SCPI Commands

CMA5000(a) Platform

Third Edition

For safety and warning information, please read this manual before attempting to use the equipment.
Keep this manual with the equipment.

ANRITSU CORPORATION

Part No.: 42541

SCPI Commands – CMA5000(a) Standard OTDR Application
10 October 2009 (Third Edition)

Copyright © 2009, ANRITSU CORPORATION.
All rights reserved. No part of this manual may be reproduced without the prior written permission of the publisher.
The contents of this manual may be changed without prior notice.

Composed in the USA

About This Manual

This operation manual describes the SCPI (Standard Commands for Programmable Instruments) commands for the CMA5000 Platform.

Table of Contents

This section details the SCPI commands for the CMA5000 platform.

About This Manual iii
1 Star (IEEE 488.2) Subsystem Commands 1
1.1 *CLS Clear Status Command 1
1.2 *ESE Standard Event Status Enable Command 1
1.3 *ESE? Standard Event Status Enable Query 1
1.4 *ESR? Standard Event Status Register Query 2
1.5 *IDN? Identification Query 2
1.6 *OPC Operation Complete Command 2
1.7 *OPC? Operation Complete Query 3
1.8 *RST Reset Command 3
1.9 *SRE Service Request Enable Command 4
1.10 *SRE? Service Request Enable Query 4
1.11 *STB? Read Status Byte Query 5
1.12 *TST? Self-Test Query 5
1.13 *WAI Wait-to-Continue Command 5
2 System Subsystem Commands 6
2.1 SYSTem:ERRor? 6
2.2 SYSTem:VERSion? 6
3 Status Subsystem Commands 7
3.1 STATus:OPERation[:EVENt]? 7
3.2 PERation:CONDition? 7
3.3 STATus:OPERation:BIT#:CONDition? 7
3.4 STATus:OPERation:BIT#:ENABle 7
3.5 STATus:OPERation:BIT#:ENABle? 8
3.6 STATus:OPERation:BIT#[:EVENt]? 8
3.7 STATus:OPERation:ENABle 8
3.8 STATus:OPERation:ENABle? 8
3.9 STATus:OPERation:INSTrument:CONDition? 9
3.10 STATus:OPERation:INSTrument:ENABle 9
3.11 STATus:OPERation:INSTrument:ENABle? 9
3.12 STATus:OPERation:INSTrument[:EVENt]? 9
3.13 STATus:OPERation:INSTrument:ISUMmary#:CONDition? 10
3.14 STATus:OPERation:INSTrument:ISUMmary#:ENABle 10
3.15 STATus:OPERation:INSTrument:ISUMmary#:ENABle? 10
3.16 STATus:OPERation:INSTrument:ISUMmary#[:EVENt]? 10
3.17 STATus:QUEStionable[:EVENt]? 11
3.18 STATus:QUEStionable:CONDition? 11
3.19 STATus:QUEStionable:BIT#:CONDition? 11
3.20 STATus:QUEStionable:BIT#:ENABle 11
3.21 STATus:QUEStionable:BIT#:ENABle? 12

3.22 STATus:QUEStionable:BIT#[:EVENt]? ... 12
3.23 STATus:QUEStionable:ENABle ... 12
3.24 STATus:QUEStionable:ENABle? ... 12
3.25 STATus:QUEStionable:INSTrument:CONDition? ... 13
3.26 STATus:QUEStionable:INSTrument:ENABle ... 13
3.27 STATus:QUEStionable:INSTrument:ENABle? ... 13
3.28 STATus:QUEStionable:INSTrument[:EVENt]? ... 13
3.29 STATus:QUEStionable:INSTrument:ISUMmary#:CONDition? ... 14
3.30 STATus:QUEStionable:INSTrument:ISUMmary#:ENABle ... 14
3.31 STATus:QUEStionable:INSTrument:ISUMmary#:ENABle? ... 14
3.32 STATus:QUEStionable:INSTrument:ISUMmary#[:EVENt]? ... 14
3.33 STATus:PRESet ... 15
4 Instrument Subsystem Commands ... 16
4.1 INSTrument:CATalog? ... 16
4.2 INSTrument:CATalog:FULL? ... 16
4.3 INSTrument:NSELect ... 16
4.4 INSTrument:NSELect? ... 16
4.5 INSTrument[:SELect] ... 17
4.6 INSTrument[:SELect]? ... 17
4.7 INSTrument:STATe ... 17
4.8 INSTrument:STATe? ... 17
5 Configuration ... 18
5.1 General Requirements ... 18
5.1.1 Client and Server ... 18
5.1.2 Connection Port ... 18
5.1.3 Startup ... 18
5.2 Connecting from Hyper Terminal ... 18
5.3 Writing Custom C Client ... 20
5.4 Writing Custom Java Client ... 25
5.5 Generic Error Codes ... 29

1 Star (IEEE 488.2) Subsystem Commands

1.1 *CLS Clear Status Command

Syntax:	*CLS
Description:	The Clear Status command *CLS clears all the event registers summarized in the Status Byte register. Except for the output queue, all queues summarized in the Status Byte register are emptied. The error queue is emptied. Neither the Standard Event Status Enable register, nor the Service Request Enable register are affected by this command. The command's effect on individual logical instrument depends on the instrument implementation and is described in the instrument's documentation.
Parameters:	None
Response:	None
Example:	`*CLS`

1.2 *ESE Standard Event Status Enable Command

Syntax:	*ESE<wsp><value>
Description:	The standard Event Status Enable command (*ESE) sets bits in the Standard Event Status Enable register. A 1 in a bit in the enable register enables the corresponding bit in the Standard Event Status register. The register is cleared at power-on. The *RST and *CLS commands do not affect the register.
Parameters:	The bit value for the register (a **short** or a **float**): 7 (MSB) Power On 128 6 User Request 64 5 Command Error 32 4 Execution Error 16 3 Device Dependent Error 8 2 Query Error 4 1 Request Control 2 0 (LSB) Operation Complete 1
Response:	None
Example:	`*ESE 21`

1.3 *ESE? Standard Event Status Enable Query

Syntax:	*ESE?
Description:	The standard Event Status Enable query *ESE? returns the contents of the Standard Event Status Enable register (see *ESE for information on this register).
Parameters:	None
Response:	The bit value for the register (a **short** value).
Example:	`*ESE? –> 21<END>`

1.4 *ESR? Standard Event Status Register Query

Syntax:	*ESR?
Description:	The standard Event Status Register query *ESR? returns the contents of the Standard Event Status register. The register is cleared after being read.
Parameters:	None
Response:	The bit value for the register (a **short** value): 7 (MSB) Power On 128 6 User Request 64 5 Command Error 32 4 Execution Error 16 3 Device Dependent Error 8 2 Query Error 4 1 Request Control 2 0 (LSB) Operation Complete 1
Example:	`*ESR? –> 22<END>`

1.5 *IDN? Identification Query

Syntax:	*IDN?
Description:	The Identification query *IDN? gets the instrument identification over the interface.
Parameters:	None
Response:	The identification string terminated by <END>
Example:	`*IDN? –> ANRITSU, CMA5000, 1990.0 1.00<END>`

1.6 *OPC Operation Complete Command

Syntax:	*OPC
Description:	A device is in the Operation Complete Command Active State (OCAS) after it has executed *OPC. The device returns to the Operation Complete Command Idle State (OCIS) whenever the No Operation Pending flag is TRUE, at the same time setting the OPC bit of the SESR TRUE. The following events force the device into OCIS without setting the No Operation Pending flag TRUE and without setting the OPC bit of the SESR: **.** Power-on **.** the Device Clear Active State is asserted on the interface. **.** *CLS **.** *RST
Parameters:	None
Response:	None
Example:	`*OPC`

1.7 *OPC? Operation Complete Query

Syntax:	*OPC?
Description:	A device is in the Operation Complete Query Active State (OQAS) after it has executed *OPC?. The device returns to the Operation Complete Query Idle State (OQIS) whenever the No Operation Pending flag is TRUE, at the same time placing a “1" in the Output Queue. The following actions cancel the *OPC? query (and put the instrument into Operation Complete, Command Idle State): . Power-on . the Device Clear Active State is asserted on the interface. . *CLS . *RST
Parameters:	None
Response:	0<END> or 1<END>
Example:	`*OPC? –> 1<END>`

1.8 *RST Reset Command

Syntax:	*RST
Description:	The ReSeT command *RST sets the instrument to reset setting (standard setting) stored in internal storage. Pending *OPC? actions are cancelled. The instrument is placed in the idle state awaiting a command. The *RST command clears the error queue. The following are not changed: . Output queue . Service Request Enable register (SRE) . Standard Event Status Enable register (ESE)
Parameters:	None
Response:	None
Example:	`*RST`

1.9 *SRE Service Request Enable Command

Syntax:	*SRE<wsp><value>
Description:	The standard Service Request Enable command (*SRE) sets bits in the Service Request Enable register. A 1 in a bit in the enable register enables the corresponding bit in the Service Request Enable register. The register is cleared at power-on. The *RST and *CLS commands do not affect the register.
Parameters:	The bit value for the register (a **short** or a **float**): 7 Operation Status Summary 128 6 Master Summary Status (MSS) / Request Service (RQS) 64 5 Standard Event Status Summary (ESB) 32 4 Message Available (MAV) 16 3 Questionable Status Summary 8 2 Error/Event Queue Summary 4 1 Available 2 0 Available 1
Response:	None
Example:	`*SRE 64`

1.10 *SRE? Service Request Enable Query

Syntax:	*SRE?
Description:	The Service Request Enable query *SRE? returns the contents of the Service Request Enable register (see *SRE for information on this register).
Parameters:	None
Response:	The bit value for the register (a **short** value). 7 Operation Status Summary 128 6 Master Summary Status (MSS) / Request Service (RQS) 64 5 Standard Event Status Summary (ESB) 32 4 Message Available (MAV) 16 3 Questionable Status Summary 8 2 Error/Event Queue Summary 4 1 Available 2 0 Available 1
Example:	`*SRE? –> 21<END>`

1.11 *STB? Read Status Byte Query

Syntax:	*STB?
Description:	The Status Byte query *STB? returns the contents of the Status Byte register. The Master Summary Status (MSS) bit is true when any enabled bit of the STB register is set (excluding Bit 6). The Status Byte register including, the master summary bit, MSS, is not directly altered because of an *STB? query.
Parameters:	None
Response:	The bit value for the register (a **short** value): 7 Operation Status Summary 128 6 Master Summary Status (MSS) / Request Service (RQS) 64 5 Standard Event Status Summary (ESB) 32 4 Message Available (MAV) 16 3 Questionable Status Summary 8 2 Error/Event Queue Summary 4 1 Available 2 0 Available 1
Example:	`*STB? –> 78<END>`

1.12 *TST? Self-Test Query

Syntax:	*TST?
Description:	The self-test query *TST? makes the currently selected logical instrument to perform a self-test and place the results of the test in the output queue. No further commands are allowed while the test is running. After the self test the instrument is returned to the setting that was active at the time the self-test query was processed.
Parameters:	None
Response:	The sum of the results for the individual tests (a **32-bit signed integer** value)
Example:	`*TST? –> 0<END>`

1.13 *WAI Wait-to-Continue Command

Syntax:	*WAI
Description:	The Wait command *WAI prevents the instrument from executing any further commands until the current command has finished executing. All pending operations are completed during the wait period.
Parameters:	None
Response:	None
Example:	`*WAI`

2 System Subsystem Commands

2.1 SYSTem:ERRor?

Syntax:	SYSTem:ERRor?
Description:	Returns the contents of the SCPI error queue. Removes the returned entry from the queue.
Parameters:	None
Response:	The number of the latest error, and its meaning.
Example:	`syst:err? -> -113,"Undefined header"<END>`

2.2 SYSTem:VERSion?

Syntax:	SYSTem:VERSion?
Description:	Returns the SCPI revision to which the system complies.
Parameters:	None
Response:	The revision year and number string.
Example:	`SYST:VERS? -> 1995.0<END>`

3 Status Subsystem Commands

3.1 STATus:OPERation[:EVENt]?

Syntax:	STATus:OPERation[:EVENt]?
Description:	Queries the operation event register.
Parameters:	None
Response:	The bit value for the operation event register as a **short** value (0 .. +32767)
Example:	`STAT:OPER? –> 0<END>`

3.2 PERation:CONDition?

Syntax:	STATus:OPERation:CONDition?
Description:	Queries the operation condition register.
Parameters:	None
Response:	The bit value for the operation condition register as a **short** value (0 .. +32767)
Example:	`STAT:OPER:COND? –> 16<END>`

3.3 STATus:OPERation:BIT#:CONDition?

Syntax:	STATus:OPERation:BIT<n>:CONDition?
Description:	This command accesses the user-definable bits in the OPERation register set. The value of <n> is restricted from 8 to 12 and represents bits 8 through 12 in the :STATus:OPERation status register.
Parameters:	None
Response:	The bit value for the operation condition register as a short value (0 .. 1)
Example:	`STAT:OPER:BIT8:COND? –> 1<END>`

3.4 STATus:OPERation:BIT#:ENABle

Syntax:	STATus:OPERation:BIT<n>:ENABle<wsp><value>
Description:	Sets the operation enable mask for the event register specified bit. The value of <n> is restricted from 8 to 12 and represents bits 8 through 12
Parameters:	The bit value for the operation enable mask as a **short** value (0 .. 1)
Response:	None
Example:	`STAT:OPER:BIT11:ENAB 1`

3.5 STATus:OPERation:BIT#:ENABle?

Syntax:	STATus:OPERation:BIT<n>:ENABle?
Description:	Returns the operation enable mask for the event register specified bit. The value of <n> is restricted from 8 to 12 and represents bits 8 through 12
Parameters:	None
Response:	The bit value for the operation enable mask as a **short** value (0 .. 1)
Example:	`STAT:OPER:BIT9:ENAB? –> 0<END>`

3.6 STATus:OPERation:BIT#[:EVENt]?

Syntax:	STATus:OPERation:BIT<n>[:EVENt]?
Description:	Queries the operation event register specified bit. The value of <n> is restricted from 8 to 12 and represents bits 8 through 12
Parameters:	None
Response:	The bit value for the operation event register as a **short** value (0 .. 1)
Example:	`STAT:OPER:BIT10:EVEN? –> 0<END>`

3.7 STATus:OPERation:ENABle

Syntax:	STATus:OPERation:ENABle<wsp><value>
Description:	Sets the operation enable mask for the event register.
Parameters:	The bit value for the operation enable mask as a **short** value (0 .. +32767)
Response:	None
Example:	`STAT:OPER:ENAB 128`

3.8 STATus:OPERation:ENABle?

Syntax:	STATus:OPERation:ENABle?
Description:	Returns the operation enable mask for the event register.
Parameters:	None
Response:	The bit value for the operation enable mask as a **short** value (0 .. +32767)
Example:	`STAT:OPER:ENAB? –> 128<END>`

3.9 STATus:OPERation:INSTrument:CONDition?

Syntax:	STATus:OPERation:INSTrument:CONDition?
Description:	Queries the instrument operation condition register.
Parameters:	None
Response:	The bit value for the operation condition register as a **short** value (0 .. +32767)
Example:	`STAT:OPER:INST:COND? –> 16<END>`

3.10 STATus:OPERation:INSTrument:ENABle

Syntax:	STATus:OPERation:INSTrument:ENABle<wsp><value>
Description:	Sets the instrument operation enable mask for the event register.
Parameters:	The bit value for the operation enable mask as a **short** value (0 .. +32767)
Response:	None
Example:	`STAT:OPER:INST:ENAB 128`

3.11 STATus:OPERation:INSTrument:ENABle?

Syntax:	STATus:OPERation:INSTrument:ENABle?
Description:	Returns the instrument operation enable mask for the event register.
Parameters:	None
Response:	The bit value for the operation enable mask as a short value (0 .. +32767)
Example:	`STAT:OPER:INST:ENAB? –> 128<END>`

3.12 STATus:OPERation:INSTrument[:EVENt]?

Syntax:	STATus:OPERation:INSTrument[:EVENt]?
Description:	Queries the instrument operation event register.
Parameters:	None
Response:	The bit value for the operation event register as a **short** value (0 .. +32767)
Example:	`STAT:OPER:INST? –> 0<END>`

3.13 STATus:OPERation:INSTrument:ISUMmary#:CONDition?

Syntax:	STATus:OPERation:INSTrument:ISUMmary<n>:CONDition?
Description:	Queries the instrument operation condition register of the specified instrument. The value of <n> is restricted from 1 to 14 and represents the logical instruments id assigned to the SCPI controlled instrument by the INSTrument subsystem.
Parameters:	None
Response:	The bit value for the operation condition register as a **short** value (0 .. +32767)
Example:	`STAT:OPER:INST:ISUM4:COND? –> 16<END>`

3.14 STATus:OPERation:INSTrument:ISUMmary#:ENABle

Syntax:	STATus:OPERation:INSTrument:ISUMmary<n>:ENABle<wsp><value>
Description:	Sets the instrument operation enable mask for the event register of the specified instrument. The value of <n> is restricted from 1 to 14 and represents the logical instruments id assigned to the SCPI controlled instrument by the INSTrument subsystem.
Parameters:	The bit value for the operation enable mask as a **short** value (0 .. +32767)
Response:	None
Example:	`STAT:OPER:INST:ISUM1:ENAB 128`

3.15 STATus:OPERation:INSTrument:ISUMmary#:ENABle?

Syntax:	STATus:OPERation:INSTrument:ISUMmary<n>ENABle?
Description:	Returns the instrument operation enable mask for the event register of the specified instrument. The value of <n> is restricted from 1 to 14 and represents the logical instruments id assigned to the SCPI controlled instrument by the INSTrument subsystem.
Parameters:	None
Response:	The bit value for the operation enable mask as a **short** value (0 .. +32767)
Example:	`STAT:OPER:INST:ISUM4:ENAB? –> 128<END>`

3.16 STATus:OPERation:INSTrument:ISUMmary#[:EVENt]?

Syntax:	STATus:OPERation:INSTrument:ISUMmary<n>[:EVENt]?
Description:	Queries the instrument operation event register of the specified instrument. The value of <n> is restricted from 1 to 14 and represents the logical instruments id assigned to the SCPI controlled instrument by the INSTrument subsystem.
Parameters:	None
Response:	The bit value for the operation event register as a **short** value (0 .. +32767)
Example:	`STAT:OPER:INST:ISUM3? –> 0<END>`

3.17 STATus:QUEStionable[:EVENt]?

Syntax:	STATus:QUEStionable[:EVENt]?
Description:	Queries the questionable event register
Parameters:	None
Response:	The bit value for the questionable event register as a **short** value (0 .. +32767)
Example:	`STAT:QUES? –> 0<END>`

3.18 STATus:QUEStionable:CONDition?

Syntax:	STATus:QUEStionable:CONDition?
Description:	Queries the questionable condition register
Parameters:	None
Response:	The bit value for the questionable condition register as a **short** value (0 .. +32767)
Example:	`STAT:QUES:COND? –> 8<END>`

3.19 STATus:QUEStionable:BIT#:CONDition?

Syntax:	STATus:QUEStionable:BIT<n> :CONDition?
Description:	Queries the questionable condition register specified bit. The value of <n> is restricted from 9 to 12 and represents bits 9 through 12
Parameters:	None
Response:	The bit value for the questionable condition register as a **short** value (0 .. 1)
Example:	`STAT:QUES:BIT9:COND? –> 0<END>`

3.20 STATus:QUEStionable:BIT#:ENABle

Syntax:	STATus:QUEStionable:BIT<n>:ENABle<wsp><value>
Description:	Sets the questionable enable mask for the event register specified bit. The value of <n> is restricted from 9 to 12 and represents bits 9 through 12
Parameters:	The bit value for the questionable enable mask as a **short** value (0 .. 1)
Response:	None
Example:	`STAT:QUES:BIT11:ENAB 1`

3.21 STATus:QUEStionable:BIT#:ENABle?

Syntax:	STATus:QUEStionable:BIT<n>:ENABle?
Description:	Returns the questionable enable mask for the event register specified bit. The value of <n> is restricted from 9 to 12 and represents bits 9 through 12
Parameters:	None
Response:	The bit value for the questionable enable mask as a **short** value (0 .. 1)
Example:	`STAT:QUES:BIT10:ENAB? –> 1<END>`

3.22 STATus:QUEStionable:BIT#[:EVENt]?

Syntax:	STATus:QUEStionable:BIT<n>[:EVENt]?
Description:	Queries the questionable event register specified bit. The value of <n> is restricted from 9 to 12 and represents bits 9 through 12
Parameters:	None
Response:	The bit value for the questionable event register as a **short** value (0 .. 1)
Example:	`STAT:QUES:BIT9:EVEN? –> 0<END>`

3.23 STATus:QUEStionable:ENABle

Syntax:	STATus:QUEStionable:ENABle<wsp><value>
Description:	Sets the questionable enable mask for the event register
Parameters:	The bit value for the questionable enable mask as a **short** value (0 .. +32767)
Response:	None
Example:	`STAT:QUES:ENAB 128`

3.24 STATus:QUEStionable:ENABle?

Syntax:	STATus:QUEStionable:ENABle?
Description:	Returns the questionable enable mask for the event register.
Parameters:	None
Response:	The bit value for the questionable enable mask as a **short** value (0 .. +32767)
Example:	`STAT:QUES:ENAB? –> 128<END>`

3.25 STATus:QUEStionable:INSTrument:CONDition?

Syntax:	STATus:QUEStionable:INSTrument :CONDition?
Description:	Queries the questionable instrument condition register.
Parameters:	None
Response:	The bit value for the questionable condition register as a **short** value (0 .. +32767)
Example:	`STAT:QUES:INST:COND? –> 8<END>`

3.26 STATus:QUEStionable:INSTrument:ENABle

Syntax:	STATus:QUEStionable:INSTrument:ENABle<wsp><value>
Description:	Sets the questionable instrument enable mask for the event register
Parameters:	The bit value for the questionable enable mask as a **short** value (0 .. +32767)
Response:	None
Example:	`STAT:QUES:INST:ENAB 128`

3.27 STATus:QUEStionable:INSTrument:ENABle?

Syntax:	STATus:QUEStionable:INSTrument:ENABle?
Description:	Returns the questionable instrument enable mask for the event register.
Parameters:	None
Response:	The bit value for the questionable enable mask as a **short** value (0 .. +32767)
Example:	`STAT:QUES:INST:ENAB? –> 128<END>`

3.28 STATus:QUEStionable:INSTrument[:EVENt]?

Syntax:	STATus:QUEStionable:INSTrument[:EVENt]?
Description:	Queries the questionable instrument event register
Parameters:	None
Response:	The bit value for the questionable event register as a **short** value (0 .. +32767)
Example:	`STAT:QUES:INST:EVEN? –> 0<END>`

3.29 STATus:QUEStionable:INSTrument:ISUMmary#:CONDition?

Syntax:	STATus:QUEStionable:INSTrument:ISUMmary<n> :CONDition?
Description:	Queries the specified logical instrument questionable instrument condition register. The value of <n> is restricted from 1 to 14 and represents the logical instruments id assigned to the SCPI controlled instrument by the INSTrument subsystem.
Parameters:	None
Response:	The bit value for the questionable condition register as a **short** value (0 .. +32767)
Example:	`STAT:QUES:INST:ISUM2:COND? –> 0<END>`

3.30 STATus:QUEStionable:INSTrument:ISUMmary#:ENABle

Syntax:	STATus:QUEStionable:INSTrument:ISUMmary<n>:ENABle<wsp><value>
Description:	Sets the questionable instrument enable mask for the event register of the specified logical instrument. The value of <n> is restricted from 1 to 14 and represents the logical instruments id assigned to the SCPI controlled instrument by the INSTrument subsystem.
Parameters:	The bit value for the questionable enable mask as a **short** value (0 .. +32767)
Response:	None
Example:	`STAT:QUES:INST:ISUM3:ENAB 128`

3.31 STATus:QUEStionable:INSTrument:ISUMmary#:ENABle?

Syntax:	STATus:QUEStionable:INSTrument:ISUMmary<n>:ENABle?
Description:	Returns the questionable instrument enable mask for the event register of the specified logical instrument. The value of <n> is restricted from 1 to 14 and represents the logical instruments id assigned to the SCPI controlled instrument by the INSTrument subsystem.
Parameters:	None
Response:	The bit value for the questionable enable mask as a **short** value (0 .. +32767)
Example:	`STAT:QUES:INST:ISUM3:ENAB? –> 136<END>`

3.32 STATus:QUEStionable:INSTrument:ISUMmary#[:EVENt]?

Syntax:	STATus:QUEStionable:INSTrument:ISUMmary[:EVENt]?
Description:	Queries the questionable instrument event register of the specified logical instrument. The value of <n> is restricted from 1 to 14 and represents the logical instruments id assigned to the SCPI controlled instrument by the INSTrument subsystem.
Parameters:	None
Response:	The bit value for the questionable event register as a **short** value (0 .. +32767)
Example:	`STAT:QUES:INST:ISUM4:EVEN? –> 0<END>`

3.33 STATus:PRESet

Syntax:	STATus:PRESet
Description:	Resets both the operation enable mask and questionable enable mask to 0
Parameters:	None
Response:	None
Example:	STAT:PRES

4 Instrument Subsystem Commands

4.1 INSTrument:CATalog?

Syntax:	INSTrument:CATalog?
Description:	Returns the list of SCPI controllable instruments on CMA5000 that are identified as SCPI controllable and have available plug-ins to run them
Parameters:	None
Response:	comma-separated list of string identifiers of all SCPI controllable logical instruments
Example:	`INST:CAT? –> STATUS1, OTDR_STD1, ATX_SONET1<END>`

4.2 INSTrument:CATalog:FULL?

Syntax:	INSTrument:CATalog:FULL?
Description:	Returns the list of SCPI controllable instruments on CMA5000 that are identified as SCPI controllable and have available plug-ins to run them
Parameters:	None
Response:	Returns a list of string - number pairs. The string contains the name identifier of the logical instrument. The immediately following NR1-formatted number is its associated logical instrument number. All response data elements are comma separated.
Example:	`INSTrument:CATalog:FULL? –> STATUS1,1, OTDR_STD1,2<END>`

4.3 INSTrument:NSELect

Syntax:	INSTrument:NSELect<wsp><num_id>
Description:	Sets the specified logical instrument to be currently selected instrument.
Parameters:	The numeric value identifier assigned by the Instrument subsystem for the instrument to be selected as a **short** value
Response:	None
Example:	`INSTrument:NSELect 2`

4.4 INSTrument:NSELect?

Syntax:	INSTrument:NSELect?
Description:	Returns the numeric value identifier of the currently selected logical instrument
Parameters:	None
Response:	Returns a numeric value of the currently selected logical instrument
Example:	`INSTrument:NSELect? –> 2<END>`

4.5 INSTrument[:SELect]

Syntax:	INSTrument:SELect<wsp><string_id>
Description:	Sets the specified logical instrument to be currently selected instrument
Parameters:	The string instrument identifier assigned by the Instrument subsystem for the instrument to be selected as a **string** value
Response:	None
Example:	`INSTrument:SELect OTDR_STD1`

4.6 INSTrument[:SELect]?

Syntax:	INSTrument:SELect?
Description:	Returns the string value identifier of the currently selected logical instrument
Parameters:	None
Response:	Returns the string value identifier of the currently selected logical instrument
Example:	`INSTrument:SELect? –> OTDR_STD1<END>`

4.7 INSTrument:STATe

Syntax:	INSTrument:STATe<wsp><boolean>
Description:	Turns the currently selected logical instrument ON or OFF
Parameters:	The Boolean data (ON or 1) to turn instrument on and (OFF or 0) to turn instrument off
Response:	None
Example:	`INSTrument: STATe ON`

4.8 INSTrument:STATe?

Syntax:	INSTrument:STATe?
Description:	Returns the state of the currently selected logical instrument
Parameters:	None
Response:	Returns a boolean value of the state of the currently selected logical instrument (1 if the instrument is active, 0 if the instrument is inactive)
Example:	`INSTrument: STATe? –> 1<END>`

5 Configuration

5.1 General Requirements

5.1.1 Client and Server

CMA5000 represents server which executes SCPI service. It allows remote clients connecting to it one at the time.

Client can be implemented in many different ways. It can be properly configured Windows Hyper Terminal, small user application or large control system that manages many units simultaneously.

CMA5000 SCPI server uses an UNIX socket configured to listen at port 2288. It receives processes and executes SCPI commands. While CMA5000 responds to the SCPI client requests, its graphical user interface (GUI) is simultaneously available as well.

5.1.2 Connection Port

For outside connections, CMA5000 SCPI server is listening at port 2288.

5.1.3 Startup

Before powering CMA5000 on for remote connection, it needs to be connected to the local area network (LAN). This is important, as during the boot process, the unit establishes it's network connection, making it's services available to the network. Connecting CMA5000 to the LAN after the boot will not allow outside connections.

Here is the list of initial commands required for operation of any instrument via SCPI interface:

INSTrument:CATalog?

Identifies the list of available remote instruments. The response may look like: "STATUS1,OSA1". STATUS1 is the CMA5000's default status instrument. OSA1 is plugged in OSA instrument. It could show any other instrument which represents the module inserted into CMA5000.

INSTrument:SELect OSA1

Makes OSA instrument current.

INSTrument:STATe ON

Instantiates the current instrument by setting its state to "ON". At this point OSA instrument should be ready to receive commands.

5.2 Connecting from Hyper Terminal

Windows Hyper Terminal may be used to quickly verify the SCPI operation. It can not handle binary responses, but works good with the text and most commands have textual response.

Figure 5.2.1 illustrates the connection screen of the Hyper Terminal. The Host Address is the IP Address of the CMA5000 unit connected to the LAN. Figures 5.2.2 and 5.2.3 show other important settings. Figure 5.2.4 shows examples of commands executed through Hyper Terminal.

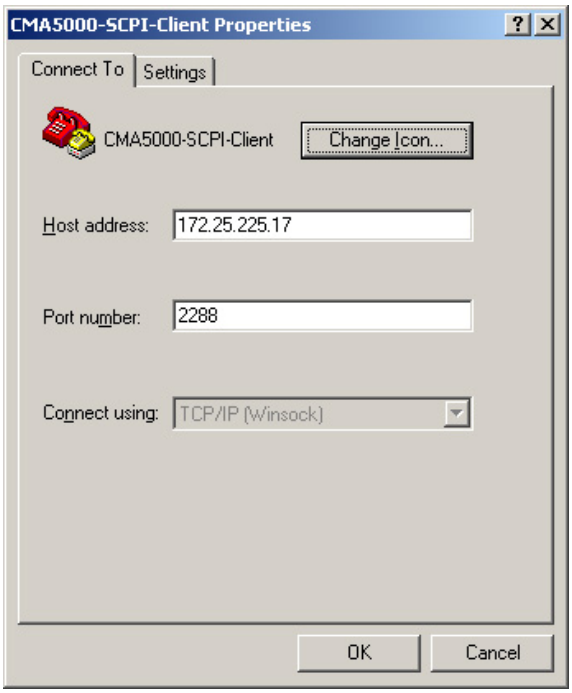

Figure 5.2.1 Connection screen

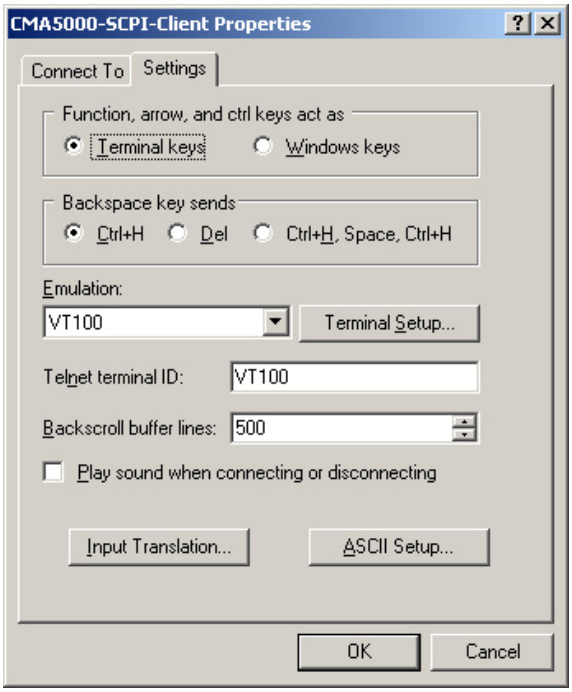

Figure 5.2.2 Settings

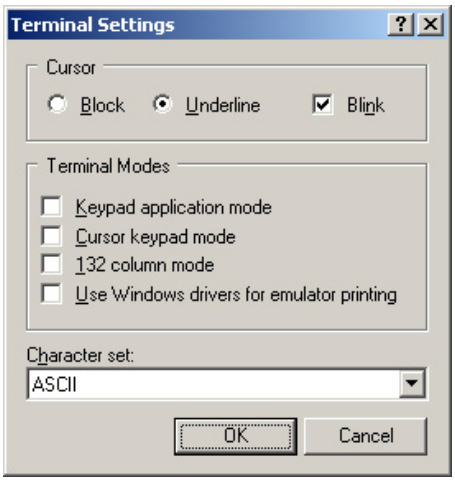

Figure 5.2.3 ASCII Setup

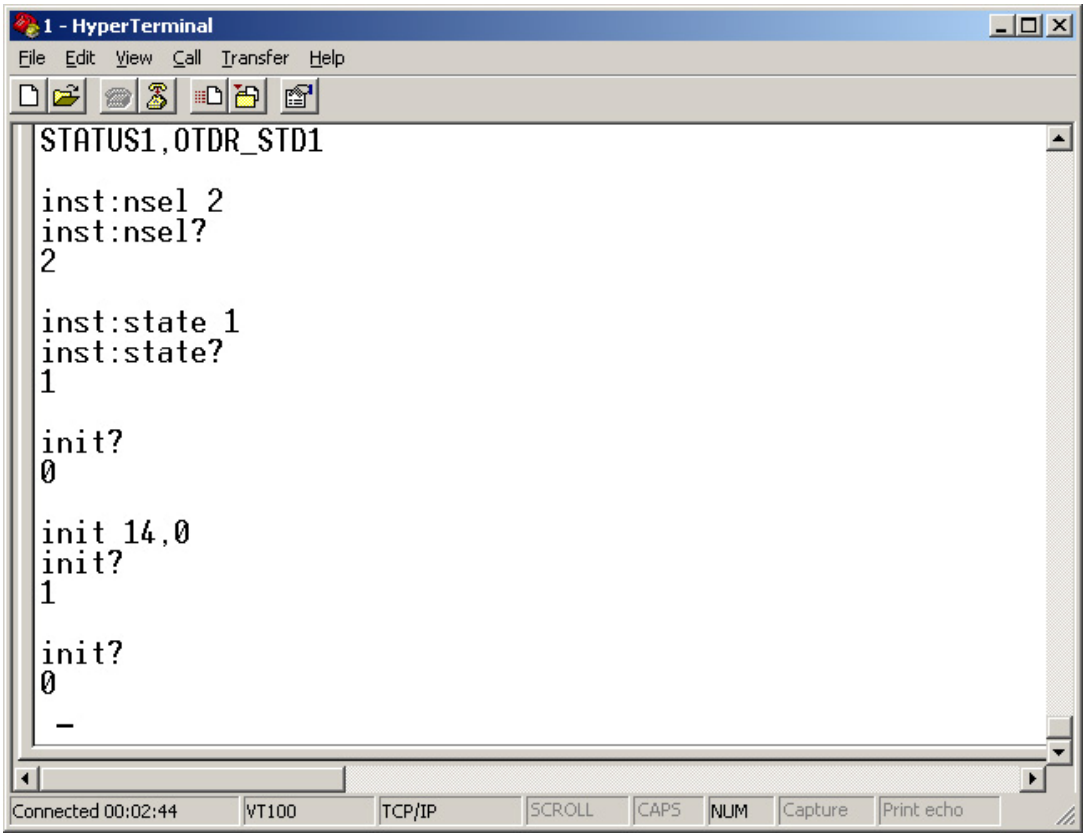

Figure 5.2.4 Hyper Terminal window

5.3 Writing Custom C Client

Below is an example of a simple C programming language client that demonstrates that can be used as a starting point of program that is customized for a particular application. Sample is developed using Microsoft Visual C++ version 6.0. Sample can controls OTDR module hosted by

the CMA5000 controller. The IP address of the CMA5000 unit, has to be passed as a command line argument

```
#include "stdafx.h"
#include "winsock2.h"
#include <stdio.h>
#include <stdlib.h>

#define PORT 2288

void Send(SOCKET s, char* pszData)
{
  WORD wTx;

  WORD wRemaining = strlen(pszData);
  WORD wIndex = 0;

  while (wRemaining != 0)
  {
       wTx = send(s, &pszData[wIndex], wRemaining, 0);
       if (wTx == -1)
       {
              perror("send");
              return;
       }

       wRemaining -= wTx;
       wIndex += wTx;
  }

  // Flush command ... send new line
  wRemaining = 1;
  while (wRemaining != 0)
  {
       wTx = send(s, "\n", wRemaining, 0);
       if (wTx == -1)
       {
              perror("send");
              return;
       }

       wRemaining -= wTx;
  }
}
```

```
void Receive(SOCKET s, char* pszData, WORD wRemaining)
{
  WORD wIndex = 0;

  char* pEOT = NULL;

  while (pEOT == NULL)
  {
       WORD wRx = recv(s, &pszData[wIndex], wRemaining, 0);
       if (wRx == -1)
       {
              perror("Receive");
              return;
       }

       wRemaining -= wRx;
       wIndex += wRx;

       pEOT = strstr(pszData, "\n");
  }

  // remove terminator
  if (pEOT != NULL)
       *pEOT = 0;
}

void SendCmd(SOCKET s, char* pszCmd, char* pszReply, WORD wRemaining)
{
  Send(s, pszCmd);
  Receive(s, pszReply, wRemaining);
  printf("Command [%s] received [%s]\n", pszCmd, pszReply);
}

void main(int argc, const char* argv[])
{
  char hostIp[256];
  if (argc == 2)
  {
       sprintf(hostIp, "%s", argv[1]);
       printf("Connecting to: %s\n", hostIp);
  }
  else
  {
```

```
     printf("Wrong number of arguments! Please pass IP address.\n");
     return;
}

char szReply[512];
char szCmd[512];

//----------------------
// Initialize Winsock
WSADATA wsaData;
int iResult = WSAStartup(MAKEWORD(2,2), &wsaData);
if (iResult != NO_ERROR)
     printf("Error at WSAStartup()\n");

//----------------------
// Create a SOCKET for connecting to server
SOCKET ConnectSocket;
ConnectSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (ConnectSocket == INVALID_SOCKET)
{
     printf("Error at socket(): %ld\n", WSAGetLastError());
     WSACleanup();
     return;
}

//----------------------
// The sockaddr_in structure specifies the address family,
// IP address, and port of the server to be connected to.
sockaddr_in clientService;
clientService.sin_family = AF_INET;
clientService.sin_addr.s_addr = inet_addr( hostIp );
clientService.sin_port = htons( PORT );

//----------------------
// Connect to server.
if    (    connect(    ConnectSocket,    (SOCKADDR*)    &clientService,
sizeof(clientService) ) == SOCKET_ERROR)
{
     printf( "Failed to connect!\n" );
     WSACleanup();
     return;
}

printf("Connected to server.\n");
```

```
// Retrieve Version from CMA
SendCmd(ConnectSocket, "SYST:ERR?", szReply, sizeof(szReply));
SendCmd(ConnectSocket, "*IDN?", szReply, sizeof(szReply));
SendCmd(ConnectSocket, "SYST:VERS?", szReply, sizeof(szReply));
SendCmd(ConnectSocket, "INST:CAT:FULL?", szReply, sizeof(szReply));

// Select OTDR
sprintf(szCmd, "inst:sel  OTDR_STD1");
Send(ConnectSocket, szCmd);
sprintf(szCmd, "inst:sel?");
SendCmd(ConnectSocket, szCmd, szReply, sizeof(szReply));

// Start OTDR Standard
sprintf(szCmd, "%s 1;%s?", "inst:stat", "inst:stat");
SendCmd(ConnectSocket, szCmd, szReply, sizeof(szReply));

// Start Scan
sprintf(szCmd, "init 14,0");
Send(ConnectSocket, szCmd);

// Check Status
sprintf(szCmd, "init?");
SendCmd(ConnectSocket, szCmd, szReply, sizeof(szReply));

while (atoi(szReply) == 1)
{
     // Check Status
     sprintf(szCmd, "init?");
     Sleep(1000);
     SendCmd(ConnectSocket, szCmd, szReply, sizeof(szReply));
}

// Check Status
sprintf(szCmd, "init?");
SendCmd(ConnectSocket, szCmd, szReply, sizeof(szReply));

// Stop OTDR Standard
sprintf(szCmd, "%s 0;%s?", "inst:stat", "inst:stat");
SendCmd(ConnectSocket, szCmd, szReply, sizeof(szReply));

WSACleanup();

printf("Press any key to close ...\n");
```

```
  while (!kbhit())
       Sleep(100);

  return;
}
```

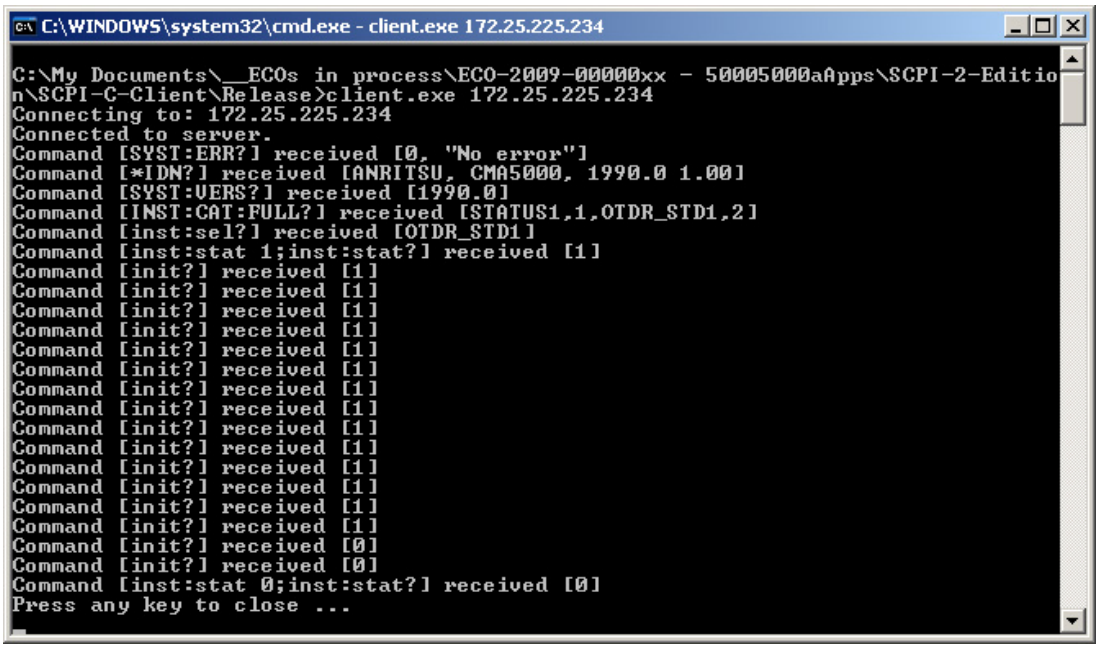

Figure 5.3.1 Program output

Figure 5.3.1 displays the screen shot of the program output. SCPI commands can be subdivided into two different categories: query commands which return requested information and set commands which do not return any information. Function SendCmd is suitable for query commands as it returns and prints the result of the executed query command. Function Send executes commands of the second type and does not return anything.

Worth noting, that every query command is terminated by the sender with new line character. When processing such a command, read operation must execute until the new line character is received. It is practically important to observe such behavior when working with buffered input stream, as there is no guarantee that the entire response will come in one read operation. Another possibility is that the string returned from read operation, may contain some data past the new line character. This is an acceptable behavior and client must interpret the data past new line as the return from the next command.

5.4 Writing Custom Java Client

For added convenience an example Java program is included below. It implements simple SCPI client. Sample is developed using Java SDK version 1.4. It controls OTDR module hosted by the CMA5000 controller. The IP address of the CMA5000 unit has to be passed as a command line argument. Figure 5.4.1 displays the screen shot of the program output.

```
package scpi;

import java.io.*;
import java.net.*;

public class Client
{
  static final int PORT = 2288;

  private Socket socket_ = null;
  private BufferedReader reader_ = null;
  private PrintWriter writer_ = null;

  public Client(String hostIp)
  {
    try
    {
      socket_ = new Socket(hostIp, PORT);
      reader_     =     new     BufferedReader(new     InputStreamReader(new
   DataInputStream(socket_.getInputStream())));
      writer_ = new PrintWriter(socket_.getOutputStream(), true /* autoFlush
   */);
    }
    catch (Exception e)
    {
      System.out.println("Failed to connect!");
    }

    String szReply;

    System.out.println("Connected to server.");

    // Retrieve Version from CMA
    szReply = Send(writer_, reader_, "SYST:ERR?", true);

    szReply = Send(writer_, reader_, "*IDN?", true);

    szReply = Send(writer_, reader_, "SYST:VERS?", true);

    szReply = Send(writer_, reader_, "INST:CAT:FULL?", true);

    // Select OTDR
    Send(writer_, reader_, "inst:sel  OTDR_STD1", false);
```

```
szReply = Send(writer_, reader_, "inst:sel?", true);

// Start OTDR Standard
szReply = Send(writer_, reader_, "inst:stat 1;inst:stat?", true);

// Start Scan
Send(writer_, reader_, "init 14,0", false);

// Check Status
szReply = Send(writer_, reader_, "init?", true);

while (szReply.equals("1"))
{
  // Check Status
  try { Thread.sleep(1000); }
  catch (Exception e) { e.printStackTrace(); }
  szReply = Send(writer_, reader_, "init?", true);
}

// Check Status
szReply = Send(writer_, reader_, "init?", true);

// Stop OTDR Standard
szReply = Send(writer_, reader_, "inst:stat 0;inst:stat?", true);

try
{
  if (socket_ != null)
    socket_.close();
}
catch (Exception e)
{
  e.printStackTrace();
}

System.out.println("Press any key to close ...\n");

try
{
  while (System.in.available() <= 0)
  {
    Thread.sleep(100);
  }
```

```
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }

  String Send(PrintWriter writer, BufferedReader reader, String cmd, boolean
   readReply)
  {
    if (writer != null && reader != null)
    {
      try
      {
        writer.println(cmd);
        if (readReply)
        {
          String reply = reader.readLine();
          System.out.println("Command [" + cmd + "] received [" + reply + "]");
          return reply;
        }
        System.out.println("Command [" + cmd + "]");
      }
      catch (IOException ioe)
      {
        ioe.printStackTrace();
      }
    }
    return "";
  }

  public static void main(String[] args)
  {
    String hostIp;
    if (args.length == 1)
    {
      hostIp = args[0];
      System.out.println("Connecting to: " + hostIp);
      Client client = new Client(hostIp);
    }
    else
    {
      System.out.println("Wrong number of arguments! Please pass IP address.");
      return;
```

```
    }
  }
}
```

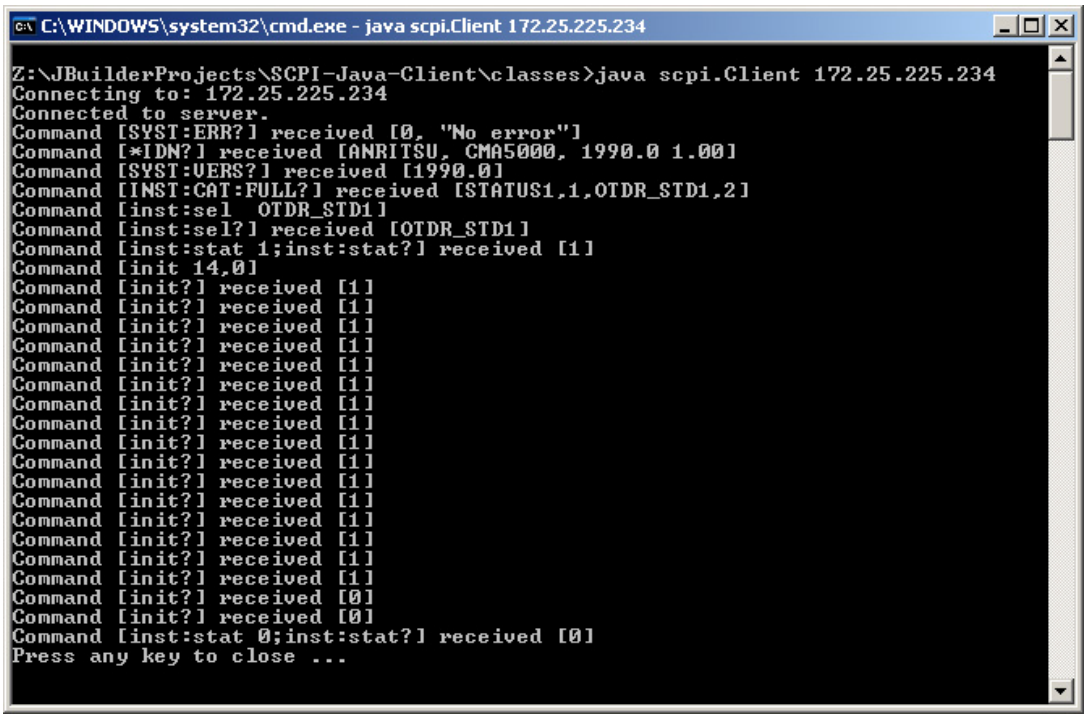

Figure 5.4.1 Program output

5.5 Generic Error Codes

This section details the list of Generic Error Codes that can be returned by SCPI Remote Control "SYSTem:ERRor?" command.

(0, "No error") No errors in the queue
(-350,"Queue overflow") Error queue is full (more then 12 errors generated and not queried)
(-100, "Command error") Command is not recognized by the SCPI Remote Control
(-102, "Syntax error") Command syntax error
(-104, "Data type error") Wrong parameter type passed to the command
(-108, "Parameter not allowed") Extra parameter passed to the command
(-109, "Missing parameter") Missing parameter
(-200, "Error INITIALIZING") Instrument subsystem could not initialize

Note, that particular Instrument Subsystem commands have list of error codes under each command description section in the provided document.