Timer/Counter/ Analyzer

PM6690

Programming Manual

FLUKE®

4822 872 20305
April 2005 - First Edition

GENERAL INFORMATION

Method of Notation

This manual contains directions for use that apply to the Timer/Counter/Analyzer PM6690.
In order to simplify the references, the PM6690 is further referred to throughout this manual as the '90'.

Warranty

The *Warranty Statement* is part of the *Getting Started Manual* that is included with the shipment.

Declaration of Conformity

The complete text with formal statements concerning product identification, manufacturer and standards used for type testing is available on request.

This page is intentionally left blank.

Table of Contents

9 Index

This page is intentionally left blank.

Chapter 1

Getting Started

Finding Your Way Through This Manual

You should use this Programming Manual together with the Operators Manual. That manual contains specifications for the counter and explanations of the possibilities and limitations of the different measuring functions.

Sections

The chapters in this manual are divided into three sections aimed at different levels of reader knowledge.

The 'General' Section, which can be disregarded by the users who know the IEEE-488 and SCPI standards:

– *Chapter 2, Default Settings, summarizes the instrument settings after sending the *RST command.*

– *Chapter 3, Introduction to SCPI, explains syntax data formats, status reporting, etc.*

The Practical Section of this manual contains:

– *Chapter 4, Programming Examples, with typical programs for a number of applications. These programs are written in C and are also available as text files on the included Manual CD.*

The 'Programmer's Reference' Section of this manual contains:

– *Chapter 5, Instrument Model, explains how the instrument looks from the bus. This instrument is not quite the same as the one used from the front panel.*

– *Chapter 6, Using the Subsystems, explains more about each subsystem.*

– *Chapter 7, Error Messages, contains a list of all error messages that can be generated during bus control.*

– *Chapter 8, Command Reference, gives complete information on all commands. The subsystems and commands are sorted alphabetically.*

Index

You can also use the index to get an overview of the commands. The index is also useful when looking for additional information on the command you are currently working with.

Manual Conventions

Syntax Specification Form

This manual uses the EBNF (Extended Backus-Naur Form) notation for describing syntax. This notation uses the following types of symbols:

■ Printable Characters:

Printable characters such as Command headers, etc., are printed just as they are, e.g. period means that you should type the word PERIOD.

The following printable characters have a special meaning and will only be used in that meaning: # ‘ “ () : ; *
Read Chapter 3’ Introduction to SCPI’ for more information.

■ Non-printable Characters:

Two non-printable characters are used:

␣ *indicates the space character (ASCII code 32).*

↵ *_ indicates the new line character (ASCII code 10).*

■ Specified Expressions: < >

Symbols and expressions that are further specified elsewhere in this manual are placed between the <> signs.
For example <Dec. data.>. The following explanation is found on the same page: “Where <Dec. data> is a four-digit number between 0.1 and 8*10-9.

■ Alternative Expressions Giving Different Result:

Alternative expressions giving different results are separated by |. For example, On|Off means that the function may be switched on or off.

■ Grouping: « »

Example: FORMat␣«ASCII|REAL» specifies the command header FORMat followed by a space character and either ASCII or REAL.

■ Optionality: [ ]

An expression placed within [ ] is optional.

Example: `[:VOLT]:FREQuency`

means that the command FREQuency may or may not be preceded by :VOLT.

■ Repetition: { }

An expression placed within { } can be repeated zero or more times.

■ Equality: =

Equality is specified with =
Example: <Separator>= ,

Mnemonic Conventions

■ Truncation Rules

All commands can be truncated to shortforms. The truncation rules are as follows:

– The shortform is the first four characters of the command.

– If the fourth character in the command is a vowel, then the shortform is the first three characters of the command. This rule is not

used if the command is only four characters.

– If the last character in the command is a digit, then this digit is appended to the shortform.

Examples:

Longform	Shortform
:MEASURE	:MEAS
:NEGATIVE	:NEG
:DREGISTER0	:DREG0
:EXTERNAL4	:EXT4

The shortform is always printed in CAPITALS in this manual: :MEASure, :NEGative, :DREGister0, :EXTernal4 etc.

■ Example Language

Small examples are given at various places in the text. These examples are not in BASIC or C, nor are they written for any specific controller. They only contain the characters you should send to the counter and the responses that you should read with the controller.

Example:

SEND→ `MEAS:FREQ?`

This means that you should program the controller so that it addresses the counter and outputs this string on the GPIB.

READ← `1.234567890E6`

This means that you should program the controller so that it can receive this data from the GPIB, then address the counter and read the data.

Setting Up the Instrument

Setting the GPIB Address

The address of the counter is set to 10 when it is delivered. Press **USER OPT** → **Interface** to see the active address above the soft key labeled **GPIB address**.

If you want to use another bus address, you can press **GPIB address** to enter a value menu where you can set the address between 0 and 30 by means of the numeric keys.

The address can also be set via a GPIB command. The set address is stored in nonvolatile memory and remains until you change it.

Power-On

When turned on, the counter starts with the setting it had when turned off.

■ Standby

When the counter is in REMOTE mode, you cannot switch it off. You must first enable Local control by pressing the **Cancel** ("**C**") key.

Testing the Bus

To test that the instrument is operational via the bus, use *IDN? to identify the instrument and *OPT? to identify which options are installed. (See ‘System Subsystem’ , *IDN? and *OPT?)

Interface Functions

What can I do with the Bus?

All the capabilities of the interface for the '90' are explained below.

■ Summary

Description,	Code
Source handshake,	SH1
Acceptor handshake,	AH1
Control function,	C0
Talker Function,	T6
Listener function,	L4
Service request,	SR1
Remote/local function,	RL1
Parallel poll,	PP0
Device clear function,	DC1
Device trigger function,	DT1
Bus drivers,	E2

■ SH1 and AH1

These simply mean that the counter can exchange data with other instruments or a controller using the bus handshake lines: DAV, NRFD, NADC.

■ Control Function, C0

The counter does not function as a controller.

■ Talker Function, T6

The counter can send responses and the results of its measurements to other devices or to the controller. T6 means that it has the following functions:

– Basic talker.

– No talker only.

– It can send out a status byte as response to a serial poll from the controller.

– Automatic un-addressing as a talker when it is addressed as a listener.

■ Listener Function, L4

The counter can receive programming instructions from the controller. L4 means that it has the following functions:

– Basic listener.

– No listen only.

– Automatic un-addressing as listener when addressed as a talker.

■ Service Request, SR1

The counter can call for attention from the controller, e.g., when a measurement is completed and a result is available.

■ Remote/Local, RL1

You can control the counter 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.

■ Parallel Poll, PP0

The counter does not have any parallel poll facility.

■ Device Clear, DC1

The controller can reset the counter via interface message DCL (Device clear) or SDC (Selective Device Clear).

■ Device Trigger, DT1

You can start a new measurement from the controller via interface message GET (Group Execute Trigger).

■ Bus Drivers, E2

The GPIB interface has tri-state bus drivers.

Using the USB Interface

The counter is equipped with a USB full speed interface, which supports the same SCPI 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 can be found 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 for several operating systems, from National Instruments (www.ni.com). The Windows version is supplied on the CD.

In order to test the communication and send single commands, the National Instruments utility supplied with the NI-VISA drivers can be used 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 found in Chapter .

Instruments connected to the USB bus are identified with:

Vendor ID: 0x14EB for Pendulum Instruments.

Model ID: 0x0090 for the '90' and serial number of the instrument.

This data is combined to form a unique identifier string such as:

USB\VID_14EB&PID_0090\991234 or

"USB0::0x14EB::0x0090::991234::INSTR"

When connecting to the instrument, any part of this string may be used to identify the instrument, for example any instrument from this vendor, any instrument of a certain type or a specific instrument serial number.

Chapter 2

Default Settings

Default settings (after *RST)

PARAMETER	VALUE/ SETTING
Inputs A & B	
Trigger Level	AUTO
Impedance	1 MΩ
Manual Trigger Level (Controlled by autotrigger)	0V
Manual Attenuator (Controlled by autotrigger)	1X
Coupling	AC
Trigger Slope	POS
Filter	OFF
Arming	
Start	OFF
Start Slope	POS
Start Arm Delay	200 µs
Stop	OFF
Stop Slope	POS
Channel	Ext Arm Input E
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
No. of Samples	100

PARAMETER	VALUE/ SETTING
Mathematics	
Mathematics State	OFF
Constants	K=M=1, L=0
Limits	
Limit State	OFF
Limit Mode	ABOVE
Lower Limit	0
Upper Limit	0
Burst	
Sync Delay	200 µs
Start Delay	200 µs
Meas. Time	200 µs
Freq. Limit	300 MHz
Miscellaneous	
Function	FREQ A
Meas. Time	200 ms
Memory Protection (Memory 1 to 10)	Not changed by *RST
Smart Time Interval	OFF
Auto Trig Low Freq Lim	100 Hz
Timebase Reference	INT
Trigger Idle State (equivalent to sending :INIT:CONT OFF)	ON

Chapter 3

Introduction to SCPI

What is SCPI?

SCPI (Standard Commands for Programmable Instruments) is a standardized set of commands used to remotely control programmable test and measurement instruments. The instrument firmware contains the SCPI. It defines the syntax and semantics that the controller must use to communicate with the instrument.

This chapter is an overview of SCPI and shows how SCPI is used in Fluke Frequency Counters and Timer/Counters.

SCPI is based on IEEE-488.2 to which it owes much of its structure and syntax. SCPI can, however, be used with any of the standard interfaces, such as GPIB (=IEC625/IEEE-488), VXI and RS-232.

Reason for SCPI

For each instrument function, SCPI defines a specific command set. The advantage of SCPI is that programming an instrument is only function dependent and no longer instrument dependent. Several different types of instruments, for example an oscilloscope, a counter and a multimeter, can carry out the same function, such as frequency measurement. If these instruments are SCPI compatible, you can use the same commands to measure the frequency on all three instruments, although there may be differences in accuracy, resolution, speed, etc.

Compatibility

SCPI provides two types of compatibility: Vertical and horizontal.

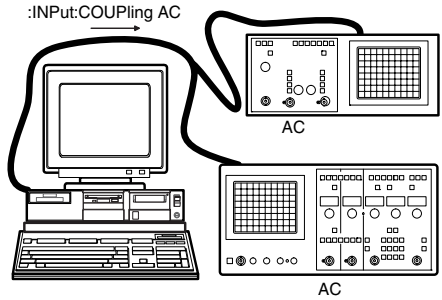


Figure 3-1 Vertical

This means that all instruments of the same type have identical controls. For example, oscilloscopes will have the same controls for timebase, triggers and voltage settings.

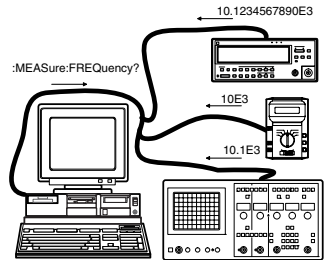


Figure 3-2 Horizontal

This means that instruments of different types that perform the same functions have the same commands. For example, a DMM, an oscilloscope, and a counter can all measure frequency with the same commands.

Management and Maintenance of Programs

SCPI simplifies maintenance and management of the programs. Today changes and additions in a good working program are hardly possible because of the great diversity in program messages and instruments. Programs are difficult to understand for anyone other than the original programmer. After some time even the programmer may be unable to understand them.

A programmer with SCPI experience, however, will understand the meaning and reasons of a SCPI program, because of his knowledge of the standard. Changes, extensions, and additions are much easier to make in an existing application program. SCPI is a step towards portability of instrument programming software and, as a consequence, it allows the exchange of instruments.

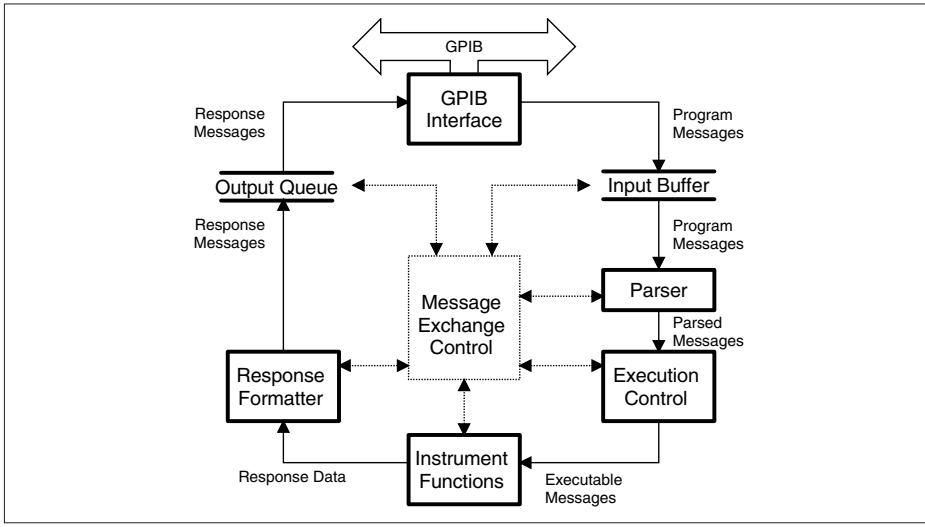


Figure 3-3 *Overview of the firmware in a SCPI instrument.*

How does SCPI Work in the Instrument?

The functions inside an instrument that control the operation provide SCPI compatibility. shows a simplified logical model of the message flow inside a SCPI instrument.

When the controller sends a message to a SCPI instrument, roughly the following happens:

– The GPIB controller addresses the instrument as listener.

– The GPIB interface function places the message in the Input Buffer.

– The Parser fetches the message from the Input Buffer, parses (decodes) the message, and checks for the correct syntax. The instrument reports incorrect syntax by sending command errors via the status system to the controller. Moreover, the parser will detect if the controller requires a response. This is the case when the input message is a query (command with a "?" appended).

The Parser will transfer the executable messages to the Execution Control block in token form (internal codes). The Execution Control block will gather the information required for a device action and will initiate the requested task at the appropriate time. The instrument reports execution errors via the status system over the GPIB and places them in the Error Queue.

– When the controller addresses the instrument as talker, the instrument takes data from the Output Queue and sends it over the GPIB to the controller.

Message Exchange Control protocol

Another important function is the Message Exchange Control, defined by IEEE 488.2. The Message Exchange Control protocol specifies the interactions between the several functional elements that exist between the GPIB functions and the device-specific functions, see .

The Message Exchange Control protocol specifies how the instrument and controller should exchange messages. For example, it specifies exactly how an instrument shall handle program and response messages that it receives from and returns to a controller.

This protocol introduces the idea of commands and queries; queries are program messages that require the device to send a response. When the controller does not read this response, the device will generate a Query Error. On the other hand, commands will not cause the device to generate a response. When the controller tries to read a response anyway, the device then generates a Query Error.

The Message Exchange Control protocol also deals with the order of execution of program messages. It defines how to respond if Command Errors, Query Errors, Execution Errors, and Device-Specific errors occur. The protocol demands that the instrument report any violation of the IEEE-488.2 rules to the controller, even when it is the controller that violates these rules.

The IEEE 488.2 standard defines a set of operational states and actions to implement the message exchange protocol. These are shown in the following table:

State	Purpose
IDLE	Wait for messages
READ	Read and execute messages
QUERY	Store responses to be sent
SEND	Send responses
RESPONSE	Complete sending responses
DONE	Finished sending responses
DEADLOCK	The device cannot buffer more data

Action,	Reason
Unterminated,	The controller attempts to read the device without first having sent a complete query message
Interrupted,	The device is interrupted by a new program message before it finishes sending a response message

Protocol Requirements

In addition to the above functional elements, which process the data, the message exchange protocol has the following characteristics:

– The controller must end a program message containing a query with a message terminator before reading the response from the device (address the device as talker). If the controller breaks this rule, the device will report a query error (unterminated action).

– The controller must read the response to a query in a previously (terminated) program message before sending a new program message. When the controller violates this rule, the device will report a query error (interrupted action).

– The instrument sends only one response message for each query message. If the query message resulted in more than one answer, all answers will be sent in one response message.

■ Order of Execution

Deferred Commands

Execution control collects commands until the end of the message, or until it finds a query or other special command that forces execution. It then checks that the setting resulting from the commands is a valid one: No range limits are exceeded, no coupled parameters are in conflict, etc. If this is the case, the commands are executed in the sequence they have been received; otherwise, an execution error is generated, and the commands are discarded.
This deferred execution guarantees the following:

– All valid commands received before a query are executed before the query is executed.

– All queries are executed in the order they are received.

– The order of execution of commands is never reversed.

■ Sequential and Overlapped Commands

There are two classes of commands: sequential and overlapped commands. All commands in the counter are sequential, that is one command finishes before the next command executes.

Remote Local Protocol

■ Definitions

Remote Operation

When an instrument operates in remote, all local controls, except the local key, are disabled.

Local Operation

An instrument operates in local when it is not in remote mode as defined above.

Local Lockout

In addition to the remote state, an instrument can be set to remote with 'local lockout'. This disables the return-to-local button. In theory, the state local with local lockout is also possible; then, all local controls except the return-to-local key are active.

The Counter in Remote Operation

When the Counter is in remote operation, it disables all its local controls except the LOCAL key.

The Counter in Local Operation

When the Counter is in local operation the instrument is fully programmable both from the front panel and from the bus. If a bus message arrives while a change is being entered from the front panel, the front panel entry is interrupted and the bus message is executed.

We recommend you to use Remote mode when using counters from the bus. If not, the counter measures continously and the initiation command :INIT will have no effect.

Program and Response Messages

The communication between the system controller and the SCPI instruments connected to the GPIB takes place through Program and Response Messages. A Program Message is a sequence of one or more commands sent from the controller to an instrument. Conversely, a Response Message is the data from the instrument to the controller.

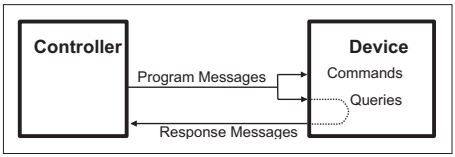


Figure 3-4 *Program and response messages.*

The GPIB controller instructs the device through program messages. The device will only send responses when explicitly requested to do so; that is, when the controller sends a query. Queries are recognized by the question mark at the end of the header, for example: *IDN? (requests the instrument to send identity data).

Syntax and Style

■ Syntax of Program Messages

A command or query is called a program message unit. A program message unit consists of a header followed by one or more parameters, as shown in .

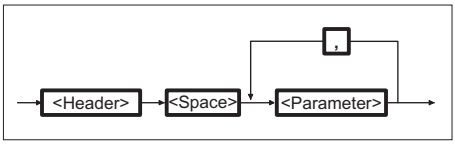


Figure 3-5 *Syntax of a Program Message Unit.*

One or more program message units (commands) may be sent within a simple program message, see Fig. 3-6.

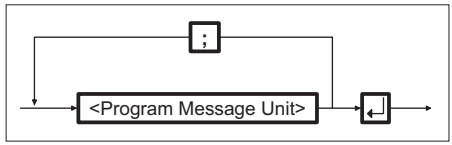


Fig 3-6 *Syntax of a terminated Program Message.*

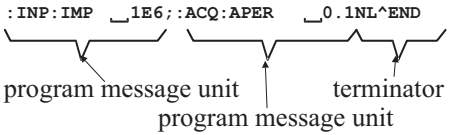
The ↵ is the pmt (program message terminator) and it must be one of the following codes:

↵	NL^END	This is <new line> code sent concurrently with the END message on the GPIB.
	NL	This is the <new line> code.
	<dab>^END	This is the END message sent concurrently with the last data byte <dab>.

NL is the same as the ASCII LF (<line feed> = ASCII $10_{decimal}$). The END message is sent via the EOI-line of the GPIB.
The ^ character stands for 'at the same time as'.

Most controller programming languages send these terminators automatically, but allow changing it. So make sure that the terminator is as above.

Example of a terminated program message:



This program message consists of two message units. The unit separator (semicolon) separates message units.

Basically there are two types of commands:

Common Commands

The common command header starts with the asterisk character (*), for example *RST.

SCPI Commands

SCPI command headers may consist of several keywords (mnemonics), separated by the colon character (:).

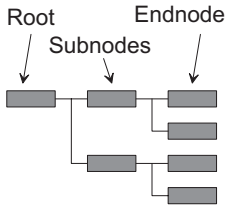


Figure 3-7 *The SCPI command tree.*

Each keyword in a SCPI command header represents a node in the SCPI command tree. The leftmost keyword (INPut in the previous example) is the root level keyword, representing the highest hierarchical level in the command tree.

The keywords following represent subnodes under the root node. See 'COMMAND TREE' on page 3-10 for more details of this subject.

Forgiving Listening

The syntax specification of a command is as follows:

ACQuisition:APERture␣<numeric value>

Where: ACQ and APER specify the shortform, and ACQuisition and APERture specify the longform. However, ACQU or APERT are not allowed and cause a command error.

In program messages either the long or the shortform may be used in upper or lower case letters. You may even mix upper and lower case. There is no semantic difference between upper and lower case in program messages. This instrument behavior is called forgiving listening.

For example, an application program may send the following characters over the bus:

SEND→ iNp:ImP␣1E6

The example shows the shortform used in a mix of upper and lower case

SEND→ Input:Imp␣1E6

The example shows a mix of longform and shortform and a mix of upper and lower case.

Notation Habit in Command Syntax

To clarify the difference between the forms, the shortform in a syntax specification is shown in upper case letters and the remaining part of the longform in lower case letters.

Notice however, that this does not specify the use of upper and lower case characters in the message that you actually sent. Upper and lower case letters, as used in syntax specifications, are only a notation convention to ease the distinction between longform and shortform.

■ Syntax of Response Messages

The response of a SCPI instrument to a query (response message unit) consists of one or more parameters (data elements) as the following syntax diagram shows. There is no header returned.

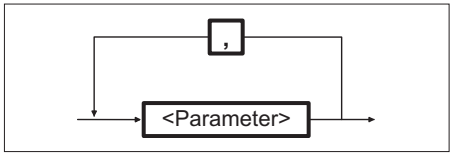


Figure 3-8 *Syntax of a Response Message Unit.*

If there are multiple queries in a program message, the instrument groups the multiple response message units together in one response message according to the following syntax:

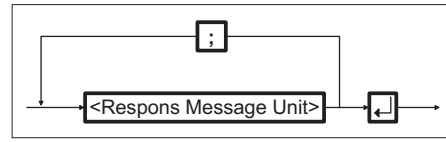


Fig 3-9 *Syntax of a Terminated Response Message.*

The response message terminator (rmt) is always NL^END, where:

NL^END is <new line> code (equal to <line feed> code = ASCII 10 decimal) sent concurrently with the END message. The END message is sent by asserting the EOI line of the GPIB bus.

Responses:

A SCPI instrument always sends its response data in shortform and in capitals.

Example:

You program an instrument with the following command:

SEND→ `:ROSCillator:SOURce␣EXTernal`

Then you send the following query to the instrument:

SEND→ `:ROSCillator:SOURce?`

The instrument will return:

READ← `EXT`

response in shortform and in capitals.

Command Tree

Command Trees like the one below are used to document the SCPI command set in this manual. The keyword (mnemonic) on the root level of the command tree is the name of the subsystem. The following example illustrates the Command Tree of the INPut1 subsystem.

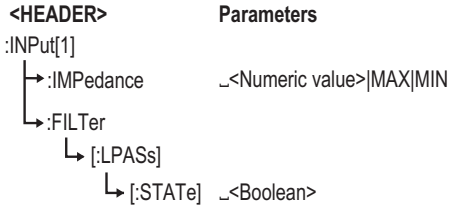


Figure 3-10 *Example of an INPut subsystem command tree.*

The keywords placed in square brackets are optional nodes. This means that you may omit them from the program message.

Example:

SEND→ `INPUT1:FILTER:LPASS :STATE␣ON`

is the same as

SEND→ `INPUT:FILTER␣ON`

Moving down the Command Tree

The command tree shows the paths you should use for the command syntax. A single command header begins from the root level downward to the 'leaf nodes' of the command tree. (Leaf nodes are the last keywords in the command header, before the parameters.)

■ **Example:**

SEND→ `INPut:EVENt:HYSTeresis`

Where: INPut is the root node and HYSTeresis is the leaf node.

Each colon in the command header moves the current path down one level from the root in the command tree. Once you reach the leaf node level in the tree, you can add several leaf nodes without having to repeat the path from the root level.

Just follow the rules below:

– Always give the full header path, from the root, for the first command in a new program message.

– For the following commands within the same program message, omit the header path and send only the leaf node (without colon).

You can only do this if the header path of the new leaf-node is the same as that of the previous one. If not, the full header path must be given starting with a colon.

Command header = Header path + leaf node

– Once you send the pmt (program message terminator), the first command in a new program message must start from the root.

■ **Example:**

SEND→ `INPut:EVENt:HYSTeresis MIN;LEVel␣0.5`

This is the command where:
INPut:EVENt is the header path and :HYSTeresis is the first leaf-node and LEVel is the second leaf node because LEVel is also a leaf-node under the header path INPut:EVENt.

There is no colon before LEVel!

Parameters

Numeric Data

Decimal data are printed as numerical values throughout this manual. Numeric values may contain both a decimal point and an exponent (base 10).

These numerals are often represented as NRf (NR = NumeRic, f = flexible) format.

■ Keywords

In addition to entering decimal data as numeric values, several keywords can exist as special forms of numeric data, such as MINimum, MAXimum, DEFault, STEP, UP, DOWN, NAN (Not A Number), INFinity, NINF (Negative INFinity). The Command Reference chapters explicitly specify which keywords are allowed by a particular command. Valid keywords for the counter are MAXimum and MINimum.

MINimum

This keyword sets a parameter to its minimum value.

MAXimum

This keyword sets a parameter to its maximum value.

The instrument always allows MINimum and MAXimum as a data element in commands, where the parameter is a numeric value. MIN and MAX values of a parameter can always be queried.

Example:

SEND→ `INP:LEV? MAX`

This query returns the maximum range value.

■ Suffixes

You can use suffixes to express a unit or multiplier that is associated with the decimal numeric data. Valid suffixes are s (seconds), ms (milliseconds), mohm (megaohm), kHz (kilohertz), mV (millivolt).

Example:

SEND→ `:SENS:ACQ:APER 100ms`

Where: ms is the suffix for the numeric value 100.

Notice that you may also send ms as MS or mS. MS does still mean milliseconds, not Mega Siemens!

Response messages do not have suffixes. The returned value is always sent using standard units such as V, S, Hz, unless you explicitly specify a default unit by a FORMat command.

Boolean Data

A Boolean parameter specifies a single binary condition which is either true or false.

Boolean parameters can be one of the following:

– ON or 1 means condition true.

– OFF or 0 means condition false.

■ **Example**

SEND→ :SYST:TOUT␣ON or
:SYST:TOUT␣1

This switches timeout monitoring on. A query, for instance :SYSTem:TOUT?, will return 1 or 0; never ON or OFF.

Expression Data

You must enclose expression program data in parentheses (). Three possibilities of expression data are as follows:

– <numeric expression data> <parameter list>

– <channel list>

An example of <numeric expression data> is: (X – 10.7E6) This subtracts a 10.7 MHz intermediate frequency from the measured result.

An example of <parameter list> is: (5,0.02) This is a list of two parameters; the first one is 5 and the second one 0.02.

An example of <channel list> is: (@3),(@1) This specifies channel 3 as the main channel and channel 1 as the second channel.

Other Data Types

Other data types that can be used for parameters are the following:

– String data: Always enclosed between single or double quotes, for example "This is a string" or 'This is a string.'

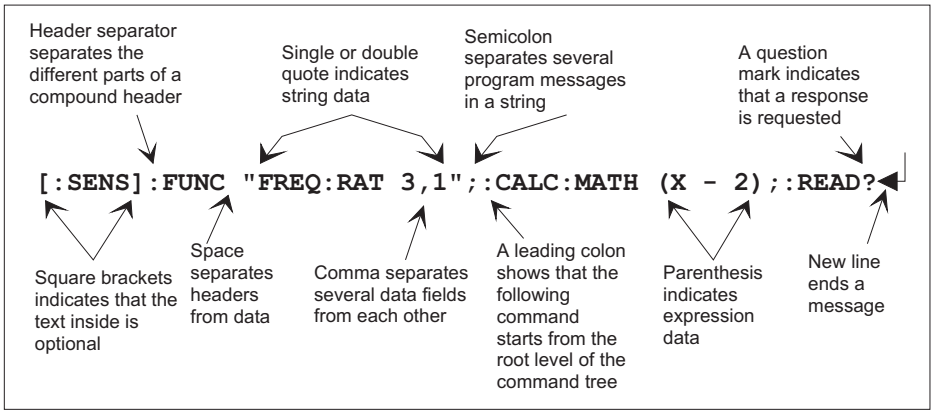
– Character data: For this data type, the same rules apply as for the command header mnemonics. For example: POSitive, NEGative, EITHer.

– Non-decimal data: For instance, #H3A for hexa decimal data.

– Block data: Used to transfer any 8-bit coded data. This data starts with a preamble that contains information about the length of the parameter.

Example:
#218INP:IMP␣50;SENS␣10

Summary



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 counter. 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 a counter command, the counter will execute the macro when macros are enabled (*EMC␣1), and it will execute the counter 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 counter will execute the sequence of commands.

Use the following syntax:

*DMC <macro-label>, <commands>

■ Simple Macros

Example:

```
SEND→ *DMC␣'MyInputSetting'
      #255:INP:IMP␣50;HYST␣1
      ;LEV␣0.55;:INP:HYST:AUTO
      ␣0;
```

This example defines a macro MyInputSetting, which sets the impedance to 50 Ω, sets the sensitivity to 1 V, the trigger level to +0.55 V, and switches off auto sensitivity and auto trigger level.

■ 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, etc.

Example:

```
SEND→ *DMC␣'AUTO',#247
      :INP:HYST:AUTO␣$1;
      :INP:IMP␣$2
```

This example defines a macro AUTO, which takes two arguments, i.e., auto «ON|OFF|ONCE» ($1) and impedance «50|1E6» ($2) .

```
SEND→ AUTO␣OFF,50
```

Switches off both auto sensitivity and auto trigger level and sets the input impedance to 50 Ω.

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.

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 a counter 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: `*EMC␣0` disables all macros.
Enabling: `*EMC␣1`

■ *EMC? Enable Macro Query

Use this query to determine if macros are enabled.

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:

```
SEND→ *DMC␣'LIMITMON','
      :CALC:STAT␣ON;
      :CALC:LIM:STAT␣ON;
      :CALC:LIM:LOW:DATA
      $1;STAT␣ON;
      :CALC:LIM:UPP:DATA
      $2;STAT␣ON'
SEND→ *EMC␣1
```

Now sending the command

```
SEND→ LIMITMON␣1E6,1.1E6
```

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

Retrieve a Macro

■ *GMC? Get Macro Contents Query

This query gives a response containing the definition of the macro you specified when sending the query.

Example using the above defined macro:

```
SEND→ *GMC?␣'LIMITMON'
READ← #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 gives a response containing the labels of all the macros stored in the Timer/Counter.

Example:

```
SEND→ *LMC?
 READ←"MYINPSETTING","LIMITMON
       "
```

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

Status Reporting System

Introduction

Status reporting is a method to let the controller know what the counter is doing. You can ask the counter what status it is in whenever you want to know.

You can select some conditions in the counter that should be reported in the Status Byte Register. You can also select if some bits in the Status Byte should generate a Service Request (SRQ).
(An SRQ is the instrument's way to call the controller for help.)

Read more about the Status Subsystem in Chapter 6.

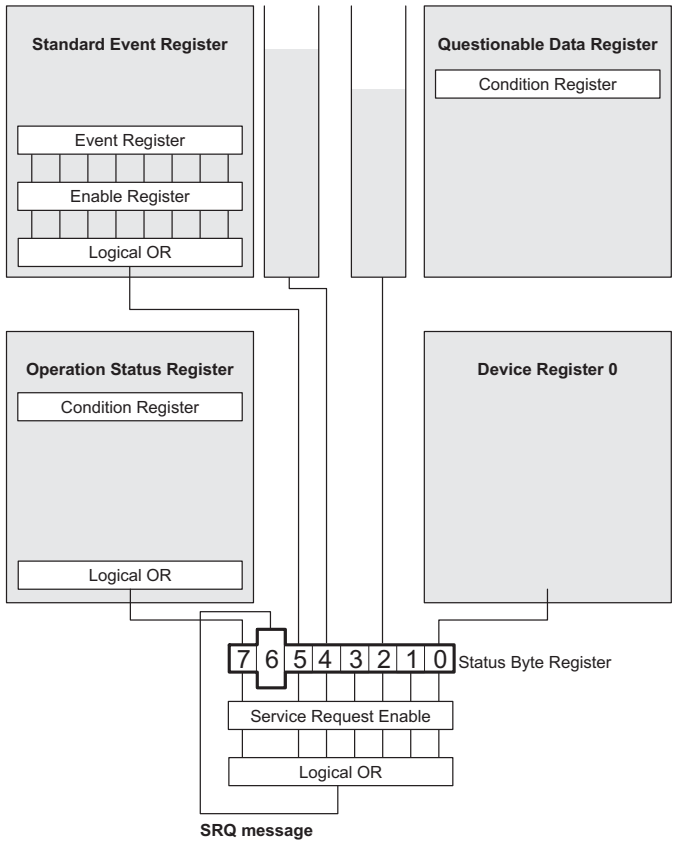


Figure 3-11 *Model '90' status register structure.*

Error Reporting

The counter will place a detected error in its Error Queue. This queue is a FIFO (First-In First-Out) buffer. When you read the queue, the first error will come out first, the last error last.

If the queue overflows, an overflow message is placed last in the queue, and further errors are thrown away until there is room in the queue again.

■ Detecting Errors in the Queue

Bit 2 in the Status Byte Register shows if the instrument has detected errors. It is also possible to enable this bit for Service Request on the GPIB. This can then interrupt the GPIB controller program when an error occurs.

■ Read the Error/Event Queue

This is done with the :SYSTem:ERRor? query.

Example:

```
SEND→ :SYSTem:ERRor?
READ← -100,␣"Command␣Error"
```

The query returns the error number followed by the error description.

Further description of all error numbers can be found in the Error Messages chapter

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 the following:

–350, "Queue overflow"

If more errors occur, they will be discarded.

■ Standardized Error Numbers

The instrument reports four classes of standardized errors in the Standard Event Status and in the Error/Event Queue as shown in the following table:

Error Class	Range of Error Numbers	Standard Event Register
Command Error	–100 to –199	bit 5 - CME
Execution Error	–200 to –299	bit 4 - EXE
Device Specific Error	–300 to –399 +100 to +32767	bit 3 - DDE
Query Error	–400 to –499	bit 2 -QYE

■ Command Error

This error shows that the instrument detected a syntax error.

■ Execution Error

This error shows that the instrument has received a valid program message which it cannot execute because of some device specific conditions.

■ Device-specific Error

This error shows that the instrument could not properly complete some device specific operations.

■ Query Error

This error will occur when the Message Exchange Protocol is violated, for example, when you send a query to the instrument and then send a new command without first reading the response data from the previous query. Also, trying to read data from the instrument without first sending a query to the instrument will cause this error.

Initialization and Resetting

Reset Strategy

There are three levels of initialization:

– Bus initialization

– Message exchange initialization

– Device initialization

■ Bus Initialization

This is the first level of initialization. The controller program should start with this, which initializes the IEEE-interfaces of all connected instruments. It puts the complete system into remote enable (REN-line active) and the controller sends the interface clear (IFC) command. The command or the command sequence for this initialization is controller and language dependent. Refer to the user manual of the system controller in use.

■ Message Exchange Initialization

Device clear is the second level of initialization. It initializes the bus message exchange, but does not affect the device functions.

Device clear can be signaled either with DCL to *all* instruments or SDC (Selective device-clear) only to the addressed instruments. The instrument action on receiving DCL and SDC is identical, they will do the following:

– Clear the input buffer.

– Clear the output queue.

– Reset the parser.

– Clear any pending commands.

The device-clear commands will not do the following:

– Change the instrument settings or stored data in the instrument.

– Interrupt or affect any device operation in progress.

– Change the status byte register other than clearing the MAV bit as a result of clearing the output queue.

Many older IEEE-instruments, that are not IEEE-488.2 compatible returned to the power-on default settings when receiving a device-clear command. IEEE-488.2 does not allow this.

When to use a Device-clear Command

The command is useful to escape from erroneous conditions without having to alter the current settings of the instrument. The instrument will then discard pending commands and will clear responses from the output queue. For example; suppose you are using the Counter in an automated test equipment system where the controller program returns to its main loop on any error condition in the system or the tested unit. To ensure that no unread query response remains in the output queue and that no unparsed message is in the input buffer, it is wise to use device-clear. (Such remaining responses and commands could influence later commands and queries.)

■ Device Initialization

The third level of initialization is on the device level. This means that it concerns only the addressed instruments.

*The *RST Command*

Use this command to reset a device. It initializes the device-specific functions in the Counter.

The following happens when you use the *RST command:

– You set the Counter-specific functions to a known default state. The *RST condition for each command is given in the command reference chapters.

– You disable macros.

– You set the counter in an idle state (outputs are disabled), so that it can start new operations.

*The *CLS Command*

Use this command to clear the status data structures. See ‘Status Reporting system’ in this chapter.

The following happens when you use the *CLS command:

– The instrument clears all event registers summarized in the status byte register.

– It empties all queues, which are summarized in the status byte register, except the output queue, which is summarized in the MAV bit.

Chapter 4

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 Windows NT and later operating systems.

Even if you use other platforms for your applications, you can benefit from studying the examples. They give you a good insight into programming the instrument efficiently.

To be able to run these programs without modification, the address of your counter must be set to 10.

Example 1: Individual Measurements

Example 2: Block Measurements

Example 3: Fast Measurements

Example 4: USB Communication

For your convenience the examples can also be found on the included manual CD. You are at liberty to copy them for educational purposes.

Individual Mesurements (Ex. #1)

Sample program to perform individual measurements on the '90'.
Written for National AT-GPIB/TNT for Windows NT and later.

```
/*
**
Sample program to perform individual measurements on the '90'.
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 counter, const char *string);

void sleep (long mswait);

void main() {
     int address = 10;
     int i, counter; /* file descriptor for counter */
     char reading[50];
     char   buf[100];

     printf ("Connecting to the '90' on address %d using
National Instruments GPIB card.\n", address);
     if ((counter = ibdev(0, address, 0, T10s, 1, 0)) < 0) {
          printf("Could not connect to counter");
          exit(1);
     }
     ibclr(counter);

     do {
          ibwrite(counter, "syst:err?");
          ibrd(counter, buf, 100L); buf[ibcnt]=0;
          printf("Errors before start: %s\n", buf);
     } while (atoi(buf)!=0);

     ibwrite(counter, "*idn?");
     ibrd(counter, buf, 100L); buf[ibcnt]=0;
     printf("Counter identification string: %s\n", buf);
     printf("Setup\n");
   // Reset counter to known state
   ibwrite(counter, "*rst;*cls");
```

```
   // Setup for pulse width measurement
   ibwrite(counter, "CONF:PWID (@1)");

   // Some settings...
   ibwrite(counter, "AVER:STAT OFF;:ACQ:APER MIN");
   ibwrite(counter, "INP:LEV:AUTO OFF; :INP:LEV 0");
   ibwrite(counter, "FORMAT:TINF ON;:FORMAT ASCII");
   // Check that setup was OK, all commands correctly spelled
etc
   ibwrite(counter, "syst:err?");
   ibrd(counter, buf, 100L);  buf[ibcnt]=0;
   printf("Setup error: %s\n", buf);

   // Measure 20 samples
   for (i=0; i<20; i++) {
     ibwrite(counter, "READ?");
     ibrd(counter, reading, 49L); reading[ibcnt]=0;
     printf("Result %d:%s", i, reading);
   }
do {
     ibwrite(counter, "syst:err?");
     ibrd(counter, buf, 100L); buf[ibcnt]=0;
     printf("End error: %s\n", buf);
     } while (atoi(buf)!=0);
     ibonl(counter, 0);
}
/********************
* Support functions *
********************/
void ibwrite(int counter, const char *string) {
     ibwrt(counter, (char*) string, strlen(string));
}

void sleep (long mswait) {
     time_t EndWait = clock() + mswait * (CLOCKS_PER_SEC/1000);
     while (clock() < EndWait);
}
```

Block Measurements (Ex. #2)

Sample program to perform fast measurements on the '90' using block measurements. Written for National AT-GPIB/TNT for Windows NT and later.

```
/*
**
** Sample program to perform fast measurements on the '90'
** 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 counter, const char *string);

void sleep (long mswait);

time_t StartMain, Start, Stop, StopMain;

void main() {
     int address = 10;
     int i, j, counter; /* file descriptor for counter */
     char bigbuf[30000], *pbuf;     char buf[100];
     char   Status;

     printf ("Connecting to the '90' on address %d using
National Instruments GPIB card.\n", address);
     if ((counter = ibdev(0, address, 0, T10s, 1, 0)) < 0) {
          printf("Could not connect to counter");
          exit(1);
     }
     ibclr(counter);

     do {
          ibwrite(counter, "syst:err?");
          ibrd(counter, buf, 100L); buf[ibcnt]=0;
          printf("Errors before start: %s\n", buf);
     } while (atoi(buf)!=0);

     ibwrite(counter, "*idn?");
     ibrd(counter, buf, 100L); buf[ibcnt]=0;   printf("Counter
identification string: %s\n", buf);
     printf("Setup\n");
```

```
      // Reset counter to known state      ibwrite(counter,
"*rst;*cls");

      // Setup for period measurement      ibwrite(counter, "FUNC
'PER 1'");

      // Some settings...
      ibwrite(counter, "INP:LEV:AUTO OFF;:INP:LEV 0;COUP DC");
ibwrite(counter, "TRIG:COUNT 1000;:ARM:COUNT 1");
      ibwrite(counter, "DISP:ENAB ON");    ibwrite(counter,
"FORMAT ASCII;:FORMAT:TINF OFF");
      ibwrite(counter, "*ESE 1;*SRE 32");

      // On the safe side: Check that setup was OK, all commands
correctly spelled etc
      ibwrite(counter, "syst:err?");
      ibrd(counter, buf, 100L);  buf[ibcnt]=0;  printf("Setup
error: %s\n", buf);

      // Measure 1000 samples
      Start = clock();
      ibwrite(counter, "INIT;*OPC");

      // Wait for completion
     ibwait(counter, RQS);

      /* Read status and event registers to clear them */
      ibrsp(counter, &Status);
      ibwrite(counter, "*ESR?");
      ibrd(counter, buf, 100L);

      ibwrite(counter, "FETC:ARR? 1000");

      ibrd(counter, 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(counter, "syst:err?");
          ibrd(counter, buf, 100L); buf[ibcnt]=0;
          printf("End error: %s\n", buf);
     } while (atoi(buf)!=0);

     ibonl(counter, 0);
}
/********************
* Support functions *
********************/

void ibwrite(int counter, const char *string) {
          ibwrt(counter, (char*) string, strlen(string));
}
void sleep (long mswait) {
          time_t EndWait = clock() + mswait *
(CLOCKS_PER_SEC/1000);
          while (clock() < EndWait);
}
```

Fast Measurements (Ex. #3)

Sample program to perform fast measurements on the '90' 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 '90'
** 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 counter, 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, counter;    /* file descriptor for counter */
     char reading[30];
     char buf[100];
     r2d Result;

     printf ("Connecting to the '90' on address %d using
National Instruments GPIB card.\n", address);
     if ((counter = ibdev(0, address, 0, T10s, 1, 0)) < 0) {
          printf("Could not connect to counter");
          exit(1);
     }
     sleep(100);
     ibclr(counter);
     sleep(100);
```

```
     ibwrite(counter, "*idn?");
     ibrd(counter, buf, 100L); buf[ibcnt]=0;
     printf("Counter identification string: %s\n", buf);

     printf("Setup\n");
   if ((counter = ibdev(0, address, 0, T3s, 1, 0)) < 0) {
          printf("Could not connect to counter");
          exit(1);
     }
     // Reset counter to known state
     ibwrite(counter, "*rst;*cls");
     ibwrite(counter, "*ESE 0; *SRE 0");

     // Setup for frequency measurement
     ibwrite(counter, "FUNC 'per 1'");

     // Some settings...
     ibwrite(counter, "INP:LEV:AUTO OFF;:INP:LEV .5;:inp:coup
dc");
     ibwrite(counter, "TRIG:COUNT 1;:ARM:COUNT 1");
     ibwrite(counter, "ACQ:APER 1e-7");

     ibwrite(counter, "DISP:ENAB OFF");   // Disable display to
get maximum speed
     ibwrite(counter, "FORMAT REAL;:FORMAT:TINF OFF");
// Floating point output, no timestamps

     ibwrite(counter, "FORMAT:BORDER swap");
// Intel byte order on results

     ibwrite(counter, "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(counter, "syst:err?");
          ibrd(counter, 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(counter);      // Generate GET signal
```

```
            ibrd(counter, 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(counter, "DISP:ENAB ON");
      do {
            ibwrite(counter, "syst:err?");
            ibrd(counter, buf, 100L); buf[ibcnt]=0;
            printf("End error: %s\n", buf);
      } while (atoi(buf)!=0);

      ibonl(counter, 0);
}

/********************
* Support functions *
********************/

void ibwrite(int counter, const char *string) {
            ibwrt(counter, string, strlen(string));
}

void sleep (long mswait) {
            time_t EndWait = clock() + mswait *
(CLOCKS_PER_SEC/1000);
            while (clock() < EndWait);
}
```

USB Communication (Ex. #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 something made by Pendulum
    Status = viFindRsrc(defaultRM,
                        "USB?*INSTR{VI_ATTR_MANF_ID==0x14EB}",
                        &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() )
        ;
}
```

Chapter 5

Instrument Model

Introduction

The figure below shows how the instrument functions are categorized. This instrument model is fully compatible with the SCPI generalized instrument model.

The generalized SCPI instrument model, contains three major instrument categories as shown in the following table:

Function	Instrument Type	Examples
Signal acquisition	Sense instruments	Voltmeter, Oscilloscope, Counter
Signal generation	Source instruments	Pulse generator, Power supply
Signal routing	Switch instruments	Scanner, (de)-multiplexer

An instrument may use a combination of the above functions. The counters belong to the signal acquisition category, and only that category is described in this manual.

The instrument model in defines where elements of the counter language are assigned in the command hierarchy. The major signal function areas are shown broken into blocks. Each of these blocks are major command sub-trees in the counter command language.

The instrument model also shows how measurement data and applied signals flow through the instrument. The model does not include the administrative data flow associated with queries, commands, performing calibrations etc.

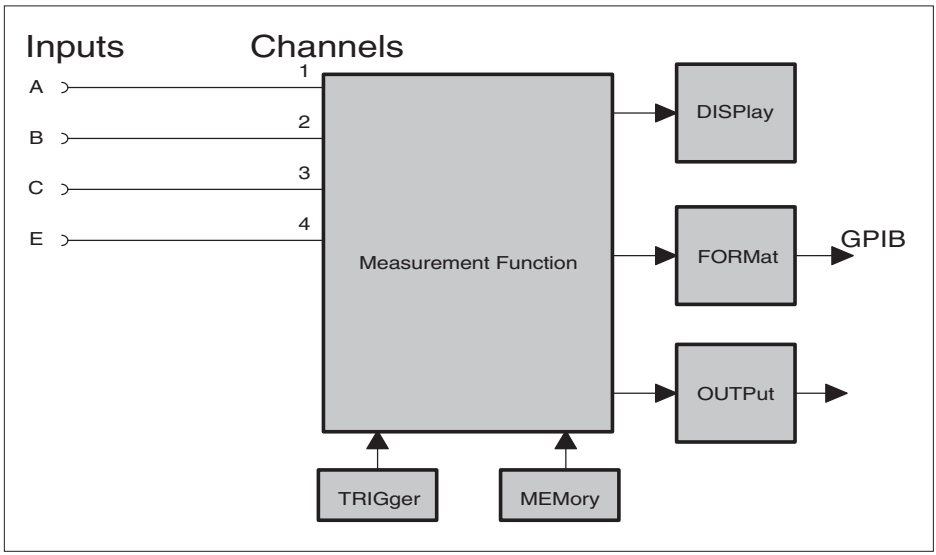


Figure 5-1 *Model '90' instrument model.*

Measurement Function Block

The measurement function block converts the input signals into an internal data format that is available for formatting into GPIB bus data. The measurement function is divided into three different blocks: INPut, SENSe and CALCulate. See .

■ INPut

The INPut block performs all the signal conditioning of the input signal before it is converted into data by the SENSe block. The INPut block includes coupling, impedance, filtering etc.

■ SENSe

The SENSe block converts the signals into internal data that can be processed by the CALCulate block. The SENSe commands control various characteristics of the measurement and acquisition process. These include: gate time, measurement function, resolution, etc.

■ CALCulate

The CALCulate block performs all the necessary calculations to get the required data. These calculations include: calibration, statistics, mathematics, etc.

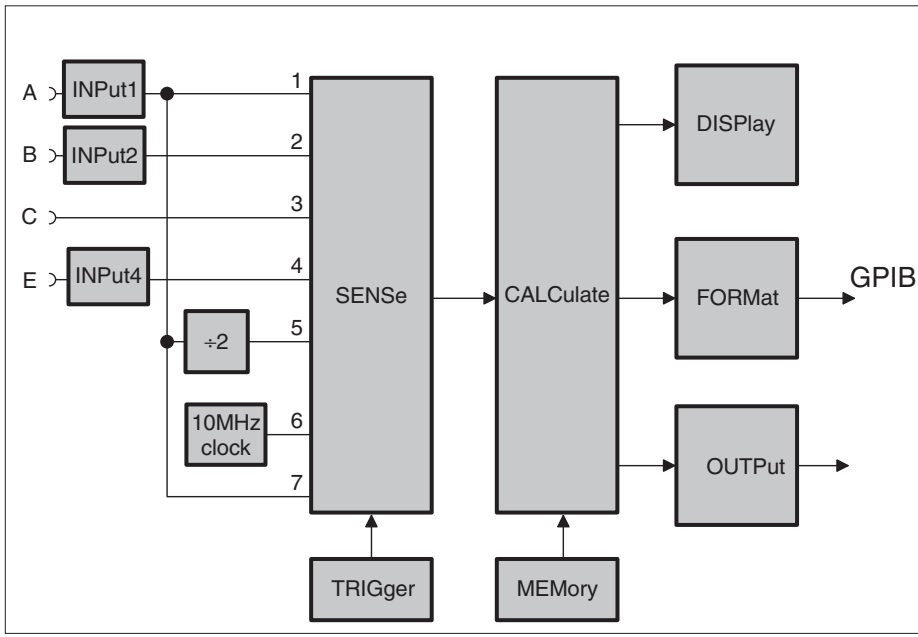


Figure 5-2 *Model '90' measurement model.*

Other Subsystems

In addition to the major functions (subsystems), there are several other subsystems in the instrument model.

The different blocks have the following functions.

■ CALibration

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

■ DISPlay

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

■ FORMat

The FORMat block 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.

■ MEMory

The MEMory block holds macro and instrument state data inside the counter.

■ STATus

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

■ Synchronization

This subsystem can be used to synchronize the measurements with the controller.

■ SYSTem

This subsystem controls some system parameters like timeout.

■ TEST

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

■ TRIGger

The trigger block provides the counter with synchronization capability with external events. Commands in this block control the trigger and arming functions of the Timer/ Counter.

Order of Execution

– All commands in the counters are sequential, i.e., they are executed in the same order as they are received.

– If a new measurement command is received when a measurement is already in progress, the measurement in progress will be aborted unless ✷`WAI` is used before the command.

MEASurement Function

In addition to the subsystems of the instrument model, which control the instrument functions, SCPI has signal-oriented functions to obtain measurement results. This group of MEASure functions has a different level of compatibility and flexibility. 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. See .

■ MEASure?

This is the most simple command to use, but it does not offer much flexibility. The `MEASure?` query lets the counter configure itself for an optimal measurement, start the data acquisition, and return the result.

■ CONFigure; READ?

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

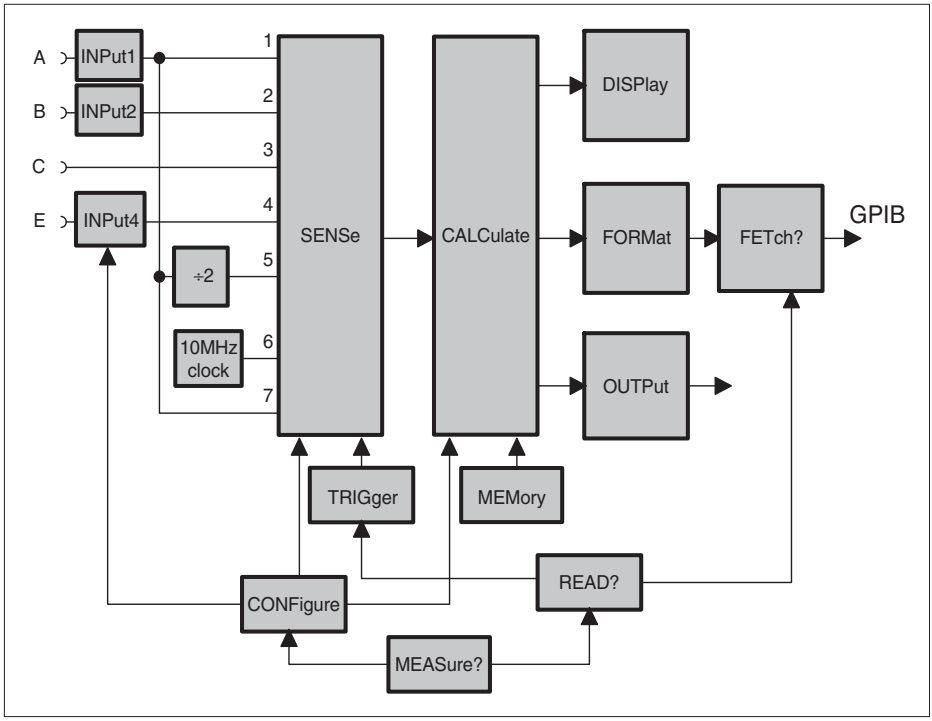


Figure 5-3 *Model '90' measurement function.*

This sequence does the same as the MEASure command, but now it is possible to insert commands between CONFigure and READ? to adjust the setting of a particular function (called fine tuning). For instance, you can set an input attenuator at a required value.

■ 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.

Versatility of Measurement Commands	
MEASure?	Simple to use; few additional possibilities.
CONFigure READ?	Somewhat more difficult; some extra possibilities.
CONFigure INITiate FETCh?	Most difficult to use; many extra features.

Chapter 6

Using the Subsystems

Introduction

Although SCPI is intended to be self explanatory, we feel that some hints and tips on how to use the different subsystems may be useful. This chapter does not explain each and every command, but only those for which we believe extra explanations are necessary.

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 counter itself monitors the result and alerts you when the limits are exceeded.

■ Mathematics

The mathematical functions are the same as on the front panel.

■ Statistics

The '90' can calculate and display the *MIN, MAX, MEAN* and *standard deviation* of a given number of samples. The statistical functions are the same as on the front panel.

■ Limit Monitoring

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. The bits are enabled 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/off will not influence the status register mask bits which determine whether or not a service request will be generated 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.

Configure Function

The CONFigure command sets up the counter 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.

Read more about Configure under MEASure.

Format Subsystem

Time Stamp Readout Format

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

The first will be the measured value, expressed in the basic unit of the measurement function, and the next one will be the timestamp value in seconds.

In :FORMat ASCii mode, the result will be given as a floating-point number, followed by a floating point timestamp value.

In :FORMat REAL mode, the result will be 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 will consist of alternating measurement values and timestamp values, formatted in a similar way as for scalar readout. All values will be separated by commas.

An overview of the different output formats can be gained by studying Chapter 8, Command Reference.

See the following subdivisions with page references:

P. 8-29 ff. — Fetch Function
P. 8-33 ff. — Format Subsystem
P. 8-66 — Measurement Function

Input Subsystems

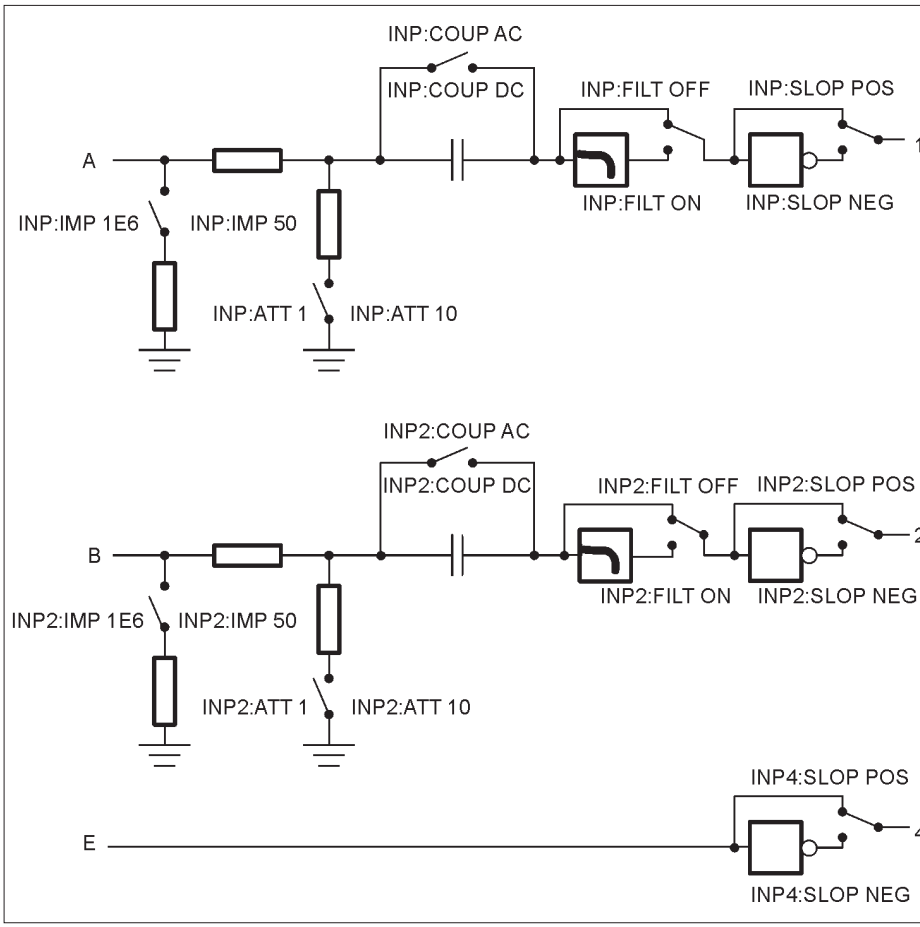


Figure 6-1 *Summary of Model 90 input amplifier settings.*

Measurement Function

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.

- **Example:**

SEND→ `MEASure:FREQ?`

This will execute a frequency measurement and the result will be 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→ `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. (1 Hz)

Also the channel numbers can be specified, for example:

SEND→ `MEASure:FREQ?␣(@3)`
SEND→ `MEASure:FREQ?␣20E6,`
`1,(@1)`

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 1 MΩ to 50 Ω.

■ **Example:**

SEND→ `CONFigure:FREQ␣2E6,1`

2E6 is the expected value
1 is the required resolution (1Hz)

SEND→ `INPut:IMPedance␣50`

Sets input impedance to 50 Ω

SEND→ `READ?`

Starts 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.

■ **Example:**

SEND→ `CONFigure:FREQ␣20E6,1`

20E6 is the expected signal value
1 is the required resolution

SEND→ `INPut:IMPedance␣1E6`

Sets input impedance to 1 MΩ

SEND→ `INITiate`

Starts measurement

SEND→ `FETCh?`

Fetches the result

Versatility of measurement commands	
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.

Sense Command Subsystems

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. With the '90' you can select from two different basic inputs (A and B), on which the counter can measure 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 counter has a max repetition rate of 160 MHz.

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:PREScaler` is set to ON. This is also the default condition.

Status Subsystem

Introduction

Status reporting is a method to let the controller know what the counter is doing. You can ask the counter what status it is in whenever you want to know.

You can select some conditions in the counter that should be reported in the Status Byte Register. You can also select if some bits in the Status Byte should generate a Service Request (SRQ).
(An SRQ is the instrument's way to call the controller for help.)

Status Reporting Model

■ The Status Structure

The status reporting model used is standardized in IEEE 488.2 and SCPI, so you will find similar status reporting in most modern instruments. Figure 6-6 shows an overview of the complete status register structure. It has four registers, two queues, and a status byte:

– The **Standard Event Register** reports the standardized IEEE 488.2 errors and conditions.

– The **Operation Status Register** reports the status of the measurement cycle (see also ARM-TRIG model, page -).

– The **Questionable Data Register** reports when the output data from the counter may not be trusted.

– The **Device Register 0** reports when the measuring result has exceeded preprogrammed limits.

– The **Output Queue status** reports if there is output data to be fetched.

– The **Error Queue status** reports if there are error messages available in the error queue.

– The **Status Byte** contains eight bits. Each bit shows if there is information to be fetched in the above described registers and queues of the status structure.

Using the Registers

Each status register monitors several conditions at once. If something happens to any one of the monitored conditions, a summary bit is set true in the Status Byte Register.

Enable registers are available so that you can select what conditions should be reported in the status byte, and what bits in the status byte should cause SRQ.

A register bit is TRUE, i.e., something has happened, when it is set to 1. It is FALSE when set to 0.

Note that all event registers and the status byte record positive events. That is when a condition changes from inactive to active, the bit in the event register is set true. When the condition changes from active to inactive, the event register bits are not affected at all.

When you read the contents of a register, the counter answers with the decimal sum of the bits in the register.

Example:

The counter answers 40 when you ask for the contents of the Standard Event Status Register.

– Convert this to binary form. It will give you 101000.

– Bit 5 is true showing that a command error has occurred.

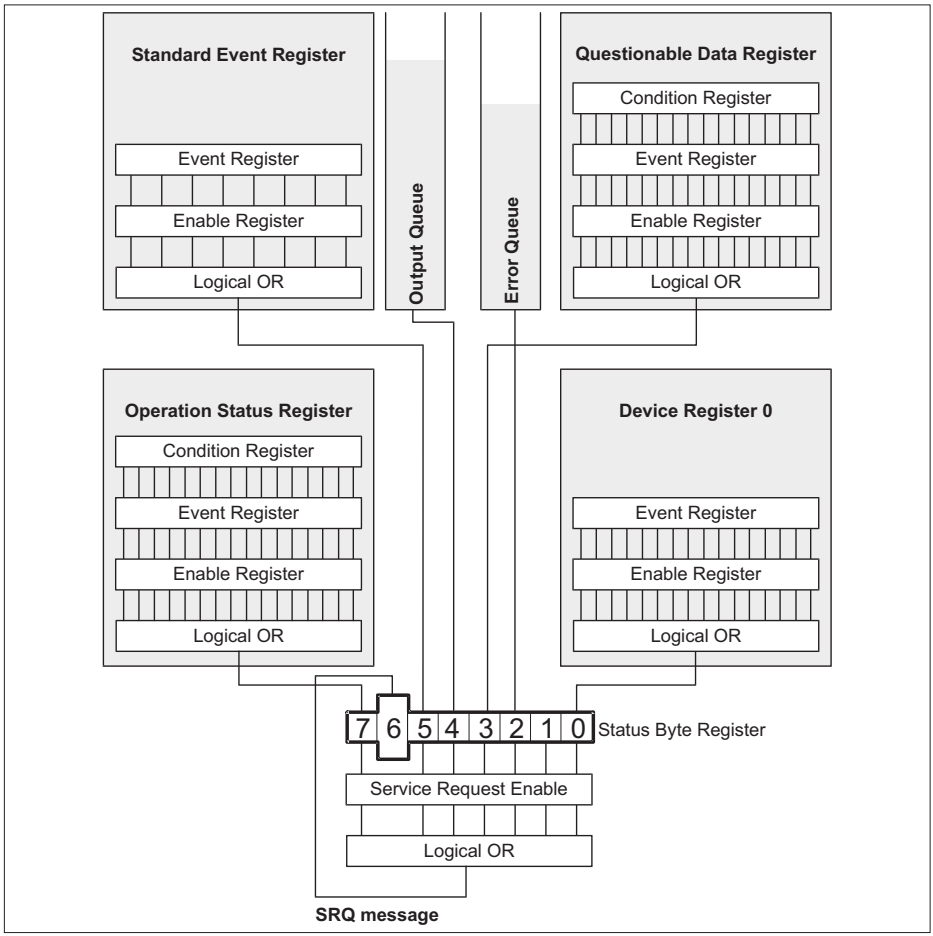


Figure 6-2 *Model 90 status register structure.*

– Bit 3 is also true, showing that a device dependent error has occurred.

Use the same technique when you program the enable registers.

– Select which bits should be true.

– Convert the binary expression to decimal data.

– Send the decimal data to the instrument.

Clearing/Setting all bits

– You can clear an enable register by programming it to zero. You can set all bits true in a 16-bit event enable register by programming it to 32767 (bit 16 not used).

– You set all bits true in 8-bit registers by programming them to 255 (Service Request Enable and Standard Event Enable.)

■ Using the Queues

The two queues, where the counter stores output data and error messages, may contain data or be empty. Both these queues have their own status bit in the Status Byte. If this bit is true there is data to be fetched.

When the controller reads data, it will also remove the data from the queue. The queue status bit in the status byte will remain true for as long as the queue holds

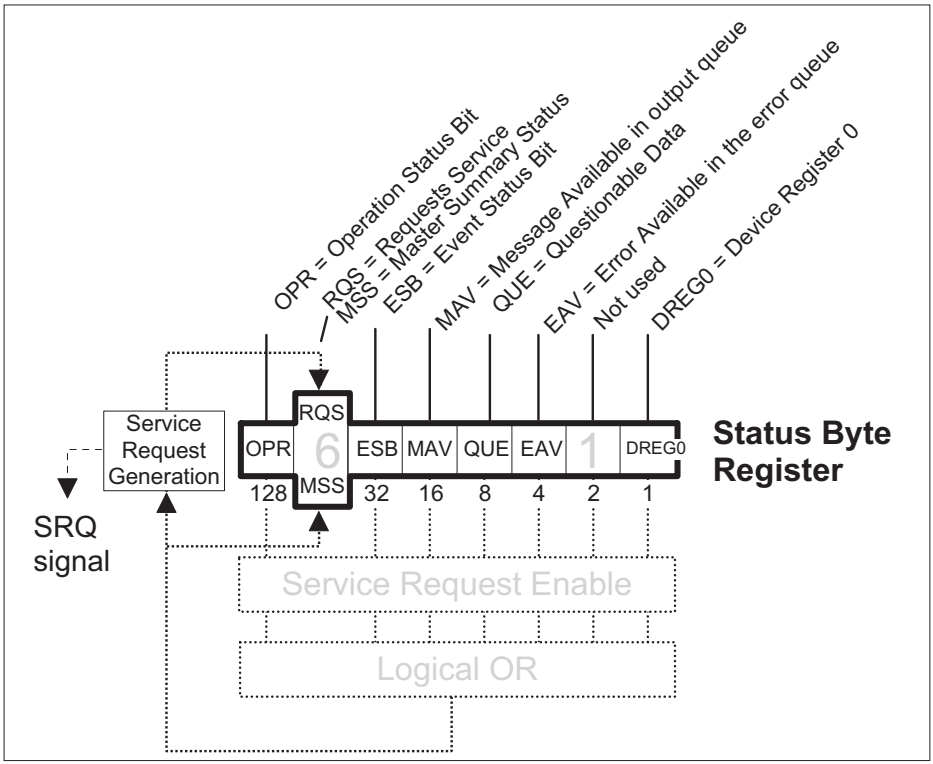


Figure 6-3 *The status byte bits.*

one or more data bytes. When the queue is empty, the queue status bit is set false.

Status of the Output Queue (MAV)

The MAV (message available) queue status message appears in bit 4 of the status byte register. It indicates if there are bytes ready to be read over the GPIB in the GPIB output queue of the instrument. The output queue is where the formatted data appears before it is transferred to the controller.

The controller reads this queue by addressing the instrument as a talker. The command to do this differs between different programming languages. Examples are IOENTERS and IBREAD.

Status of the Error Message Queue (EAV)

The EAV (error message available) queue status message appears in bit 2 of the status byte register. Use the `:SYSTem:ERRor?` query to read the error messages. Chapter 7 explains all possible error messages .

■ Using the Status Byte

The status byte is an eight bit status message. It is sent to the controller as a response to a serial poll or a *STB? query, see . Each bit in the status byte contains a summary message from the status structure. You can select what bits in the status byte should generate a service request to alert the controller.

When a service request occurs, the SRQ-line of the GPIB will be activated. Whether or not the controller will react on the service request depends on the controller program. The controller may be interrupted on occurrence of a service request, it may regularly test the SRQ-line, it may regularly make serial poll or *STB?, or the controller may not react at all. The preferred method is to use SRQ because it presents a minimum of disturbance to the measurement process.

Selecting Summary Message to Generate SRQ

The counter does not generate any SRQ by default. You must first select which summary message(s) from the status byte register should give SRQ. You do that with the Service Request Enable command `*SRE <bit mask>`.

Example:

```
*SRE␣16
```

This sets bit 4 ($16=2^4$) in the service request enable register (see). This makes the instrument signal SRQ when a message is available in the output queue.

RQS/MSS

The original status byte of IEEE 488.1 is sent as a response to a serial poll, and bit 6 means requested service, RQS.

IEEE 488.2 added the `*STB?` query and expanded the status byte with a slightly different bit 6, the MSS. This bit is true as long as there is unfetched data in any of the status event registers.

– The Requested Service bit, RQS, is set true when a service request has been signalled. If you read the status byte via a Serial Poll, bit 6 represents RQS. Reading the status byte with a serial poll will set the RQS bit false, showing that the status byte has been read.

– The Master Summary Status bit, MSS, is set true if any of the bits that generates SRQ is true. If you read the status byte us-

ing *STB?, bit 6 represents MSS. MSS remains true until all event registers are cleared and all queues are empty.

Setting up the Counter to Report Status

Include the following steps in your program when you want to use the status reporting feature.

– *CLS Clears all event registers and the error queue

– *ESE <bit mask> Selects what conditions in the Standard Event Status register should be reported in bit 5 of the status byte

– :STATus:OPERation:ENABle <bit mask> Selects which conditions in the Operation Status register should be reported in bit 7 of the status byte

– :STATus:QUEStionable:ENABle <bit mask> Selects which conditions in the Questionable Status register should be reported in bit 3 of the status byte

– :STATus:DREGister0:ENABle <bit mask> Selects which conditions in Device Register 0 should be reported in bit 0 of the status byte

– *SRE <bit mask> Selects which bits in the status byte should cause a Service Request

A programming example using status reporting is available in Chapter 7.

Reading and Clearing Status

■ Status Byte

As explained earlier, you can read the status byte register in two ways:

Using the Serial Poll (IEEE-488.1 defined).

– Response:

  – Bit 6: RQS message, shows that the counter has requested service via the SRQ signal.

  – Other bits show their summary messages

  – A serial poll sets the RQS bit FALSE, but does not change other bits.

*Using the Common Query *STB?*

– Response:

  – Bit 6: MSS message, shows that there is a reason for service request.

  – Other bits show their summary messages.

  – Reading the response will not alter the status byte.

■ Status Event Registers

You read the Status Event registers with the following queries:

– *ESR? Reads the Standard Event Status register

– :STATus:OPERation? Reads the Operation Status Event register

– :STATus:QUEStionable? Reads the Questionable Status Event register

– :STATus:DREGister0? Reads the Device Event register

When you read these registers, you will clear the register you read and the summary message bit in the status byte.

You can also clear all event registers with the `*CLS` (Clear Status) command.

■ Status Condition Registers

Two of the status register structures also have condition registers: The Status Operation and the Status Questionable register.
The condition registers differ from the event registers in that they are not latched. That is, if a condition in the counter goes on and then off, the condition register indicates true while the condition is on and false when the condition goes off. The Event register that monitors the same condition continues to indicate true until you read the register.

– `:STATus:OPERation:CONDition?` Reads the Operation Status Condition register

– `:STATus:QUEStionable:CONDition?` Reads the Questionable Status Condition register

Reading the condition register will not affect the contents of the register.

Why Two Types of Registers?

Let's say that the counter measures continuously and you want to monitor the measurement cycle by reading the Operation Status register.

Reading the Event Register will always show that a measurement has started, that waiting for triggering and bus arming has occurred and that the measurement is stopped. This information is not very useful.

Reading the Condition Register on the other hand gives *only* the status of the measurement cycle, for instance "Measurement stopped".

Although it is possible to read the condition registers directly, we recommend that you use SRQ when monitoring the measurement cycle. The measurement cycle is disturbed when you read condition registers.

■ Summary:

The way to work when writing your bus program is as follows:

Set up

– Set up the enable registers so that the events you are interested in are summarized in the status byte.

– Set up the enable masks so that the conditions you want to be alerted about generate SRQ. It is good practice to generate SRQ on the EAV bit. So, enable the EAV-bit via `*SRE`.

Check & Action

– Check if an SRQ has been received.

  – Make a serial poll of the instruments on the bus until you find the instrument that issued the SRQ (the instrument that has RQS bit true in the Status Byte).

  – When you find it, check which bits in the Status Byte Register are true.

  – Let's say that bit 7, OPR, is true. Then read the contents of the Operation Status Register. In this register you can see what caused the SRQ.

  – Take appropriate actions depending on the reason for the SRQ.

Standard Status Registers

These registers are called the *standard* status data structure because they are mandatory in all instruments that fulfill the IEEE 488.2 standard.

■ Standard Event Status Register

Bit 7 (weight 128) — Power-on (PON)

Shows that the counter's power supply has been turned off and on (since the last time the controller read or cleared this register).

Bit 6 (weight 64)—User Request (URQ)

Shows that the user has pressed a key on the front panel. The URQ bit will be set regardless of the remote local state of the counter. The purpose of this signal is, for example, to call for the attention of the controller by generating a service request.

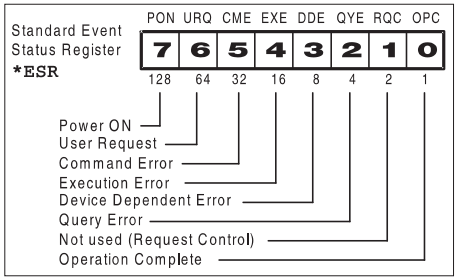


Figure 6-5 *Bits in the standard event status register*

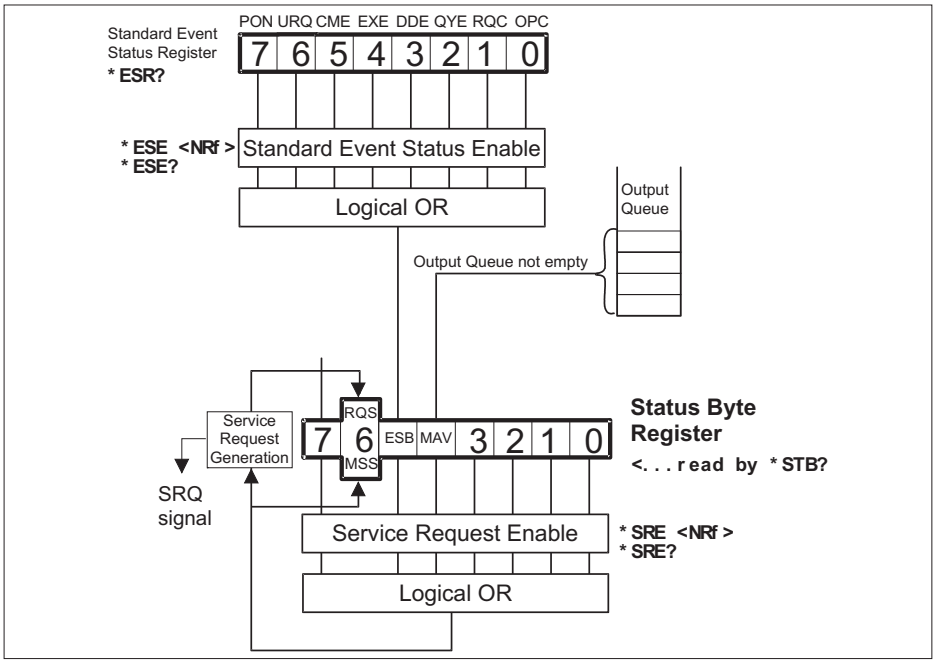


Figure 6-4 *Standard status data structures, overview.*

Bit 5 (weight 32) — Command Error (CME)

Shows that the instrument has detected a command error. This means that it has received data that violates the syntax rules for program messages.

Bit 4 (weight 16) — Execution Error (EXE)

Shows that the counter detected an error while trying to execute a command. (See 'Error reporting' on page 3-17.) The command is syntactically correct, but the counter cannot execute it, for example because a parameter is out of range.

Bit 3 (weight 8) — Device-dependent Error (DDE)

A device-dependent error is any device operation that did not execute properly because of some internal condition, for instance error queue overflow. This bit shows that the error was not a command, query or execution error.

Bit 2 (weight 4) — Query Error (QYE)

The output queue control detects query errors. For example the QYE bit shows the unterminated, interrupted, and deadlock conditions. For more details, see 'Error reporting' on page 3-17.

Bit 1 (weight 2)—Request Control (RQC)

Shows the controller that the device wants to become the active controller-in-charge. Not used in this counter.

Bit 0 (weight 1) — Operation Complete (OPC)

The counter only sets this bit TRUE in response to the operation complete command (*OPC). It shows that the counter has completed all previously started actions.

■ Summary, Standard Event Status Reporting

**ESE <bit mask>*

Enable reporting of Standard Event Status in the status byte.

**SRE 32*

Enable SRQ when the Standard Event structure has something to report.

ESR?

Reading and clearing the event register of the Standard Event structure.

SCPI-defined Status Registers

The counter has two 16-bit SCPI-defined status structures: The *operation status register* and the *questionable data register*. These are 16 bits wide, while the status byte and the standard status groups are 8 bits wide.

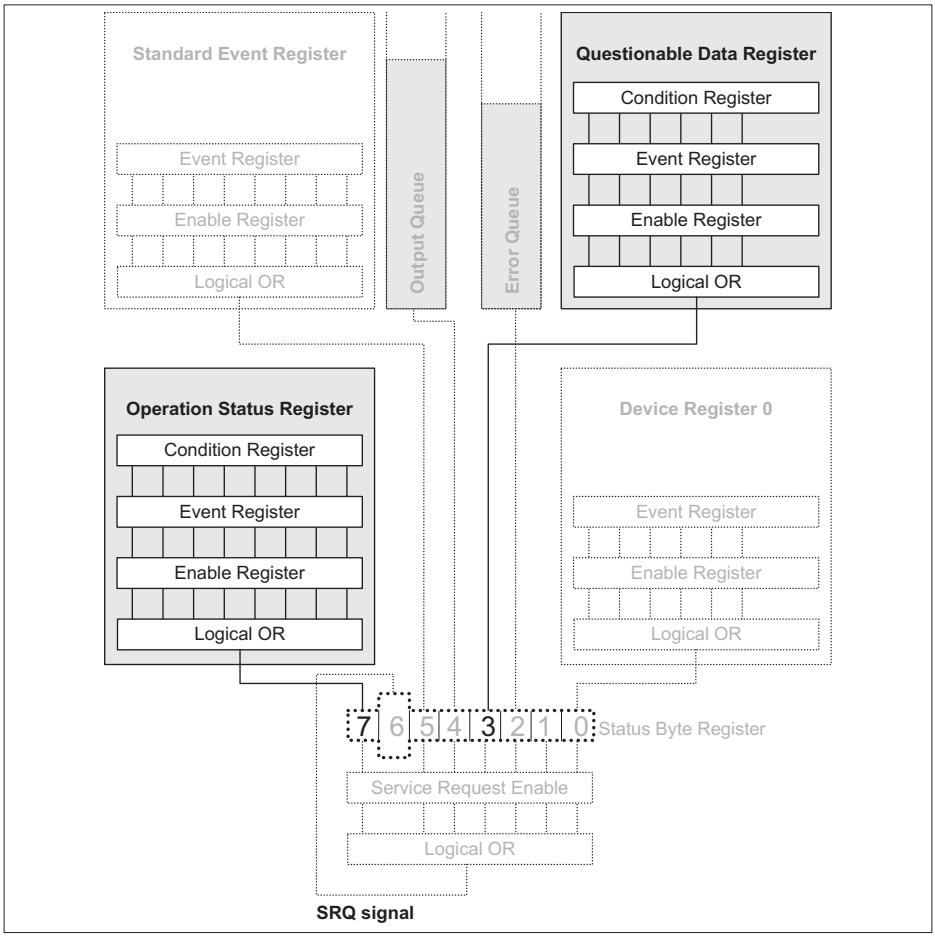


Figure 6-6 *Status structure 7, Operation Status Group, and Status structure 3, Questionable Data Group are SCPI defined.*

■ Operation Status Group

This group reports the status of the counter measurement cycle.

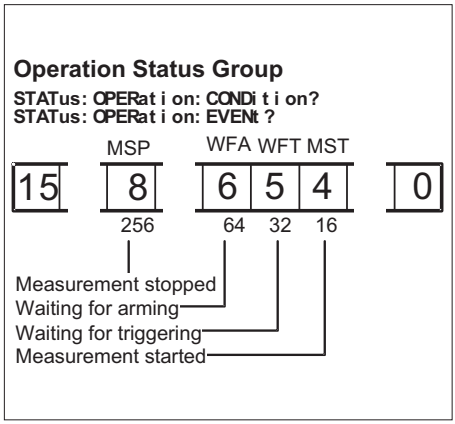


Figure 6-7 *Bits in the Opeation Status Register.*

Bit 8 (weight 256) — Measurement Stopped (MSP)

This bit shows that the counter is not measuring. It is set when the measurement, or sequence of measurements, stops.

Bit 6 (weight 64) — Wait for Bus Arming (WFA)

This bit shows that the counter is waiting for bus arming in the arm state of the trigger model.

Bit 5 (weight 32) — Waiting for Trigger and/or External Arming (WFT)

This bit shows when the counter is ready to start a new measurement via the trigger control option (488.2), for the shortest possible trigger delay. The counter is now in the wait for the trigger state of the trigger model.

Bit 4 (weight 16) — Measurement Started (MST)

This bit shows that the counter is measuring. It is set when the measurement or sequence of measurements starts.

■ Summary, Operation Status Reporting

:STAT:OPER:ENAB

Enable reporting of Operation Status in the status byte.

**SRE 128*

Enable SRQ when operation status has someting to report.

:STAT:OPER?

Reading and clearing the event register of the Operation Status Register structure

:STAT:OPER:COND?

Reading the condition register of the Operation Status Register structure.

Questionable Data/Signal Status Group

This group reports when the output data from the counter may not be trusted.

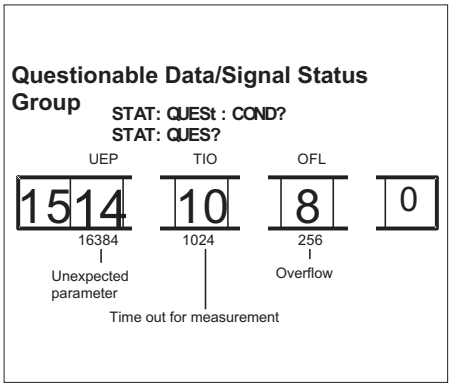


Figure 6-8 *Bits in Questionable data register.*

Bit 14 (weight 16384) — Unexpected Parameter (UEP)

This bit shows that the counter has received a parameter that it cannot execute, although the parameter is valid according to SCPI. This means that when this bit is true, the instrument has not performed a measurement exactly as requested.

Bit 10 (weight 1024) — Timeout for Measurement (TIO)

The counter sets this bit true when it abandons the measurement because the internal timeout has elapsed, or no signal has been detected.
See also `:SYST:TOUT` and `:SYST:SDET`.

Bit 8 (weight 256) Overflow (OFL)

The counter sets this bit true when it cannot complete the measurement due to overflow.

■ Summary, Questionable Data/Signal Status Reporting

:STAT:QUES:ENAB <bit mask>

Enable reporting of Questionable data/signal status in the status byte.

**SRE 8*

Enable SRQ when data/signal is questionable.

:STAT:QUES?

Reading and clearing the event register of the Questionable data/signal Register structure.

:STAT:QUES:COND?

Reading the condition register of Questionable data/signal Register structure.

Device-defined Status Structure

The counter has one device-defined status structure called the Device Register 0. It summarizes this structure in bit 0 of the status byte. Its purpose is to report when the measuring result has exceeded pre-programmed limits.

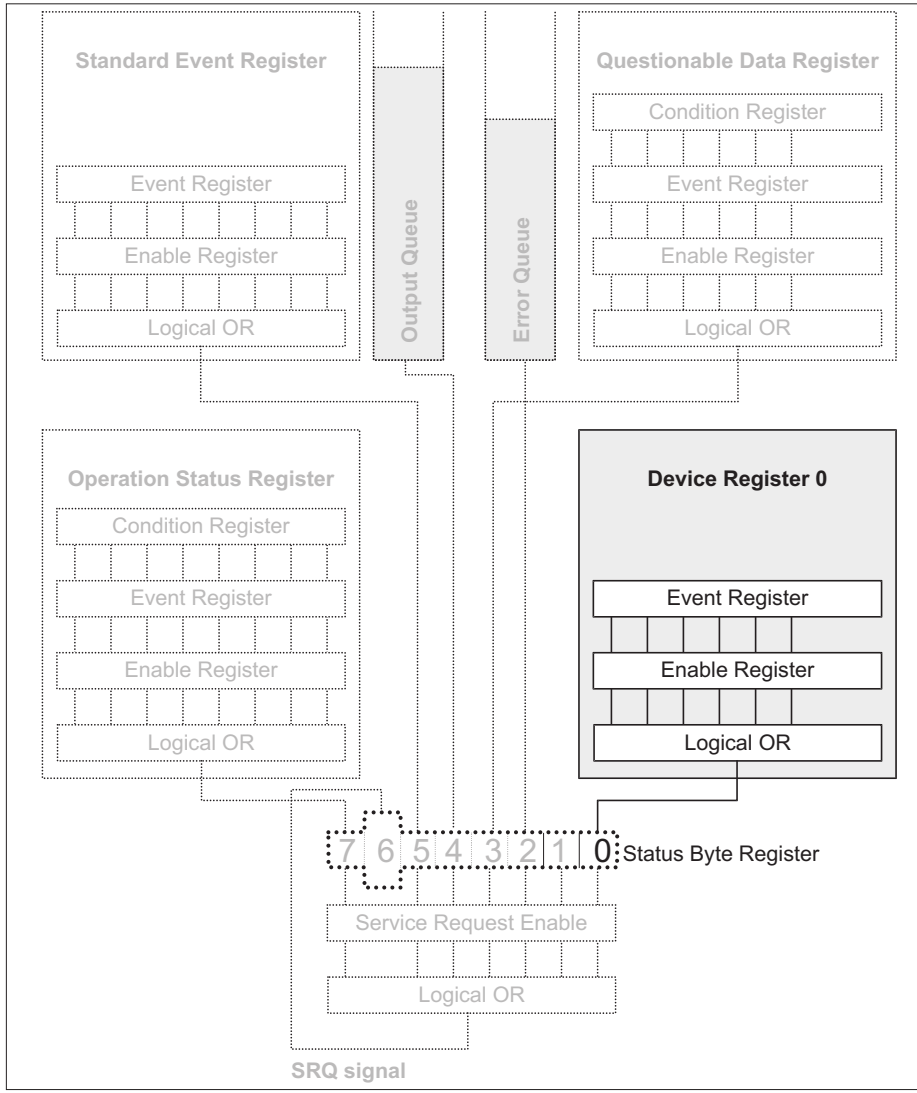


Figure 6-9 *Device-defined status data structures (model).*

You set the limits with the following commands in the calculate subsystem.
`:CALCulate:LIMit:UPPer` and
`:CALCulate:LIMit:LOWer`

Bit Definition

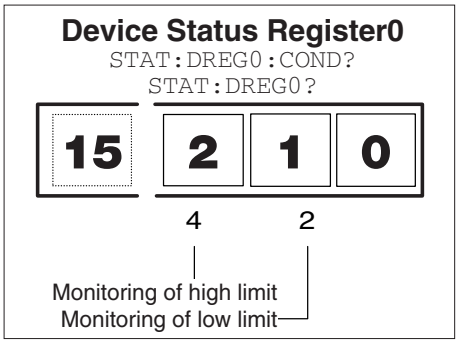


Figure 6-10 *Bits in the Device Status Register number 0.* `:STATus:DREGister0?` *Reads out the contents of the Device Status event Register 0 and clears the register.*

Bit 2 (weight 4) — Monitor of Low Limit

This bit is set when the low limit is passed from above.

Bit 1 (weight 2) — Monitor of High Limit

This bit is set when the high limit is passed from below.

■ Summary, Device-defined Status Reporting

:STAT:DREG0:ENAB <bit mask>

Enable reporting of device-defined status in the status byte.

**SRE 1*

Enable SRQ when a limit is exceeded.

:STAT:DREG0?

Reading and clearing the event register of Device Register structure 0.

– If bit 1 is true, the high limit has been exceeded.
– If bit 2 is true, the low limit has been exceeded.

Power-on Status Clear

Power-on clears all event enable registers and the service request enable register if the power-on status clear flag is set TRUE (see the common command `*PSC`.)

■ Preset the Status Reporting Structure

You can preset the complete status structure to a known state with a single command, the `STATus:PRESet` command, which does the following:

– Disables all bits in the Standard Event Register, the Operation Status Register, and the Questionable Data Register
– Enables all bits in Device Register 0
– Leaves the Service Request Enable Register unaffected.

Trigger/Arming Subsystem

The SCPI TRIGger subsystem enables synchronization of instrument actions with specified internal or external events. The following list gives some examples.

Instrument Action

Some examples of events to synchronize with are as follows:

- measurement
- bus trigger
- external signal level or pulse
- 10 occurrences of a pulse on the external trigger input
- other instrument ready
- signal switching
- input signal present
- 1 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.

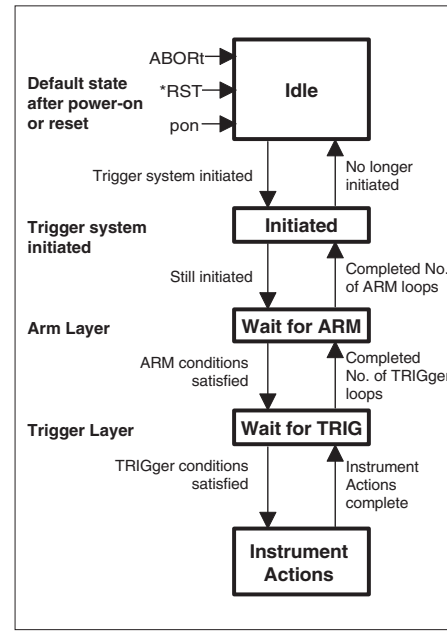


Figure 6-11 *Generalized ARM-TRIG model.*

This trigger configuration is sufficient for most instruments. More complex instruments, such as the '90', 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; see .

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.

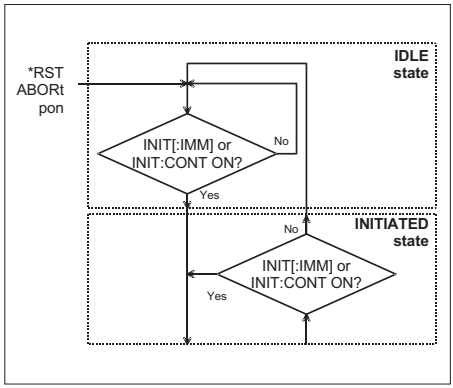


Figure 6-12 *Flow diagram of IDLE and INITIADED layers.*

■ Structure of an Event-detection Layer

The general structure of all event-detection layers is identical and is roughly depicted by the flow diagram in .

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 counters, 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 IEEE 488.1.

*When to use *TRG and GET*

The `*TRG` and the GET commands have the same effect on the instrument. If the Counter is in idle, i.e., not parsing or executing any commands, GET will execute much faster than `*TRG` since the instrument must always parse `*TRG`.

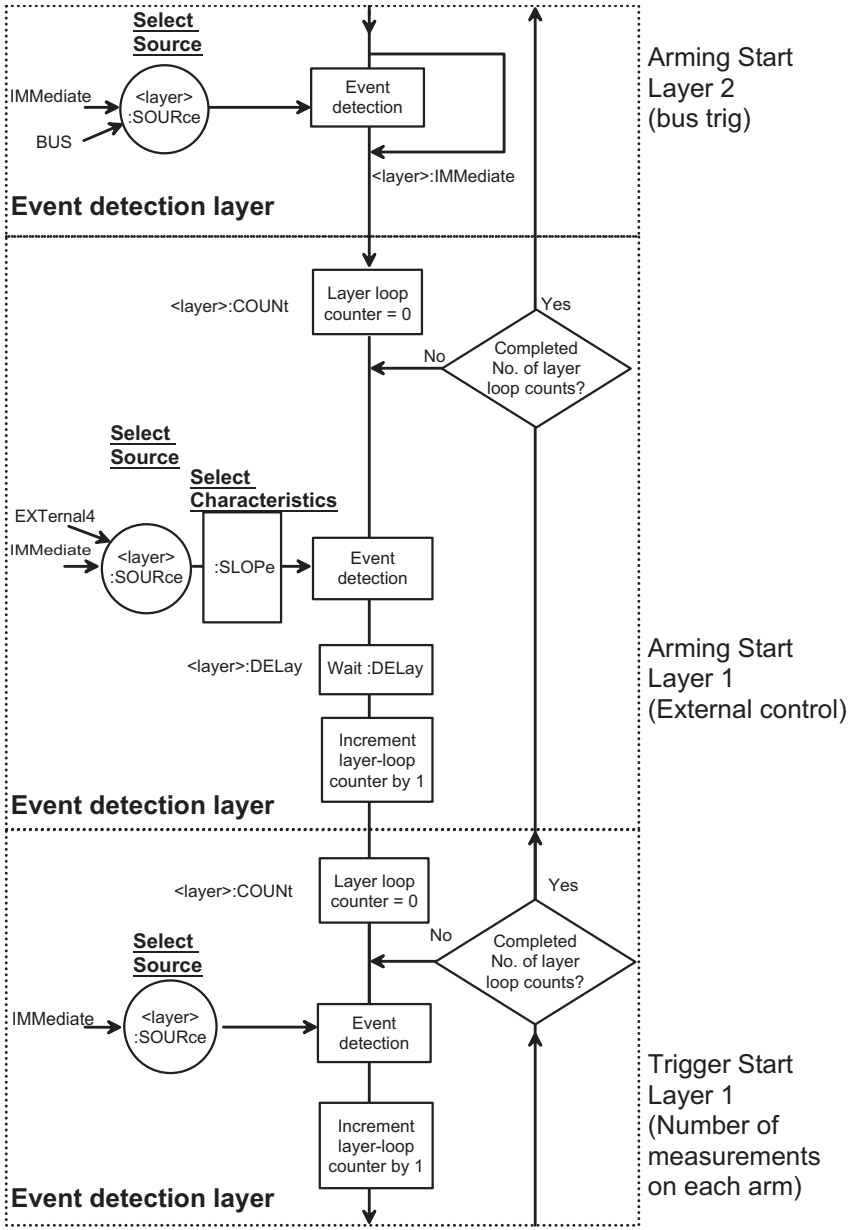


Figure 6-13 *Structure of event detection layers.*

Chapter 7

Error Messages

Read the Error/Event Queue

You read the error queue with the `:SYSTem:ERRor?` query.

Example:

SEND→ `:SYSTem:ERRor?`
READ← `-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.

Read more about how to use error reporting in the Introduction to SCPI chapter

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 indicates only 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 Syntax error; unrecognized data	An unrecognized command or data type was encountered; for example, a string was received when the counter does not accept strings.
–103	Invalid separator	The parser was expecting a separator and encountered 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 encountered.

Command Errors		
Error Number	**Error Description**	**Description/Explanation/Examples**
–105	GET not allowed	A Group Execute Trigger was received within a program message (see IEEE-488.2, 7.7).
–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 counter cannot detect the more specific errors described for errors –111 though –119.
–111	Header separator error	A character that is not a legal header separator was encountered 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 counter; for example, ∗XYZ is not defined for any device.
–114	Header suffix out of range	Indicates that a non-header character has been encountered in what the parser expects is a header element.
–120	Numeric data error	This error, as well as errors –121 through –129, are generated when parsing a data element that appears to be of a numeric type. This particular error message is used when the counter 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 encountered; for example, an alpha in a decimal numeric or a “0" in octal data.

Command Errors		
Error Number	**Error Description**	**Description/Explanation/Examples**
–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 counter does not accept it in this position for the header.
–130	Suffix error	This error as well as errors –131 through –139 is generated when parsing a suffix. This particular error message is used when the counter 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 counter.
–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 encountered after a numeric element that does not allow suffixes.
–140	Character data error	This error as well as errors 141 through –149 is generated when parsing a character data element. This particular error message is used when the counter cannot detect a more specific error.
–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 encountered where prohibited by the counter.
–150	String data error	This error as well as errors –151 through –159 is generated when parsing a string data element. This particular error message is used when the counter 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	

Command Errors		
Error Number	**Error Description**	**Description/Explanation/Examples**
–158	String data not allowed	A string data element was encountered but was not allowed at this point in parsing.
–160	Block data error	This error as well as 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 encountered but was not allowed by the counter at this point in parsing.
–170	Expression data error	This error as well as errors –171 through –179 is generated when parsing an expression data element. This particular error message is used if the counter 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.

Command Errors		
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 encountered but was not allowed by the counter at this point in parsing.
–180	Macro error	This error as well as errors –181 through –189 is generated when defining a macro or executing a macro. This particular error message is used when the counter cannot detect a more specific error.
–181	Invalid outside macro definition	Indicates that a macro parameter placeholder ($<number) was encountered outside of a macro definition.
–183	Invalid inside macro definition	Indicates that the program message unit sequence, sent with a ∗DDT or ∗DMC command, is syntactically invalid (see IEEE-10.7.6.3).
–184	Macro parameter error	Indicates that a command inside the macro definition had the wrong number or type of parameters.
	Macro parameter error; unused parameter	The parameter numbers given are not continuous; one or more numbers have been skipped.
	Macro parameter error; badly formed placeholder	The'$' 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 errors		
Error Number	**Error Description**	**description/explanation/examples**
–200	Execution error	This is the generic syntax error for devices that cannot detect more specific errors. This code indicates only 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 counter but was ignored because of counter timing considerations; for example, the counter was not ready to respond.
–212	Arm ignored	Indicates that an arming signal was received and recognized by the counter 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 counter 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 counter 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	

Execution errors		
Error Number	**Error Description**	**description/explanation/examples**
–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 counter (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 given 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 counter could handle due to memory or related counter-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.
–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 counter.
–240	Hardware error	Indicates that a legal program command or query could not be executed because of a hardware problem in the counter. Definition of what constitutes a hardware problem is completely device specific. This error message is used when the counter cannot detect the more specific errors described for errors –241 through –249.

Execution errors		
Error Number	**Error Description**	**description/explanation/examples**
–241	Hardware missing Hardware missing; (prescaler)"	Indicates that a legal program command or query could not be executed because of missing counter hardware; for example, an option was not installed. Definition of what constitutes missing hardware is completely device specific.
–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 counter 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 counter 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)

Execution errors		
Error Number	**Error Description**	**description/explanation/examples**
–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 counter (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).
–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 counter 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 counter 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.

Standardized Device specific errors		
Error Number	**Error Description**	**description/explanation/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 counter's memory. Contact your local service center.
–312	PUD memory lost	Indicates that the protected user data saved by the ∗PUD command has been lost. Contact your local service center.
–314	Save/recall memory lost	Indicates that the nonvolatile calibration data used by the ∗SAV? command has been 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.

Query errors		
Error Number	**Error Description**	**description/explanation/examples**
–400	Query error	This code indicates only that a Query Error as defined in IEEE-488.2, 11.5.1.1.7 and 6.3 has occurred.
–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 counter 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 counter 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.)

Device specific errors		
Error Number	**Error Description**	**description/explanation/examples**
(1)100	Device operation gave floating-point underflow	A floating-point error occurred during a counter operation.
(1)101	Device operation gave floating-point overflow	A floating-point error occurred during a counter operation.
(1)102	Device operation gave 'not a number'	A floating-point error occurred during a counter operation.
(1)110	Invalid measurement function	The counter 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.
(1)130	Unsupported command	Indicates a mismatch between bus and counter 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 counter has an internal command queue with room for about 100 commands. A large number of 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), Ohm (Ω) and V (Volt).
(1)138	Unexpected command to device execution	A command reached counter execution which should have been handled by the bus.
(1)139	Unexpected query to device execution	A query reached counter 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 counter, and this was not it.

Device specific errors		
Error Number	**Error Description**	**description/explanation/examples**
(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 like an incoming byte, address change, etc.
(1)205	No listener on bus when trying to respond	This error is generated when the counter 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 has been known to do)
(1)210	Mnemonic table error	An abnormal condition occurred in connection with the mnemonics tables (generic error).
(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 has been 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.

Device specific errors		
Error Number	**Error Description**	**description/explanation/examples**
(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.

This page is intentionally left blank.

Chapter 8

Command Reference

This page is intentionally left blank.

Abort

:ABORt

:ABORt

Abort Measurement

The ABORt command terminates a measurement. The trigger subsystem state is set to “idle-state”.

Type of command:

Aborts all previous measurements if *WAI is not used.

Complies with standards: SCPI 1991.0, confirmed.

Arming Subsystem

```
:ARM
[ :STARt / :SEQuence [ 1 ] ]
        :LAYer2
            :[ IMMediate ]
            :SOURce      ␣  BUS | IMMediate
        [ :LAYer [ 1 ] ]
            :COUNt       ␣  <Numeric value> | MIN | MAX
            :DELay       ␣  <Numeric value> | MIN | MAX
            :SLOPe       ␣  POSitive | NEGative
            :SOURce      ␣  EXTernal1 | EXTernal2 | External4 | IMMediate
:STOP / SEQuence2

        [ :LAYer [ 1 ] ]
            :SLOPe       ␣  POSitive | NEGative
            :SOURce      ␣  EXTernal1 | EXTernal2 | EXTernal4 | IMMediate
```

:ARM :COUNt

␣«<Numeric value>|MIN|MAX»

No. of Measurements on each Bus arm

This count variable controls the upward exit of the “wait-for-bus-arm” state (:ARM:STARt:LAY1). The counter loops the trigger subsystem downwards COUNt number of times before it exits to the idle state.

This means that a COUNt No. of measurements can be done for each Bus arming or INITiate.

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

Parameters:

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

MIN gives 1

MAX gives 2 147 483 647

Returned format: <Numeric value>↵

Example:
SEND→ :ARM:COUN␣100↵

***RST condition:** 1

Complies with standards: SCPI 1991.0, confirmed

Delay after External Start Arming

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

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

Parameters:

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

MIN gives 0 which switches the delay OFF.

MAX gives 2 s

Returned format: <Numeric value>↲

Example:
SEND→ :ARM:DEL ␣0.1↲

***RST condition:** 0

Complies with standards: SCPI 1991.0, confirmed.

:ARM :LAYer2

Bus Arming Override

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

The counter 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.

Example:
SEND→ :ARM:LAY2↲

Complies with standards: SCPI 1991.0, confirmed.

:ARM :LAYer2 :SOURce

␣ «BUS | IMMediate»

Bus Arming On/Off

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

If the counter 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).

Returned format: BUS|IMM↵

Example:
SEND→ :ARM:LAY2:SOUR ␣ BUS↵

Complies with standards: SCPI 1991.0, confirmed.

:ARM :SLOPe

␣ «POSitive|NEGative»

External Arming Start Slope

Sets the slope for the start arming condition.

Returned format: POS|NEG↵

Example:
SEND→ :ARM:SLOP ␣ NEG↵

***RST condition:** POS

Complies with standards: SCPI 1991.0, confirmed.

:ARM :SOURce
␣ «EXTernal1 | EXTernal2 | EXTernal4 | IMMediate»

External Arming Start Source

Selects channel 4 (Input E) as arming input, or switches off the start arming function. When switched off the DELay is inactive.

Parameters:

EXTernal1 is input A

EXTernal2 is input B

EXTernal4 is input E

IMMediate is Start arming OFF

Returned format: EXT1 | EXT2 | EXT4 | IMM↲

Example:
SEND→ `:ARM:SOUR ␣ EXT4`↲

***RST condition:** IMM

Complies with standards: SCPI 1991.0, confirmed.

:ARM :STOP :SLOPe
␣ «POSitive | NEGative»

External Stop Arming Slope

Sets the slope for the stop arming condition.

Returned format: POS|NEG↲

Example:
SEND→ `:ARM:STOP:SLOP ␣ NEG`↲

***RST condition:** POS

Complies with standards: SCPI 1991.0, confirmed.

:ARM :STOP :SOURce

␣ «EXTernal1 | EXTernal2 | EXTernal4 | IMMediate»

External Stop Arming Source

Selects between channel 2 (Input B) and channel 4 (Input E) as stop arming input, or switches off the stop arming function.

Parameters:

EXTernal1 is input A

EXTernal2 is input B

EXTernal4 is input E

IMMediate is Stop arming OFF

Returned format: EXT1 | EXT2 | EXT4 | IMM↵

Example:

SEND→ `:ARM:STOP:SOUR ␣ EXT4↵`

***RST condition:** IMM

Complies with standards: SCPI 1991.0, confirmed.

Calculate Subsystem

:CALCulate

:STATe	␣ ON|OFF
:DATA?	
:IMMediate	
:MATH	
[:EXPRession]	␣ (<Numeric expression>)
:STATe	␣ ON|OFF
:AVERage	
[:STATe]	␣ ON|OFF
:TYPE	␣ MIN|MAX|MEAN|SDEViation|ADEViation
:COUNt	␣ <Numeric value>|MIN|MAX
:LIMit	
[:STATe]	␣ ON|OFF
:FAIL?	
:CLEar	
:AUTO	␣ ON|OFF
:FCOunt?	
:FCOunt	
:LOWer?	
:UPPer?	
:PCOunt?	
:UPPer	
[:DATA]	␣ <Numeric value>|MIN|MAX
:STATe	␣ ON|OFF
:LOWer	
[:DATA]	␣ <Numeric value>|MIN|MAX
:STATe	␣ ON|OFF

:CALCulate :AVERage :COUNt

␣ < No. of samples>

Sample Size for Statistics

Sets the number of samples to use in statistics sampling.

Parameters: <No. of samples> is an integer in the range 2 to $2*10^9$.

Returned format: < No. of samples>↵

***RST condition:** 100

$2*10^9$

:CALCulate :AVERage :STATe

␣ < Boolean >

Enable Statistics

Switches On/Off the statistical function. Note that 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 counter will keep the trigger subsystem initiated until the `:CALC:AVER: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 counter 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).

Parameters

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

Returned format: <1|0↵

***RST condition:** OFF

:CALCulate :AVERage :TYPE

␣ «MAX|MIN|MEAN|SDEViation|ADEViation»

Statistical Type

Selects the statistical function to be performed.

You must use :CALC:DATA? *to read the result of statistical operations.* :READ?, :FETC? *will only send the results that the statistical operation is based on.*

Parameters:

MAX returns the max. value of all samples taken under `:CALC:AVER` control.

MIN returns the min. value of all samples taken under `:CALC:AVER` control.

MEAN returns the mean value of the samples taken: $\bar{x} = \frac{1}{N}\sum_{i=1}^{N} X_i$

SDEV returns the standard deviation: $s = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N-1}}$

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

Returned format: MAX|MIN|MEAN|SDEV|ADEV↲

***RST condition:** MEAN

:CALCulate :DATA?

Fetch calculated data

Fetches data calculated in the post processing block. Use this command to fetch the calculated result without making a new measurement.

Returned Format:

<Decimal data>↲

Example:

SEND→ `:CALC:MATH:STAT␣ON;:CALC:MATH␣(((1␣*␣X)␣-␣10.7E6)␣/␣1)`
`;:init; *OPC`

Wait for operation complete

SEND→ `:CALC:DATA?`

READ← `<Measurement ␣ result ␣ minus ␣ 10.7E6>`

***RST condition:**

Event, no *`RST` condition.

Complies with standards: SCPI 1991.0, Confirmed

:CALCulate :IMMediate

Recalculate Data

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

Returned format: <Decimal data>↵
Where: <Decimal data> is the recalculated data.

Example:
SEND→ :CALC:AVER:STAT ␣ ON;TYPE ␣ SDEV;:INIT;*OPC
Wait for operation complete

SEND→ :CALC:DATA?
READ← <Value ␣ of ␣ standard ␣ deviation>
SEND→ :CALC:AVER:TYPE ␣ MEAN
SEND→ :CALC:IMM?
READ← <Mean ␣ value>

***RST condition:** Event, no *RST condition.

Complies with standards: SCPI 1991.0, Confirmed.

:CALCulate :LIMit
␣ <Boolean>

Enable Monitoring of Parameter Limits

Turns On/Off the limit-monitoring calculations.
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. The bits are enabled 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/Off will not influence the status register mask bits, which determine whether or not a service request will be generated 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.

Parameters <Boolean> = (1/ON | 0/OFF)

Returned format: 1|0↵

***RST condition:** OFF

See also: Example 1 in Chapter deals with limit-monitoring.

Complies with standards: SCPI 1991.0, confirmed.

:CALCulate :LIMit :CLEar

Clear Limit Failure Count

The command resets the counter that reports its result over the `:CALCulate:LIMit:FCOunt?` query command.

:CALCulate :LIMit :CLEar :AUTO

␣ <Boolean>

Automatic Reset of Limit Failure Count

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

Parameters <Boolean> = (1/ON | 0/OFF)

***RST condition:** OFF

:CALCulate :LIMit :FAIL?

Limit Fail

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:STAT OFF → :CALC:LIM:STAT ␣ ON transition

– Reading a 1 with this command.

Returned format: 1| 0↲

Example:

SEND→ SENS:FUNC ␣ FREQ;:CALC:LIM:STAT ␣ ON;:CALC:LIM ␣ :HIGH␣ 1E3;READ?;*WAI;:CALC:LIM:FAIL?

READ← 1

if frequency ia above 1kHz, otherwize 0

Complies with standards: SCPI 1991.0, confirmed.

:CALCulate :LIMit :FCOunt?

Number of Limit Failure Counts

The command returns the number of times the set limits have been exceeded since the counter was last reset by :CALC:LIM:CLEAR or automatically by :INIT if :CALC:LIM:CLEAR:AUTO ON has been activated.

Returned format: < No. of counts>↲

:CALCulate :LIMit :LOWer

␣ «<Decimal data>|MAX|MIN»

Set Low Limit

Sets the value of the 'Lower Limit' , i.e., the lowest measurement result allowed before the counter generates a 1 that can be read with `:CALCulate:LIMit:FAIL?`, or by reading the corresponding status byte.

Parameters

Parameter range: $-9.9*10^{+37}$ to $+9.9*10^{+37}$.

Returned format: < Decimal data>↲

***RST condition:** 0

Complies with standards: SCPI 1991.0, confirmed.

:CALCulate :LIMit :LOWer :STATe

␣ <Boolean>

Check Against Lower Limit

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

Parameters <Boolean> = (1/ON | 0/OFF)

Returned format: 1| 0 ↲

***RST condition:** 0

Complies with standards: SCPI 1991.0 confirmed.

:CALCulate :LIMit :UPPer

␣ «<Decimal data>|MAX|MIN»

Set Upper Limit

Sets the value of the 'Upper Limit', i.e., the highest measurement result allowed before the counter generates a 1 that can be read with `:CALCulate:LIMit:FAIL?`, or by reading the corresponding status byte.

Parameters

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

Returned format: <Decimal data>↵

***RST condition:** 0

Complies with standards: SCPI 1991.0, confirmed.

:CALCulate :LIMit :UPPer :STATe

␣ <Boolean>

Check Against Upper Limit

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

Parameters <Boolean> = (1/ON | 0/OFF)

Returned format: 1| 0 ↵

***RST condition:** 0

Complies with standards: SCPI 1991.0, confirmed.

Select Mathematical Expression

Defines the mathematical expression used for mathematical operations.

The data type <expression data> must be typed within parentheses.

Parameters

<expression> is one of the following four mathematical expressions:

((K␣␣X)␣+␣L) or ((K␣/␣X)␣+␣L) or*

(((K␣␣X)␣+␣L)␣/␣M) or (((K␣/␣X)␣+␣L)␣/␣M)* ***No deviations are allowed.***

K, L and M can be any positive or negative numerical constant, or use XOLD for the last, previously measured value.

Each operator must be surrounded by space characters.

Example

SEND→:CALC:MATH ␣ (((10 ␣ * ␣ X ␣ + ␣ -1e6) ␣ / ␣ 1000000)

This example gives a relative result from the last measuring result.

***RST condition:**

((1* *X*)+0) (No calculation)

Returned format: <expression>↵

Complies with standards: SCPI 1991.0 Confirmed.

Enable Mathematics

Switches on/off the mathematical function. Note that 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.

Parameters:

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

Returned format: 1|0

Example

SEND→:CALC:MATH:STAT ␣ 1

This example switches on mathematics.

***RST condition:** OFF

Complies with standards: SCPI 1991.0, confirmed.

:CALCulate :STATe

␣ <Boolean>

Enable Calculation

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

Parameter

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

SEND→ `:CALC:STAT ␣ 1`

Switches on Post Processing.

Returned format: 1|0↵

***RST condition:** OFF

Complies with standards: SCPI 1991.0, Confirmed

Calibration Subsystem

:CALibration
:INTerpolator
:AUTO ␣ <Boolean>

:CALibration :INTerpolator :AUTO

␣ <Boolean>

Calibration of Interpolator

The '90' is a reciprocal counter that uses an interpolating technique to increase the resolution. In time measurements, for example, interpolation increases the resolution from 10 ns to 0.1 ns.

The counter calibrates the interpolators automatically once for every measurement when this command is ON. When this command is OFF, the counter 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.

Parameters

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

Returned format: 1|0↵

***RST condition:** ON

Configure Function

Set up Instrument for Measurement

:CONFigure
[:SCALar]<Measuring Function> ␣ <Parameters>,(<Channels>)]
:ARRay<Measuring Function> ␣ (<Array Size>)[,<Parameters>,(<Channels>)]

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

Measuring Function, Parameters and Channels are explained on page 8-48.

The counter uses the default Parameters and Channels when you omit them in the command.

:CONFigure :<Measuring Function>

[␣ <parameters>[,(<channels>)]]

Configure the counter for a single measurement

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 counter 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 counter has chosen to configure itself for the measurement, send an *RST before doing other manual set up measurements.

Parameters

<Measuring Function>, <Parameters> and <Channels> are defined on page 8-48.

The optional parameter :SCALar means that one measurement is to be done.

Returned format: <String>↵

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

Example:

SEND→ :CONF:FREQ:RAT␣(@3),(@1)

Configures the counter for freq. ratio C/A.

See also: 'Explanations of the Measuring Functions' starting on page 8-53.

Complies with standards: SCPI 1991.0, confirmed.

:CONFigure :ARRay :<Measuring Function>

␣ (<array size>)[,<parameters> [,(<channels>)]]

Configure the counter for an array of measurements

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

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

Parameters <array size> sets the number of measurements in the array (1 to 2500).

<Measuring Function>, <Parameters>, and <Channels> are defined on page 8-48.

Example:

SEND→ :CONF:ARR:PER ␣ (7),5E-3,1E-6,(@4)

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

To make the measurements and fetch the seven measurement results:

SEND→ :READ:ARR? ␣ 7

READ← 5.23421E-3,5.12311E-3,5.87526E-3, ␣
5.50345E-3,5.33901E-3,5.25501E-3, ␣ 5.03571E-3

Complies with standards: SCPI 1991.0, confirmed.

This page is intentionally left blank.

Display Subsystem

:DISPlay

:ENABle␣ ON OFF

:DISPlay :ENABle

␣ < Boolean >

Display State

Turns On/Off the updating of the entire display section. This can be used for security reasons or to improve the GPIB speed, since the display does not need to be updated.

Parameters: <Boolean> = (1 / ON | 0 / OFF)

Returned format: 1|0 ↵

***RST condition:** ON

Complies with standards: SCPI 1991.0, confirmed.

Fetch Function

:FETCh

[:SCALar]?

:ARRay?␣ <Array Size>|MAX

:FETCh?

Fetch One Result

The fetch query retrieves one measuring result from the measurement result buffer of the counter without making new measurements. Fetch does not work unless a measurement has been made by the :INITiate, :MEASure?, or :READ? commands.

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

The same measuring result can be fetched again and again, as long as the result is valid, i.e., 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 has been started, the fetch query completes when a new measuring result becomes valid. If no new measurement has been started, an error is returned.

The optional :SCALar means that one result is retrieved.

Returned format: <data>↵

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

		FORMAT		
		ASCii	**REAL**	**PACKED**
:FORMAT:TINF	**OFF**	<Val>	#18<Val>	#18<Val>
	ON	<Val>,<TS>	#18<Val>,#18<TS>	#216<Val><TS>

Val = measurement value (double precision in REAL and PACKed)
TS = timestamp value (double precision in REAL and int64 ps in PACKed)

If no valid result can be returned, e.g. due to time-out, the returned data will depend on the chosen GPIB mode according to the table under :FETCh:ARRay?

Complies with standards: SCPI 1991.0, confirmed.

Fetch an Array of Results

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

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

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

When the counter 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 has been 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 gives 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.

Parameters

:ARRay means that an array of retrievals are done for each :FETCh command. <fetch array size> is the number of retrievals in the array. This number must not exceed the number of measuring results in the measurement result buffer. The <SIZE> parameter maximum limit is depending on the :SENSe:INTernal:FORMat command as follows:

Measuring function	Array Size	
	CAL:ON	CAL:OFF
Frequency, Period, Ratio	375k	750k
Pulse Width, Time Interval, Rise/Fall Time	375k	750k
Phase, Duty Cycle	375k	750k
Volt	10k	10k
Smart Frequency and Period	30k	30k

MAX means that all the results in the output buffer will be fetched.

Returned format: <data>[,<data>]↲

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

Example:

If `:MEAS:ARR:FREQ? ␣(4)` *gives the results* `1.1000,1.2000,1.3000,1.4000`
`:FETC:ARR ␣2` *fetches the results* `1.1000,1.2000`
`:FETC:ARR ␣2` *once more fetches the results* `1.3000,1.4000`
`:FETC:ARR ␣−1` *always fetches the last result* `1.4000`

		FORMAT		
		ASCii	**REAL**	**PACKED**
:FORMAT:TINF	**OFF**	<Val>,<Val>,...	#18<Val>,#18<Val>,...	#ddd<Val><Val><Val>...
	ON	<Val>,<TS>,<Val>,...	#18<Val>,#18<TS>, #18<Val>,...	#ddd<Val><TS><Val>...

Val = measurement value (double precision in REAL and PACKed)
TS = timestamp value (double precision in REAL and int64 ps in PACKed)

If no valid result can be returned, e.g. due to time-out, the returned data will depend on the chosen GPIB mode according to the table below.

	GPIB MODE	
FORMAT	**NATIVE**	**COMPATIBLE**
ASCii	<LF>	9.91E37
REAL	#18<Binary NaN>	#18<9.91E37 in binary format>
PACKED	#18<Binary NaN>	#18<9.91E37 in binary format>

NaN = *Not a Number*, a standardized bit pattern indicating that the transferred data is not a valid result.

Complies with standards: SCPI 1991.0, confirmed.

Format Subsystem

:FORMat

[:DATA]	␣	ASCii | REAL | PACKed[, <Numeric value>]
:FIXed	␣	ON OFF
:TINFormation[:STATe]	␣	<Boolean>

:FORMat

␣ «ASCii|REAL|PACKed»[, <Numeric value>

Response Data Type

Sets the format in which the result will be sent on the bus.

Parameters:

ASCii: The length controls the number of digits in the mantissa and may be set to values from 2 to 12 or 0. When set to 0, the length will be automatically controlled by the resolution of each measurement result.

The 0 will be ignored when `:DISPlay:ENABled` *␣ OFF is selected.*

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

PACK: See REAL.

Returned format: ASC|REAL, <Numeric value>↵

***RST condition:** ASCii, 0

See also: `:FORMat :TINFormation` command

Complies with standards: SCPI 1991.0, confirmed.

:FORMat :FIXed

␣ <Boolean>

Response Data Format

Sets the ASCii format to fixed. This results in the following response format:

<sign><mantissa value>E<sign><exponent value>

Where:

<sign> = +|–
<mantissa value> = 12 digits plus one decimal point.
<exponent value> = 3 digits

Parameters <Boolean> = (1 / ON | 0 / OFF)

Returned format: 1|0 ↵

***RST condition:** OFF

Timestamping On/Off

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 function on page 8-29 ff.

For :FETCh:SCALar?, :READ:SCALar? and :MEASure:SCALar? the readout will consist of two values instead of one. The first will be the measured value and the next one will be the timestamp value.

In :FORMat ASCii mode, both the measured value and the timestamp value will be given as floating-point numbers expressed in the basic units (e.g. Hz - s or s - s).

In :FORMat REAL mode, the result will be 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 will consist of alternating measurement values and timestamp values, formatted in a similar way as for scalar readout. All values will be separated by commas. See also the :MEASure:ARRay:TSTAmp? command on page 8-66 for more information on the output format.

Parameters <Boolean> = (1 / ON | 0 / OFF)

Returned format: 1|0 ↲

***RST condition:** OFF

This page is intentionally left blank.

Hard Copy

:HCOPy
:SDUMp
:DATA?

:HCOPy :SDUMp :DATA?

Screen Dump

Return block data containing screen dump in Windows BMP format.

Returned Format:

#43880<Binary BMP Data>

The '4' means that the following four digits (3880) tell how many data bytes will succeed. The total number of transferred data bits (3880 × 8) equals the number of display pixels (320 × 97).

Initiate Subsystem

:INITiate
[:IMMediate]
:CONTinuous␣ ON | OFF

:INITiate

Initiate Measurement

The :INITiate command initiates a measurement. Executing an :INITiate command changes the counter's trigger subsystem state from "idle-state" to "wait-for-bus-arm-state" (see). 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 also 'How to use the Trigger Subsystem' at the end of this chapter.

Complies with standards: SCPI 1991.0, confirmed.

:INITiate :CONTinuous

␣ <Boolean>

Continuously Initiated

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", i.e., the counter is continuously measuring and storing response data.

Returned format: 1|0↵

***RST condition:** OFF

Complies with standards: SCPI 1991.0, confirmed.

Input Subsystems

■ INPUT A

:INPut[1]

:ATTenuation		␣	<Numeric value>|MIN|MAX (1|10)
:COUPling		␣␣	AC|DC
:IMPedance		␣	<Numeric value>|MIN|MAX
[:EVENt]			
:LEVel		␣	<Numeric value>|MIN|MAX
	:AUTO	␣␣	ON|OFF|ONCE
:SLOPe		␣	POS|NEG
:FILTer			
[:LPASs]			
	[:STATe]	␣	ON|OFF

■ INPUT B

:INPut2

:ATTenuation		␣␣	<Numeric value>|MIN|MAX (1|10)
:COUPling		␣␣	AC|DC
:IMPedance		␣␣	<Numeric value>|MIN|MAX
[:EVENt]			
:LEVel		␣␣	<Numeric value>|MIN|MAX
	:AUTO	␣␣	ON|OFF|ONCE
:SLOPe		␣	POS|NEG
:FILTer			
[:LPASs]			
	[:STATe]	␣␣	ON|OFF
:COMMon		␣	ON|OFF

■ INPUT E

:INPut4

[:EVENt]		
:SLOPe	␣	POS|NEG

:INPut«[1]|2» :ATTenuation

␣ «<Numeric value>|MAX|MIN»

Attenuation

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

Parameters:

<Numeric values> ≤ 5, and MIN gives attenuation 1.
<Numeric values> > 5, and MAX gives attenuation 10.

Returned format:

1.00000000000E+000|1.00000000000E+001 ↵

Example for Input A (1)

SEND→ `:INP:ATT ␣ 10`

Example for Input B (2)

SEND→ `:INP2:ATT ␣ 10`

***RST condition** Input A (1) and Input B (2): 1 (but set by autotrigger since AUTO is on after *`RST`. (`:INP:LEV:AUTO ␣` ON).

Complies with standards: SCPI 1991.0, confirmed.

:INPut«[1]|2» :COUPling

␣ «AC|DC»

AC/DC Coupling

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

Returned format: AC|DC↵

Example for Input A (1)

SEND→ `:INP:COUP ␣ DC`

Example for Input B (2)

SEND→ `:INP2:COUP ␣ AC`

***RST condition**

Input A (1): AC

Input B (2): AC

Complies with standards: SCPI 1991.0, confirmed.

:INPut«[1]|2» :FILTer
␣ <Boolean>

Low Pass Filter

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

Parameters:
<Boolean> is (1 / ON | 0 / OFF)

Returned format: 1|0↵

***RST condition** OFF

Complies with standards: SCPI 1991.0, confirmed.

:INPut«[1]|2» :IMPedance
␣ «<Decimal data>|MAX|MIN»

Input Impedance

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

Parameters

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.

Returned format:
5.00000000000E+001|1.00000000000E+6↵

Example for Input A (1)
SEND→ `:INP:IMP ␣ 50`
Sets the input A impedance to 50 Ω.

Example for Input B (2)
SEND→ `:INP2:IMP ␣ 50`
Sets the input B impedance to 50 Ω.

***RST condition** 1 MΩ

Complies with standards: SCPI 1991.0, confirmed.

:INPut«[1]|2» :LEVel

␣ «<Decimal data>|MAX|MIN»

Fixed Trigger Level

Input A and input B can be individually set to autotrigger or to fixed trigger levels of between –5 V and +5 V in steps of 2.5 mV. If the attenuator is set to 10X, the range is –50 V and +50 V in 25 mV steps.

For autotrigger, see the following page.

Parameters: <Decimal data> is a number between –5 V and +5 V if att=1X and between –50 V and +50 V if att=10X.

MAX gives +50 V and MIN gives –50 V

When using MAX and MIN as data, the counter always tries to set the trigger level to +50 V and –50 V. If the attenuator is set to 1X, it is impossible to set this trigger level, and the counter will return an error message.

Returned format: <Decimal data>↵

Example for Input A (1)
SEND→ `:INP:LEV ␣ 0.01`

Example for Input B (2)
SEND→ `:INP2:LEV ␣ 2.0`

***RST condition** 0 (but controlled by Autotrigger since AUTO is on after *`RST`)

:INPut«[1]|2» :LEVel :AUTO

␣ «<Boolean>»

Autotrigger

If set to AUTO, the counter automatically controls both the trigger level and the attenuation[1]. If you have a stable amplitude, use the `:AUTO ␣` ONCE, and the autotrigger will determine the trigger level once and then set a fixed level.

From the bus, input A and input B are always set to autotrigger individually.

Parameters:

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

ONCE means that the autotrigger switches on, checks the signal, stores the trigger levels as manually set levels, and then switches off auto. This improves measuring speed.

Example for Input A (1)
SEND→ `:INP:LEV:AUTO ␣ OFF`

Example for Input B (2)
SEND→ `:INP2:LEV:AUTO ␣ ON`

Returned format: 1|0↵

***RST condition** ON

1 The autotrigger function normally sets the trigger levels to 50 % of the signal amplitude. Two exceptions exists however:
Rise/Fall time measurements: Here the input 1 (A) trigger level is set to 10% and the Input 2 (B) trigger level is set to 90% of the amplitude.
Variable Hysteresis mode (channel 7): The input 1 (A) trigger level is set to 75% and the Input 2 (B) trigger level is set to 25% of the amplitude

:INPut«[1]|2» :SLOPe

␣ «POS|NEG»

Trigger Slope

Selects if the counter 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.

Returned format: POS | NEG ↲

Example for Input A (1)
SEND→ `:INP:SLOP ␣ POS`

Example for Input B (2)
SEND→ `:INP2:SLOP ␣ NEG`

***RST condition** POS

Complies with standards: SCPI 1991.0, confirmed.

This page is intentionally left blank.

Measurement Function

Set up the Instrument, Perform Measurement, and Read Data

```
:MEASure
      [:SCALar]<Measuring Function>?
[<Parameters>][,(<Channels>)]
      :ARRay<Measuring Function>?          ␣ (<Array Size>)[,<Parameters>][,(<Channels>)]
      :MEMory?                             ␣ [<N>]
      :MEMory<N>?
```

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

The default channels, which the counter uses when you omit the channels in the command, are printed in italics in the channel list on the following pages.

If you want to check what function and channels the counter is currently using, send :CONF?
This query gives the same answer as :FUNC? *in the SENSe subsystem*

```
:MEASure|:CONFigure
 [:SCALar]
    [:VOLTage]
       :FREQuency
          [:CW]?          [ [<exp. value>[,<resol.>],][(@1|@2|@3|@4|@6)]]
          :BURSt?         [ [<exp. value>[,<resol.>],][(@1|@2|@3)]]
          :PRF?           [ [<exp. value>[,<resol.>],][(@1|@2|@3)]]
          :RATio?         [ [<exp. value>[,<resol.>],][(@1|@2|@3),(@1|@2|@3)]]
       :NCYCles?          [ (@1|@2|@3)]
       :PDUTycycle|:DCYCle? [ [<reference>,][(@1|@2)]]
       :NDUTycycle?       [ [<reference>,][(@1|@2)]]
       :MAXimum?          [ (@1|@2)]
       :MINimum?          [ (@1|@2)]
       :PTPeak?           [ (@1|@2)]
       :RATio?            [ (@1|@2),(@1|@2)]
       :PSLEwrate?        [ (@1|@2)]
       :NSLEwrate?        [ (@1|@2)]
       :PERiod?           [ [<exp. value>[,<resol.>],][(@1|@2|@3)]]
       :PERiod:AVERage?   [ [<exp. value>[,<resol.>],][(@1|@2|@3)]]
       :PHASe?            [ [<exp. value>[,<resol.>],][(@1|@2),(@1|@2)]]
       «:RISE:TIME|:RTIM»? [ [<lo thresh.>[,<hi thresh.>[,<exp. value>[,<resol.>]]],][(@1|@2)]]
       «:FALL:TIME|:FTIM»? [ [<lo thresh.>[,<hi thresh.>[,<exp. value>[,<resol.>]]],][(@1|@2)]]
       :TINTerval?        [ [<exp. value>[,<resol.>],][(@1|@2),(@1|@2)]]
       :PWIDth?           [ [<reference>,][(@1|@2)]]
       :NWIDth?           [ [<reference>,][(@1|@2)]]

 :ARRay
    [:VOLTage]
       :FREQuency
          [:CW]?          (<Size>)[,[<exp. value>[,<resol.>],] [(@1|@2|@3|@4|@6)]]
          :BURSt?         (<Size>)[,[<exp. value>[,<resol.>],][(@1|@2|@3)]]
          :PRF?           (<Size>)[,[<exp. value>[,<resol.>],][(@1|@2|@3)]]
          :RATio?         (<Size>)[,[<exp. value>[,<resol.>],][(@1|@2|@3),(@1|@2|@3)]]
       :NCYCles?          (<Size>)[,(@1|@2|@3)]
       :PDUTycycle|:DCYCle? (<Size>)[,[<exp. value>[,<resol.>],][(@1|@2)]]
       :NDUTycycle?       (<Size>)[,[<exp. value>[,<resol.>],][(@1|@2)]]
       :MAXimum?          (<Size>)[,(@1|@2)]
       :MINimum?          (<Size>)[,(@1|@2)]
       :PTPeak?           (<Size>)[,(@1|@2)]
       :RATio?            (<Size>)[,(@1|@2),(@1|@2)]
       :PSLEwrate?        (<Size>)[,(@1|@2)]
       :NSLEwrate?        (<Size>)[,(@1|@2)]
       :PERiod?           (<Size>)[,[<exp. value>[,<resol.>],][(@1|@2|@3)]]
       :PERiod:AVERage?   (<Size>)[,[<exp. value>[,<resol.>],][(@1|@2|@3)]]
       :PHASe?            (<Size>)[,[<exp. value>[,<resol.>],][(@1|@2),(@1|@2)]]
       «:RISE:TIME|:RTIM»? (<Size>)[,[<lo thresh.>[,<hi thresh.>[,<exp. value>[,<resol.>]]],][(@1|@2)]]
       «:FALL:TIME|:FTIM»? (<Size>)[,[<lo thresh.>[,<hi thresh.>[,<exp. value>[,<resol.>]]],][(@1|@2)]]
       :TINTerval?        (<Size>)[,[<exp. value>[,<resol.>],][(@1|@2),(@1|@2)]]
       :PWIDth?           (<Size>)[,[<exp. value>[,<resol.>],][(@1|@2)]]
       :NWIDth?           (<Size>)[,[<exp. value>[,<resol.>],][(@1|@2)]]
       :TSTAmp?           (<Size>)[,(@1|@2)]
```

(@1) means input A
(@2) means input B
(@3) means input C (RF input option)
(@4) means the rear panel arming input
(@6) means the internal reference

:MEASure :<Measuring Function>?

[␣ [<parameters>][,(<channels>)]]

Make one measurement

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.

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

You cannot use the `:MEASure?` query for `:TOTalize:CONTinuous`, since this function measures without stopping (continuously forever).

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

Parameters:

<Measuring Function>, <Parameters> and <Channels> are defined on page 8-48. You may omit <parameters> and <Channels>, which are then set to default.

Returned format: <data>↲

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

Example:

SEND→ `:MEAS:FREQ? ␣ (@3)`
READ← `1.78112526833E+009`

This example measures the frequency on the C-input and outputs the result to the controller.

Type of command: Aborts all previous measurement commands if *WAI is not used.

See also: 'Explanations of the Measuring Functions' starting on page 8-53.

Complies with standards: SCPI 1991.0, confirmed.

:MEASure :ARRay :<Measuring Function>?

␣ (<array size>)[,[<parameters>] [,(<channels>)]]

Make an array of measurements

The :MEASure:ARRay query differs from the :MEASure 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.

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

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

Parameters:

<array size> sets the number of measurements in the array.

Returned format:

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

Example:

SEND→ :MEAS:ARR:FREQ? ␣ (10)

Ten measuring results will be returned.

Type of command:

Aborts all previous measurement commands if not *WAI is used, see page 8-118.

Complies with standards: SCPI 1991.0, confirmed.

:MEASure:MEMory<N>?

Memory Recall, Measure and Fetch Result

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

`:MEAS:MEM1?` recalls the contents of memory 1 and reads out the result,
`:MEAS:MEM2?` recalls the contents of memory two and reads out the result etc.

The equivalent command sequence is `*RCL1;READ?`

The allowed range for <N> is 1 to 9. Use the somewhat slower `:MEAS:MEMory?␣N` command described below if you must use memories 10 to 19.

Returned format:

<measurement result>↵

Complies with standards: SCPI 1991.0, confirmed

:MEASure:MEMory? ␣ <N>

Memory Recall, Measure and Fetch Result

Same as above command but somewhat slower. Allows use of all memories (1 to 19).

Example: `:MEAS:MEM ␣ 13`

This example recalls the instrument setting in memory number 13, makes a measurement, and fetches the result.

Complies with standards: SCPI 1991.0, confirmed

This page is intentionally left blank.

EXPLANATIONS OF THE MEASURING FUNCTIONS

This sub-chapter explains the various measurements that can be done with `:MEASure` and `:CONFigure;:READ`. Only the queries for single measurements using the measure command are given here, but all of the information is also valid for the `:CONFigure` command and for both scalar (single) and array measurements.

:MEASure :FREQuency?

[␣ [<expected value>[,<resolution>]] [,<(@«1|2|3|4|6»)>]]

Frequency

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

Example:

SEND→ `:MEAS:FREQ? ␣ (@3)`

READ← `1.78112526833E+009`

This example measures the frequency at input C.

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

Parameters:

<expected value> is the expected frequency,

<resolution> is the required resolution.

<(@«1|2|3|4|6»)> is the channel to measure on:

(@1) means input A[1]

(@2) means input B

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

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

(@6) means the internal reference

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

1 The A input is always prescaled by 2 when measuring Frequency A and prescaled by 1 for all other functions.

Complies with standards: SCPI 1991.0, confirmed.

Burst Carrier Frequency

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 counter uses the <expected value> and <resolution> to select a Measurement Time (`[:SENSe]:ACQuisition:APERture`), and then sets the sync delay (`[:SENSe]:SDELay`) to 1.5 * Measurement Time.

Parameters:

<expected value> is the expected carrier frequency,
<resolution> is the required resolution, e.g., 1 gives 1Hz resolution.

<(@«1|2|3|4|5|6|7»)> is the measurement channel:
(@1) means input A
(@2) means input B
(@3) means input C (HF-input option)
(@4) means input E (Rear panel arming input)
(@5) means input A prescaled by 2
(@6) means the internal reference
(@7) means input A with the variable hysteresis mode

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

Complies with standards: SCPI 1992.0, confirmed.

:MEASure :FREQuency :PRF?

[␣ [<exp. val.>[,<res.>]][,<(@«1|2|3|4|5|6|7»)>]]

Pulse Repetition Frequency

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

It is better to set up the measurement with the [:SENS]:FUNC ":FREQ:PRF" *command when measuring pulse repetition frequency. This command will allow you to set a suitable sync delay with the* [:SENSe]:Sync:DELay *command.*

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

Parameters: <exp. val.> is the expected PRF,

<res.> is the required resolution.

<(@«1|3|4|5|6»)> is the measurement channel:

(@1) means input A
(@2) means input B
(@3) means input C (HF-input option)
(@4) means input E (Rear panel arming input)
(@5) means input A prescaled by 2
(@6) means the internal reference
(@7) means input A with the variable hysteresis mode

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

The <expected value> and <resolution> are used to calculate the Measurement Time ([:SENSe]:ACQuisition:APERture). The Sync. Delay is always 10 µs (default value)

Complies with standards: SCPI 1992.0, confirmed.

:MEASure :FREQuency :RATio?

[␣ [<expected value> [,<resolution>]][,<(@«1|2|3»)>,<(@«1|2|3»)>]]

Frequency Ratio

Frequency ratio measurements between two inputs.

Example:

SEND→ `:MEAS:FREQ:RAT? ␣ (@1),(@3)`

READ← `2.345625764333E+000`

This example measures the ratio between input A and input C.

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

Parameters: <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.

Complies with standards: SCPI 1991.0, confirmed.

:MEASure [:VOLT] :NCYCles?

[␣ [(@1|@2|@3)]

Number of Cycles in Burst

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

Returned format: <Numeric value (integer)>

Example:

SEND→ `:MEAS:NCYC? (@3)`

READ← `2356`↵

This example shows a measurement on the RF channel.

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

<(@«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)

:MEASure_«:PDUTycycle | :DCYCle»?

[␣ [<threshold>] [,(@«1|2»)]]

Positive duty cycle: Duty Factor

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

Parameters

<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).

Example:
SEND→ `MEAS:PDUT?`
READ← `+5.097555E-001`
In this example, the duty cycle is 50.97%

Complies with standards: SCPI 1991.0, confirmed.

:MEASure_:NDUTycycle?

[␣ [<threshold>] [,(@«1|2»)]]

Negative duty cycle: Duty Factor

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

Parameters

<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).

Example:
SEND→ `MEAS:PDUT?`
READ← `+5.097555E-001`
In this example, the duty cycle is 50.97%

Complies with standards: SCPI 1991.0, confirmed.

:MEASure [:VOLT] :MAXimum?

[␣ («@1|@2»)]

Positive Peak Voltage

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

Parameters:

(«@1|@2») is the measurement channel
(@1) means input A
(@2) means input B

Complies with standards: SCPI 1991.0, confirmed.

:MEASure [:VOLT] :MINimum?

[␣ («@1|@2»)]

Negative Peak Voltage

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

Parameters:

(«@1|@2») is the measurement channel
(@1) means input A
(@2) means input B

Complies with standards: SCPI 1991.0, confirmed.

:MEASure [:VOLT] :PTPeak?

[␣ (@«1|2»)].

Peak-to-Peak Voltage

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

Parameters:

(@«1|2») is the measurement channel
(@1) means input A
(@2) means input B

Complies with standards: SCPI 1991.0, confirmed.

:MEASure [:VOLT] :RATio?

[␣(@1|@2),(@1|@2)]

Peak-to-Peak Voltage Ratio in dB

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

Parameters:

(@«1|2») is the measurement channel
(@1) means input A
(@2) means input B

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

Positive Slew Rate

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

Parameters:

(@«1|2») is the measurement channel
(@1) means input A
(@2) means input B

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

Negative Slew Rate

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

Parameters:

(@«1|2») is the measurement channel
(@1) means input A
(@2) means input B

:MEASure :PERiod?

[␣ [<expected value> [,<resolution>]][,<(@«1|2|3»)>]]

Period

A traditional period time measurement is performed on a single period. Measuring time set by the :ACQ:APER command does not affect the measurement.

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

Parameters:

<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).

Complies with standards: SCPI 1991.0, confirmed.

:MEASure :PERiod :AVERage?

[␣ [<expected value> [,<resolution>]][,<(@«1|2|3»)>]]

Period

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

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

Parameters:

<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).

Complies with standards: SCPI 1991.0, confirmed.

:MEASure :PHASe?

[␣ [<expected value>[,<resolution>]] [,(@«1|2»),(@«1|2»)]]

Phase

A traditional PHASe measurement is performed.

Parameters:

<expected value> and <resolution> are ignored by the counter

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.

Complies with standards: SCPI 1991-0, approved.

:MEASure «:RISE :TIME | :RTIM»?

[␣ [<lower threshold> [,<upper threshold>[,<expected value>[,<resolution>]]]] [,(@1|@2)]

Rise-time

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.

Parameters:

<lower threshold>, <upper threshold>, <expected value> and <resolution are all ignored by the counter

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

Complies with standards: SCPI 1991.0, confirmed.

:MEASure «:FALL :TIME | :FTIM»?

[␣ [<lower threshold> [,<upper threshold>[,<expected value>[,<resolution>]]]]
[,(@1|@2)]]

Fall-time

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.

Parameters:

<lower threshold>, <upper threshold>, <expected value> and <resolution> are all ignored by the counter

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

Complies with standards: SCPI 1991.0, confirmed.

:MEASure :TINTerval?

␣ (@«1|2»),(@«1|2»)]

Time-Interval

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.

Parameters:

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 :PWIDth?
[␣ [<threshold>] [,<(@«1|2»)>]]

Positive Pulse Width

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.

Parameters

<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.

Complies with standards: SCPI 1991.0, confirmed.

:MEASure :NWIDth?
[␣ [<threshold>] [,<(@«1|2»)>]]

Negative Pulse Width

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.

Parameters

<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.

Complies with standards: SCPI 1991.0, confirmed.

:MEASure :ARRay :TSTAmp?

␣(<array size>)[,(@1)|(@2)]

Time Stamp

Time stamps are taken of all positive and negative trigger level crossings of the selected input channel. The commands **:MEAS** and **:CONF** automatically invoke **:FORMat:TINFormation ON** to get the time stamp data, but when **:FUNC** is used instead, you should normally let it be preceded by the **:FORMat:TINFormation ON** command explicitly. Otherwise the TS[1] values will be 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 are 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.

Parameters:

<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**. See the first paragraph above.

Returned format:

<zero result>,<TS for neg. crossing>,<number of pos. crossings>,<TS for pos. crossing>,<zero result>,<TS for neg. crossing>,<number of pos. crossings>,<TS for pos. crossing>...deadtime...<zero result>,<TS for neg. crossing>... etc.

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

Memory Subsystem

```
:MEMory
    :DELete
                :MACRo␣  '<Macro name>'
    :FREE
    :NSTates?
    :MACRo?
```

Related Common Commands:

```
*DMC
*EMC
*GMC?
*LMC?
*LRN?
*PMC
*RCL
*RMC
*SAV
```

:MEMory :DELete :MACRo

␣ '<Macro name>'

Delete one Macro

This command removes an individual MACRo[1].

Parameters

'<Macro name>' is the name of the macro you want to delete.

<Macro name> is String data that must be surrounded by quotation marks.

See also:

*PMC, if you want to delete all macros.

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

:MEMory :FREE :MACRo?

Memory Free for Macros

This command gives information of the free memory available for MACRos in the counter. If no macros are specified, 1160 bytes are available.

Returned format:

<Bytes available>, <Bytes used>↲

Complies with standards: SCPI 1991.0, confirmed.

Memory States

The Number of States query (only) requests the number of *SAV/ *RCL instrument setting memory states available in the counter. The counter 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.)

Returned format:

<the number of states available>↵

Complies with standards: SCPI 1991.0, confirmed

This page is intentionally left blank.

Read Function

Perform Measurement and Read Data

```
:READ
    [:SCALar]?
    :ARRay? ␣  <Array Size>|MAX
```

:READ?

Read one Result

The read function performs new measurements and reads out a measuring result without reprogramming the counter. Using the :READ? query in conjunction with the :CONFigure command gives you a measure capability where you can fine tune the measurement.

If the counter 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? to fetch other measuring results from the output buffer. The :READ? query is identical to :ABORt;:INITiate;:FETCh?

Returned format: <data>↵

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

Example:

SEND→ :CONF:FREQ;:INP:FILT ␣ ON;:READ?

This example configures the counter to make a standard frequency measurement with the 100 kHz filter on. The counter is triggered, and data from the measurement are read out with the :READ? query.

SEND→ :READ?

This makes a new measurement and fetches the result without changing the programming of the counter.

Type of command: Aborts all previous measurement commands if *WAI is not used.

Complies with standards: SCPI 1991.0, confirmed.

Read an array of results

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 ␣ or ␣ :MEASure:ARRAy?.

The :READ:ARRay? query is identical to:
:ABORt;:INITiate;:FETCh:ARRay?␣<array size for FETCh>

The <array size for FETCh*> does not tell* :READ *to make that many measurements, only to fetch that many results.* :CONF:ARR, ␣ :MEAS:ARR, :ARM:LAY1:COUN or :TRIG:LAY1:COUN sets the number of measurements.

Parameters:

<array size for FETCh> sets the number of measuring results in the array. This size must be equal or less than the number of measurements specified with :CONFigure.

MAX means that all the results in the output buffer will be fetched.

Returned format: <data>[,<data>]↲
The format of the returned data is determined by the format commands :FORMat and :FORMat:FIXed.

SEND→ :ARM:COUN ␣ 10;:READ:ARR? ␣ 5

This example configures the counter to make an array of 10 standard measurements. The counter is triggered and data from the first five measurements are read out with the :READ? query.

Type of command: Aborts all previous measurement commands if *WAI is not used.

Complies with standards: SCPI 1991.0, confirmed.

This page is intentionally left blank.

Sense Command Subsystem

■ Sense Subsystem Command Tree

[:SENSe]

:ACQuisition

:APERture ␣ <meas time> | MIN | MAX

:HOFF

[:STATe] ␣ ON | OFF

:MODE ␣ TIME

:TIME ␣ <numeric value> | MIN | MAX

:FREQuency

:BURSt

:APERture ␣ <numeric value> | MIN | MAX

:STARt

:DELay ␣ <numeric value> | MIN | MAX

:SYNC

:PERiod ␣ <numeric value> | MIN | MAX

:PREScaler

[:STATe] ␣ ON | OFF

:RANGe

:LOWer ␣ <Minimum frequency for autotrigger> | MIN | MAX

:FUNCtion ␣ 'Measuring function [Primary channel [, Secondary channel]]

:ROSCillator

:SOURce ␣ INTernal | EXTernal

:ACQuisition :APERture

␣ «<Decimal value > | MIN | MAX»

Set the Measurement Time

Sets the gate time for one measurement.

If you want to switch between Average and Single measurements, use the :AVERage:STATe␣ ON|OFF *in the Sense Subsystem.*

When Single is selected and an array measurement is done, the Measurement Time, set by `:Acquisition:APERture`, sets the time between the measurements in the array. This means that if you want a very high speed, you must set `:AVER:STATE ␣` OFF and `:ACQ:APER ␣` MIN.

Parameters: <decimal value> is 20 ns to 1000 s.
MIN gives 20 ns and MAX gives 1000 s.

Returned format: <Decimal value >↵

***RST condition:** 10 ms

SYST:PRESet condition: 200 ms

:ACQuisition :HOFF

␣ <boolean>

Hold Off On/Off

Switches the Hold Off function On/Off.

Parameters:

<Boolean> = 1 / ON | 0 / OFF

Returned format: 1 | 0↵

***RST condition:** OFF

:ACQuisition :HOFF :TIME

␣ «<Decimal value> |MIN|MAX»

Hold Off Time

Sets the Hold Off time value.

Parameters:

<Decimal data> = a number between 20E–9 and 2.0

Returned format:

<Decimal value>↵

***RST condition:**

200 µs

:FREQuency :BURSt :APERture

␣ «<Numeric value>|MIN|MAX»

Burst Measuring Time

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

Parameters: <Numeric value> = a number between 20 ns and 2 s.

Returned format: <Numeric value>↵

***RST condition:** 200 µs

:FREQuency :BURSt :PREScaler [:STATe]

␣ <Boolean>

Prescaler ÷2 on Input A & Input B

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

Parameters: <Boolean> = (1/ON | 0/OFF)

Returned format: 1|0↵

***RST condition:** ON

:FREQuency :BURSt :STARt :DELay

␣ «<Numeric value>|MIN|MAX»

Burst Start Delay

Sets the burst start delay, i.e. the time length between the burst start and the actual start of the burst measuring time. This parameter is used for controlling the point of time when a measurement sample is taken.

Parameters: <Numeric value> = a number between 20 ns and 2 s.

Returned format: <Numeric value>↵

***RST condition:** 200 µs

:FREQuency :BURSt :SYNC :PERiod

␣ «<Numeric value>|MIN|MAX»

Burst Sync Delay

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

Parameters: <Numeric value> = a number between 1 μs and 20 s.

Returned format: <Numeric value>↵

***RST condition:** 200 μs

:FREQuency :RANGe :LOWer

␣ «<Numeric value>|MIN|MAX»

High Speed Voltage Measurements

Use this command to speed up voltage measurements and Autotrigger functions when you don't need to measure on low frequencies.

Time to determine trigger levels (typical)			
Measuring function	**Min. frequency limit (1 Hz)**	**Default (20 Hz)**	**Max. frequency limit (50 kHz)**
Freq A	8 s	400 ms	20 ms
Time A-B			

Parameters:

<Numeric value> between 1 and 50000 (Hz)
MIN gives 1 Hz
MAX gives 50 kHz

Returned format: <Numeric value>↵

***RST condition:** 20 (Hz)

Complies with standards: SCPI 1991.0, confirmed.

:FUNCtion

␣ '<Measuring function>[_<Primary channel> [,<Secondary channel>]]'

Select Measuring Function

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

Parameters:

<Measuring function> is the function you want to select, according to the SENSe subsystem command trees on page 8-75.

<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.

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

Returned format: "<Measuring function>_<Primary channel>[,<Secondary channel>]"↵

Example Select a pulse period measurement on input A (channel 1):
Send → `:FUNC ␣ 'PER ␣ 1'`

***RST condition:** FREQuency_1

Complies with standards: SCPI 1991.0, confirmed.

■ Functions and Channels

:FREQuency [:CW]	[␣ ‘1 | 2 | 3 | 4 | 6‘]
:FREQuency [:CW] :RATio	[␣ ‘1 | 2 | 3 , 1 | 2 | 3 ‘]
:FREQuency :BURSt	[␣␣ ‘1 | 2 | 3 ‘]
:FREQuency :PRF	[␣ ‘1 | 2 | 3 ‘]
:NCYCles	[␣ ‘1 | 2 | 3 ‘]
: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 ‘]
[:VOLT]:MAXimum	[␣␣ ‘1 | 2 ‘]
[:VOLT]:MINimum	[␣␣ ‘1 | 2 ‘]
[:VOLT]:PTPeak	[␣␣ ‘1 | 2 ‘]
[:VOLT]:RATIO	[␣ ‘1 | 2 , 1 | 2 ‘]

■ Input Channels

1 means input A

2 means input B

3 means input C (RF input option)

4 means input E (rear panel arming input)

6 means the internal reference

:ROSCillator :SOURce

␣ «INT|EXT|AUTO»

Select Reference Oscillator

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, external reference will be used if present.

Returned format: <INT|EXT|AUTO>↵

***RST condition:** AUTO

Complies with standards: SCPI 1991.0, confirmed.

Status Subsystem

:STATus

:DREGister0

:ENABle ␣ <bit mask>
[:EVENt]?

:OPERation

:CONDition?

:ENABle ␣ <bit mask>
[:EVENt]?

:QUEStionable

:CONDition?

:ENABle ␣ <bit mask>
[:EVENt]?

:PRESet

■ Related Common Commands:

*CLS

*ESE ␣ <bit mask>
*ESR?

*PSC ␣ <bit mask>
*SRE ␣ <bit mask>
*STB?

:STATus :DREGister0?

Read Device Status Event Register

This query reads out the contents of the Device Event Register. Reading the Device Event Register clears the register. See .

Returned format:

<dec.data> = the sum (between 0 and 6) of all bits that are true. See table below:

Bit No.	Weight	Condition
2	4	Last measurement below low limit.
1	2	Last measurement above high limit.

:STATus :DREGister0 :ENABle

␣ <bit mask>

Enable Device Status Reporting

This command sets the enable bit of the Device Register 0.

Parameters:

<dec.data> = the sum (between 0 and 6) of all bits that are true. See table below:

Bit No.	Weight	Condition
2	4	Enable monitoring of low limit
1	2	Enable monitoring of high limit

Returned format: <bit mask>↵

:STATus :OPERation :CONDition?

Read Operation Status Condition Register

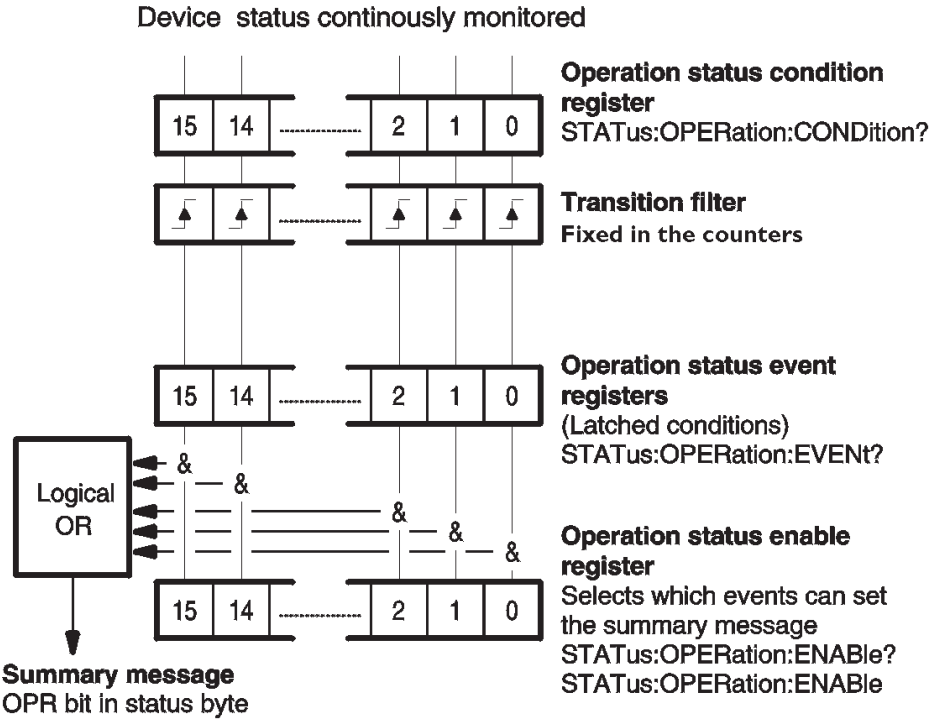
Reads out the contents of the operation status condition register. This register reflects the state of the measurement process. See table below.

Returned Format:

<Decimal data> = the sum (between 0 and 368) of all bits that are true. See table below:

Bit No.	Weight	Condition
11	2048	Computing statistics
10	1024	In limit
9	512	Using internal reference
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

Complies with standards: SCPI 1991.0, confirmed.



:STATus :OPERation :ENABle

␣ <Decimal data>

Enable Operation Status Reporting

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 on page -.

An enabled bit will set bit #7, OPR (Operation Status Bit), in the Status Byte Register if the enabled event occurs. See also status reporting on page 3-10.

Power-on will clear this register if power-on clearing is enabled via *PSC.

Parameters: <dec.data> = the sum (between 0 and 368) of all bits that are true. See table below:

Bit No.	Weight	Condition
8	256	No measurement
6	64	Waiting for bus arming
5	32	Waiting for triggering and/or external arming
4	16	Measurement

Returned Format: <Decimal data>↵

Example:

SEND→ :STAT:OPER:ENAB ␣ 288

In this example, waiting for triggering, bit 5, and Measurement stopped, bit 8, will set the OPR-bit of the Status Byte. (This method is faster than using *OPC if you want to know when the measurement is ready.)

Complies with standards: SCPI 1991.0, confirmed.

:STATus:OPERation?

Read Operation Status, Event

Reads out the contents of the operation event status register. Reading the Operation Event Register clears the register. See figure on page -.

Returned Format: <Decimal data>↵

<dec.data> = the sum (between 0 and 368) of all bits that are true. See table on page 8-86.

Complies with standards: SCPI 1991.0, confirmed.

:STATus :PRESet

Enable Device Status Reporting

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 counter 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 counter, 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.

:STATus :QUEStionable :CONDition?

Read Questionable Data/Signal Condition Register

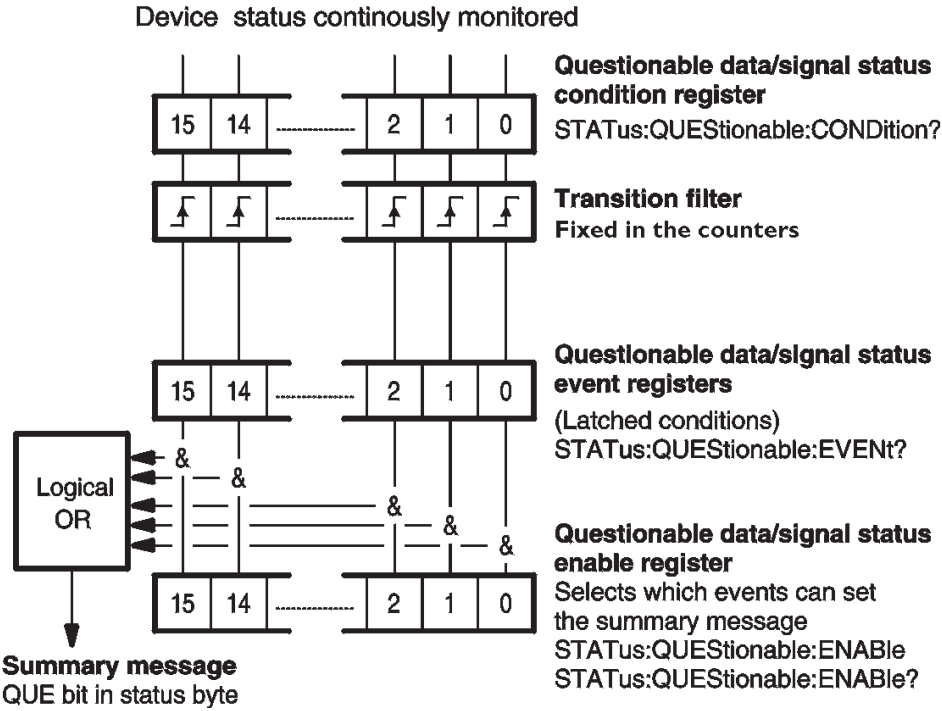
Reads out the contents of the status questionable condition register.

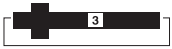
Returned Format:

<dec.data> = the sum (between 0 and 17920) of all bits that are true. See table below:

Bit No.	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

Complies with standards: SCPI 1991.0, confirmed.





:STATus :QUEStionable :ENABle

␣ <Decimal data>

Enable Questionable Data/Signal Status Reporting

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 on page -.

An enabled bit will set bit #3, QUE (Questionable Status Bit), in the Status Byte Register if the enabled event occurs. See also status reporting on page 3-10.

Power-on will clear this register if power-on clearing is enabled via *PSC.

Parameters:

<dec.data> = the sum (between 0 and 17920) of all bits that are true. See the table on page 8-88.

Returned Format: <Decimal data> ↲

Example:

Send → :STAT:QUES:ENAB ␣ 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.

Complies with standards: SCPI 1991.0, confirmed.

:STATus :QUEStionable?

Read Questionable Data/Signal Event Register

Reads out the contents of the status questionable event register. Reading the Status Questionable Event Register clears the register. See figure on page -.

Returned Format:

<dec.data> = the sum (between 0 and 17920) of all bits that are true. See the table on page 8-88.

Complies with standards: SCPI 1991.0, confirmed.

This page is intentionally left blank.

System Subsystem

```
:SYSTem
    :COMMunicate
        :GPIB
            :ADDRess  ␣ <Numeric value> | MIN | MAX
    :ERRor?
    :LANGuage         ␣ NATive | COMPatible
    :PRESet
    :SET              ␣ <Block data>
    :TEMPerature?
    :TOUT
        [:STATe]      ␣ ON | OFF
        :TIME             ␣ <timeout value>
    :UNPRotect
```

■ Related common commands:

```
*IDN?
*OPT?
*PUD ␣ <arbitrary block program data>
*RST
```

:SYSTem :COMMunicate :GPIB :ADDRess

␣ «<Numeric value>|MAX|MIN» [,«<Numeric value>|MAX|MIN»]

Set 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.

Parameters:

<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 '90'.

`[:SELF]` ␣ This optional parameter is accepted by the '90'.

Returned format:> <Numeric value>↲

Example:

SEND→ `:SYST:COMM:GPIB:ADDR ␣ 12`

This example sets the bus address to 12.

Complies with standards: SCPI 1991.0, confirmed.

:SYSTem :ERRor?

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.

Returned format:

<error number>,"<Error Description String>"↲

Where:

<Error Description String> = an error description as ASCii text.

See also: Chapter 7, error messages.

Complies with standards: SCPI 1991.0, confirmed.

Preset

This command recalls the same default settings that are entered when the front panel key AUTOSET is pressed twice within two seconds (double-clicking).

These are not exactly the same settings as after *RST, :SYST:PRES *gives 200 ms Measurement Time and signal detection ON, while* *RST *gives 10 ms Measurement Time and signal detection OFF.* *RST *also sets the instrument in the same condition as after* :INIT:CONT OFF*, whereas* :SYST:PRES *activates the opposite setting, i.e.* :INIT:CONT ON*.* *See page 8-40*

See also: Default settings on page 2-2.

Complies with standards: SCPI 1991.0, confirmed.

:SYSTem :SET
␣ <Block data>

Read or Send Settings

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 `:SYST:SET?`. The query form of this command returns a definite block data element.

Parameters:

<Block data> is the instrument setting previously retrieved via the `:SYSTem:SET?` query.

Returned format: <Block data>↵

Where:

<Block data> is #292<92 data bytes> for the '90'

SEND→ `:SYST:SET?`

READ← `#2764...-␣ .␣ .Çä......d...␣`
`+.................-.c-.............␣ ␣ ?.............d`

Complies with standards: SCPI 1991.0, confirmed.

:SYSTem :LANGuage

␣ NATive | COMPatible

Select GPIB Mode

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 counters 53131 and 53132.

The command set described in this manual refers to the native mode only.

:SYSTem :TOUT

␣ <Boolean>

Timeout On/Off

This command switches on or off the timeout. When timeout is enabled, the measurement attempt will be abandoned when the time set with `:SYST:TOUT:TIME` has elapsed. A zero result will be sent to the controller instead of a measurement result and the timeout bit in the STATus QUEStionable register will be set.

Returned format: 0 means no timeout; 1 means that the timeout set by `:SYSTem:TOUT:TIME` is used.

Example:

SEND→ `:SYST:TOUT ␣ 1;TOUT:TIME ␣ 0.5;:STAT:QUES:ENAB ␣ 1024;:*SRE ␣ 8`

This example turns on timeout, sets the timeout to 0.5 s, enables status reporting of questionable data at timeout, and enables service request on questionable data.

```
SEND→ *STB?       If bit 3 in the status byte is set, read the
      questionable data status.
SEND→ :STAT:QUES:EVEN?              This query reads the ques-
      tionable data status.
READ← «1024|0»          1024 means timeout has occurred, and 0
      means no timeout.
```

***RST condition:** 0

Timeout, Set

This command sets the timeout in seconds.

The timeout starts when a measurement starts and, if no result is obtained when the set timeout has elapsed the measurement is terminated.

Note that you must enable timeout using :SYST:TOUT_ON for this setting to take effect.

Parameters:

<Numeric value> is the timeout in seconds. The range is 0.01 to 1000 (s)

MIN gives 0.01 s

MAX gives 1000 s

Returned format: <Numeric value>↵

***RST condition:** 0.1 (s)

Complies with standards: SCPI 1991.0, confirmed.

:SYSTem :TEMPerature?

Read Temperature

This command returns the temperature in°C at the fan control sensor inside the instrument housing.

Returned format: <Numeric value>↵

Example:
SEND→ `:SYST:TEMP?`
READ← `50`

:SYSTem :UNPRotect

Unprotect

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.

Example

Send → `:SYST:UNPR; *PUD ␣ #240Calibrated ␣ 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 Subsystem

:TEST

:SELect ␣ RAM | ROM | LOGic | DISPlay | ALL

■ Related common command:

*TST

:TEST :SELect

␣ «RAM | ROM | LOGic | DISPlay | ALL»

Select Self-tests

Selects which internal self-tests shall be used when self-test is requested by the *TST command.

Returned format:

«RAM | ROM | LOGic | DISPlay | ALL»↲

***RST condition:** ALL

Trigger Subsystem

```
:TRIGger
 [ :STARt | :SEQuence [ 1 ] ]
        [ :LAYer [ 1 ] ]
              :COUNt␣   <Numeric value> | MIN | MAX
 :SOURCE
 :TIMER
```

■ Related common command:

*TRG

:TRIGger:COUNt

␣ «<Numeric value> | MIN | MAX»

No. of Triggerings on each Ext Arm start

Sets how many measurements the instrument should make for each ARM:STARt 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.

The actual number of measurements made on each INIT *equals to:*
(:ARM:START:COUN)*(:TRIG:START:COUNT)

Parameters:

<Numeric value> is a number between 1 and 65535.
MAX gives 65535
MIN gives 1

Example:
SEND→ `:TRIG:COUN ␣ 50`

Returned format: <Numeric value>↵

***RST condition:** 1

Complies with standards: SCPI 1991.0, confirmed.

:TRIGger:SOURce

␣TIMer | IMMediate

Pacing

Enables or disables the pacing function, i.e. the sample rate control. The pacing time is set by the **:TRIG:TIM** command.

Parameters:

TIMer - enables pacing
IMMediate - disables pacing

***RST condition:** IMM

Set Pacing Time

This command sets the sample rate, for instance in conjunction with the statistics functions.

Parameters:

<Numeric value> is a time length between 2 μs and 500 s, entered in seconds.
MIN means 2 μs.
MAX means 500 s.

Returned format: <Numeric value>↵

***RST condition:** 20 ms

This page is intentionally left blank.

Common Commands

*CLS

*DMC ␣ <Macro label> , <Program messages>
*EMC ␣ ␣ <Decimal data>
*ESE ␣ <Decimal data>
*ESR?

*GMC? ␣ <Macro label>
*IDN?

*LMC?

*LRN?

*OPC

*OPC?

*OPT?
*PMC

*PSC ␣ <Decimal data>
*PUD ␣ ␣ <Arbitrary block program data>
*RCL ␣ ␣ <Decimal data>
*RMC ␣ <Macro name>
*RST

*SAV ␣ <Decimal data>
*SRE ␣ <Decimal data>
*STB?

*TRG

*TST?

*WAI

*CLS

Clear Status Command

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?.

Example:

Send → *CLS

Complies with standards: IEEE 488.2 1987.

Define Macro

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 will be executed.

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

Parameters:

<Macro label> = 1 to 12-character macro label. (String data must be surrounded by " " or ' ' as in the example below.)

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

Example 1:

SEND→ `*DMC 'FREQUENCY?',":FUNC ␣ 'FREQ ␣ 1';:INP:LEV:AUTO ␣ ON ;:ARM:START:LAY2:SOURCE ␣ BUS;:INIT:CONT ␣ ON;*TRG"`

This example defines a macro called FREQUENCY?.

SEND→ `FREQUENCY?`

The macro makes a single frequency measurement with automatic trigger level setting and places the result in the output queue. (Macros must be enabled; otherwise, the `:FREQUENCY?` query will not execute, see ☛EMC)).

READ←+31.415926536E+006

Example 2:

SEND→ `*DMC ␣ 'AUTOFILT',":INP:LEV:AUTO ␣ $1;:INP:FILT ␣ $1;:INP2:LEV:AUTO ␣ $1;:INP2:FILT ␣ $1"`

This example defines a macro called AUTOFILT which takes one Boolean argument, i.e. «ON|OFF» ($1) .

SEND→ `AUTOFILT ␣ OFF`

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

Complies with standards: IEEE 488.2 1987.

*EMC

␣<Decimal data>

Enable Macros

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.)

Parameters:

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

Note that 1 or 0 is <Decimal data>, not <Boolean>!
ON|OFF is `not` *allowed here!*

Returned format: «1|0» ↲

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

Example:

SEND→`*EMC ␣ 1`

Enables macro expansion and execution.

Complies with standards: IEEE 488.2 1987.

Standard Event Status Enable

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 also status reporting on page 3-10.

Parameters: <dec.data> = the sum (between 0 and 255) of all bits that are true.

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

Returned Format: <Decimal data> ↲

Example:

SEND→ `*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.

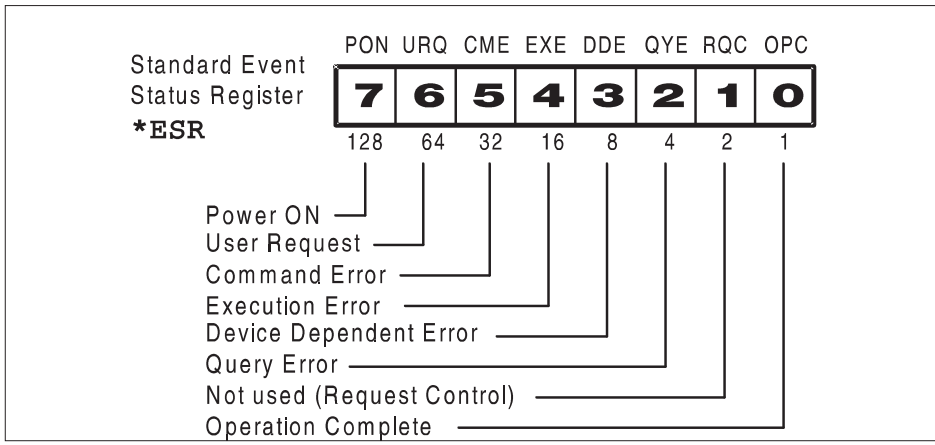


Figure 8-3 *Bits in the standard event status register.*

Complies with standards: IEEE 488.2 1987.

*ESR?

Event Status Register

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

Returned Format:

<dec.data> = the sum (between 0 and 255) of all bits that are true. See table on page 8-107.

Complies with standards: IEEE 488.2 1987.

*GMC?

␣ < macro label>

Get Macro Definition

This command makes the counter respond with the current definition of the given macro label.

Parameters:

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

Returned Format: <Block data>↵

Example:

SEND→ `*GMC? ␣ 'AMPLITUDE?'`

Gives a block data response, for instance:

READ←

#255:FUNC 'FREQ 1';:INP:HYST:AUTO ONCE;:INP:HYST?;INP:LEV?

Complies with standards: IEEE 488.2 1987.

*IDN?

Identification query

Reads out 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>.

Example

SEND →`*IDN?`

READ← `<MANUFACTURER>, <MODEL>, 1234567, V1.01 28 Jun 2004`

Complies with standards: IEEE 488.2 1987.

*LMC?

Learn Macro

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

Returned Format:

<String> { ,<String> }↲

<String> = a Macro label. (String data will be surrounded by " " as in the example below.)

Example:

SEND→ `*LMC?`

May give the following response:

READ←`"AUTOFILT","AMPLITUDE?"`

Complies with standards: IEEE 488.2 1987.

*LRN?

Learn Device Setup

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.

Returned Format:

:SYST:SET_<Block data>↵

Where:

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

Example

SEND→ *LRN?

Complies with standards: IEEE 488.2 1987.

*OPC

Operation Complete

The Operation Complete command causes the device to set the operation complete bit in the Standard Event Status Register when all pending selected device operations have been finished. See also Example 4 in Chapter .

Example:

Enable OPC-bit
SEND→ *ESE ␣ 1

Start measurement (INIT). *OPC will set the operation complete bit in the status register when the measurement is done.
SEND→ :INIT;*OPC

Wait 1s for the measurement to stop. Read serial poll register, will reset service request
SPOLL

Check the Operation complete bit (0) in the serial poll byte. If it is true the measurement is completed and you can fetch the result.
SEND→ FETCh?

Then read the event status register to reset it:
SEND→ *ESR?

If bit 0 is false, abort the measurement.
SEND→ :ABORt

Complies with standards: IEEE 488.2 1987.

*OPC?

Operation Complete Query

Operation Complete query. The Operation Complete query places an ASCii character 1 into the device's Output Queue when all pending selected device operations have been finished.

Returned Format: 1↵

See also:
Example 6 is Chapter .

Complies with standards: IEEE 488.2 1987.

*OPT?

Option Identification

Response is a list of all detectable options present in the instrument, with absent options represented with an ASCii '0'.

Returned format:
<Timebase option>,<Prescaler option>, <Reserved>↵

Where:

<Timebase option> = Standard|Option 30|Option 40
<Prescaler option> = 0|Option 10|Option 13|Option 14B
0 for prescaler option means that no prescaler is installed.
The position <Reserved> is 0 until further notice.

Complies with standards: IEEE 488.2 1987.

*PMC

Purge Macros

Removes all macro definitions.

Example: *PMC

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

Complies with standards: IEEE 488.2 1987.

*PSC

␣ <Decimal data>

Power-on Status Clear

Enables/disables automatic power-on 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)

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

Parameters: <Decimal data> = a number that rounds to 0 turns off automatic power-on clearing. Any other value turns it on.

Returned Format: «1 | 0» ↵

1 is enabled and 0 is disabled.

Example: *PSC ␣ 1

This example enables automatic power-on clearing.

Complies with standards: IEEE 488.2 1987.

Protected User Data

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. A typical use would be to hold calibration information, usage time, inventory control numbers, etc.

The content at delivery is: #234 FACTORY CALIBRATED ON: 19YY-MM-DD

– YY = year, MM = month, DD = day

Returned format: <Arbitrary block response data>↵

– Where:
<arbitrary block program data> is the data last programmed with *PUD.

Example

Send → `:SYST:UNPR; *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.

Complies with standards: IEEE 488.2 1987.

Recall

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.

Parameters:
<Decimal data> = a number between 0 and 19.

Example:
SEND→ `*RCL ␣ 10`↵

Complies with standards: IEEE 488.2 1987.

*RMC

␣ '<Macro name>'

Delete one Macro

This command removes an individual MACRo.

Parameters:

'<Macro name>' is the name of the macro you want to delete.

<Macro name> is String data that must be surrounded by quotation marks.

See also:

*PMC, if you want to delete all macros.

*RST

Reset

The Reset command resets the counter. It is the third level of reset in a 3-level reset strategy, and it primarily affects the counter functions, not the IEEE 488 bus.

The counter settings will be set to the default settings listed on page 2-2. All previous commands are discarded, macros are disabled, and the counter is prepared to start new operations.

Example: *RST

See also:

Default settings on page 2-2.

Complies with standards: IEEE 488.2 1987.

Save

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

Note that memory positions 1 to 10 can be protected from the front panel USER OPT menu. If this has been done, use the `:SYSTem:UNPRotect` command to alter these memory positions.

Parameters

<Decimal data> = a number between 1 and 19.

Example:

SEND→ `*SAV ␣ 11↲`

Complies with standards: IEEE 488.2 1987

*SRE

␣ <Decimal data>

Service Request Enable

The Service Request Enable command sets 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.

Parameters: <dec.data> = the sum (between 0 and 255) of all bits that are true. See table below:

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

Returned Format: <Integer>↵

Where:

<Integer> = the sum of all bits that are set.

Example: `*SRE ␣ 16`

In this example, the counter generates a service request when a message is available in the output queue.

Complies with standards: IEEE 488.2 1987.

Status Byte Query

Reads out 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.

Returned Format:

<Integer> = the sum (between 0 and 255) of all bits that are true. See table below:

Status Byte Register (1 = true)			
Bit	*Weight*	*Name*	*Condition*
7	128	OPR	Enabled operation status has occurred.
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

See also: If you want to read the status byte with the RQS bit, use serial poll.

Complies with standards: IEEE 488.2 1987.

Trigger

The trigger command *TRG starts the measurement and places the result in the output queue.

It is the same as:
`:ARM:STARt: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 has been received, and parsed by the counter.

However, GET is much faster than *TRG, since GET is a hardware signal that does not have to be parsed by the counter.

Example:

SEND→ `:ARM:START:LAY2:SOURCE ␣ BUS`
SEND→ `:INIT:CONT ␣ ON`
SEND→ `*TRG`
READ← `+3.2770536E+004`

Type of Command:

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

Complies with standards: IEEE 488.2 1987.

*TST?

Self Test

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.

Returned Format: <Integer>↵

Where:

<Integer> = a number indicating errors according to the table below.

<Integer> =	Error
0	No Error
1	RAM Failure
2	ROM Failure
4	Logic Failure
8	Display Failure
16	
32	

Complies with standards: IEEE 488.2 1987

*WAI

Wait-to-continue

The Wait-to-Continue command prevents the device from executing any further commands or queries until execution of all previous commands or queries has been completed.

Example:

SEND→`:MEAS:FREQ?; *WAI;:MEAS: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.

READ← `+5.1204004E+002;+1.250030E-001`

Complies with standards: IEEE 488.2 1987.

Chapter 9

Index

Index

D

E

F

M

N

O

P

Q

R

S

T

U

V

W

X