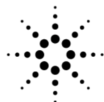
Programming Guide

Agilent Technologies
E8663B Analog Signal Generator

For the latest revision of this guide, go to **http://www.agilent.com/find/e8663b**.
Click Technical Support > Get a Manual.



Agilent Technologies

Manufacturing Part Number:
E8663-90005

Printed in USA

June 2006

© Copyright 2006 Agilent Technologies, Inc.

Notice

The material contained in this document is provided “as is”, and is subject to being changed, without notice, in future editions.

Further, to the maximum extent permitted by applicable law, Agilent disclaims all warranties, either express or implied with regard to this manual and to any of the Agilent products to which it pertains, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. Agilent shall not be liable for errors or for incidental or consequential damages in connection with the furnishing, use, or performance of this document or any of the Agilent products to which it pertains. Should Agilent have a written contract with the User and should any of the contract terms conflict with these terms, the contract terms shall control.

Contents

1 Getting Started with Remote Operation

Programming and Software/Hardware Layers. 2
Interfaces 3
IO Libraries and Programming Languages 4
- Agilent IO Libraries Suite 4
- Windows NT and Agilent IO Libraries M (and Earlier) 5
- Select IO Libraries for GPIB 6
- Selecting IO Libraries for LAN 7
- Programming Languages. 7

Using the Web Browser 8
- Enabling the Signal Generator Web Server 9

Preferences 11
- Configuring the Display for Remote Command Setups 12
- Setting the Help Mode 12

Error Messages 13
- Error Message File 13
- Error Message Types 14

2 Using IO Interfaces

Using GPIB 16
- Installing the GPIB Interface 16
- Set Up the GPIB Interface 18
- Verify GPIB Functionality. 18
- GPIB Interface Terms 19

GPIB Programming Interface Examples 20
- Before Using the GPIB Examples. 20
- Interface Check using HP Basic and GPIB. 20
- Interface Check Using NI-488.2 and C++. 20

Using LAN 22
- Setting Up the LAN Interface 23
- Verifying LAN Functionality 26
- Using VXI-11 30
- Using Sockets LAN 31
- Using Telnet LAN 32
- Using FTP 36

Using RS-232 38
- Selecting IO Libraries for RS-232 38
- Setting Up the RS-232 Interface 39
- Verifying RS-232 Functionality 41
- Character Format Parameters 42

Contents

If You Have Problems . . . 42

RS-232 Programming Interface Examples . . . 43
- Before Using the Examples . . . 43
- Interface Check Using HP BASIC . . . 43
- Interface Check Using VISA and C . . . 43
- Queries Using HP Basic and RS-232 . . . 44
- Queries for RS-232 Using VISA and C . . . 44

3 Programming Examples

Using the Programming Interface Examples . . . 46
- Programming Examples Development Environment . . . 46
- Running C++ Programs . . . 47
- Running C# Examples. . . . 48
- Running Basic Examples . . . 48
- Running Java Examples. . . . 49
- Running MATLAB Examples. . . . 49
- Running Perl Examples . . . 49

Using GPIB . . . 50
- Installing the GPIB Interface Card . . . 50

GPIB Programming Interface Examples . . . 52
- Before Using the GPIB Examples . . . 52
- GPIB Function Statements (Command Messages) . . . 52
- Interface Check using HP Basic and GPIB . . . 56
- Interface Check Using NI-488.2 and C++ . . . 57
- Interface Check for GPIB Using VISA and C . . . 58
- Local Lockout Using HP Basic and GPIB . . . 59
- Local Lockout Using NI-488.2 and C++. . . . 60
- Queries Using HP Basic and GPIB. . . . 62
- Queries Using NI-488.2 and Visual C++ . . . 63
- Queries for GPIB Using VISA and C . . . 65
- Generating a CW Signal Using VISA and C . . . 67
- Generating an Externally Applied AC-Coupled FM Signal Using VISA and C. . . . 69
- Generating an Internal FM Signal Using VISA and C . . . 71
- Generating a Step-Swept Signal Using VISA and C++ . . . 73
- Generating a Swept Signal Using VISA and Visual C++ . . . 74
- Saving and Recalling States Using VISA and C . . . 77
- Reading the Data Questionable Status Register Using VISA and C. . . . 79
- Reading the Service Request Interrupt (SRQ) Using VISA and C. . . . 83

LAN Programming Interface Examples. . . . 87
- VXI-11 Programming . . . 87
- VXI-11 Programming Using SICL and C++ . . . 88

VXI-11 Programming Using VISA and C++ . . . 89
Sockets LAN Programming and C . . . 91
Queries for Lan Using Sockets . . . 94
Sockets LAN Programming Using Java . . . 115
Sockets LAN Programming Using PERL . . . 116
RS-232 Programming Interface Examples . . . 118
Before Using the Examples . . . 118
Interface Check Using HP BASIC . . . 118
Interface Check Using VISA and C . . . 119
Queries Using HP Basic and RS-232 . . . 121
Queries for RS-232 Using VISA and C . . . 122

4 Programming the Status Register System

Overview . . . 126
Status Register Bit Values . . . 129
Example: Enable a Register . . . 129
Example: Query a Register . . . 129
Accessing Status Register Information . . . 130
Determining What to Monitor . . . 130
Deciding How to Monitor . . . 130
Status Register SCPI Commands . . . 132
Status Byte Group . . . 134
Status Byte Register . . . 135
Service Request Enable Register . . . 135
Status Groups . . . 136
Standard Event Status Group . . . 137
Standard Operation Status Group . . . 139
Data Questionable Status Group . . . 142
Data Questionable Power Status Group . . . 145
Data Questionable Frequency Status Group . . . 148
Data Questionable Modulation Status Group . . . 151
Data Questionable Calibration Status Group . . . 154

5 Creating and Downloading User-Data Files

Save and Recall Instrument State Files . . . 158
Save and Recall SCPI Commands . . . 158
Save and Recall Programming Example Using VISA and C# . . . 158
Download User Flatness Corrections Using C++ and VISA . . . 169

1 Getting Started with Remote Operation

- “Programming and Software/Hardware Layers” on page 2
- “Interfaces” on page 3
- “IO Libraries and Programming Languages” on page 4
- “Using the Web Browser” on page 8
- “Preferences” on page 11
- “Error Messages” on page 13

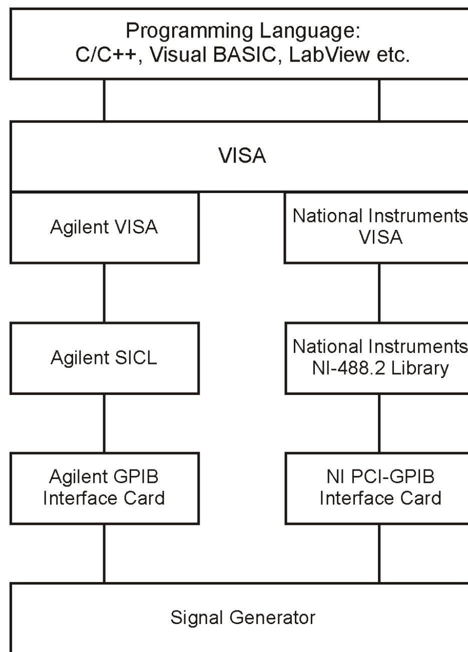
Programming and Software/Hardware Layers

Agilent E8663B signal generators support the following interfaces:

- GPIB
- LAN
- GPIB, LAN, and ANSI/EIA232 (RS-232) serial connection[1]

Use these interfaces, in combination with IO libraries and programming languages, to remotely control a signal generator. Figure 1-1 uses GPIB as an example of the relationships between the interface, IO libraries, programming language, and signal generator.

Figure 1-1 Software/Hardware Layers



1. The E8663B's **AUXILIARY INTERFACE** connector is compatible with ANSI/EIA232 (RS-232) serial connection but GPIB and LAN are recommended for making faster measurements and when downloading files. Refer to "Using RS-232" on page 38 and the *User's Guide*.

Interfaces

GPIB	GPIB is used extensively when a dedicated computer is available for remote control of each instrument or system. Data transfer is fast because GPIB handles information in 8-bit bytes with data transfer rates of up to 8 Mbytes/s. GPIB is physically restricted by the location and distance between the instrument/system and the computer; cables are limited to an average length of two meters per device with a total length of 20 meters. For more information on configuring the signal generator to communicate over the GPIB, refer to "Using GPIB" on page 16.
LAN	Data transfer using the LAN is fast as the LAN handles packets of data. The distance between a computer and the signal generator is limited to 100 meters. The E8663B is designed to connect with a 10Base-T LAN. For more information on LAN communication refer to *http://www.ieee.org*. The following protocols can be used to communicate with the signal generator over the LAN: • VXI-11 (Recommended) • Sockets LAN • TELNET • FTP For more information on configuring the signal generator to communicate over the LAN, refer to "Using LAN" on page 22
RS-232[a]	RS-232 is a common method used to communicate with a single instrument; its primary use is to control printers and external disk drives, and connect to a modem. Communication over RS-232 is much slower than with GPIB, or LAN because data is sent and received one bit at a time. It also requires that certain parameters, such as baud rate, be matched on both the computer and signal generator. For more information on configuring the signal generator to communicate over the GPIB, refer to "Using RS-232" on page 38.

a. The E8663B's **AUXILIARY INTERFACE** connector is compatible with ANSI/EIA232 (RS-232) serial connection but GPIB and LAN are recommended for making faster measurements and when downloading files. Refer to "Using RS-232" on page 38 and the *User's Guide*.

IO Libraries and Programming Languages

The IO libraries is a collection of functions used by a programming language to send instrument commands and receive instrument data. Before you can communicate and control the signal generator, you must have an IO library installed on your computer. The Agilent IO libraries are included on an Automation-Ready CD with your signal generator and Agilent GPIB interface board, or they can be downloaded from the Agilent website: *http://www.agilent.com*.

NOTE To learn about using IO libraries with Windows XP or newer operating systems, refer to the Agilent IO Libraries Suite's help located on the Automation-Ready CD that ships with your signal generator, Agilent GPIB interface board, or that can be downloaded from the Agilent website: *http://www.agilent.com*.

To better understand setting up Windows XP operating systems and newer, using PC LAN port settings, refer to Chapter 2.

Agilent IO Libraries Suite

The Agilent IO Libraries Suite replaces earlier versions of the Agilent IO Libraries and is supported on all platforms except Windows NT. If you are using the Windows NT platform, you must use Agilent IO Libraries version M or earlier.

Windows 98 and Windows ME are not supported in the Agilent IO Libraries Suite version 14.1 and higher.

NOTE The signal generator ships with an Automation-Ready CD that contains the Agilent IO Libraries Suite 14.0 for users who need support for Windows 98 and Windows ME.

Once the libraries are loaded, you can use the Agilent Connection Expert, Interactive IO, or VISA Assistant to configure and communicate with the signal generator over different IO interfaces. Follow instructions in the setup wizard to install the libraries.

NOTE Before setting the LAN interface the signal generator must be configured for VXI-11 SCPI. Refer to “Configuring the VXI-11 for LAN” on page 23.

Refer to the Agilent IO Libraries Suite Help documentation for details about this software.

Windows NT and Agilent IO Libraries M (and Earlier)

NOTE Windows NT is not supported on Agilent IO Libraries 14.0 and newer.

The following sections are specific to Agilent IO Libraries versions M and earlier and apply only to the Windows NT platform.

Using IO Config for Computer-to-Instrument Communication with VISA (Automatic or Manually)

After installing the Agilent IO Libraries version M or earlier, you can configure the interfaces available on your computer by using the IO Config program. This program can setup the interfaces that you want to use to control the signal generator. The following steps set up the interfaces.

1. Install GPIB interface boards before running IO Config.

NOTE You can also connect GPIB instruments using the Agilent 82357A USB/GPIB Interface Converter, which eliminates the need for a GPIB card. For more information, go to *http://www.agilent.com/find/gpib*.

---

2. Run the IO Config program. The program automatically identifies available interfaces.
3. Click on the interface type you want to configure, such as GPIB, in the Available Interface Types text box.
4. Click the **Configure** button. Set the Default Protocol to AUTO.
5. Click **OK** to use the default settings.
6. Click **OK** to exit the IO Config program.

VISA Assistant

VISA is an industry standard IO library API. It allows the user to send SCPI commands to instruments and to read instrument data in a variety of formats. You can use the VISA Assistant, available with the Agilent IO Libraries versions M and earlier, to send commands to the signal generator. If the interface you want to use does not appear in the VISA Assistant then you must manually configure the interface. See the Manual VISA Configuration section below. Refer to the VISA Assistant Help menu and the Agilent VISA User's Manual (available on Agilent's website) for more information.

VISA Configuration (Automatic)

1. Run the VISA Assistant program.
2. Click on the interface you want to use for sending commands to the signal generator.
3. Click the **Formatted I/O** tab.
4. Select **SCPI** in the **Instr. Lang.** section.

You can enter SCPI commands in the text box and send the command using the **viPrintf** button.

VISA Configuration (Manual)

Perform the following steps to use IO Config and VISA to manually configure an interface.

1. Run the **IO Config** Program.
2. Click on **GPIB** in the **Available Interface Types** text box.
3. Click the **Configure** button. Set the **Default Protocol** to **AUTO** and then Click **OK** to use the default settings.
4. Click on **GPIB0** in the **Configured Interfaces** text box.
5. Click **Edit…**
6. Click the **Edit VISA Config…** button.
7. Click the **Add device** button.
8. Enter the GPIB address of the signal generator.
9. Click the **OK** button in this form and all other forms to exit the IO Config program.

Select IO Libraries for GPIB

The IO libraries are included with the GPIB interface card, and can be downloaded from the National Instruments website or the Agilent website. See also, "IO Libraries and Programming Languages" on page 4 for information on IO libraries. The following is a discussion on these libraries.

VISA — VISA is an IO library used to develop IO applications and instrument drivers that comply with industry standards. It is recommended that the VISA library be used for programming the signal generator. The NI-VISA™ and Agilent VISA libraries are similar implementations of VISA and have the same commands, syntax, and functions. The differences are in the lower level IO libraries; NI-488.2 and SICL respectively. It is best to use the Agilent VISA library with the Agilent GPIB interface card or NI-VISA with the NI PCI-GPIB interface card.[1]

SICL — Agilent SICL can be used without the VISA overlay. The SICL functions can be called from a program. However, if this method is used, executable programs will not be portable to other hardware platforms. For example, a program using SICL functions will not run on a computer with NI libraries (PCI-GPIB interface card).

CAUTION Because of the potential for portability problems, running Agilent SICL without the VISA overlay is not recommended by Agilent Technologies.

NI-488.2 — NI-488.2 can be used without the VISA overlay. The NI-488.2 functions can be called from a program. However, if this method is used, executable programs will not be portable to other hardware platforms. For example, a program using NI-488.2 functions will not run on a computer with Agilent SICL (Agilent GPIB interface card).

1. NI-VISA is a registered trademark of National Instruments Corporation

Selecting IO Libraries for LAN

The TELNET and FTP protocols do not require IO libraries to be installed on your computer. However, to write programs to control your signal generator, an IO library must be installed on your computer and the computer configured for instrument control using the LAN interface.

The Agilent IO libraries Suite is available on the Automation-Ready CD, which was shipped with your signal generator. The libraries can also be downloaded from the Agilent website. The following is a discussion on these libraries.

Agilent VISA — VISA is an IO library used to develop IO applications and instrument drivers that comply with industry standards. Use the Agilent VISA library for programming the signal generator over the LAN interface.

SICL — Agilent SICL is a lower level library that is installed along with Agilent VISA.

CAUTION Because of the potential for portability problems, running Agilent SICL without the VISA overlay is not recommended by Agilent Technologies.

Programming Languages

Along with Standard Commands for Programming Instructions (SCPI) and IO library functions, you use a programming language to remotely control the signal generator. Common programming languages include:

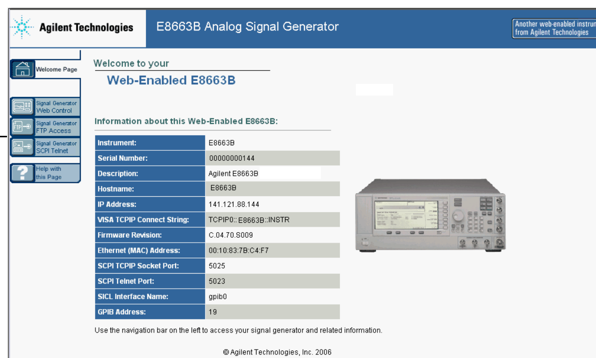
- C/C++
- C#
- MATLAB® (MATLAB is a registered trademark of The MathWorks.)
- HP Basic
- LabView
- Java™ (Java is a U.S. trademark of Sun Microsystems, Inc.)
- Visual Basic® (Visual Basic is a registered trademark of Microsoft Corporation.)
- PERL

Using the Web Browser

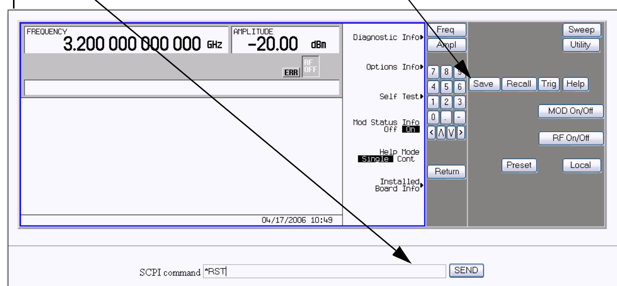
The signal generator can be used as a Web Server. The signal generator can function as a Web Server using a client/server model where the client is the web browser on your PC or workstation and the server is the signal generator. When you enable the Web Server, you can access a web page that resides on the signal generator.

The web-enabled signal generator web page, shown at right and page 10, provides general information on the signal generator, FTP access to files stored on the signal generator, and a means to control the instrument using either a remote front-panel interface or SCPI commands. The web page also has links to Agilent’s products, support, manuals, and website.

The Web Server service is compatible with the Microsoft© Internet Explorer (6.0 and newer) web browser and operating systems Windows 2000. Windows XP and newer. For more information on using the signal generator as a Web Server, refer to “Enabling the Signal Generator Web Server” on page 9.



To operate the signal generator, either click keys, or enter SCPI commands and click SEND.

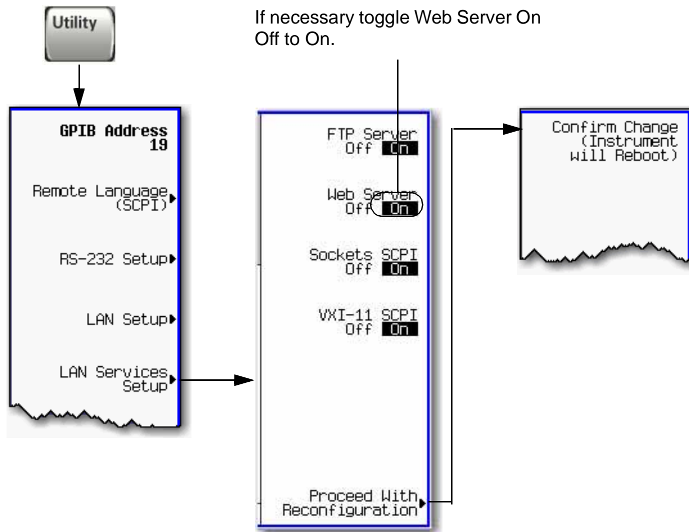


The results of a SCPI command display on a separate web page titled, “SCPI Command Processed.” You can continue using this web page to enter SCPI commands or you can return to the front panel web page. If the web page does not update, use the Web browser Refresh function.

Enabling the Signal Generator Web Server

1. If it is not already enabled, turn on the Web server: Refer to "E8663B Web Server On" on page 9.

E8663B Web Server On



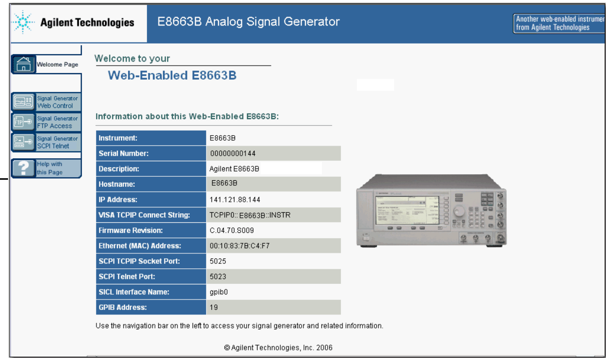
2. Launch the PC or workstation web browser.
3. In the web browser address field, enter the signal generator's IP address. For example, *http://101.101.101.101* (where *101.101.101.101* is the signal generator's IP address).

   The IP (internet protocol) address can change depending on the LAN configuration (see "Using LAN" on page 22).

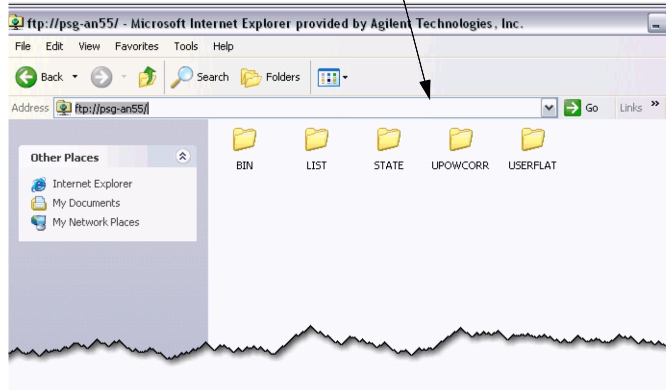
4. On the computer’s keyboard, press **Enter**. The web browser displays the signal generator’s homepage.

5. Click the Signal Generator Web Control menu button on the left of the page. The front panel web page displays.

   To control the signal generator, either click the front panel keys or enter SCPI commands.



The FTP access softkey opens to show the folders containing the signal generator’s memory catalog files.



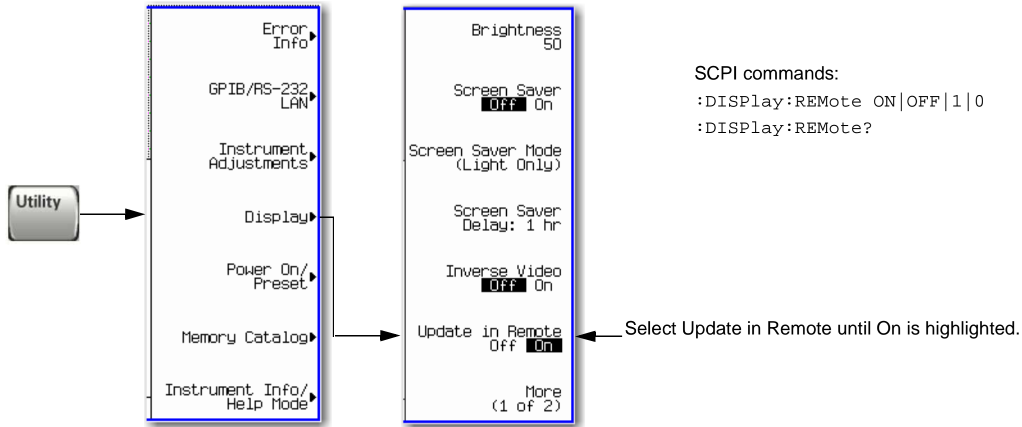
Preferences

The following commonly used manual command sections are included here:

“Configuring the Display for Remote Command Setups” on page 12

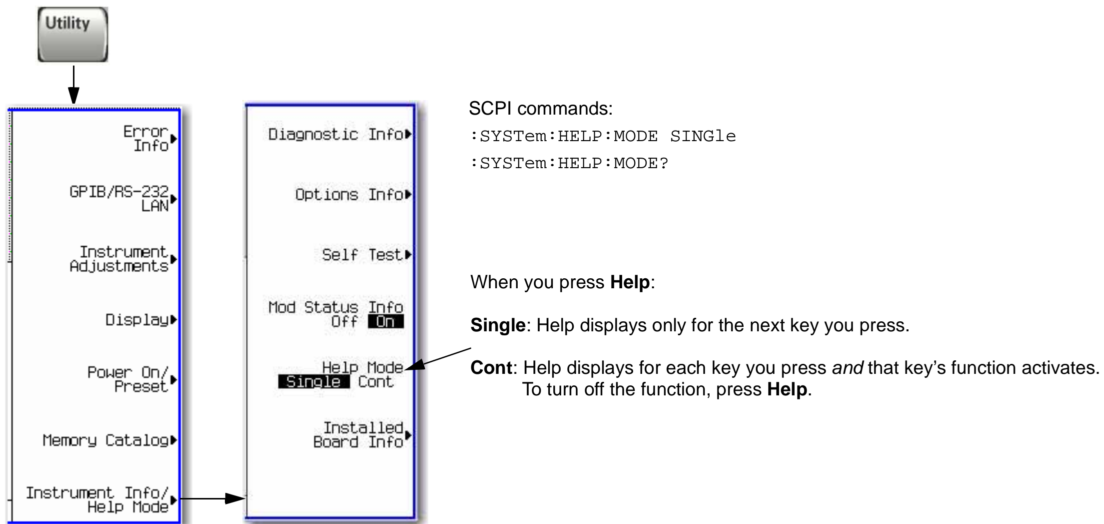
“Setting the Help Mode” on page 12

Configuring the Display for Remote Command Setups



For details on each key, use the *Key Reference*.

Setting the Help Mode



For details on each key, use the *Key Reference*.

Error Messages

If an error condition occurs in the signal generator, it is reported to both the SCPI (remote interface) error queue and the front panel display error queue. These two queues are viewed and managed separately; for information on the front panel display error queue, refer to the *User's Guide*.

NOTE For additional general information on troubleshooting problems with your connections, refer to the Help in the Agilent IO Libraries and documentation.

When accessing error messages using the SCPI (remote interface) error queue, the error numbers and the <error_description> portions of the error query response are displayed on the host terminal.

Characteristic	SCPI Remote Interface Error Queue
Capacity (#errors)	30
Overflow Handling	Linear, first-in/first-out. Replaces newest error with: `-350, Queue overflow`
Viewing Entries	Use SCPI query `SYSTem:ERRor[:NEXT]?`
Clearing the Queue	Power up Send a `*CLS` command Read last item in the queue
Unresolved Errors[a]	Re-reported after queue is cleared.
No Errors	When the queue is empty (every error in the queue has been read, or the queue is cleared), the following message appears in the queue: `+0, "No error"`

a. Errors that must be resolved. For example, unlock.

Error Message File

A complete list of error messages is provided in the file *errormessages.pdf*, on the CD-ROM supplied with your instrument. In the error message list, an explanation is generally included with each error to further clarify its meaning. The error messages are listed numerically. In cases where there are multiple listings for the same error number, the messages are in alphabetical order.

Error Message Types

Events do not generate more than one type of error. For example, an event that generates a query error will not generate a device-specific, execution, or command error.

Query Errors (–499 to –400) indicate that the instrument's output queue control has detected a problem with the message exchange protocol described in IEEE 488.2, Chapter 6. Errors in this class set the query error bit (bit 2) in the event status register (IEEE 488.2, section 11.5.1). These errors correspond to message exchange protocol errors described in IEEE 488.2, 6.5. In this case:

- Either an attempt is being made to read data from the output queue when no output is either present or pending, or
- data in the output queue has been lost.

Device Specific Errors (–399 to –300, 201 to 703, and 800 to 810) indicate that a device operation did not properly complete, possibly due to an abnormal hardware or firmware condition. These codes are also used for self-test response errors. Errors in this class set the device-specific error bit (bit 3) in the event status register (IEEE 488.2, section 11.5.1).

The <error_message> string for a *positive* error is not defined by SCPI. A positive error indicates that the instrument detected an error within the GPIB system, within the instrument's firmware or hardware, during the transfer of block data, or during calibration.

Execution Errors (–299 to –200) indicate that an error has been detected by the instrument's execution control block. Errors in this class set the execution error bit (bit 4) in the event status register (IEEE 488.2, section 11.5.1). In this case:

- Either a <PROGRAM DATA> element following a header was evaluated by the device as outside of its legal input range or is otherwise inconsistent with the device's capabilities, or
- a valid program message could not be properly executed due to some device condition.

Execution errors are reported *after* rounding and expression evaluation operations are completed. Rounding a numeric data element, for example, is not reported as an execution error.

Command Errors (–199 to –100) indicate that the instrument's parser detected an IEEE 488.2 syntax error. Errors in this class set the command error bit (bit 5) in the event status register (IEEE 488.2, section 11.5.1). In this case:

- Either an IEEE 488.2 syntax error has been detected by the parser (a control-to-device message was received that is in violation of the IEEE 488.2 standard. Possible violations include a data element that violates device listening formats or whose type is unacceptable to the device.), or
- an unrecognized header was received. These include incorrect device-specific headers and incorrect or unimplemented IEEE 488.2 common commands.

2 Using IO Interfaces

Using the programming examples with GPIB, LAN, and RS232 interfaces:

- “Using GPIB” on page 16
- “Using LAN” on page 22
- “Using RS-232” on page 38

Using GPIB

GPIB enables instruments to be connected together and controlled by a computer. GPIB and its associated interface operations are defined in the ANSI/IEEE Standard 488.1-1987 and ANSI/IEEE Standard 488.2-1992. See the IEEE website, *http://www.ieee.org*, for details on these standards.

The following sections contain information for installing a GPIB interface card or NI-GPIB interface card for your PC or UNIX-based system.

- "Installing the GPIB Interface" on page 16
- "Set Up the GPIB Interface" on page 18
- "Verify GPIB Functionality" on page 18

Installing the GPIB Interface

NOTE You can also connect GPIB instruments to a PC LAN port using the Agilent 82357A USB/GPIB Interface Converter, which eliminates the need for a GPIB card. For more information, refer to table on page 16 or go to *http://www.agilent.com/find/gpib*.

A GPIB interface card must be installed in the computer. Two common GPIB interface cards are the National Instruments (NI) PCI–GPIB card and the Agilent GPIB interface card. Follow the interface card instructions for installing and configuring the card. The following table provide lists on some of the available interface cards. Also, see the Agilent website, http://*www.agilent.com* for details on GPIB interface cards.

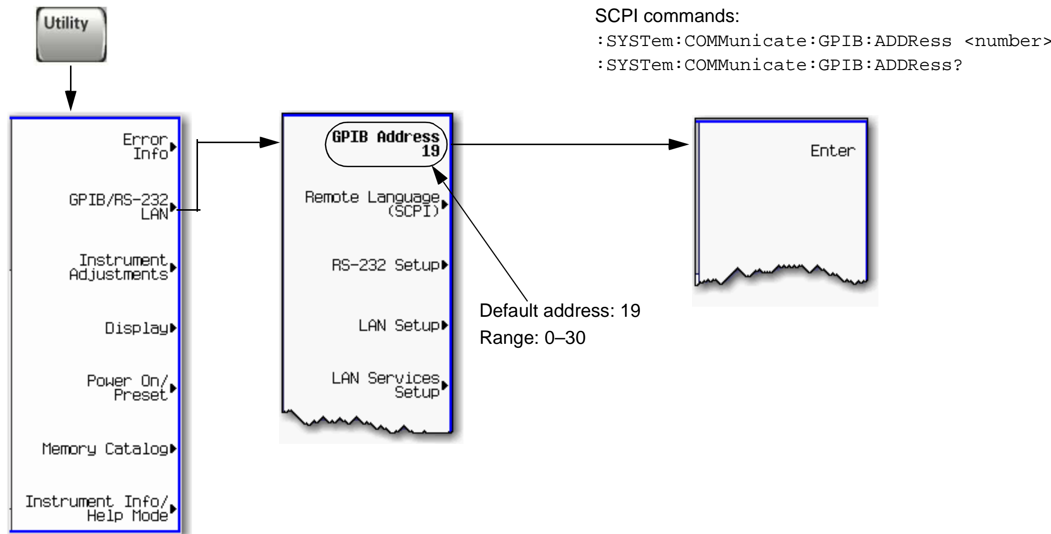
Interface Type	Operating System	IO Library	Languages	Backplane/ BUS	Max IO (kB/sec)	Buffering
Agilent USB/GPIB Interface Converter for PC-Based Systems						
Agilent 82357A Converter	Windows[a] 98 SE/ME/ 2000®/XP	VISA / SICL	C/C++, Visual Basic, Agilent VEE, HP Basic for Windows, NI Labview	ISA/EISA, 16 bit	850	Built-in
Agilent GPIB Interface Card for PC-Based Systems						
Agilent 82341C for ISA bus computers	Windows[b] 95/98/NT /2000®	VISA / SICL	C/C++, Visual Basic, Agilent VEE, HP Basic for Windows	ISA/EISA, 16 bit	750	Built-in
Agilent 82341D Plug&Play for PC	Windows 95	VISA / SICL	C/C++, Visual Basic, Agilent VEE, HP Basic for Windows	ISA/EISA, 16 bit	750	Built-in
Agilent 82350A for PCI bus computers	Windows 95/98/NT /2000	VISA / SICL	C/C++, Visual Basic, Agilent VEE, HP Basic for Windows	PCI 32 bit	750	Built-in

Interface Type	Operating System	IO Library	Languages	Backplane/ BUS	Max IO (kB/sec)	Buffering
Agilent USB/GPIB Interface Converter for PC-Based Systems						
Agilent 82350B for PCI bus computers	Windows 98(SE)/ME/2000 /XP	VISA / SICL	C/C++, Visual Basic, Agilent VEE, HP Basic for Windows	PCI 32 bit	> 900	Built-in
NI-GPIB Interface Card for PC-Based Systems						
National Instruments PCI-GPIB	Windows 95/98/2000/ ME/NT	VISA NI-488.2™[c]	C/C++, Visual BASIC, LabView	PCI 32 bit	1.5 MBps	Built-in
National Instruments PCI-GPIB+	Windows NT	VISA NI-488.2	C/C++, Visual BASIC, LabView	PCI 32 bit	1.5 MBps	Built-in
Agilent-GPIB Interface Card for HP-UX Workstations						
Agilent E2071C	HP-UX 9.x, HP-UX 10.01	VISA/SICL	ANSI C, Agilent VEE, Agilent BASIC, HP-UX	EISA	750	Built-in
Agilent E2071D	HP-UX 10.20	VISA/SICL	ANSI C, Agilent VEE, Agilent BASIC, HP-UX	EISA	750	Built-in
Agilent E2078A	HP-UX 10.20	VISA/SICL	ANSI C, Agilent VEE, Agilent BASIC, HP-UX	PCI	750	Built-in

a. Windows 95, 98, NT, 2000 and XP are registered trademarks of Microsoft Corporation
b. Windows 95, 98, NT, 2000 and XP are registered trademarks of Microsoft Corporation
c. NI-488.2 is a trademark of National Instruments Corporation

Set Up the GPIB Interface

Figure 2-1 Setting the GPIB Address on the E8663B



For details on each key, use the *Key Reference*. For information describing the key help, refer to the *User's Guide*.

Connect a GPIB interface cable between the signal generator and the computer. (The following table lists cable part numbers.)

Model	10833A	10833B	10833C	10833D	10833F	10833G
Length	1 meter	2 meters	4 meters	.5 meter	6 meters	8 meters

Verify GPIB Functionality

To verify GPIB functionality, use the VISA Assistant, available with the Agilent IO Library or the Getting Started Wizard available with the National Instrument IO Library. These utility programs enable you to communicate with the signal generator and verify its operation over GPIB. For information and instructions on running these programs, refer to the Help menu available in each utility.

If You Have Problems

1. Verify that the signal generator's address matches the address declared in the program (example programs in Chapter 2 use address 19).
2. Remove all other instruments connected via GPIB and rerun the program.
3. Verify that the GPIB card's name or id number matches the GPIB name or id number configured for your PC.

GPIB Interface Terms

An instrument that is part of a GPIB network is categorized as a listener, talker, or controller, depending on its current function in the network.

listener	A listener is a device capable of receiving data or commands from other instruments. Several instruments in the GPIB network can be listeners simultaneously.
talker	A talker is a device capable of transmitting data. To avoid confusion, a GPIB system allows only one device at a time to be an active talker.
controller	A controller, typically a computer, can specify the talker and listeners (including itself) for an information transfer. Only one device at a time can be an active controller.

GPIB Programming Interface Examples

- "Interface Check using HP Basic and GPIB" on page 20
- "Interface Check Using NI-488.2 and C++" on page 20

Before Using the GPIB Examples

CAUTION Because of the potential for portability problems, running Agilent SICL without the VISA overlay is not recommended by Agilent Technologies.

If the Agilent GPIB interface card is used, the Agilent VISA library should be installed along with Agilent SICL. If the National Instruments PCI-GPIB interface card is used, the NI-VISA library along with the NI-488.2 library should be installed. Refer to "Select IO Libraries for GPIB" on page 6 and the documentation for your GPIB interface card for details.

HP Basic addresses the signal generator at 719. The GPIB card is addressed at 7 and the signal generator at 19. The GPIB address designator for other libraries is typically GPIB0 or GPIB1.

The following sections contain HP Basic and C++ lines of programming removed from the programming interface examples in Chapter 3, these portions of programming demonstrate the important features to consider when developing programming for use with the GPIB interface.

Interface Check using HP Basic and GPIB

This portion of the program from the example program "Interface Check using HP Basic and GPIB" on page 56, causes the signal generator to perform an instrument reset. The SCPI command *RST places the signal generator into a pre-defined state and the remote annunciator (R) appears on the front panel display.

The following program example is available on the signal generator Documentation CD-ROM as basicex1.txt. For the full text of this program, refer to "Interface Check using HP Basic and GPIB" on page 56 or to the signal generator's documentation CD-ROM.

```
160   Sig_gen=719     ! Declares a variable to hold the signal generator's address
170   LOCAL Sig_gen   ! Places the signal generator into Local mode
180   CLEAR Sig_gen   ! Clears any pending data I/O and resets the parser
190   REMOTE 719      ! Puts the signal generator into remote mode
200   CLEAR SCREEN    ! Clears the controllers display
210   REMOTE 719
220   OUTPUT Sig_gen;"*RST"  ! Places the signal generator into a defined state
```

Interface Check Using NI-488.2 and C++

This portion of the program from the example program "Interface Check Using NI-488.2 and C++" on page 57, uses the NI-488.2 library to verify that the GPIB connections and interface are functional.

The following program example is available on the signal generator Documentation CD-ROM as niex1.cpp. For the full text of this program, refer to "Interface Check Using NI-488.2 and C++" on page 57 or to the signal generator's documentation CD-ROM.

```
#include "stdafx.h"
#include <iostream>
#include "windows.h"
#include "Decl-32.h"
using namespace std;

int GPIB0=   0;         // Board handle
Addr4882_t Address[31]; // Declares an array of type Addr4882_t

int main(void)

{
     int sig;                         // Declares a device descriptor variable
     sig = ibdev(0, 19, 0, 13, 1, 0); // Aquires a device descriptor
     ibclr(sig);                      // Sends device clear message to signal generator
     ibwrt(sig, "*RST", 4);           // Places the signal generator into a defined state
```

Using LAN

The signal generator can be remotely programmed via a 10Base-T LAN interface and LAN-connected computer using one of several LAN interface protocols. The LAN allows instruments to be connected together and controlled by a LAN-based computer. LAN and its associated interface operations are defined in the IEEE 802.2 standard. For more information refer to http://www.ieee.org.

NOTE For more information on configuring your signal generator for LAN, refer to the *User's Guide* for your signal generator.

The signal generator supports the following LAN interface protocols:

- VXI-11 (See page 30)
- Sockets LAN (See page 31)
- Telephone Network (TELNET) (See page 32)
- File Transfer Protocol (FTP) (See page 36)

VXI-11 and sockets LAN are used for general programming using the LAN interface, TELNET is used for interactive, one command at a time instrument control, and FTP is for file transfer.

NOTE For more information on configuring the signal generator to communicate over the LAN, refer to "Using VXI-11" on page 30.

The following sections contain information on selecting and connecting IO libraries and LAN interface hardware that are required to remotely program the signal generator via LAN to a LAN-based computer and combining those choices with one of several possible LAN interface protocols.

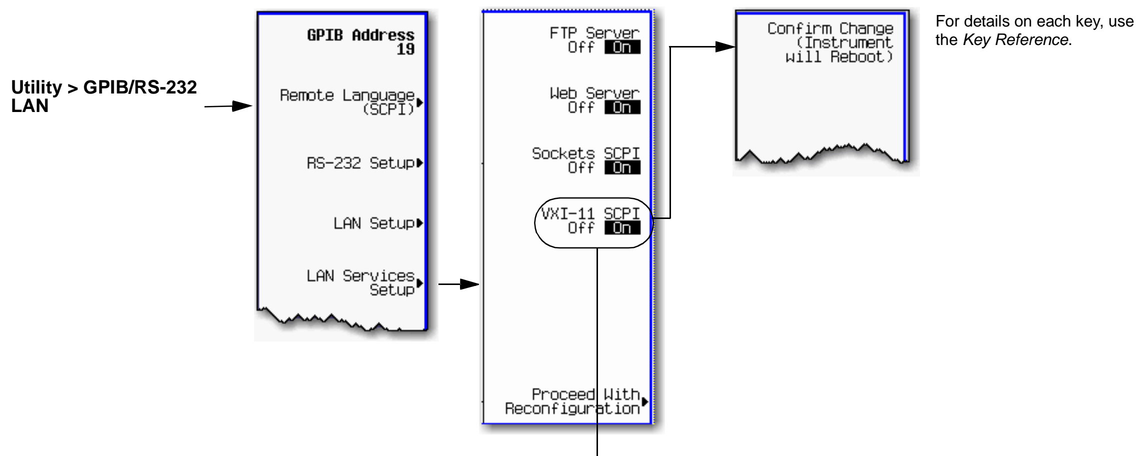
- "Setting Up the LAN Interface" on page 23
- "Verifying LAN Functionality" on page 26

Setting Up the LAN Interface

For LAN operation, the signal generator must be connected to the LAN, and an IP address must be assigned to the signal generator either manually or by using DHCP client service. Your system administrator can tell you which method to use.

NOTE Verify that the signal generator is connected to the 10Base-T LAN cable. For more information on 10Base-T LAN, refer to “Using LAN” on page 22.

Configuring the VXI-11 for LAN



NOTE
To communicate with the signal generator over the LAN, you must enable the VXI-11 SCPI service. Select VXI-11 until On is highlighted. (Default condition is On.)

Use a 10Base-T LAN cable to connect the signal generator to the LAN. For more information refer to *http://www.ieee.org*.

Manual Configuration

The **Hostname** softkey is only available when **LAN Config Manual DHCP** is set to **Manual**.

To remotely access the signal generator from a different LAN subnet, you must also enter the subnet mask and default gateway. See your system administrator for more information.

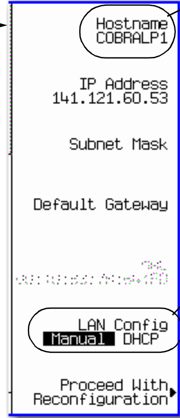
For more information on the manual configuration, refer to “Manually Configuring the E8663B LAN” on page 24.

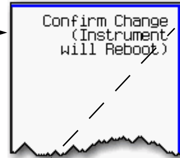
Manually Configuring the E8663B LAN

Utility > IO Config

The Hostname softkey is available only when LAN config Manual DHCP is set to Manual or when AUTO (DHCP) is unable to connect automatically and defaults to Manual.

Your hostname can be up to 20 characters long.






```
SCPI commands:
:SYSTem:COMMunicate:LAN:CONFig MANual
:SYSTem:COMMunicate:LAN:CONFig?
```

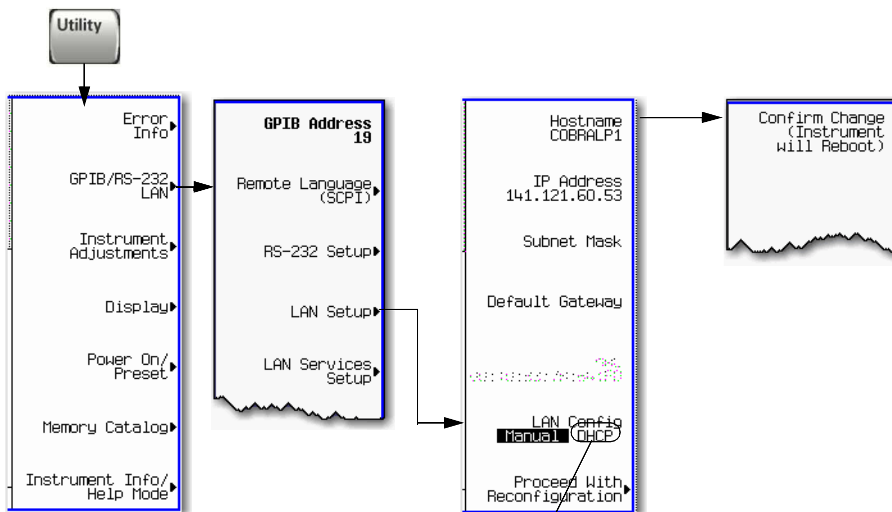
For details on each key, use the *Key Reference*.

DHCP Configuration

If the DHCP server uses dynamic DNS to link the hostname with the assigned IP address, the hostname may be used in place of the IP address. Otherwise, the hostname is not usable.

For more information on the DHCP configuration, refer to "Configuring the DHCP LAN" on page 25.

Configuring the DHCP LAN



DHCP: Request a new IP address from the DHCP server each power cycle.

Confirming this action configures the signal generator as a DHCP client. In DHCP mode, the signal generator will request a new IP address from the DHCP server upon rebooting to determine the assigned IP address.

NOTE
Use a 10Base-T LAN cable to connect the signal generator to the LAN.

```
SCPI commands:
:SYSTem:COMMunicate:LAN:CONFig DHCP
:SYSTem:COMMunicate:LAN:CONFig?
```

For details on each key, use the Key Reference.

Verifying LAN Functionality

Verify the communications link between the computer and the signal generator remote file server using the ping utility. Compare your ping response to those described in Table 2-1 on page 27.

NOTE For additional information on troubleshooting your LAN connection, refer to the Help in the Agilent IO Libraries and documentation for LAN connections and problems.

From a UNIX® workstation, type (UNIX is a registered trademark of the Open Group):

```
ping <hostname or IP address> 64 10
```

where `<hostname or IP address>` is your instrument's name or IP address, 64 is the packet size, and 10 is the number of packets transmitted. Type `man ping` at the UNIX prompt for details on the ping command.

From the MS-DOS® Command Prompt or Windows environment, type:[1]

```
ping -n 10 <hostname or IP address>
```

where `<hostname or IP address>` is your instrument's name or IP address and 10 is the number of echo requests. Type `ping` at the command prompt for details on the ping command.

NOTE In DHCP mode, if the DHCP server uses dynamic DNS to link the hostname with the assigned IP address, the hostname may be used in place of the IP address. Otherwise, the hostname is not usable and you must use the IP address to communicate with the signal generator over the LAN.

For additional information on troubleshooting your LAN connection, refer to the Help in the Agilent IO Libraries and documentation for LAN connections and problems.

---

1. MS-DOS is a registered trademark of Microsoft Corporation

Table 2-1 Ping Responses

Normal Response for UNIX	A normal response to the ping command will be a total of 9 or 10 packets received with a minimal average round-trip time. The minimal average will be different from network to network. LAN traffic will cause the round-trip time to vary widely.
Normal Response for DOS or Windows	A normal response to the ping command will be a total of 9 or 10 packets received if 10 echo requests were specified.
Error Messages	If error messages appear, then check the command syntax before continuing with troubleshooting. If the syntax is correct, resolve the error messages using your network documentation or by consulting your network administrator. If an unknown host error message appears, try using the IP address instead of the hostname. Also, verify that the host name and IP address for the signal generator have been registered by your IT administrator. Check that the hostname and IP address are correctly entered in the node names database. To do this, enter the `nslookup <hostname>` command from the command prompt.
No Response	If there is no response from a ping, no packets were received. Check that the typed address or hostname matches the IP address or hostname assigned to the signal generator in the System **LAN Setup** menu. For more information, refer to "Configuring the DHCP LAN" on page 25. Ping each node along the route between your workstation and the signal generator, starting with your workstation. If a node doesn't respond, contact your IT administrator. If the signal generator still does not respond to ping, you should suspect a hardware problem. • Check the signal generator LAN connector lights • Verify the hostname is not being used with DHCP addressing
Intermittent Response	If you received 1 to 8 packets back, there maybe a problem with the network. In networks with switches and bridges, the first few pings may be lost until these devices 'learn' the location of hosts. Also, because the number of packets received depends on your network traffic and integrity, the number might be different for your network. Problems of this nature are best resolved by your IT department.

Using Interactive IO

Use the VISA Assistant utility available in the Agilent IO Libraries Suite to verify instrument communication over the LAN interface. Refer to the section on the "IO Libraries and Programming Languages" on page 4 for more information.

The Agilent IO Libraries Suite is supported on all platforms except Windows NT. If you are using Windows NT, refer to section below on using the VISA Assistant to verify LAN communication. See the section on "Windows NT and Agilent IO Libraries M (and Earlier)" on page 5 for more information.

NOTE The following sections are specific to Agilent IO Libraries versions M and earlier and apply only to the Windows NT platform.

Using VISA Assistant

Use the VISA Assistant, available with the Agilent IO Library versions M and earlier, to communicate with the signal generator over the LAN interface. However, you must manually configure the VISA LAN client. Refer to the Help menu for instructions on configuring and running the VISA Assistant program.

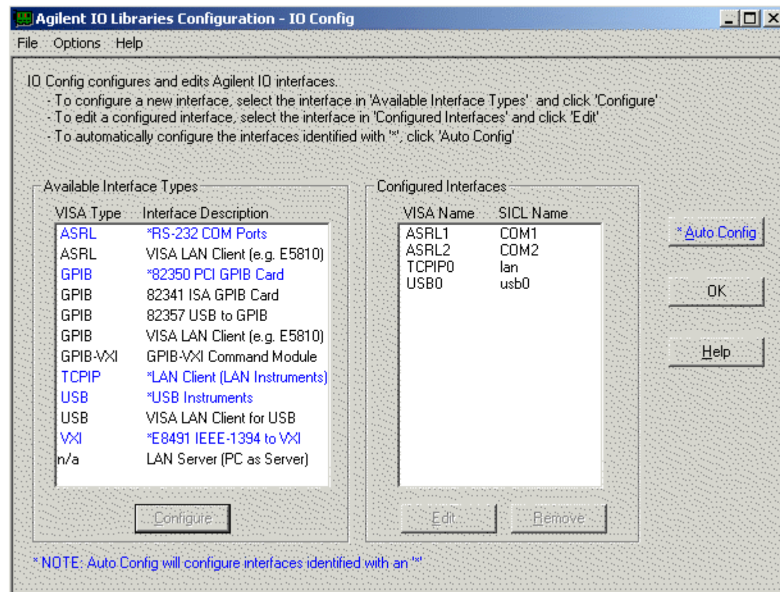
1. Run the IO Config program.
2. Click on TCPIP0 in the Available Interface Types text box.
3. Click the **Configure** button. Then Click **OK** to use the default settings.
4. Click on TCPIP0 in the Configured Interfaces text box.
5. Click **Edit...**
6. Click the **Edit VISA Config...** button.
7. Click the **Add device** button.
8. Enter the TCPIP address of the signal generator. Leave the Device text box empty.
9. Click the **OK** button in this form and all subsequent forms to exit the IO Config program.

If You Have Problems

1. Verify the signal generator's IP address is valid and that no other instrument is using the IP address.
2. Switch between manual LAN configuration and DHCP using the front-panel **LAN Config** softkey and run the ping program using the different IP addresses.

NOTE For Agilent IO Libraries versions M and earlier, you must manually configure the VISA LAN client in the IO Config program if you want to use the VISA Assistant to verify LAN configuration. Refer to the IO Libraries Installation Guide for information on configuring IO interfaces. The IO Config program interface is shown in Figure 2-3 on page 31.

Figure 2-2 IO Config Form (Windows NT)



Check to see that the Default Protocol is set to Automatic.

1. Run the IO Config program
2. Click on TCPIP in the Configured Interfaces text box. If there is no TCPIP0 in the box, follow the steps shown in the section "Using VISA Assistant" on page 28
3. Click the **Edit** button.
4. Click the radio button for AUTO (automatically detect protocol).
5. Click **OK**, **OK** to end the IO Config program.

Using VXI-11

The signal generator supports the LAN interface protocol described in the VXI-11 standard. VXI-11 is an instrument control protocol based on Open Network Computing/Remote Procedure Call (ONC/RPC) interfaces running over TCP/IP. It is intended to provide GBIB capabilities such as SRQ (Service Request), status byte reading, and DCAS (Device Clear State) over a LAN interface. This protocol is a good choice for migrating from GPIB to LAN as it has full Agilent VISA/SICL support.

NOTE It is recommended that the VXI-11 protocol be used for instrument communication over the LAN interface.

Configuring for VXI-11

The Agilent IO library has a program, IO Config, that is used to setup the computer/signal generator interface for the VXI-11 protocol. Download the latest version of the Agilent IO library from the Agilent website. Refer to the Agilent IO library user manual, documentation, and Help menu for information on running the IO Config program and configuring the VXI-11 interface.

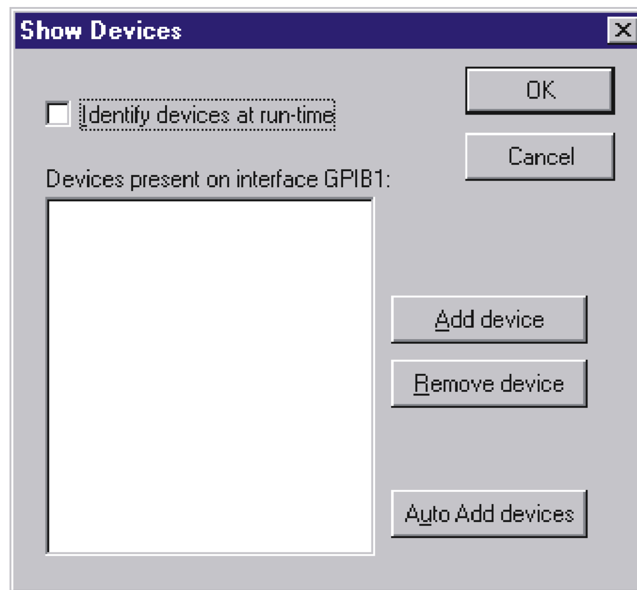
Use the IO Config program to configure the LAN client. Once the computer is configured for a LAN client, you can use the VXI-11 protocol and the VISA library to send SCPI commands to the signal generator over the LAN interface. Example programs for this protocol are included in “LAN Programming Interface Examples” on page 87 of this programming guide.

NOTE To communicate with the signal generator over the LAN interface you must enable the VXI-11 SCPI service. For more information, refer to “Configuring the DHCP LAN” on page 25.

If you are using the Windows NT platform, refer to “Windows NT and Agilent IO Libraries M (and Earlier)” on page 5 for information on using Agilent IO Libraries versions M or earlier to configure the interface.

For Agilent IO library version J.01.0100, the “Identify devices at run-time” check box must be unchecked. Refer to Figure 2-3.

Figure 2-3 Show Devices Form (Agilent IO Library version J.01.0100)



Using Sockets LAN

NOTE Windows XP operating systems and newer can use this section to better understand how to use the signal generator with port settings. For more information, refer to the help software of the IO libraries being used.

Sockets LAN is a method used to communicate with the signal generator over the LAN interface using the Transmission Control Protocol/Internet Protocol (TCP/IP). A socket is a fundamental technology used for computer networking and allows applications to communicate using standard mechanisms built into network hardware and operating systems. The method accesses a port on the signal generator from which bidirectional communication with a network computer can be established.

Sockets LAN can be described as an internet address that combines Internet Protocol (IP) with a device port number and represents a single connection between two pieces of software. The socket can be accessed using code libraries packaged with the computer operating system. Two common versions of socket libraries are the Berkeley Sockets Library for UNIX systems and Winsock for Microsoft operating systems.

Your signal generator implements a sockets Applications Programming Interface (API) that is compatible with Berkeley sockets, for UNIX systems, and Winsock for Microsoft systems. The signal generator is also compatible with other standard sockets APIs. The signal generator can be controlled using SCPI commands that are output to a socket connection established in your program.

Before you can use sockets LAN, you must select the signal generator's sockets port number to use:

- Standard mode. Available on port 5025. Use this port for simple programming.
- TELNET mode. The telnet SCPI service is available on port 5023.

NOTE On the E8663B, the signal generator will accept references to telnet SCPI service at port 7777 and sockets SCPI service at port 7778.

An example using sockets LAN is given in "LAN Programming Interface Examples" on page 87 of this programming guide.

Using Telnet LAN

Telnet provides a means of communicating with the signal generator over the LAN. The Telnet client, run on a LAN connected computer, will create a login session on the signal generator. A connection, established between computer and signal generator, generates a user interface display screen with `SCPI>` prompts on the command line.

Using the Telnet protocol to send commands to the signal generator is similar to communicating with the signal generator over GPIB. You establish a connection with the signal generator and then send or receive information using SCPI commands. Communication is interactive: one command at a time.

NOTE The Windows 2000®[1] operating systems use a command prompt style interface for the Telnet client. Refer to the Figure 2-6 on page 35 for an example of this interface.

Windows XP operating systems and newer can use this section to better understand how to use the signal generator with port settings. For more information, refer to the help software of the IO libraries being used.

The following telnet LAN connections are discussed:

- "Using Telnet and MS-DOS Command Prompt" on page 32
- "Using Telnet On a PC With a Host/Port Setting Menu GUI" on page 33
- "Using Telnet On Windows 2000" on page 34
- "The Standard UNIX Telnet Command" on page 35

A Telnet example is provided in "Unix Telnet Example" on page 35.

Using Telnet and MS-DOS Command Prompt

1. On your PC, click **Start** > **Programs** > **Command Prompt**.
2. At the command prompt, type in `telnet`.

---

1. Windows 2000 is a registered trademark of Microsoft Corporation.

3. Press the **Enter** key. The Telnet display screen will be displayed.
4. Click on the **Connect** menu then select **Remote System**. A connection form (Figure 2-4) is displayed.

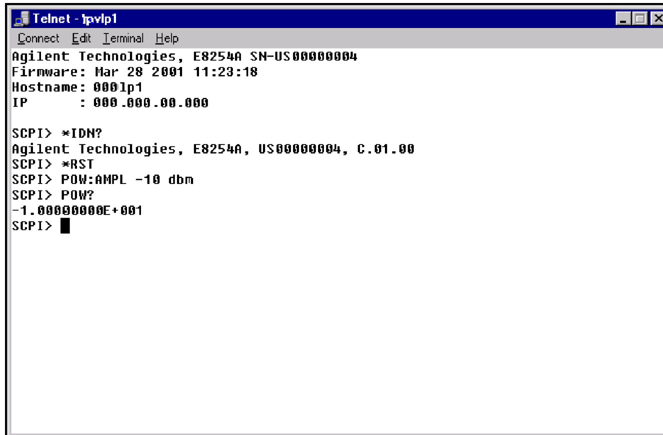
Figure 2-4Connect Form (Agilent IO Library version J.01.0100)

5. Enter the **hostname**, **port number**, and **TermType** then click **Connect**.
   - Host Name–IP address or hostname
   - Port–5023
   - Term Type–vt100
6. At the `SCPI>` prompt, enter SCPI commands. Refer to Figure 2-5 on page 34.
7. To signal device clear, press **Ctrl-C** on your keyboard.
8. Select **Exit** from the **Connect** menu and type `exit` at the command prompt to end the Telnet session.

Using Telnet On a PC With a Host/Port Setting Menu GUI

1. On your PC, click **Start** > **Run**.
2. Type `telnet` then click the **OK** button. The Telnet connection screen will be displayed.
3. Click on the **Connect** menu then select **Remote System**. A connection form is displayed. See Figure 2-4.
4. Enter the **hostname**, **port number**, and **TermType** then click **Connect**.
   - Host Name–signal generator's IP address or hostname
   - Port–5023
   - Term Type–vt100
5. At the `SCPI>` prompt, enter SCPI commands. Refer to Figure 2-5 on page 34.
6. To signal device clear, press **Ctrl-C**.
7. Select **Exit** from the **Connect** menu to end the Telnet session.

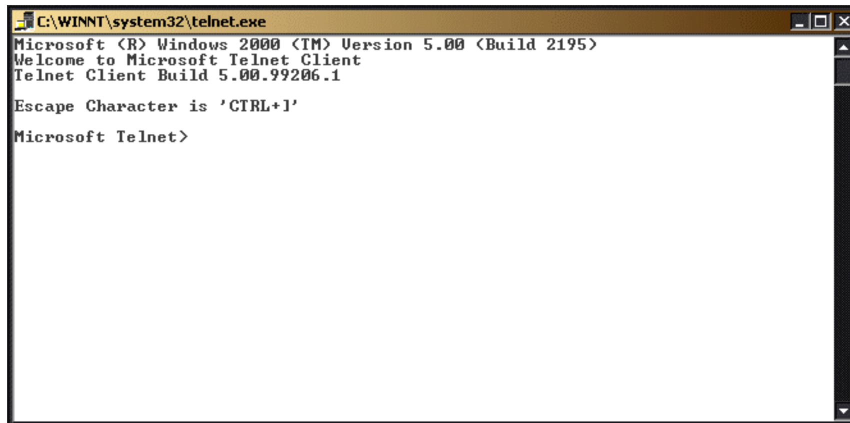
Figure 2-5 Telnet Window (Windows 2000)



Using Telnet On Windows 2000

1. On your PC, click **Start** > **Run**.
2. Type `telnet` in the run text box, then click the **OK** button. The Telnet connection screen will be displayed. See Figure 2-6 on page 35 (Windows 2000).
3. Type `open` at the prompt and then press the **Enter** key. The prompt will change to `(to)`.
4. At the `(to)` prompt, enter the signal generator's IP address followed by a space and `5023`, which is the Telnet port associated with the signal generator.
5. At the `SCPI>` prompt, enter SCPI commands. Refer to commands shown in Figure 2-5 on page 34.
6. To escape from the SCPI> session type `Ctrl-]`.
7. Type `quit` at the prompt to end the Telnet session.

Figure 2-6 Telnet 2000 Window



The Standard UNIX Telnet Command

Synopsis

`telnet [host [port]]`

Description

This command is used to communicate with another host using the Telnet protocol. When the command `telnet` is invoked with `host` or `port` arguments, a connection is opened to the host, and input is sent from the user to the host.

Options and Parameters

The command `telnet` operates in character-at-a-time or line-by-line mode. In line-by-line mode, typed text is echoed to the screen. When the line is completed (by pressing the **Enter** key), the text line is sent to `host`. In character-at-a-time mode, text is echoed to the screen and sent to `host` as it is typed. At the UNIX prompt, type `man telnet` to view the options and parameters available with the `telnet` command.

NOTE If your Telnet connection is in line-by-line mode, there is no local echo. This means you cannot see the characters you are typing until you press the **Enter** key. To remedy this, change your Telnet connection to character-by-character mode. Escape out of Telnet, and at the `telnet>` prompt, type `mode char`. If this does not work, consult your Telnet program's documentation.

Unix Telnet Example

To connect to the instrument with host name `myInstrument` and port number `7778`, enter the following command on the command line: `telnet myInstrument 5023`

When you connect to the signal generator, the UNIX window will display a welcome message and a SCPI command prompt. The instrument is now ready to accept your SCPI commands. As you type SCPI commands, query results appear on the next line. When you are done, break the Telnet connection using an escape character. For example, **Ctrl-]**,where the control key and the **]** are pressed at the same time. The following example shows Telnet commands:

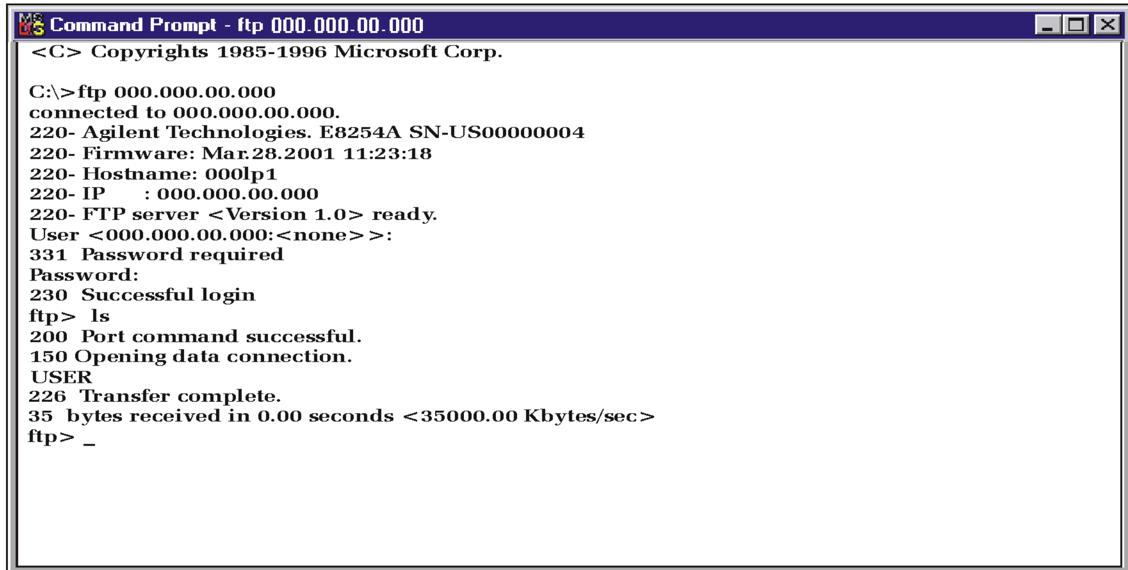
```
$ telnet myinstrument 5023

Trying....

Connected to signal generator

Escape character is '^]'.

Agilent Technologies, E44xx SN-US00000001

Firmware:

Hostname: your instrument

IP :xxx.xx.xxx.xxx

SCPI>
```

Using FTP

FTP allows users to transfer files between the signal generator and any computer connected to the LAN. For example, you can use FTP to download instrument screen images to a computer. When logged onto the signal generator with the FTP command, the signal generator's file structure can be accessed. Figure 2-7 shows the FTP interface and lists the directories in the signal generator's user level directory.

NOTE File access is limited to the signal generator's /user directory.

Figure 2-7 FTP Screen



The following steps outline a sample FTP session from the MS-DOS Command Prompt:

1. On the PC click **Start** > **Programs** > **Command Prompt**.
2. At the command prompt enter:

   `ftp < IP address >` or `< hostname >`

3. At the user name prompt, press **enter**.
4. At the password prompt, press **enter**.

   You are now in the signal generator’s user directory. Typing `help` at the command prompt will show you the FTP commands that are available on your system.

5. Type `quit` or `bye` to end your FTP session.
6. Type `exit` to end the command prompt session.

Using RS-232

NOTE The E8663B's **AUXILIARY INTERFACE** connector is compatible with ANSI/EIA232 (RS-232) serial connection but GPIB and LAN are recommended for making faster measurements and when downloading files. Refer to the *User's Guide*.

The RS-232 serial interface can be used to communicate with the signal generator. The RS-232 connection is standard on most PCs and can be connected to the signal generator's rear-panel connector using the cable described in Table 2-2 on page 40. Many functions provided by GPIB, with the exception of indefinite blocks, parallel polling, serial polling, GET, non-SCPI remote languages, SRQ, and remote mode are available using the RS-232 interface.

The serial port sends and receives data one bit at a time, therefore RS-232 communication is slow. The data transmitted and received is usually in ASCII format with SCPI commands being sent to the signal generator and ASCII data returned.

The following sections contain information on selecting and connecting IO libraries and RS-232 interface hardware on the signal generator to a computer's RS-232 connector.

- "Selecting IO Libraries for RS-232" on page 38
- "Setting Up the RS-232 Interface" on page 39
- "Verifying RS-232 Functionality" on page 41

Selecting IO Libraries for RS-232

The IO libraries can be downloaded from the National Instrument website, *http://www.ni.com*, or Agilent's website, *http://www.agilent.com*. The following is a discussion on these libraries.

HP Basic — The HP Basic language has an extensive IO library that can be used to control the signal generator over the RS-232 interface. This library has many low level functions that can be used in BASIC applications to control the signal generator over the RS-232 interface.

VISA — VISA is an IO library used to develop IO applications and instrument drivers that comply with industry standards. It is recommended that the VISA library be used for programming the signal generator. The NI-VISA and Agilent VISA libraries are similar implementations of VISA and have the same commands, syntax, and functions. The differences are in the lower level IO libraries used to communicate over the RS-232; NI-488.2 and SICL respectively.

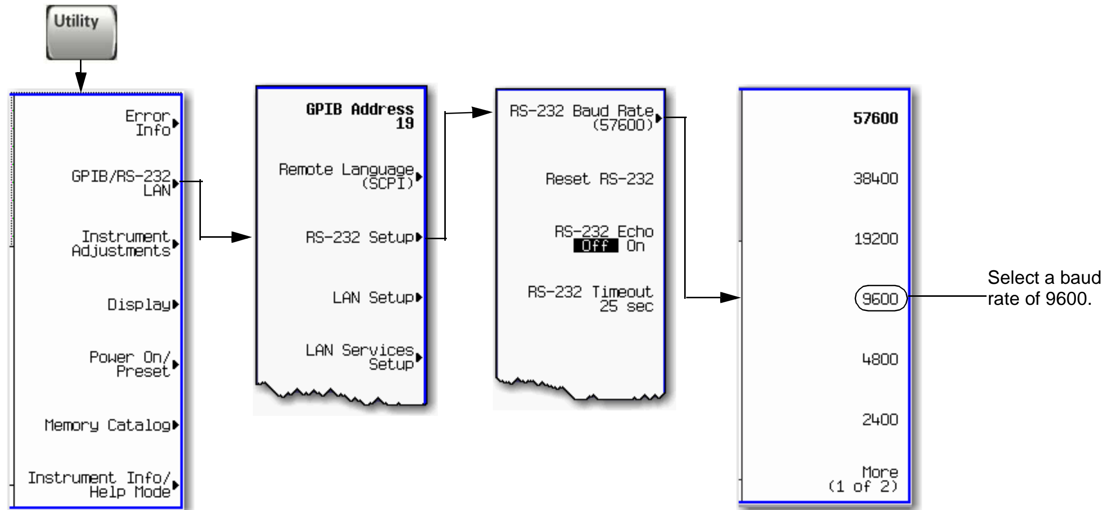
NI-488.2 — NI-488.2 IO libraries can be used to develop applications for the RS-232 interface. See National Instrument's website for information on NI-488.2.

SICL — Agilent SICL can be used to develop applications for the RS-232 interface. See Agilent's website for information on SICL.

CAUTION Because of the potential for portability problems, running Agilent SICL without the VISA overlay is not recommended by Agilent Technologies.

Setting Up the RS-232 Interface

1. Setting the RS-232 Interface Baud Rate



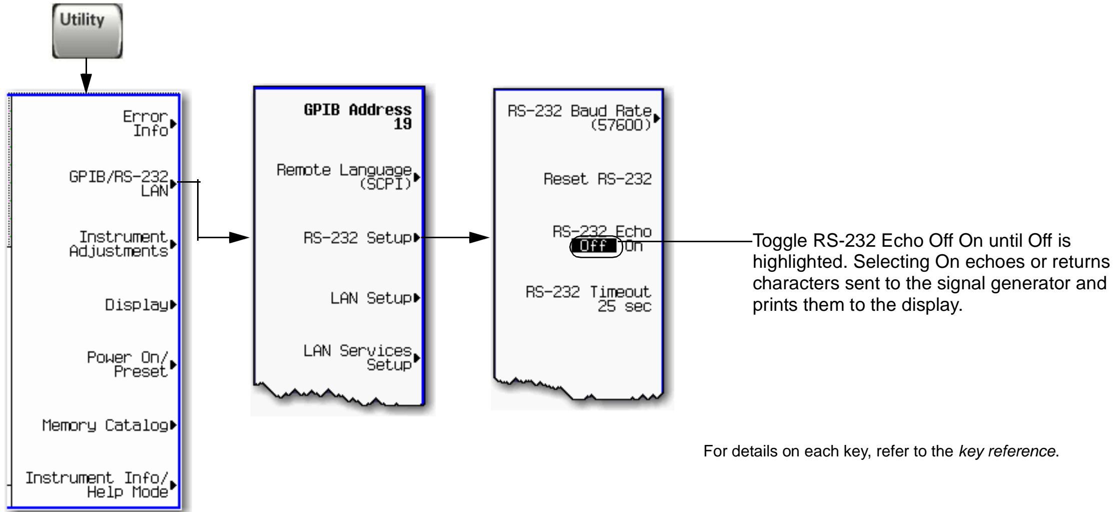
For details on each key, refer to the *Key Reference*.

NOTE Use baud rates 57600 or lower only. Select the signal generator's baud rate to match the baud rate of your computer or UNIX workstation or adjust the baud rate settings on your computer to match the baud rate setting of the signal generator.

The default baud rate for VISA is 9600. This baud rate can be changed with the "VI_ATTR_ASRL_BAUD" VISA attribute.

---

2. Setting the RS-232 Echo Softkey



3. Connect an RS-232 cable from the computer's serial connector to the E8663B's **AUXILIARY INTERFACE** connector. Refer to Table 2-2 for RS-232 cable information.

Table 2-2 RS-232 Serial Interface Cable

Quantity	Description	Agilent Part Number
1	Serial RS-232 cable 9-pin (male) to 9-pin (female)	8120-6188

NOTE Any 9 pin (male) to 9 pin (female) straight-through cable that directly wires pins 2, 3, 5, 7, and 8 may be used.

Verifying RS-232 Functionality

You can use the HyperTerminal program available on your computer to verify the RS-232 interface functionality. To run the HyperTerminal program, connect the RS-232 cable between the computer and the signal generator and perform the following steps:

1. On the PC click **Start** > **Programs** > **Accessories** > **Communications** > **HyperTerminal**.
2. Select **HyperTerminal**.
3. Enter a name for the session in the text box and select an icon.
4. Select **COM1** (COM2 can be used if COM1 is unavailable).
5. In the COM1 (or COM2, if selected) properties, set the following parameters:
   - Bits per second: 9600 must match signal generator’s baud rate; for more information, refer to “Setting Up the RS-232 Interface” on page 39.
   - Data bits: 8
   - Parity: None
   - Stop bits: 1
   - Flow Control: None

NOTE Flow control, via the RTS line, is driven by the signal generator. For the purposes of this verification, the controller (PC) can ignore this if flow control is set to None. However, to control the signal generator programmatically or download files to the signal generator, you *must* enable RTS-CTS (hardware) flow control on the controller. Note that only the RTS line is currently used.

---

6. Go to the HyperTerminal window and select **File** > **Properties**.
7. Go to **Settings** > **Emulation** and select **VT100**.
8. Leave the **Backscroll buffer lines** set to the default value.
9. Go to **Settings** > **ASCII Setup**.
10. Check the first two boxes and leave the other boxes as default values.

Once the connection is established, enter the SCPI command `*IDN?` followed by `<Ctrl j>` in the HyperTerminal window. The `<Ctrl j>` is the new line character (on the keyboard press the **Cntrl** key and the **j** key simultaneously).

The signal generator should return a string similar to the following, depending on model:

Agilent Technologies *<instrument model name and number>*`, US40000001,C.02.00`

Character Format Parameters

The signal generator uses the following character format parameters when communicating via RS-232:

- Character Length: Eight data bits are used for each character, excluding start, stop, and parity bits.
- Parity Enable: Parity is disabled (absent) for each character.
- Stop Bits: One stop bit is included with each character.

If You Have Problems

1. Verify that the baud rate, parity, and stop bits are the same for the computer and signal generator.
2. Verify that the RS-232 cable is identical to the cable specified in Table 2-2.
3. Verify that the application is using the correct computer COM port and that the RS-232 cable is properly connected to that port.
4. Verify that the controller’s flow control is set to RTS-CTS.

RS-232 Programming Interface Examples

- "Interface Check Using HP BASIC" on page 43
- "Interface Check Using VISA and C" on page 43
- "Queries Using HP Basic and RS-232" on page 44
- "Queries for RS-232 Using VISA and C" on page 44

Before Using the Examples

Before using the examples: On the signal generator select the following settings:

- Baud Rate - 9600 must match computer's baud rate
- RS-232 Echo - Off

The following sections contain HP Basic and C lines of programming removed from the programming interface examples in Chapter 3, these portions of programming demonstrate the important features to consider when developing programming for use with the RS-232 interface.

NOTE For LAN programming examples, refer to "LAN Programming Interface Examples" on page 87.

Interface Check Using HP BASIC

This portion of the program from the example program "Interface Check Using HP BASIC" on page 118, causes the signal generator to perform an instrument reset. The SCPI command *RST will place the signal generator into a pre-defined state.

The serial interface address for the signal generator in this example is 9. The serial port used is COM1 (Serial A on some computers). Refer to "Using RS-232" on page 38 for more information.

The following program example is available on the signal generator's Documentation CD-ROM as rs232ex1.txt. For the full text of this program, refer to "Interface Check Using HP BASIC" on page 118 or to the signal generator's documentation CD-ROM.

```
170     CONTROL 9,0;1        ! Resets the RS-232 interface
180     CONTROL 9,3;9600     ! Sets the baud rate to match the sig gen
190     STATUS 9,4;Stat      ! Reads the value of register 4
200     Num=BINAND(Stat,7)   ! Gets the AND value
210     CONTROL 9,4;Num      ! Sets parity to NONE
220     OUTPUT 9;"*RST"      ! Outputs reset to the sig gen
```

Interface Check Using VISA and C

This portion of the program from the example program "Interface Check Using VISA and C" on page 43, uses VISA library functions to communicate with the signal generator. The program verifies that the RS-232 connections and interface are functional. In this example the COM2 port is used. The serial port is referred to in the VISA library as 'ASRL1' or 'ASRL2' depending on the computer serial port you are using.

The following program example is available on the signal generator Documentation CD-ROM as

rs232ex1.cpp. For the full text of this program, refer to "Interface Check Using VISA and C" on page 43 or to the signal generator's documentation CD-ROM.

```
int baud=9600;// Set baud rate to 9600

ViSession defaultRM, vi;// Declares a variable of type ViSession
// for instrument communication on COM 2 port
ViStatus viStatus = 0;
                        // Opens session to RS-232 device at serial port 2
viStatus=viOpenDefaultRM(&defaultRM);
viStatus=viOpen(defaultRM, "ASRL2::INSTR", VI_NULL, VI_NULL, &vi);

viStatus=viEnableEvent(vi, VI_EVENT_IO_COMPLETION, VI_QUEUE,VI_NULL);

viClear(vi);// Sends device clear command
// Set attributes for the session
viSetAttribute(vi,VI_ATTR_ASRL_BAUD,baud);
viSetAttribute(vi,VI_ATTR_ASRL_DATA_BITS,8);
```

Queries Using HP Basic and RS-232

This portion of the program from the example program "Queries Using HP Basic and RS-232" on page 44, example program demonstrates signal generator query commands over RS-232. Query commands are of the type *IDN? and are identified by the question mark that follows the mnemonic.

Start HP Basic, type in the following commands, and then RUN the program:

The following program example is available on the signal generator Documentation CD-ROM as rs232ex2.txt. For the full text of this program, refer to "Queries Using HP Basic and RS-232" on page 44 or to the signal generator's documentation CD-ROM.

```
190    OUTPUT 9;"*IDN?"           ! Querys the sig gen ID
200    ENTER 9;Str$               ! Reads the ID
210    WAIT 2                     ! Waits 2 seconds
220    PRINT "ID =",Str$          ! Prints ID to the screen
230    OUTPUT 9;"POW:AMPL -5 dbm" ! Sets the the power level to -5 dbm
240    OUTPUT 9;"POW?"            ! Querys the power level of the sig gen
```

Queries for RS-232 Using VISA and C

This portion of the program the example program "Queries for RS-232 Using VISA and C" on page 44, uses VISA library functions to communicate with the signal generator. The program verifies that the RS-232 connections and interface are functional.

The following program example is available on the signal generator Documentation CD-ROM as rs232ex2.cpp. For the full text of this program, refer to "Queries for RS-232 Using VISA and C" on page 44 or to the signal generator's documentation CD-ROM.

```
status = viOpenDefaultRM(&defaultRM);// Initializes the system
// Open communication with Serial Port 2
status = viOpen(defaultRM, "ASRL2::INSTR", VI_NULL, VI_NULL, &instr);
```

3 Programming Examples

- "Using the Programming Interface Examples" on page 46
- "GPIB Programming Interface Examples" on page 52
- "LAN Programming Interface Examples" on page 87
- "RS-232 Programming Interface Examples" on page 118

Using the Programming Interface Examples

The programming examples for remote control of the signal generator use the GPIB, LAN, and RS-232 interfaces and demonstrate instrument control using different IO libraries and programming languages. Many of the example programs in this chapter are interactive; the user will be prompted to perform certain actions or verify signal generator operation or functionality. Example programs are written in the following languages:

HP Basic	C#
C/C++	Microsoft Visual Basic 6.0
Java	MATLAB
Perl	

These example programs are also available on the signal generator Documentation CD-ROM, enabling you to cut and paste the examples into a text editor.

NOTE The example programs set the signal generator into remote mode; front panel keys, except the **Local** key are disabled. Press the **Local** key to revert to manual operation.

To have the signal generator's front panel update with changes caused by remote operations, enable the signal generator's Update in Remote function.

NOTE The Update in Remote function will slow test execution. For faster test execution, disable the Update in Remote function. (For more information, refer to "Configuring the Display for Remote Command Setups" on page 12.

Programming Examples Development Environment

The C/C++ examples were written using an IBM-compatible personal computer (PC), configured as follows:

- Pentium® processor (Pentium is a registered trademark of Intel Corporation.)
- Windows NT 4.0 operating system
- C/C++ programming language with the Microsoft Visual C++ 6.0 IDE
- National Instruments PCI-GPIB interface card or Agilent GPIB interface card
- National Instruments VISA Library or Agilent VISA library
- COM1 or COM2 serial port available
- LAN interface card

The HP Basic examples were run on a UNIX 700 series workstation.

Running C++ Programs

When using Microsoft Visual C++ 6.0 to run the example programs, include the following files in your project.

When using the VISA library:

- add the visa32.lib file to the Resource Files
- add the visa.h file to the Header Files

When using the NI-488.2 library:

- add the GPIB-32.OBJ file to the Resource Files
- add the windows.h file to the Header Files
- add the Deci-32.h file to the Header Files

For information on the NI-488.2 library and file requirements refer to the National Instrument website. For information on the VISA library see the Agilent website or National Instrument's website.

NOTE To communicate with the signal generator over the LAN interface you must enable the VXI-11 SCPI service. For more information, refer to "Configuring the VXI-11 for LAN" on page 23.

C/C++ Examples

- "Interface Check for GPIB Using VISA and C" on page 58
- "Queries for RS-232 Using VISA and C" on page 122
- "Local Lockout Using NI-488.2 and C++" on page 60
- "Queries Using NI-488.2 and Visual C++" on page 63
- "Queries for GPIB Using VISA and C" on page 65
- "Generating a CW Signal Using VISA and C" on page 67
- "Generating an Externally Applied AC-Coupled FM Signal Using VISA and C" on page 69
- "Generating an Internal FM Signal Using VISA and C" on page 71
- "Generating a Step-Swept Signal Using VISA and C++" on page 73
- "Reading the Data Questionable Status Register Using VISA and C" on page 79
- "Reading the Service Request Interrupt (SRQ) Using VISA and C" on page 83
- "VXI-11 Programming Using SICL and C++" on page 88
- "VXI-11 Programming Using VISA and C++" on page 89
- "Sockets LAN Programming and C" on page 91
- "Interface Check Using VISA and C" on page 119
- "Queries for RS-232 Using VISA and C" on page 122

Running C# Examples

To run the example program *State_Files.cs* on page 159, you must have the .NET framework installed on your computer. You must also have the Agilent IO Libraries installed on your computer. The .NET framework can be downloaded from the Microsoft website. For more information on running C# programs using .NET framework, see Chapter 5.

NOTE To communicate with the signal generator over the LAN interface you must enable the VXI-11 SCPI service. For more information, refer to "Configuring the VXI-11 for LAN" on page 23.

Running Basic Examples

The BASIC programming interface examples provided in this chapter use either HP Basic or Visual Basic 6.0 languages.

Visual Basic 6.0® Programming Examples [1]

To run the example programs written in Visual Basic 6.0 you must include references to the IO Libraries. For more information on VISA and IO libraries, refer to the Agilent VISA User's Manual, available on Agilent's website: *http://www.agilent.com*. In the Visual Basic IDE (Integrated Development Environment) go to Project–References and place a check mark on the following references:

- Agilent VISA COM Resource Manager 1.0
- VISA COM 1.0 Type Library

NOTE If you want to use VISA functions such as viWrite, then you must add the visa32.bas module to your Visual Basic project.

The signal generator's VXI-11 SCPI service must be on before you can run the Download Visual Basic 6.0 programming example.

NOTE To communicate with the signal generator over the LAN interface you must enable the VXI-11 SCPI service. For more information, refer to "Configuring the DHCP LAN" on page 25.

---

1. Visual Basic is a registered trademark of Microsoft corporation

You can start a new Standard EXE project and add the required references. Once the required references are included, you can copy the example programs into your project and add a command button to `Form1` that will call the program.

The example Visual Basic 6.0 programs are available on the signal generator Documentation CD-ROM, enabling you to cut and paste the examples into your project.

Visual Basic Examples

The Visual Basic examples enable the use of waveform files and are located in Chapter 5.

- "Creating I/Q Data—Little Endian Order" on page 272
- "Downloading I/Q Data" on page 275

HP Basic Examples

- "Interface Check using HP Basic and GPIB" on page 56
- "Local Lockout Using HP Basic and GPIB" on page 59
- "Queries Using HP Basic and GPIB" on page 62
- "Queries Using HP Basic and RS-232" on page 121

Running Java Examples

The Java program "Sockets LAN Programming Using Java" on page 115, connects to the signal generator via sockets LAN. This program requires Java version 1.1 or later be installed on your PC. For more information on sockets LAN programming with Java, refer to "Sockets LAN Programming Using Java" on page 115.

Running MATLAB Examples

For information regarding programming examples and files required to create and play waveform files, refer to Chapter 5.

NOTE To communicate with the signal generator over the LAN interface you must enable the VXI-11 SCPI service. For more information, refer to "Configuring the DHCP LAN" on page 25.

Running Perl Examples

The Perl example "Sockets LAN Programming Using PERL" on page 116, uses PERL script to control the signal generator over the sockets LAN interface.

Using GPIB

GPIB enables instruments to be connected together and controlled by a computer. GPIB and its associated interface operations are defined in the ANSI/IEEE Standard 488.1-1987 and ANSI/IEEE Standard 488.2-1992. See the IEEE website, *http://www.ieee.org*, for details on these standards.

The following sections contain information for installing a GPIB interface card or NI-GPIB interface card for your PC or UNIX-based system.

- "Installing the GPIB Interface Card" on page 50
- "Set Up the GPIB Interface" on page 18
- "Verify GPIB Functionality" on page 18

NOTE You can also connect GPIB instruments to a PC LAN port using the Agilent 82357A USB/GPIB Interface Converter, which eliminates the need for a GPIB card. For more information, go to *http://www.agilent.com/find/gpib*.

Installing the GPIB Interface Card

A GPIB interface card must be installed in the computer. Two common GPIB interface cards are the National Instruments (NI) PCI–GPIB card and the Agilent GPIB interface card. Follow the interface card instructions for installing and configuring the card. The following table provide lists on some of the available interface cards. Also, see the Agilent website, http://*www.agilent.com* for details on GPIB interface cards.

Interface Card	Operating System	IO Library	Languages	Backplane/ BUS	Max IO (kB/sec)	Buffering
Agilent GPIB Interface Card for PC-Based Systems						
Agilent 82341C for ISA bus computers	Windows[a] 95/98/NT /2000®	VISA / SICL	C/C++, Visual Basic, Agilent VEE, HP Basic for Windows	ISA/EISA, 16 bit	750	Built-in
Agilent 82341D Plug&Play for PC	Windows 95	VISA / SICL	C/C++, Visual Basic, Agilent VEE, HP Basic for Windows	ISA/EISA, 16 bit	750	Built-in
Agilent 82350A for PCI bus computers	Windows 95/98/NT /2000	VISA / SICL	C/C++, Visual Basic, Agilent VEE, HP Basic for Windows	PCI 32 bit	750	Built-in
Agilent 82350B for PCI bus computers	Windows 98(SE)/ME/2000 /XP	VISA / SICL	C/C++, Visual Basic, Agilent VEE, HP Basic for Windows	PCI 32 bit	> 900	Built-in

Interface Card	Operating System	IO Library	Languages	Backplane/ BUS	Max IO (kB/sec)	Buffering
NI-GPIB Interface Card for PC-Based Systems						
National Instruments PCI-GPIB	Windows 95/98/2000/ ME/NT	VISA NI-488.2™[b]	C/C++, Visual BASIC, LabView	PCI 32 bit	1.5 MBps	Built-in
National Instruments PCI-GPIB+	Windows NT	VISA NI-488.2	C/C++, Visual BASIC, LabView	PCI 32 bit	1.5 MBps	Built-in
Agilent-GPIB Interface Card for HP-UX Workstations						
Agilent E2071C	HP-UX 9.x, HP-UX 10.01	VISA/SICL	ANSI C, Agilent VEE, Agilent BASIC, HP-UX	EISA	750	Built-in
Agilent E2071D	HP-UX 10.20	VISA/SICL	ANSI C, Agilent VEE, Agilent BASIC, HP-UX	EISA	750	Built-in
Agilent E2078A	HP-UX 10.20	VISA/SICL	ANSI C, Agilent VEE, Agilent BASIC, HP-UX	PCI	750	Built-in

a. Windows 95, 98, NT, 2000 and XP are registered trademarks of Microsoft Corporation
b. NI-488.2 is a trademark of National Instruments Corporation

GPIB Programming Interface Examples

- "Interface Check using HP Basic and GPIB" on page 56
- "Interface Check Using NI-488.2 and C++" on page 57
- "Interface Check for GPIB Using VISA and C" on page 58
- "Local Lockout Using HP Basic and GPIB" on page 59
- "Local Lockout Using NI-488.2 and C++" on page 60
- "Queries Using HP Basic and GPIB" on page 62
- "Queries Using NI-488.2 and Visual C++" on page 63
- "Queries for GPIB Using VISA and C" on page 65
- "Generating a CW Signal Using VISA and C" on page 67
- "Generating an Externally Applied AC-Coupled FM Signal Using VISA and C" on page 69
- "Generating an Internal FM Signal Using VISA and C" on page 71
- "Generating a Step-Swept Signal Using VISA and C++" on page 73
- "Generating a Swept Signal Using VISA and Visual C++" on page 74
- "Saving and Recalling States Using VISA and C" on page 77
- "Reading the Data Questionable Status Register Using VISA and C" on page 79
- "Reading the Service Request Interrupt (SRQ) Using VISA and C" on page 83

Before Using the GPIB Examples

HP Basic addresses the signal generator at 719. The GPIB card is addressed at 7 and the signal generator at 19. The GPIB address designator for other libraries is typically GPIB0 or GPIB1.

GPIB Function Statements (Command Messages)

Function statements are the basis for GPIB programming and instrument control. These function statements, combined with SCPI, provide management and data communication for the GPIB interface and the signal generator.

This section describes functions used by different IO libraries. For more information, refer to the NI-488.2 Function Reference Manual for Windows, Agilent Standard Instrument Control Library reference manual, and Microsoft® Visual C++ 6.0[1] documentation.

---

1. Microsoft is a registered trademark of Microsoft Corporation.

Abort Function

The HP Basic function ABORT and the other listed IO library functions terminate listener/talker activity on the GPIB and prepare the signal generator to receive a new command from the computer. Typically, this is an initialization command used to place the GPIB in a known starting condition.

Library	Function Statement	Initialization Command
HP Basic	The ABORT function stops all GPIB activity.	10 ABORT 7
VISA Library	In VISA, the viTerminate command requests a VISA session to terminate normal execution of an asynchronous operation. The parameter list describes the session and job id.	viTerminate (parameter list)
NI-488.2	The NI-488.2 library function aborts any asynchronous read, write, or command operation that is in progress. The parameter ud is the interface or device descriptor.	ibstop(int ud)
SICL	The Agilent SICL function aborts any command currently executing with the session id. This function is supported with C/C++ on Windows 3.1 and Series 700 HP-UX.	iabort (id)

Remote Function

The HP Basic function REMOTE and the other listed IO library functions change the signal generator from local operation to remote operation. In remote operation, the front panel keys are disabled except for the **Local** key and the line power switch. Pressing the **Local** key restores manual operation.

Library	Function Statement	Initialization Command
HP Basic	The REMOTE 719 function disables the front panel operation of all keys with the exception of the **Local** key.	10 REMOTE 719
VISA Library	The VISA library, at this time, does not have a similar command.	N/A
NI-488.2	The NI-488.2 library function asserts the Remote Enable (REN) GPIB line. All devices listed in the parameter list are put into a listen-active state although no indication is generated by the signal generator. The parameter list describes the interface or device descriptor.	EnableRemote (parameter list)
SICL	The Agilent SICL function puts an instrument, identified by the id parameter, into remote mode and disables the front panel keys. Pressing the **Local** key on the signal generator front panel restores manual operation. The parameter id is the session identifier.	iremote (id)

Local Lockout Function

The HP Basic function LOCAL LOCKOUT and the other listed IO library functions disable the front panel keys including the **Local** key. With the **Local** key disabled, only the controller (or a hard reset of line power) can restore local control.

Library	Function Statement	Initialization Command
HP Basic	The LOCAL LOCKOUT function disables all front-panel signal generator keys. Return to local control can occur only by cycling power on the instrument, when the LOCAL command is sent or if the **Preset** key is pressed.	10 LOCAL LOCKOUT 719
VISA Library	The VISA library, at this time, does not have a similar command.	N/A
NI-488.2	The LOCAL LOCKOUT function disables all front-panel signal generator keys. Return to local control can occur only by cycling power on the instrument, when the LOCAL command is sent or if the **Preset** key is pressed.	SetRWLS (parameter list)
SICL	The Agilent SICL igpibllo prevents function prevents user access to front panel keys operation. The function puts an instrument, identified by the id parameter, into remote mode with local lockout. The parameter id is the session identifier and instrument address list.	igpibllo (id)

Local Function

The HP Basic function LOCAL and the other listed functions return the signal generator to local control with a fully enabled front panel.

Library	Function Statement	Initialization Command
HP Basic	The LOCAL 719 function returns the signal generator to manual operation, allowing access to the signal generator's front panel keys.	10 LOCAL 719
VISA Library	The VISA library, at this time, does not have a similar command.	N/A
NI-488.2	The NI-488.2 library function places the interface in local mode and allows operation of the signal generator's front panel keys. The ud parameter in the parameter list is the interface or device descriptor.	ibloc (int ud)
SICL	The Agilent SICL function puts the signal generator into Local operation; enabling front panel key operation. The id parameter identifies the session.	iloc(id)

Clear Function

The HP Basic function CLEAR and the other listed IO library functions clear the signal generator.

Library	Function Statement	Initialization Command
HP Basic	The CLEAR 719 function halts all pending output-parameter operations, resets the parser (interpreter of programming codes) and prepares for a new programming code, stops any sweep in progress, and turns off continuous sweep.	10 CLEAR 719
VISA Library	The VISA library uses the viClear function. This function performs an IEEE 488.1 clear of the signal generator.	viClear(ViSession vi)
NI-488.2	The NI-488.2 library function sends the GPIB Selected Device Clear (SDC) message to the device described by ud.	ibclr(int ud)
SICL	The Agilent SICL function clears a device or interface. The function also discards data in both the read and write formatted IO buffers. The id parameter identifies the session.	iclear (id)

Output Function

The HP Basic IO function OUTPUT and the other listed IO library functions put the signal generator into a listen mode and prepare it to receive ASCII data, typically SCPI commands.

Library	Function Statement	Initialization Command
HP Basic	The function OUTPUT 719 puts the signal generator into remote mode, makes it a listener, and prepares it to receive data.	10 OUTPUT 719
VISA Library	The VISA library uses the above function and associated parameter list to output data. This function formats according to the format string and sends data to the device. The parameter list describes the session id and data to send.	viPrintf(parameter list)
NI-488.2	The NI-488.2 library function addresses the GPIB and writes data to the signal generator. The parameter list includes the instrument address, session id, and the data to send.	ibwrt(parameter list)
SICL	The Agilent SICL function converts data using the format string. The format string specifies how the argument is converted before it is output. The function sends the characters in the format string directly to the instrument. The parameter list includes the instrument address, data buffer to write, and so forth.	iprintf (parameter list)

Enter Function

The HP Basic function ENTER reads formatted data from the signal generator. Other IO libraries use similar functions to read data from the signal generator.

Library	Function Statement	Initialization Command
HP Basic	The function ENTER 719 puts the signal generator into remote mode, makes it a talker, and assigns data or status information to a designated variable.	10 ENTER 719;
VISA Library	The VISA library uses the viScanf function and an associated parameter list to receive data. This function receives data from the instrument, formats it using the format string, and stores the data in the argument list. The parameter list includes the session id and string argument.	viScanf (parameter list)
NI-488.2	The NI-488.2 library function addresses the GPIB, reads data bytes from the signal generator, and stores the data into a specified buffer. The parameter list includes the instrument address and session id.	ibrd (parameter list)
SICL	The Agilent SICL function reads formatted data, converts it, and stores the results into the argument list. The conversion is done using conversion rules for the format string. The parameter list includes the instrument address, formatted data to read, and so forth.	iscanf (parameter list)

Interface Check using HP Basic and GPIB

This simple program causes the signal generator to perform an instrument reset. The SCPI command *RST places the signal generator into a pre-defined state and the remote annunciator (R) appears on the front panel display.

The following program example is available on the signal generator Documentation CD-ROM as basicex1.txt.

```
10    !******************************************************************************
20    !
30    !  PROGRAM NAME:         basicex1.txt
40    !
50    !  PROGRAM DESCRIPTION:  This program verifies that the GPIB connections and
60    !                        interface are functional.
70    !
80    !  Connect a controller to the signal generator using a GPIB cable.
90    !
100   !
110   !  CLEAR and RESET the controller and type in the following commands and then
120   !  RUN the program:
130   !
```

```
140   !***************************************************************************
150   !
160   Sig_gen=719    ! Declares a variable to hold the signal generator's address
170   LOCAL Sig_gen  ! Places the signal generator into Local mode
180   CLEAR Sig_gen  ! Clears any pending data I/O and resets the parser
190   REMOTE 719     ! Puts the signal generator into remote mode
200   CLEAR SCREEN   ! Clears the controllers display
210   REMOTE 719
220   OUTPUT Sig_gen;"*RST"  ! Places the signal generator into a defined state
230   PRINT "The signal generator should now be in REMOTE."
240   PRINT
250   PRINT "Verify that the remote [R] annunciator is on.  Press the `Local' key, "
260   PRINT "on the front panel to return the signal generator to local control."
270   PRINT
280   PRINT "Press RUN to start again."
290   END   ! Program ends
```

Interface Check Using NI-488.2 and C++

This example uses the NI-488.2 library to verify that the GPIB connections and interface are functional. Launch Microsoft Visual C++ 6.0, add the required files, and enter the following code into your .cpp source file.

The following program example is available on the signal generator Documentation CD-ROM as `niex1.cpp`.

```
// ****************************************************************************************
//
// PROGRAM NAME: niex1.cpp
//
// PROGRAM DESCRIPTION: This program verifies that the GPIB connections and
// interface are functional.
//
// Connect a GPIB cable from the PC GPIB card to the signal generator
// Enter the following code into the source .cpp file and execute the program
//
// ****************************************************************************************

#include "stdafx.h"
#include <iostream>
#include "windows.h"
#include "Decl-32.h"
using namespace std;

int GPIB0=   0;         // Board handle
```

```
Addr4882_t Address[31]; // Declares an array of type Addr4882_t

int main(void)

{
     int sig;                         // Declares a device descriptor variable
     sig = ibdev(0, 19, 0, 13, 1, 0); // Aquires a device descriptor
     ibclr(sig);                      // Sends device clear message to signal generator
     ibwrt(sig, "*RST", 4);           // Places the signal generator into a defined state

                                      // Print data to the output window
     cout << "The signal generator should now be in REMOTE. The remote indicator"<<endl;
     cout <<"annunciator R should appear on the signal generator display"<<endl;

  return 0;

}
```

Interface Check for GPIB Using VISA and C

This program uses VISA library functions and the C language to communicate with the signal generator. The program verifies that the GPIB connections and interface are functional. Launch Microsoft Visual C++ 6.0, add the required files, and enter the following code into your .cpp source file. `visaex1.cpp` performs the following functions:

- verifies the GPIB connections and interface are functional
- switches the signal generator into remote operation mode

The following program example is available on the signal generator Documentation CD-ROM as visaex1.cpp.

```
//****************************************************************************************
// PROGRAM NAME:visaex1.cpp
//
// PROGRAM DESCRIPTION:This example program verifies that the GPIB connections and
// and interface are functional.
// Turn signal generator power off then on and then run the program
//
//****************************************************************************************

#include <visa.h>
#include <stdio.h>
#include "StdAfx.h"
#include <stdlib.h>

void main ()
```

```
{
ViSession defaultRM, vi;        // Declares a variable of type ViSession
                                // for instrument communication
ViStatus viStatus = 0;
                                      // Opens a session to the GPIB device
                                      // at address 19
viStatus=viOpenDefaultRM(&defaultRM);
viStatus=viOpen(defaultRM, "GPIB::19::INSTR", VI_NULL, VI_NULL, &vi);
if(viStatus){
printf("Could not open ViSession!\n");
printf("Check instruments and connections\n");
printf("\n");
exit(0);}

viPrintf(vi, "*RST\n");         // initializes signal generator
                                // prints to the output window
printf("The signal generator should now be in REMOTE. The remote                        indicator\n");
printf("annunciator R should appear on the signal generator display\n");
printf("\n");

viClose(vi);                    // closes session
viClose(defaultRM);             // closes default session
}
```

Local Lockout Using HP Basic and GPIB

This example demonstrates the Local Lockout function. Local Lockout disables the front panel signal generator keys. basicex2.txt performs the following functions:

- resets instrument
- places signal generator into local
- places signal generator into remote

The following program example is available on the signal generator Documentation CD-ROM as basicex2.txt.

```
10    !************************************************************************
20    !
30    !  PROGRAM NAME:          basicex2.txt
40    !
50    !  PROGRAM DESCRIPTION:   In REMOTE mode, access to the signal generators
60    !                         functional front panel keys are disabled except for
70    !                         the Local and Contrast keys.  The LOCAL LOCKOUT
80    !                         command will disable the Local key.
90    !                         The LOCAL command, executed from the controller, is then
100   !                         the only way to return the signal generator to front panel,
```

```
110   !                        Local, control.
120   !**************************************************************************
130   Sig_gen=719       ! Declares a variable to hold signal generator address
140   CLEAR Sig_gen     ! Resets signal generator parser and clears any output
150   LOCAL Sig_gen     ! Places the signal generator in local mode
160   REMOTE Sig_gen    ! Places the signal generator in remote mode
170   CLEAR SCREEN      ! Clears the controllers display
180   OUTPUT Sig_gen;"*RST"       ! Places the signal generator in a defined state
190   ! The following print statements are user prompts
200   PRINT "The signal generator should now be in remote."
210   PRINT "Verify that the 'R' and 'L' annunciators are visable"
220   PRINT ".......... Press Continue"
230   PAUSE
240   LOCAL LOCKOUT 7   ! Puts the signal generator in LOCAL LOCKOUT mode
250   PRINT             ! Prints user prompt messages
260   PRINT "Signal generator should now be in LOCAL LOCKOUT mode."
270   PRINT
280   PRINT "Verify that all keys including `Local' (except Contrast keys) have no effect."
290   PRINT
300   PRINT ".......... Press Continue"
310   PAUSE
320   PRINT
330   LOCAL 7           ! Returns signal generator to Local control
340   ! The following print statements are user prompts
350   PRINT "Signal generator should now be in Local mode."
360   PRINT
370   PRINT "Verify that the signal generator's front-panel keyboard is functional."
380   PRINT
390   PRINT "To re-start this program press RUN."
400   END
```

Local Lockout Using NI-488.2 and C++

This example uses the NI-488.2 library to set the signal generator local lockout mode. Launch Microsoft Visual C++ 6.0, add the required files, and enter the following code into your .cpp source file. `niex2.cpp` performs the following functions:

- all front panel keys, except the contrast key
- places the signal generator into remote
- prompts the user to verify the signal generator is in remote
- places the signal generator into local

The following program example is available on the signal generator Documentation CD-ROM as `niex2.cpp`.

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

```
// PROGRAM NAME: niex2.cpp
//
// PROGRAM DESCRIPTION: This program will place the signal generator into
// LOCAL LOCKOUT mode. All front panel keys, except the Contrast key, will be disabled.
// The local command, 'ibloc(sig)' executed via program code, is the only way to
// return the signal generator to front panel, Local, control.
// ************************************************************************************

#include "stdafx.h"
#include <iostream>
#include "windows.h"
#include "Decl-32.h"
using namespace std;
int GPIB0=   0;                            // Board handle
Addr4882_t Address[31];                    // Declares a variable of type Addr4882_t

int main()

{
      int sig;                           // Declares variable to hold interface descriptor
      sig = ibdev(0, 19, 0, 13, 1, 0);   // Opens and initialize a device descriptor
      ibclr(sig);                        // Sends GPIB Selected Device Clear (SDC) message
      ibwrt(sig, "*RST", 4);             // Places signal generator in a defined state
      cout << "The signal generator should now be in REMOTE. The remote mode R "<<endl;
      cout <<"annunciator should appear on the signal generator display."<<endl;
      cout <<"Press Enter to continue"<<endl;
      cin.ignore(10000,'\n');
      SendIFC(GPIB0);                    // Resets the GPIB interface
      Address[0]=19;                     // Signal generator's address
      Address[1]=NOADDR;                 // Signifies end element in array. Defined in
                                         // DECL-32.H
      SetRWLS(GPIB0, Address);           // Places device in Remote with Lockout State.

      cout<< "The signal generator should now be in LOCAL LOCKOUT. Verify that all
             keys"<<endl;
      cout<< "including the 'Local' key are disabled (Contrast keys are not
              affected)"<<endl;
      cout <<"Press Enter to continue"<<endl;
      cin.ignore(10000,'\n');
      ibloc(sig);                        // Returns signal generator to local control
      cout<<endl;
      cout <<"The signal generator should now be in local mode\n";
 return 0;}
}
```

Queries Using HP Basic and GPIB

This example demonstrates signal generator query commands. The signal generator can be queried for conditions and setup parameters. Query commands are identified by the question mark as in the identify command *IDN? basicex3.txt performs the following functions:

- clears the signal generator
- queries the signal generator's settings

The following program example is available on the signal generator Documentation CD-ROM as basicex3.txt.

```
10    !***********************************************************************
20    !
30    !  PROGRAM NAME:         basicex3.txt
40    !
50    !  PROGRAM DESCRIPTION:  In this example, query commands are used with response
60    !                        data formats.
70    !
80    !  CLEAR and RESET the controller and RUN the following program:
90    !
100   !***********************************************************************
110   !
120   DIM A$[10],C$[100],D$[10]   ! Declares variables to hold string response data
130   INTEGER B                   ! Declares variable to hold integer response data
140   Sig_gen=719                 ! Declares variable to hold signal generator address
150   LOCAL Sig_gen               ! Puts signal generator in Local mode
160   CLEAR Sig_gen               ! Resets parser and clears any pending output
170   CLEAR SCREEN                ! Clears the controller's display
180   OUTPUT Sig_gen;"*RST"       ! Puts signal generator into a defined state
190   OUTPUT Sig_gen;"FREQ:CW?"   ! Querys the signal generator CW frequency setting
200   ENTER Sig_gen;F             ! Enter the CW frequency setting
210   ! Print frequency setting to the controller display
220   PRINT "Present source CW frequency is: ";F/1.E+6;"MHz"
230   PRINT
240   OUTPUT Sig_gen;"POW:AMPL?"  ! Querys the signal generator power level
250   ENTER Sig_gen;W             ! Enter the power level
260   ! Print power level to the controller display
270   PRINT "Current power setting is: ";W;"dBM"
280   PRINT
290   OUTPUT Sig_gen;"FREQ:MODE?" ! Querys the signal generator for frequency mode
300   ENTER Sig_gen;A$            ! Enter in the mode: CW, Fixed or List
310   ! Print frequency mode to the controller display
320   PRINT "Source's frequency mode is: ";A$
330   PRINT
```

```
340   OUTPUT Sig_gen;"OUTP OFF"   ! Turns signal generator RF state off
350   OUTPUT Sig_gen;"OUTP?"      ! Querys the operating state of the signal generator
360   ENTER Sig_gen;B             ! Enter in the state (0 for off)
370   ! Print the on/off state of the signal generator to the controller display
380   IF B>0 THEN
390     PRINT "Signal Generator output is: on"
400   ELSE
410     PRINT "Signal Generator output is: off"
420   END IF
430   OUTPUT Sig_gen;"*IDN?"      ! Querys for signal generator ID
440   ENTER Sig_gen;C$            ! Enter in the signal generator ID
450   ! Print the signal generator ID to the controller display
460   PRINT
470   PRINT "This signal generator is a ";C$
480   PRINT
490   ! The next command is a query for the signal generator's GPIB address
500   OUTPUT Sig_gen;"SYST:COMM:GPIB:ADDR?"
510   ENTER Sig_gen;D$            ! Enter in the signal generator's address
520   ! Print the signal generator's GPIB address to the controllers display
530   PRINT "The GPIB address is ";D$
540   PRINT
550   ! Print user prompts to the controller's display
560   PRINT "The signal generator is now under local control"
570   PRINT "or  Press RUN to start again."
580   END
```

Queries Using NI-488.2 and Visual C++

This example uses the NI-488.2 library to query different instrument states and conditions. Launch Microsoft Visual C++ 6.0, add the required files, and enter the following code into your .cpp source file. `niex3.cpp` performs the following functions:

- resets the signal generator
- queries the signal generator for various settings
- reads the various settings

The following program example is available on the signal generator Documentation CD-ROM as `niex3.cpp`.

```
//****************************************************************************************
// PROGRAM NAME: niex3.cpp
//
// PROGRAM DESCRIPTION: This example demonstrates the use of query commands.
//
// The signal generator can be queried for conditions and instrument states.
// These commands are of the type "*IDN?" where the question mark indicates
```

```
// a query.
//
//************************************************************************************

#include "stdafx.h"
#include <iostream>
#include "windows.h"
#include "Decl-32.h"
using namespace std;
int GPIB0=   0;                          // Board handle
Addr4882_t Address[31];                  // Declare a variable of type Addr4882_t

int main()

{
  int sig;                           // Declares variable to hold interface descriptor
  int num;
  char rdVal[100];                   // Declares variable to read instrument responses
  sig = ibdev(0, 19, 0, 13, 1, 0); // Open and initialize a device descriptor
  ibloc(sig);                        // Places the signal generator in local mode
  ibclr(sig);                        // Sends Selected Device Clear(SDC) message
  ibwrt(sig, "*RST", 4);             // Places signal generator in a defined state
  ibwrt(sig, ":FREQuency:CW?",14); // Querys the CW frequency
  ibrd(sig, rdVal,100);              // Reads in the response into rdVal
  rdVal[ibcntl] = '\0';              // Null character indicating end of array
  cout<<"Source CW frequency is "<<rdVal;   // Print frequency of signal generator
  cout<<"Press any key to continue"<<endl;
  cin.ignore(10000,'\n');
  ibwrt(sig, "POW:AMPL?",10);        // Querys the signal generator
  ibrd(sig, rdVal,100);              // Reads the signal generator power level
  rdVal[ibcntl] = '\0';              // Null character indicating end of array
                                   // Prints signal generator power level
  cout<<"Source power (dBm) is : "<<rdVal;
  cout<<"Press any key to continue"<<endl;
  cin.ignore(10000,'\n');
  ibwrt(sig, ":FREQ:MODE?",11);      // Querys source frequency mode
  ibrd(sig, rdVal,100);              // Enters in the source frequency mode
  rdVal[ibcntl] = '\0';              // Null character indicating end of array
  cout<<"Source frequency mode is "<<rdVal; // Print source frequency mode
  cout<<"Press any key to continue"<<endl;
  cin.ignore(10000,'\n');
  ibwrt(sig, "OUTP OFF",12);         // Turns off RF source
```

```
  ibwrt(sig, "OUTP?",5);           // Querys the on/off state of the instrument
  ibrd(sig,rdVal,2);               // Enter in the source state
  rdVal[ibcntl] = '\0';
  num = (int (rdVal[0]) -('0'));
  if (num > 0){
      cout<<"Source RF state is : On"<<endl;
  }else{
      cout<<"Source RF state is : Off"<<endl;}
  cout<<endl;
  ibwrt(sig, "*IDN?",5);           // Querys the instrument ID
  ibrd(sig, rdVal,100);            // Reads the source ID
  rdVal[ibcntl] = '\0';            // Null character indicating end of array
  cout<<"Source ID is : "<<rdVal;  // Prints the source ID
  cout<<"Press any key to continue"<<endl;
  cin.ignore(10000,'\n');
  ibwrt(sig, "SYST:COMM:GPIB:ADDR?",20); //Querys source address
  ibrd(sig, rdVal,100);            // Reads the source address
  rdVal[ibcntl] = '\0';            // Null character indicates end of array
                                 // Prints the signal generator address
  cout<<"Source GPIB address is : "<<rdVal;
  cout<<endl;
  cout<<"Press the 'Local' key to return the signal generator to LOCAL control"<<endl;      cout<<endl;
return 0;
}
```

Queries for GPIB Using VISA and C

This example uses VISA library functions to query different instrument states and conditions. Launch Microsoft Visual C++ 6.0, add the required files, and enter the following code into your .cpp source file. visaex3.cpp performs the following functions:

- verifies the GPIB connections and interface are functional
- resets the signal generator
- queries the instrument (CW frequency, power level, frequency mode, and RF state)
- reads responses into the rdBuffer (CW frequency, power level, and frequency mode)
- turns signal generator RF state off
- verifies RF state off

The following program example is available on the signal generator Documentation CD-ROM as visaex3.cpp.

```
//****************************************************************************************
// PROGRAM FILE NAME:visaex3.cpp
//
// PROGRAM DESCRIPTION:This example demonstrates the use of query commands. The signal
// generator can be queried for conditions and instrument states. These commands are of
// the type "*IDN?"; the question mark indicates a query.
```

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

#include <visa.h>
#include "StdAfx.h"
#include <iostream>
#include <conio.h>
#include <stdlib.h>
using namespace std;


void main ()
{
ViSession defaultRM, vi;    // Declares variables of type ViSession
                            // for instrument communication
ViStatus viStatus = 0;      // Declares a variable of type ViStatus
                            // for GPIB verifications
char rdBuffer [256];        // Declares variable to hold string data
int num;                    // Declares variable to hold integer data
                            // Initialize the VISA system
viStatus=viOpenDefaultRM(&defaultRM);
                            // Open session to GPIB device at address 19
viStatus=viOpen(defaultRM, "GPIB::19::INSTR", VI_NULL, VI_NULL, &vi);
if(viStatus){               // If problems, then prompt user
    printf("Could not open ViSession!\n");
    printf("Check instruments and connections\n");
    printf("\n");
    exit(0);}
viPrintf(vi, "*RST\n");         // Resets signal generator
viPrintf(vi, "FREQ:CW?\n");     // Querys the CW frequency
viScanf(vi, "%t", rdBuffer);    // Reads response into rdBuffer
                                // Prints the source frequency
printf("Source CW frequency is : %s\n", rdBuffer);
printf("Press any key to continue\n");
printf("\n");                   // Prints new line character to the display
getch();
viPrintf(vi, "POW:AMPL?\n");    // Querys the power level
viScanf(vi, "%t", rdBuffer);    // Reads the response into rdBuffer
                                // Prints the source power level
printf("Source power (dBm) is : %s\n", rdBuffer);
printf("Press any key to continue\n");
printf("\n");                   // Prints new line character to the display
```

```
getch();
viPrintf(vi, "FREQ:MODE?\n");   // Querys the frequency mode
viScanf(vi, "%t", rdBuffer);    // Reads the response into rdBuffer
                                // Prints the source freq mode
printf("Source frequency mode is : %s\n", rdBuffer);
printf("Press any key to continue\n");
printf("\n");                   // Prints new line character to the display
getch();
viPrintf(vi, "OUTP OFF\n");     // Turns source RF state off
viPrintf(vi, "OUTP?\n");        // Querys the signal generator's RF state
viScanf(vi, "%1i", &num);       // Reads the response (integer value)
                                // Prints the on/off RF state
 if (num > 0 ) {
printf("Source RF state is : on\n");
}else{
printf("Source RF state is : off\n");
}
                                  // Close the sessions
viClose(vi);
viClose(defaultRM);
}
```

Generating a CW Signal Using VISA and C

This example uses VISA library functions to control the signal generator. The signal generator is set for a CW frequency of 500 kHz and a power level of −2.3 dBm. Launch Microsoft Visual C++ 6.0, add the required files, and enter the code into your .cpp source file. `visaex4.cpp` performs the following functions:

- verifies the GPIB connections and interface are functional
- resets the signal generator
- queries the instrument (CW frequency, power level, frequency mode, and RF state)
- reads responses into the rdBuffer (CW frequency, power level, and frequency mode)
- turns signal generator RF state off
- verifies RF state off

The following program example is available on the signal generator Documentation CD-ROM as visaex4.cpp.

```
//****************************************************************************************
// PROGRAM FILE NAME:   visaex4.cpp
//
// PROGRAM DESCRIPTION: This example demonstrates query commands. The signal generator
// frequency and power level.
// The RF state of the signal generator is turn on and then the state is queried. The
// response will indicate that the RF state is on. The RF state is then turned off and
// queried. The response should indicate that the RF state is off. The query results are
```

```
// printed to the to the display window.
//
//****************************************************************************************

#include "StdAfx.h"
#include <visa.h>
#include <iostream>
#include <stdlib.h>
#include <conio.h>

void main ()
{
 ViSession   defaultRM, vi;      // Declares variables of type ViSession
                                // for instrument communication
ViStatus viStatus = 0;          // Declares a variable of type ViStatus
                                // for GPIB verifications
char rdBuffer [256];            // Declare variable to hold string data
int num;                        // Declare variable to hold integer data

viStatus=viOpenDefaultRM(&defaultRM);     // Initialize VISA system
                               // Open session to GPIB device at address 19
viStatus=viOpen(defaultRM, "GPIB::19::INSTR", VI_NULL, VI_NULL, &vi);
if(viStatus){                   // If problems then prompt user
printf("Could not open ViSession!\n");
printf("Check instruments and connections\n");
printf("\n");
exit(0);}

viPrintf(vi, "*RST\n");         // Reset the signal generator
viPrintf(vi, "FREQ 500 kHz\n"); // Set the source CW frequency for 500 kHz
viPrintf(vi, "FREQ:CW?\n");     // Query the CW frequency
viScanf(vi, "%t", rdBuffer);    // Read signal generator response
printf("Source CW frequency is : %s\n", rdBuffer);  // Print the frequency
viPrintf(vi, "POW:AMPL -2.3 dBm\n");  // Set the power level to -2.3 dBm
viPrintf(vi, "POW:AMPL?\n");    // Query the power level
viScanf(vi, "%t", rdBuffer);    // Read the response into rdBuffer
printf("Source power (dBm) is : %s\n", rdBuffer); // Print the power level
viPrintf(vi, "OUTP:STAT ON\n"); // Turn source RF state on
viPrintf(vi, "OUTP?\n");        // Query the signal generator's RF state
viScanf(vi, "%1i", &num);       // Read the response (integer value)
  // Print the on/off RF state
if (num > 0 ) {
```

```
printf("Source RF state is : on\n");
}else{
printf("Source RF state is : off\n");
}
printf("\n");
printf("Verify RF state then press continue\n");
printf("\n");
getch();
viClear(vi);
viPrintf(vi,"OUTP:STAT OFF\n"); // Turn source RF state off
viPrintf(vi, "OUTP?\n");        // Query the signal generator's RF state
viScanf(vi, "%1i", &num);       // Read the response
  // Print the on/off RF state
 if (num > 0 ) {
printf("Source RF state is now: on\n");
}else{
printf("Source RF state is now: off\n");
}
                                     // Close the sessions
printf("\n");
viClear(vi);
viClose(vi);
viClose(defaultRM);
}
```

Generating an Externally Applied AC-Coupled FM Signal Using VISA and C

In this example, the VISA library is used to generate an ac-coupled FM signal at a carrier frequency of 700 MHz, a power level of –2.5 dBm, and a deviation of 20 kHz. Before running the program:

- Connect the output of a modulating signal source to the signal generator’s EXT 2 input connector.
- Set the modulation signal source for the desired FM characteristics.

Launch Microsoft Visual C++ 6.0, add the required files, and enter the code into your .cpp source file. `visaex5.cpp` performs the following functions:

- error checking
- resets the signal generator
- sets up the EXT 2 connector on the signal generator for FM
- sets up FM path 2 coupling to AC
- sets up FM path 2 deviation to 20 kHz
- sets carrier frequency to 700 MHz
- sets the power level to -2.5 dBm
- turns on frequency modulation and RF output

The following program example is available on the signal generator Documentation CD-ROM as visaex5.cpp.

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

```
// PROGRAM FILE NAME:visaex5.cpp
//
// PROGRAM DESCRIPTION:This example sets the signal generator FM source to External 2,
// coupling to AC, deviation to 20 kHZ, carrier frequency to 700 MHz and the power level
// to -2.5 dBm. The RF state is set to on.
//
//****************************************************************************************

#include <visa.h>
#include "StdAfx.h"
#include <iostream>
#include <stdlib.h>
#include <conio.h>

void main ()
{
 ViSession defaultRM, vi;           // Declares variables of type ViSession
                                   // for instrument communication
ViStatus viStatus = 0;             // Declares a variable of type ViStatus
                                   // for GPIB verifications
                                   // Initialize VISA session
viStatus=viOpenDefaultRM(&defaultRM);
                               // open session to gpib device at address 19
viStatus=viOpen(defaultRM, "GPIB::19::INSTR", VI_NULL, VI_NULL, &vi);
if(viStatus){                        // If problems, then prompt user
    printf("Could not open ViSession!\n");
    printf("Check instruments and connections\n");
    printf("\n");
    exit(0);}

printf("Example program to set up the signal generator\n");
printf("for an AC-coupled FM signal\n");
printf("Press any key to continue\n");
printf("\n");
getch();
printf("\n");

viPrintf(vi, "*RST\n");              // Resets the signal generator
viPrintf(vi, "FM:SOUR EXT2\n");      // Sets EXT 2 source for FM
viPrintf(vi, "FM:EXT2:COUP AC\n");   // Sets FM path 2 coupling to AC
viPrintf(vi, "FM:DEV 20 kHz\n");     // Sets FM path 2 deviation to 20 kHz
viPrintf(vi, "FREQ 700 MHz\n");      // Sets carrier frequency to 700 MHz
```

```
viPrintf(vi, "POW:AMPL -2.5 dBm\n"); // Sets the power level to -2.5 dBm
viPrintf(vi, "FM:STAT ON\n");        // Turns on frequency modulation
viPrintf(vi, "OUTP:STAT ON\n");      // Turns on RF output
                                     // Print user information
printf("Power level : -2.5 dBm\n");
printf("FM state : on\n");
printf("RF output : on\n");
printf("Carrier Frequency : 700 MHZ\n");
printf("Deviation : 20 kHZ\n");
printf("EXT2 and AC coupling are selected\n");
printf("\n");                        // Prints a carrage return
                                     // Close the sessions
viClose(vi);
viClose(defaultRM);
}
```

Generating an Internal FM Signal Using VISA and C

In this example the VISA library is used to generate an internal FM signal at a carrier frequency of 900 MHz and a power level of –15 dBm. The FM rate will be 5 kHz and the peak deviation will be 100 kHz. Launch Microsoft Visual C++ 6.0, add the required files, and enter the following code into your .cpp source file. `visaex6.cpp` performs the following functions:

- error checking
- resets the signal generator
- sets up the signal generator for FM path 2 and internal FM rate of 5 kHz
- sets up FM path 2 deviation to 100 kHz
- sets carrier frequency to 900 MHz
- sets the power level to -15 dBm
- turns on frequency modulation and RF output

The following program example is available on the signal generator Documentation CD-ROM as visaex6.cpp.

```
//****************************************************************************************
// PROGRAM FILE NAME:visaex6.cpp
//
// PROGRAM DESCRIPION:This example generates an internal FM signal at a 900
// MHz carrier frequency and a power level of -15 dBm. The FM rate is 5 kHz and the peak
// deviation 100 kHz
//
//****************************************************************************************

#include <visa.h>
#include "StdAfx.h"
#include <iostream>
#include <stdlib.h>
```

```
#include <conio.h>

void main ()
{
ViSession defaultRM, vi;           // Declares variables of type ViSession
                                   // for instrument communication
ViStatus viStatus = 0;             // Declares a variable of type ViStatus
                                   // for GPIB verifications

viStatus=viOpenDefaultRM(&defaultRM); // Initialize VISA session
                              // open session to gpib device at address 19
viStatus=viOpen(defaultRM, "GPIB::19::INSTR", VI_NULL, VI_NULL, &vi);
if(viStatus){                         // If problems, then prompt user
printf("Could not open ViSession!\n");
printf("Check instruments and connections\n");
printf("\n");
exit(0);}

printf("Example program to set up the signal generator\n");
printf("for an AC-coupled FM signal\n");
printf("\n");
printf("Press any key to continue\n");
getch();
viClear(vi);                          // Clears the signal generator
viPrintf(vi, "*RST\n");               // Resets the signal generator
viPrintf(vi, "FM2:INT:FREQ 5 kHz\n"); // Sets FM path 2 to internal at a modulation rate of 5 kHz
viPrintf(vi, "FM2:DEV 100 kHz\n");    // Sets FM path 2 modulation deviation rate of 100 kHz
viPrintf(vi, "FREQ 900 MHz\n");       // Sets carrier frequency to 900 MHz
viPrintf(vi, "POW -15 dBm\n");        // Sets the power level to -15 dBm
viPrintf(vi, "FM2:STAT ON\n");        // Turns on frequency modulation
viPrintf(vi, "OUTP:STAT ON\n");       // Turns on RF output
printf("\n");                         // Prints a carriage return
                                       // Print user information
printf("Power level : -15 dBm\n");
printf("FM state : on\n");
printf("RF output : on\n");
printf("Carrier Frequency : 900 MHZ\n");
printf("Deviation : 100 kHZ\n");
printf("Internal modulation : 5 kHz\n");
printf("\n");                        // Print a carrage return
                    // Close the sessions
viClose(vi);
```

```
viClose(defaultRM);
}
```

Generating a Step-Swept Signal Using VISA and C++

In this example the VISA library is used to set the signal generator for a continuous step sweep on a defined set of points from 500 MHz to 800 MHz. The number of steps is set for 10 and the dwell time at each step is set to 500 ms. The signal generator will then be set to local mode which allows the user to make adjustments from the front panel. Launch Microsoft Visual C++ 6.0, add the required files, and enter the following code into your .cpp source file. `visaex7.cpp` performs the following functions:

- clears and resets the signal generator
- sets up the instrument for continuous step sweep
- sets up the start and stop sweep frequencies
- sets up the number of steps
- sets the power level
- turns on the RF output

The following program example is available on the signal generator Documentation CD-ROM as `visaex7.cpp`.

```
//****************************************************************************************
// PROGRAM FILE NAME:visaex7.cpp
//
// PROGRAM DESCRIPTION:This example will program the signal generator to perform a step
// sweep from 500-800 MHz with a .5 sec dwell at each frequency step.
//
//****************************************************************************************

#include <visa.h>
#include "StdAfx.h"
#include <iostream>

void main ()
{
ViSession defaultRM, vi;// Declares variables of type ViSession
// vi establishes instrument communication
ViStatus viStatus = 0;// Declares a variable of type ViStatus
                        // for GPIB verifications

viStatus=viOpenDefaultRM(&defaultRM); // Initialize VISA session
             // Open session to GPIB device at address 19
viStatus=viOpen(defaultRM, "GPIB::19::INSTR", VI_NULL, VI_NULL, &vi);
if(viStatus){// If problems, then prompt user
```

```
printf("Could not open ViSession!\n");
printf("Check instruments and connections\n");
printf("\n");
exit(0);}

viClear(vi);                          // Clears the signal generator
viPrintf(vi, "*RST\n");               // Resets the signal generator
viPrintf(vi, "*CLS\n");               // Clears the status byte register
viPrintf(vi, "FREQ:MODE LIST\n");     // Sets the sig gen freq mode to list
viPrintf(vi, "LIST:TYPE STEP\n");     // Sets sig gen LIST type to step
viPrintf(vi, "FREQ:STAR 500 MHz\n");  // Sets start frequency
viPrintf(vi, "FREQ:STOP 800 MHz\n");  // Sets stop frequency
viPrintf(vi, "SWE:POIN 10\n");        // Sets number of steps (30 mHz/step)
viPrintf(vi, "SWE:DWEL .5 S\n");      // Sets dwell time to 500 ms/step
viPrintf(vi, "POW:AMPL -5 dBm\n");    // Sets the power level for -5 dBm
viPrintf(vi, "OUTP:STAT ON\n");       // Turns RF output on
viPrintf(vi, "INIT:CONT ON\n");       // Begins the step sweep operation
                                      // Print user information
printf("The signal generator is in step sweep mode. The frequency range                          is\n");
printf("500 to 800 mHz. There is a .5 sec dwell time at each 30 mHz                           step.\n");
printf("\n");                         // Prints a carriage return/line feed
 viPrintf(vi, "OUTP:STAT OFF\n");      // Turns the RF output off
printf("Press the front panel Local key to return the\n");
printf("signal generator to manual operation.\n");
                                      // Closes the sessions
printf("\n");
viClose(vi);
viClose(defaultRM);
}
```

Generating a Swept Signal Using VISA and Visual C++

This example sets up the signal generator for a frequency sweep from 1 to 2 GHz with 101 points and a .01 second dwell period for each point. A loop is used to generator 5 sweep operations. The signal generator triggers each sweep with the `:INIT` command. There is a wait introduced in the loop to allow the signal generator to complete all operations such as set up and retrace before the next sweep is generated. `visaex11.cpp` performs the following functions:

- sets up the signal generator for a 1 to 2 GHz frequency sweep
- sets up the signal generator to have a dwell time of .01 seconds and 101 points in the sweep
- sleep function is used to allow the instrument to complete its sweep operation

The following program example is available on the signal generator Documentation CD-ROM as visaex11.cpp.

```
//****************************************************************************
// PROGRAM FILE NAME: visaex11.cpp
//
// PROGRAM DESCRIPTION: This program sets up the signal generator to
// sweep from 1-2 GHz. A loop and counter are used to generate 5 sweeps.
// Each sweep consists of 101 points with a .01 second dwell at each point.
//
// The program uses a Sleep function to allow the signal generator to
// complete it's sweep operation before the INIT command is sent.
// The Sleep function is available with the windows.h header file which is
// included in the project.
//
// NOTE: Change the TCPIP0 address in the instOpenString declaration to
// match the IP address of your signal generator.
//
//****************************************************************************

#include "stdafx.h"
#include "visa.h"
#include <iostream>
#include <windows.h>

void main ()
 {
     ViStatus stat;
     ViSession defaultRM,inst;

     int npoints = 101;
     double dwell = 0.01;
     int intCounter=5;

     char* instOpenString = "TCPIP0::141.121.93.101::INSTR";

     stat = viOpenDefaultRM(&defaultRM);
     stat = viOpen(defaultRM,instOpenString,VI_NULL,VI_NULL, &inst);
     // preset to start clean

     stat = viPrintf( inst, "*RST\n" );
     // set power level for -10dBm
     stat = viPrintf(inst, "POW -10DBM\n");
```

```
    // set the start and stop frequency for the sweep
    stat = viPrintf(inst, "FREQ:START 1GHZ\n");
    stat = viPrintf(inst, "FREQ:STOP 2GHZ\n");
    // setup dwell per point
    stat = viPrintf(inst, "SWEEP:DWELL %e\n", dwell);
    // setup number of points
    stat = viPrintf(inst, "SWEEP:POINTS %d\n", npoints);

    // set interface timeout to double the expected sweep time
    // sweep takes (~15ms + dwell) per point * number of points
    // the timeout should not be shorter then the sweep, set it
    // longer
    long timeoutMS = long(2*npoints*(.015+dwell)*1000);
    // set the VISA timeout
    stat = viSetAttribute(inst, VI_ATTR_TMO_VALUE, timeoutMS);

    // set continuous trigger mode off
    stat = viPrintf(inst, "INIT:CONT OFF\n");
    // turn list sweep on
    stat = viPrintf(inst, "FREQ:MODE LIST\n");

    int sweepNo = 0;
    while(intCounter>0 )
    {
        // start the sweep (initialize)
        stat = viPrintf(inst, "INIT\n");
        printf("Sweep %d started\n",++sweepNo);
        // wait for the sweep completion with *OPC?
        int res ;
        stat = viPrintf(inst, "*OPC?\n");
        stat = viScanf(inst, "%d", &res);
        // handle possible errors here (most likely a timeout)
        // err_handler( inst, stat );
        puts("Sweep ended");
        // delay before sending next INIT since instrument
        // may not be ready to receive it yet
        Sleep(15);

intCounter = intCounter-1;

    }
printf("End of Program\n\n");
```

```
}
```

Saving and Recalling States Using VISA and C

In this example, instrument settings are saved in the signal generator's save register. These settings can then be recalled separately; either from the keyboard or from the signal generator's front panel. Launch Microsoft Visual C++ 6.0, add the required files, and enter the following code into your .cpp source file. `visaex8.cpp` performs the following functions:

- error checking
- clears the signal generator
- resets the status byte register
- resets the signal generator
- sets up the signal generator frequency, ALC off, power level, RF output on
- checks for operation complete
- saves to settings to instrument register number one
- recalls information from register number one
- prompts user input to put instrument into Local and checks for operation complete

The following program example is available on the signal generator Documentation CD-ROM as visaex8.cpp.

```
//****************************************************************************************
// PROGRAM FILE NAME:visaex8.cpp
//
// PROGRAM DESCRIPTION:In this example, instrument settings are saved in the signal
// generator's registers and then recalled.
// Instrument settings can be recalled from the keyboard or, when the signal generator
// is put into Local control, from the front panel.
// This program will initialize the signal generator for an instrument state, store the
// state to register #1. An *RST command will reset the signal generator and a *RCL
// command will return it to the stored state. Following this remote operation the user
// will be instructed to place the signal generator in Local mode.
//
//****************************************************************************************

#include <visa.h>
#include "StdAfx.h"
#include <iostream>
#include <conio.h>

void main ()
{
```

```
 ViSession defaultRM, vi;// Declares variables of type ViSession
// for instrument communication
ViStatus viStatus = 0;// Declares a variable of type ViStatus
                          // for GPIB verifications
long lngDone = 0;         // Operation complete flag

viStatus=viOpenDefaultRM(&defaultRM);     // Initialize VISA session
// Open session to gpib device at address 19
viStatus=viOpen(defaultRM, "GPIB::19::INSTR", VI_NULL, VI_NULL, &vi);
if(viStatus){// If problems, then prompt user
    printf("Could not open ViSession!\n");
    printf("Check instruments and connections\n");
    printf("\n");
    exit(0);}
printf("\n");
viClear(vi);                          // Clears the signal generator
viPrintf(vi, "*CLS\n");               // Resets the status byte register
                                      // Print user information
printf("Programming example using the  *SAV,*RCL   SCPI commands\n");
printf("used to save and recall an instrument's state\n");
printf("\n");
viPrintf(vi, "*RST\n");               // Resets the signal generator
viPrintf(vi, "FREQ 5 MHz\n");         // Sets sig gen frequency
viPrintf(vi, "POW:ALC OFF\n");        // Turns ALC Off
viPrintf(vi, "POW:AMPL -3.2 dBm\n");  // Sets power for -3.2 dBm
viPrintf(vi, "OUTP:STAT ON\n");       // Turns RF output On
viPrintf(vi, "*OPC?\n");              // Checks for operation complete
while (!lngDone)
    viScanf (vi ,"%d",&lngDone);      // Waits for setup to complete
viPrintf(vi, "*SAV 1\n");             // Saves sig gen state to register #1
                                      // Print user information
printf("The current signal generator operating state will be saved\n");
printf("to Register #1. Observe the state then press Enter\n");
printf("\n");                         // Prints new line character
getch();                              // Wait for user input
lngDone=0;                            // Resets the operation complete flag
viPrintf(vi, "*RST\n");               // Resets the signal generator
viPrintf(vi, "*OPC?\n");              // Checks for operation complete
while (!lngDone)
    viScanf (vi ,"%d",&lngDone);      // Waits for setup to complete
                                      // Print user infromation
printf("The instrument is now in it's Reset operating state. Press the\n");
```

```
printf("Enter key to return the signal generator to the Register #1                         state\n");
printf("\n");                        // Prints new line character
getch();                             // Waits for user input
lngDone=0;                           // Reset the operation complete flag
viPrintf(vi, "*RCL 1\n");            // Recalls stored register #1 state
viPrintf(vi, "*OPC?\n");             // Checks for operation complete
while (!lngDone)
    viScanf (vi ,"%d",&lngDone);     // Waits for setup to complete
                                     // Print user information
printf("The signal generator has been returned to it's Register #1                         state\n");
printf("Press Enter to continue\n");
printf("\n");                        // Prints new line character
getch();                             // Waits for user input
lngDone=0;                           // Reset the operation complete flag
viPrintf(vi, "*RST\n");              // Resets the signal generator
viPrintf(vi, "*OPC?\n");             // Checks for operation complete
while (!lngDone)
    viScanf (vi ,"%d",&lngDone);     // Waits for setup to complete
                                     // Print user information
printf("Press Local on instrument front panel to return to manual mode\n");
printf("\n");                        // Prints new line character
                                     // Close the sessions
viClose(vi);
viClose(defaultRM);
}
```

Reading the Data Questionable Status Register Using VISA and C

In this example, the signal generator’s data questionable status register is read. You will be asked to set up the signal generator for error generating conditions. The data questionable status register will be read and the program will notify the user of the error condition that the setup caused. Follow the user prompts presented when the program runs. Launch Microsoft Visual C++ 6.0, add the required files, and enter the following code into your .cpp source file. `visaex9.cpp` performs the following functions:

- error checking
- clears the signal generator
- resets the signal generator
- the data questionable status register is enabled to read an unleveled condition
- prompts user to manually set up the signal generator for an unleveled condition
- queries the data questionable status register for any set bits and converts the string data to numeric
- based on the numeric value, program checks for a corresponding status check value
- similarly checks for over or undermodulation condition

The following program example is available on the signal generator Documentation CD-ROM as `visaex9.cpp`.

```
//****************************************************************************************
// PROGRAM NAME:visaex9.cpp
//
// PROGRAM DESCRIPTION:In this example, the data questionable status register is read.
// The data questionable status register is enabled to read an unleveled condition.
// The signal generator is then set up for an unleveled condition and the data
// questionable status register read. The results are then displayed to the user.
// The status questionable register is then setup to monitor a modulation error condition.
// The signal generator is set up for a modulation error condition and the data
// questionable status register is read.
// The results are displayed to the active window.
//
//****************************************************************************************

#include <visa.h>
#include "StdAfx.h"
#include <iostream>
#include <conio.h>

void main ()
{
ViSession defaultRM, vi;// Declares a variables of type ViSession
                        // for instrument communication
ViStatus viStatus = 0;// Declares a variable of type ViStatus
// for GPIB verifications
int num=0;// Declares a variable for switch statements

char rdBuffer[256]={0};         // Declare a variable for response data

viStatus=viOpenDefaultRM(&defaultRM);     // Initialize VISA session
                             // Open session to GPIB device at address 19

viStatus=viOpen(defaultRM, "GPIB::19::INSTR", VI_NULL, VI_NULL, &vi);
if(viStatus){                   // If problems, then prompt user
printf("Could not open ViSession!\n");
printf("Check instruments and connections\n");
printf("\n");
exit(0);}
printf("\n");
viClear(vi);// Clears the signal generator
// Prints user information
printf("Programming example to demonstrate reading the signal generator's
                         Status Byte\n");
```

```
printf("\n");
printf("Manually set up the sig gen for an unleveled output condition:\n");
printf("* Set signal generator output amplitude to +20 dBm\n");
printf("* Set frequency to maximum value\n");
printf("* Turn On signal generator's RF Output\n");
printf("* Check signal generator's display for the UNLEVEL annunciator\n");
printf("\n");
printf("Press Enter when ready\n");
printf("\n");
getch();                                  // Waits for keyboard user input
viPrintf(vi, "STAT:QUES:POW:ENAB 2\n");   // Enables the Data Questionable
                                          // Power Condition Register Bits
         // Bits '0' and '1'
viPrintf(vi, "STAT:QUES:POW:COND?\n");    // Querys the register for any
         // set bits
viScanf(vi, "%s", rdBuffer);              // Reads the decimal sum of the
         // set bits
num=(int (rdBuffer[1]) -('0'));           // Converts string data to
         // numeric

switch (num)                              // Based on the decimal value
{
    case 1:
printf("Signal Generator Reverse Power Protection                                          Tripped\n");
printf("/n");
break;
    case 2:
printf("Signal Generator Power is Unleveled\n");
printf("\n");
break;
    default:
printf("No Power Unleveled condition detected\n");
printf("\n");
}
viClear(vi);                              // Clears the signal generator
                                          // Prints user information
printf("--------------------------------------------------------------\n");
printf("\n");
printf("Manually set up the sig gen for an unleveled output condition:\n");
printf("\n");
printf("* Select AM modulation\n");
printf("* Select AM Source Ext 1 and Ext Coupling AC\n");
```

```
printf("* Turn On the modulation.\n");
printf("* Do not connect any source to the input\n");
printf("* Check signal generator's display for the EXT1 LO annunciator\n");
printf("\n");
printf("Press Enter when ready\n");
printf("\n");
getch();                                  // Waits for keyboard user input
viPrintf(vi, "STAT:QUES:MOD:ENAB 16\n");  // Enables the Data Questionable
                                          // Modulation Condition Register
         // bits '0','1','2','3' and  '4'
 viPrintf(vi, "STAT:QUES:MOD:COND?\n");    // Querys the register for any
         // set bits
  viScanf(vi, "%s", rdBuffer);              // Reads the decimal sum of the
         // set bits
num=(int (rdBuffer[1]) -('0')); // Converts string data to numeric

switch (num)                              // Based on the decimal value
{
    case 1:
printf("Signal Generator Modulation 1 Undermod\n");
printf("\n");
break;
    case 2:
printf("Signal Generator Modulation 1 Overmod\n");
printf("\n");
break;
    case 4:
printf("Signal Generator Modulation 2 Undermod\n");
printf("\n");
break;
    case 8:
printf("Signal Generator Modulation 2 Overmod\n");
printf("\n");
break;
    case 16:
printf("Signal Generator Modulation Uncalibrated\n");
printf("\n");
break;
    default:
printf("No Problems with Modulation\n");
printf("\n");
}
```

```
// Close the sessions
viClose(vi);
viClose(defaultRM);

}
```

Reading the Service Request Interrupt (SRQ) Using VISA and C

This example demonstrates use of the Service Request (SRQ) interrupt. By using the SRQ, the computer can attend to other tasks while the signal generator is busy performing a function or operation. When the signal generator finishes its operation, or detects a failure, then a Service Request can be generated. The computer will respond to the SRQ and, depending on the code, can perform some other operation or notify the user of failures or other conditions.

This program sets up a step sweep function for the signal generator and, while the operation is in progress, prints out a series of asterisks. When the step sweep operation is complete, an SRQ is generated and the printing ceases.

Launch Microsoft Visual C++ 6.0, add the required files, and enter the following code into your .cpp source file. `visaex10.cpp` performs the following functions:

- error checking
- clears the signal generator
- resets the signal generator
- prompts user to manually begin the step sweep and waits for response
- clears the status register
- sets up the operation status group to respond to an end of sweep
- the data questionable status register is enabled to read an unleveled condition
- prompts user to manually set up the signal generator for an unleveled condition
- queries the data questionable status register for any set bits and converts the string data to numeric
- based on the numeric value, program checks for a corresponding status check value
- similarly checks for over or undermodulation condition

The following program example is available on the signal generator Documentation CD-ROM as `visaex10.cpp`.

```
//******************************************************************************
//
// PROGRAM FILE NAME:visaex10.cpp
//
// PROGRAM DESCRIPTION: This example demonstrates the use of a Service Request (SRQ)
// interrupt. The program sets up conditions to enable the SRQ and then sets the signal
// generator for a step mode sweep. The program will enter a printing loop which prints
// an * character and ends when the sweep has completed and an SRQ received.
//
//******************************************************************************
```

```
#include "visa.h"
#include <stdio.h>
#include "StdAfx.h"
#include "windows.h"
#include <conio.h>

#define  MAX_CNT 1024

int sweep=1;  // End of sweep flag

/* Prototypes */

ViStatus _VI_FUNCH interupt(ViSession vi, ViEventType eventType, ViEvent event, ViAddr addr);

int main ()
{
ViSession defaultRM, vi;// Declares variables of type ViSession
// for instrument communication
ViStatus viStatus = 0;// Declares a variable of type ViStatus
                      // for GPIB verifications
char rdBuffer[MAX_CNT];// Declare a block of memory data

viStatus=viOpenDefaultRM(&defaultRM);// Initialize VISA session
if(viStatus < VI_SUCCESS){// If problems, then prompt user
printf("ERROR initializing VISA... exiting\n");
printf("\n");
return -1;}
                              // Open session to gpib device at address 19
viStatus=viOpen(defaultRM, "GPIB::19::INSTR", VI_NULL, VI_NULL, &vi);
if(viStatus){                 // If problems then prompt user
printf("ERROR: Could not open communication with
                                         instrument\n");
printf("\n");
return -1;}

viClear(vi);                  // Clears the signal generator
viPrintf(vi, "*RST\n");       // Resets signal generator
                              // Print program header and information
printf("** End of Sweep Service Request **\n");
printf("\n");
printf("The signal generator will be set up for a step sweep mode
      operation.\n");
printf("An '*' will be printed while the instrument is sweeping. The end of
 \n");
```

```
printf("sweep will be indicated by an SRQ on the GPIB and the program will
                         end.\n");
printf("\n");
printf("Press Enter to continue\n");
printf("\n");
getch();

viPrintf(vi, "*CLS\n");// Clears signal generator status byte
viPrintf(vi, "STAT:OPER:NTR 8\n");// Sets the Operation Status Group  // Negative Transition Filter to
indicate a  // negative transition in Bit 3 (Sweeping)
// which will set a corresponding event in   // the Operation Event Register. This occurs   // at the end
of a sweep.
viPrintf(vi, "STAT:OPER:PTR 0\n");// Sets the Operation Status Group  // Positive Transition Filter so
that no
// positive transition on Bit 3 affects the  // Operation Event Register. The positive  // transition
occurs at the start of a sweep.
viPrintf(vi, "STAT:OPER:ENAB 8\n");// Enables Operation Status Event Bit 3  // to report the event to
Status Byte  // Register Summary Bit 7.
viPrintf(vi, "*SRE 128\n");// Enables Status Byte Register Summary Bit 7
// The next line of code indicates the  // function to call on an event
viStatus = viInstallHandler(vi, VI_EVENT_SERVICE_REQ, interupt, rdBuffer);
// The next line of code enables the  // detection of an event
viStatus = viEnableEvent(vi, VI_EVENT_SERVICE_REQ, VI_HNDLR, VI_NULL);

viPrintf(vi, "FREQ:MODE LIST\n");// Sets frequency mode to list
viPrintf(vi, "LIST:TYPE STEP\n");// Sets sweep to step
viPrintf(vi, "LIST:TRIG:SOUR IMM\n");// Immediately trigger the sweep
viPrintf(vi, "LIST:MODE AUTO\n");// Sets mode for the list sweep
viPrintf(vi, "FREQ:STAR 40 MHZ\n"); // Start frequency set to 40 MHz
viPrintf(vi, "FREQ:STOP 900 MHZ\n");// Stop frequency set to 900 MHz
viPrintf(vi, "SWE:POIN 25\n");// Set number of points for the step sweep
viPrintf(vi, "SWE:DWEL .5 S\n");// Allow .5 sec dwell at each point
viPrintf(vi, "INIT:CONT OFF\n");// Set up for single sweep
viPrintf(vi, "TRIG:SOUR IMM\n");// Triggers the sweep
viPrintf(vi, "INIT\n");  // Takes a single sweep
printf("\n");
// While the instrument is sweeping have the
// program busy with printing to the display.
// The Sleep function, defined in the header
// file windows.h, will pause the program
// operation for .5 seconds
while (sweep==1){
printf("*");
Sleep(500);}
printf("\n");
```

```
// The following lines of code will stop the
// events and close down the session

viStatus = viDisableEvent(vi, VI_ALL_ENABLED_EVENTS,VI_ALL_MECH);
viStatus = viUninstallHandler(vi, VI_EVENT_SERVICE_REQ, interupt,
                                               rdBuffer);
viStatus = viClose(vi);
viStatus = viClose(defaultRM);
return 0;

}

// The following function is called when an SRQ event occurs. Code specific to your
// requirements would be entered in the body of the function.

ViStatus _VI_FUNCH interupt(ViSession vi, ViEventType eventType, ViEvent event, ViAddr
                            addr)
{
ViStatus status;
ViUInt16 stb;

 status = viReadSTB(vi, &stb);// Reads the Status Byte
sweep=0;// Sets the flag to stop the '*' printing
printf("\n");// Print user information
printf("An SRQ, indicating end of sweep has occurred\n");
viClose(event);// Closes the event
return VI_SUCCESS;
}
```

LAN Programming Interface Examples

NOTE Before using the LAN examples: The LAN programming examples in this section demonstrate the use of VXI-11 and Sockets LAN to control the signal generator. For details on using FTP and TELNET refer to “Using FTP” on page 36 and “Using Telnet LAN” on page 32 of this guide.

To use these programming examples you must change references to the IP address and hostname to match the IP address and hostname of your signal generator.

- “VXI-11 Programming Using SICL and C++” on page 88
- “VXI-11 Programming Using VISA and C++” on page 89
- “Sockets LAN Programming and C” on page 91
- “Sockets LAN Programming Using Java” on page 115
- “Sockets LAN Programming Using PERL” on page 116

For additional LAN programming examples that work with user-data files, refer to:

- “Save and Recall Instrument State Files” on page 158

VXI-11 Programming

The signal generator supports the VXI-11 standard for instrument communication over the LAN interface. Agilent IO Libraries support the VXI-11 standard and must be installed on your computer before using the VXI-11 protocol. Refer to “Using VXI-11” on page 30 for information on configuring and using the VXI-11 protocol.

The VXI-11 examples use TCPIP0 as the board address.

Using VXI-11 with GPIB Programs

The GPIB programming examples that use the VISA library, and are listed in this section, can be easily changed to use the LAN VXI-11 protocol by changing the address string. For example, change the "GPIB::19::INSTR" address string to "TCPIP::hostname::INSTR" where hostname is the IP address or hostname of the signal generator. The VXI-11 protocol has the same capabilities as GPIB. See the section “Setting Up the LAN Interface” on page 23 for more information.

NOTE To communicate with the signal generator over the LAN interface you must enable the VXI-11 SCPI service. For more information, refer to “Configuring the DHCP LAN” on page 25.

VXI-11 Programming Using SICL and C++

CAUTION Because of the potential for portability problems, running Agilent SICL without the VISA overlay is not recommended by Agilent Technologies.

The following program uses the VXI-11 protocol and SICL to control the signal generator. Before running this code, you must set up the interface using the Agilent IO Libraries IO Config utility. `vxisicl.cpp` performs the following functions:

- sets signal generator to 1 GHz CW frequency
- queries signal generator for an ID string
- error checking

The following program example is available on the signal generator Documentation CD-ROM as `vxisicl.cpp`.

```
//****************************************************************************************
//
// PROGRAM NAME:vxisicl.cpp
//
// PROGRAM DESCRIPTION:Sample test program using SICL and the VXI-11 protocol
//
// NOTE: You must have the Agilent IO Libraries installed to run this program.
//
// This example uses the VXI-11 protocol to set the signal generator for a 1 gHz CW // frequency. The
signal generator is queried for operation complete and then queried
// for its ID string. The frequency and ID string are then printed to the display.
//
// IMPORTANT: Enter in your signal generators hostname in the instrumentName declaration
// where the "xxxxx" appears.
//
//****************************************************************************************

#include "stdafx.h"
#include <sicl.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char* argv[])
{

INST id;                                    // Device session id
int opcResponse;                            // Variable for response flag
```

```
char instrumentName[] = "xxxxx"; // Put your instrument's hostname here
char instNameBuf[256];// Variable to hold instrument name
char buf[256];// Variable for id string
ionerror(I_ERROR_EXIT);// Register SICL error handler

                // Open SICL instrument handle using VXI-11 protocol

sprintf(instNameBuf, "lan[%s]:inst0", instrumentName);
id = iopen(instNameBuf);// Open instrument session
itimeout(id, 1000);// Set 1 second timeout for operations
printf("Setting frequency to 1 Ghz...\n");
iprintf(id, "freq 1 GHz\n");// Set frequency to 1 GHz

printf("Waiting for source to settle...\n");
iprintf(id, "*opc?\n");// Query for operation complete
iscanf(id, "%d", &opcResponse);  // Operation complete flag
if (opcResponse != 1)// If operation fails, prompt user
 {
   printf("Bad response to 'OPC?'\n");
   iclose(id);
   exit(1);
}
iprintf(id, "FREQ?\n");// Query the frequency
iscanf(id, "%t", &buf);// Read the signal generator frequency
printf("\n");// Print the frequency to the display
printf("Frequency of signal generator is  %s\n", buf);
ipromptf(id, "*IDN?\n", "%t", buf);// Query for id string
printf("Instrument ID: %s\n", buf);// Print id string to display
iclose(id);// Close the session

return 0;
}
```

VXI-11 Programming Using VISA and C++

The following program uses the VXI-11 protocol and the VISA library to control the signal generator. The signal generator is set to a –5 dBm power level and queried for its ID string. Before running this code, you must set up the interface using the Agilent IO Libraries IO Config utility. `vxivisa.cpp` performs the following functions:

- sets signal generator to a –5 dBm power level
- queries signal generator for an ID string
- error checking

The following program example is available on the signal generator Documentation CD-ROM as `vxivisa.cpp`.

```
//****************************************************************************************
// PROGRAM FILE NAME:vxivisa.cpp
// Sample test program using the VISA libraries and the VXI-11 protocol
//
// NOTE: You must have the Agilent Libraries installed on your computer to run
// this program
//
// PROGRAM DESCRIPTION:This example uses the VXI-11 protocol and VISA to query
// the signal generator for its ID string. The ID string is then printed to the
// screen. Next the signal generator is set for a -5 dBm power level and then
// queried for the power level. The power level is printed to the screen.
//
// IMPORTANT: Set up the LAN Client using the IO Config utility
//
//****************************************************************************************

#include <visa.h>
#include <stdio.h>
#include "StdAfx.h"
#include <stdlib.h>
#include <conio.h>

#define MAX_COUNT 200

int main (void)

{

ViStatus status;// Declares a type ViStatus variable
ViSession defaultRM, instr;// Declares a type ViSession variable
ViUInt32 retCount;// Return count for string I/O
ViChar buffer[MAX_COUNT];// Buffer for string I/O

status = viOpenDefaultRM(&defaultRM);    // Initialize the system
                                         // Open communication with Serial
                                         // Port 2
status = viOpen(defaultRM, "TPCIP0::19::INSTR", VI_NULL, VI_NULL, &instr);

if(status){                              // If problems then prompt user
printf("Could not open ViSession!\n");
printf("Check instruments and connections\n");
printf("\n");
```

```
exit(0);}
                                             // Set timeout for 5 seconds
viSetAttribute(instr, VI_ATTR_TMO_VALUE, 5000);
                                         // Ask for sig gen ID string
 status = viWrite(instr, (ViBuf)"*IDN?\n", 6, &retCount);

                                         // Read the sig gen response
 status = viRead(instr, (ViBuf)buffer, MAX_COUNT, &retCount);
buffer[retCount]= '\0';                  // Indicate the end of the string
printf("Signal Generator ID = ");        // Print header for ID
printf(buffer);                          // Print the ID string
printf("\n");                            // Print carriage return
                                         // Flush the read buffer
                                         // Set sig gen power to -5dbm
status = viWrite(instr, (ViBuf)"POW:AMPL -5dbm\n", 15, &retCount);
                                         // Query the power level
status = viWrite(instr, (ViBuf)"POW?\n",5,&retCount);
                                         // Read the power level
status = viRead(instr, (ViBuf)buffer, MAX_COUNT, &retCount);
buffer[retCount]= '\0';                  // Indicate the end of the string
printf("Power level = ");                // Print header to the screen
printf(buffer);                          // Print the queried power level
printf("\n");
status = viClose(instr);                 // Close down the system
status = viClose(defaultRM);
return 0;
}
```

Sockets LAN Programming and C

The program listing shown in “Queries for Lan Using Sockets” on page 94 consists of two files; lanio.c and getopt.c. The lanio.c file has two main functions; `int main()` and an `int main1()`.

The `int main()` function allows communication with the signal generator interactively from the command line. The program reads the signal generator's hostname from the command line, followed by the SCPI command. It then opens a socket to the signal generator, using port 5025, and sends the command. If the command appears to be a query, the program queries the signal generator for a response, and prints the response.

The `int main1()`, after renaming to `int main()`, will output a sequence of commands to the signal generator. You can use the format as a template and then add your own code.

This program is available on the signal generator Documentation CD-ROM as `lanio.c`.

Sockets on UNIX

In UNIX, LAN communication via sockets is very similar to reading or writing a file. The only

difference is the `openSocket()` routine, which uses a few network library routines to create the TCP/IP network connection. Once this connection is created, the standard `fread()` and `fwrite()` routines are used for network communication. The following steps outline the process:

1. Copy the lanio.c and getopt.c files to your home UNIX directory. For example, `/users/mydir/`.
2. At the UNIX prompt in your home directory type: `cc -Aa -O -o lanio lanio.c`
3. At the UNIX prompt in your home directory type: `./lanio xxxxx "*IDN?"` where `xxxxx` is the hostname for the signal generator. Use this same format to output SCPI commands to the signal generator.

The `int main1()` function will output a sequence of commands in a program format. If you want to run a program using a sequence of commands then perform the following:

1. Rename the lanio.c `int main1()` to `int main()` and the original `int main()` to `int main1()`.
2. In the `main()`, `openSocket()` function, change the “your hostname here” string to the hostname of the signal generator you want to control.
3. Resave the lanio.c program.
4. At the UNIX prompt type: `cc -Aa -O -o lanio lanio.c`
5. At the UNIX prompt type: `./lanio`

The program will run and output a sequence of SCPI commands to the signal generator. The UNIX display will show a display similar to the following:

```
unix machine: /users/mydir
$ ./lanio
ID: Agilent Technologies, E4438C, US70000001, C.02.00

Frequency: +2.5000000000000E+09
Power Level: -5.00000000E+000
```

Sockets on Windows

In Windows, the routines send() and recv() must be used, since fread() and fwrite() may not work on sockets. The following steps outline the process for running the interactive program in the Microsoft Visual C++ 6.0 environment:

1. Rename the lanio.c to lanio.cpp and getopt.c to getopt.cpp and add them to the Source folder of the Visual C++ project.

NOTE The int main() function in the lanio.cpp file will allow commands to be sent to the signal generator in a line-by-line format; the user types in SCPI commands. The int main1(0) function can be used to output a sequence of commands in a “program format.” See Programming Using main1() Function below.

---

2. Click **Rebuild All** from **Build** menu. Then Click **Execute Lanio.exe**. The Debug window will appear with a prompt “Press any key to continue.” This indicates that the program has compiled and can be used to send commands to the signal generator.

3. Click **Start**, click **Programs**, then click **Command Prompt**. The command prompt window will appear.
4. At the command prompt, `cd` to the directory containing the lanio.exe file and then to the Debug folder. For example `C:\SocketIO\Lanio\Debug`.
5. After you cd to the directory where the lanio.exe file is located, type in the following command at the command prompt: `lanio xxxxx "*IDN?"`. For example: `C:\SocketIO\Lanio\Debug>lanio xxxxx "*IDN?"` where the xxxxx is the hostname of your signal generator. Use this format to output SCPI commands to the signal generator in a line by line format from the command prompt.
6. Type `exit` at the command prompt to quit the program.

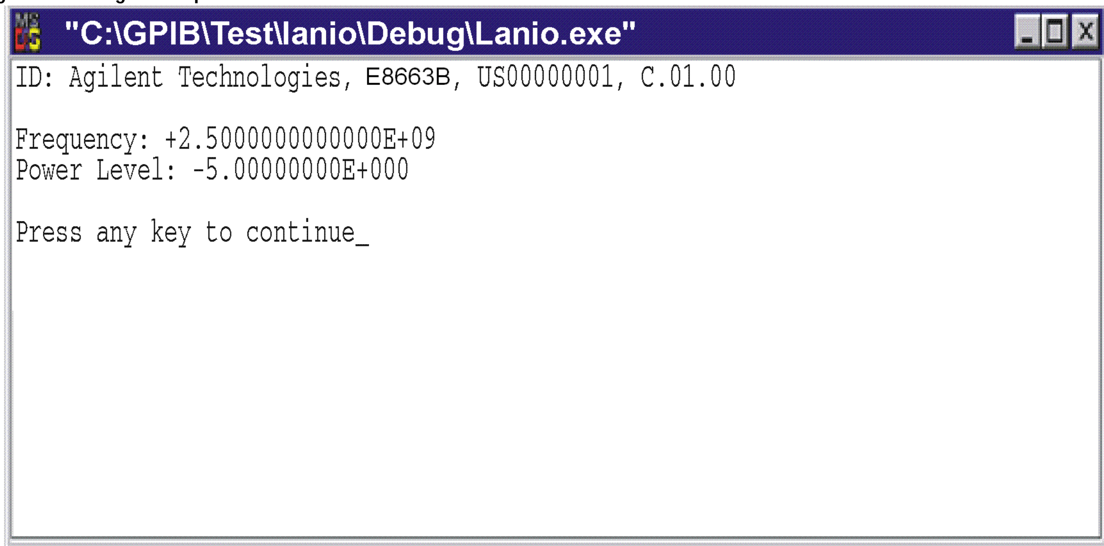
Programming Using main1() Function

The `int main1()` function will output a sequence of commands in a program format. If you want to run a program using a sequence of commands then perform the following:

1. Enter the hostname of your signal generator in the openSocket function of the `main1()` function of the lanio.cpp program.
2. Rename the lanio.cpp `int main1()` function to `int main()` and the original `int main()` function to `int main1()`.
3. Select **Rebuild All** from **Build** menu. Then select **Execute Lanio.exe**.

The program will run and display results similar to those shown in Figure 3-1.

Figure 3-1 Program Output Screen



Queries for Lan Using Sockets

lanio.c and getopt.c perform the following functions:

- establishes TCP/IP connection to port 5025
- resultant file descriptor is used to “talk” to the instrument using regular socket I/O mechanisms
- maps the desired hostname to an internal form
- error checks
- queries signal generator for ID
- sets frequency on signal generator to 2.5 GHz
- sets power on signal generator to –5 dBm
- gets option letter from argument vector and checks for end of file (EOF)

The following programming examples are available on the signal generator Documentation CD-ROM as lanio.c and getopt.c.

```
/***************************************************************************
 *  $Header: lanio.c 04/24/01
 *  $Revision: 1.1 $
 *  $Date: 10/24/01
 *  PROGRAM NAME:   lanio.c
 *
 *  $Description:     Functions to talk to an Agilent signal generator
 *                    via TCP/IP.  Uses command-line arguments.
 *
 *                    A TCP/IP connection to port 5025 is established and
 *                    the resultant file descriptor is used to "talk" to the
 *                    instrument using regular socket I/O mechanisms. $
 *
 *
 *
 *  Examples:
 *
 *   Query the signal generator frequency:
 *        lanio xx.xxx.xx.x 'FREQ?'
 *
 *   Query the signal generator power level:
 *        lanio xx.xxx.xx.x  'POW?'
 *
 *   Check for errors (gets one error):
 *        lanio xx.xxx.xx.x  'syst:err?'
 *
 *   Send a list of commands from a file, and number them:
```

```
 *        cat scpi_cmds | lanio -n xx.xxx.xx.x
 *
 ***************************************************************************
 *
 *  This program compiles and runs under
 *     - HP-UX 10.20 (UNIX), using HP cc or gcc:
 *           + cc -Aa    -O -o lanio  lanio.c
 *           + gcc -Wall -O -o lanio  lanio.c
 *
 *     - Windows 95, using Microsoft Visual C++ 4.0 Standard Edition
 *     - Windows NT 3.51, using Microsoft Visual C++ 4.0
 *           + Be sure to add  WSOCK32.LIB  to your list of libraries!
 *           + Compile both lanio.c and getopt.c
 *           + Consider re-naming the files to lanio.cpp and getopt.cpp
 *
 *  Considerations:
 *     - On UNIX systems, file I/O can be used on network sockets.
 *       This makes programming very convenient, since routines like
 *       getc(), fgets(), fscanf() and fprintf() can be used.  These
 *       routines typically use the lower level read() and write() calls.
 *
 *     - In the Windows environment, file operations such as read(), write(),
 *       and close() cannot be assumed to work correctly when applied to
 *       sockets.  Instead, the functions send() and recv() MUST be used.
 *****************************************************************************/

/* Support both Win32 and HP-UX UNIX environment */

#ifdef _WIN32     /* Visual C++ 6.0 will define this */
#  define WINSOCK
#endif

#ifndef WINSOCK
#  ifndef _HPUX_SOURCE
#  define _HPUX_SOURCE
#  endif
#endif

#include <stdio.h>         /* for fprintf and NULL  */
#include <string.h>        /* for memcpy and memset */
#include <stdlib.h>        /* for malloc(), atol() */
#include <errno.h>         /* for strerror          */
```

```
#ifdef WINSOCK

#include <windows.h>

#  ifndef _WINSOCKAPI_
#  include <winsock.h>   // BSD-style socket functions
#  endif

#else                        /* UNIX with BSD sockets */

#  include <sys/socket.h>    /* for connect and socket*/
#  include <netinet/in.h>    /* for sockaddr_in       */
#  include <netdb.h>         /* for gethostbyname     */

#  define SOCKET_ERROR (-1)
#  define INVALID_SOCKET (-1)

   typedef  int SOCKET;

#endif /* WINSOCK */

#ifdef WINSOCK
  /* Declared in getopt.c.  See example programs disk. */
  extern char *optarg;
  extern int  optind;
  extern int getopt(int argc, char * const argv[], const char* optstring);
#else
#  include <unistd.h>           /* for getopt(3C) */
#endif

#define COMMAND_ERROR  (1)
#define NO_CMD_ERROR  (0)

#define SCPI_PORT  5025
#define INPUT_BUF_SIZE (64*1024)



/***************************************************************************
 * Display usage
 ***************************************************************************/
```

```
static void usage(char *basename)
{
    fprintf(stderr,"Usage: %s [-nqu] <hostname> [<command>]\n", basename);
    fprintf(stderr,"       %s [-nqu] <hostname> < stdin\n", basename);
    fprintf(stderr,"  -n, number output lines\n");
    fprintf(stderr,"  -q, quiet; do NOT echo lines\n");
    fprintf(stderr,"  -e, show messages in error queue when done\n");
}



#ifdef WINSOCK
int init_winsock(void)
{
    WORD wVersionRequested;
    WSADATA wsaData;
    int err;
    wVersionRequested = MAKEWORD(1, 1);
    wVersionRequested = MAKEWORD(2, 0);

    err = WSAStartup(wVersionRequested, &wsaData);

    if (err != 0) {
        /* Tell the user that we couldn't find a useable */
        /* winsock.dll.     */
        fprintf(stderr, "Cannot initialize Winsock 1.1.\n");
        return -1;
    }
    return 0;
}

int close_winsock(void)
{
    WSACleanup();
    return 0;
}
#endif /* WINSOCK */



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

```
 > $Function: openSocket$
 *
 * $Description:  open a TCP/IP socket connection to the instrument $
 *
 * $Parameters:  $
 *    (const char *) hostname . . . . Network name of instrument.
 *                                    This can be in dotted decimal notation.
 *    (int) portNumber  . . . . . . . The TCP/IP port to talk to.
 *                                    Use 5025 for the SCPI port.
 *
 * $Return:     (int)  . . . . . . . . A file descriptor similar to open(1).$
 *
 * $Errors:     returns -1 if anything goes wrong $
 *
 ***************************************************************************/
SOCKET openSocket(const char *hostname, int portNumber)
{
    struct hostent *hostPtr;
    struct sockaddr_in peeraddr_in;
    SOCKET s;


    memset(&peeraddr_in, 0, sizeof(struct sockaddr_in));

    /***********************************************/
    /* map the desired host name to internal form. */
    /***********************************************/
    hostPtr = gethostbyname(hostname);
    if (hostPtr == NULL)
    {
        fprintf(stderr,"unable to resolve hostname '%s'\n", hostname);
        return INVALID_SOCKET;
    }

    /*******************/
    /* create a socket */
    /*******************/
    s = socket(AF_INET, SOCK_STREAM, 0);
    if (s == INVALID_SOCKET)
    {
        fprintf(stderr,"unable to create socket to '%s': %s\n",
                hostname, strerror(errno));
```

```
        return INVALID_SOCKET;
    }

    memcpy(&peeraddr_in.sin_addr.s_addr, hostPtr->h_addr, hostPtr->h_length);
    peeraddr_in.sin_family = AF_INET;
    peeraddr_in.sin_port = htons((unsigned short)portNumber);

    if (connect(s, (const struct sockaddr*)&peeraddr_in,
                sizeof(struct sockaddr_in)) == SOCKET_ERROR)
    {
        fprintf(stderr,"unable to create socket to '%s': %s\n",
                hostname, strerror(errno));
        return INVALID_SOCKET;
    }

    return s;
}



/***************************************************************************
 *
 > $Function: commandInstrument$
 *
 * $Description:  send a SCPI command to the instrument.$
 *
 * $Parameters:  $
 *     (FILE *) . . . . . . . . . file pointer associated with TCP/IP socket.
 *     (const char *command)  . . SCPI command string.
 * $Return:  (char *) . . . . . . a pointer to the result string.
 *
 * $Errors:   returns 0 if send fails $
 *
 ***************************************************************************/
int commandInstrument(SOCKET sock,
                      const char *command)
{
    int count;

    /* fprintf(stderr, "Sending \"%s\".\n", command);  */
    if (strchr(command, '\n') == NULL) {
       fprintf(stderr, "Warning: missing newline on command %s.\n", command);
```

```
    }

    count = send(sock, command, strlen(command), 0);
    if (count == SOCKET_ERROR) {
        return COMMAND_ERROR;
    }

    return NO_CMD_ERROR;
}


/***************************************************************************
 * recv_line(): similar to fgets(), but uses recv()
 ***************************************************************************/
char * recv_line(SOCKET sock, char * result, int maxLength)
{
#ifdef WINSOCK
    int cur_length = 0;
    int count;
    char * ptr = result;
    int err = 1;

    while (cur_length < maxLength) {
        /* Get a byte into ptr */
        count = recv(sock, ptr, 1, 0);

        /* If no chars to read, stop. */
        if (count < 1) {
            break;
        }
        cur_length += count;

        /* If we hit a newline, stop. */
        if (*ptr == '\n') {
            ptr++;
            err = 0;
            break;
        }
        ptr++;

    }
```

```
    *ptr = '\0';

    if (err) {
        return NULL;
    } else {
        return result;
    }
#else
    /**********************************************************************
     * Simpler UNIX version, using file I/O.  recv() version works too.
     * This demonstrates how to use file I/O on sockets, in UNIX.
     **********************************************************************/
    FILE * instFile;
    instFile = fdopen(sock, "r+");
    if (instFile == NULL)
    {
        fprintf(stderr, "Unable to create FILE * structure : %s\n",
                strerror(errno));
        exit(2);
    }
    return fgets(result, maxLength, instFile);
#endif
}



/***************************************************************************
 *
 > $Function: queryInstrument$
 *
 * $Description:  send a SCPI command to the instrument, return a response.$
 *
 * $Parameters:  $
 *     (FILE *) . . . . . . . . . file pointer associated with TCP/IP socket.
 *     (const char *command)  . . SCPI command string.
 *     (char *result) . . . . . . where to put the result.
 *     (size_t) maxLength . . . . maximum size of result array in bytes.
 *
 * $Return:  (long) . . . . . . . The number of bytes in result buffer.
 *
 * $Errors:   returns 0 if anything goes wrong. $
 *
```

```
 ****************************************************************************/
long queryInstrument(SOCKET sock,
                      const char *command, char *result, size_t maxLength)
{
    long ch;
    char tmp_buf[8];
    long resultBytes = 0;
    int command_err;
    int count;

    /*********************************************************
     * Send command to signal generator
     *********************************************************/
    command_err = commandInstrument(sock, command);
    if (command_err) return COMMAND_ERROR;


    /*********************************************************
     * Read response from signal generator
     ********************************************************/
    count = recv(sock, tmp_buf, 1, 0); /* read 1 char */
    ch = tmp_buf[0];

    if ((count < 1) || (ch == EOF)  || (ch == '\n'))
    {
        *result = '\0';  /* null terminate result for ascii */
        return 0;
    }

    /* use a do-while so we can break out */
    do
    {
        if (ch == '#')
        {
            /* binary data encountered - figure out what it is */
            long numDigits;
            long numBytes = 0;
            /* char length[10]; */

            count = recv(sock, tmp_buf, 1, 0); /* read 1 char */
            ch = tmp_buf[0];
            if ((count < 1) || (ch == EOF)) break; /* End of file */
```

```
if (ch < '0' || ch > '9') break;  /* unexpected char */
numDigits = ch - '0';

if (numDigits)
{
    /* read numDigits bytes into result string. */
    count = recv(sock, result, (int)numDigits, 0);
    result[count] = 0;  /* null terminate */
    numBytes = atol(result);
}

if (numBytes)
{
    resultBytes = 0;
    /* Loop until we get all the bytes we requested. */
    /* Each call seems to return up to 1457 bytes, on HP-UX 9.05 */
    do {
        int rcount;
        rcount = recv(sock, result, (int)numBytes, 0);
        resultBytes += rcount;
        result      += rcount;  /* Advance pointer */
    } while ( resultBytes < numBytes );

    /************************************************************
     * For LAN dumps, there is always an extra trailing newline
     * Since there is no EOI line.  For ASCII dumps this is
     * great but for binary dumps, it is not needed.
     ************************************************************/
    if (resultBytes == numBytes)
    {
        char junk;
        count = recv(sock, &junk, 1, 0);
    }
}
else
{
    /* indefinite block ... dump til we can an extra line feed */
    do
    {
        if (recv_line(sock, result, maxLength) == NULL) break;
        if (strlen(result)==1 && *result == '\n') break;
```

```
                    resultBytes += strlen(result);
                    result += strlen(result);
                } while (1);
            }
        }
        else
        {
            /* ASCII response (not a binary block) */
            *result = (char)ch;
            if (recv_line(sock, result+1, maxLength-1) == NULL) return 0;

            /* REMOVE trailing newline, if present.  And terminate string. */
            resultBytes = strlen(result);
            if (result[resultBytes-1] == '\n') resultBytes -= 1;
            result[resultBytes] = '\0';
        }
    } while (0);

    return resultBytes;
}




/*************************************************************************
 *
 > $Function: showErrors$
 *
 * $Description: Query the SCPI error queue, until empty.  Print results. $
 *
 * $Return:  (void)
 *
 *************************************************************************/
void showErrors(SOCKET sock)
{
    const char * command = "SYST:ERR?\n";
    char result_str[256];

    do {
        queryInstrument(sock, command, result_str, sizeof(result_str)-1);

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

```
         * Typical result_str:
         *     -221,"Settings conflict; Frequency span reduced."
         *     +0,"No error"
         * Don't bother decoding.
         ******************************************************************/
        if (strncmp(result_str, "+0,", 3) == 0) {
            /* Matched +0,"No error" */
            break;
        }
        puts(result_str);
    } while (1);

}


/***************************************************************************
 *
 > $Function: isQuery$
 *
 * $Description: Test current SCPI command to see if it a query. $
 *
 * $Return:  (unsigned char) . . . non-zero if command is a query.  0 if not.
 *
 ***************************************************************************/
unsigned char isQuery( char* cmd )
{
    unsigned char q = 0 ;
    char *query ;

    /*********************************************************/
    /* if the command has a '?' in it, use queryInstrument.  */
    /* otherwise, simply send the command.                   */
    /* Actually, we must be a more specific so that   */
    /* marker value querys are treated as commands.         */
    /* Example:  SENS:FREQ:CENT (CALC1:MARK1:X?)             */
    /*********************************************************/
    if ( (query = strchr(cmd,'?')) != NULL)
    {
        /* Make sure we don't have a marker value query, or
         * any command with a '?' followed by a ')' character.
         * This kind of command is not a query from our point of view.
         * The signal generator does the query internally, and uses the result.
```

```
         */
        query++ ;       /* bump past '?' */
        while (*query)
        {
            if (*query == ' ') /* attempt to ignore white spc */
                query++ ;
            else break ;
        }

        if ( *query != ')' )
        {
            q = 1 ;
        }
    }
    return q ;
}

/***************************************************************************
 *
 > $Function: main$
 *
 * $Description: Read command line arguments, and talk to signal generator.
                 Send query results to stdout. $
 *
 * $Return:  (int) . . . non-zero if an error occurs
 *
 ***************************************************************************/

int main(int argc, char *argv[])
{

    SOCKET instSock;
    char *charBuf = (char *) malloc(INPUT_BUF_SIZE);
    char *basename;
    int chr;
    char command[1024];
    char *destination;
    unsigned char quiet = 0;
    unsigned char show_errs = 0;
    int number = 0;

    basename = strrchr(argv[0], '/');
```

```
if (basename != NULL)
    basename++ ;
else
    basename = argv[0];

while ( ( chr = getopt(argc,argv,"qune")) != EOF )
    switch (chr)
    {
        case 'q':  quiet = 1; break;
        case 'n':  number = 1; break ;
        case 'e':  show_errs = 1; break ;
        case 'u':
        case '?':  usage(basename); exit(1) ;
    }

/* now look for hostname and optional <command>*/
if (optind < argc)
{
    destination = argv[optind++] ;
    strcpy(command, "");
    if (optind < argc)
    {
        while (optind < argc) {
            /* <hostname> <command> provided; only one command string */
            strcat(command, argv[optind++]);
            if (optind < argc) {
                strcat(command, " ");
            } else {
                strcat(command, "\n");
            }
        }
    }
    else
    {
        /*Only <hostname> provided; input on <stdin> */
        strcpy(command, "");

        if (optind > argc)
        {
            usage(basename);
            exit(1);
        }
```

```
        }
    }
    else
    {
        /* no hostname! */
        usage(basename);
        exit(1);
    }

    /*******************************************************
    /* open a socket connection to the instrument
    /*******************************************************/

#ifdef WINSOCK
    if (init_winsock() != 0) {
        exit(1);
    }
#endif /* WINSOCK */

    instSock = openSocket(destination, SCPI_PORT);
    if (instSock == INVALID_SOCKET) {
        fprintf(stderr, "Unable to open socket.\n");
        return 1;
    }
    /* fprintf(stderr, "Socket opened.\n"); */

    if (strlen(command) > 0)
    {
    /*******************************************************
    /* if the command has a '?' in it, use queryInstrument. */
    /* otherwise, simply send the command.                  */
    /*******************************************************/
        if ( isQuery(command) )
        {
            long bufBytes;
            bufBytes = queryInstrument(instSock, command,
                                       charBuf, INPUT_BUF_SIZE);
            if (!quiet)
            {
                fwrite(charBuf, bufBytes, 1, stdout);
                fwrite("\n", 1, 1, stdout) ;
                fflush(stdout);
```

```
            }
        }
        else
        {
            commandInstrument(instSock, command);
        }
    }
    else
    {
        /* read a line from <stdin> */
        while ( gets(charBuf) != NULL )
        {
            if ( !strlen(charBuf) )
                continue ;

            if ( *charBuf == '#' || *charBuf == '!' )
                continue ;

            strcat(charBuf, "\n");

            if (!quiet)
            {
                if (number)
                {
                    char num[10];
                    sprintf(num,"%d: ",number);
                    fwrite(num, strlen(num), 1, stdout);
                }
                fwrite(charBuf, strlen(charBuf), 1, stdout) ;
                fflush(stdout);
            }

            if ( isQuery(charBuf) )
            {
                long bufBytes;

                /* Put the query response into the same buffer as the*/
                /* command string appended after the null terminator.*/

                bufBytes = queryInstrument(instSock, charBuf,
                                           charBuf + strlen(charBuf) + 1,
                                           INPUT_BUF_SIZE -strlen(charBuf) );
```

```
                if (!quiet)
                {
                    fwrite("  ", 2, 1, stdout) ;
                    fwrite(charBuf + strlen(charBuf)+1, bufBytes, 1, stdout);
                    fwrite("\n", 1, 1, stdout) ;
                    fflush(stdout);
                }
            }
            else
            {
                commandInstrument(instSock, charBuf);
            }
            if (number) number++;
        }
    }

    if (show_errs) {
        showErrors(instSock);
    }

#ifdef WINSOCK
    closesocket(instSock);
    close_winsock();
#else
    close(instSock);
#endif /* WINSOCK */

    return 0;
}

/* End of lanio.cpp  *



/***************************************************************************/
/* $Function: main1$                                                       */
/* $Description: Output a series of SCPI commands to the signal generator */
/*               Send query results to stdout. $                           */
/*                                                                         */
/* $Return:  (int) . . . non-zero if an error occurs                       */
/*                                                                         */
/***************************************************************************/
```

```
/* Rename this int main1() function to int main(). Re-compile and the     */
/* execute the program                                                    */
/*************************************************************************/

int main1()
{

SOCKET instSock;
long bufBytes;
    char *charBuf = (char *) malloc(INPUT_BUF_SIZE);


    /*********************************************/
    /* open a socket connection to the instrument*/
    /*********************************************/

#ifdef WINSOCK
    if (init_winsock() != 0) {
        exit(1);
    }
#endif /* WINSOCK */

    instSock = openSocket("xxxxxx", SCPI_PORT); /* Put your hostname here */
    if (instSock == INVALID_SOCKET) {
        fprintf(stderr, "Unable to open socket.\n");
        return 1;
    }
    /* fprintf(stderr, "Socket opened.\n"); */

   bufBytes = queryInstrument(instSock, "*IDN?\n", charBuf, INPUT_BUF_SIZE);
   printf("ID: %s\n",charBuf);
   commandInstrument(instSock, "FREQ 2.5 GHz\n");
   printf("\n");
   bufBytes = queryInstrument(instSock, "FREQ:CW?\n", charBuf, INPUT_BUF_SIZE);
   printf("Frequency: %s\n",charBuf);
   commandInstrument(instSock, "POW:AMPL -5 dBm\n");
   bufBytes = queryInstrument(instSock, "POW:AMPL?\n", charBuf, INPUT_BUF_SIZE);
   printf("Power Level: %s\n",charBuf);
   printf("\n");


#ifdef WINSOCK
```

```
    closesocket(instSock);
    close_winsock();
#else
    close(instSock);
#endif /* WINSOCK */

    return 0;
}
/***************************************************************************

 getopt(3C)                                                     getopt(3C)

 PROGRAM FILE NAME: getopt.c
 getopt - get option letter from argument vector

 SYNOPSIS
      int getopt(int argc, char * const argv[], const char *optstring);
      extern char *optarg;
      extern int optind, opterr, optopt;

 PRORGAM DESCRIPTION:
      getopt returns the next option letter in argv (starting from argv[1])
      that matches a letter in optstring.  optstring is a string of
      recognized option letters; if a letter is followed by a colon, the
      option is expected to have an argument that may or may not be
      separated from it by white space.  optarg is set to point to the start
      of the option argument on return from getopt.

      getopt places in optind the argv index of the next argument to be
      processed.  The external variable optind is initialized to 1 before
      the first call to the function getopt.

      When all options have been processed (i.e., up to the first non-option
      argument), getopt returns EOF.  The special option -- can be used to
      delimit the end of the options; EOF is returned, and -- is skipped.

 ***************************************************************************/


#include <stdio.h>      /* For NULL, EOF */
#include <string.h>     /* For strchr() */
```

```
char    *optarg;        /* Global argument pointer. */
int     optind = 0;     /* Global argv index. */

static char     *scan = NULL;   /* Private scan pointer. */

int getopt( int argc, char * const argv[], const char* optstring)
{
    char c;
    char *posn;

    optarg = NULL;

    if (scan == NULL || *scan == '\0') {
        if (optind == 0)
            optind++;

        if (optind >= argc || argv[optind][0] != '-' || argv[optind][1] == '\0')
            return(EOF);
        if (strcmp(argv[optind], "--")==0) {
            optind++;
            return(EOF);
        }

        scan = argv[optind]+1;
        optind++;
    }

    c = *scan++;
    posn = strchr(optstring, c);        /* DDP */

    if (posn == NULL || c == ':') {
        fprintf(stderr, "%s: unknown option -%c\n", argv[0], c);
        return('?');
    }

    posn++;
    if (*posn == ':') {
        if (*scan != '\0') {
            optarg = scan;
            scan = NULL;
        } else {
            optarg = argv[optind];
```

```
            optind++;
        }
    }

    return(c);
}
```

Sockets LAN Programming Using Java

In this example the Java program connects to the signal generator via sockets LAN. This program requires Java version 1.1 or later be installed on your PC. To run the program perform the following steps:

1. In the code example below, type in the hostname or IP address of your signal generator. For example, `String instrumentName = (your signal generator's hostname)`.
2. Copy the program as `ScpiSockTest.java` and save it in a convenient directory on your computer. For example save the file to the `C:\jdk1.3.0_2\bin\javac` directory.
3. Launch the Command Prompt program on your computer. Click **Start** > **Programs** > **Command Prompt**.
4. Compile the program. At the command prompt type: `javac ScpiSockTest.java`.
   The directory path for the Java compiler must be specified. For example:
   `C:\>jdk1.3.0_02\bin\javac ScpiSockTest.java`
5. Run the program by typing `java ScpiSockTest` at the command prompt.
6. Type `exit` at the command prompt to end the program.

Generating a CW Signal Using Java

The following program example is available on the signal generator Documentation CD-ROM as `javaex.txt`.

```
//****************************************************************************
// PROGRAM NAME: javaex.txt                                          // Sample java
program to talk to the signal generator via SCPI-over-sockets
// This program requires Java version 1.1 or later.
// Save this code as ScpiSockTest.java
// Compile by typing: javac ScpiSockTest.java
// Run by typing: java ScpiSockTest
// The signal generator is set for 1 GHz and queried for its id string
//****************************************************************************

import java.io.*;
import java.net.*;
class ScpiSockTest
{
    public static void main(String[] args)
    {
       String instrumentName = "xxxxx";         // Put instrument hostname here
try
        {
       Socket t = new Socket(instrumentName,5025);  // Connect to instrument
                                                      // Setup read/write mechanism
            BufferedWriter out =
            new BufferedWriter(
```

```
            new OutputStreamWriter(t.getOutputStream()));
            BufferedReader in =
            new BufferedReader(
            new InputStreamReader(t.getInputStream()));
            System.out.println("Setting frequency to 1 GHz...");
            out.write("freq 1GHz\n");               // Sets frequency
            out.flush();
            System.out.println("Waiting for source to settle...");
            out.write("*opc?\n");                   // Waits for completion
            out.flush();
            String opcResponse = in.readLine();
            if (!opcResponse.equals("1"))
             {
              System.err.println("Invalid response to '*OPC?'!");
              System.exit(1);
             }
        System.out.println("Retrieving instrument ID...");
        out.write("*idn?\n");                       // Querys the id string
        out.flush();
        String idnResponse = in.readLine();         // Reads the id string
                                                    // Prints the id string
        System.out.println("Instrument ID: " + idnResponse);
        }
        catch (IOException e)
        {
        System.out.println("Error" + e);
      }
    }
}
```

Sockets LAN Programming Using PERL

This example uses PERL script to control the signal generator over the sockets LAN interface. The signal generator frequency is set to 1 GHz, queried for operation complete and then queried for it's identify string. This example was developed using PERL version 5.6.0 and requires a PERL version with the IO::Socket library.

1. In the code below, enter your signal generator's hostname in place of the `xxxxx` in the code line: `my $instrumentName= "xxxxx";` .
2. Save the code listed below using the filename `lanperl`.
3. Run the program by typing `perl lanperl` at the UNIX term window prompt.

Setting the Power Level and Sending Queries Using PERL

The following program example is available on the signal generator Documentation CD-ROM as

perl.txt.

```
#!/usr/bin/perl
# PROGRAM NAME: perl.txt
# Example of talking to the signal generator via SCPI-over-sockets
#
use IO::Socket;
# Change to your instrument's hostname
my $instrumentName = "xxxxx";

# Get socket
$sock = new IO::Socket::INET ( PeerAddr => $instrumentName,
                               PeerPort => 5025,
                               Proto => 'tcp',
                               );
die "Socket Could not be created, Reason: $!\n" unless $sock;

# Set freq
print "Setting frequency...\n";
print $sock "freq 1 GHz\n";

# Wait for completion
print "Waiting for source to settle...\n";
print $sock "*opc?\n";
my $response = <$sock>;
chomp $response;           # Removes newline from response
if ($response ne "1")
{
   die "Bad response to '*OPC?' from instrument!\n";
}

# Send identification query
print $sock "*IDN?\n";
$response = <$sock>;
chomp $response;
print "Instrument ID: $response\n";
```

RS-232 Programming Interface Examples

- "Interface Check Using HP BASIC" on page 118
- "Interface Check Using VISA and C" on page 119
- "Queries Using HP Basic and RS-232" on page 121
- "Queries for RS-232 Using VISA and C" on page 122

Before Using the Examples

Before using the examples: On the signal generator select the following settings:

- Baud Rate - 9600 must match computer's baud rate
- RS-232 Echo - Off

NOTE For LAN programming examples, refer to "LAN Programming Interface Examples" on page 87.

Interface Check Using HP BASIC

This example program causes the signal generator to perform an instrument reset. The SCPI command *RST will place the signal generator into a pre-defined state.

The serial interface address for the signal generator in this example is 9. The serial port used is COM1 (Serial A on some computers). Refer to "Using RS-232" on page 38 for more information.

Watch for the signal generator's Listen annunciator (L) and the 'remote preset....' message on the front panel display. If there is no indication, check that the RS-232 cable is properly connected to the computer serial port and that the manual setup listed above is correct.

If the compiler displays an error message, or the program hangs, it is possible that the program was typed incorrectly. Press the signal generator's **Reset RS-232** softkey and re-run the program. Refer to "If You Have Problems" on page 42 for more help.

The following program example is available on the signal generator's Documentation CD-ROM as rs232ex1.txt.

```
10    !******************************************************************************
20    !
30    !  PROGRAM NAME:        rs232ex1.txt
40    !
50    !  PROGRAM DESCRIPTION:  This program verifies that the RS-232 connections and
60    !                        interface are functional.
70    !
80    !  Connect the UNIX workstation to the signal generator using an RS-232 cable
90    !
100   !
110   !  Run HP BASIC, type in the following commands and then RUN the program
120   !
```

```
130   !
140   !*****************************************************************************
150   !
160    INTEGER Num
170    CONTROL 9,0;1        ! Resets the RS-232 interface
180    CONTROL 9,3;9600     ! Sets the baud rate to match the sig gen
190    STATUS 9,4;Stat      ! Reads the value of register 4
200    Num=BINAND(Stat,7)   ! Gets the AND value
210    CONTROL 9,4;Num      ! Sets parity to NONE
220    OUTPUT 9;"*RST"      ! Outputs reset to the sig gen
230    END                  ! End the program
```

Interface Check Using VISA and C

This program uses VISA library functions to communicate with the signal generator. The program verifies that the RS-232 connections and interface are functional. In this example the COM2 port is used. The serial port is referred to in the VISA library as ‘ASRL1’ or ‘ASRL2’ depending on the computer serial port you are using. Launch Microsoft Visual C++, add the required files, and enter the following code into the .cpp source file. `rs232ex1.cpp` performs the following functions:

- prompts the user to set the power on the signal generator to 0 dBm
- error checking
- resets the signal generator to power level of –135 dBm

The following program example is available on the signal generator Documentation CD-ROM as rs232ex1.cpp.

```
//****************************************************************************************
// PROGRAM NAME:       rs232ex1.cpp
//
// PROGRAM DESCRIPTION: This code example uses the RS-232 serial interface to
// control the signal generator.
//
// Connect the computer to the signal generator using an RS-232 serial cable.
// The user is asked to set the signal generator for a 0 dBm power level
// A reset command *RST is sent to the signal generator via the RS-232
// interface and the power level will reset to the -135 dBm level.The default
// attributes e.g. 9600 baud, no parity, 8 data bits,1 stop bit are used.
// These attributes can be changed using VISA functions.
//
// IMPORTANT: Set the signal generator BAUD rate to 9600 for this test
//****************************************************************************************

#include <visa.h>
#include <stdio.h>
#include "StdAfx.h"
#include <stdlib.h>
```

```
#include <conio.h>



void main ()
{

int baud=9600;// Set baud rate to 9600
printf("Manually set the signal generator power level to 0 dBm\n");
printf("\n");
printf("Press any key to continue\n");
getch();
printf("\n");
ViSession defaultRM, vi;// Declares a variable of type ViSession
// for instrument communication on COM 2 port
ViStatus viStatus = 0;
                        // Opens session to RS-232 device at serial port 2
viStatus=viOpenDefaultRM(&defaultRM);
viStatus=viOpen(defaultRM, "ASRL2::INSTR", VI_NULL, VI_NULL, &vi);

if(viStatus){// If operation fails, prompt user
    printf("Could not open ViSession!\n");
    printf("Check instruments and connections\n");
    printf("\n");
    exit(0);}
// initialize device
viStatus=viEnableEvent(vi, VI_EVENT_IO_COMPLETION, VI_QUEUE,VI_NULL);

viClear(vi);// Sends device clear command
// Set attributes for the session
viSetAttribute(vi,VI_ATTR_ASRL_BAUD,baud);
viSetAttribute(vi,VI_ATTR_ASRL_DATA_BITS,8);

viPrintf(vi, "*RST\n");// Resets the signal generator
printf("The signal generator has been reset\n");
printf("Power level should be -135 dBm\n");
printf("\n");// Prints new line character to the display
viClose(vi);// Closes session
viClose(defaultRM);// Closes default session
}
```

Queries Using HP Basic and RS-232

This example program demonstrates signal generator query commands over RS-232. Query commands are of the type `*IDN?` and are identified by the question mark that follows the mnemonic. `rs232ex2.txt` performs the following functions:

- resets the RS-232 interface
- sets the baud rate to match the signal generator rate
- reads the value of register 4
- queries the signal generator ID
- sets and queries the power level

Start HP Basic, type in the following commands, and then RUN the program:

The following program example is available on the signal generator Documentation CD-ROM as `rs232ex2.txt`.

```
10    !*******************************************************************************
20    !
30    !  PROGRAM NAME:         rs232ex2.txt
40    !
50    !  PROGRAM DESCRIPTION:  In this example, query commands are used to read
60    !                        data from the signal generator.
70    !
80    !  Start HP Basic, type in the following code and then RUN the program.
90    !
100   !*******************************************************************************
110   !
120    INTEGER Num
130    DIM Str$[200],Str1$[20]
140    CONTROL 9,0;1               ! Resets the RS-232 interface
150    CONTROL 9,3;9600            ! Sets the baud rate to match signal generator rate
160    STATUS 9,4;Stat             ! Reads the value of register 4
170    Num=BINAND(Stat,7)          ! Gets the AND value
180    CONTROL 9,4;Num             ! Sets the parity to NONE
190    OUTPUT 9;"*IDN?"            ! Querys the sig gen ID
200    ENTER 9;Str$                ! Reads the ID
210    WAIT 2                      ! Waits 2 seconds
220    PRINT "ID =",Str$           ! Prints ID to the screen
230    OUTPUT 9;"POW:AMPL -5 dbm"  ! Sets the the power level to -5 dbm
240    OUTPUT 9;"POW?"             ! Querys the power level of the sig gen
250    ENTER 9;Str1$               ! Reads the queried value
260    PRINT "Power = ",Str1$      ! Prints the power level to the screen
270    END                         ! End the program
```

Queries for RS-232 Using VISA and C

This example uses VISA library functions to communicate with the signal generator. The program verifies that the RS-232 connections and interface are functional. Launch Microsoft Visual C++, add the required files, and enter the following code into your .cpp source file. rs232ex2.cpp performs the following functions:

- error checking
- reads the signal generator response
- flushes the read buffer
- queries the signal generator for power
- reads the signal generator power

The following program example is available on the signal generator Documentation CD-ROM as rs232ex2.cpp.

```
//****************************************************************************************
//
// PROGRAM NAME:       rs232ex2.cpp
//
// PROGRAM DESCRIPTION: This code example uses the RS-232 serial interface to control
// the signal generator.
//
// Connect the computer to the signal generator using the RS-232 serial cable
// and enter the following code into the project .cpp source file.
// The program queries the signal generator ID string and sets and queries the power
// level. Query results are printed to the screen. The default attributes e.g. 9600 baud,
// parity, 8 data bits,1 stop bit are used. These attributes can be changed using VISA
// functions.
//
// IMPORTANT: Set the signal generator BAUD rate to 9600 for this test
//****************************************************************************************


#include <visa.h>
#include <stdio.h>
#include "StdAfx.h"
#include <stdlib.h>
#include <conio.h>

#define MAX_COUNT 200

int main (void)
{
```

```
ViStatusstatus;  // Declares a type ViStatus variable
ViSessiondefaultRM, instr;// Declares type ViSession variables
ViUInt32retCount;  // Return count for string I/O
ViCharbuffer[MAX_COUNT];// Buffer for string I/O

status = viOpenDefaultRM(&defaultRM);// Initializes the system
// Open communication with Serial Port 2
status = viOpen(defaultRM, "ASRL2::INSTR", VI_NULL, VI_NULL, &instr);

if(status){// If problems, then prompt user
printf("Could not open ViSession!\n");
printf("Check instruments and connections\n");
printf("\n");
exit(0);}
                                        // Set timeout for 5 seconds
viSetAttribute(instr, VI_ATTR_TMO_VALUE, 5000);
// Asks for sig gen ID string
 status = viWrite(instr, (ViBuf)"*IDN?\n", 6, &retCount);

// Reads the sig gen response
 status = viRead(instr, (ViBuf)buffer, MAX_COUNT, &retCount);
buffer[retCount]= '\0';// Indicates the end of the string
printf("Signal Generator ID: "); // Prints header for ID
printf(buffer);// Prints the ID string to the screen
printf("\n");// Prints carriage return
// Flush the read buffer
// Sets sig gen power to -5dbm
status = viWrite(instr, (ViBuf)"POW:AMPL -5dbm\n", 15, &retCount);
// Querys the sig gen for power level
status = viWrite(instr, (ViBuf)"POW?\n",5,&retCount);
// Read the power level
status = viRead(instr, (ViBuf)buffer, MAX_COUNT, &retCount);
buffer[retCount]= '\0';// Indicates the end of the string
printf("Power level = ");// Prints header to the screen
printf(buffer);// Prints the queried power level
printf("\n");
status = viClose(instr);// Close down the system
status = viClose(defaultRM);
return 0;
}
```

4 Programming the Status Register System

- "Overview" on page 126
- "Status Register Bit Values" on page 129
- "Accessing Status Register Information" on page 130
- "Status Byte Group" on page 134
- "Status Groups" on page 136

Overview

NOTE Some of the status bits do not apply to the E8663B. For more specific information on each exception, refer to the following:

- Standard Operation Condition Register bits (see Table 4-5 on page 140)
- Data Questionable Condition Register bits (see Table 4-6 on page 143)
- Data Questionable Power Condition Register bits (see Table 4-7 on page 146)
- Data Questionable Frequency Condition Register bits (see Table 4-8 on page 149)
- Data Questionable Modulation Condition Register bits (see Table 4-9 on page 152)
- Data Questionable Calibration Condition Register bit (see Table 4-10 on page 155)

During remote operation, you may need to monitor the status of the signal generator for error conditions or status changes. For more information on using the signal generator's SCPI commands to query the signal generator's error queue, refer to signal generator's SCPI command reference guide, to see if any errors have occurred. An alternative method uses the signal generator's status register system to monitor error conditions, or condition changes, or both.

The signal generator's status register system provides two major advantages:

- You can monitor the settling of the signal generator using the settling bit of the Standard Operation Status Group's condition register.
- You can use the service request (SRQ) interrupt technique to avoid status polling, therefore giving a speed advantage.

The signal generator's instrument status system provides complete SCPI Standard data structures for reporting instrument status using the register model.

The SCPI register model of the status system has multiple registers that are arranged in a hierarchical order. The lower-priority status registers propagate their data to the higher-priority registers using summary bits. The Status Byte Register is at the top of the hierarchy and contains the status information for lower level registers. The lower level registers monitor specific events or conditions.

The lower level status registers are grouped according to their functionality. For example, the Data Quest. Frequency Status Group consists of five registers. This chapter may refer to a group as a register so that the cumbersome correct description is avoided. For example, the Standard Operation Status Group's Condition Register can be referred to as the Standard Operation Status register. Refer to "Status Groups" on page 136 for more information.

Figure 4-1 and Figure 4-2 shows the signal generator's status byte register system and hierarchy.

The status register systems use IEEE 488.2 commands (those beginning with *) to access the higher-level summary registers (refer to the *SCPI Reference*). Access Lower-level registers by using STATus commands.

Figure 4-1 E8663B: Overall Status Byte Register System (1 of 2)

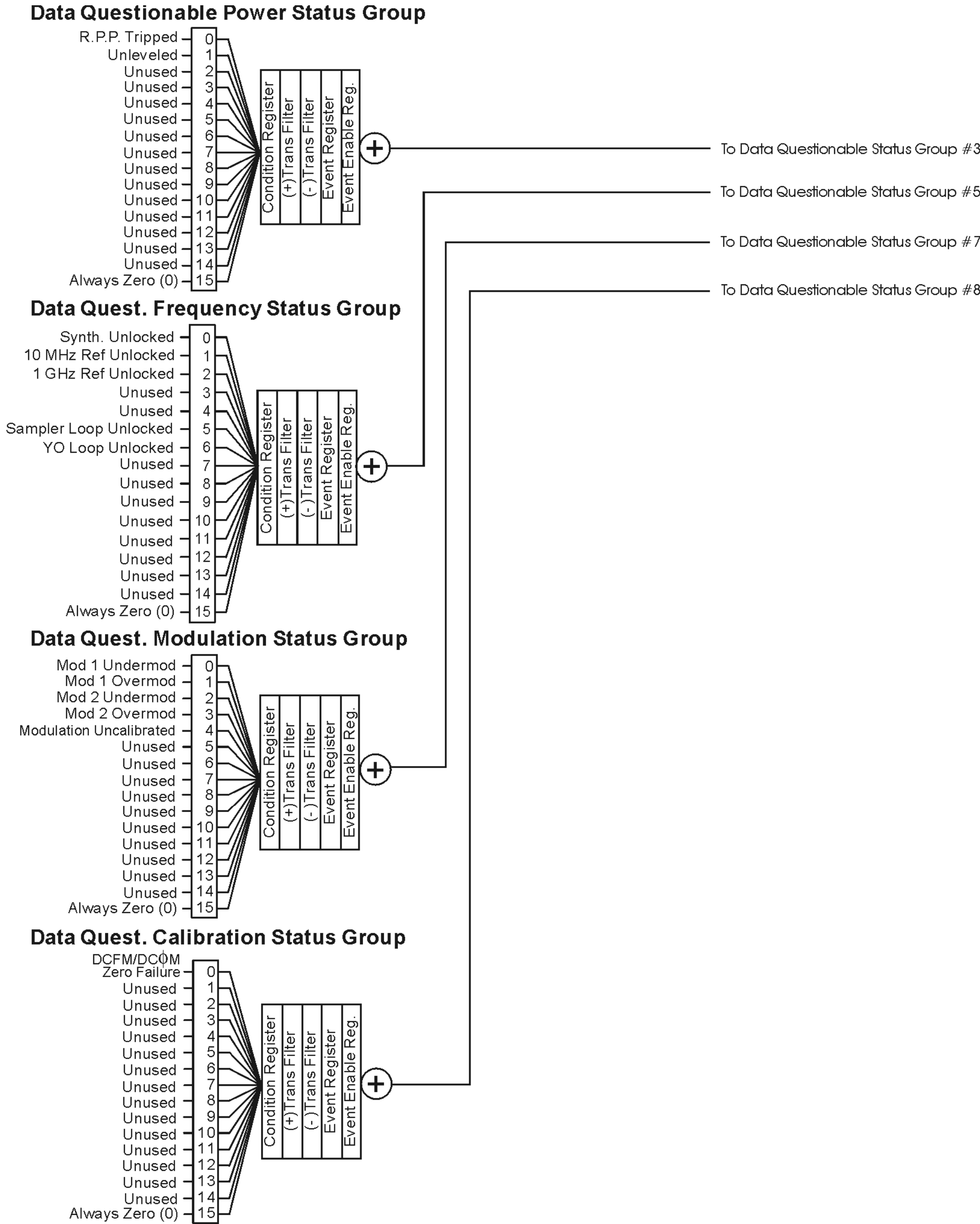
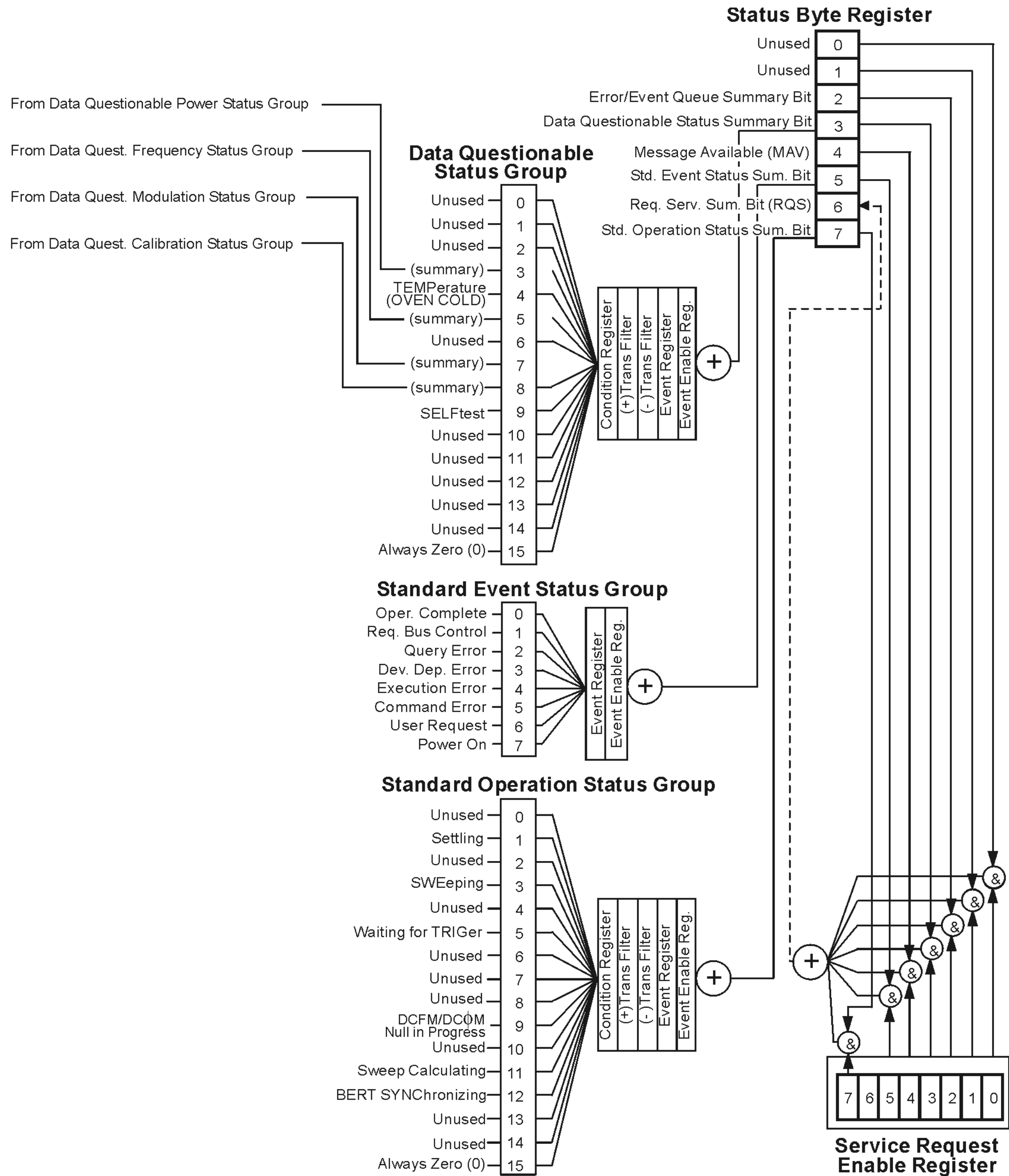


Figure 4-2 E8663B: Overall Status Byte Register System (2 of 2)



Status Register Bit Values

Each bit in a register is represented by a decimal value based on its location in the register (see Table 4-1).

- To enable a particular bit in a register, send its value with the SCPI command. Refer to the signal generator's SCPI command listing for more information.
- To enable more than one bit, send the sum of all the bits that you want to enable.
- To verify the bits set in a register, query the register.

Example: Enable a Register

To enable bit 0 and bit 6 of the Standard Event Status Group's Event Register:

1. Add the decimal value of bit 0 (1) and the decimal value of bit 6 (64) to give a decimal value of 65.
2. Send the sum with the command: `*ESE 65`.

Example: Query a Register

To query a register for a condition, send a SCPI query command. For example, if you want to query the Standard Operation Status Group's Condition Register, send the command:

STATus:OPERation:CONDition?

If bit 7, bit 3 and bit 2 in this register are set (bits = 1) then the query will return the decimal value 140. The value represents the decimal values of bit 7, bit 3 and bit 2: 128 + 8 + 4 = 140.

Table 4-1 Status Register Bit Decimal Values

Decimal Value	Always 0	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
Bit Number	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**

NOTE Bit 15 is not used and is always set to zero.

Accessing Status Register Information

1. Determine which register contains the bit that reports the condition. Refer to Figure 4-1 on page 127 and Figure 4-2 on page 128 for register location and names.
2. Send the unique SCPI query that reads that register.
3. Examine the bit to see if the condition has changed.

Determining What to Monitor

You can monitor the following conditions:

- current signal generator hardware and firmware status
- whether a particular condition (bit) has occurred

Monitoring Current Signal Generator Hardware and Firmware Status

To monitor the signal generator's operating status, you can query the condition registers. These registers represent the current state of the signal generator and are updated in real time. When the condition monitored by a particular bit becomes true, the bit sets to 1. When the condition becomes false, the bit resets to 0.

Monitoring Whether a Condition (Bit) has Changed

The transition registers determine which bit transition (condition change) should be recorded as an event. The transitions can be positive to negative, negative to positive, or both. To monitor a certain condition, enable the bit associated with the condition in the associated positive and negative registers.

Once you have enabled a bit via the transition registers, the signal generator monitors it for a change in its condition. If this change in condition occurs, the corresponding bit in the event register will be set to 1. When a bit becomes true (set to 1) in the event register, it stays set until the event register is read or is cleared. You can thus query the event register for a condition even if that condition no longer exists.

To clear the event register, query its contents or send the *CLS command, which clears *all* event registers.

Monitoring When a Condition (Bit) Changes

Once you enable a bit, the signal generator monitors it for a change in its condition. The transition registers are preset to register positive transitions (a change going from 0 to 1). This can be changed so the selected bit is detected if it goes from true to false (negative transition), or if either transition occurs.

Deciding How to Monitor

You can use either of two methods described below to access the information in status registers (both methods allow you to monitor one or more conditions).

- **The polling method**

  In the polling method, the signal generator has a passive role. It tells the controller that conditions have changed only when the controller asks the right question. This is accomplished by a program loop that continually sends a query.

The polling method works well if you do not need to know about changes the moment they occur. Use polling in the following situations:

– when you use a programming language/development environment or IO interface that does not support SRQ interrupts
– when you want to write a simple, single-purpose program and don't want the added complexity of setting up an SRQ handler

• **The service request (SRQ) method**

In the SRQ method (described in the following section), the signal generator takes a more active role. It tells the controller when there has been a condition change without the controller asking. Use the SRQ method to detect changes using the polling method, where the program must repeatedly read the registers.

Use the SRQ method if you must know immediately when a condition changes. Use the SRQ method in the following situations:

– when you need time-critical notification of changes
– when you are monitoring more than one device that supports SRQs
– when you need to have the controller do something else while waiting
– when you can't afford the performance penalty inherent to polling

Using the Service Request (SRQ) Method

The programming language, I/O interface, and programming environment must support SRQ interrupts (for example: BASIC or VISA used with GPIB and VXI-11 over the LAN). Using this method, you must do the following:

1. Determine which bit monitors the condition.
2. Send commands to enable the bit that monitors the condition (transition registers).
3. Send commands to enable the summary bits that report the condition (event enable registers).
4. Send commands to enable the status byte register to monitor the condition.
5. Enable the controller to respond to service requests.

The controller responds to the SRQ as soon as it occurs. As a result, the time the controller would otherwise have used to monitor the condition, as in a loop method, can be used to perform other tasks. The application determines how the controller responds to the SRQ.

When a condition changes and that condition has been enabled, the request service summary (RQS) bit in the status byte register is set. In order for the controller to respond to the change, the Service Request Enable Register needs to be enabled for the bit(s) that will trigger the SRQ.

Generating a Service Request

The Service Request Enable Register lets you choose the bits in the Status Byte Register that will trigger a service request. Send the *SRE <num> command where <num> is the sum of the decimal values of the bits you want to enable.

For example, to enable bit 7 on the Status Byte Register (so that whenever the Standard Operation Status register summary bit is set to 1, a service request is generated) send the command *SRE 128. Refer to Figure 4-1 on page 127 and Figure 4-2 on page 128 for bit positions and values.

The query command *SRE? returns the decimal value of the sum of the bits previously enabled with the *SRE <num> command.

To query the Status Byte Register, send the command *STB?. The response will be the decimal sum of the bits which are set to 1. For example, if bit 7 and bit 3 are set, the decimal sum will be 136 (bit 7 = 128 and bit 3 = 8).

NOTE Multiple Status Byte Register bits can assert an SRQ, however only one bit at a time can set the RQS bit. All bits that are asserting an SRQ will be read as part of the status byte when it is queried or serial polled.

The SRQ process asserts SRQ as true and sets the status byte's RQS bit to 1. Both actions are necessary to inform the controller that the signal generator requires service. Asserting SRQ informs the controller that some device on the bus requires service. Setting the RQS bit allows the controller to determine which signal generator requires service.

This process is initiated if both of the following conditions are true:

- The corresponding bit of the Service Request Enable Register is also set to 1.
- The signal generator does not have a service request pending.

  A service request is considered to be pending between the time the signal generator's SRQ process is initiated and the time the controller reads the status byte register.

If a program enables the controller to detect and respond to service requests, it should instruct the controller to perform a serial poll when SRQ is true. Each device on the bus returns the contents of its status byte register in response to this poll. The device whose request service summary (RQS) bit is set to 1 is the device that requested service.

NOTE When you read the signal generator's Status Byte Register with a serial poll, the RQS bit is reset to 0. Other bits in the register are not affected.

If the status register is configured to SRQ on end-of-sweep or measurement and the mode set to continuous, restarting the measurement (INIT command) can cause the measuring bit to pulse low. This causes an SRQ when you have not actually reached the "end-of-sweep" or measurement condition. To avoid this, do the following:

1. Send the command `INITiate:CONTinuous OFF`.
2. Set/enable the status registers.
3. Restart the measurement (send INIT).

Status Register SCPI Commands

Most monitoring of signal generator conditions is done at the highest level using the IEEE 488.2 common commands listed below. You can set and query individual status registers using the commands in the STATus subsystem.

*CLS (clear status) clears the Status Byte Register by emptying the error queue and clearing all the event registers.

*ESE, *ESE? (event status enable) sets and queries the bits in the Standard Event Enable Register which is part of the Standard Event Status Group.

*ESR? (event status register) queries and clears the Standard Event Status Register which is part of the Standard Event Status Group.

*OPC, *OPC? (operation complete) sets bit #0 in the Standard Event Status Register to 1 when all commands have completed. The query stops any new commands from being processed until the current processing is complete, then returns a 1.

*PSC, *PSC? (power-on state clear) sets the power-on state so that it clears the Service Request Enable Register, the Standard Event Status Enable Register, and device-specific event enable registers at power on. The query returns the flag setting from the *PSC command.

*SRE, *SRE? (service request enable) sets and queries the value of the Service Request Enable Register.

*STB? (status byte) queries the value of the status byte register without erasing its contents.

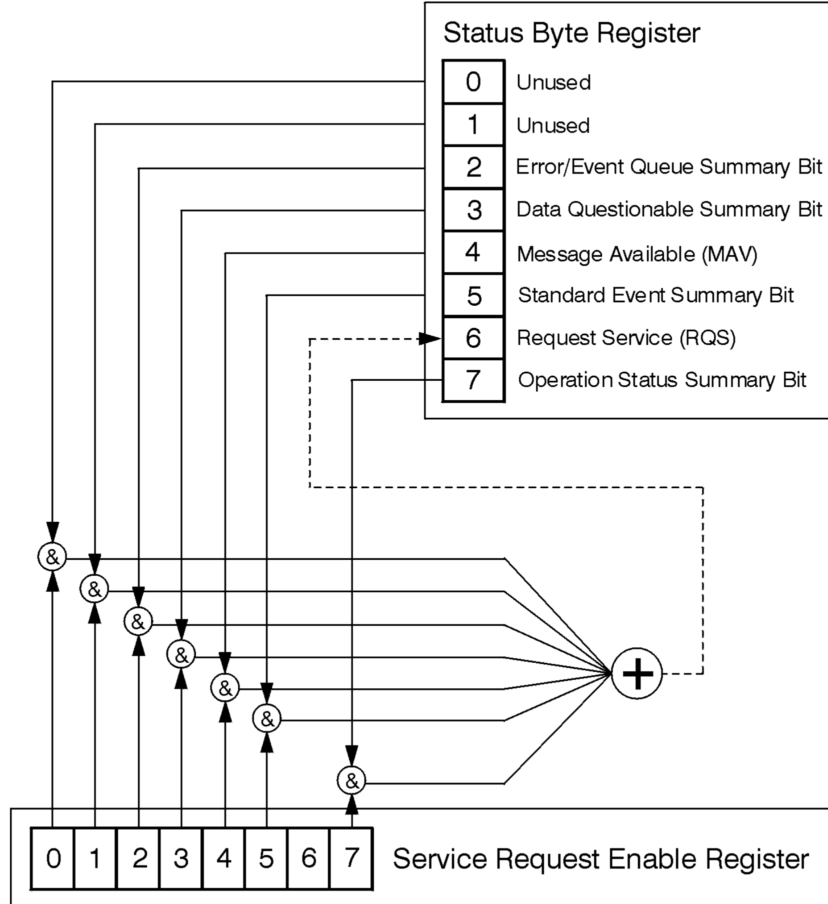
:STATus:PRESet presets all transition filters, non-IEEE 488.2 enable registers, and error/event queue enable registers. (Refer to Table 4-2.)

Table 4-2 Effects of :STATus:PRESet

Register	**Value after :STATus:PRESet**
:STATus:OPERation:ENABle	0
:STATus:OPERation:NTRansition	0
:STATus:OPERation:PTRransition	32767
:STATus:QUEStionable:CALibration:ENABle	32767
:STATus:QUEStionable:CALibration:NTRansition	32767
:STATus:QUEStionable:CALibration:PTRansition	32767
:STATus:QUEStionable:ENABle	0
:STATus:QUEStionable:NTRansition	0
:STATus:QUEStionable:PTRansition	32767
:STATus:QUEStionable:FREQuency:ENABle	32767
:STATus:QUEStionable:FREQuency:NTRansition	32767
:STATus:QUEStionable:FREQuency:PTRansition	32767
:STATus:QUEStionable:MODulation:ENABle	32767
:STATus:QUEStionable:MODulation:NTRansition	32767
:STATus:QUEStionable:MODulation:PTRansition	32767
:STATus:QUEStionable:POWer:ENABle	32767
:STATus:QUEStionable:POWer:NTRansition	32767
:STATus:QUEStionable:POWer:PTRansition	32767

Status Byte Group

The Status Byte Group includes the Status Byte Register and the Service Request Enable Register.



Status Byte Register

Table 4-3 Status Byte Register Bits

Bit	Description
0,1	**Unused**. These bits are always set to 0.
2	**Error/Event Queue Summary Bit**. A 1 in this bit position indicates that the SCPI error queue is not empty. The SCPI error queue contains at least one error message.
3	**Data Questionable Status Summary Bit**. A 1 in this bit position indicates that the Data Questionable summary bit has been set. The Data Questionable Event Register can then be read to determine the specific condition that caused this bit to be set.
4	**Message Available**. A 1 in this bit position indicates that the signal generator has data ready in the output queue. There are no lower status groups that provide input to this bit.
5	**Standard Event Status Summary Bit**. A 1 in this bit position indicates that the Standard Event summary bit has been set. The Standard Event Status Register can then be read to determine the specific event that caused this bit to be set.
6	**Request Service (RQS) Summary Bit**. A 1 in this bit position indicates that the signal generator has at least one reason to require service. This bit is also called the Master Summary Status bit (MSS). The individual bits in the Status Byte are individually ANDed with their corresponding service request enable register, then each individual bit value is ORed and input to this bit.
7	**Standard Operation Status Summary Bit**. A 1 in this bit position indicates that the Standard Operation Status Group's summary bit has been set. The Standard Operation Event Register can then be read to determine the specific condition that caused this bit to be set.

Query: `*STB?`

Response: The *decimal* sum of the bits set to 1 including the master summary status bit (MSS) bit 6.

Example: The decimal value 136 is returned when the MSS bit is set low (0).

Decimal sum = 128 (bit 7) + 8 (bit 3)

The decimal value 200 is returned when the MSS bit is set high (1).

Decimal sum = 128 (bit 7) + 8 (bit 3) + 64 (MSS bit)

Service Request Enable Register

The Service Request Enable Register lets you choose which bits in the Status Byte Register trigger a service request.

`*SRE <data>` <data> is the sum of the decimal values of the bits you want to enable except bit 6. Bit 6 cannot be enabled on this register. Refer to Figure 4-1 on page 127 and Figure 4-2 on page 128.

Example: To enable bits 7 and 5 to trigger a service request when either corresponding status group register summary bit sets to 1, send the command `*SRE 160` (128 + 32).

Query: `*SRE?`

Response: The decimal value of the sum of the bits previously enabled with the `*SRE <data>` command.

Status Groups

The Standard Operation Status Group and the Data Questionable Status Group consist of the registers listed below. The Standard Event Status Group is similar but does *not* have negative or positive transition filters or a condition register.

Condition Register
: A condition register continuously monitors the hardware and firmware status of the signal generator. There is no latching or buffering for a condition register; it is updated in real time.

Negative Transition Filter
: A negative transition filter specifies the bits in the condition register that will set corresponding bits in the event register when the condition bit changes from 1 to 0.

Positive Transition Filter
: A positive transition filter specifies the bits in the condition register that will set corresponding bits in the event register when the condition bit changes from 0 to 1.

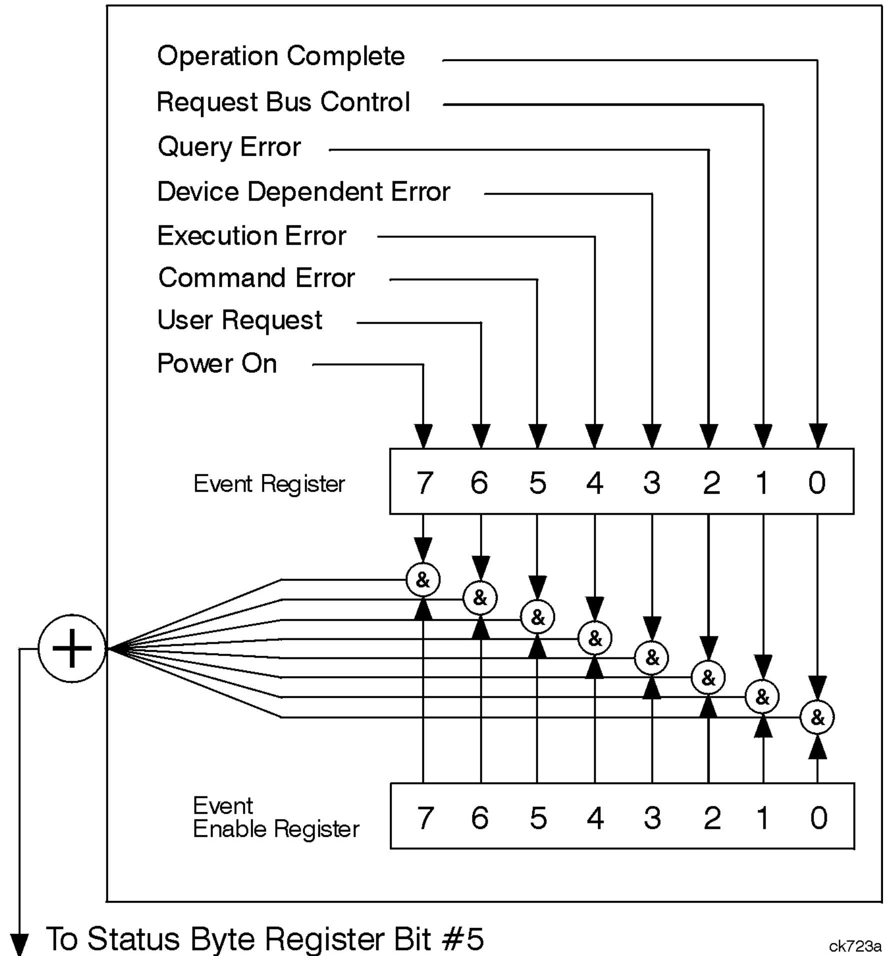
Event Register
: An event register latches transition events from the condition register as specified by the positive and negative transition filters. Once the bits in the event register are set, they remain set until cleared by either querying the register contents or sending the `*CLS` command.

Event Enable Register
: An enable register specifies the bits in the event register that generate the summary bit. The signal generator logically ANDs corresponding bits in the event and enable registers and ORs all the resulting bits to produce a summary bit. Summary bits are, in turn, used by the Status Byte Register.

A status group is a set of related registers whose contents are programmed to produce status summary bits. In each status group, corresponding bits in the condition register are filtered by the negative and positive transition filters and stored in the event register. The contents of the event register are logically ANDed with the contents of the enable register and the result is logically ORed to produce a status summary bit in the Status Byte Register.

Standard Event Status Group

The Standard Event Status Group is used to determine the specific event that set bit 5 in the Status Byte Register. This group consists of the Standard Event Status Register (an event register) and the Standard Event Status Enable Register.



Standard Event Status Register

Table 4-4 Standard Event Status Register Bits

Bit	Description
0	**Operation Complete**. A 1 in this bit position indicates that all pending signal generator operations were completed following execution of the *OPC command.
1	**Request Control**. This bit is always set to 0. (The signal generator does not request control.)
2	**Query Error**. A 1 in this bit position indicates that a query error has occurred. Query errors have instrument error numbers from –499 to –400.
3	**Device Dependent Error**. A 1 in this bit position indicates that a device dependent error has occurred. Device dependent errors have instrument error numbers from –399 to –300 and 1 to 32767.
4	**Execution Error**. A 1 in this bit position indicates that an execution error has occurred. Execution errors have instrument error numbers from –299 to –200.
5	**Command Error**. A 1 in this bit position indicates that a command error has occurred. Command errors have instrument error numbers from –199 to –100.
6	**User Request Key (Local)**. A 1 in this bit position indicates that the Local key has been pressed. This is true even if the signal generator is in local lockout mode.
7	**Power On**. A 1 in this bit position indicates that the signal generator has been turned off and then on.

Query: `*ESR?`

Response: The *decimal* sum of the bits set to 1

Example: The decimal value 136 is returned. The decimal sum = 128 (bit 7) + 8 (bit 3).

Standard Event Status Enable Register

The Standard Event Status Enable Register lets you choose which bits in the Standard Event Status Register set the summary bit (bit 5 of the Status Byte Register) to 1.

`*ESE <data>` <data> is the sum of the decimal values of the bits you want to enable.

Example: To enable bit 7 and bit 6 so that whenever either of those bits are set to 1, the Standard Event Status summary bit of the Status Byte Register is set to 1. Send the command `*ESE 192` (128 + 64).

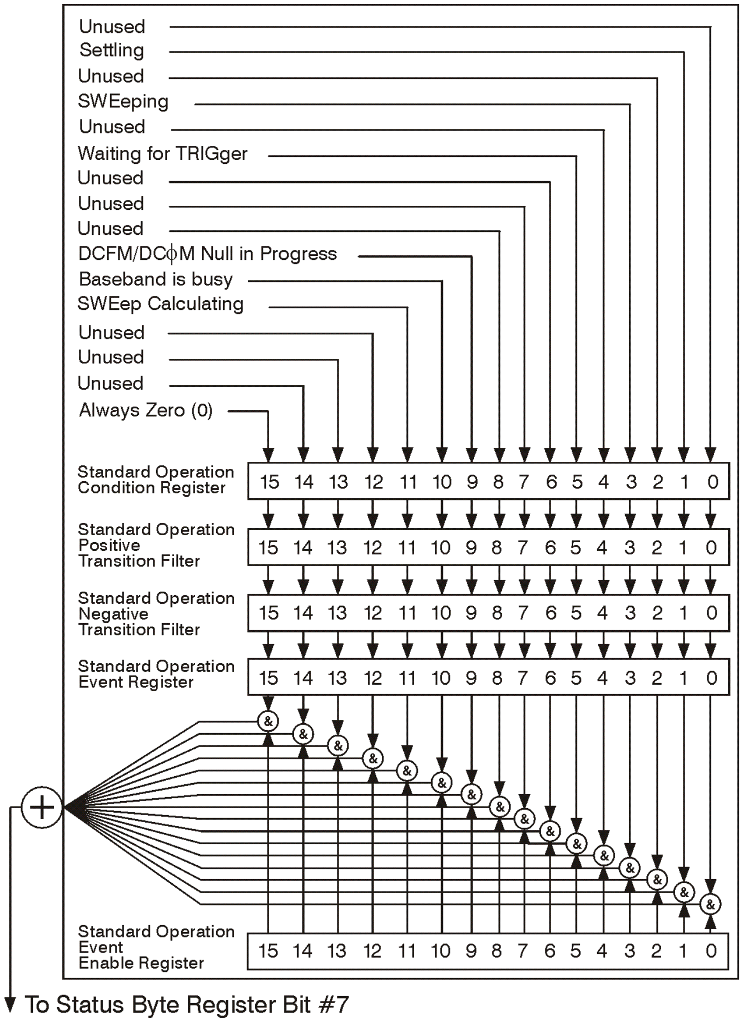
Query: `*ESE?`

Response: Decimal value of the sum of the bits previously enabled with the `*ESE <data>` command.

Standard Operation Status Group

NOTE Some of the bits in this status group do not apply to the E8663B and returns zero when queried. See Table 4-5 on page 140 for more information.

The Operation Status Group is used to determine the specific event that set bit 7 in the Status Byte Register. This group consists of the Standard Operation Condition Register, the Standard Operation Transition Filters (negative and positive), the Standard Operation Event Register, and the Standard Operation Event Enable Register.



Standard Operation Condition Register

The Standard Operation Condition Register continuously monitors the hardware and firmware status of the signal generator. Condition registers are read only.

Table 4-5 Standard Operation Condition Register Bits

Bit	Description
0	**Unused.** These bits are always set to 0.
1	**Settling**. A 1 in this bit position indicates that the signal generator is settling.
2	**Unused**. This bit position is always set to 0.
3	**Sweeping**. A 1 in this bit position indicates that a sweep is in progress.
4	**Unused.** These bits are always set to 0.
5	**Waiting for Trigger**. A 1 in this bit position indicates that the source is in a “wait for trigger” state.
6,7,8	**Unused**. These bits are always set to 0.
9	**DCFM/DCϕM Null in Progress**. A 1 in this bit position indicates that the signal generator is currently performing a DCFM/DCΦM zero calibration.
10	**Unused.** These bits are always set to 0.
11	**Sweep Calculating**. A 1 in this bit position indicates that the signal generator is currently doing the necessary pre-sweep calculations.
12, 13, 14	**Unused**. These bits are always set to 0.
15	**Always 0**.

Query: `STATus:OPERation:CONDition?`

Response: The *decimal* sum of the bits set to 1

Example: The decimal value 520 is returned. The decimal sum = 512 (bit 9) + 8 (bit 3).

Standard Operation Transition Filters (negative and positive)

The Standard Operation Transition Filters specify which types of bit state changes in the condition register set corresponding bits in the event register. Changes can be positive (0 to 1) or negative (1 to 0).

Commands: `STATus:OPERation:NTRansition <value>` (negative transition), or
`STATus:OPERation:PTRansition <value>` (positive transition), where
`<value>` is the sum of the decimal values of the bits you want to enable.

Queries: `STATus:OPERation:NTRansition?`
`STATus:OPERation:PTRansition?`

Standard Operation Event Register

The Standard Operation Event Register latches transition events from the condition register as specified by the transition filters. Event registers are destructive read only. Reading data from an event register clears the content of that register.

Query: `STATus:OPERation[:EVENt]?`

Standard Operation Event Enable Register

The Standard Operation Event Enable Register lets you choose which bits in the Standard Operation Event Register set the summary bit (bit 7 of the Status Byte Register) to 1.

Command: `STATus:OPERation:ENABle <value>`, where
`<value>` is the sum of the decimal values of the bits you want to enable.

Example: To enable bit 9 and bit 3 so that whenever either of those bits are set to 1, the Standard Operation Status summary bit of the Status Byte Register is set to 1. Send the command `STAT:OPER:ENAB 520` (512 + 8).

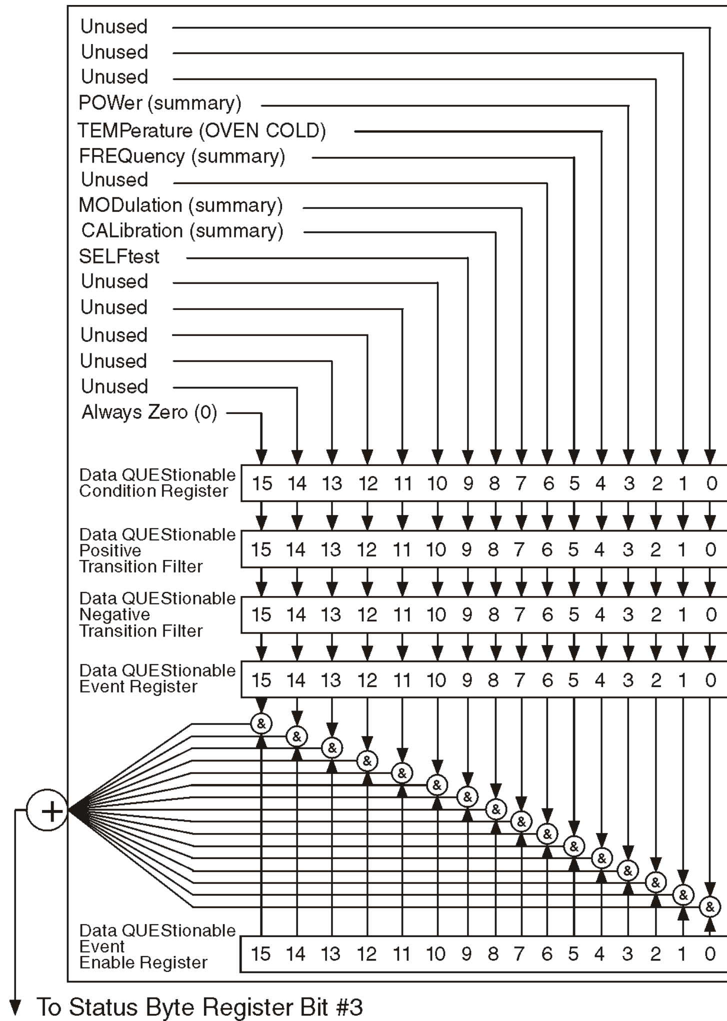
Query: `STATus:OPERation:ENABle?`

Response: Decimal value of the sum of the bits previously enabled with the `STATus:OPERation:ENABle <value>` command.

Data Questionable Status Group

NOTE Some of the bits in this status group do not apply to the E8663B and returns zero when queried. Other bits have changed state content. See Table 4-6 on page 143 for more information.

The Data Questionable Status Group is used to determine the specific event that set bit 3 in the Status Byte Register. This group consists of the Data Questionable Condition Register, the Data Questionable Transition Filters (negative and positive), the Data Questionable Event Register, and the Data Questionable Event Enable Register.



Data Questionable Condition Register

The Data Questionable Condition Register continuously monitors the hardware and firmware status of the signal generator. Condition registers are read only.

Table 4-6 Data Questionable Condition Register Bits

Bit	Description
0, 1, 2	**Unused**. These bits are always set to 0.
3	**Power (summary)**. This is a summary bit taken from the QUEStionable:POWer register. A 1 in this bit position indicates that one of the following may have happened: The ALC (Automatic Leveling Control) is unable to maintain a leveled RF output power (i.e., ALC is UNLEVELED), the reverse power protection circuit has been tripped. See the “Data Questionable Power Status Group” on page 145 for more information.
4	**Temperature (OVEN COLD)**. A 1 in this bit position indicates that the internal reference oscillator (reference oven) is cold.
5	**Frequency (summary)**. This is a summary bit taken from the QUEStionable:FREQuency register. A 1 in this bit position indicates that one of the following may have happened: synthesizer PLL unlocked, 10 MHz reference VCO PLL unlocked, 1 GHz reference unlocked, sampler, YO loop unlocked or baseband 1 unlocked. For more information, see the “Data Questionable Frequency Status Group” on page 148.
6	**Unused**. This bit is always set to 0.
7	**Modulation (summary)**. This is a summary bit taken from the QUEStionable:MODulation register. A 1 in this bit position indicates that one of the following may have happened: modulation source 1 underrange, modulation source 1 overrange, modulation source 2 underrange, modulation source 2 overrange, or modulation uncalibrated. See the “Data Questionable Modulation Status Group” on page 151 for more information.
8	**Calibration (summary)**. This is a summary bit taken from the QUEStionable:CALibration register. A 1 in this bit position indicates that an error has occurred in the DCFM/DCΦM zero calibration. See the “Data Questionable Calibration Status Group” on page 154 for more information.
9	**Self Test**. A 1 in this bit position indicates that a self-test has failed during power-up. Reset this bit by cycling the signal generator’s line power. *CLS will not clear this bit.
10–14	**Unused**. These bits are always set to 0.
15	**Always 0**.

Query: `STATus:QUEStionable:CONDition?`

Response: The *decimal* sum of the bits set to 1

Example: The decimal value 520 is returned. The decimal sum = 512 (bit 9) + 8 (bit 3).

Data Questionable Transition Filters (negative and positive)

The Data Questionable Transition Filters specify which type of bit state changes in the condition register set corresponding bits in the event register. Changes can be positive (0 to 1) or negative (1 to 0).

Commands: STATus:QUEStionable:NTRansition <value> (negative transition), or

STATus:QUEStionable:PTRansition <value> (positive transition), where

<value> is the sum of the decimal values of the bits you want to enable.

Queries: STATus:QUEStionable:NTRansition?

STATus:QUEStionable:PTRansition?

Data Questionable Event Register

The Data Questionable Event Register latches transition events from the condition register as specified by the transition filters. Event registers are destructive read-only. Reading data from an event register clears the content of that register.

Query: STATus:QUEStionable[:EVENt]?

Data Questionable Event Enable Register

The Data Questionable Event Enable Register lets you choose which bits in the Data Questionable Event Register set the summary bit (bit 3 of the Status Byte Register) to 1.

Command: STATus:QUEStionable:ENABle <value> where <value> is the sum of the decimal values of the bits you want to enable.

Example: Enable bit 9 and bit 3 so that whenever either of those bits are set to 1, the Data Questionable Status summary bit of the Status Byte Register is set to 1. Send the command STAT:QUES:ENAB 520 (512 + 8).

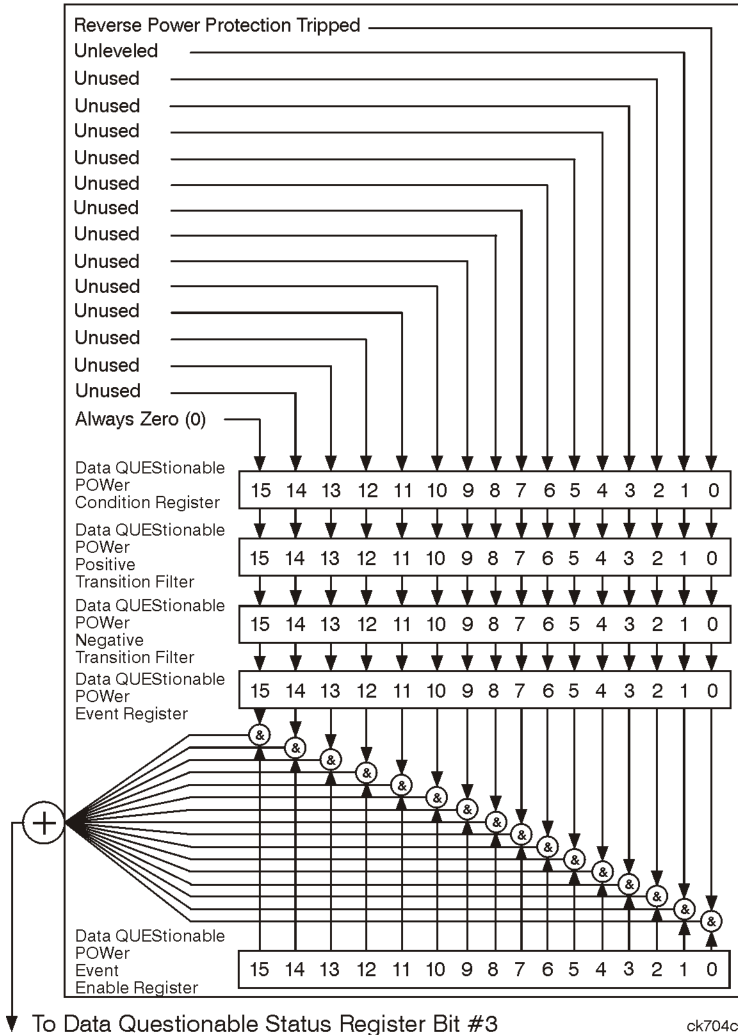
Query: STATus:QUEStionable:ENABle?

Response: Decimal value of the sum of the bits previously enabled with the STATus:QUEStionable:ENABle <value> command.

Data Questionable Power Status Group

NOTE Some of the bits in this status group do not apply to the E8663B and returns zero when queried. See Table 4-7 on page 146 for more information.

The Data Questionable Power Status Group is used to determine the specific event that set bit 3 in the Data Questionable Condition Register. This group consists of the Data Questionable Power Condition Register, the Data Questionable Power Transition Filters (negative and positive), the Data Questionable Power Event Register, and the Data Questionable Power Event Enable Register.



Data Questionable Power Condition Register

The Data Questionable Power Condition Register continuously monitors the hardware and firmware status of the signal generator. Condition registers are read only.

Table 4-7 Data Questionable Power Condition Register Bits

Bit	Description
0	**Reverse Power Protection Tripped**. A 1 in this bit position indicates that the reverse power protection (RPP) circuit has been tripped. There is no output in this state. Any conditions that may have caused the problem should be corrected. Reset the RPP circuit by sending the remote SCPI command: OUTput:PROTection:CLEar. Resetting the RPP circuit bit, resets this bit to 0.
1	**Unleveled**. A 1 in this bit position indicates that the output leveling loop is unable to set the output power.
2–14	**Unused**. These bits are always set to 0.
15	**Always 0.**

Query: `STATus:QUEStionable:POWer:CONDition?`

Response: The *decimal* sum of the bits set to 1.

Data Questionable Power Transition Filters (negative and positive)

The Data Questionable Power Transition Filters specify which type of bit state changes in the condition register set corresponding bits in the event register. Changes can be positive (0 to 1) or negative (1 to 0).

Commands: `STATus:QUEStionable:POWer:NTRansition <value>` (negative transition), or
`STATus:QUEStionable:POWer:PTRansition <value>` (positive transition), where
`<value>` is the sum of the decimal values of the bits you want to enable.

Queries: `STATus:QUEStionable:POWer:NTRansition? STATus:QUEStionable:POWer:PTRansition?`

Data Questionable Power Event Register

The Data Questionable Power Event Register latches transition events from the condition register as specified by the transition filters. Event registers are destructive read-only. Reading data from an event register clears the content of that register.

Query: `STATus:QUEStionable:POWer[:EVENt]?`

Data Questionable Power Event Enable Register

The Data Questionable Power Event Enable Register lets you choose which bits in the Data Questionable Power Event Register set the summary bit (bit 3 of the Data Questionable Condition Register) to 1.

Command: `STATus:QUEStionable:POWer:ENABle <value>` where `<value>` is the sum of the decimal values of the bits you want to enable

Example: Enable bit 3 and bit 2 so that whenever either of those bits are set to 1, the Data Questionable Power summary bit of the Data Questionable Condition Register is set to 1. Send the command `STAT:QUES:POW:ENAB 520` (8 + 4).

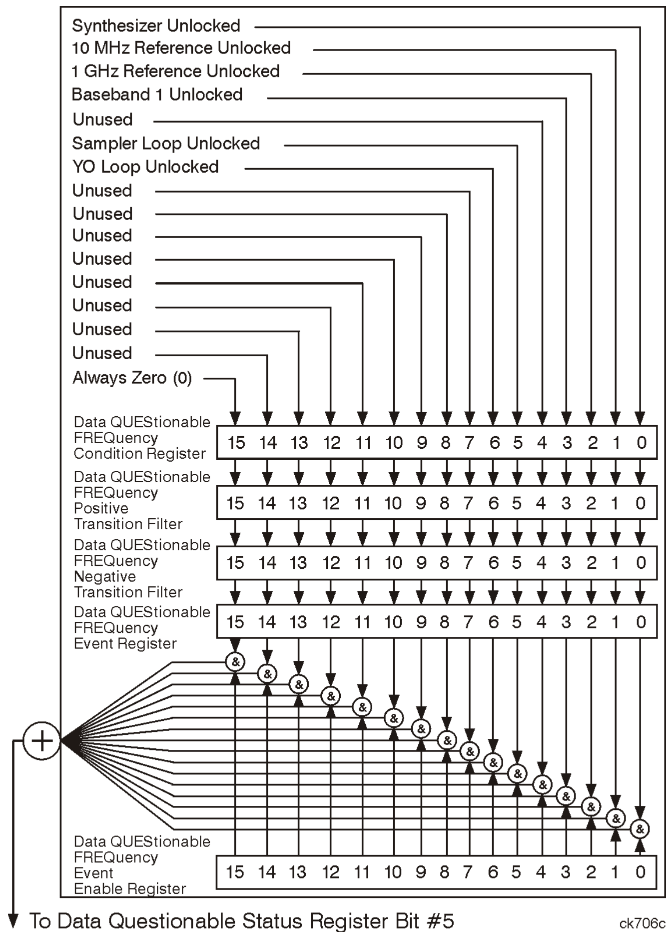
Query: `STATus:QUEStionable:POWer:ENABle?`

Response: Decimal value of the sum of the bits previously enabled with the `STATus:QUEStionable:POWer:ENABle <value>` command.

Data Questionable Frequency Status Group

NOTE Some bits in this status group do not apply to the E8663B and returns zero when queried. See Table 4-8 on page 149 for more information.

The Data Questionable Frequency Status Group is used to determine the specific event that set bit 5 in the Data Questionable Condition Register. This group consists of the Data Questionable Frequency Condition Register, the Data Questionable Frequency Transition Filters (negative and positive), the Data Questionable Frequency Event Register, and the Data Questionable Frequency Event Enable Register.



Data Questionable Frequency Condition Register

The Data Questionable Frequency Condition Register continuously monitors the hardware and firmware status of the signal generator. Condition registers are read-only.

Table 4-8 Data Questionable Frequency Condition Register Bits

Bit	Description
0	**Synth. Unlocked**. A 1 in this bit position indicates that the synthesizer is unlocked.
1	**10 MHz Ref Unlocked**. A 1 in this bit position indicates that the 10 MHz reference signal is unlocked.
2	**1 GHz Ref Unlocked**. A 1 in this bit position indicates that the 1 GHz reference signal is unlocked.
3	**Unused.** This bit is always set to 0.
4	**Unused**. This bit is always set to 0.
5	**Sampler Loop Unlocked**. A 1 in this bit position indicates that the sampler loop is unlocked.
6–14	**Unused**. These bits are always set to 0.
15	**Always 0**.

Query: `STATus:QUEStionable:FREQuency:CONDition?`

Response: The *decimal* sum of the bits set to 1.

Data Questionable Frequency Transition Filters (negative and positive)

Specifies which types of bit state changes in the condition register set corresponding bits in the event register. Changes can be positive (0 to 1) or negative (1 to 0).

Commands: `STATus:QUEStionable:FREQuency:NTRansition <value>` (negative transition) or `STATus:QUEStionable:FREQuency:PTRansition <value>` (positive transition) where `<value>` is the sum of the decimal values of the bits you want to enable.

Queries: `STATus:QUEStionable:FREQuency:NTRansition?`
`STATus:QUEStionable:FREQuency:PTRansition?`

Data Questionable Frequency Event Register

Latches transition events from the condition register as specified by the transition filters. Event registers are destructive read-only. Reading data from an event register clears the content of that register.

Query: `STATus:QUEStionable:FREQuency[:EVENt]?`

Data Questionable Frequency Event Enable Register

Lets you choose which bits in the Data Questionable Frequency Event Register set the summary bit (bit 5 of the Data Questionable Condition Register) to 1.

Command: `STATus:QUEStionable:FREQuency:ENABle <value>`, where `<value>` is the sum of the decimal values of the bits you want to enable.

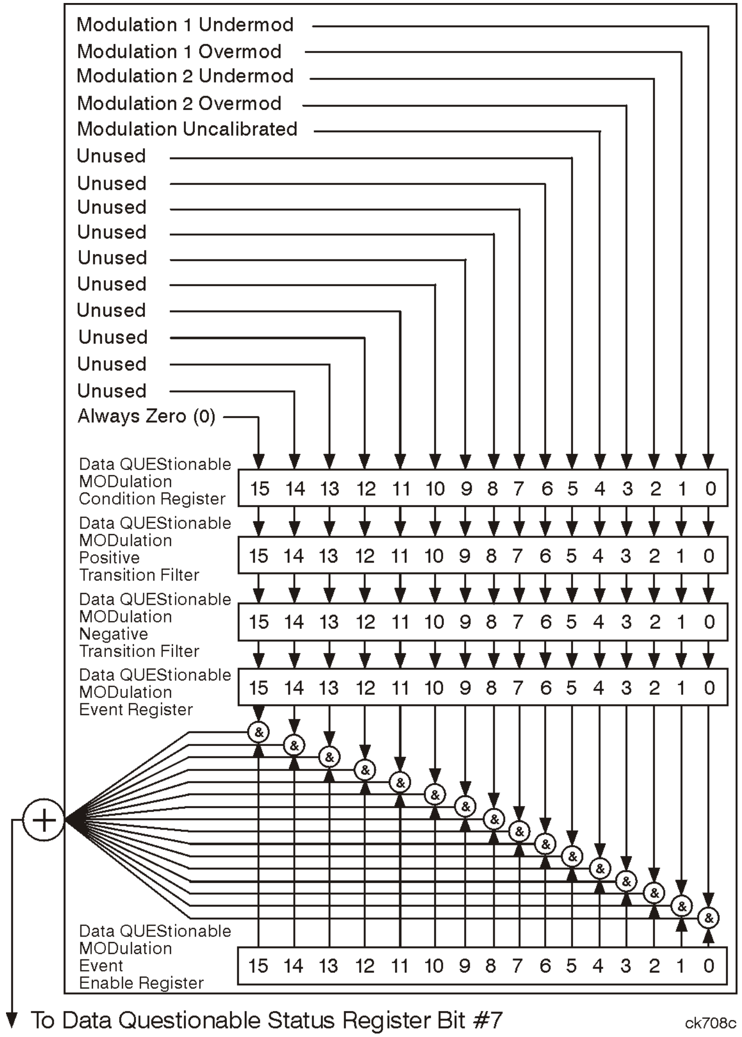
Example: Enable bit 4 and bit 3 so that whenever either of those bits are set to 1, the Data Questionable Frequency summary bit of the Data Questionable Condition Register is set to 1. Send the command `STAT:QUES:FREQ:ENAB 520` (16 + 8).

Query: `STATus:QUEStionable:FREQuency:ENABle?`

Response: Decimal value of the sum of the bits previously enabled with the `STATus:QUEStionable:FREQuency:ENABle <value>` command.

Data Questionable Modulation Status Group

The Data Questionable Modulation Status Group is used to determine the specific event that set bit 7 in the Data Questionable Condition Register. This group consists of the Data Questionable Modulation Condition Register, the Data Questionable Modulation Transition Filters (negative and positive), the Data Questionable Modulation Event Register, and the Data Questionable Modulation Event Enable Register.



Data Questionable Modulation Condition Register

The Data Questionable Modulation Condition Register continuously monitors the hardware and firmware status of the signal generator. Condition registers are read-only.

Table 4-9 Data Questionable Modulation Condition Register Bits

Bit	Description
0	**Modulation 1 Undermod**. A 1 in this bit position indicates that the External 1 input, ac coupling on, is less than 0.97 volts.
1	**Modulation 1 Overmod**. A 1 in this bit position indicates that the External 1 input, ac coupling on, is more than 1.03 volts.
2	**Modulation 2 Undermod**. A 1 in this bit position indicates that the External 2 input, ac coupling on, is less than 0.97 volts.
3	**Modulation 2 Overmod**. A 1 in this bit position indicates that the External 2 input, ac coupling on, is more than 1.03 volts.
4	**Modulation Uncalibrated**. A 1 in this bit position indicates that modulation is uncalibrated.
5–14	**Unused**. This bit is always set to 0.
15	**Always 0**.

Query: `STATus:QUEStionable:MODulation:CONDition?`

Response: The *decimal* sum of the bits set to 1

Data Questionable Modulation Transition Filters (negative and positive)

The Data Questionable Modulation Transition Filters specify which type of bit state changes in the condition register set corresponding bits in the event register. Changes can be positive (0 to 1) or negative (1 to 0).

Commands: `STATus:QUEStionable:MODulation:NTRansition <value>` (negative transition), or

`STATus:QUEStionable:MODulation:PTRansition <value>` (positive transition), where `<value>` is the sum of the decimal values of the bits you want to enable.

Queries: `STATus:QUEStionable:MODulation:NTRansition?`

`STATus:QUEStionable:MODulation:PTRansition?`

Data Questionable Modulation Event Register

The Data Questionable Modulation Event Register latches transition events from the condition register as specified by the transition filters. Event registers are destructive read-only. Reading data from an event register clears the content of that register.

Query: `STATus:QUEStionable:MODulation[:EVENt]?`

Data Questionable Modulation Event Enable Register

The Data Questionable Modulation Event Enable Register lets you choose which bits in the Data Questionable Modulation Event Register set the summary bit (bit 7 of the Data Questionable Condition Register) to 1.

Command: `STATus:QUEStionable:MODulation:ENABle <value>` where `<value>` is the sum of the decimal values of the bits you want to enable.

Example: Enable bit 4 and bit 3 so that whenever either of those bits are set to 1, the Data Questionable Modulation summary bit of the Data Questionable Condition Register is set to 1. Send the command `STAT:QUES:MOD:ENAB 520` (16 + 8).

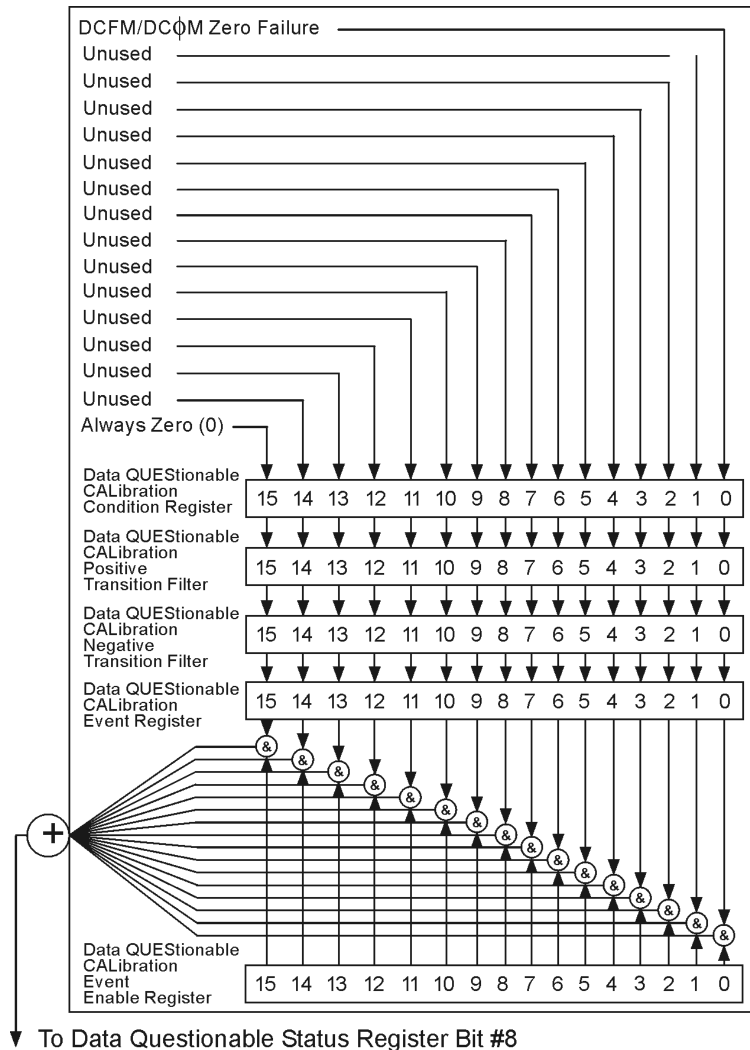
Query: `STATus:QUEStionable:MODulation:ENABle?`

Response: Decimal value of the sum of the bits previously enabled with the `STATus:QUEStionable:MODulation:ENABle <value>` command.

Data Questionable Calibration Status Group

NOTE Some bits in this status group do not apply to the E8663B, and return zero when queried. See Table 4-10 on page 155 for more information.

The Data Questionable Calibration Status Group is used to determine the specific event that set bit 8 in the Data Questionable Condition Register. This group consists of the Data Questionable Calibration Condition Register, the Data Questionable Calibration Transition Filters (negative and positive), the Data Questionable Calibration Event Register, and the Data Questionable Calibration Event Enable Register.



Data Questionable Calibration Condition Register

The Data Questionable Calibration Condition Register continuously monitors the calibration status of the signal generator. Condition registers are read only.

Table 4-10 Data Questionable Calibration Condition Register Bits

Bit	Description
0	**DCFM/DCΦM Zero Failure**. A 1 in this bit position indicates that the DCFM/DCΦM zero calibration routine has failed. This is a critical error. The output of the source has no validity until the condition of this bit is 0.
1–14	**Unused**. These bits are always set to 0.
15	**Always 0**.

Query: `STATus:QUEStionable:CALibration:CONDition?`

Response: The *decimal* sum of the bits set to 1.

Data Questionable Calibration Transition Filters (negative and positive)

The Data Questionable Calibration Transition Filters specify which type of bit state changes in the condition register set corresponding bits in the event register. Changes can be positive (0 to 1) or negative (1 to 0).

Commands: `STATus:QUEStionable:CALibration:NTRansition <value>` (negative transition), or
`STATus:QUEStionable:CALibration:PTRansition <value>` (positive transition), where `<value>` is the sum of the decimal values of the bits you want to enable.

Queries: `STATus:QUEStionable:CALibration:NTRansition?`
`STATus:QUEStionable:CALibration:PTRansition?`

Data Questionable Calibration Event Register

The Data Questionable Calibration Event Register latches transition events from the condition register as specified by the transition filters. Event registers are destructive read-only. Reading data from an event register clears the content of that register.

Query: `STATus:QUEStionable:CALibration[:EVENt]?`

Data Questionable Calibration Event Enable Register

The Data Questionable Calibration Event Enable Register lets you choose which bits in the Data Questionable Calibration Event Register set the summary bit (bit 8 of the Data Questionable Condition register) to 1.

Command:	`STATus:QUEStionable:CALibration:ENABle <value>`, where `<value>` is the sum of the decimal values of the bits you want to enable.
Example:	Enable bit 1 and bit 0 so that whenever either of those bits are set to 1, the Data Questionable Calibration summary bit of the Data Questionable Condition Register is set to 1. Send the command `STAT:QUES:CAL:ENAB 520` (2 + 1).
Query:	`STATus:QUEStionable:CALibration:ENABle?`
Response:	Decimal value of the sum of the bits previously enabled with the `STATus:QUEStionable:CALibration:ENABle <value>` command.

5 Creating and Downloading User-Data Files

NOTE The following sections and procedures contain remote SCPI commands. For front panel key commands, refer to the *User's Guide* or to the Key help in the signal generator.

- "Save and Recall Instrument State Files" on page 158
- "Download User Flatness Corrections Using C++ and VISA" on page 169

Save and Recall Instrument State Files

The signal generator can save instrument state settings to memory. An instrument state setting includes any instrument state that does not survive a signal generator preset or power cycle such as frequency, amplitude, attenuation, and other user–defined parameters. The instrument state settings are saved in memory and organized into sequences and registers. There are 10 sequences with 100 registers per sequence available for instrument state settings. These instrument state files are stored in the USER/STATE directory.

The save function does not store data such as arb formats, table entries, list sweep data and so forth. Use the store commands or store softkey functions to store these data file types to the signal generator's memory catalog. The save function will save a reference to the data file name associated with the instrument state.

Before saving an instrument state that has a data file associated with it, store the data file. For example, if you are editing a multitone arb format, store the multitone data to a file in the signal generator's memory catalog (multitone files are stored in the USER/MTONE directory). Then save the instrument state associated with that data file. The settings for the signal generator such as frequency and amplitude and a reference to the multitone file name will be saved in the selected sequence and register number. Refer to the signal generator's *User's Guide*, the *Key Reference*, or the signal generator's Help hardkey for more information on the save and recall functions.

Save and Recall SCPI Commands

The following command sequence saves the current instrument state, using the *SAV command, in sequence 1, register 01. A comment is then added to the instrument state.

```
*SAV 01,1
:MEM:STAT:COMM 01,1, "Instrument state comment"
```

If there is a data file associated with the instrument state, there will be a file name reference saved along with the instrument state. However, the data file must be stored in the signal generator's memory catalog as the *SAV command does not save data files. For more information on storing file data such as modulation formats, arb setups, and table entries refer to the Storing Files to the Memory Catalog section in the signal generator's *User's Guide*.

NOTE File names are referenced when an instrument state is saved, but a file will NOT be stored with the save function.

The recall function will recall the saved instrument state. If there is a data file associated with the instrument state, the file will be loaded along with the instrument state. The following command recalls the instrument state saved in sequence 1, register 01.

```
*RCL 01,1
```

Save and Recall Programming Example Using VISA and C#

The following programming example uses VISA and C# to save and recall signal generator instrument states. Instruments states are saved to and recalled from your computer. This console program prompts the user for an action: Backup State Files, Restore State Files, or Quit.

The Backup State Files choice reads the signal generator's state files and stores it on your computer in the same directory where the State_Files.exe program is located. The Restore State Files selection downloads instrument state files, stored on your computer, to the signal generator's State directory. The Quit selection exists the program. The figure below shows the console interface and the results obtained after selecting the Restore State Files operation.

The program uses VISA library functions. Refer to the Agilent VISA User's Manual available on Agilent's website: *http:\\www.agilent.com* for more information on VISA functions.

The program listing for the State_Files.cs program is shown below. It is available on the CD–ROM in the programming examples section under the same name.

```
C:\WINNT\Microsoft.NET\Framework\v1.1.4322\State_Files1.exe
1) Backup state files
2) Restore state files
3) Quit
Enter 1,2,or 3. Your choice: 2
Restoring sequence #0, register #01
Restoring sequence #0, register #02
Restoring sequence #0, register #03
Restoring sequence #0, register #04
Restoring sequence #0, register #05
Restoring sequence #0, register #06
Restoring sequence #0, register #07
Restoring sequence #0, register #08
Restoring sequence #1, register #00
Restoring sequence #1, register #01
Restoring sequence #1, register #02
Restoring sequence #1, register #03
Restoring sequence #1, register #04
Restoring sequence #1, register #05
1) Backup state files
2) Restore state files
3) Quit
Enter 1,2,or 3. Your choice:
```

C# and Microsoft .NET Framework

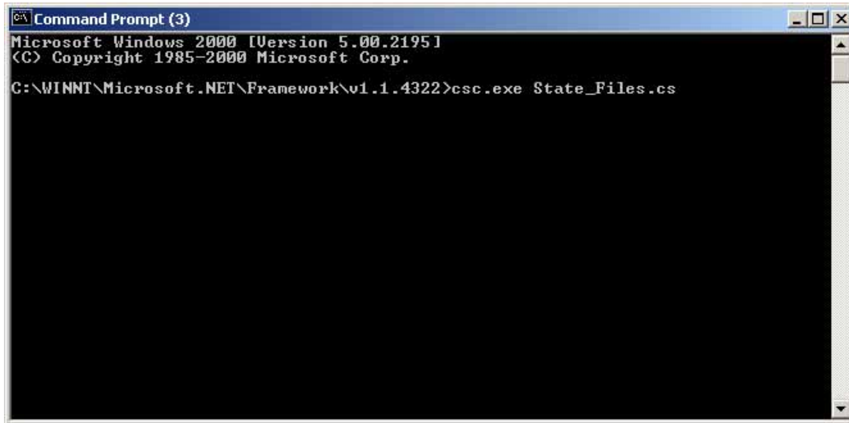
The Microsoft .NET Framework is a platform for creating Web Services and applications. There are three components of the .NET Framework: the common language runtime, class libraries, and Active Server Pages, called ASP.NET. Refer to the Microsoft website for more information on the .NET Framework.

The .NET Framework must be installed on your computer before you can run the State_Files program. The framework can be downloaded from the Microsoft website and then installed on your computer.

Perform the following steps to run the State_Files program.

1. Copy the State_Files.cs file from the CD–ROM programming examples section to the directory where the .NET Framework is installed.
2. Change the TCPIP0 address in the program from TCPIP0::000.000.000.000 to your signal generator's address.
3. Save the file using the .cs file name extension.
4. Run the Command Prompt program. Start > Run > "cmd.exe". Change the directory for the command prompt to the location where the .NET Framework was installed.
5. Type `csc.exe State_Files.cs` at the command prompt and then press the Enter key on the keyboard to run the program. The following figure shows the command prompt interface.

6. Follow the prompts in the program to save and recall signal generator instrument states.



The State_Files.cs program is listed below. You can copy this program from the examples directory on the signal generator's CD–ROM.

NOTE The *State_Files.cs* example uses the ESG in the programming code but can be used with the E8663B.

```
//****************************************************************************************
// FileName: State_Files.cs
//
// This C# example code saves and recalls signal generator instrument states. The saved
// instrument state files are written to the local computer directory computer where the
// State_Files.exe is located. This is a console application that uses DLL importing to
// allow for calls to the unmanaged Agilent IO Library VISA DLL.
//
// The Agilent VISA library must be installed on your computer for this example to run.
// Important: Replace the visaOpenString with the IP address for your signal generator.
//
//****************************************************************************************
using System;
using System.IO;
using System.Text;
using System.Runtime.InteropServices;
using System.Collections;
using System.Text.RegularExpressions;
```

```
namespace State_Files

{
    class MainApp
    {
        // Replace the visaOpenString variable with your instrument's address.

        static public string visaOpenString = "TCPIP0::000.000.000.000"; //"GPIB0::19";
      //"TCPIP0::ESG3::INSTR";


public const uint DEFAULT_TIMEOUT = 30 * 1000;// Instrument timeout 30 seconds.
        public const int MAX_READ_DEVICE_STRING = 1024; // Buffer for string data reads.
      public const int TRANSFER_BLOCK_SIZE = 4096;// Buffer for byte data.

        // The main entry point for the application.

        [STAThread]

static void Main(string[] args)
        {

    uint defaultRM;// Open the default VISA resource manager
    if (VisaInterop.OpenDefaultRM(out defaultRM) == 0) // If no errors, proceed.
      {
         uint device;
         // Open the specified VISA device: the signal generator
     if (VisaInterop.Open(defaultRM, visaOpenString,VisaAccessMode.NoLock,
                          DEFAULT_TIMEOUT, out device) == 0)
        // if no errors proceed.
       {
       bool quit = false;
         while (!quit)// Get user input
          {
          Console.Write("1) Backup state files\n" +
                        "2) Restore state files\n" +
                        "3) Quit\nEnter 1,2,or 3. Your choice: ");
             string choice = Console.ReadLine();
           switch (choice)
                {
                 case "1":
                 {
               BackupInstrumentState(device);  // Write instrument state
               break;                    // files to the computer
```

```
                }
                            case "2":
                 {
               RestoreInstrumentState(device); // Read instrument state
               break;// files to the sig gen
                 }
             case "3":
                 {
              quit = true;
              break;
                 }
              default:
                 {
              break;
                 }
                }
              }
          VisaInterop.Close(device);// Close the device
              }
           else
              {
           Console.WriteLine("Unable to open " + visaOpenString);
          }
               VisaInterop.Close(defaultRM);   // Close the default resource manager
          }
           else
              {
        Console.WriteLine("Unable to open the VISA resource manager");
          }
        }

       /* This method restores all the sequence/register state files located in
        the local directory (identified by a ".STA" file name extension)
        to the signal generator.*/

static public void RestoreInstrumentState(uint device)
   {
    DirectoryInfo di = new DirectoryInfo(".");// Instantiate object class
    FileInfo[] rgFiles = di.GetFiles("*.STA");  // Get the state files
    foreach(FileInfo fi in rgFiles)
      {
      Match m = Regex.Match(fi.Name, @"^(\d)_(\d\d)");
```

```
      if (m.Success)
        {
        string sequence = m.Groups[1].ToString();
        string register = m.Groups[2].ToString();
        Console.WriteLine("Restoring sequence #" + sequence +
                              ", register #" + register);

/* Save the target instrument's current state to the specified sequence/
register pair. This ensures the index file has an entry for the specified
sequence/register pair. This workaround will not be necessary in future
revisions of firmware.*/

        WriteDevice(device,"*SAV " + register + ", " + sequence + "\n",
                        true); // << on SAME line!
        // Overwrite the newly created state file with the state
        // file that is being restored.
        WriteDevice(device, "MEM:DATA \"/USER/STATE/" + m.ToString() + "\",",
                        false); // << on SAME line!
        WriteFileBlock(device, fi.Name);
        WriteDevice(device, "\n", true);
                  }
              }
        }

/* This method reads out all the sequence/register state files from the signal
generator and stores them in your computer's local directory with a ".STA"
extension */

static public void BackupInstrumentState(uint device)
   {
   // Get the memory catalog for the state directory
      WriteDevice(device, "MEM:CAT:STAT?\n", false);
      string catalog = ReadDevice(device);
      /* Match the catalog listing for state files which are named
         (sequence#)_(register#)  e.g.  0_01, 1_01, 2_05*/
      Match m = Regex.Match(catalog, "\"(\\d_\\d\\d),");
      while (m.Success)
         {
        // Grab the matched filename from the regular expresssion
        string nextFile = m.Groups[1].ToString();
       // Retrieve the file and store with a .STA extension
       // in the current directory
```

```
        Console.WriteLine("Retrieving state file: " + nextFile);
          WriteDevice(device, "MEM:DATA? \"/USER/STATE/" + nextFile + "\"\n", true);
        ReadFileBlock(device, nextFile + ".STA");
        // Clear newline
        ReadDevice(device);
        // Advance to next match in catalog string
        m = m.NextMatch();
         }
    }

/*  This method writes an ASCII text string (SCPI command) to the signal generator.
If the bool "sendEnd" is true, the END line character will be sent at the
conclusion of the write. If "sendEnd is false the END line will not be sent.*/

static public void WriteDevice(uint device, string scpiCmd, bool sendEnd)
    {
     byte[] buf = Encoding.ASCII.GetBytes(scpiCmd);
     if (!sendEnd) // Do not send the END line character
       {
      VisaInterop.SetAttribute(device, VisaAttribute.SendEndEnable, 0);
       }
     uint retCount;
     VisaInterop.Write(device, buf, (uint)buf.Length, out retCount);
     if (!sendEnd) // Set the bool sendEnd true.
       {
      VisaInterop.SetAttribute(device, VisaAttribute.SendEndEnable, 1);
       }
    }

// This method reads an ASCII string from the specified device
static public string ReadDevice(uint device)
    {
     string retValue = "";
     byte[] buf = new byte[MAX_READ_DEVICE_STRING]; // 1024 bytes maximum read
     uint retCount;
     if (VisaInterop.Read(device, buf, (uint)buf.Length -1, out retCount) == 0)
       {
       retValue = Encoding.ASCII.GetString(buf, 0, (int)retCount);
       }
     return retValue;
    }
```

```
/* The following method reads a SCPI definite block from the signal generator
and writes the contents to a file on your computer. The trailing
newline character is NOT consumed by the read.*/

static public void ReadFileBlock(uint device, string fileName)
   {
    // Create the new, empty data file.
    FileStream fs = new FileStream(fileName, FileMode.Create);
    // Read the definite block header:  #{lengthDataLength}{dataLength}
    uint retCount = 0;
    byte[] buf = new byte[10];
    VisaInterop.Read(device, buf, 2, out retCount);
    VisaInterop.Read(device, buf, (uint)(buf[1]-'0'), out retCount);
     uint fileSize = UInt32.Parse(Encoding.ASCII.GetString(buf, 0, (int)retCount));
    // Read the file block from the signal generator
    byte[] readBuf = new byte[TRANSFER_BLOCK_SIZE];
    uint bytesRemaining = fileSize;

    while (bytesRemaining != 0)
      {
       uint bytesToRead = (bytesRemaining < TRANSFER_BLOCK_SIZE) ?
       bytesRemaining : TRANSFER_BLOCK_SIZE;
       VisaInterop.Read(device, readBuf, bytesToRead, out retCount);
       fs.Write(readBuf, 0, (int)retCount);
       bytesRemaining -= retCount;
      }
    // Done with file
    fs.Close();
   }

/* The following method writes the contents of the specified file to the
specified file in the form of a SCPI definite block.  A newline is
NOT appended to the block and END is not sent at the conclusion of the
write.*/

static public void WriteFileBlock(uint device, string fileName)
   {
    // Make sure that the file exists, otherwise sends a null block
    if (File.Exists(fileName))
     {
     FileStream fs = new FileStream(fileName, FileMode.Open);
     // Send the definite block header: #{lengthDataLength}{dataLength}
```

```
        string fileSize = fs.Length.ToString();
        string fileSizeLength = fileSize.Length.ToString();
        WriteDevice(device, "#" + fileSizeLength + fileSize, false);
        // Don't set END at the end of writes
        VisaInterop.SetAttribute(device, VisaAttribute.SendEndEnable, 0);
        // Write the file block to the signal generator
        byte[] readBuf = new byte[TRANSFER_BLOCK_SIZE];
        int numRead = 0;
        uint retCount = 0;
        while ((numRead = fs.Read(readBuf, 0, TRANSFER_BLOCK_SIZE)) != 0)
           {
           VisaInterop.Write(device, readBuf, (uint)numRead, out retCount);
           }
          // Go ahead and set END on writes
        VisaInterop.SetAttribute(device, VisaAttribute.SendEndEnable, 1);
        // Done with file
        fs.Close();
        }
        else
        {
        // Send an empty definite block
        WriteDevice(device, "#10", false);
        }
       }
     }

// Declaration of VISA device access constants
public enum VisaAccessMode
    {
        NoLock = 0,
        ExclusiveLock = 1,
        SharedLock = 2,
        LoadConfig = 4
    }

// Declaration of VISA attribute constants
public enum VisaAttribute
    {
        SendEndEnable = 0x3FFF0016,
        TimeoutValue  = 0x3FFF001A
    }
```

```
// This class provides a way to call the unmanaged Agilent IO Library VISA C
// functions from the C# application

public class VisaInterop
    {
        [DllImport("agvisa32.dll", EntryPoint="viClear")]
        public static extern int Clear(uint session);

        [DllImport("agvisa32.dll", EntryPoint="viClose")]
        public static extern int Close(uint session);

        [DllImport("agvisa32.dll", EntryPoint="viFindNext")]
        public static extern int FindNext(uint findList, byte[] desc);

        [DllImport("agvisa32.dll", EntryPoint="viFindRsrc")]
        public static extern int FindRsrc(
            uint session,
            string expr,
            out uint findList,
            out uint retCnt,
            byte[] desc);

        [DllImport("agvisa32.dll", EntryPoint="viGetAttribute")]
public static extern int GetAttribute(uint vi, VisaAttribute attribute, out uint attrState);

        [DllImport("agvisa32.dll", EntryPoint="viOpen")]
        public static extern int Open(
            uint session,
            string rsrcName,
            VisaAccessMode accessMode,
            uint timeout,
            out uint vi);

        [DllImport("agvisa32.dll", EntryPoint="viOpenDefaultRM")]
        public static extern int OpenDefaultRM(out uint session);

        [DllImport("agvisa32.dll", EntryPoint="viRead")]
        public static extern int Read(
            uint session,
            byte[] buf,
            uint count,
            out uint retCount);
```

```
        [DllImport("agvisa32.dll", EntryPoint="viSetAttribute")]
public static extern int SetAttribute(uint vi, VisaAttribute attribute, uint attrState);

        [DllImport("agvisa32.dll", EntryPoint="viStatusDesc")]
        public static extern int StatusDesc(uint vi, int status, byte[] desc);

        [DllImport("agvisa32.dll", EntryPoint="viWrite")]
        public static extern int Write(
            uint session,
            byte[] buf,
            uint count,
            out uint retCount);
    }
}
```

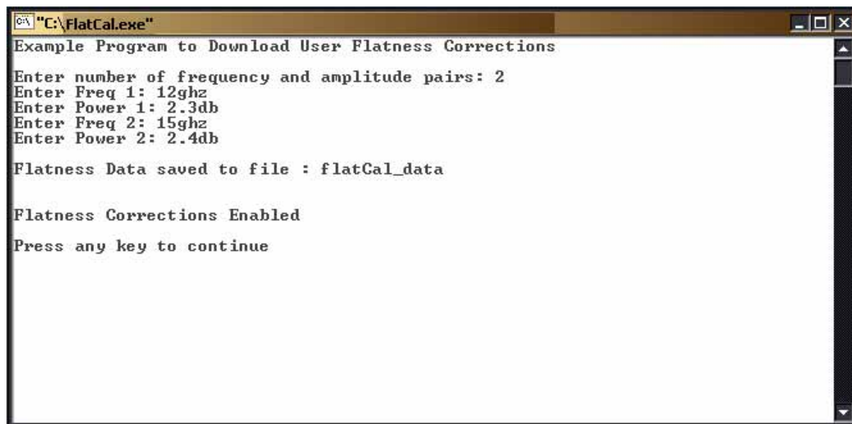
Download User Flatness Corrections Using C++ and VISA

This sample program uses C++ and the VISA libraries to download user–flatness correction values to the signal generator. The program uses the LAN interface but can be adapted to use the GPIB interface by changing the address string in the program.

You must include header files and resource files for library functions needed to run this program. Refer to "Running C++ Programs" on page 47 for more information.

The FlatCal program asks the user to enter a number of frequency and amplitude pairs. Frequency and amplitude values are entered by using the keyboard and displayed on in the console interface. The values are then downloaded to the signal generator and stored to a file named flatCal_data. The file is then loaded into the signal generator's memory catalog and corrections are turned on. The figure below shows the console interface and several frequency and amplitude values. Use the same format, shown in the figure below, for entering frequency and amplitude pairs (for example, 12ghz, 1.2db).

Figure 5-1 **FlatCal Console Application**



The program uses VISA library functions. The non–formatted viWrite VISA function is used to output data to the signal generator. Refer to the Agilent VISA User's Manual available on Agilent's website: *http:\\www.agilent.com* for more information on VISA functions.

The program listing for the FlatCal program is shown below. It is available on the CD–ROM in the programming examples section as flatcal.cpp.

```
//****************************************************************************************
// PROGRAM NAME:FlatCal.cpp
//
// PROGRAM DESCRIPTION:C++ Console application to input frequency and amplitude
// pairs and then download them to the signal generator.
//
// NOTE: You must have the Agilent IO Libraries installed to run this program.
//
// This example uses the LAN/TCPIP interface to download frequency and amplitude
// correction pairs to the signal generator. The program asks the operator to enter
// the number of pairs and allocates a pointer array listPairs[] sized to the number
// of pairs.The array is filled with frequency nextFreq[] and amplitude nextPower[]
// values entered from the keyboard.
//
//****************************************************************************************
// IMPORTANT: Replace the 000.000.000.000 IP address in the instOpenString declaration
// in the code below with the IP address of your signal generator.
//****************************************************************************************

#include <stdlib.h>
#include <stdio.h>
#include  "visa.h"
#include <string.h>

//   IMPORTANT:
//   Configure the following IP address correctly before compiling and running

char* instOpenString ="TCPIP0::000.000.000.000::INSTR";//your signal generator's IP address

const int MAX_STRING_LENGTH=20;//length of frequency and power strings
const int BUFFER_SIZE=256;//length of SCPI command string

int main(int argc, char* argv[])
{
    ViSession defaultRM, vi;
    ViStatus status = 0;

    status = viOpenDefaultRM(&defaultRM);//open the default resource manager

    //TO DO: Error handling here

    status = viOpen(defaultRM, instOpenString, VI_NULL, VI_NULL, &vi);
```

```
 if (status)//if any errors then display the error and exit the program
 {
     fprintf(stderr, "viOpen failed (%s)\n", instOpenString);
     return -1;
 }

printf("Example Program to Download User Flatness Corrections\n\n");
printf("Enter number of frequency and amplitude pairs: ");
 int num = 0;

 scanf("%d", &num);

 if (num > 0)
 {
     int lenArray=num*2;//length of the pairsList[] array. This array
     //will hold the frequency and amplitude arrays

     char** pairsList = new char* [lenArray]; //pointer array

     for (int n=0; n < lenArray; n++)//initialize the pairsList array
                                //pairsList[n]=0;

     for (int i=0; i < num; i++)
     {
       char* nextFreq = new char[MAX_STRING_LENGTH+1]; //frequency array
       char* nextPower = new char[MAX_STRING_LENGTH+1];//amplitude array
       //enter frequency and amplitude pairs i.e 10ghz .1db
       printf("Enter Freq %d: ", i+1);
       scanf("%s", nextFreq);
       printf("Enter Power %d: ",i+1);
       scanf("%s", nextPower);
       pairsList[2*i] = nextFreq;//frequency
       pairsList[2*i+1]=nextPower;//power correction
     }

 unsigned char str[256];//buffer used to hold SCPI command

 //initialize the signal generator's user flatness table
 sprintf((char*)str,":corr:flat:pres\n"); //write to buffer
 viWrite(vi, str,strlen((char*str),0);    //write to signal generator
 char c = ',';//comma separator for SCPI command
 for (int j=0; j< num; j++)   //download pairs to the signal generator
```

```
      {
       sprintf((char*)str,":corr:flat:pair %s %c %s\n",pairsList[2*j], c,
              pairsList[2*j+1]); // << on SAME line!
       viWrite(vi, str,strlen((char*)str),0);
     }
    //store the downloaded correction pairs to signal generator memory
    const char* fileName = "flatCal_data";//user flatness file name
    //write the SCPI command to the buffer str
     sprintf((char*)str, ":corr:flat:store \"%s\"\n", fileName);//write to buffer
    viWrite(vi,str,strlen((char*)str),0);//write the command to the signal generator
    printf("\nFlatness Data saved to file : %s\n\n", fileName);

   //load corrections
   sprintf((char*)str,":corr:flat:load \"%s\"\n", fileName); //write to buffer
   viWrite(vi,str,strlen((char*)str),0); //write command to the signal generator
   //turn on corrections
   sprintf((char*)str, ":corr on\n");
   viWrite(vi,str,strlen((char*)str),0");
   printf("\nFlatness Corrections Enabled\n\n");
   for (int k=0; k< lenArray; k++)
     {
      delete [] pairsList[k];//free up memory
     }
      delete [] pairsList;//free up memory
    }

    viClose(vi);//close the sessions
    viClose(defaultRM);

    return 0;
}
```

Index

Symbols

.NET framework, 158

A

abort function, 52, 53
address
  GPIB address, 18
  IP address, 23
Agilent
  BASIC, *See* HP BASIC
  e8663b
    global settings, configuring, 12
    setting GPIB address, 18
    web server, on, 9
  IO Libraries, 5
    Version J, 30
    Version M, 5, 27, 30
  IO Libraries Suite, 27
  SICL, 6, 20, 52
  VISA, 6, 20, 38, 52
  VISA COM Resource Manager 1.0, 48
Agilent I/O libraries
  See IO libraries
Agilent IO libraries
  See IO libraries
Agilent IO Libraries Suite, 4
Agilent VISA, 7
ASCII
  data, 55
AUXILIARY INTERFACE, *See* RS-232

B

BASIC
  ABORT, 52
  CLEAR, 55
  ENTER, 56
  LOCAL, 54, 55
  LOCAL LOCKOUT, 54
  OUTPUT, 55
  REMOTE, 53
  See HP BASIC
bit status, monitoring, 130
bit values, 129

C

C, 79
  AC-coupled FM signals, generating externally applied, 69
  CW signals, generating, 67
  data questionable status register, reading, 79
  FM signals, generating internally applied, 71
  reading the service request interrupt, 83
  Sockets LAN, programming, 91
C, 79 (*continued*)
  states, saving and recalling, 77
C and VISA
  GPIB queries, 65
  GPIB, interface check, 58
C#
  programming examples, 48
  remote control, 7
  VISA, example, 158
C++
  programming examples, 47
  VISA, generating a step-swept signal, 73
C++ and VISA
  generating a step-swept signal, 73
C/C++, 7
clear
  command, 55
  function, 55
CLS command, 132
command prompt, 26, 115
commands
  e8663b, 12
  GPIB, 52, 53, 54, 55, 56
computer interface, 3
condition registers, description, 136
configuring, VXI-11, 30
connection
  wizard, 4
connection expert, 4
controller, 19
csc.exe, 158
CW signals, generate using VISA and C, 67

D

data questionable filters
  calibration transition, 155
  frequency transition, 149
  modulation transition, 152
  power transition, 146
  transition, 144
data questionable groups
  calibration status, 154
  frequency status, 148
  modulation status, 151
  power status, 145
  status, 142
data questionable registers
  calibration condition, 155
  calibration event, 155
  calibration event enable, 156
  condition, 143
  event, 144
  event enable, 144

Index

data questionable registers (*continued*)
frequency condition, 149
frequency event, 149
frequency event enable, 150
modulation condition, 152
modulation event, 153
modulation event enable, 153
power condition, 146
power event, 147
power event enable, 147
data questionable status register, reading, 79
using VISA, 79
developing programs, 46
device, add, 6
DHCP, 8, 25
DNS, 26
DOS command prompt, 32
download
user flatness, 158
waveform data
user-data files, using, 157
download libraries, 6, 7

E

edit visa config, 6
EnableRemote, 53
enter function, 56
errors, 13, 27
ESE commands, 132
event enable register
description, 136
event registers
description, 136
examples
save and recall, 158
Telnet, 36
externally applied AC-coupled FM signals
generate, using C, 69
generate, using VISA, 69

F

file transfer, 36
files
error messages, 13
filters
See also transition filters
negative transition, description, 136
positive transition, description, 136
firmware status, monitoring, 130
flatness corrections, 169
FTP
using, 36

G

Getting Started Wizard, 18
global settings
e8663b, 12
GPIB
address, 18, 87
configuration, 18
controller, 19
interface, 3, 18
interface cards, 16, 50
IO libraries, 6
listener, 19
overview, 16, 50
program examples, 20, 52, 58, 65
SCPI commands, 19
talker, 19
troubleshooting, 18
using VISA and C, 58
verifying operation, 18
GPIB address
e8663b, setting, 18

H

hardware layers
remote programming, 2
hardware status, monitoring, 130
help mode
setting
e8663b, 12
hostname, 23, 87
hostname, setting, 25
e8663b, 24
e8663b menus, 23
HP BASIC, 7
HP Basic
I/O library, 38
local lockout, 59
queries, 62
RS-232
control, 38
queries, 44, 121
HyperTerminal, 41

I

I/O libraries
See IO libraries
iabort, 52
ibloc, 54, 55
ibstop, 52
ibwrt, 55
iclear, 55
IEEE standard, 16, 50
igpibllo, 54

iloc, 54
include files, 169
instrument communication, 5
instrument state files, 158
instrument status, monitoring, 126
interactive IO, 27
interactive io, 4
interface
  cards, 16, 50
  GPIB, 18
  LAN, 3
  RS-232, 3
internally applied FM signals
  generate, using C, 71
  generate, using VISA, 71
IO Config, 4, 5, 6, 28
IO interface, 5
IO libraries, 2, 4, 6, 16, 19, 27, 28, 38, 50
IP address, 23
  LAN interface, 23
  setting, 23, 25
  setting e8663b, 24
iremote, 53

J

JAVA, 49, 115
Java, 7
  example, 49, 115

L

LabView, 7
LAN
  DHCP configuration, 25
  hostname, 23
  interface, 3
  IO libraries, 7
  manual configuration, 23
  overview, 22
  program examples, 49, 87, 115, 116
  queries using sockets, 94
  sockets, 87
  sockets LAN, 22
  Telnet, 32
  verifying operation, 26
  VXI-11, 87
    examples, using, 87
    interface protocols, 22
    perl, using, 116
    programming examples, LAN, 87
    sockets, programming, 49, 115
LAN config, 28
LAN configuration
  e8663b, 24
LAN configuration (*continued*)
  menu, e8663b, 23, 25
  web server, 8
LAN programming, 49, 115
  using JAVA, 49, 115
libraries, 19
  GPIB I/O libraries, selecting, 6
  IO, Agilent, 2, 4
  RS-232, 38
  selecting, for computer, 7
list, error messages, 13
listener, 19
local
  echo, telnet, 35
  function, 54
local lockout
  HP Basic, using, 59
local lockout function, 54

M

manual operation, 53
MATLAB, 7
  programming, introduction, 7
Microsoft .NET Framework
  overview, 159
MS-DOS Command Prompt, 26, 32

N

National Instruments
  NI-488.2, 20, 52
  VISA, 6, 7, 20, 38, 52
negative transition filter, description, 136
NI libraries
  SICL
    GPIB I/O libraries, selecting, 6
NI-488.2
  EnableRemote, 53
  functions, 6
  GPIB I/O libraries, selecting, 6
  iblcr, 55
  ibloc, 54, 55
  ibrd, 56
  ibstop, 52
  ibwrt, 55
  LAN I/O libraries, selecting, 7
  queries using C++, 63
  RS-232 I/O libraries, selecting, 38
  SetRWLS, 54
  VISA, 6, 38

O

OPC commands, 132
output command, 55

Index

output function, 55

P

PCI- GPIB, 20, 52
PERL
  example, 116
ping program, 26
ping responses, 27
polling method (status registers), 130
ports, 91
positive transition filter, description, 136
programming examples
  C#, 48, 159
  C++, 47
  RS- 232, queries using VISA and C, 44, 122
  RS- 232, using VISA and C, 43, 119
  using, 46
  using GPIB, 20, 52, 58, 65
  using LAN, 49, 87, 115, 116
  using RS- 232, 43, 118
  VXI- 11, 87

Q

queries
  HP Basic, using, 62
queue, error, 13

R

recall states, 158
register system overview, 126
registers
  See also status registers
  condition, description, 136
  data questionable
    condition, 143
    event, 144
    event enable, 144
  data questionable calibration
    condition, 155
    event, 155
    event enable, 156
  data questionable frequency
    condition, 149
    event, 149
    event enable, 150
  data questionable modulation
    condition, 152
    event, 153
    event enable, 153
  data questionable power
    condition, 146
    event, 147
    event enable, 147

registers (*continued*)
  e8663b overall system, 127, 128
  standard event
    status, 138
    status enable, 138
  standard operation
    condition, 140
    event, 141
    event enable, 141
  status byte, 135
  status groups, register type descriptions, 136
remote, 12
  annunciator, 118
remote function
  HP Basic, 53
  setting
    e8663b, 12
remote interface
  programming, 2
  RS- 232, 38
remote programming
  hardware layers, 2
  software layers, 2
RS- 232
  address, 43, 118
  AUXILIARY INTERFACE connector, 2
  baud rate, 39
  cable, 40
  configuration, 39
  echo, setting, 39
  format parameters, 42
  interface, 39
  interfaces, 3
  IO libraries, 38
  overview, 38
  program examples, 43, 118
  programming examples, queries using C, 44, 122
  programming examples, queries using VISA, 44, 122
  programming examples, using C, 119
  programming examples, using VISA, 43, 119
  programming examples, using VISA C, 43
  settings, baud rate, 43, 118
  verifying operation, 41
RS- 232 queries
  HP Basic, using, 44, 121

S

save and recall, 158
SCPI, 7, 8, 16, 50
SCPI commands
  for status registers
    IEEE 488.2 common commands, 132
  GPIB function statements, 19

SCPI error queue, 13
SCPI register model, 126
service request interrupt
  reading, using VISA and C, 83
service request method
  status registers, 131
  using, 131
SetRWLS, 54
setting
  help mode
    e8663b, 12
SICL, 6, 7, 38
  GPIB examples, 20, 52
  iabort, 52
  iclear, 55
  igpibllo, 54
  iloc, 54
  iprintf, 55
  iremote, 53
  iscanf, 56
  NI libraries, 6
  VXI-11, programming, 88
signal generator
  monitoring status, 126
sockets
  example, 91, 94
  Java, 49, 115
  LAN, 31, 87, 91
  PERL, 116
  UNIX, 91
  Windows, 92
software layers
  remote programming, 2
software libraries, IO, 4
SRE commands, 132
SRQ command, 131
SRQ method, status registers, 131
standard event status
  enable register, 138
  group, 137
  register, 138
standard operation
  condition register, 140
  event enable register, 141
  event register, 141
  status group, 139
  transition filters, 140
state files, 158
states
  saving and recalling, using VISA and C, 77
status byte
  e8663b overall register system, 127, 128
  group, 134
  register, 135
status groups
  data questionable
    calibration, 154
    frequency, 148
    modulation, 151
    overview, 142
    power, 145
  registers, 136
  standard
    event, 137
    operation, 139
  status byte, 134
status registers
  See also registers
  accessing information, 130
  bit values, 129
  hierarchy, 126
  in status groups, 136
  monitoring, 130
  programming, 125
  SCPI commands, 132
  SCPI model, 126
  setting and querying, 132
  standard event
    bits, 138
    status enable, 138
  system overview, 126
  using, 129
STB command, 132
system requirements, 46

T

talker, 19
TCP/IP, 8
TCPIP, 5, 28, 87
Telnet
  DOS command prompt, 32
  example, 36
  PC, 33
  UNIX, 35
  using, 32
  Windows 2000, 34
  Windows XP, 34
transition filters
  See also filters
  data questionable
    negative and positive, 144
  data questionable calibration, 155
  data questionable frequency, 149
  data questionable modulation, 152
  data questionable power, 146
  description, 136
  standard operation, 140

Index

troubleshooting
GPIB, 18
ping response errors, 27
RS-232, 42
VISA assistant, 28

U

user flatness, 158, 169
user-data files, 157
creating, 157
downloading, 157
using C
data questionable status register, reading, 79
using VISA
data questionable status register, reading, 79

V

version M, 4
viPrintf, 55, 169
VISA, 7, 38
C++, generating a step-swept signal, 73
configuration (automatic), 5
configuration (manual), 6
CW signals, generating, 67
FM signals, generating internally applied, 71
generating externally applied AC-coupled FM signals, 69
I/O libraries, 6
LAN, using, 7
library, 20, 52
NI-488.2, 6
RS-232, using, 38
scanf, 56
service request interrupt, reading, 83
states, saving and recalling, 77
viPrintf, 55
Visual C++, generating a swept signal, 74
viTerminate, 52
VXI-11, 87
VISA and C
GPIB queries, 65
GPIB, interface check for, 58
VISA Assistant
configuring and running, 28
GPIB functionality, verifying, 18
IO Config, 5
IO, Using interactive, 27
troubleshooting, 28
verifying instrument communication, 27
VISA COM IO Library, 48
VISA configuration
automatic, 5
manual, 6
VISA LAN client, 27
visa.h, 169
Visual Basic, 7
IDE, 48
references, 48
Visual C++
NI-488.2, queries using, 63
VISA, generating a swept signal, 74
Visual C++ and VISA
generating a swept signal, 74
viTerminate, 52
viWrite, 169
VXI-11, 87
configuration, 30
programming, 87
using, 30
VISA, using, 89
with SICL, 88
with VISA, 89

W

web server
communicating with, 8
e8663b, 9
Windows
2000, 34
98, 4
ME, 4
NT, 4, 5
XP, 4, 34
Windows ME, 4
Windows NT, 4