between SYSTem:PRESet and *RST

	SYSTem:PRESet	*RST
Measurement time	200 ms	10 ms
INITiate:CONTinuous state	ON	OFF

Group System

Syntax `SYSTem:PRESet`

Related Commands *RST

SYSTem:SET

Transmits in binary form the complete current state of the instrument. This data can be sent to the instrument to later restore this setting. This command has the same function as the *LRN? common command with the exception that it returns the data only as response to `SYSTem:SET`?. The query form of this command returns a definite block data element.

Group System

Syntax
```
SYSTem:SET <Block data>
SYSTem:SET?
```

Arguments `<BLOCK DATA>` is the instrument setting previously retrieved via the `SYSTem:SET`? query.

Returns <Block data>

Examples `SYSTEM:SET`? might return `#41686<data byte 1><data byte 2>...<data byte 1686>`.

NOTE. *The real number of data bytes will probably differ from the one specified above and depends on the instrument type and the firmware version.*

SYSTem:TALKonly (No Query Form)

The main purpose is to transfer streaming data fast in monitoring systems without predefined limits for time or number of samples. It is a non-reversible command; you can only return to normal bus mode by sending IFC or by pushing the Esc button on the front panel.

The Talk Only output buffer can hold one value. If a new measurement result is ready for output before the previous one was transferred, the new value is rejected and the previous transfer is left undisturbed.

A pause during the reading will cause the first value read after the pause to be the first measurement finished after the latest pre-pause value was read. The second

value read will be that of the most recently finished measurement. All values in between are lost. The same applies if there is a pause between turning on Talk Only and starting to read values. Therefore a dummy read is recommended in many cases.

Prerequisites DISPlay:ENABle should be OFF. FORMat should be REAL or PACKed. ARM:COUNt and TRIGger:COUNt should both be 1. INITiate:CONTinuous should be ON.

Smart Period/Frequency/Time Interval or any functions using voltage measurement or timestamp cannot be used with Talk Only.

Group System

Syntax `SYSTem:TALKonly ON`

SYSTem:TEMPerature? (Query Only)

This command returns the temperature in degrees C at the fan control sensor inside the instrument housing.

Group System

Syntax `SYSTem:TEMPerature?`

Returns <Numeric value>

Examples `SYSTEM:TEMPERATURE?` might return `50`.

SYSTem:TOUT

This command switches the time-out on or off. When time-out is enabled, the measurement attempt is abandoned when the time set with SYSTem:TOUT:TIME has elapsed. Depending on GPIB mode and output format, a special response message is sent to the controller instead of a measurement result, and the time-out bit in the STATus:QUEStionable? register is set.

Group System

Syntax

```
SYSTem:TOUT <Boolean>
SYSTem:TOUT?
```

Examples

Send the command sequence:

```
SYSTEM:TOUT 1
SYSTEM:TOUT:TIME 0.5
STATUS:QUESTIONABLE:ENABLE 1024
*SRE 8
```

This example turns on time-out, sets the time-out to 0.5 s, enables status reporting of questionable data at time-out, and enables service request on questionable data.

Now send the command sequence:

`*STB?` // If bit 3 in the status byte is set, read the questionable data status.

`STATUS:QUESTIONABLE?` // This query reads the questionable data status and might return 1024 or 0. 1024 means time-out has occurred, and 0 means no time-out.

SYSTem:TOUT:AUTO

This command is primarily intended for use with long measurement times to quickly determine if there is any signal at all present at the input, without having to wait for the entire measurement to time out.

If ON there is a short time-out of 2 timer ticks (10-20ms) from the INIT/ARM to the first start trigger, independent of any other time-out setting.

Group System

Syntax

```
SYSTem:TOUT:AUTO <Boolean>
SYSTem:TOUT:AUTO?
```

SYSTem:TOUT:TIME

This command sets the time-out in seconds with a resolution of 10 ms.

The 10 ms timer ticks start to be counted after either a measurement INIT (if Arming is not selected) or an external arming event (if Arming is selected). The counting stops at the stop trigger of the measurement. For block measurements a time-out results in the whole block timing out. The measurement start is not involved. See also SYSTem:TOUT:AUTO if you need a command dealing with unnecessarily long timeouts due to absence of input signal.

Note that you must enable time-out using SYSTem:TOUT ON for this setting to take effect.

Group System

Syntax
```
SYSTem:TOUT:TIME {<Numeric value>|MIN|MAX}
SYSTem:TOUT:TIME?
```

Arguments `<NUMERIC VALUE>` is the time-out in seconds. The range is 0.01 to 1000(s)

`MIN` sets 0.01s

`MAX` sets 1000 s

Returns <Numeric value>

SYSTem:UNPRotect (No Query Form)

This command will unprotect the user data (set/read by * PUD) and front setting memories 1-10 until the next PMT (Program message terminator) or Device clear or Reset (*RST). This makes it necessary to send an unprotect command in the same message as for instance *PUD.

Group System

Syntax
```
SYSTem:UNPRotect
```

Examples
```
SYSTEM:UNPROTECT;*PUD #240 Calibrated 1992-11-17, inventory
No.1234
```

Where: # means that < arbitrary block program data> will follow. 2 means that the two following digits will specify the length of the data block 40 is the number of characters in this example.

TEST:SELect

Selects which internal self-tests shall be used when self-test is requested by the *TST? command.

Group Test

Syntax
```
TEST:SELect {RAM | ROM | LOGic | DISPlay | ALL}
TEST:SELect?
```

Returns {RAM | ROM | LOGic | DISPlay | ALL}

TIError:FREQuency

An arbitrary frequency in the range 1 Hz to 100 MHz can be entered (increment = 1 Hz). Subsequent TIE measurements are made by continuous time stamping of the input signal and the internal/external time base clock. Observations of Wander, for instance, can easily be made with this command and the function MEASure:ARRay:TIError? in conjunction with the built-in statistics/graphics facilities.

Group Sense

Syntax

```
TIError:FREQuency <Numeric value>
TIError:FREQuency?
```

Arguments <NUMERIC VALUE> = a number between 1 and 1E8 (Hz) in 1 Hz increments.

TIError:FREQuency:AUTO

If AUTO is ON, a check measurement is made at the start of the block to determine if the frequency of the input signal, rounded to 4 significant digits, is listed for automatic recognition, for instance:

4, 8, 15.75, 64 kHz or 1.544, 2.048, 5, 10, 27, 34, 45, 52 MHz

If the command is successful, the found value is stored and can be recalled with a query command. Subsequent TIE measurements will use this value until it is changed by sending this command once more or by sending the setting command TIError:FREQuency <Numeric value>, which will deliberately fix the frequency.

Group Sense

Syntax

```
TIError:FREQuency:AUTO {ON|OFF}
TIError:FREQuency:AUTO?
```

Returns 1 | 0

TINTerval:AUTO

Using four time stamps (two on each channel), the instrument can determine which event precedes the other. Thus you do not have to set aside Input A as the start channel.

Group Sense

Syntax

```
TINTerval:AUTO {Boolean}
TINTerval:AUTO?
```

Arguments

```
<BOOLEAN> = {1 | ON} | {0 | OFF}
```

Returns 1 | 0

TOTalize:GATE

Open/closes the gate for CONFigure:TOTalize[:CONTinuous].

NOTE. *Before opening the gate with this command, the instrument must be in the 'continuously initiated' state (INITiate:CONTinuous ON), or else the totalizing will not start.*

Group Measurement

Syntax

```
TOTalize:GATE ON | OFF
TOTalize:GATE?
```

Arguments

```
<BOOLEAN> = (1 | ON | 0 | OFF)
```

Returns <Boolean>

Examples Example command sequence:

`CONFIGURE:TOTALIZE (@1),(@2)` - Select totalizing on inputs A & B and reset registers

`INITIATE:CONTINUOUS ON`

`TOTALIZE:GATE ON` - Initiate totalizing

`FETCH:ARRAY? -1` - Read intermediate results (A & B)

`TOTALIZE:GATE OFF` - Stop totalizing

`TOTALIZE:GATE ON` - Start totalizing and accumulate results

`TOTALIZE:GATE OFF` - Stop totalizing

`FETCH:ARRAY? −1` - Read final results (separated by a comma)

*TRG (No Query Form)

Starts the measurement and places the result in the output queue.

It is the same as ARM:LAYer2:IMM; *WAI; FETCh?.

The Trigger command is the device-specific equivalent of the IEEE 488.1 defined Group Execute Trigger (GET). It has exactly the same effect as a GET after it is received and parsed by the instrument. However, GET is much faster than *TRG because GET is a hardware signal that does not have to be parsed by the instrument.

NOTE. *Aborts all previous measurement commands if *WAI is not used.*

Group Common

Syntax

```
*TRG
```

TRIGger:COUNt

Sets how many measurements the instrument should make for each arm condition, (block arming).

These measurements are done without any additional arming conditions before the measurement. This also means that stop arming is disabled for the measurements inside a block.

NOTE. *The actual number of measurements made on each INIT equals to: (ARM:COUNT)*(TRIGGER:START:COUNT)*

Group Trigger

Syntax

```
TRIGger:COUNt {<Numeric value> | MIN | MAX}
TRIGger:COUNt?
```

Arguments `<NUMERIC VALUE>` is a number between 1 and 16777215 (2^{24} -1).

`MAX` sets 16777215

`MIN` sets 1

Returns <Numeric value>

Examples
```
TRIGGER:COUNT 50
```

TRIGger:SOURce

Enables or disables the pacing function (the sample rate control). The pacing time is set by the TRIGger:TIMer command.

Group Trigger

Syntax
```
TRIGger:SOURce TIMer | IMMediate
TRIGger:SOURce?
```

Arguments `TIMER` - enables pacing

`IMMediate` - disables pacing

TRIGger:TIMer

This command sets the sample rate, for instance in conjunction with the statistics functions.

Group Trigger

Syntax
```
TRIGger:TIMer <Numeric value> | MIN | MAX
TRIGger:TIMer?
```

Arguments `<NUMERIC VALUE>` is a time length between 2 ms and 500 s, entered in seconds.

`MIN` means 2 ms.

`MAX` means 500 s.

Returns <Numeric value>

*TST? (Query Only)

The self-test query causes an internal self-test and generates a response indicating whether or not the device completed the self-test without any detected errors.

Group Common

Syntax `*TST?`

Returns <Integer>

Where <Integer> = a number indicating specific errors as listed in the following table:

Integer	Error
0	No Error
1	RAM Failure
2	ROM Failure
4	Logic Failure
8	Display Failure
16	
32	

*WAI (No Query Form)

The Wait-to-Continue command sets the instrument to not execute any further commands or queries until execution of all previous commands or queries is completed.

Group Common

Syntax `*WAI`

Examples

```
MEASURE:FREQUENCY?; *WAI; MEASURE:PDUT?
```

In this example, *WAI makes the instrument perform both the frequency and the Duty Cycle measurement. Without *WAI, only the Duty Cycle measurement would be performed. This command sequence might return `+5.1204004E+002; +1.250030E-001`.

Status and Events

Status and Events

The instrument provides a status and event reporting system for the GPIB and USB interfaces. This system informs you of certain significant events that occur within the instrument.

The instrument status handling system consists of four 8-bit registers and two queues. The following text describes these registers and components. They also explain how the event handling system operates.

Registers

Overview

The registers in the event handling system fall into two functional groups:

- Status Registers contain information about the status of the instrument. They include the Standard Event Status Register (SESR).
- Enable Registers determine whether selected types of events are reported to the Status Registers and the Event Queue. They include the Device Event Status Enable Register (DESER), the Event Status Enable Register (ESER), and the Service Request Enable Register (SRER).

Status Registers

The Standard Event Status Register (SESR) and the Status Byte Register (SBR) record certain types of events that may occur while the instrument is in use. IEEE Std 488.2-1987 defines these registers.

Each bit in a Status Register records a particular type of event, such as an execution error or message available. When an event of a given type occurs, the instrument sets the bit that represents that type of event to a value of one. (You can disable bits so that they ignore events and remain at zero. See Enable Registers). Reading the status registers tells you what types of events have occurred.

The Standard Event Status Register (SESR). The SESR records eight types of events that can occur within the instrument. Use the *ESR? query to read the SESR register. Reading the register clears the bits of the register so that the register can accumulate information about new events.

7	6	5	4	3	2	1	0
PON	URQ	CME	EXE	DDE	QYE	RQC	OPC

Figure 3-1: The Standard Event Status Register (SESR)

Table 3-1: SESR bit functions

Bit	Function	
7 (MSB)	PON	Power On. Shows that the instrument was powered on. On completion, the diagnostic self tests also set this bit.

Table 3-1: SESR bit functions (cont.)

Bit	Function	
6	URQ	User Request. Shows that an application event has occurred. *See note.
5	CME	Command Error. Shows that an error occurred while the instrument was parsing a command or query.
4	EXE	Execution Error. Shows that an error occurred while executing a command or query.
3	DDE	Device Error. Shows that a device error occurred.
2	QYE	Query Error. Either an attempt was made to read the Output Queue when no data was present or pending, or that data in the Output Queue was lost.
1	RQC	Request Control. This is not used.
0 (LSB)	OPC	Operation Complete. Shows that the operation is complete. This bit is set when all pending operations complete following an *OPC command.

The Status Byte Register (SBR). Records whether output is available in the Output Queue, whether the instrument requests service, and whether the SESR has recorded any events.

Use a Serial Poll or the *STB? query to read the contents of the SBR. The bits in the SBR are set and cleared depending on the contents of the SESR, the Event Status Enable Register (ESER), and the Output Queue. When you use a Serial Poll to obtain the SBR, bit 6 is the RQS bit. When you use the *STB? query to read the SBR, bit 6 is the MSS bit. Reading the SBR does not clear the bits.

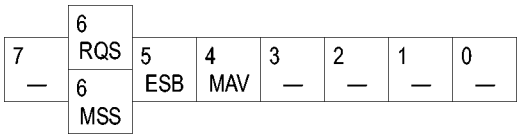


Figure 3-2: The Status Byte Register (SBR)

Table 3-2: SBR bit functions

Bit	Function	
7 (MSB)	OPR	Operation status.
6 (serial poll)	RQS	Request Service. Obtained from a serial poll. Shows that the instrument requests service from the GPIB controller.
6 (*STB? query)	MSS	Master Status Summary. Obtained from *STB? query. Summarizes the ESB and MAV bits in the SBR.
5	ESB	Event Status Bit. Shows that status is enabled and present in the SESR.
4	MAV	Message Available. Shows that output is available in the Output Queue.

Table 3-2: SBR bit functions (cont.)

Bit	Function	
3	QUE	Questionable Data/Signal Status.
2	EAV	Error Available.
1	———	Not used.
0 (LSB)	DEV	Device Status.

Enable Registers

ESER and SRER allow you to select which events are reported to the Status Registers and the Event Queue. Each Enable Register acts as a filter to a Status Register and can prevent information from being recorded in the register or queue.

Each bit in an Enable Register corresponds to a bit in the Status Register it controls. In order for an event to be reported to a bit in the Status Register, the corresponding bit in the Enable Register must be set to one. If the bit in the Enable Register is set to zero, the event is not recorded.

Various commands set the bits in the Enable Registers. The Enable Registers and the commands used to set them are described below.

The Event Status Enable Register (ESER). This register controls which types of events are summarized by the Event Status Bit (ESB) in the SBR. Use the *ESE command to set the bits in the ESER. Use the *ESE? query to read it.

7	6	5	4	3	2	1	0
PON	URQ	CME	EXE	DDE	QYE	RQC	OPC

Figure 3-3: The Event Status Enable Register (ESER)

The Service Request Enable Register (SRER). This register controls which bits in the SBR generate a Service Request and are summarized by the Master Status Summary (MSS) bit.

Use the *SRE command to set the SRER. Use the *SRE? query to read the register. The RQS bit remains set to one until either the Status Byte Register is read with a Serial Poll or the MSS bit changes back to a zero.

7	6	5	4	3	2	1	0
OPR	RQS	ESB	MAV	QUE	EAV	—	DEV

Figure 3-4: The Service Request Enable Register (SRER)

*PSC Command

The *PSC command controls the Enable Registers contents at power-on. Sending *PSC 1 sets the Enable Registers at power on as follows:

- ESER 0 (equivalent to an *ESE 0 command)
- SRER 0 (equivalent to an *SRE 0 command)

Sending *PSC 0 sets the Enable Register to store register values in nonvolatile memory through a power cycle.

NOTE. *To enable the PON (Power On) event to generate a Service Request, send *PSC 0, use the *ESE command to enable PON in the ESER, and use the *SRE command to enable bit 5 in the SRER. Subsequent power-on cycles will generate a Service Request.*

Queues

The *PSC command controls the Enable Registers contents at power-on. Sending *PSC 1 sets the Enable Registers to clear at power on.

Output Queue

The instrument stores query responses in the Output Queue and empties this queue each time it receives a new command or query message after an <EOM>. The controller must read a query response before it sends the next command (or query) or it will lose responses to earlier queries.

CAUTION. *When a controller sends a query, an <EOM>, and a second query, the instrument normally clears the first response and outputs the second while reporting a Query Error (QYE bit in the ESER) to indicate the lost response. A fast controller, however, may receive a part or all of the first response as well. To avoid this situation, the controller should always read the response immediately after sending any terminated query message or send a DCL (Device Clear) before sending the second query.*

Event Queue

The Event Queue stores detailed information on up to 32 events. If more than 32 events stack up in the Event Queue, the 32nd event is replaced by event code 350, "Queue Overflow."

Before reading an event from the Event Queue, you must use the *ESR? query to read the summary of the event from the SESR. Reading the SESR erases any events that were summarized by previous *ESR? reads but not read from the Event Queue. Events that follow an *ESR? read are put in the Event Queue but are not available until *ESR? is used again.

Event Handling Sequence

The following figure shows how to use the status and event handling system. In the explanation that follows, numbers in parentheses refer to numbers in the figure.

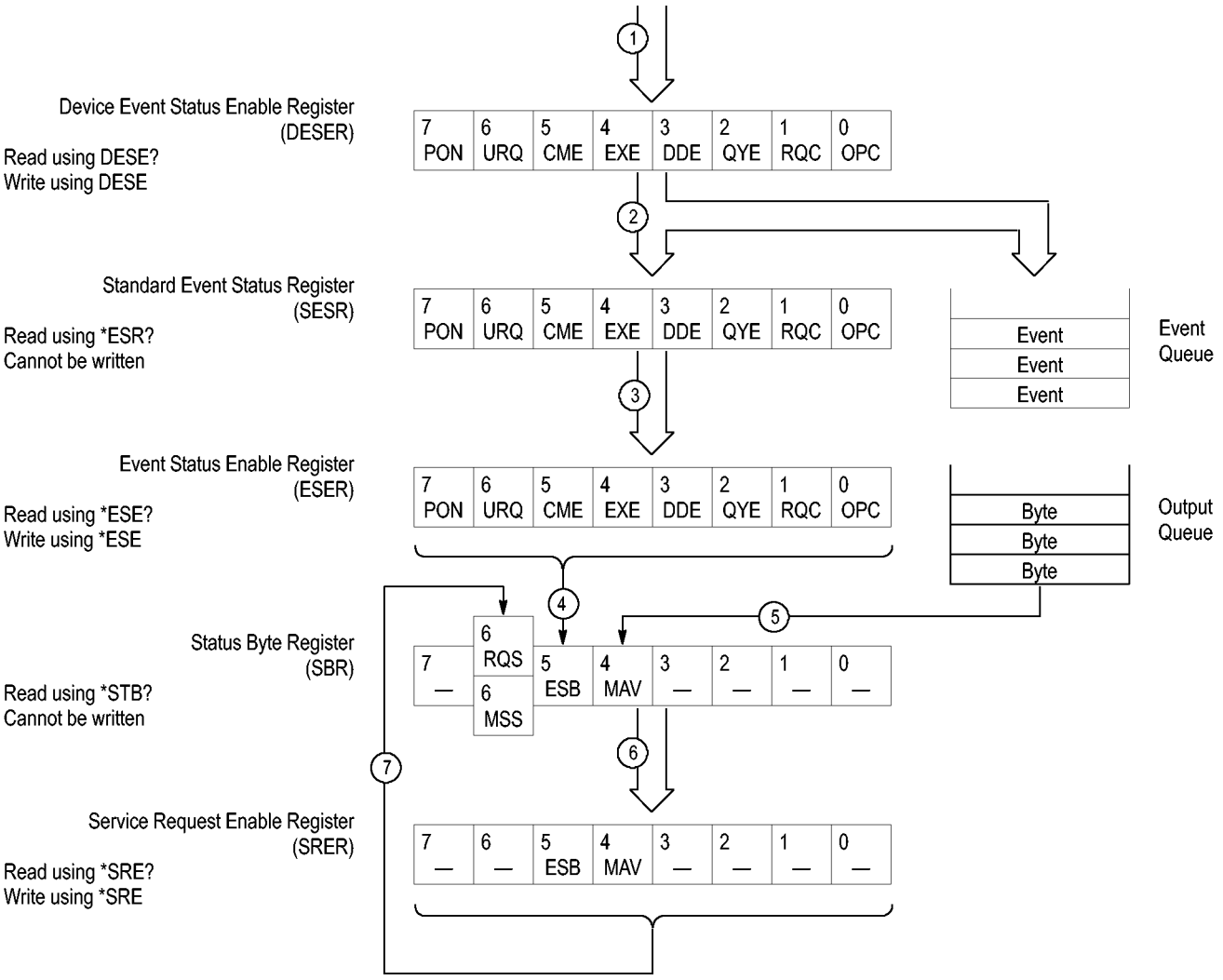


Figure 3-5: Status and event handling process

When an event occurs, a signal is sent to the DESER (1). If that type of event is enabled in the DESER (that is, if the bit for that event type is set to 1), the appropriate bit in the SESR is set to one, and the event is recorded in the Event Queue (2). If the corresponding bit in the ESER is also enabled (3), then the ESB bit in the SBR is set to one (4).

When output is sent to the Output Queue, the MAV bit in the SBR is set to one (5).

When a bit in the SBR is set to one and the corresponding bit in the SRER is enabled (6), the MSS bit in the SBR is set to one and a service request is generated (7).

Synchronization Methods

Overview

Although most commands are completed almost immediately after being received by the instrument, some commands start a process that requires time. For example, once a single sequence acquisition command is executed, depending upon the applied signals and trigger settings, it may take an extended period of time before the acquisition is complete. Rather than remain idle while the operation is in process, the instrument will continue processing other commands. This means that some operations will not be completed in the order that they were sent.

Sometimes the result of an operation depends on the result of an earlier operation. A first operation must complete before the next one is processed. The instrument status and event reporting system is designed to accommodate this process.

The Operation Complete (OPC) bit of the Standard Event Status Register (SESR) can be programmed to indicate when certain instrument operations have completed and, by setting the Event Status Enable Register (ESER) to report OPC in the Event Status Bit (ESB) of the Status Byte Register (SBR) and setting the Service Request Enable Register (SRER) to generate a service request upon a positive transition of the ESB, a service request (SRQ) interrupt can be generated when certain operations complete as described in this section.

The following instrument operations can generate an OPC:

Table 3-3: Commands that can set the OPC bit

Command	Conditions
CONFigure:ARRay:<MeasuringFunction>	
CONFigure:<MeasuringFunction>	
CONFigure:TOTalize[:CONTinuous]	
HCOPy:SDUMp:DATA?	
INITiate	
INITiate:CONTinuous	
*RCL	
SYSTem:PRESet	
SYSTem:SET	

For example, you could use the following command sequence to take a series of fast period measurements on a signal:

```
/** Set up for period measurement **/
FUNCTION “PERIOD 1”
INPUT:LEVEL 0; :AUTO OFF; :COUPLING DC
TRIGGER:COUNT 1000; :ARM COUNT 1
DISPLAY:ENABLE ON
FORMAT ASCII; :TINFORMATION OFF
/** Start measurement**/
INITIATE
```

```
/** Read results **/
FETCH:ARRAY? 1000
```

Measurements require extended processing time. They may not finish before the controller attempts to read the results. To be sure the instrument completes the measurements before the controller reads them, you can synchronize the program.

You can use three commands to synchronize the operation of the instrument with your application program: *WAI, *OPC, and *OPC?

Using the *WAI Command

The *WAI command forces completion of previous commands that generate an OPC message. No commands after the *WAI are processed before the OPC message(s) are generated

The same command sequence using the *WAI command for synchronization looks like this:

```
/** Set up for period measurement **/
FUNCTION “PERIOD 1”
INPUT:LEVEL 0; :AUTO OFF; :COUPLING DC
TRIGGER:COUNT 1000; :ARM COUNT 1
DISPLAY:ENABLE ON
FORMAT ASCII; :TINFORMATION OFF
/** Start measurement**/
INITIATE
/* Wait until the measurements are complete before
reading them*/
*/
*WAI
/** Read results **/
FETCH:ARRAY? 1000
```

The controller can continue to write commands to the input buffer of the instrument, but the commands will not be processed by the instrument until all in-process OPC operations are complete. If the input buffer becomes full, the controller is unable to write commands to the buffer. This can cause a time-out.

Using the *OPC Command

If the corresponding status registers are enabled, the *OPC command sets the OPC bit in the Standard Event Status Register (SESR) when an operation is complete. You achieve synchronization by using this command with either a serial poll or service request handler.

Serial poll method. Enable the OPC bit in the Event Status Enable Register (ESER) using the *ESE command.

When the operation is complete, the OPC bit in the Standard Event Status Register (SESR) is enabled and the Event Status Bit (ESB) in the Status Byte Register is enabled.

The same command sequence using the *OPC command for synchronization with serial polling looks like this:

```
/** Set up for period measurement **/
FUNCTION “PERIOD 1”
INPUT:LEVEL 0; :AUTO OFF; :COUPLING DC
TRIGGER:COUNT 1000; :ARM COUNT 1
DISPLAY:ENABLE ON
FORMAT ASCII; :TINFORMATION OFF
/* Enable the status registers */
*ESE 1
*SRE 0
/** Start measurement**/
INITIATE
/* Wait until the measurements are complete before
reading them*/
*OPC
While serial poll = 0, keep looping
/** Read results **/
FETCH:ARRAY? 1000
```

Service request method. Enable the OPC bit in the Event Status Enable Register (ESER) using the *ESE command.

You can also enable service requests by setting the ESB bit in the Service Request Enable Register (SRER) using the *SRE command. When the operation is complete, the instrument will generate a Service Request.

The same command sequence using the *OPC command for synchronization looks like this

```
/** Set up for period measurement **/
FUNCTION “PERIOD 1”
INPUT:LEVEL 0; :AUTO OFF; :COUPLING DC
TRIGGER:COUNT 1000; :ARM COUNT 1
DISPLAY:ENABLE ON
FORMAT ASCII; :TINFORMATION OFF
/* Enable the status registers */
*ESE 1
*SRE 32
/** Start measurement**/
INITIATE
/* Wait until the measurements are complete before
reading them*/
*OPC
```

The program can now do different tasks such as talk to other devices. The SRQ, when it comes, interrupts those tasks and returns control to this task.

```
/** Read results **/
FETCH:ARRAY? 1000
```

Using the *OPC? Query

The *OPC? query places a 1 in the Output Queue once an operation that generates an OPC message is complete. The *OPC? query does not return until all pending OPC operations have completed. Therefore, your time-out must be set to a time at least as long as the longest expected time for the operations to complete.

The same command sequence using the *OPC? query for synchronization looks like this:

```
/** Set up for period measurement **/
FUNCTION “PERIOD 1”
INPUT:LEVEL 0; :AUTO OFF; :COUPLING DC
TRIGGER:COUNT 1000; :ARM COUNT 1
DISPLAY:ENABLE ON
FORMAT ASCII; :TINFORMATION OFF
/** Start measurement**/
INITIATE
/* Wait until the measurements are complete before
reading them*/
*OPC?
```

Wait for read from Output Queue.

```
/** Read results **/
FETCH:ARRAY? 1000
```

This is the simplest approach. It requires no status handling or loops. However, you must set the controller time-out for longer than the acquisition operation.

Messages

The information contained in the topic tabs above covers all the programming interface messages the instrument generates in response to commands and queries.

For most messages, a secondary message from the instrument returns detail about the cause of the error or the meaning of the message. This message is part of the message string and is separated from the main message by a semicolon.

Each message is the result of an event. Each type of event sets a specific bit in the SESR and is controlled by the equivalent bit in the DESER. Thus, each message is associated with a specific SESR bit. In the message tables, the associated SESR bit is specified in the table title, with exceptions noted with the error message text.

No Event

The following table shows the messages when the system has no events or status to report. These have no associated SESR bit.

Table 3-4: No Event messages

Code	Message
0	No events to report; queue empty
1	No events to report; new events pending *ESR?

Error Messages

You read the error queue with the SYSTem:ERRor? query.

Example SYSTem:ERRor? might return –100, "Command Error"

The query returns the error number followed by the error description.

If more than one error occurred, the query will return the error that occurred first. When you read an error, you will also remove it from the queue. You can read the next error by repeating the query. When you have read all errors, the queue is empty, and the :SYSTem:ERRor? query will return: 0, "No error"

When errors occur and you do not read these errors, the Error Queue may overflow. Then the instrument will overwrite the last error in the queue with:

–350, "Queue overflow"

If more errors occur they will be discarded.

NOTE. *Read more about how to use error reporting in the Introduction to SCPI chapter*

Command Errors

The following table shows the command error messages generated by improper syntax. Check that the command is properly formed and that it follows the rules in the section on command Syntax.

Table 3-5: Command errors

Error Number	Error Description	Description/Explanation/Examples
0	No error	
–100	Command error	This is the generic syntax error for devices that cannot detect more specific errors. This code means that a Command Error defined in IEEE-488.2, 11.5.1.1.4 has occurred.
–101	Invalid character	A syntactic element contains a character which is invalid for that type; for example, a header containing an ampersand, SETUP&. This error might be used in place of errors –114, –121, –141, and perhaps some others.
–102	Syntax error	An unrecognized command or data type was detected; for example, a string was received when the instrument does not accept strings.
	Syntax error; unrecognized data	
–103	Invalid separator	The parser was expecting a separator and detected an illegal character; for example, the semicolon was omitted after a program message unit, *EMC1:CH1:VOLTS5.
–104	Data type error	The parser recognized a data element different than one allowed; for example, numeric or string data was expected but block data was detected.
–105	GET not allowed	A Group Execute Trigger was received within a program message (see IEEE-488.2, 7.7).

Table 3-5: Command errors (cont.)

Error Number	Error Description	Description/Explanation/Examples
–108	Parameter not allowed	More parameters were received than expected for the header; for example, the *EMC common command accepts only one parameter, so receiving *EMC0,,1 is not allowed.
–109	Missing parameter	Fewer parameters were received than required for the header; for example, the *EMC common command requires one parameter, so receiving *EMC is not allowed.
–110	Command header error	An error was detected in the header. This error message is used when the instrument cannot detect the more specific errors described for errors 111–119.
–111	Header separator error	A character that is not a legal header separator was detected while parsing the header; for example, no space followed the header, thus *GMC"MACRO" is an error.
–112	Program mnemonic too long	The header contains more than 12 characters (see IEEE-488.2, 7.6.1.4.1).
–113	Undefined header	The header is syntactically correct, but it is undefined for this specific instrument; for example, *XYZ is not defined for any device.
–114	Header suffix out of range	A non-header character was detected in what the parser expects is a header element.
–120	Numeric data error	This error, and errors –121 through –129, are generated when parsing a data element that appears to be a numeric type. This particular error message is used when the instrument cannot detect a more specific error.
	Numeric data error; overflow from conversion	
	Numeric data error; underflow from conversion	
	Numeric data error; not a number from conversion	
–121	Invalid character in number	An invalid character for the data type being parsed was detected; for example, an alpha in a decimal numeric or a “0" in octal data.
–123	Exponent too large	The magnitude of the exponent was larger than 32000 (see IEEE-488.2, 7.7.2.4.1).
–124	Too many digits	The mantissa of a decimal numeric data element contained more than 255 digits excluding leading zeros (see IEEE-488.2, 7.7.2.4.1).
–128	Numeric data not allowed	A legal numeric data element was received, but the instrument does not accept it in this position for the header.
–130	Suffix error	This error and errors –131 through –139 is generated when parsing a suffix. This particular error message is used when the instrument cannot detect a more specific error.
–131	Invalid suffix	The suffix does not follow the syntax described in IEEE-488.2, 7.7.3.2, or the suffix is inappropriate for this instrument.
–134	Suffix too long	The suffix contained more than 12 characters (see IEEE-488.2, 7.7.3.4).
–138	Suffix not allowed	A suffix was detected after a numeric element that does not allow suffixes.
–140	Character data error	This error and errors 141 through –149 is generated when parsing a character data element. This particular error message is used when the instrument cannot detect a more specific error.

Table 3-5: Command errors (cont.)

Error Number	Error Description	Description/Explanation/Examples
–141	Invalid character data	Either the character data element contains an invalid character or the particular element received is not valid for the header.
–144	Character data too long	The character data element contains more than 12 characters (see IEEE-488.2, 7.7.1.4).
–148	Character data not allowed	A legal character data element was detected where prohibited by the instrument.
–150	String data error	This error and errors –151 through –159 is generated when parsing a string data element. This particular error message is used when the instrument cannot detect a more specific error.
–151	Invalid string data	A string data element was expected, but was invalid for some reason (see IEEE-488.2, 7.7.5.2); for example, an END message was received before the terminal quote character.
	Invalid string data; unexpected end of message	
–158	String data not allowed	A string data element was detected but was not allowed at this point in parsing.
–160	Block data error	This error and errors –161 through –169 is generated when parsing a block data element. This particular error message is used when the instrument cannot detect a more specific error.
–161	Invalid block data	A block data element was expected, but was invalid for some reason (see IEEE-488.2, 7.7.6.2); for example, an END message was received before the length was satisfied.
–168	Block data not allowed	A legal block data element was detected but was not allowed by the instrument at this point in parsing.
–170	Expression data error	This error and errors –171 through –179 is generated when parsing an expression data element. This particular error message is used if the instrument cannot detect a more specific error.
	Expression data error; floating-point underflow	The floating-point operations specified in the expression caused a floating-point error.
	Expression data error; floating-point overflow	
	Expression data error; not a number	
	Expression data error; different number of channels given	Two channel list specifications, giving primary and secondary channels for 2-channel measurements, contained a different number of channels.

Table 3-5: Command errors (cont.)

Error Number	Error Description	Description/Explanation/Examples
–171	Invalid expression data	The expression data element was invalid (see IEEE-488.2, 7.7.7.2); for example, unmatched parentheses or an illegal character were used.
	Invalid expression data; bad mnemonic	A mnemonic data element in the expression was not valid.
	Invalid expression data; illegal element	The expression contained a hexadecimal element not permitted in expressions.
	Invalid expression data; unexpected end of message	End of message occurred before the closing parenthesis.
	Invalid expression data; unrecognized expression type	The expression could not be recognized as either a mathematical expression, a data element list or a channel list.
–178	Expression data not allowed	A legal expression data was detected but was not allowed by the instrument at this point in parsing.
–180	Macro error	This error and errors –181 through –189 is generated when defining a macro or executing a macro. This particular error message is used when the instrument cannot detect a more specific error.
–181	Invalid outside macro definition	Indicates that a macro parameter placeholder ($<number) was detected outside of a macro definition.
–183	Invalid inside macro definition	Indicates that the program message unit sequence, sent with a *DDT or*DMC command, is syntactically invalid (see IEEE-10.7.6.3).
–184	Macro parameter error	Indicates that a command inside the macro definition had the wrong number or type of parameters.
	Macro parameter error; unused parameter	The specified parameter numbers are not continuous; one or more numbers have been skipped.
	Macro parameter error; badly formed placeholder	The '$' sign was not followed by a single digit between 1 and 9.
	Macro parameter error; parameter count mismatch	The macro was invoked with a different number of parameters than used in the definition.

Execution Error

The following table lists the execution errors that are detected during execution of a command.

Table 3-6: Execution errors

Error Number	Error Description	Explanation and examples
–200	Execution error	This is the generic syntax error for devices that cannot detect more specific errors. This code shows that an Execution Error as defined in IEEE-488.2, 11.5.1.1.5 has occurred.
–210	Trigger error	
–211	Trigger ignored	Indicates that a GET, *TRG, or triggering signal was received and recognized by the instrument but was ignored because of instrument timing considerations; for example, the instrument was not ready to respond.

Table 3-6: Execution errors (cont.)

Error Number	Error Description	Explanation and examples
–212	Arm ignored	Indicates that an arming signal was received and recognized by the instrument but was ignored.
–213	Init ignored	Indicates that a request for a measurement initiation was ignored because another measurement was already in progress.
–214	Trigger deadlock	Indicates that the trigger source for the initiation of a measurement is set to GET and subsequent measurement query is received. The measurement cannot be started until a GET is received, but the GET would cause an INTERRUPTED error.
–215	Arm deadlock	Indicates that the arm source for the initiation of a measurement is set to GET and subsequent measurement query is received. The measurement cannot be started until a GET is received, but the GET would cause an INTERRUPTED error.
–220	Parameter error	Indicates that a program-data-element related error occurred. This error message is used when the instrument cannot detect the more specific errors –221 to –229.
–221	Settings conflict	Indicates that a legal program data element was parsed but could not be executed due to the current instrument state (see IEEE-488.2, 6.4.5.3 and 11.5.1.1.5.)
	Settings conflict; PUD memory is protected	
	Settings conflict; invalid combination of channel and function	
–222	Data out of range	Indicates that a legal program data element was parsed but could not be executed because the interpreted value was outside the legal range as defined by the instrument (see IEEE-488.2, 11.5.1.1.5.).
	Data out of range; exponent too large	The expression was too large for the internal floating-point format.
	Data out of range; below minimum	Data below minimum for this function/parameter.
	Data out of range; above maximum	Data above maximum for this function/ parameter.
	Data out of range; (Save/recall memory number)	A number outside 0 to 19 was specified for the save/recall memory.
–223	Too much data	Indicates that a legal program data element of block, expression, or string type received that contained more data than the instrument could handle due to memory or related instrument-specific requirements.
	Too much data; *PUD string too long	
	Too much data; String or block too long	
–224	Illegal parameter value	Used where exact value, from a list of possible values, was expected.
–230	Data corrupt or stale	Possibly invalid data; new reading started but not completed since last access.

Table 3-6: Execution errors (cont.)

Error Number	Error Description	Explanation and examples
–231	Data questionable	
	Data questionable; one or more data elements ignored	One or more data elements sent with a MEASure or CONFigure command was ignored by the instrument.
–240	Hardware error	Indicates that a legal program command or query could not be executed because of a hardware problem in the instrument. Definition of what constitutes a hardware problem is completely device specific. This error message is used when the instrument cannot detect the more specific errors described for errors –241 through –249.
–241	Hardware missing	Indicates that a legal program command or query could not be executed because of missing instrument hardware; for example, an option was not installed. Definition of what constitutes missing hardware is completely device specific.
	Hardware missing; (prescaler)"	
–254	Media full	Indicates that a legal program command or query could not be executed because the media was full; for example, there is no room on the disk. The definition of what constitutes a full media is device specific.
–258	Media protected	Indicates that a legal program command or query could not be executed because the media was protected; for example, the write-protect tab on a disk was present. The definition of what constitutes protected media is device specific.
–260	Expression error	Indicates that an expression-program data-element- related error occurred. This error message is used when the instrument cannot detect the more specific errors described for errors –261 through –269.
–261	Math error in expression	Indicates that a syntactically correct expression program data element could not be executed due to a math error; for example, a divide-by-zero was attempted.
–270	Macro error	Indicates that a macro-related execution error occurred. This error message is used when the instrument cannot detect the more specific error described for errors –271 through –279.
	Macro error; out of name space	No room for any more macro names.
	Macro error; out of definition space	No room for this macro definition.
–271	Macro syntax error	Indicates that a syntactically correct macro program data sequence, according to IEEE-488.2 10.7.2, could not be executed due to a syntax error within the macro definition (see IEEE-488.2, 10.7.6.3)
–272	Macro execution error	Indicates that a syntactically correct macro program data sequence could not be executed due to some error in the macro definition (see IEEE-488.2, 10.7.6.3)
–273	Illegal macro label	Indicates that the macro label defined in the *DMC command was a legal string syntax, but could not be accepted by the instrument (see IEEE-488.2, 10.7.3 and 10.7.6.2); for example, the label was too long, the same as a common command header, or contained invalid header syntax.
–274	Macro parameter error	Indicates that the macro definition improperly used a macro parameter place holder (see IEEE-488.2, 10.7.3).

Table 3-6: Execution errors (cont.)

Error Number	Error Description	Explanation and examples
–275	Macro definition too long	Indicates that a syntactically correct macro program data sequence could not be executed because the string or block contents were too long for the instrument to handle (see IEEE-488.2, 10.7.6.1).
–276	Macro recursion error	Indicates that a syntactically correct macro program data sequence could not be executed because the instrument found it to be recursive (see IEEE-488.2, 10.7.6.6).
–277	Macro redefinition not allowed	Indicates that a syntactically correct macro label in the *DMC command could not be executed because the macro label was already defined (see IEEE-488.2, 10.7.6.4).
–278	Macro header not found	Indicates that a syntactically correct macro label in the *GMC? query could not be executed because the header was not previously defined.

Device Errors

The following table lists the device errors that can occur during instrument operation. These errors may indicate that the instrument needs repair.

Table 3-7: Standardized device-specific errors

Error Number	Error Description	Explanation and examples
–300	Device specific error	This code indicates only that a Device-Dependent Error as defined in IEEE-488.2, 11.5.1.1.6 has occurred. Contact your local service center.
–311	Memory error	Indicates that an error was detected in the instrument's memory. Contact your local service center.
–312	PUD memory lost	Indicates that the protected user data saved by the *PUD command was lost. Contact your local service center.
–314	Save/recall memory lost	Indicates that the nonvolatile calibration data used by the *SAV? command was lost. Contact your local service center.
–330	Self-test failed	Contact your local service center.
–350	Queue overflow	A specific code entered into the queue in lieu of the code that caused the error. This code indicates that there is no room in the queue and an error occurred but was not recorded.

System Query Errors

The following table lists the system event messages. These messages are generated whenever certain system conditions occur.

Table 3-8: Query errors

Error Number	Error Description	Explanation and examples
–400	Query error	This code indicates that a Query Error as defined in IEEE-488.2, 11.5.1.1.7 and 6.3 has occurred.

Table 3-8: Query errors (cont.)

Error Number	Error Description	Explanation and examples
–410	Query INTERRUPTED	Indicates that a condition causing an INTERRUPTED Query error occurred (see IEEE-488.2, 6.3.2.3); for example, a query was followed by DAB or GET before a response was completely sent.
	Query INTERRUPTED; in send state	The additional information indicates the IEEE-488.2 message exchange state where the error occurred.
	Query INTERRUPTED; in query state	
	Query INTERRUPTED; in response state	
–420	Query UNTERMINATED	Indicates that a condition causing an UNTERMINATED Query error occurred (see IEEE-488.2, 6.3.2.2); for example, the instrument was addressed to talk and an incomplete program message was received.
	Query UNTERMINATED; in idle state	The additional information indicates the IEEE-488.2 message exchange state where the error occurred
	Query UNTERMINATED; in read state	
	Query UNTERMINATED; in send state	
–430	Query DEADLOCKED	Indicates that a condition causing an DEADLOCKED Query error occurred (see IEEE-488.2, 6.3.1.7); for example, both input buffer and output buffer are full and the instrument cannot continue.
–440	Query UNTERMINATED after indefinite response	Indicates that a query was received in the same program message after an query requesting an indefinite response was executed (see IEEE-488.2, 6.5.7.5.7.)

Table 3-9: Device-specific errors

Error Number	Error Description	Explanation and examples
(1)100	Device operation gave floating-point underflow	A floating-point error occurred during a instrument operation.
(1)101	Device operation gave floating-point overflow	A floating-point error occurred during a instrument operation.
(1)102	Device operation gave ‘not a number’	A floating-point error occurred during a instrument operation.
(1)110	Invalid measurement function	The instrument was requested to set a measurement function it could not make.
(1)120	Save/recall memory protected	An attempt was made to write in a protected memory.

Table 3-9: Device-specific errors (cont.)

Error Number	Error Description	Explanation and examples
(1)130	Unsupported command	Indicates a mismatch between bus and instrument capabilities.
(1)131	Unsupported boolean command	
(1)132	Unsupported decimal command	
(1)133	Unsupported enumerated command	
(1)134	Unsupported auto command	
(1)135	Unsupported single shot command	
(1)136	Command queue full; last command discarded	The instrument has an internal command queue with room for about 100 commands. Many commands arrived fast without any intervening query.
(1)137	Inappropriate suffix unit	A suffix unit was not appropriate for the command. Recognized units are Hz (Hertz), s (seconds), ohms (Ω) and V (Volt).
(1)138	Unexpected command to device execution	A command reached instrument execution which should have been handled by the bus.
(1)139	Unexpected query to device execution	A query reached instrument execution which should have been handled by the bus.
(1)150	Bad math expression format	Only a fixed, specific math expression is recognized by the instrument, and this was not it.
(1)160	Measurement broken off	A new bus command caused a running measurement to be broken off.
(1)170	Instrument set to default	An internal setting inconsistency caused the instrument to go to default setting.
(1)190	Error during calibration	An error has occurred during calibration of the instrument.
(1)191	Hysteresis calibration failed	The input hysteresis values found by the calibration routine was out of range. Did you remember to remove the input signal?
(1)200	Message exchange error	An error occurred in the message exchange handler (generic error).
(1)201	Reset during bus input	The instrument was waiting for more bus input, but the waiting was broken by the operator.
(1)202	Reset during bus output	The instrument was waiting for more bus output to be read, but the waiting was broken by the operator.
(1)203	Bad message exchange control state	An internal error in the message exchange handler.
(1)204	Unexpected reason for GPIB interrupt	A spurious GPIB interrupt occurred, not conforming to any valid reason such as an incoming byte or address change.
(1)205	No listener on bus when trying to respond	This error is generated when the instrument is an active talker, and tries to send a byte on the bus, but there are no active listeners. (This may occur if the controller issues the device talker address before its own listener address, which some PC controller cards are known to do)
(1)210	Mnemonic table error	An abnormal condition occurred in connection with the mnemonics tables (generic error).

Table 3-9: Device-specific errors (cont.)

Error Number	Error Description	Explanation and examples
(1)211	Wrong macro table checksum found	The macro definitions have been corrupted (could be loss of memory).
(1)212	Wrong hash table checksum found	The hash table is corrupted. Could be bad memory chips or address logic. Contact your local service center.
(1)213	RAM failure to hold information (hash table)	The memory did not retain information written to it. Could be bad memory chips or address logic. Contact your local service center.
(1)214	Hash table overflow	The hash table was too small to hold all mnemonics. Ordinarily indicates a failure to read (RAM or ROM) correctly. Contact your local service center.
(1)220	Parser error	Generic error in the parser.
(1)221	Illegal parser call	The parser was called when it should not be active.
(1)222	Unrecognized input character	A character not in the valid IEEE488.2 character set was part of a command.
(1)223	Internal parser error	The parser reached an unexpected internal state.
(1)230	Response formatter error	Generic error in the response formatter.
(1)231	Bad response formatter call	The response formatter was called when it should not be active.
(1)232	Bad response formatter call (eom)	The response formatter was called to output an end of message, when it should not be active.
(1)233	Invalid function code to response formatter	The response formatter was requested to output data for an unrecognized function.
(1)234	Invalid header type to response formatter	The response formatter was called with bad data for the response header (normally empty)
(1)235	Invalid data type to response formatter	The response formatter was called with bad data for the response data.
(1)240	Unrecognized error number in error queue	An error number was found in the error queue for which no matching error information was found.

Programming Examples

Programming Examples

Introduction

The program examples in this chapter are written in standard 'C' extended with a dedicated library for the National AT-GPIB/TNT controller board.

The programs can be run on PCs using Microsoft Windows NT and later operating systems.

Even if you use other platforms for your applications, these examples provide a good insight into how to program the instrument.

NOTE. *To be able to run these programs without modification, the address of your instrument must be set to 10.*

Five examples are included:

- Example 1: Individual Measurements
- Example 2: Block Measurements
- Example 3: Fast Measurements
- Example 4: USB Communication
- Example 5: Continuous Measurements

Individual Measurements (Example #1)

This sample program takes individual measurements on the instrument. Written for National AT-GPIB/TNT for Windows NT and later.

```
/*
**
Sample program to perform individual measurements on the instrument. Written
for National AT-GPIB/TNT for Windows NT and later.
**
*/
#include <windows.h>
#include <stdio.h>
#include <time.h>
#include “decl-32.h”
void ibwrite(int instrument, const char *string);
void sleep (long mswait);
void main() {
    int address = 10;
    int i, instrument;/* file descriptor for instrument */
    char reading[50];
    char buf[100];
   printf (“Connecting to the instrument on address %d using National Instruments
GPIB card.\n”, address);
    if ((instrument = ibdev(0, address, 0, T10s, 1, 0)) < 0) {
        printf(“Could not connect to instrument”);
         exit(1);
    }
    ibclr(instrument);
    do {
        ibwrite(instrument, “syst:err?”);
        ibrd(instrument, buf, 100L); buf[ibcnt]=0;
        printf(“Errors before start: %s\n”, buf);
    } while (atoi(buf)!=0);
    ibwrite(instrument, “*idn?”);
    ibrd(instrument, buf, 100L); buf[ibcnt]=0;
    printf(“instrument identification string: %s\n”, buf);
    printf(“Setup\n”);
// Reset instrument to known state
    ibwrite(instrument, “*rst;*cls”);
// Setup for pulse width measurement
    ibwrite(instrument, “CONF:PWID (@1)”);
// Some settings...
    ibwrite(instrument, “AVER:STAT OFF;:ACQ:APER MIN”);
    ibwrite(instrument, “INP:LEV:AUTO OFF; :INP:LEV 0");
    ibwrite(instrument, ”FORMAT:TINF ON;:FORMAT ASCII");
// Check that setup was OK, all commands correctly spelled etc
    ibwrite(instrument, “syst:err?”);
    ibrd(instrument, buf, 100L); buf[ibcnt]=0;
```

```
    printf("Setup error:  %s\n", buf);
// Measure 20 samples
    for (i=0; i<20; i++) {
        ibwrite(instrument, "READ?");
        ibrd(instrument, reading, 49L); reading[ibcnt]=0;
        printf("Result %d:%s", i, reading);
    }
    do {
        ibwrite(instrument, "syst:err?");
        ibrd(instrument, buf, 100L); buf[ibcnt]=0;
        printf("End error:  %s\n", buf);
    } while (atoi(buf)!=0);
    ibonl(instrument, 0);
}
/********************
* Support functions *
********************/
void ibwrite(int instrument, const char *string) {
    ibwrt(instrument, (char*) string, strlen(string));
}
void sleep (long mswait) {
    time_t EndWait = clock() + mswait * (CLOCKS_PER_SEC/1000);
    while (clock() < EndWait);
}
```

Block Measurements (Example #2)

Sample program to perform fast measurements on the instrument using block measurements. Written for National AT-GPIB/TNT for Windows NT and later.

```
/*
**
** Sample program to perform fast measurements on the instrument
** using block measurements
**
** Written for National AT-GPIB/TNT for Windows NT and later
*/
#include <windows.h>
#include <stdio.h>
#include <time.h>
#include “decl-32.h”
void ibwrite(int instrument, const char *string);
void sleep (long mswait);
time_t StartMain, Start, Stop, StopMain;
void main() {
    int address = 10;
    int i, j, instrument; /* file descriptor for instrument */
    charbigbuf[30000], *pbuf; charbuf[100];
    char Status;
   printf (“Connecting to the instrument on address %d using National Instruments
GPIB card.\n”, address);
    if ((instrument = ibdev(0, address, 0, T10s, 1, 0)) < 0) {
        printf(“Could not connect to instrument”);
        exit(1);
    }
    ibclr(instrument);
    do {
        ibwrite(instrument, “syst:err?”);
        ibrd(instrument, buf, 100L); buf[ibcnt]=0;
        printf(“Errors before start: %s\n”, buf);
    } while (atoi(buf)!=0);
    ibwrite(instrument, “*idn?”);
    ibrd(instrument, buf, 100L); buf[ibcnt]=0; printf(“instrument identification
string: %s\n”, buf);
    printf(“Setup\n”);
// Reset instrument to known state
    ibwrite(instrument, “*rst;*cls”);
// Setup for period measurement
    ibwrite(instrument, “FUNC ‘PER 1’”);
// Some settings...
    ibwrite(instrument, “INP:LEV:AUTO OFF;:INP:LEV 0;COUP DC”);
    ibwrite(instrument, “TRIG:COUNT 1000;:ARM:COUNT 1");
ibwrite(instrument, ”DISP:ENAB ON"); ibwrite(instrument, “FORMAT
ASCII;:FORMAT:TINF OFF”);
```

```
    ibwrite(instrument, “*ESE 1;*SRE 32");
// On the safe side: Check that setup was OK, all commands correctly spelled etc
    ibwrite(instrument, ”syst:err?");
    ibrd(instrument, buf, 100L); buf[ibcnt]=0; printf(“Setup error: %s\n”, buf);
// Measure 1000 samples
    Start = clock();
    ibwrite(instrument, “INIT;*OPC”);
// Wait for completion
    ibwait(instrument, RQS);
/* Read status and event registers to clear them */
    ibrsp(instrument, &Status);
    ibwrite(instrument, “*ESR?”);
    ibrd(instrument, buf, 100L);
    ibwrite(instrument, “FETC:ARR? 1000");
    ibrd(instrument, bigbuf, 30000L);
    if (ibcnt >0) {
        pbuf = bigbuf;
        for (i=0; i<1000; i++) {
           for (j=0; pbuf[j]!=’,’ && pbuf[j]!=’\0’; j++);
            pbuf[j]=’\0’;
           if (i%50 == 0) printf(”Result %d: %s\n", i, pbuf);
            pbuf+=j+1;
        }
    }
    Stop = clock();
    printf (“Block measurement: %d samples/s\n”, 10000 * 1000 / (Stop - Start));
    do {
        ibwrite(instrument, “syst:err?”);
        ibrd(instrument, buf, 100L); buf[ibcnt]=0;
        printf(“End error: %s\n”, buf);
    } while (atoi(buf)!=0);
    ibonl(instrument, 0);
}
/********************
* Support functions *
********************/
void ibwrite(int instrument, const char *string) {
    ibwrt(instrument, (char*) string, strlen(string));
}
void sleep (long mswait) {
    time_t EndWait = clock() + mswait * (CLOCKS_PER_SEC/1000);
    while (clock() < EndWait);
}
```

Fast Measurements (Example #3)

Sample program to perform fast measurements on the instrument using GET, DISP:ENAB OFF and FORMAT REAL. Written for National AT-GPIB/TNT for Windows NT and later.

```
/*
**
** Sample program to perform fast measurements on the instrument
** using GET, DISP:ENAB OFF and FORMAT REAL
**
** Written for National AT-GPIB/TNT for Windows NT and later
*/
#include <windows.h>
#include <stdio.h>
#include <time.h>
#include “decl-32.h”
void ibwrite(int instrument, const char *string);
void sleep (long mswait);
time_t StartMain, Start, Stop, StopMain;
typedef union {
    double d;
    char c[8];
} r2d;
void main() {
    int address = 10;
    int i, j, instrument; /* file descriptor for instrument */
    char reading[30];
    char buf[100];
    r2d Result;
   printf (“Connecting to the instrument on address %d using National Instruments
GPIB card.\n”, address);
    if ((instrument = ibdev(0, address, 0, T10s, 1, 0)) < 0) {
        printf(“Could not connect to instrument”);
        exit(1);
    }
    sleep(100);
    ibclr(instrument);
    sleep(100);
    ibwrite(instrument, “*idn?”);
    ibrd(instrument, buf, 100L); buf[ibcnt]=0;
    printf(“instrument identification string: %s\n”, buf);
    printf(“Setup\n”);
    if ((instrument = ibdev(0, address, 0, T3s, 1, 0)) < 0) {
        printf(“Could not connect to instrument”);
        exit(1);
    }
// Reset instrument to known state
    ibwrite(instrument, “*rst;*cls”);
```

```
    ibwrite(instrument, “*ESE 0; *SRE 0");
// Setup for frequency measurement
    ibwrite(instrument, “FUNC ‘per 1’”);
// Some settings...
    ibwrite(instrument, “INP:LEV:AUTO OFF;:INP:LEV .5;:inp:coup dc”);
    ibwrite(instrument, “TRIG:COUNT 1;:ARM:COUNT 1");
    ibwrite(instrument, ”ACQ:APER 1e-7");
    ibwrite(instrument, “DISP:ENAB OFF”);
// Disable display to get maximum speed
    ibwrite(instrument, “FORMAT REAL;:FORMAT:TINF OFF”);
// Floating point output, no timestamps
    ibwrite(instrument, “FORMAT:BORDER swap”);
// Intel byte order on results
    ibwrite(instrument, “ARM:LAY2:SOUR BUS;:INIT:CONT ON”);
// Bus arming
    sleep(100);
// On the safe side: Check that setup was OK, all commands correctly spelled etc
    do {
        ibwrite(instrument, “syst:err?”);
        ibrd(instrument, buf, 100L); buf[ibcnt]=0;
        printf(“Setup error: %s\n”, buf);
    } while (atoi(buf)!=0);
    printf(“Start\n”);
// Measure 1000 samples
    Start = clock();
    for (i=0; i<1000; i++) {
        ibtrg(instrument);

// Generate GET signal
        ibrd(instrument, reading, 29L);
        for (j=0; j<8; j++) {
            Result.c[j] = reading[3+j];
        }
       if (i%50 == 0) printf(“Result %d: %e\n”, i, Result.d);
    }
    Stop = clock();
     printf (“Total time %d ms (%f samples /s)\n”, Stop- Start,
(double)1000.0/(Stop-Start)*1000);
    ibwrite(instrument, “DISP:ENAB ON”);
    do {
        ibwrite(instrument, “syst:err?”);
        ibrd(instrument, buf, 100L); buf[ibcnt]=0;
        printf(“End error: %s\n”, buf);
    } while (atoi(buf)!=0);
    ibonl(instrument, 0);
}
/********************
* Support functions *
```

```
*******************/
void ibwrite(int instrument, const char *string) {
    ibwrt(instrument, string, strlen(string));
}
void sleep (long mswait) {
    time_t EndWait = clock() + mswait * (CLOCKS_PER_SEC/1000);
    while (clock() < EndWait);
}
```

USB Communication (Example #4)

```
#include “stdio.h”
#include “visa.h”
#include <time.h>
#define MAX_CNT 200
void Sleep( clock_t Wait );
int main(void) {
    ViStatus Status; // For checking errors
    ViUInt32 RetCount; // Return count from string I/O
    ViChar Buffer[MAX_CNT]; // Buffer for string I/O
    ViFindList fList;
    ViChar Desc[VI_FIND_BUFLEN];
    ViUInt32 numInstrs;
    ViSession defaultRM, Instr;
    int i = 0;
// Begin by initializing the system
    Status = viOpenDefaultRM(&defaultRM);
    if (Status < VI_SUCCESS) {
        printf (“Failed to initialise NI-VISA system.\n”);
        return -1;
    }
// Look for instrument
    Status = viFindRsrc(defaultRM,
        “USB?*INSTR{VI_ATTR_MANF_ID==0x0699}”,
        &fList, &numInstrs, Desc);
    if (Status < VI_SUCCESS) {
        printf (“No matching instruments found.\n”);
        return -1;
    }
// Open communication with GPIB Device
    Status = viOpen(defaultRM, Desc, VI_NULL, VI_NULL, &Instr);
    if (Status < VI_SUCCESS) {
        printf (“Cannot communicate with instrument.\n”);
        return -1;
    }
// Set the timeout for message-based communication
    Status = viSetAttribute(Instr, VI_ATTR_TMO_VALUE, 1000);
// Ask the device for identification
    Status = viWrite(Instr, “*IDN?\n”, 6, &RetCount);
    Status = viRead(Instr, Buffer, MAX_CNT, &RetCount);
    Buffer[RetCount]=0;
    printf(“%s\n”,Buffer);
    Status = viWrite(Instr, “INIT:CONT OFF;:func ‘per’\n”, 25,
        &RetCount);
    while( i++<10){
        Status = viWrite(Instr, “init;fetc?\n”, 11, &RetCount);
        if (Status != VI_SUCCESS) {
```

```
            printf("Write: status = %x, i = %d\n", Status, i);
/* Close down the system */
            Status = viClose(Instr);
            Status = viClose(defaultRM);
            return 0;
        }
        Sleep(200);
        Status = viRead(Instr, Buffer, MAX_CNT, &RetCount);
        if (Status != VI_SUCCESS) {
            printf("Read: status = %x, i = %d\n", Status, i);
/* Close down the system */
            Status = viClose(Instr);
            Status = viClose(defaultRM);
            return 0;
        }
        Buffer[RetCount]=0;
        printf("%s\n",Buffer);
        Sleep(25);
    }
    Status = viWrite(Instr, "syst:err?\n", 10, &RetCount);
    Sleep(25);
    Status = viRead(Instr, Buffer, MAX_CNT, &RetCount);
    Buffer[RetCount]=0;
    printf("%s\n",Buffer);
/* Close down the system */
    Status = viClose(Instr);
    Status = viClose(defaultRM);
    return 0;
}
void Sleep( clock_t Wait ) {
    clock_t Goal;
    Goal = Wait + clock();
    while( Goal > clock() );
}
```

Continuous Measurements (Example #5)

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <float.h>
#include <math.h>
#include <assert.h>
#include "visa.h"
// Write a null terminated string (ie, no binary data) to the
// instrument.
unsigned WriteDevice(ViSession Instr, const char *Str, int Line) {
    ViStatus Status;
    int Length;
    ViUInt32 RetLength;
    assert(Str != NULL);
    Length = strlen(Str);
   Status = viWrite(Instr, (unsigned char *)Str, Length, &RetLength);
    if (Status != VI_SUCCESS) {
       fprintf(stderr, "Write error: %x at line %d\n", (unsigned)Status, Line);
        return((unsigned)Status);
    }
    assert(Length == (int)RetLength);
    return((unsigned)Status);
}
// Read data (may be binary) into the buffer.
unsigned ReadDevice(ViSession Instr, char *Buf, int BufLength, ViUInt32
*pActualLength, int Line) {
    ViStatus Status;
    assert(Buf != NULL);
    assert(BufLength > 0);
    assert(pActualLength != NULL);
   Status = viRead(Instr, (unsigned char *)Buf, BufLength, pActualLength);
    if (Status != VI_SUCCESS) {
       fprintf(stderr, "Read error: %x at line %d\n", (unsigned)Status, Line);
    }
    return((unsigned)Status);
}
#define WriteDev(Str) WriteDevice(Instr, Str, __LINE__)
#define ReadDev(Buf, BufLength, pActualLength) ReadDevice(Instr, Buf,
BufLength, pActualLength, __LINE__)
ViSession defaultRM, Instr;
void Quit() {
    (void)viClose(Instr);
    (void)viClose(defaultRM);
    _exit(0);
}
```

```
void QuitMsg(char *Str) {
    fprintf(stderr, Str);
    Quit();
}
void ReportAndQuit() {
    char Buf[100];
    ViUInt32 ReadLength;
    int Error;
// Break the measurement.
    (void)WriteDev("abort");
// Check if everything seems to have worked out OK.
    printf("Error queue:\n");
    do {
       if (WriteDev("syst:err?") != VI_SUCCESS) {
          QuitMsg("Failed to query error queue\n");
        }
       if (ReadDev(Buf, 100, &ReadLength) != VI_SUCCESS) {
          QuitMsg("Failed to read error message\n");
        }
        Buf[ReadLength] = 0; // Null terminate.
        if (sscanf(Buf, "%d", &Error) != 1) {
          QuitMsg("Failed to scan error status number\n");
        }
        printf(Buf);
    } while (Error != 0);
// Restore the instrument to a more front panel friendly
// state.
    (void)WriteDev("syst:pres");
    (void)viClose(Instr);
    (void)viClose(defaultRM);
    _exit(0);
}
// command line arguments
struct CmdArgs {
    bool bUSB; // GPIB if false
    unsigned int nAddr; // GPIB address. Not used for USB
    double Pacing;
    char Func[64]; // measurement function
    bool bPeriod; // is Meas Func one of Period functions
// or one of Freq functions
    double RefVal, Delta;// reference value and acceptable error
// (used to check meas results)
    double RefFreq; // reference freq
};
// check if string is one of the given set. returns
// the number of matched string or -1 if no matches are found
inline int CheckStr(char const *s, int nSLen, char const *Set[], int nSetSize) {
```

```
    for ( int i = 0; i < nSetSize; i++ )
    if ( 0 == strncmp(s, Set[i], nSLen) && nSLen == strlen(Set[i]) ) {
        return i;
    }
    return -1;
}
// Parse command line.  Format:
// <Executable> USB|GPIB[:<Address>] [<Pacing>] [<Meas Func>]
// [<RefFreq>] [<Delta>]
bool ParseCmdArgs(CmdArgs *pArgs, int argc, char* argv[]) {
static char const *StrInterfaces[] = { "USB", "GPIB" };
static char const *StrMeasFuncs[] =
    {
    "PER",
    "PER:BTB",
    "FREQ:BTB" // <-nFirstFreq
    };
static int const nFirstFreq = 2;
static int const nMeasFuncs = sizeof(StrMeasFuncs) / sizeof(StrMeasFuncs[0]);
// defaults
static int const DefAddr = 10;
static double const DefPacing = 100e-6; // s
static int const DefMeasFunc = 2;
static double const DefRefFreq = 10e6; // Hz
static double const DefDelta = 10e5; // Hz
// assign some defaults
pArgs->bUSB = true;
pArgs->nAddr = DefAddr;
pArgs->Pacing = DefPacing;
strcpy(pArgs->Func, StrMeasFuncs[DefMeasFunc]);
pArgs->bPeriod = (DefMeasFunc < nFirstFreq);
pArgs->RefFreq = DefRefFreq;
pArgs->Delta = DefDelta;
// parse command line
bool bError = (argc < 2); // at least interface should be
// specified
for ( int i = 1, nArg = i; ! bError && i < argc; i++, nArg++ ) {
    char const *s = argv[i];
    switch (nArg)
    {
        case 1:  { // interface
// find ':' delimiter
            int j = 0;
            for ( j = 0; 0 != s[j] && ':' != s[j]; j++ );
// check interface and read address (if any)
            int const nInterface = CheckStr(s, j, StrInterfaces, 2);
            if ( nInterface < 0 ) { bError = true; break; }
```

```
            pArgs->bUSB = (0 == nInterface);
            sscanf(s + j, ":%d", &(pArgs->nAddr));
             break;
          }
          case 2:  { // Pacing
            if ( 1 == sscanf(s, "%lf", &(pArgs->Pacing)) ) {
               if ( pArgs->Pacing < 50e-6 ) pArgs->Pacing = 50e-6;
                break;
             }
// this is not pacing.  fallthrough to next arg
             nArg++;
          }
          case 3:  { // meas func
// copy Meas Func
             int n = strlen(s);
            if ( n >= sizeof(pArgs->Func) / sizeof(pArgs->Func[0]) ) {
// func is too long
                bError = true;
                break;
             }
            strncpy(pArgs->Func, s, n);
            pArgs->Func[n] = 0;
// determine if it is period (and if it is valid
// at all)
            n = CheckStr(s, n, StrMeasFuncs, nMeasFuncs);
             if ( n >= 0 ) {
               pArgs->bPeriod = (n < nFirstFreq);
                break;
             }
// not a function specification.  fallthrough
             nArg++;
          }
          case 4:  { // Reference Value
            if ( 1 != sscanf(s, "%lf", &(pArgs->RefFreq)) ) {
                bError = true;
             }

             break;
          }
          case 5:  { // Delta
            if ( 1 != sscanf(s, "%lf", &(pArgs->Delta)) ) {

                              bError = true;

                           }
             break;
          }
          default:  {
```

```
            bError = true;
        }
    }
// show the usage string in a case of error
    if ( bError ) {
        fprintf(stderr, "Usage:\n"
          "%s USB|GPIB[:<Address>] [<Pacing>] [<Meas Func>] [<Ref Freq>]
[<Delta>]\n\n"
           "Parameters description:\n"
          " USB|GPIB - selects particular bus interface,\n"
          " <Address> - (optional) instrument GPIB address\n"
           " (%d if omitted)\n"
         " <Pacing> - (optional) pacing time between measurements\n"
          " (%lg s if omitted)\n"
         " <Meas Func> - (optional) meas func to be used. Possible values:\n",
        argv[0], DefAddr, DefPacing);
        for ( int i = 0; i < nMeasFuncs; i++ ) {
            fprintf(stderr,
                " %s\n",
               StrMeasFuncs[i]);
            fprintf(stderr,
               " (%s if omitted)\n",
              StrMeasFuncs[DefMeasFunc]);
            fprintf(stderr,
              " <Ref Freq> - (optional) frequency to be measured\n"
               " (%lg Hz if omitted)\n"
              " <Delta> - (optional) acceptable frequency error\n"
               " (%lg Hz if omitted)\n",
               DefRefFreq, DefDelta);
            return false;
        }
// convert RefVal and Delta for Period
        pArgs->RefVal = pArgs->RefFreq;
        if ( pArgs->bPeriod ) {
           pArgs->RefVal = 1 / pArgs->RefVal;
          pArgs->Delta *= pArgs->RefVal * pArgs->RefVal;
        }
        return true;
    }
}
// check that measurement is correct
inline bool CheckMeas(double Val, CmdArgs const &Args) {
    return (_isnan(Val) ||
       Val < Args.RefVal - Args.Delta || Val > Args.RefVal + Args.Delta);
}
// check for buttonpress and exit if any
inline void CheckUserCancel() {
```

```
    if ( kbhit() ) {
        if ( 0 == getch() ) getch();
        QuitMsg("\nCancelled by the user...\n");
    }
}
// Create a buffer that should fit 10000 samples in FORMat
// PACKed.
#define BUFSIZE 170000
char Buffer[BUFSIZE];
int main(int argc, char* argv[]) {
    ViStatus Status;
    ViUInt32 ReadLength;
    ViFindList fList;
    ViChar Desc[VI_FIND_BUFLEN];
    ViUInt32 numInstrs;
    double Val;
    bool Failed;
    int Samples, Digits, i;
    __int64 TSVal, PrevTSVal, Count;
    char *pBuf, Command[200];
// Begin by initializing the system
    Status = viOpenDefaultRM(&defaultRM);
    if (Status != VI_SUCCESS) {
        fprintf(stderr, "Initialization failed\n");
        return -1;
    }
// Parse cmdline
    CmdArgs Args;
    if ( ! ParseCmdArgs(&Args, argc, argv) ) {
        viClose(defaultRM);
        return -1;
    }
// Find the instrument
    if ( Args.bUSB ) {
// Look on USB/GPIB for counter model FCA3020
// code 0x3020.
// For this sample program we'll just pick the first
// found, if any.
        sprintf(Command, "USB?*INSTR{VI_ATTR_MANF_ID==0x0699 &&
VI_ATTR_MODEL_CODE==0x3020}");
    }
    else { // GPIB
        sprintf(Command, "GPIB::%d::INSTR", Args.nAddr);
    }
    Status = viFindRsrc(defaultRM, Command, &fList, &numInstrs, Desc);
    if (Status != VI_SUCCESS) {
        fprintf(stderr, "Did not find instrument\n");
```

```
        viClose(defaultRM);
        return(-1);
    }
// Open communication with the device.
   if (viOpen(defaultRM, Desc, VI_NULL, VI_NULL, &Instr) != VI_SUCCESS)
{
       QuitMsg("Could not open connection to the instrument\n");
    }
// Set short timeout for message-based communication (1 s)
   if (viSetAttribute(Instr, VI_ATTR_TMO_VALUE, 1000) != VI_SUCCESS){
       QuitMsg("Failed to set timeout\n");
    }
// Clear the instrument
    if (viClear(Instr) != VI_SUCCESS) {
       QuitMsg("Could not clear the instrument\n");
    }
// Check IDN.
    if (WriteDev("*idn?") != VI_SUCCESS) Quit();
   if (ReadDev(Buffer, BUFSIZE, &ReadLength) != VI_SUCCESS) Quit();
    Buffer[ReadLength] = 0; // Null terminate.
    printf("%s\n", Buffer);
// Initialize the instrument.
    printf("Testing %s with pacing: %g\n", Args.Func, Args.Pacing);
    printf("Push any button to cancel.\n");
    fflush(stdout);
   if (WriteDev("*cls;*rst") != VI_SUCCESS) Quit();
   if (WriteDev("*ese 0;*sre 0") != VI_SUCCESS) Quit();
// Set Meas Func
    sprintf(Command, "CONF:%s", Args.Func);
   if (WriteDev(Command) != VI_SUCCESS) Quit();
// Do a measurement to check if all is set up OK.
   if (WriteDev("inp:lev:auto off;:inp:lev 0;:form:bord swap") != VI_SUCCESS)
Quit();
   if (WriteDev("form asc;:form:tinf on") != VI_SUCCESS) Quit();
   if (WriteDev("read?") != VI_SUCCESS) Quit();
   if (ReadDev(Buffer, BUFSIZE, &ReadLength) != VI_SUCCESS) Quit();
    Buffer[ReadLength] = 0; // Null terminate.
    if (sscanf(Buffer, "%lf", &Val) != 1) {
       QuitMsg("Failed to scan test measurement\n");
    }
    if ( CheckMeas(Val, Args) ) {
       fprintf(stderr, "Bad result: %s = %g %s\n", Args.Func, Val, (Args.bPeriod ?
"s" :  "Hz"));
        sprintf(Command, "Connect a %lg Hz signal to A and try again\n",
Args.RefFreq);
        QuitMsg(Command);
    }
```

```
// Set the timeout for message-based communication (10 s)
   if (viSetAttribute(Instr, VI_ATTR_TMO_VALUE, 10000) != VI_SUCCESS) {
       QuitMsg("Failed to set timeout\n");
    }
// Set up for "infinite" number of measurements
    printf("\n");
    sprintf(Command, "trig:coun 1;:arm:coun inf");
    if (WriteDev(Command) != VI_SUCCESS) Quit();
// set pacing. note:  for freq:btb meas time is actual pacing
    if ( Args.bPeriod ) {
       sprintf(Command, "trig:sour tim;:trig:tim %lg", Args.Pacing);
    }
    else {
       sprintf(Command, "sens:acq:aper %lg", Args.Pacing);
    }
    if (WriteDev(Command) != VI_SUCCESS) Quit();
// FORMat PACKed is the recommended format for maximum fetch
// speed and for best timestamp resolution.
    sprintf(Command, "form pack;:form:tinf on;:disp:enab off");
    if (WriteDev(Command) != VI_SUCCESS) Quit();
    PrevTSVal = 0;
    Failed = false;
    Sleep(500);
// Start the measurement.
    if (WriteDev("init") != VI_SUCCESS) Quit();
// Fetch the measurement results as it goes.
    Count = 0;
    while (true) {
        CheckUserCancel();
// The 'max' parameter means fetch as many samples as is
// currently available for fetching (but no more than
// the upper limit, which by default is 10000).
       if (WriteDev("fetc:arr? max") != VI_SUCCESS) Quit();
       if (ReadDev(Buffer, BUFSIZE, &ReadLength) != VI_SUCCESS) Quit();
       Buffer[ReadLength] = 0; // Null terminate.
       pBuf = Buffer;
// Check for fetc:arr? max 'no data' marker.
       char *p = pBuf;
       if (*p++ == '#' && *p++ == '1' && *p == '0') {
// There is no data available at the moment.  Wait a
// bit with the next fetch attempt in order to avoid
// swamping the instrument with useless operations
// which could actually starve the measurement
// handling in the instrument.
             Sleep(20);
             continue;
       }
```

```
// Scan FORMat PACKed header.
        if (*pBuf++ != '#') {
           printf("Failed to scan packed header start\n");
            WriteDev("abort");
            Quit();
        }
        Digits = *pBuf++ - '0';
        if (Digits < 1 || Digits > 9) {
           printf("Failed to scan packed header size\n");
            WriteDev("abort");
            Quit();
        }
        int Size = 0;
        for (i=0; i<Digits; i++) {
           Size = 10 * Size + (int)(*pBuf++ - '0');
        }
// With format packed and format:tinf on each sample is
// a double format measurement value and a 64 bit
// integer timestamp (in ps), for a total of
// 16 bytes / sample.
        Samples = Size / 16;
        for (i=0; i<Samples; i++) {
            Val = *((double*)pBuf);
            pBuf += 8;
            if (i == 0 && _isnan(Val)) {
// Invalid value response.
              printf("The instrument is apparently no longer measuring.\n");
                Failed = true;
                break;
             }
            TSVal = *((__int64*)pBuf);
            pBuf += 8;
// Do something with the fetched result.  For this
// test just check that the measurement result seems
// reasonable and that the timestamps increase as
// they should.
           if (CheckMeas(Val, Args)) {
             printf("Bad result of measurement %lf: %g %s\n", (double)Count, Val,
(Args.bPeriod ?  "s" :  "Hz"));
// Check that the timestamps keep increasing during
// the test run.
              if (TSVal <= PrevTSVal) {
                printf("Invalid timestamp, sample %lf, prev = %lf, Cur = %lf\n",
                 (double)Count, (double)PrevTSVal * 1e-12,
                  (double)TSVal * 1e-12);
               }
// Check for gaps in the measurement data.  This will
```

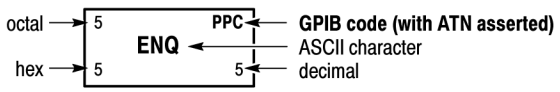
```
// happen if we try to measure faster than we can
// keep up with fetching.
                if (Count != 0 &&
                fabs((double)(TSVal - PrevTSVal) * 1e-12 - Args.Pacing) >
                 1.5 * Args.Pacing) {
                    printf("Gap: %lf -> %lf\n",
                    (double)PrevTSVal * 1e-12,
                    (double)TSVal * 1e-12);
                 }
                 PrevTSVal = TSVal;
                 Count++;
// Display some progress.
                 if (Count % 10000 == 0) {
                   printf("Sample %.0lf, value %.8le, timestamp %lf\n",
                   (double)Count, Val, (double)TSVal * 1e-12);
                 }
              }
              if (Failed) {
                 break;
              }
           }
           ReportAndQuit();
           return(0);
      }
}
```

Appendices

Appendix A: Character Set

B7 B6 B5 / BITS B4 B3 B2 B1	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
	CONTROL		NUMBERS SYMBOLS		UPPER CASE		LOWER CASE	
0 0 0 0	0 NUL 0 0	20 DLE 10 16	40 LA0 SP 20 32	60 LA16 0 30 48	100 TA0 @ 40 64	120 TA16 P 50 80	140 SA0 ` 60 96	160 SA16 p 70 112
0 0 0 1	1 GTL SOH 1 1	21 LL0 DC1 11 17	41 LA1 ! 21 33	61 LA17 1 31 49	101 TA1 A 41 65	121 TA17 Q 51 81	141 SA1 a 61 97	161 SA17 q 71 113
0 0 1 0	2 STX 2 2	22 DC2 12 18	42 LA2 " 22 34	62 LA18 2 32 50	102 TA2 B 42 66	122 TA18 R 52 82	142 SA2 b 62 98	162 SA18 r 72 114
0 0 1 1	3 ETX 3 3	23 DC3 13 19	43 LA3 # 23 35	63 LA19 3 33 51	103 TA3 C 43 67	123 TA19 S 53 83	143 SA3 c 63 99	163 SA19 s 73 115
0 1 0 0	4 SDC EOT 4 4	24 DCL DC4 14 20	44 LA4 $ 24 36	64 LA20 4 34 52	104 TA4 D 44 68	124 TA20 T 54 84	144 SA4 d 64 100	164 SA20 t 74 116
0 1 0 1	5 PPC ENQ 5 5	25 PPU NAK 15 21	45 LA5 % 25 37	65 LA21 5 35 53	105 TA5 E 45 69	125 TA21 U 55 85	145 SA5 e 65 101	165 SA21 u 75 117
0 1 1 0	6 ACK 6 6	26 SYN 16 22	46 LA6 & 26 38	66 LA22 6 36 54	106 TA6 F 46 70	126 TA22 V 56 86	146 SA6 f 66 102	166 SA22 v 76 118
0 1 1 1	7 BEL 7 7	27 ETB 17 23	47 LA7 ' 27 39	67 LA23 7 37 55	107 TA7 G 47 71	127 TA23 W 57 87	147 SA7 g 67 103	167 SA23 w 77 119
1 0 0 0	10 GET BS 8 8	30 SPE CAN 18 24	50 LA8 (28 40	70 LA24 8 38 56	110 TA8 H 48 72	130 TA24 X 58 88	150 SA8 h 68 104	170 SA24 x 78 120
1 0 0 1	11 TCT HT 9 9	31 SPD EM 19 25	51 LA9) 29 41	71 LA25 9 39 57	111 TA9 I 49 73	131 TA25 Y 59 89	151 SA9 i 69 105	171 SA25 y 79 121
1 0 1 0	12 LF A 10	32 SUB 1A 26	52 LA10 * 2A 42	72 LA26 : 3A 58	112 TA10 J 4A 74	132 TA26 Z 5A 90	152 SA10 j 6A 106	172 SA26 z 7A 122
1 0 1 1	13 VT B 11	33 ESC 1B 27	53 LA11 + 2B 43	73 LA27 ; 3B 59	113 TA11 K 4B 75	133 TA27 [5B 91	153 SA11 k 6B 107	173 SA27 { 7B 123
1 1 0 0	14 FF C 12	34 FS 1C 28	54 LA12 , 2C 44	74 LA28 < 3C 60	114 TA12 L 4C 76	134 TA28 \ 5C 92	154 SA12 l 6C 108	174 SA28 ¦ 7C 124
1 1 0 1	15 CR D 13	35 GS 1D 29	55 LA13 – 2D 45	75 LA29 = 3D 61	115 TA13 M 4D 77	135 TA29] 5D 93	155 SA13 m 6D 109	175 SA29 } 7D 125
1 1 1 0	16 SO E 14	36 RS 1E 30	56 LA14 . 2E 46	76 LA30 > 3E 62	116 TA14 N 4E 78	136 TA30 ^ 5E 94	156 SA14 n 6E 110	176 SA30 ~ 7E 126
1 1 1 1	17 SI F 15	37 US 1F 31	57 LA15 / 2F 47	77 UNL ? 3F 63	117 TA15 O 4F 79	137 UNT – 5F 95	157 SA15 o 6F 111	177 RUBOUT (DEL) 7F 127
	ADDRESSED COMMANDS	UNIVERSAL COMMANDS	LISTEN ADDRESSES		TALK ADDRESSES		SECONDARY ADDRESSES OR COMMANDS	

KEY



Tektronix
REF: ANSI STD X3.4-1977
IEEE STD 488.1-1987
ISO STD 646-2973

Appendix B: Default Command Settings

Command	Default setting
ARM:SLOPe	POS
ARM:SOURce	IMM
CALCulate:STATe	OFF
CALibration:INTerpolator:AUTO	ON
FORMat:BORDer	NORMal
INPut{[1]|2}:LEVel:RELative	Depending on function
FREQuency:BURSt:PRESCaler[:STATe]	ON
FREQuency:BURSt:APERture	200 µs
FREQuency:BURSt:SYNC:PERiod	400 µs
FREQuency:BURSt:STARt:DELay	200 µs
HF:ACQuisition[:STATe]	ON
HF:FREQuency:CENTer	300 MHz
TIError:FREQuency:AUTO	OFF
SYSTem:TOUT:AUTO	OFF.
TRIGger:SOURce	IMM
TRIGger:TIMer	20 ms
*DDT	#215ARM:LAY2;:FETC? in native mode and #14INIT in compatible mode
ARM:DELay	0
ARM:STOP:SLOPe	POS
ARM:STOP:SOURce	IMM
ARM:STOP:TIMer	0
CALCulate:AVERage:COUNt	100
CALCulate:AVERage:STATe	OFF
CALCulate:AVERage:TYPE	MEAN
CALCulate:DATA?	Event, no * RSTcondition.
CALCulate:IMMediate	Event, no * RSTcondition.
CALCulate:LIMit	OFF
CALCulate:LIMit:CLEar:AUTO	OFF
CALCulate:LIMit:LOWer	0
CALCulate:LIMit:LOWer:STATe	0
CALCulate:LIMit:UPPer	0
CALCulate:LIMit:UPPer:STATe	0
CALCulate:MATH	K=1, L=0, M=1 (No calculation)
CALCulate:MATH:STATe	OFF
CONFigure:TOTalize[:CONTinuous]	(@1),(@2)

Command	Default setting
DISPlay:ENABle	ON
FORMat	ASCII
FORMat:SMAX	Not affected
INITiate:CONTinuous	OFF
INPut{[1]|2}:COUPling	Input A (1): AC
INPut{[1]|2}:COUPling	Input B (2): AC
INPut{[1]|2}:FILTer	OFF
INPut{[1]|2}:FILTer:DIGital	OFF
INPut{[1]|2}:FILTer:DIGital:FREQuency	100 kHz
INPut{[1]|2}:IMPedance	1 MΩ
INPut{[1]|2}:LEVel	0 (but controlled by Autotrigger since AUTO is on after * RST)
INPut{[1]|2}:SLOPe	POS
ACQuisition:APERture	10 ms after *RST
ACQuisition:HOFF	OFF
ACQuisition:HOFF:TIME	200 µs
FREQuency:POWer:UNIT	DBM
FREQuency:RANGe:LOWer	100 (Hz)
FUNCtion	FREQuency_1
FREQuency:REGRession	AUTO
ROSCillator:SOURce	AUTO
TOTalize:GATE	OFF
SYSTem:TOUT	0
SYSTem:TOUT:TIME	0.1 s
TEST:SELect	ALL
TRIGger:COUNt	1

Appendix C: Instrument Settings After *RST

PARAMETER	VALUE/ SETTING
Inputs A & B	
Trigger Level	AUTO
Impedance	1M Ω
Manual Attenuator	1X
Coupling	AC
Trigger Slope	POS
Filter	OFF
Arming	
Start	OFF
Start Slope	POS
Start Arm Delay	0
Stop	OFF
Stop Slope	POS
Source	IMM
Hold-Off	
Hold-Off State	OFF
Hold-Off Time	200 µs
Time-Out	
Time-Out State	OFF
Time-Out Time	100 ms
Statistics	
Statistics State	OFF
number of Samples	100
number of Bins	20
Pacing State	OFF
Pacing Time	20 ms
Mathematics	
Mathematics State	OFF
Constants	K=M=1, L=0
Limits	
Limit State	OFF
Limit Mode	RANGE
Lower Limit	0
Upper Limit	0
Burst	
Sync Delay	400 µs

PARAMETER	VALUE/ SETTING
Start Delay	200 µs
Meas. Time	200 µs
Freq. Limit	300 MHz
Miscellaneous	
Function	FREQ A
Smart Frequency	AUTO
Smart Time Interval	OFF
Meas. Time	10 ms
Memory Protection (Memory 1 to 10)	Not changed by *RST
Auto Trig Low Freq Lim	100 Hz
Timebase Reference	AUTO
Arm-Trig State	IDLE (equivalent to sending :INIT:CONT OFF)

Appendix D: Reserved Words

*CLS
*DDT
*DMC
*EMC
*ESE
*ESR
*GMC
*IDN
*LMC
*LRN
*OPC
*OPT
*PMC
*PSC
*PUD
*RCL
*RMC
*RST
*SAV
*SRE
*STB
*TRG
*TST
*WAI
ABORt
AC
ACQuisition
ADDRess
ADEViation
ADIVB
ALARm
ALL
AMINUSB
APERture
APLUSB
ARM
ARRay
ASCII
ATTenuation
AUTO
AVERage
BORDer
BTBack
BURSt
BUS
Blockdata
Boolean
CALCulate
CALibration
CENTer
CLEar
COMMunicate
COMPatible
CONDition
CONFigure
CONTinuous
COUNt
COUPling
CURRent
DATA
DBM
DC
DCYCle
DELay
DELete
DIGital
DISPlay
DREGister0
ENABle
ERRor
EXT
EXTernal1
EXTernal2
EXTernal4
FAIL
FALL
FCOunt
FETCh
FILTer
FORMat
FREE
FREQuency
FTIM
FUNCtion
GATE
GPIB
HCOPy
HF
HOFF
IMMediate
IMPedance
INFinity
INITiate
INPut
INSTRument
INT
INTerpolator
INVerted
LANGuage
LAYer2
LEVel
LIMit
LOCK
LOGic
LOWer
MACRo
MATH
MAX
MAXimum
MEAN
MEASure
MEMory
MIN
MINNumericvalue
MINimum
N
NAME
NATive
NCYCles
NDUTycycle
NEG
NEGative
NORMal
NSLEwrate
NSTates
NWIDth
OFF
ON
ONCE
OPERation
OUTPut
PACKed
PCOunt
PDUTycycle
PERiod
PHASe
POLarity
POS
POSitive
POWer
PREScaler
PRESet
PRF
PSLEwrate
PTPeak
PULSe
PWIDth
QUEStionable
RAM
RANGe
RATio
READ
REAL
RECord
REGRession
RELative
RISE
ROM
ROSCillator
RTIM
SAVE
SCALar
SDEViation
SDUMp
SELect
SET
SETTings
SLOPe
SMAX
SOURce
STARt
STATe
STATus
STOP
STSTamp
SWAPped
SYNC
SYSTem
TALKonly
TBASe
TEMPerature
TEST
TIError
TIME
TIMer
TINFormation
TINTerval
TOTalize
TOUT
TRIGger
TSTAmp
TYPE
UNIT
UNPRotect
UPPer
VOLT
W
WIDTh

Index

A

Abbreviating commands, 2-3
ABORt, 2-29
ACQuisition:APERture, 2-29
ACQuisition:HOFF:TIME, 2-30
ACQuisition:HOFF, 2-29
ARM:COUNt, 2-30
ARM:DELay, 2-31
ARM:LAYer2, 2-31
ARM:LAYer2:SOURce, 2-32
ARM:SLOPe, 2-32
ARM:SOURce, 2-32
ARM:STOP:SLOPe, 2-33
ARM:STOP:SOURce, 2-33
ARM:STOP:TIMer, 2-34
AUTO, 2-35

C

CALCulate:AVERage:ALL?, 2-35
CALCulate:AVERage:COUNt, 2-35
CALCulate:AVERage:COUNt:CURRent?, 2-36
CALCulate:AVERage:STATe, 2-36
CALCulate:AVERage:TYPE, 2-37
CALCulate:LIMit:CLEar, 2-39
CALCulate:LIMit:CLEar:AUTO, 2-40
CALCulate:LIMit:FAIL?, 2-40
CALCulate:LIMit:FCOunt:LOWer?, 2-41
CALCulate:LIMit:FCOunt:UPPer?, 2-41
CALCulate:LIMit:FCOunt?, 2-40
CALCulate:LIMit:LOWer, 2-42
CALCulate:LIMit:LOWer:STATe, 2-42
CALCulate:LIMit:PCOunt?, 2-42
CALCulate:LIMit:UPPer, 2-43
CALCulate:LIMit:UPPer:STATe, 2-43
CALCulate:MATH:STATe, 2-44
CALCulate:TOTalize:TYPE, 2-45
CALCulate:DATA?, 2-37
CALCulate:IMMediate, 2-38
CALCulate:LIMit, 2-39
CALCulate:MATH, 2-43
CALCulate:STATe, 2-45
CALibration:INTerpolator:AUTO, 2-46
*CLS, 2-46
Command short form, 2-3
CONFigure:<MeasuringFunction>, 2-48
CONFigure:ARRay:<MeasuringFunction>, 2-47
CONFigure:TOTalize[:CONTinuous], 2-50

D

*DDT, 2-51
DISPlay:ENABle, 2-51
*DMC, 2-52

E

*EMC, 2-52
*ESE, 2-53
*ESR?, 2-54

F

FETCh[:SCALar]?, 2-55
FETCh:ARRay?, 2-54
FORMat, 2-56
FORMat:BORDer, 2-56
FORMat:SMAX, 2-57
FORMat:TINFormation, 2-57
FREQuency:BURSt:APERture, 2-58
FREQuency:BURSt:PRESCaler[:STATe], 2-58
FREQuency:BURSt:STARt:DELay, 2-58
FREQuency:BURSt:SYNC:PERiod, 2-59
FREQuency:POWer:UNIT, 2-59
FREQuency:RANGe:LOWer, 2-60
FREQuency:REGRession, 2-60
FUNCtion, 2-61

G

*GMC?, 2-62

H

HCOPy:SDUMp:DATA?, 2-63
HF:ACQuisition[:STATe], 2-63
HF:FREQuency:CENTer, 2-64

I

*IDN?, 1-1

INITiate, 2-64
INITiate:CONTinuous, 2-65
INPut{[1]|2}:ATTenuation, 2-65
INPut{[1]|2}:COUPling, 2-66
INPut{[1]|2}:FILTer:DIGital, 2-66
INPut{[1]|2}:FILTer:DIGital:FREQuency, 2-67
INPut{[1]|2}:IMPedance, 2-67
INPut{[1]|2}:LEVel:AUTO, 2-69
INPut{[1]|2}:LEVel:RELative, 2-69
INPut{[1]|2}:FILTer, 2-66
INPut{[1]|2}:LEVel, 2-68
INPut{[1]|2}:SLOPe, 2-70

L

*LMC?, 2-71
*LRN?, 2-71

M

MEASure:<MeasuringFunction>?, 2-79
MEASure:ARRay:<MeasuringFunction>?, 2-72
MEASure:ARRay:FREQuency:BTBack?, 2-71
MEASure:ARRay:PERiod:BTBack?, 2-73
MEASure:ARRay:STSTamp?, 2-74
MEASure:ARRay:TIError?, 2-75
MEASure:ARRay:TSTAmp?, 2-75
MEASure:FREQuency:BURSt?, 2-77
MEASure:FREQuency:POWer[:AC]?, 2-78
MEASure:FREQuency:PRF?, 2-78
MEASure:FREQuency:RATio?, 2-79
MEASure:MEMory<N>?, 2-81
MEASure:PERiod:AVERage?, 2-84
MEASure{:FALL:TIME|:FTIM}?, 2-76
MEASure{:PDUTycycle|:DCYCle}?, 2-83
MEASure{:RISE:TIME|:RTIM}?, 2-85
MEASure[:VOLT]:MAXimum?, 2-86
MEASure[:VOLT]:MINimum?, 2-86
MEASure[:VOLT]:NCYCles?, 2-87
MEASure[:VOLT]:NSLEwrate?, 2-87
MEASure[:VOLT]:PSLEwrate?, 2-87
MEASure[:VOLT]:PTPeak?, 2-88
MEASure[:VOLT]:RATio?, 2-88
MEASure:FREQuency?, 2-76
MEASure:MEMory?, 2-81
MEASure:NDUTycycle?, 2-82
MEASure:NWIDth?, 2-82
MEASure:PERiod?, 2-83
MEASure:PHASe?, 2-84
MEASure:PWIDth?, 2-85
MEASure:TINTerval?, 2-86
MEMory:DATA:RECord:COUNt?, 2-88
MEMory:DATA:RECord:DELete, 2-89
MEMory:DATA:RECord:FETCh:ARRay?, 2-89
MEMory:DATA:RECord:FETCh:STARt, 2-89
MEMory:DATA:RECord:FETCh?, 2-89
MEMory:DATA:RECord:NAME?, 2-90
MEMory:DATA:RECord:SAVE, 2-90
MEMory:DATA:RECord:SETTings?, 2-90
MEMory:DELete:MACRo, 2-91
MEMory:FREE:MACRo?, 2-91
MEMory:NSTates?, 2-91
MSS bit, 3-2

O

*OPC, 2-92
*OPT?, 2-92
OUTPut:POLarity, 2-93
OUTPut:TYPE, 2-93

P

*PMC, 2-94
*PSC, 2-94
*PUD, 2-95

R

*RCL, 2-96
READ?, 2-96
READ:ARRay?, 2-97
*RMC, 2-98
ROSCillator:SOURce, 2-98
RQS bit, 3-2
*RST, 2-98

S

*SAV, 2-99
SOURce:PULSe:PERiod, 2-99
SOURce:PULSe:WIDTh, 2-100
*SRE, 2-100
STATus:DREGister0:ENABle, 2-101
STATus:OPERation:CONDition?, 2-102
STATus:OPERation:ENABle, 2-103
STATus:QUEStionable:CONDition?, 2-105

STATus:QUEStionable:ENABle, 2-106
STATus:QUEStionable?, 2-105
STATus:DREGister0?, 2-101
STATus:OPERation?, 2-102
STATus:PRESet, 2-104
*STB?, 2-107
SYSTem:COMMunicate:GPIB:ADDRess, 2-108
SYSTem:ERRor?, 2-109
SYSTem:LANGuage, 2-109
SYSTem:PRESet, 2-109
SYSTem:SET, 2-110
SYSTem:TALKonly, 2-110
SYSTem:TEMPerature?, 2-111
SYSTem:TOUT, 2-111
SYSTem:TOUT:AUTO, 2-112
SYSTem:TOUT:TIME, 2-112
SYSTem:UNPRotect, 2-113

T

TEST:SELect, 2-113
TIError:FREQuency:AUTO, 2-114
TIError:FREQuency, 2-114
TINTerval:AUTO, 2-115
TOTalize:GATE, 2-115
*TRG, 2-116
TRIGger:COUNt, 2-116
TRIGger:SOURce, 2-117
TRIGger:TIMer, 2-117
Truncating commands, 2-3
*TST?, 2-118

W

*WAI, 2-118