

# Programming Guide

## Agilent Technologies E8257D/67D PSG Signal Generators

This guide applies to the following signal generator models:

**E8257D PSG Analog Signal Generator**

**E8267D PSG Vector Signal Generator**

Due to our continuing efforts to improve our products through firmware and hardware revisions, signal generator design and operation may vary from descriptions in this guide. We recommend that you use the latest revision of this guide to ensure you have up-to-date product information. Compare the print date of this guide (see bottom of page) with the latest revision, which can be downloaded from the following website:

*<http://www.agilent.com/find/psg>*



**Agilent Technologies**

**Manufacturing Part Number: E8251- 90355**

**Printed in USA**

**August 2005**

© Copyright 2004, 2005 Agilent Technologies, Inc..

## Notice

The material in this document is provided “as is,” and is subject to change 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.

## Questions or Comments about our Documentation?

We welcome any questions or comments you may have about our documentation. Please send us an E-mail at **[sources\\_manuals@am.exch.agilent.com](mailto:sources_manuals@am.exch.agilent.com)**.

<b>1. Getting Started</b>	<b>1</b>
Introduction to Remote Operation	1
Interfaces	2
I/O Libraries	3
Agilent IO Libraries Suite	3
Windows NT	3
Programming Language	5
Using GPIB	5
1. Installing the GPIB Interface Card	5
2. Selecting IO Libraries for GPIB	6
3. Setting Up the GPIB Interface	7
4. Verifying GPIB Functionality	7
GPIB Interface Terms	8
GPIB Function Statements	8
Using LAN	12
Selecting IO Libraries for LAN	12
Setting Up the LAN Interface	13
Verifying LAN Functionality	14
Using VXI-11	17
Using Sockets LAN	18
Using Telnet LAN	18
Using FTP	22
Using RS-232	23
1. Selecting IO Libraries for RS-232	24
2. Setting Up the RS-232 Interface	24
3. Verifying RS-232 Functionality	25
Character Format Parameters	26
If You Have Problems	26
Communicating with the Signal Generator Using a Web Browser	27
Error Messages	29
Error Message File	29
Error Message Types	29
<b>2. Programming Examples</b>	<b>31</b>
Using the Programming Examples	31
Programming Examples Development Environment	32
Running C/C++ Programming Examples	32

---

# Contents

Running Visual Basic 6.0® Programming Examples .....	33
Running C# Programming Examples .....	33
GPIB Programming Examples .....	34
Before Using the Examples .....	34
Interface Check using Agilent BASIC .....	35
Interface Check Using NI-488.2 and C++ .....	36
Interface Check using VISA and C .....	37
Local Lockout Using Agilent BASIC .....	38
Local Lockout Using NI-488.2 and C++ .....	39
Queries Using Agilent BASIC .....	40
Queries Using NI-488.2 and C++ .....	41
Queries Using VISA and C .....	43
Setting a CW Signal Using VISA and C .....	45
Generating an Externally Applied AC-Coupled FM Signal Using VISA and C .....	47
Generating an Internal AC-Coupled FM Signal Using VISA and C .....	49
Generating a Step-Swept Signal Using VISA and C .....	50
Saving and Recalling States Using VISA and C .....	52
Reading the Data Questionable Status Register Using VISA and C .....	54
Reading the Service Request Interrupt (SRQ) Using VISA and C .....	57
Using 8757D Pass-Thru Commands .....	60
LAN Programming Examples .....	63
Before Using the Examples .....	63
VXI-11 LAN Programming .....	63
Sockets LAN Programming using C .....	64
Sockets LAN Programming Using PERL .....	86
Sockets LAN Programming Using Java .....	87
RS-232 Programming Examples .....	89
Before Using the Examples .....	89
Interface Check Using Agilent BASIC .....	89
Interface Check Using VISA and C .....	90
Queries Using Agilent BASIC .....	92
Queries Using VISA and C .....	93
<b>3. Programming the Status Register System .....</b>	<b>95</b>
Overview .....	95
Status Register Bit Values .....	99
Accessing Status Register Information .....	99
Determining What to Monitor .....	100

Deciding How to Monitor . . . . .	100
Status Register SCPI Commands . . . . .	102
Status Byte Group . . . . .	104
Status Byte Register . . . . .	105
Service Request Enable Register . . . . .	105
Status Groups . . . . .	106
Standard Event Status Group . . . . .	106
Standard Operation Status Group . . . . .	108
Baseband Operation Status Group . . . . .	112
Data Questionable Status Group . . . . .	115
Data Questionable Power Status Group . . . . .	118
Data Questionable Frequency Status Group . . . . .	121
Data Questionable Modulation Status Group . . . . .	124
Data Questionable Calibration Status Group . . . . .	127
<b>4. Creating and Downloading Waveform Files . . . . .</b>	<b>131</b>
Overview . . . . .	131
Waveform Data Requirements . . . . .	132
Understanding Waveform Data . . . . .	132
Bits and Bytes . . . . .	132
LSB and MSB (Bit Order) . . . . .	133
Little Endian and Big Endian (Byte Order) . . . . .	134
Byte Swapping . . . . .	135
DAC Input Values . . . . .	135
2's Complement Data Format . . . . .	138
I and Q Interleaving . . . . .	138
Waveform Structure . . . . .	139
File Header . . . . .	140
Marker File . . . . .	140
I/Q File . . . . .	141
Waveform . . . . .	141
Waveform Phase Continuity . . . . .	142
Phase Discontinuity, Distortion, and Spectral Regrowth . . . . .	142
Avoiding Phase Discontinuities . . . . .	143
Waveform Memory . . . . .	144
Memory Allocation . . . . .	145
Memory Size . . . . .	146
Commands for Downloading and Extracting Waveform Data . . . . .	146

---

# Contents

Waveform Data Encryption .....	146
File Transfer Methods.....	147
SCPI Command Line Structure .....	147
Commands and File Paths for Downloading and Extracting Waveform Data .....	148
FTP Procedures.....	150
Creating Waveform Data .....	152
Code Algorithm .....	152
Downloading Waveform Data .....	157
Using Simulation Software.....	158
Using Advanced Programming Languages .....	160
Loading, Playing, and Verifying a Downloaded Waveform.....	164
Loading a File from Non-Volatile Memory .....	164
Playing the Waveform.....	164
Verifying the Waveform .....	165
Using the Download Utilities.....	166
Downloading E443xB Signal Generator Files .....	166
E443xB Data Format .....	167
Storage Locations for E443xB ARB files.....	167
SCPI Commands.....	168
Programming Examples.....	169
C++ Programming Examples .....	169
MATLAB Programming Examples .....	194
Visual Basic Programming Examples.....	200
HP Basic Programming Examples .....	205
Troubleshooting Waveform Files .....	214
<b>5. Creating and Downloading User-Data Files.....</b>	<b>215</b>
User Bit/Binary File Data Downloads .....	215
Data Requirements and Limitations .....	216
Bit and Binary Directories .....	216
Selecting Downloaded User Files as the Transmitted Data .....	219
FIR Filter Coefficients Downloads .....	219
Data Requirements and Limitations .....	219
Downloading FIR Filter Coefficients.....	220
Selecting a Downloaded User FIR Filter as the Active Filter .....	220
Downloads Directly into Pattern RAM (PRAM).....	221
Preliminary Setup .....	221
Data Requirements and Limitations .....	222

Downloading in List Format . . . . .	223
Downloading in Block Format . . . . .	223
Modulating and Activating the Carrier . . . . .	224
Viewing a PRAM Waveform . . . . .	225
Save and Recall Instrument State Files. . . . .	225
Save and Recall Programming Example . . . . .	226
Download User Flatness Corrections Using C++ and VISA . . . . .	235
Data Transfer Troubleshooting . . . . .	239
User Bit/Binary File Download Problems . . . . .	239
User FIR Filter Coefficient File Download Problems. . . . .	240
Direct PRAM File Download Problems . . . . .	240





---

# 1 Getting Started

This chapter provides the following major sections:

- [“Introduction to Remote Operation” on page 1](#)
- [“Using GPIB” on page 5](#)
- [“Using LAN” on page 12](#)
- [“Using RS-232” on page 23](#)
- [“Communicating with the Signal Generator Using a Web Browser” on page 27](#)
- [“Error Messages” on page 29](#)

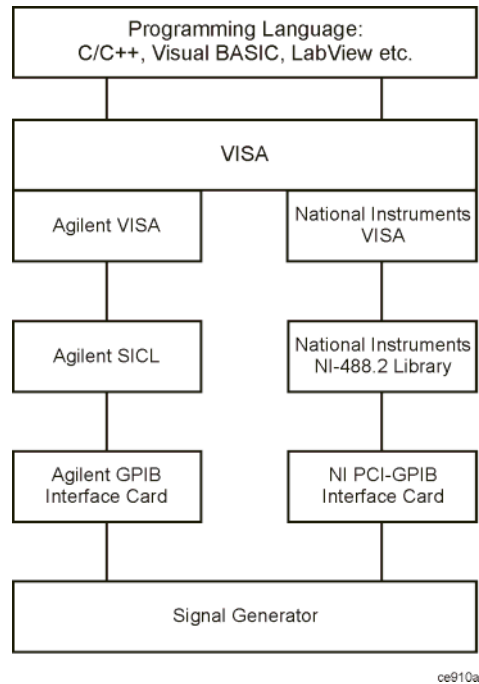
## Introduction to Remote Operation

PSG signal generators support the following interfaces:

- General Purpose Interface Bus (GPIB)
- Local Area Network (LAN)
- ANSI/EIA232 (RS-232) serial connection

Each of these interfaces, in combination with an IO library and programming language, can be used to remotely control the signal generator. [Figure 1-1](#) uses the GPIB as an example of the relationships between the interface, IO libraries, programming language, and signal generator.

Figure 1-1 Software/Hardware Layers



## 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 the GPIB handles information in 8-bit bytes. 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.

### LAN

LAN based communication is supported by the signal generator. Data transfer is fast as the LAN handles packets of data. The distance between a computer and the signal generator is limited to 100 meters (10BASE-T). The following protocols can be used to communicate with the signal generator over the LAN:

- VXI-11 (Recommended)
- Sockets LAN
- Telephone Network (Telnet)
- File Transfer Protocol (FTP)

RS-232                      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.

## I/O Libraries

An I/O library 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 with your signal generator or Agilent GPIB interface board, or they can be downloaded from the Agilent website: <http://www.agilent.com>.

---

**NOTE**      Agilent I/O libraries support the VXI-11 standard.

---

## Agilent IO Libraries Suite

The Agilent IO Libraries Suite replaces earlier versions of the Agilent IO Libraries (version M and earlier) and is supported on all platforms except Windows NT. If you are using the Windows NT platform, refer to the section on [“Windows NT” on page 3](#).

The Agilent IO Libraries Suite is available on the Automation-Ready CD that is shipped with your signal generator. The libraries can also be downloaded from the Agilent website: <http://www.agilent.com>. 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 a variety of I/O interfaces. Follow instructions in the setup wizard to install the libraries on your computer.

---

**IMPORTANT**      The VXI-11 SCPI service must be enabled before you can communicate with the signal generator over the LAN interface. Go to the **Utility > GPIB/RS-232 LAN > LAN Services Setup** menu and enable (turn On) the VXI-11 SCPI service.

---

Refer to the Agilent IO Libraries Suite Help documentation for details on the features available with this software.

## Windows NT

You must use Agilent IO Libraries version M or earlier if you have the Windows NT platform. The libraries can be downloaded from the Agilent website: <http://www.agilent.com>.

---

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

---

## IO Config Program

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.

---

**NOTE** Install GPIB interface boards before running IO Config.

---

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

## VISA Assistant

Use 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 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.

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.

## Manual Configuration

Perform the following steps 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.

## Programming Language

The programming language is used along with Standard Commands for Programming Instructions (SCPI) and IO library functions to remotely control the signal generator. Common programming languages include:<sup>1</sup>

- C/C++
- Visual Basic®
- LabView®
- Visual Basic.net®
- Agilent BASIC
- PERL
- Java™
- C#®

## Using GPIB

The GPIB allows instruments to be connected together and controlled by a computer. The 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.

### 1. Installing the GPIB Interface Card

A GPIB interface card must be installed in your computer. Two common GPIB interface cards are the National Instruments (NI) PCI-GPIB and the Agilent GPIB interface cards. Follow the GPIB interface card instructions for installing and configuring the card in your computer. The following tables provide information on interface cards.

Table 1-1 Agilent GPIB Interface Card for PC-Based Systems

Interface Card	Operating System	IO Library	Languages	Backplane/BUS	Max IO (kB/sec)	Buffering
Agilent 82341C for ISA bus computers	Windows <sup>a</sup> 95/98/NT/ 2000 <sup>®</sup>	VISA/ SICL	C/C++, Visual Basic, Agilent VEE, Agilent 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, Agilent 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, Agilent Basic for Windows	PCI 32 bit	750	Built-in

a. Windows 95, 98, NT and 2000 are registered trademarks of Microsoft Corporation

---

1. Java is a U.S. trademark of Sun Microsystems, Inc.

**Table 1-2 NI-GPIB Interface Card for PC-Based Systems**

Interface Card	Operating System	IO Library	Languages	Backplane/BUS	Max IO
National Instrument's PCI-GPIB	Windows 95/98/2000/ME/NT	VISA NI-488.2™ <sup>a</sup>	C/C++, Visual BASIC, LabView	PCI 32 bit	1.5 MB/s
National Instrument's PCI-GPIB+	Windows NT	VISA NI-488.2	C/C++, Visual BASIC, LabView	PCI 32 bit	1.5 MB/s

a. NI-488.2 is a trademark of National Instruments Corporation

**Table 1-3 Agilent-GPIB Interface Card for HP-UX Workstations**

Interface Card	Operating System	IO Library	Languages	Backplane/BUS	Max IO (kB/sec)	Buffering
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

## 2. Selecting IO Libraries for GPIB

The Agilent IO libraries Suite is available on the Automation-Ready CD which was shipped with your signal generator. In addition, IO libraries are included with your GPIB interface card or can be downloaded from the National Instruments or Agilent website. The following is a discussion on these libraries.

### VISA

VISA (Virtual Instrument Software Architecture) 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. Use the Agilent VISA library with the Agilent GPIB interface card or NI-VISA with the NI PCI-GPIB interface card.<sup>1</sup> Refer to [“Agilent IO Libraries Suite” on page 3](#) for more information on installing Agilent IO Libraries.

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

- 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).
- 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).

### 3. Setting Up the GPIB Interface

1. Press **Utility** > **GPIB/RS-232** > **GPIB Address**.
2. Use the numeric keypad, the arrow keys, or rotate the front panel knob to set the desired address.

The signal generator's GPIB address is set to 19 at the factory. The acceptable range of addresses is 0 through 30. Once initialized, the state of the GPIB address is not affected by a signal generator preset or by a power cycle. Other instruments on the GPIB cannot use the same address as the signal generator.

3. Press the **Enter** softkey.
4. Connect a GPIB interface cable between the signal generator and the computer. (Refer to [Table 1-4](#) for cable part numbers.)

**Table 1-4 Agilent GPIB Cables**

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

### 4. Verifying GPIB Functionality

Use the Agilent Connection Expert and the VISA Assistant available with the Agilent IO Libraries Suite or the Getting Started Wizard available with the National Instrument IO Library, to verify GPIB functionality. These utility programs allow you to communicate with the signal generator and verify its operation over the GPIB interface. Refer to the **Help** menu available in each utility for information and instructions on running these programs.

---

**NOTE** If you are using Windows NT refer to the section **"Windows NT"** on [page 3](#) for information on running the IO Config utility.

---

If You Have Problems

- 1. Verify the signal generator’s address matches that declared in the program (example programs in Chapter 2 use address 19).
- 2. Remove all other instruments connected to the GPIB and re-run 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 Function Statements

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. Refer to the NI-488.2 Function Reference Manual for Windows, Agilent Standard Instrument Control Library reference manual, and Microsoft® Visual C++ 6.0 <sup>1</sup>documentation for more information.

Abort Function

The Agilent 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.

Agilent BASIC	VISA	NI-488.2	Agilent SIDL
10 ABORT 7	viTerminate (parameter list)	ibstop(int ud)	iabort (id)

Agilent BASIC	The ABORT function stops all GPIB activity.
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.

1. Microsoft is a registered trademark of Microsoft Corporation.



NI-488.2 Library	The NI-488.2 library function aborts any asynchronous read, write, or command operation that is in progress. The parameter <code>ud</code> is the interface or device descriptor.
SICL	The Agilent SICL function aborts any command currently executing with the session <code>id</code> . This function is supported with C/C++ on Windows 3.1 and Series 700 HP-UX.

### Remote Function

The Agilent BASIC function `REMOTE` and the other listed IO library functions cause the signal generator to change 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 on the signal generator front panel restores manual operation.

Agilent BASIC	VISA	NI-488.2	Agilent SICL
10 <code>REMOTE</code> 719	N/A	<code>EnableRemote</code> (parameter list)	<code>iremote</code> (id)

Agilent BASIC	The <code>REMOTE</code> 719 function disables the front panel operation of all keys with the exception of the <b>Local</b> key.
VISA Library	The VISA library, at this time, does not have a similar command.
NI-488.2 Library	This 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.
SICL	The Agilent SICL function puts an instrument, identified by the <code>id</code> parameter, into remote mode and disables the front panel keys. Pressing the <b>Local</b> key on the signal generator front panel restores manual operation. The parameter <code>id</code> is the session identifier.

### Local Lockout Function

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

Agilent BASIC	VISA	NI-488.2	Agilent SICL
10 <code>LOCAL LOCKOUT</code> 719	N/A	<code>SetRWLS</code> (parameter list)	<code>igpibllo</code> (id)

Agilent BASIC	The <code>LOCAL LOCKOUT</code> function disables all front-panel signal generator keys. Return to local control can occur only with a hard on/off, when the <code>LOCAL</code> command is sent or if the <b>Preset</b> key is pressed.
VISA Library	The VISA library, at this time, does not have a similar command.

NI-488.2 Library	The NI-488.2 library function places the instrument described in the parameter list in remote mode by asserting the Remote Enable (REN) GPIB line. The lockout state is then set using the Local Lockout (LLO) GPIB message. Local control can be restored only with the EnableLocal NI-488.2 routine or hard reset. The parameter list describes the interface or device descriptor.
SICL	The Agilent SICL igpibllo function prevents user access to front panel keys operation. The function puts an instrument, identified by the <code>id</code> parameter, into remote mode with local lockout. The parameter <code>id</code> is the session identifier and instrument address list.

**Local Function**

The Agilent BASIC function `LOCAL` and the other listed functions cause the signal generator to return to local control with a fully enabled front panel.

Agilent BASIC	VISA	NI-488.2	Agilent SICL
10 <code>LOCAL 719</code>	N/A	<code>ibloc (int ud)</code>	<code>iloc(id)</code>

Agilent BASIC	The <code>LOCAL 719</code> function returns the signal generator to manual operation, allowing access to the signal generator's front panel keys.
VISA Library	The VISA library, at this time, does not have a similar command.
NI-488.2 Library	The NI-488.2 library function places the interface in local mode and allows operation of the signal generator's front panel keys. The <code>ud</code> parameter in the parameter list is the interface or device descriptor.
SICL	The Agilent SICL function puts the signal generator into local mode, enabling front panel key operation. The <code>id</code> parameter identifies the session.

**Clear Function**

The Agilent BASIC function `CLEAR` and the other listed IO library functions cause the signal generator to assume a cleared condition.

Agilent BASIC	VISA	NI-488.2	Agilent SICL
10 <code>CLEAR 719</code>	<code>viClear(ViSession vi)</code>	<code>ibclr(int ud)</code>	<code>iclear (id)</code>

Agilent BASIC	The <code>CLEAR 719</code> function causes all pending output-parameter operations to be halted, the parser (interpreter of programming codes) to reset and prepare for a new programming code, stops any sweep in progress, and continuous sweep to be turned off.
VISA Library	The VISA library uses the <code>viClear</code> function. This function performs an IEEE 488.1 clear of the signal generator.

NI-488.2 Library	The NI-488.2 library function sends the GPIB Selected Device Clear (SDC) message to the device described by 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.

Output Function

The Agilent 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.

Agilent BASIC	VISA	NI-488.2	Agilent SICL
10 OUTPUT 719	viPrintf(parameter list)	ibwrt(parameter list)	iprintf (parameter list)

Agilent BASIC	The function OUTPUT 719 puts the signal generator into remote mode, makes it a listener, and prepares it to receive data.
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.
NI-488.2 Library	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.
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.

Enter Function

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

Agilent BASIC	VISA	NI-488.2	Agilent SICL
10 ENTER 719;	viScanf (parameter list)	ibrd (parameter list)	iscanf (parameter list)

Agilent 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.
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.

NI-488.2 Library	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.
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.

## 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. See the IEEE website for more details.

The signal generator supports the following LAN interface protocols:

- VXI-11(VMEbus Extensions for Instrumentation as defined in VXI-11)
- Sockets LAN
- Telephone Network (Telnet)
- File Transfer Protocol (FTP)

VXI-11 is the best method to use for instrument communication using the LAN interface. Sockets LAN can be 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. Refer to [“VXI-11 LAN Programming” on page 63](#) for more information on the VXI-11 protocol.

---

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

---

## Selecting IO Libraries for LAN

The Telnet and FTP protocols do not require IO libraries. However, to write programs that control your signal generator over the LAN interface, an I/O 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 IO Library	The Agilent IO Library is a collection of libraries and includes the SICL and VISA Libraries. The VISA Library 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.

## Setting Up the LAN Interface

For LAN operation, the signal generator must be connected to the LAN, and a valid IP address must be assigned to the signal generator either manually or by using DHCP (Dynamic Host Configuration Protocol). Your system administrator can tell you which method to use.

---

**NOTE** Verify that the signal generator is connected to the LAN using a 10BASE-T LAN cable.

---

### Manual Configuration

1. Press **Utility > GPIB/RS-232 LAN > LAN Setup > Hostname**.

---

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

---

2. Use the labeled text softkeys and or numeric keypad to enter the desired hostname. The hostname can have up to 255 characters.  
To erase the current hostname, press **Editing Keys > Clear Text**.
3. Press the **Enter** softkey.
4. Press **LAN Config Manual DHCP** to **Manual**.
5. Press **IP Address** and enter a desired address.  
Use the left and right arrow keys to move the cursor. Use the up and down arrow keys, front panel knob, or numeric keypad to enter an IP address. To erase the current IP address, press the **Clear Text** softkey.

---

**NOTE** To remotely access the signal generator from a different LAN subnet, enter the correct subnet mask and default gateway. See your system administrator for information.

---

6. Press the **Proceed With Reconfiguration** softkey and then the **Confirm Change (Instrument will Reboot)** softkey.

This action assigns a hostname and IP address (as well as a gateway and subnet mask, if these have been configured) to the signal generator. The hostname, IP address, gateway and subnet mask are not affected by an instrument preset or by a power cycle.

### DHCP Configuration

DHCP (Dynamic Host Configuration Protocol) is a protocol used to assign a dynamic IP address to the signal generator. The network server software assigns an available IP address to the signal generator when the instrument is turned on. Different IP address may be designated at different times.

1. Press **Utility > GPIB/RS-232 LAN > LAN Setup**.

---

**NOTE** If the DHCP server uses a dynamic domain name service (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 may skip steps 2 through 4.

---

2. Press **Hostname**.

3. Use the labeled text softkeys and or numeric keypad to enter the desired hostname. To erase the current hostname, press **Editing Keys > Clear Text**.
4. Press the **Enter** softkey.
5. Press **LAN Config Manual DHCP** to select **DHCP**.
6. Press the **Proceed With Reconfiguration** softkey and then the **Confirm Change (Instrument will Reboot)** softkey.

This configures the signal generator as a DHCP client. In DHCP mode, the signal generator requests a new IP address from the DHCP server upon rebooting. You can return to the LAN Setup menu after rebooting to determine the assigned IP address.

### LAN Services Setup

Before you can use the LAN interface to control the signal generator you must enable the protocol you want to use. The signal generator supports: FTP Server, Web Server, Sockets SCPI, and VXI-11 SCPI protocols.

1. Press **Utility > GPIB/RS-232 LAN > LAN Services Setup**.
2. Press the softkey for the LAN service (s) you want to enable so that On is selected.
3. Press the **Proceed With Reconfiguration** softkey and then the **Confirm Change (Instrument will Reboot)** softkey.
4. Press the **Enter** softkey.

This action will configure the signal generator to use the selected LAN protocol.

### 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 1-5](#).

From a UNIX workstation, type:

```
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:<sup>1</sup>

```
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.

---

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

**NOTE** In DHCP mode, if the DHCP server uses a dynamic domain name service (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.

**Table 1-5 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	<p>If error messages appear, 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.</p> <p>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.</p> <p>Check that the hostname and IP address are correctly entered in the node names database. To do this, enter the nslookup &lt;hostname&gt; command from the command prompt.</p>
No Response	<p>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 <b>System Utility &gt; GPIB/RS-232 LAN &gt; LAN Setup</b> menu.</p> <p>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.</p> <p>If the signal generator still does not respond to ping, you should suspect a hardware problem.</p>
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 the 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 [“Agilent IO Libraries Suite” on page 3](#) 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” on page 3](#) 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.

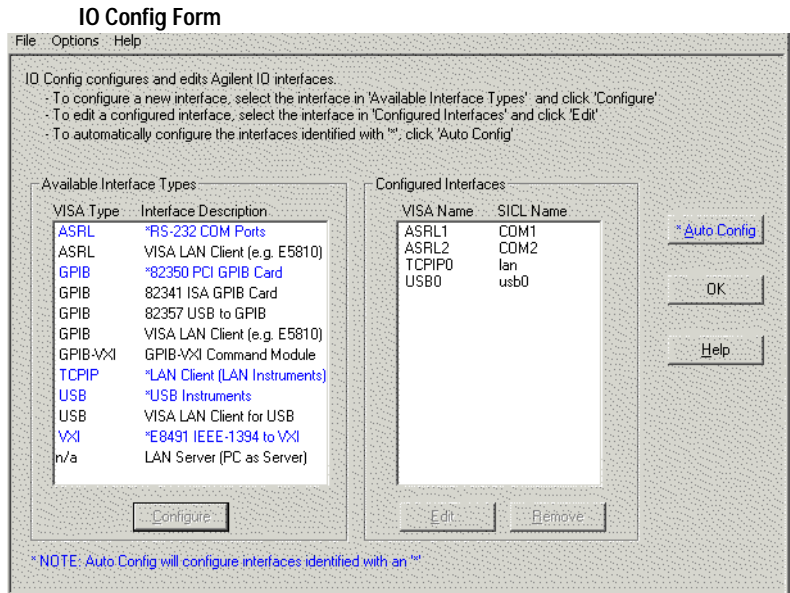
---

**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 1-2 on page 17](#).

---



Figure 1-2



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 16
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 VXI-11 protocol for instrument control using the LAN interface. The VXI-11 protocol is an industry standard, instrument communication protocol, described in the VXI-11 standard. Refer to the VXIbus Consortium.Inc website at <http://www.vxi.org/freepdfdownloads> for more information.

---

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

---

The VXI-11 protocol uses Open Network Computing/Remote Procedure Calls (ONC/RPC) 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. The VXI-11 standard allows IEEE 488.2 messages and IEEE 488.1 instrument control messages.

## Configuring the Interface

The Agilent IO Libraries Suite has utilities to help you easily connect to, and communicate with, the signal generator. Run the Interactive IO utility and VISA Assistant to verify the LAN connection. For more information, refer to the section describing the [“Agilent IO Libraries Suite” on page 3](#).

---

**IMPORTANT** The VXI-11 SCPI service must be enabled before you can communicate with the signal generator over the LAN interface. Go to the **Utility > GPIB/RS-232 LAN > LAN Services Setup** menu and enable (turn On) the VXI-11 SCPI service.

---

---

**NOTE** If you are using the Windows NT platform, refer to [“Windows NT” on page 3](#) for information on using Agilent IO Libraries versions M or earlier to configure the interface.

---

## Using Sockets LAN

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** 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 [Chapter 2](#) 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.

---

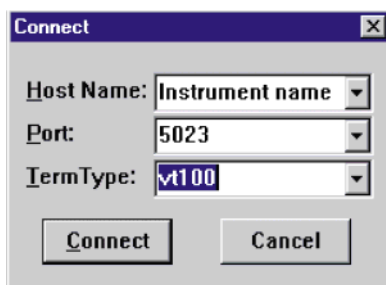
**NOTE** The Windows 2000<sup>®</sup> operating system uses a command prompt style interface for the Telnet client. Refer to the [Figure 1-5 on page 21](#) for an example of this interface.<sup>1</sup>

---

### Using Telnet and MS-DOS Command Prompt

1. On your PC, click **Start > Programs > Command Prompt**.
  2. At the command prompt, type in `telnet`.
  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 1-3](#)) is displayed.
- Connect Form

Figure 1-3



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 1-4 on page 20](#).
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

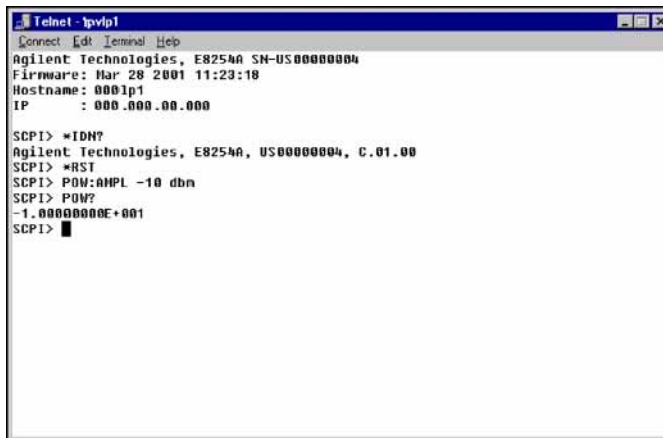
1. On your PC, click **Start > Run**.
2. Type `telnet` then click the **OK** button. The Telnet connection screen will be displayed.

---

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

3. Click on the **Connect** menu then select **Remote System**. A connection form is displayed. See [Figure 1-3](#).
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 1-4 on page 20](#).
6. To signal device clear, press Ctrl-C.
7. Select **Exit** from the **Connect** menu to end the Telnet session.

Figure 1-4 Telnet Window



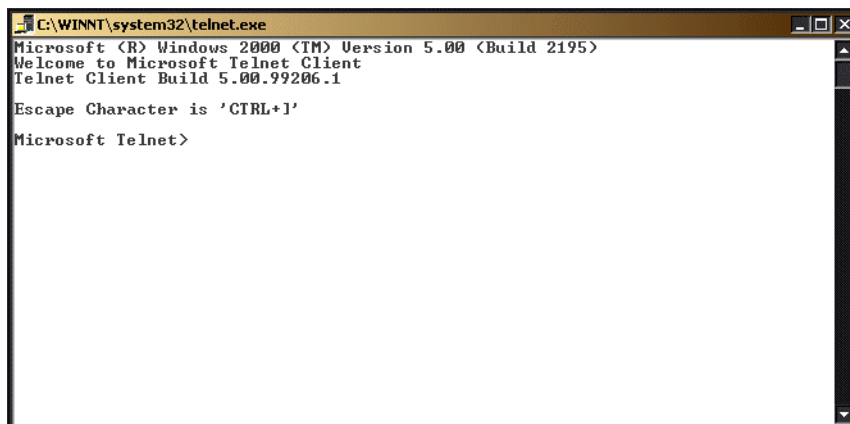
```
Telnet - hpvip1
Connect Edit Terminal Help
Agilent Technologies, E8254A SN-US00000004
Firmware: Mar 28 2001 11:23:18
Hostname: 0001p1
IP : 000.000.00.000

SCPI> *IDN?
Agilent Technologies, E8254A, US00000004, C.01.00
SCPI> *RST
SCPI> POW:AMPL -10 dbm
SCPI> POW?
-1.00000000E+001
SCPI> █
```

### 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 1-5 on page 21](#).
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 1-4 on page 20](#).
6. To escape from the SCPI> session type Ctrl-].
7. Type `quit` at the prompt to end the Telnet session.

Figure 1-5 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, E8254A 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 1-6 on page 23](#) shows the FTP interface and lists the directories in the signal generator's user level directory.

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: prompt, press the Enter key.
4. At the Password: prompt, the Enter key.

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. Use the `cd` command to change to and open a directory in the signal generator where a file is to be stored or retrieved.

You can download files to the signal generator from the directory in your PC where the command prompt is located by using the `put` command: `put "<file name>"`.

---

**NOTE** File names are limited to 23 characters.

---

An example of this command might be as follows:

`put <file_name> /USER/WAVEFORM/<new_file_name>` where `<file_name>` is the name of the file to download and `<new_file_name>` the name of the file that will appear in the signal generator's memory.

If you have a marker file associated with the waveform file, use the following command to download it to the signal generator: `put <marker file_name> /USER/MARKERS/<new_file_name>`

---

**NOTE** In the examples above the waveform and marker files are saved to the signal generator's non-volatile (NVWFM) waveform memory. You can save the files to volatile (WFM1) memory for immediate playing by the signal generator by changing the command to:  
/USER/BBG1/WAVEFORM for the waveform file and /USER/BBG1/MARKERS for the marker file. Note that the marker and waveform file have the same file name.

---

To upload a file from the signal generator to the directory in your PC where the command prompt is located use the get command: get "<file name>".

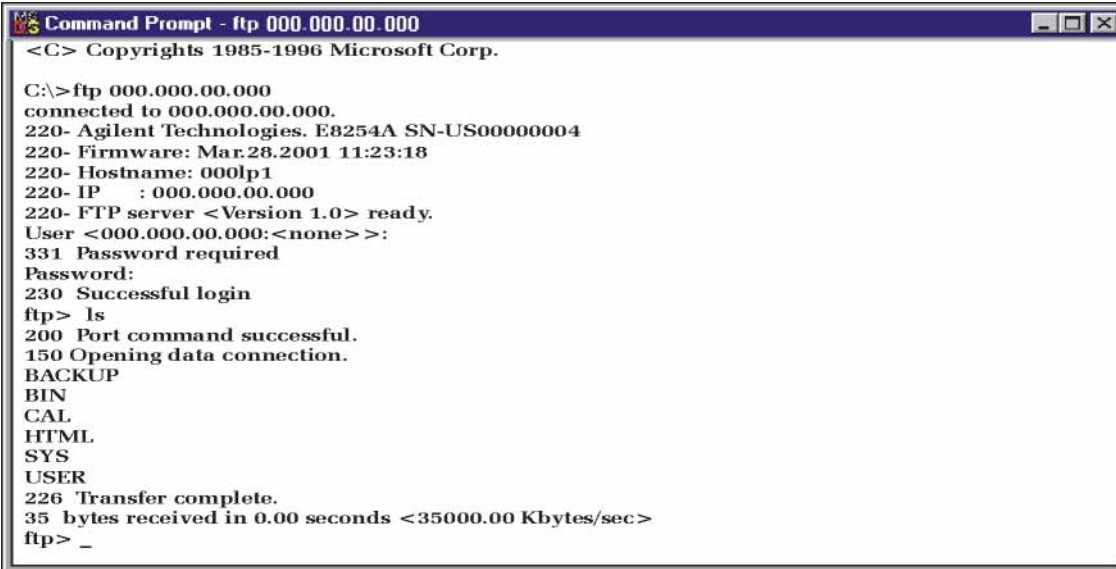
---

**NOTE** If no marker file is provided, the signal generator will automatically create a default marker file initialized with zeros.

---

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

Figure 1-6 FTP Screen



```
Command Prompt - ftp 000.000.00.000
<C> Copyrights 1985-1996 Microsoft Corp.

C:\>ftp 000.000.00.000
connected to 000.000.00.000.
220- Agilent Technologies. E8254A SN-US00000004
220- Firmware: Mar.28.2001 11:23:18
220- Hostname: 000lp1
220- IP      : 000.000.00.000
220- FTP server <Version 1.0> ready.
User <000.000.00.000:<none>>:
331 Password required
Password:
230 Successful login
ftp> ls
200 Port command successful.
150 Opening data connection.
BACKUP
BIN
CAL
HTML
SYS
USER
226 Transfer complete.
35 bytes received in 0.00 seconds <35000.00 Kbytes/sec>
ftp> _
```

ce917a

## Using RS-232

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 AUXILIARY INTERFACE connector using the cable described in [Table 1-6 on page 25](#). Many functions

provided by GPIB, with the exception of indefinite blocks, serial polling, GET, non-SCPI remote languages, 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.

## 1. Selecting IO Libraries for RS-232

The Agilent IO Libraries Suite is available on the Automation-Ready CD that is shipped with your signal generator. The libraries can also 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.

Agilent BASIC	The Agilent 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.

---

NOTE	It is recommended that the VXI-11 protocol be used for instrument communication over the RS-232 interface.
------	--

---

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.

## 2. Setting Up the RS-232 Interface

1. Press **Utility > GPIB/RS-232 LAN> RS-232 Setup > RS-232 Baud Rate > 9600**

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.

---

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

---

2. Press **Utility > GPIB/RS-232 LAN > RS-232 Setup > RS-232 Echo Off On** until **Off** is highlighted.  
Set the signal generator's RS-232 echo. Selecting **On** echoes or returns characters sent to the signal generator and prints them to the display.
3. Connect an RS-232 cable from the computer's serial connector to the signal generator's AUXILIARY INTERFACE connector. Refer to [Table 1-6](#) for RS-232 cable information.



Table 1-6 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.

---

### 3. 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 COM 1 or COM 2 serial ports and perform the following steps:

1. On the PC click **Start > Programs > Accessories > HyperTerminal** and select **HyperTerminal**.
2. Enter a name for the session in the text box and select an icon.
3. Select COM1 (COM2 can be used if COM1 is unavailable), and set the following parameters:
  - Bits per second: 9600 must match signal generator's baud rate; on the signal generator, press **Utility > GPIB/RS-232 LAN > RS-232 Setup > RS-232 Baud Rate > 9600**.
  - 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 programatically 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.

---

4. Go to the HyperTerminal window and select **File > Properties**
5. Go to **Settings > Emulation** and select **VT100**.
6. Leave the **Backscroll buffer lines** set to the default value.
7. Go to **Settings > ASCII Setup**.
8. 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 1-6](#).
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.
5. Press the **Reset RS-232** softkey and restart the HyperTerminal application.

## Communicating with the Signal Generator Using a Web Browser

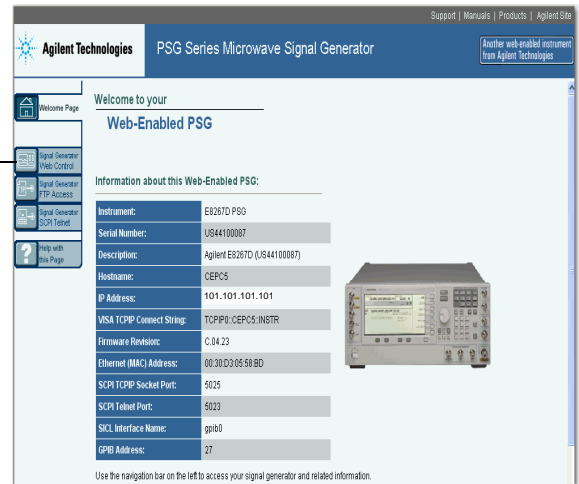
The Web Server uses a client/server model where the client is the web browser on your PC or workstation and the server is the signal generator. When you enable the Web Server, you can access a web page that resides on the signal generator.

The web-enabled signal generator web page, shown at right and [page 28](#), 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, manuals, support, and website. For additional information on memory catalog access (file storing), refer to the *E8257D/67D Signal Generators User's Guide* and "[Waveform Memory](#)" on [page 184](#) and for FTP, see "[Using FTP](#)" on [page 22](#) and "[FTP Procedures](#)" on [page 191](#).

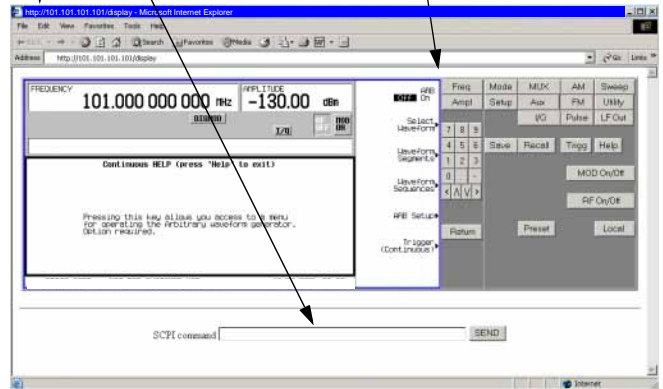
The Web Server service is compatible with the latest version of the Microsoft® Internet Explorer web browser.<sup>1</sup>

1. If it is not already enabled, turn on the Web server:
  - a. Press **Utility > GPIB/RS-232 LAN > LAN Services Setup**.
  - b. If necessary, press **> Web Server On > Proceed With Reconfiguration > Confirm Change**.
2. Launch the PC or workstation web browser.
3. In the web browser address field, enter the signal generator's IP (internet protocol) address. For example, *http://101.101.01.101* (where *101.101.01.101* is the signal generator's IP address).

The IP address can change depending on the LAN configuration (see "[Using LAN](#)" on [page 12](#)).



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

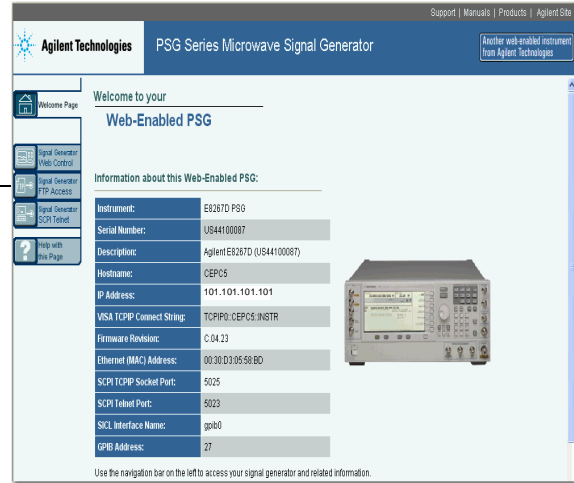


The results of a SCPI command display on a separate web page titled, "SCPI Command Processed." You can continue using this web page to enter SCPI commands or you can return to the front panel web page. If the web page does not update, use the Web browser Refresh function.

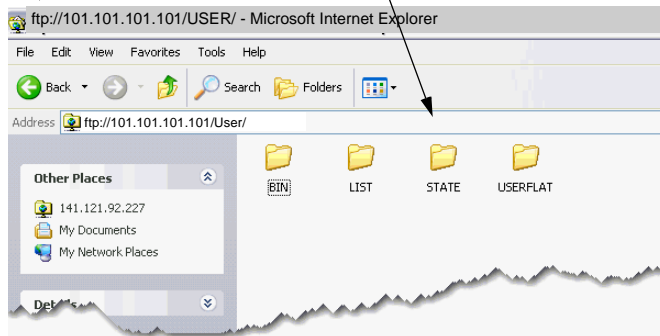
1. Microsoft is a registered trademark of Microsoft Corp.

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 button opens a window to show the folders containing the signal generator's memory catalog files.



## 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 *E8257D/67D Signal Generators User's Guide*.

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 <code>SYSTem:ERRor[:NEXT]?</code>
Clearing the Queue	Power up Send a *CLS command Read last item in the queue
Unresolved Errors <sup>a</sup>	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 Programming Examples

This chapter provides the following major sections:

- [“Using the Programming Examples” on page 31](#)
- [“GPIB Programming Examples” on page 34](#)
- [“LAN Programming Examples” on page 63](#)
- [“RS-232 Programming Examples” on page 89](#)

### Using the Programming 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 I/O 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:

- Agilent BASIC
- C/C++
- Java
- PERL
- Microsoft Visual Basic 6.0
- C#

See [Chapter 1](#) of this programming guide for information on interfaces, I/O libraries, and programming languages.

---

**NOTE** For information on downloading waveform files refer to [“Programming Examples for Generating and Downloading Files” on page 189](#).

---

The example programs are also available on the PSG 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.

---

---

**NOTE** To update the signal generator’s front panel display so that it reflects remote command setups, enable the remote display: press **Utility > Display > Update in Remote Off On** softkey until On is highlighted or send the SCPI command `:DISPlay:REMOte ON`. For faster test execution, disable front panel updates.

---

## Programming Examples Development Environment

The C/C++ examples in this guide were written using an IBM-compatible computer (PC) with the following configuration:

- Pentium® processor<sup>1</sup>
- Windows NT 4.0, and Windows 2000 operating system. Programs for creating and downloading files to the signal generator were run on a Windows 2000 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. Programs for creating and downloading files to the signal generator use the LAN interface.
- National Instruments VISA Library or Agilent VISA library
- COM1 or COM2 serial port available
- LAN interface card

The Agilent BASIC examples were run on a UNIX 700 Series workstation

## Running C/C++ Programming Examples

To run the example programs written in C/C++ you must include the required files in the Microsoft Visual C++ 6.0 project. For more information, refer to the *Agilent VISA User's Manual*, available on Agilent's website: <http://www.agilent.com>.

---

**NOTE** If you encounter the error message C1010 when running the C/C++ programs then use the *not using precompiled header* option in the IDE.

---

If you are using the VISA library do the following:

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

If you are using the NI-488.2 library do the following:

- 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

---

**IMPORTANT** The VXI-11 SCPI service must be enabled before you can communicate with the signal generator over the LAN interface. Go to the **Utility > GPIB/RS-232 LAN > LAN Services Setup** menu and enable the VXI-11 SCPI service.

---

Refer to the National Instrument website for information on the NI-488.2 library and file requirements. For information on the VISA library see the Agilent website.

---

1. Pentium is a U.S. registered trademark of Intel Corporation



The example C++ programs are available on the PSG Documentation CD-ROM, enabling you to cut and paste the examples into a text editor.

## Running Visual Basic 6.0® Programming Examples

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.

---

**IMPORTANT** The VXI-11 SCPI service must be enabled before you can communicate with the signal generator over the LAN interface. Go to the **Utility > GPIB/RS-232 LAN > LAN Services Setup** menu and enable (turn On) the VXI-11 SCPI service.

---

You can start a new Standard EXE project and add the required references. Once the required references are include, 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 PSG Documentation CD-ROM, enabling you to cut and paste the examples into your project.

## Running C# Programming Examples<sup>1</sup>

To run the example program written in C# 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.

---

**IMPORTANT** The VXI-11 SCPI service must be enabled before you can communicate with the signal generator over the LAN interface. Go to the **Utility > GPIB/RS-232 LAN > LAN Services Setup** menu and enable (turn On) the VXI-11 SCPI service.

---

1. Copy the State\_File.cs file in the examples directory to the .NET installation directory where the csc.exe file is located. The example C# program is available on the PSG Documentation CD-ROM
2. Run the MS-DOS Command Prompt program. Change the directory so that the command prompt program is in the same directory as the csc.exe and State\_File programs.

---

1. Visual Basic is a registered trademark of Microsoft corporation

3. On the command line, enter `csc State_File.cs`.
4. Follow the prompts in the program to save and recall signal generator instrument states.

## GPIB Programming Examples

- [“Interface Check using Agilent BASIC” on page 35](#)
- [“Interface Check Using NI-488.2 and C++” on page 36](#)
- [“Interface Check using VISA and C” on page 37](#)
- [“Local Lockout Using Agilent BASIC” on page 38](#)
- [“Local Lockout Using NI-488.2 and C++” on page 39](#)
- [“Queries Using Agilent BASIC” on page 40](#)
- [“Queries Using NI-488.2 and C++” on page 41](#)
- [“Queries Using VISA and C” on page 43](#)
- [“Setting a CW Signal Using VISA and C” on page 45](#)
- [“Generating an Externally Applied AC-Coupled FM Signal Using VISA and C” on page 47](#)
- [“Generating an Internal AC-Coupled FM Signal Using VISA and C” on page 49](#)
- [“Generating a Step-Swept Signal Using VISA and C” on page 50](#)
- [“Saving and Recalling States Using VISA and C” on page 52](#)
- [“Reading the Data Questionable Status Register Using VISA and C” on page 54](#)
- [“Reading the Service Request Interrupt \(SRQ\) Using VISA and C” on page 57](#)
- [“Using 8757D Pass-Thru Commands” on page 60](#)

## Before Using the Examples

If the Agilent GPIB interface card is used, then the Agilent VISA library along with the Agilent SICL library should be installed. 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 [“2. Selecting IO Libraries for GPIB” on page 6](#) and the documentation for your GPIB interface card for details.

---

**NOTE** Agilent 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.

---

## Interface Check using Agilent BASIC

This 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 PSG Documentation CD-ROM as basicex1.txt.

```

10 !*****
20 !
30 ! PROGRAM NAME:          basicex1.rtf
40 !
50 ! PROGRAM DESCRIPTION:  This program verifies that the GPIB connections
60 !                      and interface are functional.
70 !
80 ! Connect a controller to the signal generator using a GPIB cable.
90 !
100 !
110 ! LEAR and RESET the controller and type in the following commands
120 ! and then 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. Start 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 PSG 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); // Acquires 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 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. Start 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 PSG 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 Agilent BASIC

This example demonstrates the Local Lockout function. Local Lockout disables the front panel signal generator keys.

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

```
10  !*****
20  !
30  !  PROGRAM NAME:          basicex2.rtf
40  !
50  !  PROGRAM DESCRIPTION:  In REMOTE mode, access to the signal
60  ! generator's functional front panel keys are disabled except for
70  ! the Local and Contrast keys.  The LOCAL LOCKOUT command
80  ! 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 PSG address
140 CLEAR Sig_gen    ! Resets PSG 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 PSG 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 visible"
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 PSG'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. Start Microsoft Visual C++ 6.0, add the required files, and enter the following code into your .cpp source file. This example is available on the PSG 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 Agilent BASIC

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?

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

```
10  !*****
20  !
30  !  PROGRAM NAME:          basicex3.rtf
40  !
50  !  PROGRAM DESCRIPTION:  In this example, query commands are used
60  !  with response 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 data
130 INTEGER B                  ! Declares variable to hold int. response data
140 Sig_gen=719                ! Declares variable to hold PSG address
150 LOCAL Sig_gen              ! Puts PSG 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 PSG into a defined state
190 OUTPUT Sig_gen;"FREQ: CW?" ! Querys the PSG 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 PSG 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 PSG
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 PSG's GPIB address
500 OUTPUT Sig_gen;"SYST:COMM:GPIB:ADDR?"
510 ENTER Sig_gen;D$           ! Enter in the PSG'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 C++

This example uses the NI-488.2 library to query different instrument states and conditions. Start 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 PSG 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 Using VISA and C

This example uses VISA library functions to query different instrument states and conditions. Start 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 PSG 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, "%li", &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);
}
```

## Setting 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. Start Microsoft Visual C++ 6.0, add the required files, and enter the code into your .cpp source file.

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

```
/******
// PROGRAM FILE NAME:    visaex4.cpp
//
// PROGRAM DESCRIPTION: This example sets up the signal generator
// frequency and power level.
// The RF state of the signal generator is turned 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, "%li", &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, "%li", &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.

Start Microsoft Visual C++ 6.0, add the required files, and enter the code into your .cpp source file.

The following program example is available on the PSG 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 carriage return
// Close the sessions

viClose(vi);
viClose(defaultRM);
}

```



## Generating an Internal AC-Coupled FM Signal Using VISA and C

In this example the VISA library is used to generate an ac-coupled 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. Start 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 PSG Documentation CD-ROM as visaex6.cpp.

```
//*****
// PROGRAM FILE NAME:visaex6.cpp
//
// PROGRAM DESCRIPTION:This example generates an AC-coupled 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 EXT 2 source for FM
viPrintf(vi, "FM2:DEV 100 kHz\n");  // Sets FM path 2 coupling to AC
viPrintf(vi, "FREQ 900 MHz\n");     // Sets carrier frequency to 700 MHz
viPrintf(vi, "POW -15 dBm\n");      // Sets the power level to -2.3 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 carriage 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. Start 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 PSG 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 generoator to manual operation.\n");
// Closes the sessions

printf("\n");
viClose(vi);
viClose(defaultRM);
}
```

## 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. Start 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 PSG 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 information

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. Start 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 PSG 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 unlevelled 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.

Start Microsoft Visual C++ 6.0, add the required files, and enter the following code into your .cpp source file. This example is available on the PSG 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 "StdAfx.h"
#include "visa.h"
#include <stdio.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);          // Clear the signal generator
  viPrintf(vi, "*RST\n"); // Reset 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 no positive transition
// on Bit 3 affects the Operation Event Registrar
// 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, interrupt, rdBuffer);
// The next line 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 type 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");// Sets 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");// Sets up for single sweep
viPrintf(vi, "TRIG:SOUR IMM\n");// Triggers the current 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, interrupt, 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 interrupt(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("A SRQ, indicating end of sweep has occurred\n");
viClose(event); // Closes the event
return VI_SUCCESS;
}
```

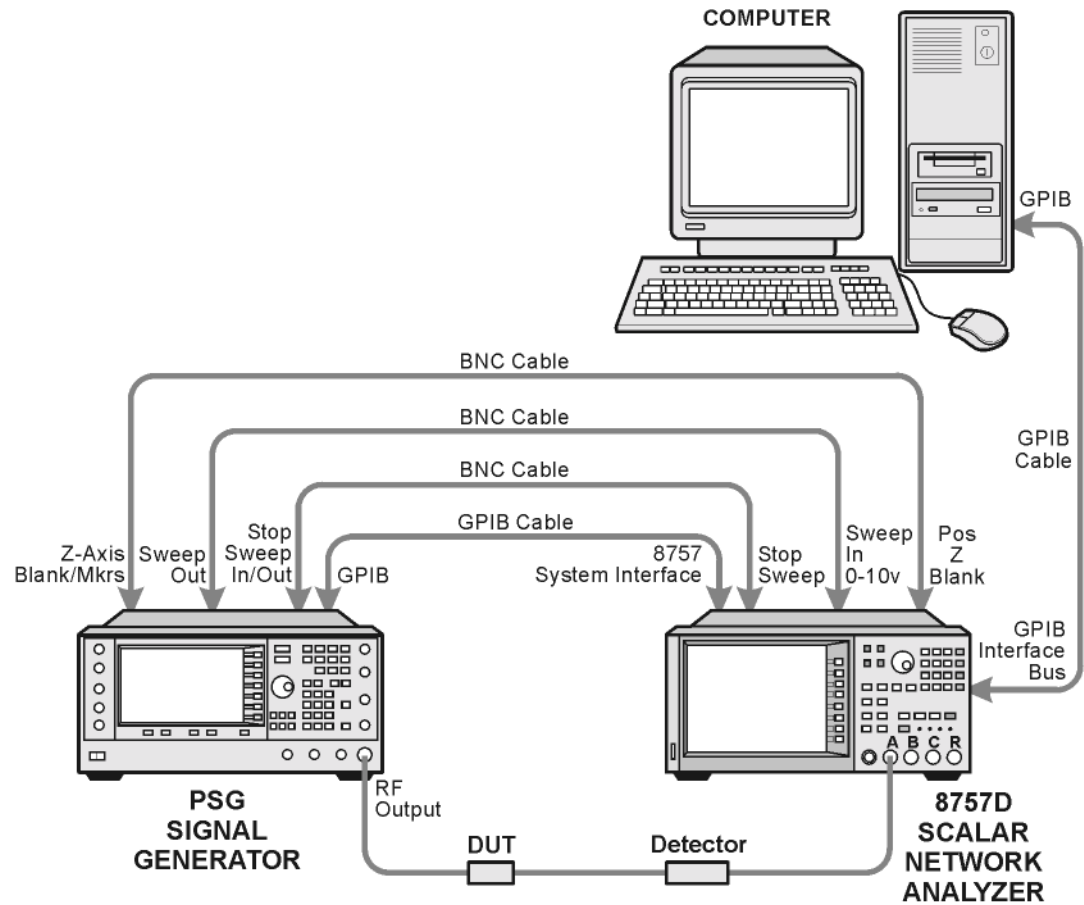
## Using 8757D Pass-Thru Commands

Pass-thru commands enable you to temporarily interrupt ramp sweep system interaction so that you can send operating instructions to the PSG. This section provides setup information and an example program for using pass-thru commands in a ramp sweep system.

Equipment Setup

To send pass-thru commands, set up the equipment as shown in [Figure 2-1](#). Notice that the GPIB cable from the computer is connected to the GPIB interface bus of the 8757D.

Figure 2-1



scalar\_netwk\_pc

GPIB Address Assignments

[Table 2-1](#) describes how GPIB addresses should be assigned for sending pass-thru commands. These are the same addresses used in [Example 2-1](#).

Table 2-1

Instrument	GPIB Address	Key Presses/Description
PSG	19	Press <b>Utility &gt; GPIB/RS-232 LAN &gt; GPIB Address &gt; 19 &gt; Enter.</b>
8757D	16	Press <b>LOCAL &gt; 8757 &gt; 16 &gt; Enter.</b>
8757D (Sweeper)	19	This address must match the PSG. Press <b>LOCAL &gt; SWEEPER &gt; 19 &gt; Enter.</b>
Pass Thru	17	The pass thru address is automatically selected by the 8757D by inverting the last bit of the 8757D address. Refer to the 8757D documentation for more information. Verify that no other instrument is using this address on the GPIB bus.

Example Pass-Thru Program

Example 2-1 on page 62 is a sample Agilent BASIC program that switches the 8757D to pass-thru mode, allowing you to send operating commands to the PSG. After the program runs, control is given back to the network analyzer. The following describes the command lines used in the program.

- Line 30 PT is set to equal the source address. C1 is added, but not needed, to specify the channel.
- Lines 40, 90 The END statement is required to complete the language transition.
- Lines 50, 100 A WAIT statement is recommended after a language change to allow all instrument changes to be completed before the next command.
- Lines 70, 80 This is added to ensure that the instrument has completed all operations before switching languages. Line 70 can only be used when the signal generator is in single sweep mode.
- Line 110 This takes the network analyzer out of pass-thru command mode, and puts it back in control. Any analyzer command can now be entered.

Example 2-1 Pass-Thru Program

```

10 ABORT 7
20 CLEAR 716
30 OUTPUT 716;"PT19;C1"
40 OUTPUT 717;"SYST:LANG SCPI";END
50 WAIT .5
60 OUTPUT 717;"POW:STAT OFF"
70 OUTPUT 717;"*OPC?"

```

```
80 ENTER 717; Reply
90 OUTPUT 717;"SYST:LANG COMP";END
100 WAIT .5
110 OUTPUT 716;"C2"
120 END
```

## LAN Programming Examples

- [VXI-11 LAN Programming](#)
- [“Setting Parameters and Sending Queries Using Sockets and C” on page 66](#)
- [“Setting the Power Level and Sending Queries Using PERL” on page 86](#)
- [“Generating a CW Signal Using Java” on page 87](#)

This section describes methods of communicating with the signal generator over the LAN interface. The VXI-11 protocol is described and program examples using C, Java, and PERL over socket LAN are shown. Telnet and FTP also use the LAN interface for instrument communication. For details on using FTP and TELNET refer to [“Using FTP” on page 22](#) and [“Using Telnet LAN” on page 18](#) of this guide.

### Before Using the Examples

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

### VXI-11 LAN Programming

The signal generator supports the VXI-11 protocol for instrument control using the LAN interface. The VXI-11 protocol is an industry standard, instrument communication protocol, described in the VXI-11 standard. Refer to the VXIbus Consortium, Inc website at <http://www.vxi.org/freepdfdownloads> for more information.

---

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

---

The VXI-11 protocol uses Open Network Computing/Remote Procedure Calls (ONC/RPC) running over TCP/IP. It is intended to provide GPIB capabilities such as SRQ (Service Request), status byte reading, and DCAS (Device Clear State) over a LAN interface. The VXI-11 standard allows IEEE 488.2 messages and IEEE 488.1 instrument control messages.

#### Configuring for VXI-11

Refer to the [“Agilent IO Libraries Suite” on page 3](#) for information on configuring the interface for LAN communication.

#### Using VXI-11 with GPIB Programs

The GPIB programming examples, listed in the [GPIB Programming Examples](#) section and using the VISA Library, 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 13](#) for more information.

---

<b>IMPORTANT</b>	The VXI-11 SCPI service must be enabled before you can communicate with the signal generator over the LAN interface. Go to the <b>Utility &gt; GPIB/RS-232 LAN &gt; LAN Services Setup</b> menu and enable (turn On) the VXI-11 SCPI service.
------------------	---

---

## Sockets LAN Programming using C

The program listing shown in [“Setting Parameters and Sending Queries Using Sockets and C” on page 66](#) consists of two files; lanio.c and getopt.c. The lanio.c file has two main functions; int main() and an int main1().

---

<b>NOTE</b>	The sockets protocol does not provide GPIB capabilities such as SRQ (Service Request) and status byte reading. It is recommended that the VXI-11 protocol be used for instrument communication over the LAN interface.
-------------	--

---

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.

---

<b>IMPORTANT</b>	Sockets SCPI must be enabled before you can communicate with the signal generator using this protocol. Go to the <b>Utility &gt; GPIB/RS-232 LAN &gt; LAN Services Setup</b> menu and enable (turn On) Sockets SCPI.
------------------	--

---

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 PSG 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 xxxxxx "*IDN?"` where xxxxxx 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, E8254A, US00000001, C.01.00

Frequency: +2.5000000000000E+09
Power Level: -5.000000000E+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.
2. Select **Rebuild All** from **Build** menu. Then select **Execute Lanio.exe**.
3. Click **Start**, click **Programs**, then click **Command Prompt**.
4. At the command prompt, `cd` to the directory containing the lanio.cpp file and then to the Debug folder. For example `C:\SocketIO\Lanio\Debug`
5. Type in lanio `xxxxx "*IDN?"` at the command prompt. 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.

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.c 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**.



```

* 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

#ifdef 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 7777
#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 = WSStartup(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 7777 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 queries 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  *

/*****
/* $Function: mainl$
/* $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 mainl() function to int main(). Re-compile and the
/* execute the program
/*****

int mainl()
{

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;
```

#### PRORGRAM 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 PERL

This example uses PERL script to control the signal generator over the sockets LAN interface. The signal generator power level is set to - 5 dBm, 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. This example is available on the PSG Documentation CD-ROM as perl.txt.

1. In the code below, enter your signal generator's hostname in place of the xxxxxx in the code line:  
my \$instrumentName= "xxxxxx"; .
2. Save the code 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

```
#!/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 name
my $instrumentName = "xxxxxx";

# Get socket
$sock = new IO::Socket::INET ( PeerAddr => $instrumentName,
                               PeerPort => 7777,
```

```

        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";

```

## 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. Run 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\_2\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 PSG 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 your hostname here
try
    {
        Socket t = new Socket(instrumentName,7777); // 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");                       // Queries 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);
}
}
}

```

## RS-232 Programming Examples

- [“Interface Check Using Agilent BASIC” on page 89](#)
- [“Interface Check Using VISA and C” on page 90](#)
- [“Queries Using Agilent BASIC” on page 92](#)
- [“Queries Using VISA and C” on page 93](#)

### Before Using the Examples

On the signal generator select the following settings:

- Baud Rate - 9600 must match computer’s baud rate
- Transmit Pace - None
- Receive Pace - None
- RTS/CTS - None
- RS-232 Echo - Off

### Interface Check Using Agilent 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 23](#) 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 8](#) for more help.

The following program example is available on the PSG 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 Agilent 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. Start Microsoft Visual C++, add the required files, and enter the following code into the .cpp source file.

The following program example is available on the PSG 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 Agilent BASIC

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.

Start Agilent BASIC, type in the following commands, and then RUN the program:

The following program example is available on the PSG 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 Agilent 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 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. Start Microsoft Visual C++, add the required files, and enter the following code into your .cpp source file.

The following program example is available on the PSG 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)
{
    ViStatus status; // Declares a type ViStatus variable
    ViSession defaultRM, instr; // Declares type ViSession variables
    ViUInt32 retCount; // Return count for string I/O
    ViChar buffer[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;
}
```

---

## 3 Programming the Status Register System

This chapter provides the following major sections:

- [“Overview” on page 95](#)
- [“Status Register Bit Values” on page 99](#)
- [“Accessing Status Register Information” on page 99](#)
- [“Status Byte Group” on page 104](#)
- [“Status Groups” on page 106](#)

### Overview

---

**NOTE** For the E8257D analog signal generator, some of the status bits and registers are not applicable for the E8257D and are always set to zero. These bits and registers are shown in the following list:

- Standard Operation Condition Register: Bits 0 and 10. Refer to [Table 3-5, “Standard Operation Condition Register Bits,” on page 109](#).
- Baseband Operation Condition Register: All
- Data Questionable Condition Register: Bit 8. Refer to [Table 3-7, “Data Questionable Condition Register Bits,” on page 116](#).
- Data Questionable Power Condition Register: Bit 2. Refer to [Table 3-8, “Data Questionable Power Condition Register Bits,” on page 119](#).
- Data Questionable Frequency Condition Register: Bit 3. Refer to [Table 3-9, “Data Questionable Frequency Condition Register Bits,” on page 122](#).
- Data Questionable Calibration Condition Register: Bit 1. Refer to [Table 3-11, “Data Questionable Calibration Condition Register Bits,” on page 128](#).
- Data Questionable Bert Condition Register: All

---

During remote operation, you may need to monitor the status of the signal generator for error conditions or status changes. The signal generator’s error queue can be read with the SCPI query `:SYSTEM:ERROR?` (Refer to `:ERROR[:NEXT]` in the 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 and/or condition changes.

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-level status registers propagate data to the higher-level 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 status registers monitor specific events or conditions, and are grouped according to their functionality. For example, the Data Questionable 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 106](#) for more information.

[Figure 3-1](#) and [Figure 3-2](#) show the signal generator's status byte register system and hierarchy.

The status register system uses IEEE 488.2 commands (those beginning with \*) to access the higher-level summary registers. Lower-level registers can be accessed using STATus SCPI commands.



Figure 3-1 The Overall Status Byte Register System (1 of 2)

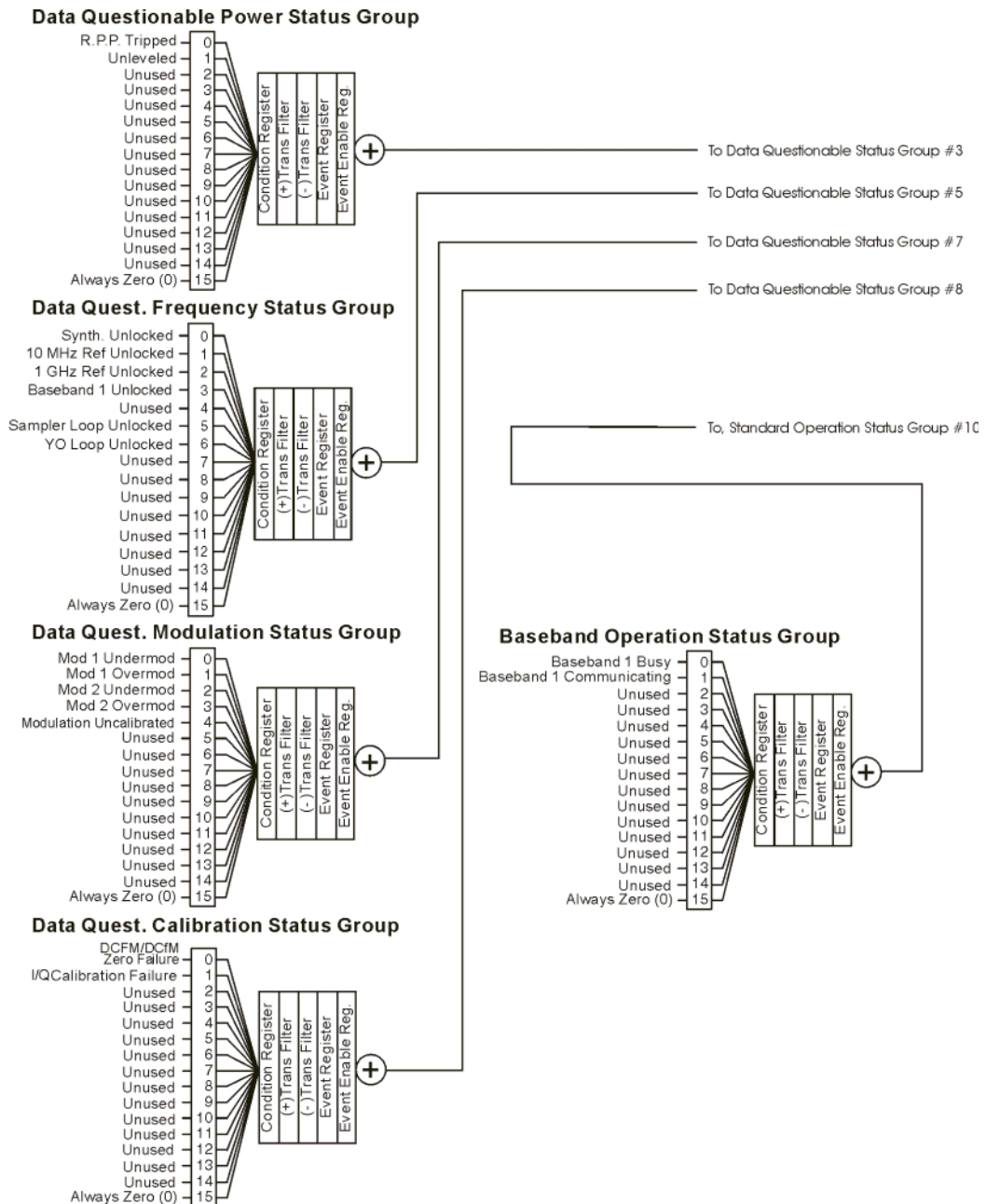
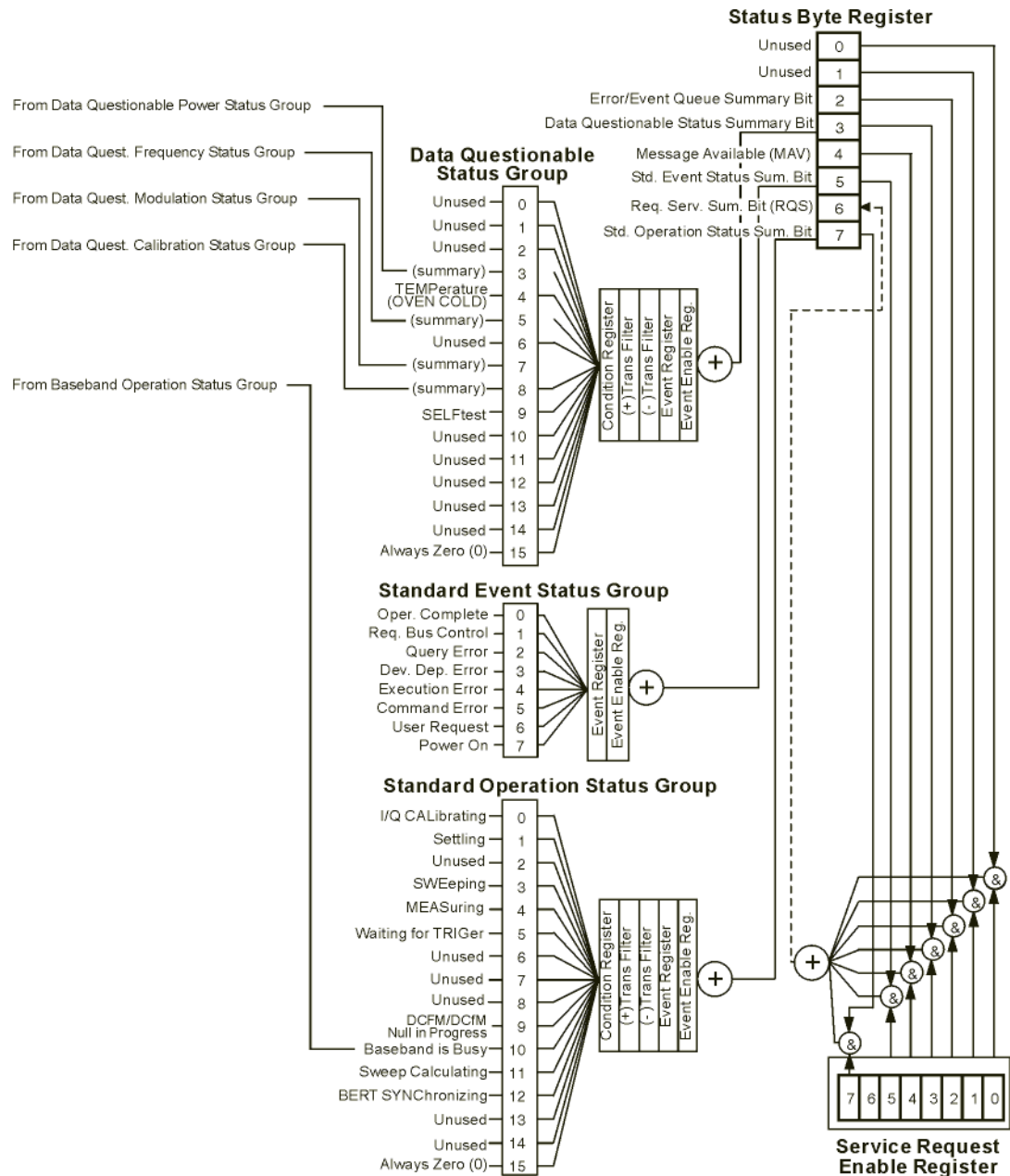


Figure 3-2 The Overall Status Byte Register System (2 of 2)



## Status Register Bit Values

Each bit in a register is represented by a decimal value based on its location in the register (see [Table 3-1](#)).

- 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 3-1 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 3-1 on page 97](#) or [Figure 3-2 on page 98](#) 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.

The event register can be cleared only by querying its contents or sending 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 I/O 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 if you must know immediately when a condition changes. (To detect a change using the polling method, the program must repeatedly read the registers.) 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 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 3-1 on page 97](#) or [Figure 3-2 on page 98](#) 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 byteregister in response to this poll. The device whose request service summary bit (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.

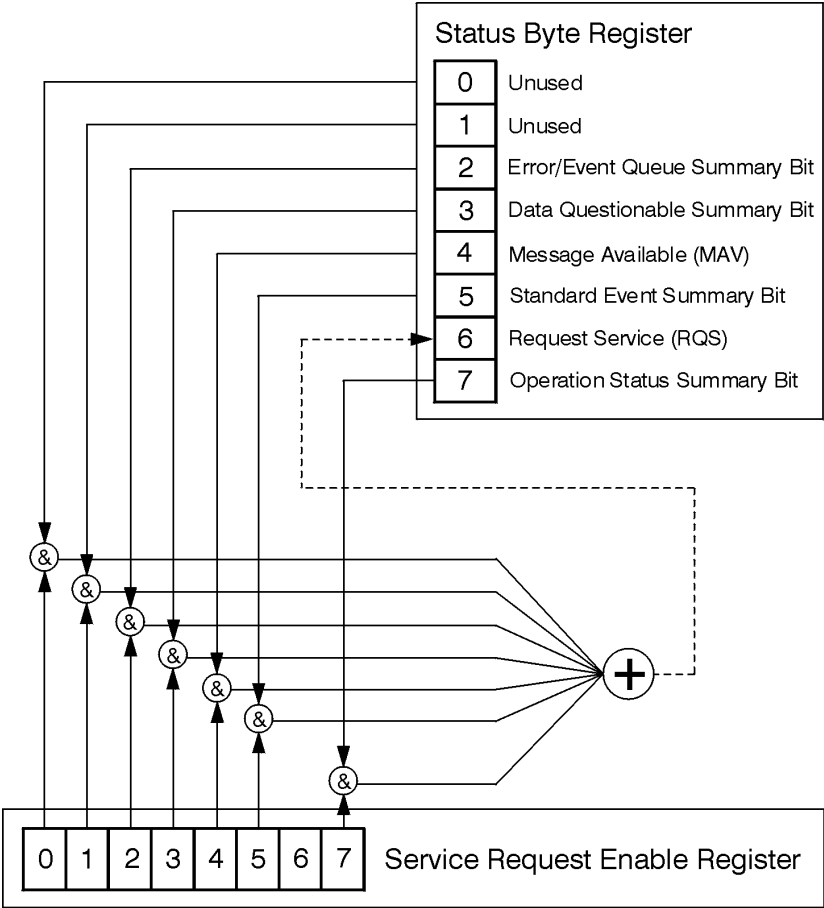
:STATus:PRESet presets all transition filters, non-IEEE 488.2 enable registers, and error/event queue enable registers. (Refer to [Table 3-2](#).)

**Table 3-2** Effects of :STATus:PRESet

Register	Value after :STATus:PRESet
:STATus:OPERation:ENABle	0
:STATus:OPERation:NTRansition	0
:STATus:OPERation:PTRransition	32767
:STATus:OPERation:BASEband:ENABle	0
:STATus:OPERation:BASEband:NTRansition	0
:STATus:OPERation:BASEband:PTRransition	32767
:STATus:QUESTionable:CALibration:ENABle	32767
:STATus:QUESTionable:CALibration:NTRansition	32767
:STATus:QUESTionable:CALibration:PTRransition	32767
:STATus:QUESTionable:ENABle	0
:STATus:QUESTionable:NTRansition	0
:STATus:QUESTionable:PTRransition	32767
:STATus:QUESTionable:FREQuency:ENABle	32767
:STATus:QUESTionable:FREQuency:NTRansition	32767
:STATus:QUESTionable:FREQuency:PTRransition	32767
:STATus:QUESTionable:MODulation:ENABle	32767
:STATus:QUESTionable:MODulation:NTRansition	32767
:STATus:QUESTionable:MODulation:PTRransition	32767
:STATus:QUESTionable:POWEr:ENABle	32767
:STATus:QUESTionable:POWEr:NTRansition	32767
:STATus:QUESTionable:POWEr:PTRransition	32767

## Status Byte Group

The Status Byte Group includes the [Status Byte Register](#) and the [Service Request Enable Register](#).



ck721a



# Status Byte Register

Table 3-3 Status Byte Register Bits

Bit	Description
0,1	<b>Unused.</b> These bits are always set to 0.
2	<b>Error/Event Queue Summary Bit.</b> 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	<b>Data Questionable Status Summary Bit.</b> 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	<b>Message Available.</b> 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	<b>Standard Event Status Summary Bit.</b> 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	<b>Request Service (RQS) Summary Bit.</b> 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	<b>Standard Operation Status Summary Bit.</b> 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 3-1 on page 97](#) or [Figure 3-2 on page 98](#).

Example: 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

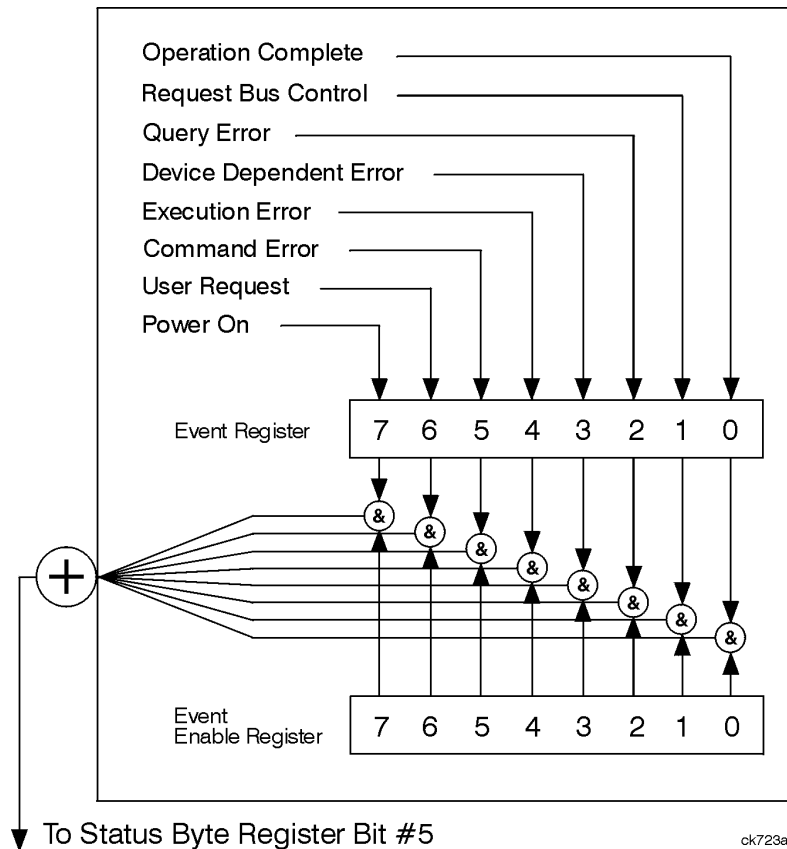
The [Standard Operation Status Group](#) and the [Data Questionable Status Group](#) consist of the registers listed below. The [Standard Event Status Group](#) is similar but does *not* have negative or positive transition filters or a condition register.

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 <a href="#">Status Byte Register</a> .

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 3-4 Standard Event Status Register Bits

Bit	Description
0	<b>Operation Complete.</b> A 1 in this bit position indicates that all pending signal generator operations were completed following execution of the *OPC command.
1	<b>Request Control.</b> This bit is always set to 0 (the signal generator does not request control).
2	<b>Query Error.</b> A 1 in this bit position indicates that a query error has occurred. Query errors have SCPI error numbers from -499 to -400.
3	<b>Device Dependent Error.</b> A 1 in this bit position indicates that a device dependent error has occurred. Device dependent errors have SCPI error numbers from -399 to -300 and 1 to 32767.

Table 3-4 Standard Event Status Register Bits

Bit	Description
4	<b>Execution Error.</b> A 1 in this bit position indicates that an execution error has occurred. Execution errors have SCPI error numbers from -299 to -200.
5	<b>Command Error.</b> A 1 in this bit position indicates that a command error has occurred. Command errors have SCPI error numbers from -199 to -100.
6	<b>User Request Key (Local).</b> A 1 in this bit position indicates that the <b>Local</b> key has been pressed. This is true even if the signal generator is in local lockout mode.
7	<b>Power On.</b> 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:                Enable bit 7 and bit 6 so that whenever either of those bits is 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)

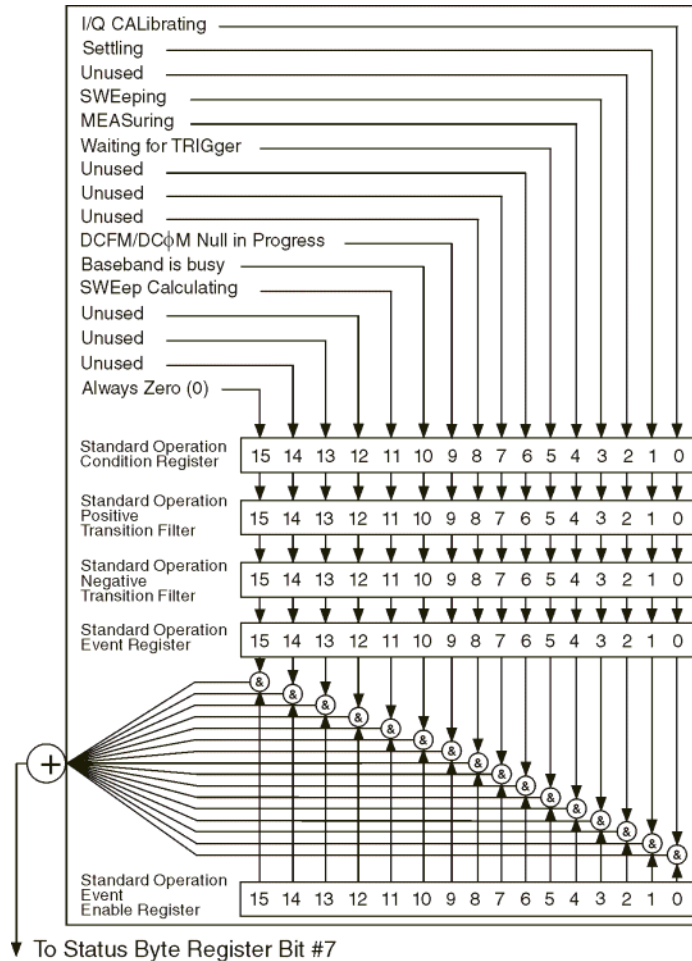
Query:                 \*ESE?

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

Standard Operation Status Group

**NOTE**    For the E8257D analog signal generator, some of the status bits will return a zero value if queried. These status bits are not active for the E8257D. For more information, refer to [Table 3-5 on page 109](#).

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 3-5 Standard Operation Condition Register Bits

Bit	Description
0 <sup>a</sup>	<b>I/Q Calibrating.</b> A 1 in this position indicates an I/Q calibration is in process. (E8267D only)
1	<b>Settling.</b> A 1 in this bit position indicates that the signal generator is settling.

Table 3-5 Standard Operation Condition Register Bits

Bit	Description
2	<b>Unused.</b> This bit position is set to 0.
3	<b>Sweeping.</b> A 1 in this bit position indicates that a sweep is in progress.
4	Measuring. A1 in this bit position indicates that a bit error rate test is in progress
5	<b>Waiting for Trigger.</b> A 1 in this bit position indicates that the source is in a “wait for trigger” state. When option 300 is enabled, a 1 in this bit position indicates that TCH/PDCH synchronization is established and waiting for a trigger to start measurements.
6,7,8	<b>Unused.</b> These bits are always set to 0.
9	<b>DCFM/DCΦM Null in Progress.</b> A 1 in this bit position indicates that the signal generator is currently performing a DCFM/DCΦM zero calibration.
10a	<b>Baseband is Busy.</b> A 1 in this bit position indicates that the baseband generator is communicating or processing. This is a summary bit. See the <a href="#">“Baseband Operation Status Group” on page 112</a> for more information.
11	<b>Sweep Calculating.</b> A 1 in this bit position indicates that the signal generator is currently doing the necessary pre-sweep calculations.
12, 13, 14	<b>Unused.</b> These bits are always set to 0.
15	<b>Always 0.</b>

a. On the E8257D, this bit is set to 0.

Query:                STATUS:OPERation:CONDition?

Response:            The *decimal* sum of the bits that are 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:        Enable bit 9 and bit 3 so that whenever either of those bits is 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` ( $512 + 8$ )

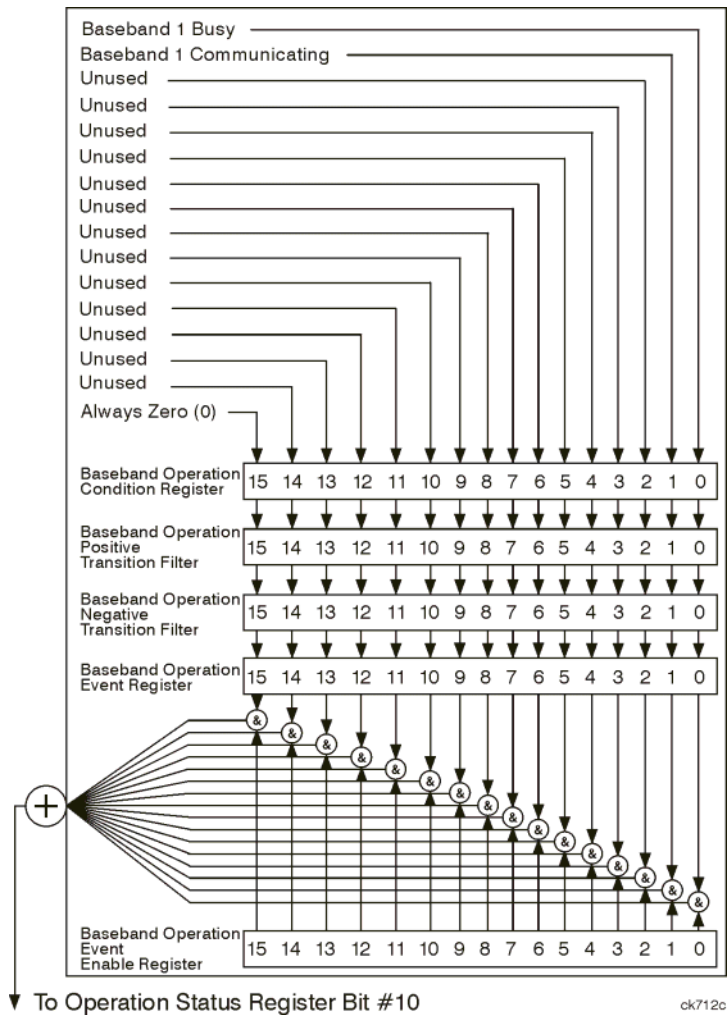
Query:            `STATUS:OPERation:ENABLE?`

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

## Baseband Operation Status Group

**NOTE** For the E8257D analog signal generator, the status bits will return a zero value if queried. This status group is not active for the E8257D.

The Baseband Operation Status Group is used to determine the specific event that set bit 10 in the [Standard Operation Status Group](#). This group consists of the [Baseband Operation Condition Register](#), the [Baseband Operation Transition Filters \(negative and positive\)](#), the [Baseband Operation Event Register](#), and the [Baseband Operation Event Enable Register](#).





## Baseband Operation Condition Register

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

**Table 3-6 Baseband Operation Condition Register Bits**

Bit	Description
0	<b>Baseband 1 Busy.</b> A 1 in this position indicates the signal generator baseband is active.
1	Baseband 1 Communicating. A 1 in this bit position indicates that the signal generator baseband generator is handling data I/O.
2–14	<b>Unused.</b> These bits are always set to 0.
15	<b>Always 0.</b>

Query:            `STATUS:OPERation:BASEband:CONDition?`

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

Example:         The decimal value 2 is returned. The decimal sum = 2 (bit 1).

## Baseband Operation Transition Filters (negative and positive)

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

Queries:           `STATUS:OPERation:BASEband:NTRansition?`  
                       `STATUS:OPERation:BASEband:PTRansition?`

## Baseband Operation Event Register

The Baseband 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 contents of that register.

Query:            `STATUS:OPERation:BASEband[:EVENT]?`

## Baseband Operation Event Enable Register

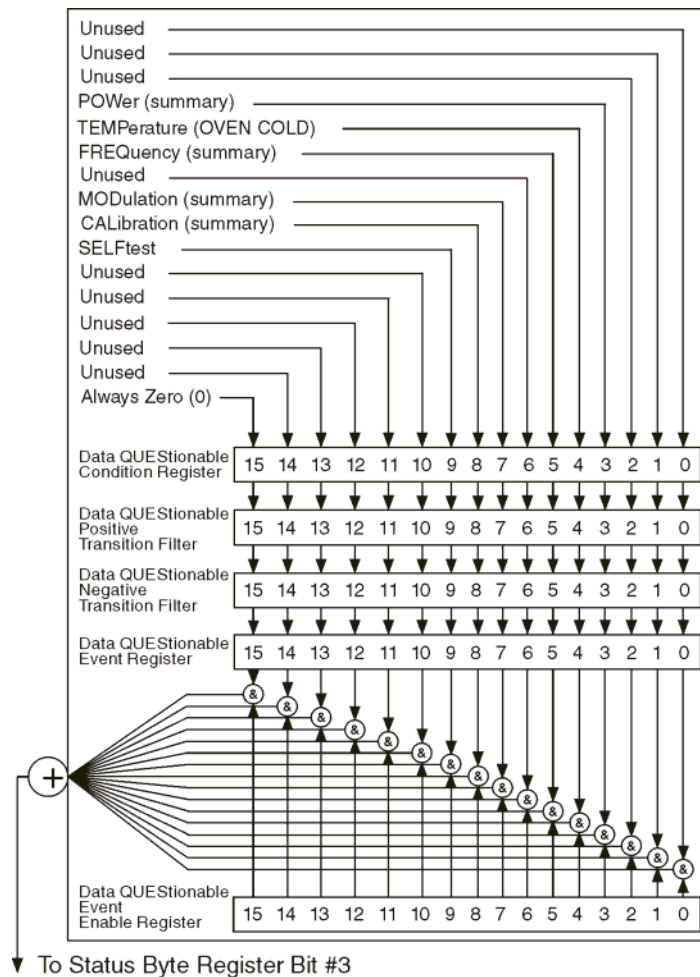
The Baseband Operation Event Enable Register lets you choose which bits in the Baseband Operation Event Register can set the summary bit (bit 10 of the Standard Operation Status Group).

Command:	STATUS:OPERation:BASEband:ENABle <value>, where <value> is the sum of the decimal values of the bits you want to enable.
Example:	Enable bit 0 and bit 1 so that whenever either of those bits are set to 1, the Baseband Operation Status summary bit of the Standard Operation Status Register is set to 1: send the command STAT:OPER:BAS:ENAB 520 (512 + 8)
Query:	STATUS:OPERation:BASEband:ENABle?
Response:	Decimal value of the sum of the bits previously enabled with the STATUS:OPERation:BASEband:ENABle <value> command.

## Data Questionable Status Group

**NOTE** For the E8257D analog signal generator, some of the status bits will return a zero value if queried. These status bits are not active for the E8257D. For more information, refer to [Table 3-7 on page 116](#)

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 3-7 Data Questionable Condition Register Bits

Bit	Description
0, 1, 2	<b>Unused.</b> These bits are always set to 0.
3	<b>Power (summary).</b> 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 <a href="#">“Data Questionable Power Status Group” on page 118</a> for more information.
4	<b>Temperature (OVEN COLD).</b> A 1 in this bit position indicates that the internal reference oscillator (reference oven) is cold.
5	<b>Frequency (summary).</b> 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 <a href="#">“Data Questionable Frequency Status Group” on page 121</a> .
6	<b>Unused.</b> This bit is always set to 0.
7	<b>Modulation (summary).</b> 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, modulation uncalibrated. See the <a href="#">“Data Questionable Modulation Status Group” on page 124</a> for more information.
8 <sup>a</sup>	<b>Calibration (summary).</b> This is a summary bit taken from the QUESTionable:CALibration register. A 1 in this bit position indicates that one of the following may have happened: an error has occurred in the DCFM/DCΦM zero calibration, an error has occurred in the I/Q calibration. See the <a href="#">“Data Questionable Calibration Status Group” on page 127</a> for more information.
9	<b>Self Test.</b> A 1 in this bit position indicates that a self-test has failed during power-up. This bit can only be cleared by cycling the signal generator’s line power. *CLS will not clear this bit.
10–14	<b>Unused.</b> These bits are always set to 0.
15	<b>Always 0.</b>

a. On the E8257D, this bit is set to 0.

Query:                STATUS:QUESTionable:CONDition?

Response:            The *decimal* sum of the bits that are 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 contents 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>` command where `<value>` is the sum of the decimal values of the bits you want to enable.

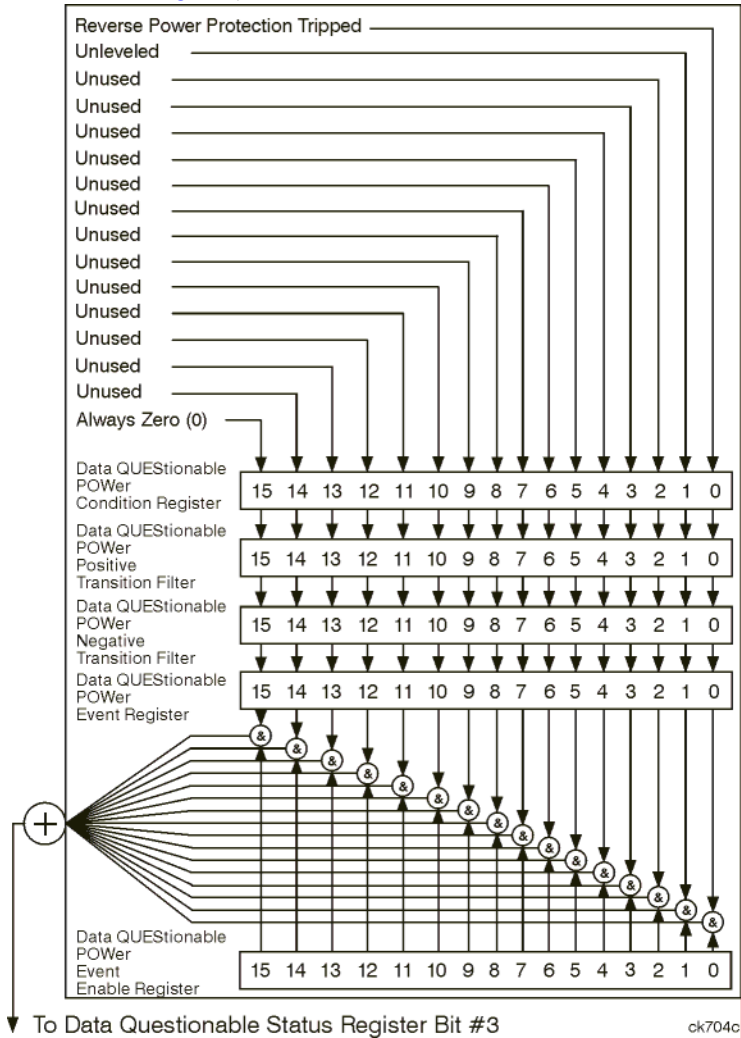
Example:         Enable bit 9 and bit 3 so that whenever either of those bits is set to 1, the Data Questionable Status summary bit of the Status Byte Register is set to 1:  
                      send the command `STAT:QUES:ENAB 520` ( $512 + 8$ )

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

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 3-8 Data Questionable Power Condition Register Bits**

Bit	Description
0	<b>Reverse Power Protection Tripped.</b> 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. The RPP circuit can be reset and bit 0 set to 0, by sending the remote SCPI command: <code>OUTput:PROTection:CLEar</code> .
1	<b>Unleveled.</b> A 1 in this bit indicates that the output leveling loop is unable to set the output power.
2–14	<b>Unused.</b> These bits are always set to 0.
15	<b>Always 0.</b>

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 contents 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.

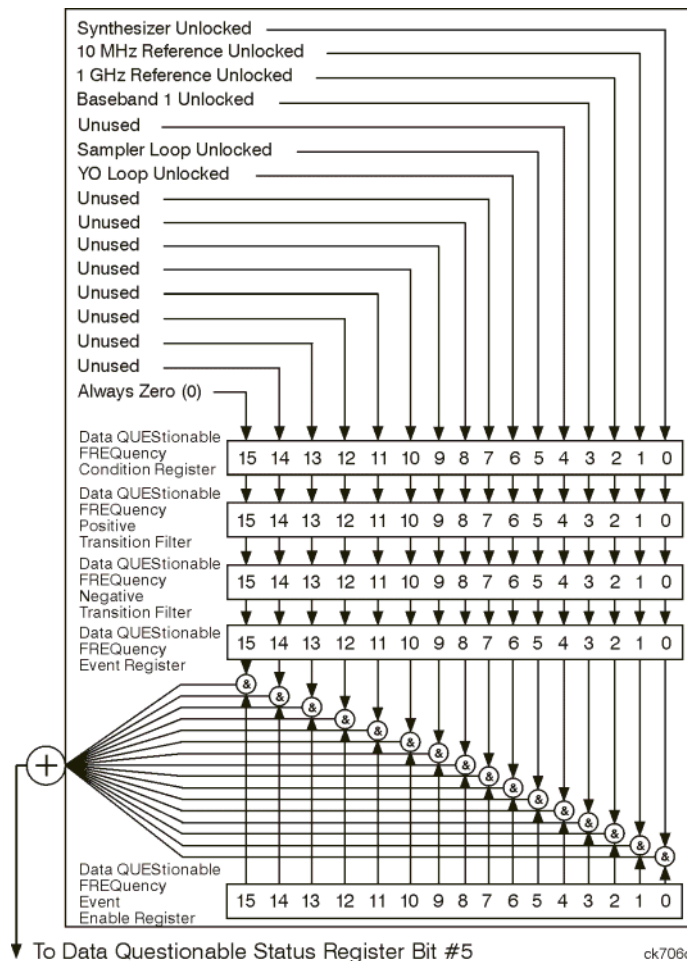
Command:	<code>STATUS:QUESTIONable:POWER:ENABLE &lt;value&gt;</code> command where <value> is the sum of the decimal values of the bits you want to enable
Example:	Enable bit 9 and bit 3 so that whenever either of those bits is set to 1, the Data Questionable Power summary bit of the Data Questionable Condition Register is set to 1: send the command <code>STAT:QUES:POW:ENAB 520 (512 + 8)</code>
Query:	<code>STATUS:QUESTIONable:POWER:ENABLE?</code>
Response:	Decimal value of the sum of the bits previously enabled with the <code>STATUS:QUESTIONable:POWER:ENABLE &lt;value&gt;</code> command.



## Data Questionable Frequency Status Group

**NOTE** For the E8257D analog signal generator, some of the status bits will return a zero value if queried. These status bits are not active for the E8257D. For more information, refer to [Table 3-9 on page 122](#).

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 3-9 Data Questionable Frequency Condition Register Bits

Bit	Description
0	<b>Synth. Unlocked.</b> A 1 in this bit indicates that the synthesizer is unlocked.
1	<b>10 MHz Ref Unlocked.</b> A 1 in this bit indicates that the 10 MHz reference signal is unlocked.
2	<b>1 Ghz Ref Unlocked.</b> A 1 in this bit indicates that the 1 Ghz reference signal is unlocked.
3 <sup>a</sup>	<b>Baseband 1 Unlocked.</b> A 1 in this bit indicates that the baseband 1 generator is unlocked.
4	<b>Unused.</b> This bit is always set to 0.
5	<b>Sampler Loop Unlocked.</b> A 1 in this bit indicates that the sampler loop is unlocked.
6	<b>YO Loop Unlocked.</b> A 1 in this bit indicates that the YO loop is unlocked.
7–14	<b>Unused.</b> These bits are always set to 0.
15	<b>Always 0.</b>

a. On the E8257D, this bit is set to 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]?`

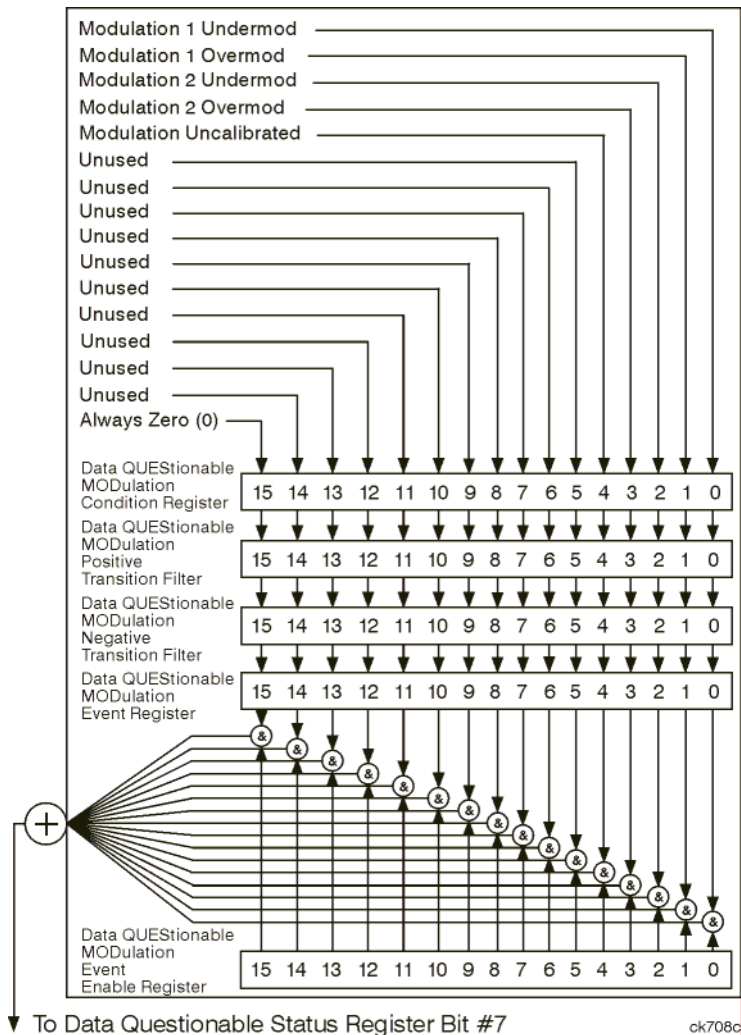
## Data Questionable Frequency Event Enable Register

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 9 and bit 3 so that whenever either of those bits is 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 (512 + 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 3-10 Data Questionable Modulation Condition Register Bits

Bit	Description
0	<b>Modulation 1 Undermod.</b> A 1 in this bit indicates that the External 1 input, ac coupling on, is less than 0.97 volts.
1	<b>Modulation 1 Overmod.</b> A 1 in this bit indicates that the External 1 input, ac coupling on, is greater than 1.03 volts.
2	<b>Modulation 2 Undermod.</b> A 1 in this bit indicates that the External 2 input, ac coupling on, is less than 0.97 volts.
3	<b>Modulation 2 Overmod.</b> A 1 in this bit indicates that the External 2 input, ac coupling on, is greater than 1.03 volts.
4	<b>Modulation Uncalibrated.</b> A 1 in this bit indicates that modulation is uncalibrated.
5–14	<b>Unused.</b> These bits are always set to 0.
15	<b>Always 0.</b>

Query:           STATUS:QUESTionable:MODulation:CONDition?

Response:       The *decimal* sum of the bits that are 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 contents of that register.

Query:           STATUS:QUESTionable:MODulation[:EVENT]?

## Data Questionable Modulation Event Enable Register

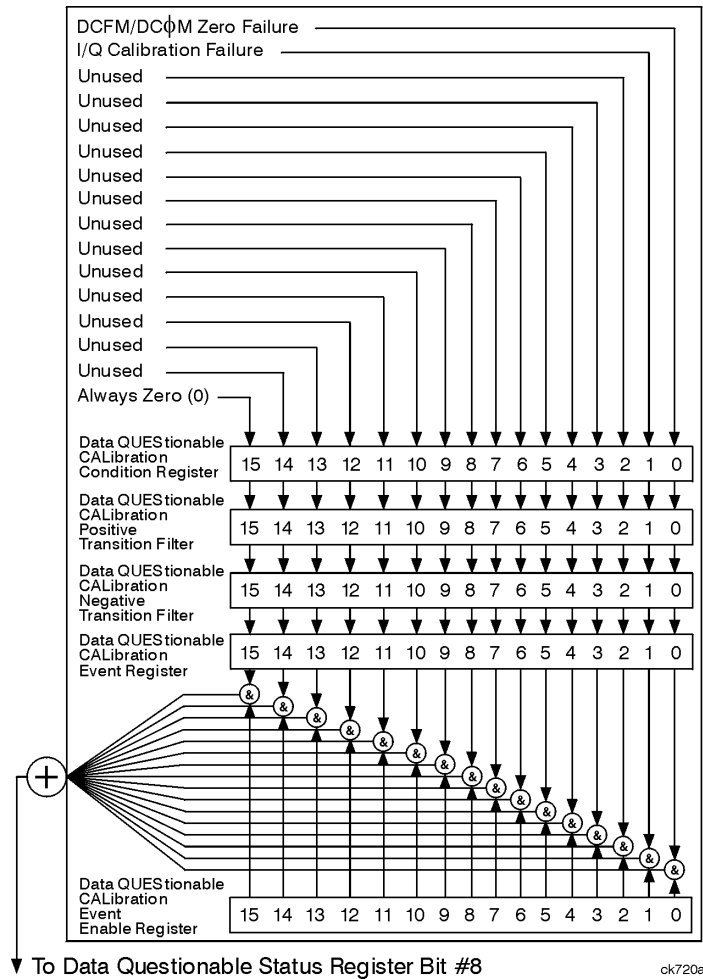
The Data Questionable Modulation Event Enable Register lets you choose which bits in the Data Questionable Modulation Event Register set the summary bit (bit 7 of the Data Questionable Condition Register) to 1.

Command:	STATus:QUESTionable:MODulation:ENABle <value> command 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 is 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 (512 + 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** For the E8257D analog signal generator, some of the status bits will return a zero value if queried. These status bits are not active for the E8257D. For more information, refer to [Table 3-11 on page 128](#).

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 3-11 Data Questionable Calibration Condition Register Bits

Bit	Description
0	<b>DCFM/DCΦM Zero Failure.</b> A 1 in this bit indicates that the DCFM/DCΦM zero calibration routine has failed. This is a critical error. The output of the source is not valid until the condition of this bit is 0.
1 <sup>a</sup>	<b>I/Q Calibration Failure.</b> A 1 in this bit indicates that the I/Q modulation calibration experienced a failure.
2–14	<b>Unused.</b> These bits are always set to 0.
15	<b>Always 0.</b>

a. On the E8257D, this bit is set to 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 9 and bit 3 so that whenever either of those bits is 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 (512 + 8)
Query:	STATus:QUESTionable:CALibration:ENABle?
Response:	Decimal value of the sum of the bits previously enabled with the STATus:QUESTionable:CALibration:ENABle <value> command.



---

## 4 Creating and Downloading Waveform Files

This chapter explains how to create Arb-based waveform data and download it into the signal generator:

- [“Overview” on page 131](#)
- [“Understanding Waveform Data” on page 132](#)
- [“Waveform Structure” on page 139](#)
- [“Waveform Phase Continuity” on page 142](#)
- [“Waveform Memory” on page 144](#)
- [“Commands for Downloading and Extracting Waveform Data” on page 146](#)
- [“Creating Waveform Data” on page 152](#)
- [“Downloading Waveform Data” on page 157](#)
- [“Loading, Playing, and Verifying a Downloaded Waveform” on page 164](#)
- [“Using the Download Utilities” on page 166](#)
- [“Downloading E443xB Signal Generator Files” on page 166](#)
- [“Programming Examples” on page 169](#)
- [“Troubleshooting Waveform Files” on page 214](#)

### Overview

---

**NOTE** Creating and downloading waveform data is available only in E8267D PSG Vector Signal Generators with Option 601 or 602.

---

The signal generator lets you download and extract waveform files. You can create these files either external to the signal generator or by using one of the internal modulation formats. The signal generator also accepts waveform files created for the earlier E443xB ESG signal generator models. For file extractions, the signal generator encrypts the waveform file information. The exception to encrypted file extraction is user-created I/Q data. The signal generator lets you extract this type of file unencrypted. After extracting a waveform file, you can download it into another Agilent signal generator that has the same option or software license required to play it. Waveform files consist of three items:

- I/Q data
- Marker data
- File header

The signal generator automatically creates the marker file and the file header if the two items are *not* part of the download. In this situation, the signal generator sets the file header information to unspecified (no settings saved) and sets all markers to zero (off).

There are two ways to download waveform files, programmatically or using one of three available free download utilities created by Agilent Technologies:

- Intuilink for PSG/ESG Signal Generators  
[www.agilent.com/find/intuilink](http://www.agilent.com/find/intuilink)
- PSG/ESG Download Assistant for use only with MATLAB®  
[www.agilent.com/find/downloadassistant](http://www.agilent.com/find/downloadassistant)<sup>1</sup>
- N7622A Signal Studio Toolkit  
[www.agilent.com/find/signalstudio](http://www.agilent.com/find/signalstudio)

## Waveform Data Requirements

To be successful in downloading files, you must first create the data in the required format.

- Signed 2's complement
- 2-byte integer values
- Input data range of –32768 to 32767
- Minimum of 60 samples per waveform (60 I and 60 Q data points)
- Interleaved I and Q data
- Big endian byte order
- The same name for the marker and I/Q file

This is only a requirement if you create and download a marker file, otherwise the signal generator automatically creates the marker file using the I/Q data file name. For more information, see “[Waveform Structure](#)” on page 139.

For more information on waveform data, see “[Understanding Waveform Data](#)” on page 132.

## Understanding Waveform Data

The signal generator accepts binary data formatted into a binary I/Q file. This section explains the necessary components of the binary data, which uses ones and zeros to represent a value.

### Bits and Bytes

Binary data uses the base-two number system. The location of each bit within the data represents a value that uses base two raised to a power ( $2^{n-1}$ ). The exponent is  $n - 1$  because the first position is zero. The first bit position, zero, is located at the far right. To find the decimal value of the binary data, sum the value of each location:

---

1. MATLAB is a U.S. registered trademark of The Math Works, Inc.

$$\begin{aligned}
 1101 &= (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) \\
 &= (1 \times 8) + (1 \times 4) + (0 \times 2) + (1 \times 1) \\
 &= 13 \text{ (decimal value)}
 \end{aligned}$$

Notice that the exponent identifies the bit position within the data, and we read the data from right to left.

The signal generator accepts data in the form of bytes. Bytes are groups of eight bits:

$$\begin{aligned}
 01101110 &= (0 \times 2^7) + (1 \times 2^6) + (1 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) \\
 &= 110 \text{ (decimal value)}
 \end{aligned}$$

The maximum value for a single unsigned byte is 255 (11111111 or  $2^8-1$ ), but you can use multiple bytes to represent larger values. The following shows two bytes and the resulting integer value:

$$01101110 \ 10110011 = 28339 \text{ (decimal value)}$$

The maximum value for two unsigned bytes is 65535. Since binary strings lengthen as the value increases, it is common to show binary values using hexadecimal (hex) values (base 16), which are shorter. The value 65535 in hex is FFFF. Hexadecimal consists of the values 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. In decimal, hex values range from 0 to 15 (F). It takes 4 bits to represent a single hex value.

1 = 0001	2 = 0010	3 = 0011	4 = 0100	5 = 0101
6 = 0110	7 = 0111	8 = 1000	9 = 1001	A = 1010
B = 1011	C = 1100	D = 1101	E = 1110	F = 1111

For I and Q data, the signal generator uses two bytes to represent an integer value.

## LSB and MSB (Bit Order)

Within groups (strings) of bits, we designate the order of the bits by identifying which bit has the highest value and which has the lowest value by its location in the bit string. The following is an example of this order.

Most Significant Bit (MSB)	This bit has the highest value (greatest weight) and is located at the far left of the bit string.
Least Significant Bit (LSB)	This bit has the lowest value (bit position zero) and is located at the far right of the bit string.

Bit Position	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data	1	0	1	1	0	1	1	1	1	1	0	1	0	0	1	1

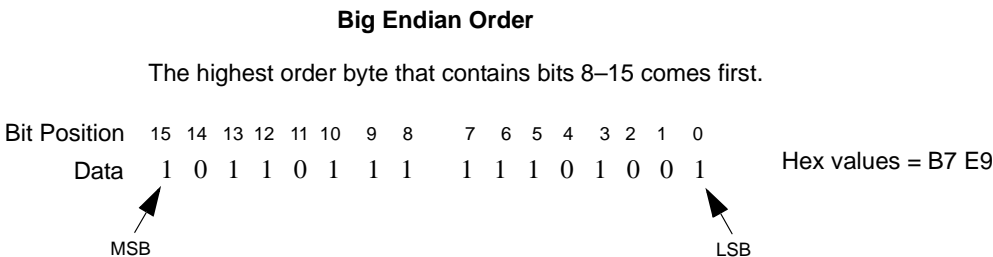
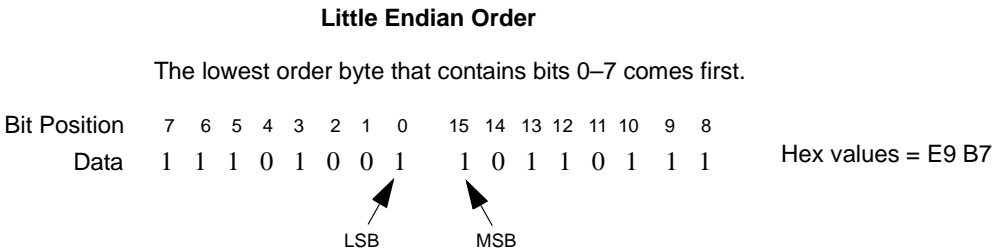
  

MSB	LSB
-----	-----

Because we are using 2-bytes of data, the MSB appears in the second byte.

## Little Endian and Big Endian (Byte Order)

When you use multiple bytes (as required for the waveform data), you must identify their order. This is similar to identifying the order of bits by LSB and MSB. To identify byte order, use the terms little endian and big endian. These terms are used by designers of computer processors.



Notice in the previous figure that the LSB and MSB positioning changes with the byte order. In little endian order, the LSB and MSB are next to each other in the bit sequence.

---

**NOTE** For I/Q data downloads, the signal generator requires big endian order. For each I/Q data point, the signal generator uses four bytes (two integer values), two bytes for the I point and two bytes for the Q point.

---

The byte order, little endian or big endian, depends on the type of processor used with your development platform. Intel<sup>1</sup> processors and its clones use little endian. Sun<sup>™</sup> and Motorola processors use big endian. The Apple PowerPC processor, while big endian oriented, also supports the little endian order. Always refer to the processor's manufacturer to determine the order they use for bytes, and if they support both, how to ensure that you are using the correct byte order.

Development platforms include any product that creates and saves waveform data to a file. This includes Agilent Technologies Advanced Design System EDA software, C++, MATLAB, and so forth. The byte order describes how the system processor stores integer values as binary data in memory.

---

1. Intel is a U.S. registered trademark of Intel Corporation.

---

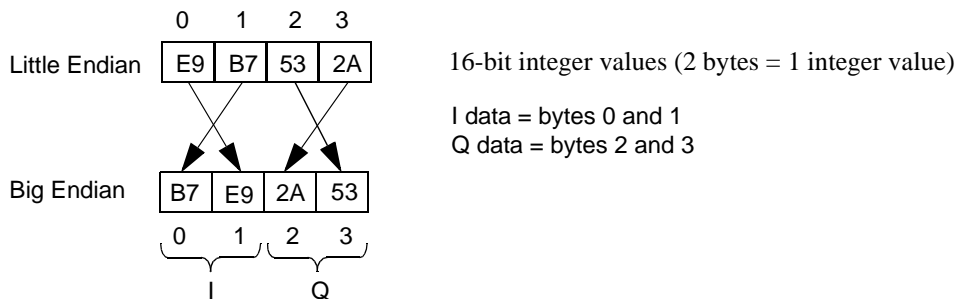
Sun is a trademark or registered trademark of Sun Microsystems, Inc. in the U.S. and other countries.

If you output data from a little endian system to a text file (ASCII text), the values are the same as viewed from a big endian system. The order only becomes important when you use the data in binary format, as is done when downloading data to the signal generator.

## Byte Swapping

While the processor for the development platform determines the byte order, the recipient of the data may require the bytes in the reverse order. In this situation, you must reverse the byte order before downloading the data. This is commonly referred to as byte swapping. You can swap bytes either programmatically or by using the Agilent Technologies IntuiLink for PSG/ESG Signal Generators software. For the signal generator, byte swapping is the method to change the byte order of little endian to big endian. For more information on little endian and big endian order, see [“Little Endian and Big Endian \(Byte Order\)”](#) on page 134.

The following figure shows the concept of byte swapping for the signal generator. Remember that we can represent data in hex format (4 bits per hex value), so each byte (8 bits) in the figure shows two example hex values.



To correctly swap bytes, you must group the data to maintain the I and Q values. One common method is to break the two-byte integer into one-byte character values (0–255). Character values use 8 bits (1 byte) to identify a character. Remember that the maximum unsigned 8-bit value is 255 ( $2^8 - 1$ ). Changing the data into character codes groups the data into bytes. The next step is then to swap the bytes to align with big endian order.

---

**NOTE** The signal generator always assumes that downloaded data is in big endian order, so there is no data order check. Downloading data in little endian order will produce an undesired output signal.

---

## DAC Input Values

The signal generator uses a 16-bit DAC (digital-to-analog convertor) to process each of the 2-byte integer values for the I and Q data points. The DAC determines the range of input values required from the I/Q data. Remember that with 16-bits we have a range of 0–65535, but the signal generator divides this range between positive and negative values:

- 32767 = positive full scale output
- 0 = 0 volts
- –32768 = negative full scale output

Because the DAC's range uses both positive and negative values, the signal generator requires signed input values. The following list illustrates the DAC's input value range.

<u>Voltage</u>	<u>DAC Range</u>	<u>Input Range</u>	<u>Binary Data</u>	<u>Hex Data</u>
Vmax	65535	32767	01111111 11111111	7FFF
⋮	⋮	⋮	⋮	⋮
⋮	32768	1	00000000 00000001	0001
0 Volts	32767	0	00000000 00000000	0000
⋮	32766	-1	11111111 11111111	FFFF
⋮	⋮	⋮	⋮	⋮
Vmin	0	-32768	10000000 00000000	8000

Notice that it takes only 15 bits ( $2^{15}$ ) to reach the Vmax (positive) or Vmin (negative) values. The MSB determines the sign of the value. This is covered in [“2's Complement Data Format” on page 138](#).

**Using E443xB ESG DAC Input Values**

The signal generator's input values differ from those of the earlier E443xB ESG models. For the E443xB models, the input values are all positive (unsigned) and the data is contained within 14 bits plus 2 bits for markers. This means that the E443xB DAC has a smaller range:

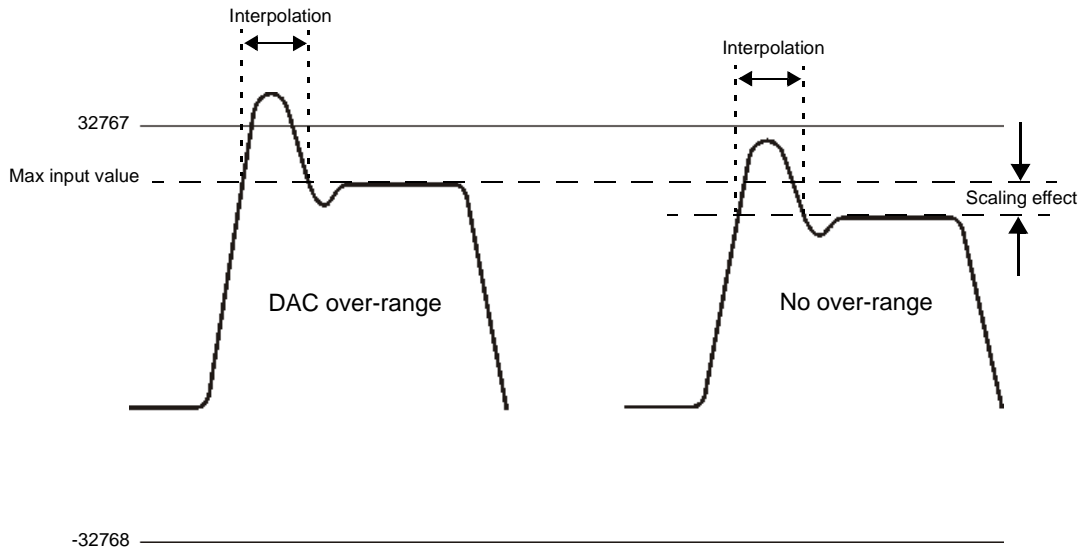
- 0 = negative full scale output
- 8192 = 0 volts
- 16383 = positive full scale output

Although the signal generator uses signed input values, it accepts unsigned data created for the E443xB and converts it to the proper DAC values. To download an E443xB files to the signal generator, use the same command syntax as for the E443xB models. For more information on downloading E443xB files, see [“Downloading E443xB Signal Generator Files” on page 166](#).

**Scaling DAC Values**

The signal generator uses an interpolation algorithm (sampling between the I/Q data points) when reconstructing the waveform. For common waveforms, this interpolation can cause overshoot, which may create a DAC over-range error condition. Because of the interpolation, the error condition can occur even when all the I and Q values are within the DAC input range. To avoid the DAC over-range problem, you must scale (reduce) the I and Q input values, so that any overshoot remains within the DAC range.





There is no single scaling value that is optimal for all waveforms. To achieve the maximum dynamic range, select the largest scaling value that does not result in a DAC over-range error. There are two ways to scale the I/Q data:

- Reduce the input values for the DAC.
- Use the SCPI command `:RADio:ARB:RSCaling <val>` or the front-panel keys, **Mode > Dual ARB > ARB Setup > More (1 of 2) > Waveform Runtime Scaling**, to set the waveform amplitude as a percentage of full scale.

---

**NOTE** The signal generator comes from the factory with scaling set to 70%. If you reduce the DAC input values, ensure that you set the signal generator scaling (`:RADio:ARB:RSCaling`) to an appropriate setting that accounts for the reduced values.

---

To further minimize overshoot problems, use the correct FIR filter for your signal type and adjust your sample rate to accommodate the filter response.

## 2's Complement Data Format

The signal generator requires signed values for the input data. For binary data, two's complement is a way to represent positive and negative values. The most significant bit (MSB) determines the sign.

- 0 equals a positive value (01011011 = 91 decimal)
- 1 equals a negative value (10100101 = -91 decimal)

Like decimal values, if you sum the binary positive and negative values, you get zero. The one difference with binary values is that you have a carry, which is ignored. The following shows how to calculate the two's complement using 16-bits. The process is the same for both positive and negative values.

Convert the decimal value to binary.

```
23710 = 01011100 10011110
```

Notice that 15 bits (0-14) determine the value and bit 16 (MSB) indicates a positive value.

Invert the bits (1 becomes 0 and 0 becomes 1).

```
10100011 01100001
```

Add one to the inverted bits. Adding one makes it a two's complement of the original binary value.

```
10100011 01100001
+ 00000000 00000001
10100011 01100010
```

The MSB of the resultant is one, indicating a negative value (-23710).

Test the results by summing the binary positive and negative values; when correct, they produce zero.

```
01011100 10011110
+ 10100011 01100001
00000000 00000000
```

## I and Q Interleaving

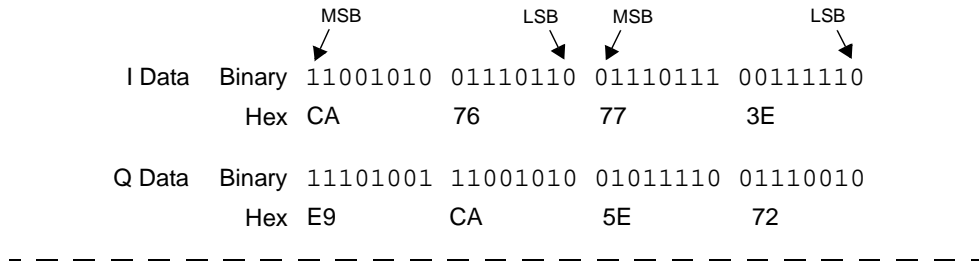
When you create the waveform data, the I and Q data points typically reside in separate arrays or files. The signal generator requires a single I/Q file for waveform data playback. The process of interleaving creates a single array with alternating I and Q data points, with the Q data following the I data. This array is then downloaded to the signal generator as a binary file. The interleaved file comprises the waveform data points where each set of data points, one I data point and one Q data point, represents one I/Q waveform point.

---

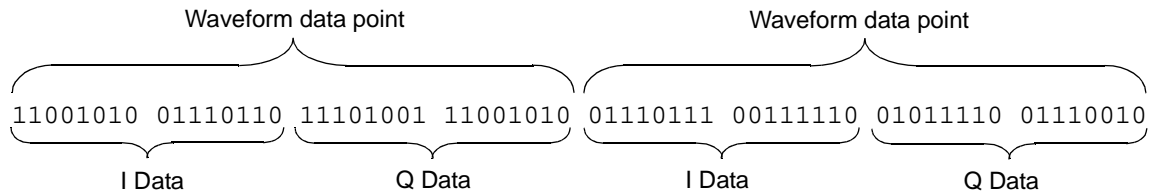
**NOTE** The signal generator can accept separate I and Q files created for the earlier E443xB ESG models. For more information on downloading E443xB files, see [“Downloading E443xB Signal Generator Files” on page 166](#).

---

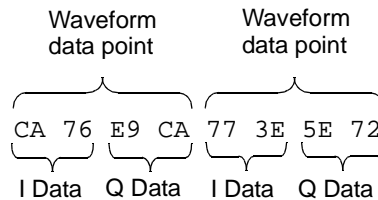
The following figure illustrates interleaving I and Q data. Remember that it takes two bytes (16 bits) to represent one I or Q data point.



### Interleaved Binary Data



### Interleaved Hex Data



## Waveform Structure

To play back waveforms, the signal generator uses data from the following three files:

- File header
- Marker file
- I/Q file

All three files have the same name, the name of the I/Q data file, but the signal generator stores each file in its respective directory (headers, markers, and waveform). When you extract the waveform file (I/Q data file), it includes the other two files, so there is no need to extract each one individually. For more information on file extractions, see [“Commands for Downloading and Extracting Waveform Data”](#) on page 146.

File Header

The file header contains settings for the ARB modulation format such as sample rate, marker polarity, I/Q modulation attenuator setting and so forth. When you create and download I/Q data, the signal generator automatically creates a file header with all saved parameters set to unspecified. With unspecified header settings, the waveform either uses the signal generator default settings, or if a waveform was previously played, the settings from that waveform. Ensure that you configure and save the file header settings for each waveform. Refer to the *E8257D/67D PSG Signal Generators User’s Guide* for more information on file headers

---

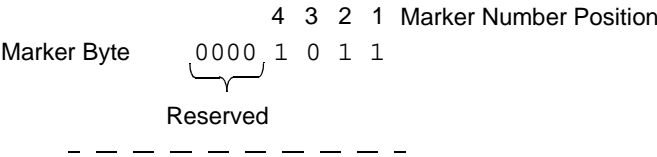
**NOTE** If you have no RF output when you play back a waveform, ensure that the marker RF blanking function has not been set for any of the markers. The marker RF blanking function is a header parameter that can be inadvertently set active for a marker by a previous waveform.

---

Marker File

The marker file uses one byte per I/Q waveform point to set the state of the four markers either on (1) or off (0) for each I/Q point. When a marker is active (on), it provides an output trigger signal to the rear panel EVENT connector that corresponds to the active marker number. Because markers are set at each waveform point, the marker file contains the same number of bytes as there are waveform points. For example, for 200 waveform points, the marker file contains 200 bytes.

Although a marker point is one byte, the signal generator uses only bits 0–3 to configure the markers; bits 4–7 are reserved and set to zero. The following example shows a marker byte.



Example of Setting a Marker Byte

Binary    0000 0101

Hex      05

Sets markers 1 and 3 on for a waveform point

The following example shows a marker binary file (all values in hex) for a waveform with 200 points. Notice the first marker point, 0f, shows all four markers on for only the first waveform point.

00000000:	0f 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01	0f = All markers on
00000010:	01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01	01 = Marker 1 on
00000020:	01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01	
00000030:	01 05 05 05 05 05 05 05 05 05 05 05 05 05 05 05	05 = Markers 1 and 3 on
00000040:	05 05 05 05 05 05 05 05 05 05 05 05 05 05 05 05	04 = Marker 3 on
00000050:	05 05 05 05 05 05 05 05 05 05 05 05 05 05 05 05	
00000060:	05 05 05 05 04 04 04 04 04 04 04 04 04 04 04 04	00 = No active markers
00000070:	04 04 04 04 04 04 04 04 04 04 04 04 04 04 04 04	
00000080:	04 04 04 04 04 04 04 04 04 04 04 04 04 04 04 04	
00000090:	04 04 04 04 04 04 00 00 00 00 00 00 00 00 00 00	
000000a0:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
000000b0:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
000000c0:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

If you create your own marker file, its name must be the same as the waveform file. If you download I/Q data without a marker file, the signal generator automatically creates a marker file with all points set to zero. For more information on markers, see the *E8257D/67D PSG Signal Generators User's Guide*.

---

**NOTE** Downloading marker data using a file name that currently resides on the signal generator overwrites the existing marker file without affecting the I/Q (waveform) file. However downloading just the I/Q data with the same file name as an existing I/Q file also overwrites the existing marker file setting all bits to zero.

---

I/Q File

The I/Q file contains the interleaved I and Q data points (signed 16-bit integers for each I and Q data point). Each I/Q point equals one waveform point. The signal generator stores the I/Q data in the waveform directory.

---

**NOTE** If you download I/Q data using a file name that currently resides on the signal generator, it also overwrites the existing marker file setting all bits to zero and the file header setting all parameters to unspecified.

---

Waveform

A waveform consists of samples. When you select a waveform for playback, the signal generator loads settings from the file header and creates the waveform samples from the data in the marker and I/Q (waveform) files. The file header, while required, does not affect the number of bytes that compose a waveform sample. One sample contains five bytes:

I/Q Data	+	Marker Data	=	1 Waveform Sample
2 bytes I 2 bytes Q (16 bits) (16 bits)		1 byte (8 bits) Bits 4–7 reserved—Bits 0–3 set		5 bytes

To create a waveform, the signal generator requires a minimum of 60 samples. To help minimize signal imperfections, use an even number of samples (for information on waveform continuity, see [“Waveform Phase Continuity” on page 142](#)). When you store waveforms, the signal generator saves changes to the waveform file, marker file, and file header.

## Waveform Phase Continuity

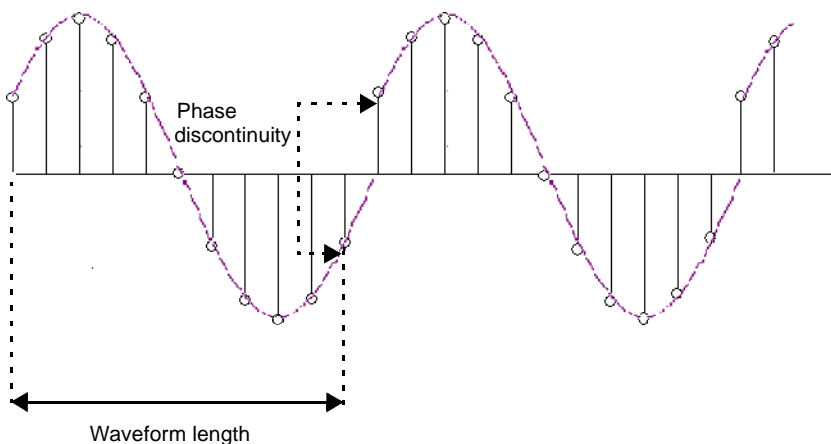
### Phase Discontinuity, Distortion, and Spectral Regrowth

The most common arbitrary waveform generation use case is to play back a waveform that is finite in length and repeat it continuously. Although often overlooked, a phase discontinuity between the end of a waveform and the beginning of the next repetition can lead to periodic spectral regrowth and distortion.

For example, the sampled sinewave segment in the following figure may have been simulated in software or captured off the air and sampled. It is an accurate sinewave for the time period it occupies, however the waveform does not occupy an entire period of the sinewave or some multiple thereof. Therefore, when repeatedly playing back the waveform by an arbitrary waveform generator, a phase discontinuity is introduced at the transition point between the beginning and the end of the waveform.

Repetitions with abrupt phase changes result in high frequency spectral regrowth. In the case of playing back the sinewave samples, the phase discontinuity produces a noticeable increase in distortion components in addition to the line spectra normally representative of a single sinewave.

**Sampled Sinewave with Phase Discontinuity**



## Avoiding Phase Discontinuities

You can easily avoid phase discontinuities for periodic waveforms by simulating an integer number of cycles when you create your waveform segment.

---

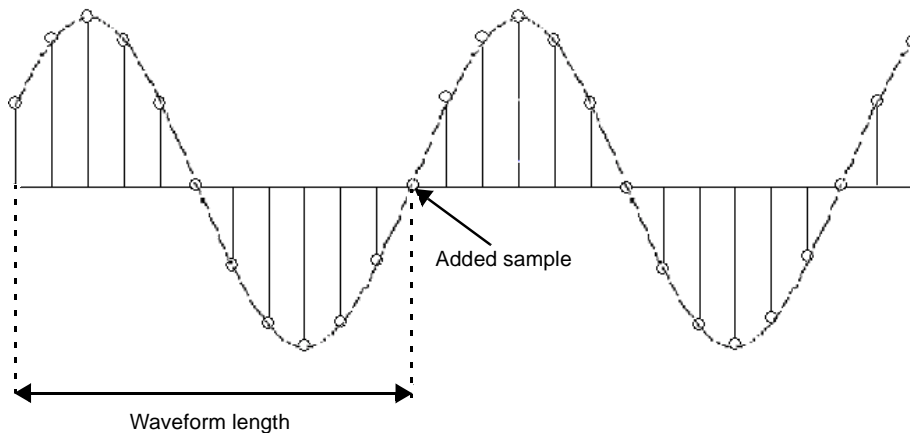
**NOTE** If there are  $N$  samples in a complete cycle, only the first  $N-1$  samples are stored in the waveform segment. Therefore, when continuously playing back the segment, the first and  $N$ th waveform samples are always the same, preserving the periodicity of the waveform.

---

By adding off time at the beginning of the waveform and subtracting an equivalent amount of off time from the end of the waveform, you can address phase discontinuity for TDMA or pulsed periodic waveforms. Consequently, when the waveform repeats, the lack of signal present avoids the issue of phase discontinuity.

However, if the period of the waveform exceeds the waveform playback memory available in the arbitrary waveform generator, a periodic phase discontinuity could be unavoidable. N5110B Baseband Studio for Waveform Capture and Playback alleviates this concern because it does not rely on the signal generator waveform memory. It streams data either from the PC hard drive or the installed PCI card for N5110B enabling very large data streams. This eliminates any restrictions associated with waveform memory to correct for repetitive phase discontinuities. Only the memory capacity of the hard drive or the PCI card limits the waveform size.

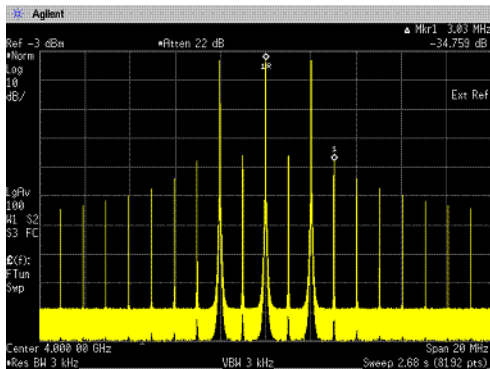
### Sampled Sinewave with No Discontinuity



The following figures illustrate the influence a single sample can have. The generated 3-tone test signal requires 100 samples in the waveform to maintain periodicity for all three tones. The measurement on the left shows the effect of using the first 99 samples rather than all 100 samples. Notice all the distortion products (at levels up to  $-35$  dBc) introduced in addition to the wanted

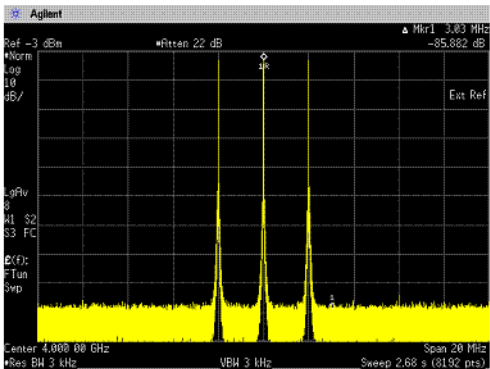
3-tone signal. The measurement on the right shows the same waveform using all 100 samples to maintain periodicity and avoid a phase discontinuity. Maintaining periodicity removes the distortion products.

Phase Discontinuity



3-tone - 20 MHz Bandwidth  
Measured distortion = 35 dBc

Phase Continuity



3-tone - 20 MHz Bandwidth  
Measured distortion = 86 dBc

## Waveform Memory

The signal generator provides two types of memory, volatile and non-volatile. You can download files to either memory type.

### Volatile

Random access memory that does not survive cycling of the signal generator power. This memory is commonly referred to as waveform memory (WFM1) or waveform playback memory. To play back waveforms, they must reside in volatile memory. The following file types share this memory:

- I/Q
- marker
- file header
- user PRAM
- waveform sequences (multiple I/Q files played together)

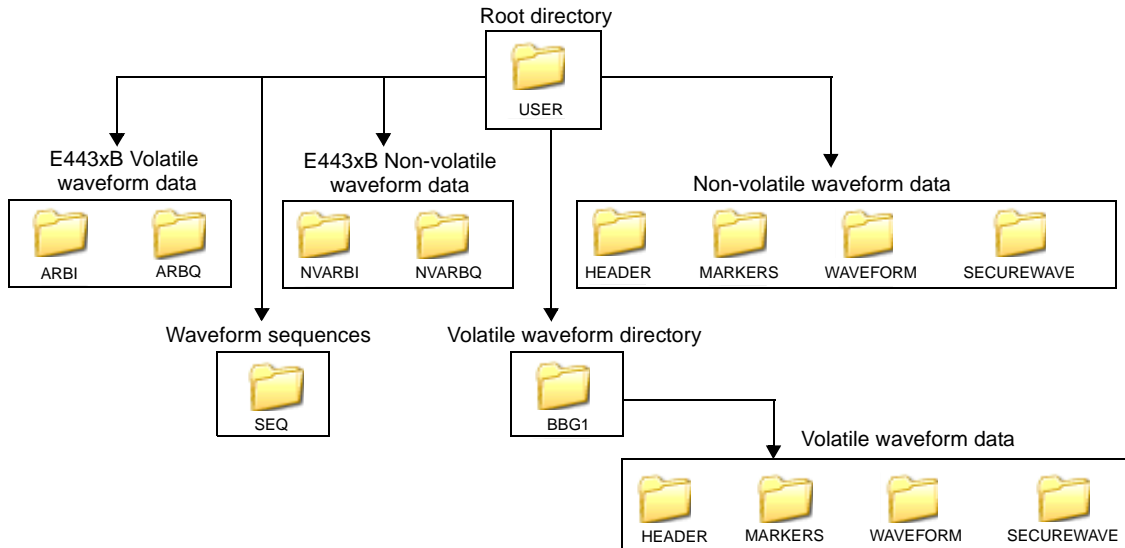
### Non-volatile

Storage memory where files survive cycling the signal generator power. Files remain until overwritten or deleted. To play back waveforms after cycling the signal generator power, you must load waveforms from non-volatile waveform memory (NVWFM) to volatile waveform memory (WFM1). The following file types share this memory:

- I/Q
- marker
- file header
- instrument state
- user data
- user PRAM
- sweep list
- waveform sequences (multiple I/Q files played together)



The following figure shows the locations within the signal generator for volatile and non-volatile waveform data.



## Memory Allocation

### Volatile Memory

The signal generator allocates volatile memory in blocks of 1024 bytes. For example, a waveform file with 60 samples (the minimum number of samples) has 300 bytes (5 bytes per sample  $\times$  60 samples), but the signal generator allocates 1024 bytes of memory. If a waveform is too large to fit into 1024 bytes, the signal generator allocates additional memory in multiples of 1024 bytes. For example, the signal generator allocates 3072 bytes of memory for a waveform with 500 samples (2500 bytes).

$$3 \times 1024 \text{ bytes} = 3072 \text{ bytes of memory}$$

As shown in the examples, waveforms can cause the signal generator to allocate more memory than what is actually used, which decreases the amount of available memory.

### Non-Volatile Memory

The signal generator allocates non-volatile memory in blocks of 512 bytes. For files less than or equal to 512 bytes, the file uses only one block of memory. For files larger than 512 bytes, the signal generator allocates additional memory in multiples of 512 byte blocks. For example, a file that has 21,538 bytes consumes 43 memory blocks (22,016 bytes).

Memory Size

The amount of available memory, volatile and non-volatile, varies by option and the size of the other files that share the memory. When we refer to waveform files, we state the memory size in samples (one sample equals five bytes). The baseband generator (BBG) options (601 and 602) contain the waveform playback memory. The following tables show the maximum available memory.

Volatile (WFM1) Memory		Non-Volatile (NVWFM) Memory	
Option	Size	Option	Size
601 (BBG)	8 MSa (40 MB)	Standard	3 MSa (15 MB)
602 (BBG)	64 MSa (320 MB)	005 (Hard disk)	1 GSa (5 GB)

Commands for Downloading and Extracting Waveform Data

You can download I/Q data and the associated file header and marker file information (collectively called waveform data) into volatile or non-volatile memory. For information on waveform structure, see [“Waveform Structure” on page 139](#).

**NOTE** Before downloading files into volatile memory (WFM1), turn off the ARB.  
 Press: **Mode > Dual Arb > ARB Off On** until Off highlights  
 Or send: [:SOURce]:RADio:ARB[:STATe] OFF

The signal generator provides the option of downloading waveform data either for extraction or not for extraction. When you extract waveform data, the signal generator encrypts the data. The SCPI download commands determine whether the waveform data is extractable.

If you use SCPI commands to download waveform data to be extracted later, you must use the MEM:DATA:UNPRotected command. If you use FTP commands, no special command syntax is necessary.

You can download or extract waveform data created in any of the following ways:

- with signal simulation software, such as MATLAB or Agilent Advanced Design System (ADS)
- with advanced programming languages, such as C++, VB or VEE
- with Agilent Signal Studio software
- with the signal generator

Waveform Data Encryption

You can download encrypted waveform data extracted from one signal generator into another signal generator with the same option or software license for the modulation format. You can also extract encrypted waveform data created with software such as MATLAB or ADS, providing the data was downloaded to the signal generator using the proper command.

When you generate a waveform from the signal generator’s internal ARB modulation format or download a waveform from an Agilent Signal Studio software product, the resulting waveform data is automatically stored in volatile memory and is available for extraction as an encrypted file.

The exception to encrypted file extraction is user-created I/Q data. You can extract this I/Q data unencrypted.

## Encrypted I/Q Files and the Securewave Directory

The signal generator uses the `securewave` directory to perform file encryption (extraction) and decryption (downloads). The `securewave` directory is not an actual storage directory, but rather a portal for the encryption and decryption process. While the `securewave` directory contains file names, these are actually pointers to the true files located in signal generator memory (volatile or non-volatile). When you download an encrypted file, the `securewave` directory decrypts the file and unpackages the contents into its file header, I/Q data, and marker data. When you extract a file, the `securewave` directory packages the file header, I/Q data, and marker data and encrypts the waveform data file.

The signal generator uses the following `securewave` directory paths for file extractions and encrypted file downloads:

Volatile                    `/user/securewave/file_name` or `swfm:file_name`

Non-volatile              `/user/bbg1/securewave/file_name` or `snvwfm1:file_name`

---

**NOTE** To extract files (other than user-created I/Q files) and to download encrypted files, you *must* use the `securewave` directory. If you attempt to extract previously downloaded encrypted files (including Signal Studio downloaded files or internally created signal generator files) *without* using the `securewave` directory, the signal generator generates an error and displays `ERROR: 221, Access Denied`.

---

## File Transfer Methods

- SCPI using VXI-11 (VMEbus Extensions for Instrumentation as defined in VXI-11)
- SCPI over the GPIB or RS 232
- SCPI with sockets LAN (using port 5025)
- File Transfer Protocol (FTP)

## SCPI Command Line Structure

The signal generator expects to see waveform data as block data (binary files). The IEEE standard 488.2-1992 section 7.7.6 defines block data. The following example shows how to structure a SCPI command for downloading waveform data (#ABC represents the block data):

```
:MMEM:DATA "<file_name>",#ABC
```

"<file\_name>"    the I/Q file name and file path within the signal generator

#                    indicates the start of the data block

A                    the number of decimal digits present in B

B                    a decimal number specifying the number of data bytes to follow in C

C                    the actual binary waveform data

The following example demonstrates this structure:

```
MMEM:DATA "WFM1:my_file",#3|240|12%S!4&07#8g*Y9@7...|
```

file\_name

A

B

C

WFM1:	the file path
my_file	the I/Q file name as it will appear in the signal generator's memory catalog
#	indicates the start of the data block
3	B has three decimal digits
240	240 bytes of data to follow in C
12%S!4&07#8g*Y9@7...	the ASCII representation of some of the binary data downloaded to the signal generator, however not all ASCII values are printable

---

**NOTE** If you use SCPI with sockets to send data to the signal generator, you must provide an end-of-file indicator, as shown in the following command:  
MMEM:DATA "WFM1:<file\_name>",<blockdata>NL^END

---

Commands and File Paths for Downloading and Extracting Waveform Data

You can download or extract waveform data using the commands and file paths in the following tables:

- [Table 4-1, “Downloading Unencrypted Files for No Extraction,” on page 148](#)
- [Table 4-2, “Downloading Encrypted Files for No Extraction,” on page 149](#)
- [Table 4-3, “Downloading Unencrypted Files for Extraction,” on page 149](#)
- [Table 4-4, “Downloading Encrypted Files for Extraction,” on page 150](#)
- [Table 4-5, “Extracting Encrypted Waveform Data,” on page 150](#)

Table 4-1 Downloading Unencrypted Files for No Extraction

Download Method/ Memory Type	Command Syntax Options
SCPI/volatile memory	MMEM:DATA "WFM1:<file_name>",<blockdata> MMEM:DATA "MKR1:<file_name>",<blockdata> MMEM:DATA "HDR1:<file_name>",<blockdata>
SCPI/volatile memory with full directory path	MMEM:DATA "user/bbgl/waveform/<file_name>",<blockdata> MMEM:DATA "user/bbgl/markers/<file_name>",<blockdata> MMEM:DATA "user/bbgl/header/<file_name>",<blockdata>
SCPI/non-volatile memory	MMEM:DATA "NVWFM:<file_name>",<blockdata> MMEM:DATA "NVMKR:<file_name>",<blockdata> MMEM:DATA "NVHDR:<file_name>",<blockdata>
SCPI/non-volatile memory with full directory path	MMEM:DATA /user/waveform/<file_name>",<blockdata> MMEM:DATA /user/markers/<file_name>",<blockdata> MMEM:DATA /user/header/<file_name>",<blockdata>

**Table 4-2 Downloading Encrypted Files for No Extraction**

Download Method /Memory Type	Command Syntax Options
SCPI/volatile memory	MMEM:DATA "user/bbgl/securewave/<file_name>",<blockdata> MMEM:DATA "SWFM1:<file_name>",<blockdata> MMEM:DATA "file_name@SWFM1",<blockdata>
SCPI/non-volatile memory	MMEM:DATA "user/securewave/<file_name>",<blockdata> MMEM:DATA "SNVWFM:<file_name>",<blockdata> MMEM:DATA "file_name@SNVWFM",<blockdata>

**Table 4-3 Downloading Unencrypted Files for Extraction**

Download Method/ Memory Type	Command Syntax Options
SCPI/volatile memory	MEM:DATA:UNProtected "/user/bbgl/waveform/file_name",<blockdata> MEM:DATA:UNProtected "/user/bbgl/markers/file_name",<blockdata> MEM:DATA:UNProtected "/user/bbgl/header/file_name",<blockdata> MEM:DATA:UNProtected "WF1:file_name",<blockdata> MEM:DATA:UNProtected "MKR1:file_name",<blockdata> MEM:DATA:UNProtected "HDR1:file_name",<blockdata> MEM:DATA:UNProtected "file_name@WF1",<blockdata> MEM:DATA:UNProtected "file_name@MKR1",<blockdata> MEM:DATA:UNProtected "file_name@HDR1",<blockdata>
SCPI/non-volatile memory	MEM:DATA:UNProtected "/user/waveform/file_name",<blockdata> MEM:DATA:UNProtected "/user/markers/file_name",<blockdata> MEM:DATA:UNProtected "/user/header/file_name",<blockdata> MEM:DATA:UNProtected "NVWFM:file_name",<blockdata> MEM:DATA:UNProtected "NVMKR:file_name",<blockdata> MEM:DATA:UNProtected "NVHDR:file_name",<blockdata> MEM:DATA:UNProtected "file_name@NVWFM",<blockdata> MEM:DATA:UNProtected "file_name@NVMKR",<blockdata> MEM:DATA:UNProtected "file_name@NVHDR",<blockdata>
FTP/volatile memory <sup>a</sup>	put <file_name> /user/bbgl/waveform/<file_name> put <file_name> /user/bbgl/markers/<file_name>
FTP/non-volatile memory <sup>a</sup>	put <file_name> /user/waveform/<file_name> put <file_name> /user/markers/<file_name>

a. See "FTP Procedures" on page 150.

**Table 4-4 Downloading Encrypted Files for Extraction**

Download Method/Memory Type	Command Syntax Options
SCPI/volatile memory	MEM:DATA:UNPRotected "/user/bbgl/securewave/file_name",<blockdata> MEM:DATA:UNPRotected "SWFM1:file_name",<blockdata> MEM:DATA:UNPRotected "file_name@SWFM1",<blockdata>
SCPI/non-volatile memory	MEM:DATA:UNPRotected "/user/securewave/file_name",<blockdata> MEM:DATA:UNPRotected "SNVWFM:file_name",<blockdata> MEM:DATA:UNPRotected "file_name@SNVWFM",<blockdata>
FTP/volatile memory <sup>a</sup>	put <file_name> /user/bbgl/securewave/<file_name>
FTP/non-volatile memory <sup>a</sup>	put <file_name> /user/securewave/<file_name>

a. See "FTP Procedures" on page 150.

**Table 4-5 Extracting Encrypted Waveform Data**

Download Method/Memory Type	Command Syntax Options
SCPI/volatile memory	MMEM:DATA? "/user/bbgl/securewave/file_name" MMEM:DATA? "SWFM1:file_name" MMEM:DATA? "file_name@SWFM1"
SCPI/non-volatile memory	MMEM:DATA? "/user/securewave/file_name" MMEM:DATA? "SNVWFM:file_name" MMEM:DATA? "file_name@SNVWFM"
FTP/volatile memory <sup>a</sup>	get /user/bbgl/securewave/<file_name>
FTP/non-volatile memory <sup>a</sup>	get /user/securewave/<file_name>

a. See [FTP Procedures](#).

## FTP Procedures

There are three ways to FTP files:

- use Microsoft's<sup>®</sup> Internet Explorer FTP feature<sup>1</sup>
- use the signal generator's internal web server
- use the PC's or UNIX command window

### Using Microsoft's Internet Explorer

1. Enter the signal generator's hostname or IP address as part of the FTP URL.

*ftp://<host name> or <IP address>*

2. Press **Enter** on the keyboard or **Go** from the Internet Explorer window.

The signal generator files appear in the Internet Explorer window.

3. Drag and drop files between the PC and the Internet Explorer window

### Using the Signal Generator's Internal Web Server

1. Enter the signal generator's hostname or IP address in the URL.

*http://<host name> or <IP address>*

2. Click the **Signal Generator FTP Access** button located on the left side of the window.

The signal generator files appear in the web browser's window.

3. Drag and drop files between the PC and the browser's window

For more information on the web server feature, see ["Communicating with the Signal Generator Using a Web Browser"](#) on page 27.

### Using the Command Window (PC or UNIX)

This procedure downloads to non-volatile memory. To download to volatile memory, change the file path.

1. From the PC command prompt or UNIX command line, change to the destination directory for the file you intend to download.
2. From the PC command prompt or UNIX command line, type `ftp <instrument name>`. Where `instrument name` is the signal generator's hostname or IP address.
3. At the `User:` prompt in the ftp window, press **Enter** (no entry is required).
4. At the `Password:` prompt in the ftp window, press **Enter** (no entry is required).
5. At the ftp prompt, type:  
`put <file_name> /user/waveform/<file_name1>`

where `<file_name>` is the name of the file to download and `<file_name1>` is the name designator for the signal generator's `/user/waveform/` directory.

- If a marker file is associated with the data file, use the following command to download it to the signal generator:

`put <marker file_name> /user/markers/<file_name1>`

where `<marker file_name>` is the name of the file to download and `<file_name1>` is the name designator for the file in the signal generator's `/user/markers/` directory. Marker files and the associated I/Q waveform data have the same name.

- 
1. Microsoft is a U.S registered trademark of Microsoft Corporation.

---

**NOTE** If no marker file is provided, the signal generator automatically creates a default marker file consisting of all zeros.

---

6. At the `ftp` prompt, type: `bye`
7. At the command prompt, type: `exit`

## Creating Waveform Data

This section examines the C++ code algorithm for creating I/Q waveform data by breaking the programming example into functional parts and explaining the code in generic terms. This is done to help you understand the code algorithm in creating the I and Q data, so you can leverage the concept into your programming environment. If you do not need this level of detail, you can find the complete programming example in [“Programming Examples” on page 169](#).

You can use various programming environments to create ARB waveform data. Generally there are two types:

- **Simulation software**— this includes MATLAB, Agilent Technologies EESof Advanced Design System (ADS), Signal Processing WorkSystem (SPW), and so forth.
- **Advanced programming languages**—this includes, C++, VB, VEE, MS Visual Studio.Net, Labview, and so forth.

No matter which programming environment you use to create the waveform data, make sure that the data conforms to the data requirements shown on [page 132](#). To learn about I/Q data for the signal generator, see [“Understanding Waveform Data” on page 132](#).

## Code Algorithm

This section uses code from the C++ programming example [“Importing, Byte Swapping, Interleaving, and Downloading I and Q Data—Big and Little Endian Order” on page 187](#) to demonstrate how to create and scale waveform data.

There are three steps in the process of creating an I/Q waveform:

1. Create the I and Q data.
2. Save the I and Q data to a text file for review.
3. Interleave the I and Q data to make an I/Q file, and swap the byte order for little-endian platforms.

For information on downloading I/Q waveform data to a signal generator, refer to [“Commands and File Paths for Downloading and Extracting Waveform Data” on page 148](#) and [“Downloading Waveform Data” on page 157](#).

### 1. Create I and Q data.

The following lines of code create scaled I and Q data for a sine wave. The I data consists of one period of a sine wave and the Q data consists of one period of a cosine wave.



**Line            Code—Create I and Q data**

```

1      const int NUMSAMPLES=500;
2      main(int argc, char* argv[]);
3      {
4          short idata[NUMSAMPLES];
5          short qdata[NUMSAMPLES];
6          int numsamples = NUMSAMPLES;
7          for(int index=0; index<numsamples; index++);
8          {
9              idata[index]=23000 * sin((2*3.14*index)/numsamples);
10             qdata[index]=23000 * cos((2*3.14*index)/numsamples);
11         }

```

Line	Code Description—Create I and Q data
1	Define the number of waveform points. Note that the maximum number of waveform points that you can set is based on the amount of available memory in the signal generator. For more information on signal generator memory, refer to <a href="#">“Waveform Memory” on page 144</a> .
2	Define the main function in C++.
4	Create an array to hold the generated I values. The array length equals the number of the waveform points. Note that we define the array as type <i>short</i> , which represents a 16-bit signed integer in most C++ compilers.
5	Create an array to hold the generated Q values (signed 16-bit integers).
6	Define and set a temporary variable, which is used to calculate the I and Q values.

Line	Code Description—Create I and Q data
7–11	<div>Create a loop to do the following:</div> <div><ul style="list-style-type: none"><li>• Generate and scale the I data (DAC values). This example uses a simple sine equation, where <math>2\pi \cdot 3.14</math> equals one waveform cycle. Change the equation to fit your application.<ul style="list-style-type: none"><li>— The array pointer, <i>index</i>, increments from 0–499, creating 500 I data points over one period of the sine waveform.</li><li>— Set the scale of the DAC values in the range of –32767 to 32768, where the values –32767 and 32768 equal full scale negative and positive respectively. This example uses 23000 as the multiplier, resulting in approximately 70% scaling. For more information on scaling, see <a href="#">“Scaling DAC Values” on page 136</a>.</li></ul></li></ul></div> <div><div>NOTE</div><div>The signal generator comes from the factory with I/Q scaling set to 70%. If you reduce the DAC input values, ensure that you set the signal generator scaling (:RADio:ARB:RSCaling) to an appropriate setting that accounts for the reduced values.</div></div> <div><ul style="list-style-type: none"><li>• Generate and scale the Q data (DAC value). This example uses a simple cosine equation, where <math>2\pi \cdot 3.14</math> equals one waveform cycle. Change the equation to fit your application.<ul style="list-style-type: none"><li>— The array pointer, <i>index</i>, increments from 0–499, creating 500 Q data points over one period of the cosine waveform.</li><li>— Set the scale of the DAC values in the range of –32767 to 32768, where the values –32767 and 32768 equal full scale negative and positive respectively. This example uses 23000 as the multiplier, resulting in approximately 70% scaling. For more information on scaling, see <a href="#">“Scaling DAC Values” on page 136</a>.</li></ul></li></ul></div>

2. Save the I/Q data to a text file to review.

The following lines of code export the I and Q data to a text file for validation. After exporting the data, open the file using Microsoft Excel or a similar spreadsheet program, and verify that the I and Q data are correct.

Line Code Description—Saving the I/Q Data to a Text File

```
12 char *ofile = "c:\\temp\\iq.txt";
13 FILE *outfile = fopen(ofile, "w");
14 if (outfile==NULL) perror ("Error opening file to write");
15 for(index=0; index<numsamples; index++)
16 {
17     fprintf(outfile, "%d, %d\n", idata[index], qdata[index]);
18 }
19 fclose(outfile);
```

Line	Code Description—Saving the I/Q Data to a Text File
12	Set the absolute path of a text file to a character variable. In this example, <i>iq.txt</i> is the file name and <i>*ofile</i> is the variable name.  For the file path, some operating systems may not use the drive prefix ('c:' in this example), or may require only a single forward slash (/), or both (" <i>/temp/iq.txt</i> ")
13	Open the text file in <i>write</i> format.
14	If the text file does not open, print an error message.
15–18	Create a loop that prints the array of generated I and Q data samples to the text file.
19	Close the text file.

### 3. Interleave the I and Q data, and byte swap if using little endian order.

This step has two sets of code:

- Interleaving and byte swapping I and Q data for little endian order
- Interleaving I and Q data for big endian order

For more information on byte order, see [“Little Endian and Big Endian \(Byte Order\)”](#) on page 134.

Line	Code—Interleaving and Byte Swapping for Little Endian Order
20	<code>char iqbuffer[NUMSAMPLES*4];</code>
21	<code>for(index=0; index&lt;numsamples; index++)</code>
22	<code>{</code>
23	<code>short ivalue = idata[index];</code>
24	<code>short qvalue = qdata[index];</code>
25	<code>iqbuffer[index*4] = (ivalue &gt;&gt; 8) &amp; 0xFF;</code>
26	<code>iqbuffer[index*4+1] = ivalue &amp; 0xFF;</code>
27	<code>iqbuffer[index*4+2] = (qvalue &gt;&gt; 8) &amp; 0xFF;</code>
28	<code>iqbuffer[index*4+3] = qvalue &amp; 0xFF;</code>
29	<code>}</code>
30	<code>return 0;</code>

Line	Code Description—Interleaving and Byte Swapping for Little Endian Order
20	Define a character array to store the interleaved I and Q data. The character array makes byte swapping easier, since each array location accepts only 8 bits (1 byte). The array size increases by four times to accommodate two bytes of I data and two bytes of Q data.







Line	Code Description–Interleaving and Byte Swapping for Little Endian Order
21–29	<div> <div>Create a loop to do the following:</div> <ul style="list-style-type: none"> <li>Save the current I data array value to a variable.</li> </ul> <div> <div>NOTE</div> <div>In rare instances, a compiler may define <i>short</i> as larger than 16 bits. If this condition exists, replace <i>short</i> with the appropriate object or label that defines a 16-bit integer.</div> </div> <ul style="list-style-type: none"> <li>Save the current Q data array value to a variable.</li> <li>Swap the low bytes (bits 0–7) of the data with the high bytes of the data (done for both</li> </ul> </div>
21–29	<div> <div>the I and Q data), and interleave the I and Q data.</div> <div>– shift the data pointer right 8 bits to the beginning of the high byte (<i>ivalue</i> &gt;&gt; 8)</div> <div> <div>Little Endian Order</div> <div> <div> <div>7 6 5 4 3 2 1 0 15 14 13 12 11 10 9 8</div> <div>Bit Position</div> </div> <div> <div>1 1 1 0 1 0 0 1 1 0 1 1 0 1 1 1</div> <div>Data</div> </div> <div> <div> <div>↑</div> <div>Data pointer</div> </div> <div> <div>→</div> <div>Data pointer shifted 8 bits</div> </div> <div> <div>Hex values = E9 B7</div> </div> </div> </div> <div> <div>– AND (boolean) the high I byte with 0xFF to make the high I byte the value to store in the IQ array–(<i>ivalue</i> &gt;&gt; 8) &amp; 0xFF</div> <div> <div> <div>15 14 13 12 11 10 9 8</div> <div>1 0 1 1 0 1 1 1</div> <div>Hex value =B7</div> </div> <div> <div> <div>1 1 1 1 1 1 1 1</div> <div>Hex value =FF</div> </div> <div> <div> <div>1 0 1 1 0 1 1 1</div> <div>Hex value =B7</div> </div> </div> </div> <div> <div>– AND (boolean) the low I byte with 0xFF (<i>ivalue</i> &amp; 0xFF) to make the low I byte the value to store in the I/Q array location just after the high byte [<i>index</i> * 4 + 1]</div> <div> <div>I Data in I/Q Array after Byte Swap (Big Endian Order)</div> <div> <div>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</div> <div>Bit Position</div> </div> <div> <div>1 0 1 1 0 1 1 1 1 1 1 0 1 0 0 1</div> <div>Data</div> </div> <div> <div>Hex value = B7 E9</div> </div> </div> <div> <div>– Swap the Q byte order within the same loop. Notice that the I and Q data interleave with each loop cycle. This is due to the I/Q array shifting by one location for each I and Q operation [<i>index</i> * 4 + n].</div> <div> <div>Interleaved I/Q Array in Big Endian Order</div> <div> <div>15..... 8 7..... 0 15..... 8 7..... 0</div> <div>Bit Position</div> </div> <div> <div>1 0 1 1 0 1 1 1 1 1 0 1 0 0 1 1 1 1 0 0 1 0 1 0 1 1 0 1 1 1</div> <div>Data</div> </div> <div> <div>I Data</div> <div>Q Data</div> </div> </div> </div> </div></div></div></div></div>

**Line                      Code—Interleaving I and Q data for Big Endian Order**

```

20     short iqbuffer[NUMSAMPLES*2];
21     for(index=0; index<numsamples; index++)
22     {
23         iqbuffer[index*2] = idata[index];
24         iqbuffer[index*2+1] = qdata[index];
25     }
26     return 0;

```

Line	Code Description—Interleaving I and Q data for Big Endian Order																												
20	<p>Define a 16-bit integer (short) array to store the interleaved I and Q data. The array size increases by two times to accommodate two bytes of I data and two bytes of Q data.</p> <hr/> <p><b>NOTE</b> In rare instances, a compiler may define <i>short</i> as larger than 16 bits. If this condition exists, replace <i>short</i> with the appropriate object or label that defines a 16-bit integer.</p> <hr/>																												
21–25	<p>Create a loop to do the following:</p> <ul style="list-style-type: none"><li>• Store the I data values to the I/Q array location [<i>index</i>*2].</li><li>• Store the Q data values to the I/Q array location [<i>index</i>*2+1].</li></ul> <p style="text-align: center;"><b>Interleaved I/Q Array in Big Endian Order</b></p> <table style="margin-left: auto; margin-right: auto;"><tr><td style="text-align: right;">15.....</td><td style="text-align: right;">8 7.....</td><td style="text-align: right;">0</td><td style="text-align: right;">15.....</td><td style="text-align: right;">8 7.....</td><td style="text-align: right;">0</td><td style="text-align: right;">Bit Position</td></tr><tr><td style="text-align: right;">1 0 1 1 0 1 1 1 1 1 0 1 0 0 1</td><td style="text-align: right;">1 1 1 0 0 1 0 1 0 1 1 0 1 0 1 1</td><td></td><td></td><td></td><td></td><td style="text-align: right;">Data</td></tr><tr><td colspan="3" style="text-align: center;"></td><td colspan="3" style="text-align: center;"></td><td></td></tr><tr><td colspan="3" style="text-align: center;">I Data</td><td colspan="3" style="text-align: center;">Q Data</td><td></td></tr></table>	15.....	8 7.....	0	15.....	8 7.....	0	Bit Position	1 0 1 1 0 1 1 1 1 1 0 1 0 0 1	1 1 1 0 0 1 0 1 0 1 1 0 1 0 1 1					Data								I Data			Q Data			
15.....	8 7.....	0	15.....	8 7.....	0	Bit Position																							
1 0 1 1 0 1 1 1 1 1 0 1 0 0 1	1 1 1 0 0 1 0 1 0 1 1 0 1 0 1 1					Data																							
																													
I Data			Q Data																										

To download the data created in the above example, see [“Using Advanced Programming Languages” on page 160](#).

## Downloading Waveform Data

This section examines methods of downloading I/Q waveform data created in MATLAB (a simulation software) and C++ (an advanced programming language). For more information on simulation and advanced programming environments, see [“Creating Waveform Data” on page 152](#).

To download data from simulation software environments, it is typically easier to use one of the free download utilities (described on [page 166](#)), because simulation software usually saves the data to a file. In MATLAB however, you can either save data to a .mat file or create a complex array. To facilitate downloading a MATLAB complex data array, Agilent created the PSG/ESG Download Assistant (one of the free download utilities), which downloads the complex data array from within the MATLAB environment. This section shows how to use the download assistant.

For advanced programming languages, this section closely examines the code algorithm for downloading I/Q waveform data by breaking the programming examples into functional parts and explaining the code in generic terms. This is done to help you understand the code algorithm in

downloading the interleaved I/Q data, so you can leverage the concept into your programming environment. While not discussed in this section, you may also save the data to a binary file and use one of the download utilities to download the waveform data (see [“Using the Download Utilities” on page 166](#)).

If you do not need the level of detail this section provides, you can find complete programming examples in [“Programming Examples” on page 169](#). Prior to downloading the I/Q data, ensure that it conforms to the data requirements shown on [page 132](#). To learn about I/Q data for the signal generator, see [“Understanding Waveform Data” on page 132](#). For creating waveform data, see [“Creating Waveform Data” on page 152](#).

---

**NOTE** Before downloading files into volatile memory (WFM1), turn off the ARB.  
Press: **Mode > Dual Arb > ARB Off On** until Off highlights  
Or send: [:SOURce]:RADio:ARB[:STATE] OFF

---

## Using Simulation Software

This procedure uses a complex data array created in MATLAB and uses the PSG/ESG Download Assistant to download the data. To obtain the PSG/ESG Download Assistant, see [“Using the Download Utilities” on page 166](#).

There are two steps in the process of downloading an I/Q waveform:

1. Open a connection session.
2. Download the I/Q data.

### 1. Open a connection session with the signal generator.

The following code establishes a LAN connection with the signal generator, sends the IEEE SCPI command `*idn?`, and if the connection fails, displays an error message.

Line	Code—Open a Connection Session
1	<code>io = agt_newconnection('tcpip','IP address');</code>
	<code>%io = agt_newconnection('gpib',&lt;primary address&gt;,&lt;secondary address&gt;);</code>
2	<code>[status,status_description,query_result] = agt_query(io,'*idn?');</code>
3	<code>if status == -1</code>
4	<code>display 'fail to connect to the signal generator';</code>
5	<code>end;</code>

Line	Code Description—Open a Connection Session with the Signal Generator
1	<p>Sets up a structure (indicated above by <i>io</i>) used by subsequent function calls to establish a LAN connection to the signal generator.</p> <ul style="list-style-type: none"> <li>• <i>agt_newconnection()</i> is the function of Agilent Download Assistant used in MATLAB to build a connection to the signal generator.</li> <li>• If you are using GPIB to connect to the signal generator, provide the board, primary address, and secondary address: <i>io = agt_newconnection('gpiB',0,19);</i> Change the GPIB address based on your instrument setting.</li> </ul>
2	<p>Send a query to the signal generator to verify the connection.</p> <ul style="list-style-type: none"> <li>• <i>agt_query()</i> is an Agilent Download Assistant function that sends a query to the signal generator.</li> <li>• If signal generator receives the query *idn?, <i>status</i> returns a zero and <i>query_result</i> returns the signal generator's model number, serial number, and firmware version.</li> </ul>
3–5	If the query fails, display a message.

## 2. Download the I/Q data

The following code downloads the generated waveform data to the signal generator, and if the download fails, displays a message.

Line	Code—Download the I/Q data
6	<code>[status, status_description] = agt_waveformload(io, IQwave, 'waveformfile1', 2000, 'no_play', 'norm_scale');</code>
7	<code>if status == -1</code>
8	<code>display 'fail to download to the signal generator';</code>
9	<code>end;</code>

Line	Code Description—Download the I/Q data
6	<p>Download the I/Q waveform data to the signal generator by using the function call (<i>agt_waveformload</i> ) from the Agilent Download Assistant. Some of the arguments are optional as indicated below, but if one is used, you must use all arguments previous to the one you require.</p> <p>Notice that with this function, you can perform the following actions:</p> <ul style="list-style-type: none"><li>• download complex I/Q data</li><li>• name the file (optional argument)</li><li>• set the sample rate (optional argument)</li></ul> <p>If you do not set a value, the signal generator uses its preset value of 100 MHz, or if a waveform was previously play, the value from that waveform.</p> <ul style="list-style-type: none"><li>• start or not start waveform playback after downloading the data (optional argument)</li></ul> <p>Use either the argument <i>play</i> or the argument <i>no_play</i>.</p> <ul style="list-style-type: none"><li>• whether to normalize and scale the I/Q data (optional argument)</li></ul> <p>If you normalize and scale the data within the body of the code, then use <i>no_normscale</i>, but if you need to normalize and scale the data, use <i>norm_scale</i>. This normalizes the waveform data to the DAC values and then scales the data to 70% of the DAC values.</p> <ul style="list-style-type: none"><li>• download marker data (optional argument)</li></ul> <p>If there is no marker data, the signal generator creates a default marker file, all marker set to zero.</p> <p>To verify the waveform data download, see <a href="#">“Loading, Playing, and Verifying a Downloaded Waveform” on page 164</a>.</p>
7–9	If the download fails, display an error message.

### Using Advanced Programming Languages

This procedure uses code from the C++ programming example [“Importing, Byte Swapping, Interleaving, and Downloading I and Q Data—Big and Little Endian Order” on page 187](#).  
For information on creating I/Q waveform data, refer to [“Creating Waveform Data” on page 152](#).

There are two steps in the process of downloading an I/Q waveform:

1. Open a connection session.
2. Download the I/Q data.

**1. Open a connection session with the signal generator.**

The following code establishes a LAN connection with the signal generator or prints an error message if the session is not opened successfully.



**Line                      Code Description—Open a Connection Session**

```

1  char* instOpenString ="lan[hostname or IP address]";
   //char* instOpenString ="gpib<primary addr>,<secondary addr>";
2  INST id=iopen(instOpenString);
3  if (!id)
4  {
5  fprintf(stderr, "iopen failed (%s)\n", instOpenString);
6  return -1;
7  }

```

Line	Code Description—Open a Connection Session
1	<p>Assign the signal generator's LAN hostname, IP address, or GPIB address to a character string.</p> <ul style="list-style-type: none"> <li>This example uses the Agilent IO library's <i>iopen()</i> SICL function to establish a LAN connection with the signal generator. The input argument, <i>lan[hostname or IP address]</i> contains the device, interface, or commander address. Change it to your signal generator host name or just set it to the IP address used by your signal generator. For example: <i>"lan[999.137.240.9]"</i></li> <li>If you are using GPIB to connect to the signal generator, use the commented line in place of the first line. Insert the GPIB address based on your instrument setting, for example <i>"gpib0,19"</i>.</li> <li>For the detailed information about the parameters of the SICL function <i>iopen()</i>, refer to the online <i>"Agilent SICL User's Guide for Windows."</i></li> </ul>
2	<p>Open a connection session with the signal generator to download the generated I/Q data.</p> <p>The SICL function <i>iopen()</i> is from the Agilent IO library and creates a session that returns an identifier to <i>id</i>.</p> <ul style="list-style-type: none"> <li>If <i>iopen()</i> succeeds in establishing a connection, the function returns a valid session <i>id</i>. The valid session <i>id</i> is not viewable, and can only be used by other SICL functions.</li> <li>If <i>iopen()</i> generates an error before making the connection, the session identifier is set to zero. This occurs if the connection fails.</li> <li>To use this function in C++, you must include the standard header <code>#include &lt;sicl.h&gt;</code> before the <code>main()</code> function.</li> </ul>
3–7	<p>If <i>id</i> = 0, the program prints out the error message and exits the program.</p>

2. Download the I/Q data.

The following code sends the SCPI command and downloads the generated waveform data to the signal generator.

Line	CodeDescription—Download the I/Q Data
8	int bytesToSend;
9	bytesToSend = numsamples*4;
10	char s[20];
11	char cmd[200];
12	sprintf(s, "%d", bytesToSend);
13	sprintf(cmd, ":MEM:DATA \\WFM1:FILE1\\", #d%d", strlen(s), bytesToSend); iwrite(id, cmd, strlen(cmd), 0, 0);
14	iwrite(id, iqbuffer, bytesToSend, 0, 0);
15	iwrite(id, "\\n", 1, 1, 0);
16	

Line	Code Description—Download the I/Q data
8	Define an integer variable ( <i>bytesToSend</i> ) to store the number of bytes to send to the signal generator.
9	Calculate the total number of bytes, and store the value in the integer variable defined in line 8.  In this code, <i>numsamples</i> contains the number of waveform points, not the number of bytes. Because it takes four bytes of data, two I bytes and two Q bytes, to create one waveform point, we have to multiply <i>numsamples</i> by four. This is shown in the following example:  <div> <div>numsamples = 500 waveform points</div> <div>numsamples × 4 = 2000 (four bytes per point)</div> <div>bytesToSend = 2000 (numsamples × 4)</div> </div> For information on setting the number of waveform points, see <a href="#">“1. Create I and Q data.” on page 152</a> .
10	Create a string large enough to hold the <i>bytesToSend</i> value as characters. In this code, string <i>s</i> is set to 20 bytes (20 characters—one character equals one byte)
11	Create a string and set its length ( <i>cmd</i> [200] ) to hold the SCPI command syntax and parameters. In this code, we define the string length as 200 bytes (200 characters).
12	Store the value of <i>bytesToSend</i> in string <i>s</i> . For example, if bytesToSend = 2000; <i>s</i> = "2000"
13	Store the SCPI command syntax and parameters in the string <i>cmd</i> . The SCPI command prepares the signal generator to accept the data. <ul style="list-style-type: none"> <li><i>sprintf()</i> is a standard function in C++, which writes string data to a string variable.</li> <li><i>strlen()</i> is a standard function in C++, which returns length of a string.</li> <li>If <i>bytesToSend</i> = 2000, then <i>s</i> = "2000", <i>strlen(s)</i> = 4, so <i>cmd</i> = :MEM:DATA "WFM1:FILE1\" #42000.</li> </ul>

Line	Code Description—Download the I/Q data
14	<p>Send the SCPI command stored in the string <i>cmd</i> to the signal generator, which is represented by the session <i>id</i>.</p> <ul style="list-style-type: none"> <li>• <i>iwrite()</i> is a SICL function in Agilent IO library, which writes the data (block data) specified in the string <i>cmd</i> to the signal generator (<i>id</i>).</li> <li>• The third argument of <i>iwrite()</i>, <i>strlen(cmd)</i>, informs the signal generator of the number of bytes in the command string. The signal generator parses the string to determine the number of I/Q data bytes it expects to receive.</li> <li>• The fourth argument of <i>iwrite()</i>, zero, means there is no END indicator for the string. This lets the session remain open, so the program can download the I/Q data.</li> </ul>
15	<p>Send the generated waveform data stored in the I/Q array (<i>iqbuffer</i>) to the signal generator.</p> <ul style="list-style-type: none"> <li>• <i>iwrite()</i> sends the data specified in <i>iqbuffer</i> to the signal generator (session identifier specified in <i>id</i>).</li> <li>• The third argument of <i>iwrite()</i>, <i>bytesToSend</i>, contains the length of the <i>iqbuffer</i> in bytes. In this example, it is 2000.</li> <li>• The fourth argument of <i>iwrite()</i>, 0, means there is no END indicator in the data.</li> </ul> <p>In many programming languages, there are two methods to send SCPI commands and data:</p> <ul style="list-style-type: none"> <li>— Method 1 where the program stops the data download when it encounters the first zero (END indicator) in the data.</li> <li>— Method 2 where the program sends a fixed number of bytes and ignores any zeros in the data. This is the method used in our program.</li> </ul> <p>For your programming language, you must find and use the equivalent of method two. Otherwise you may only achieve a partial download of the I and Q data.</p>
16	<p>Send the terminating carriage (\n) as the last byte of the waveform data.</p> <ul style="list-style-type: none"> <li>• <i>iwrite()</i> writes the data “\n” to the signal generator (session identifier specified in <i>id</i>).</li> <li>• The third argument of <i>iwrite()</i>, 1, sends one byte to the signal generator.</li> <li>• The fourth argument of <i>iwrite()</i>, 1, is the END indicator, which the program uses to terminate the data download.</li> </ul> <p>To verify the waveform data download, see <a href="#">“Loading, Playing, and Verifying a Downloaded Waveform” on page 164</a>.</p>

## Loading, Playing, and Verifying a Downloaded Waveform

The following procedures show how to perform the steps using either front-panel key presses or SCPI commands.

### Loading a File from Non-Volatile Memory

Select the downloaded I/Q file in non-volatile waveform memory (NVWFM) and load it into volatile waveform memory (WFM1). The file comprises three items: I/Q data, marker file, and file header information. Loading the I/Q file also loads the marker file and file header.

- From the front panel:
  1. Press **Mode** > **Dual ARB** > **Select Waveform** > **Waveform Segments** > **Load Store** until Load highlights.
  2. Highlight the I/Q file in the NVWFM catalog.
  3. Press **Load Segment From NVWFM Memory**.
  4. Press **Return**.
- Remotely send one of the following SCPI command to copy the I/Q file, marker file and file header information:

```
:MEMory:COpy[NAME]"<NVWFM:file_name>","<WFM1:file_name>"  
:MEMory:COpy[NAME]"<NVMKR:file_name>","<MKR1:file_name>"
```

---

<b>NOTE</b>	When you copy a waveform file or marker file information from volatile or non-volatile memory, the waveform and associated marker and header files are all copied. Conversely, when you delete an I/Q file, the associated marker and header files are deleted. It is not necessary to send separate commands to copy or delete the marker and header files.
-------------	--

---

### Playing the Waveform

Play the waveform and use it to modulate the RF carrier.

1. Select the waveform from the volatile memory waveform list:
  - From the front panel:
    - a. Press **Mode** > **Dual ARB** > **Select Waveform**.
    - b. Highlight the desired waveform.
    - c. Press **Select Waveform**.
  - Remotely send the following SCPI command:

```
[ :SOURce]:RADio:ARB:WAVEform "WFM1:<file_name>"
```
2. Play the waveform:
  - From the front panel:
    - a. Press **ARB Off On** until On is highlighted.
    - b. Press **Mod On/Off** until the MOD ON annunciator appears on the display.

- c. Press **RF On/Off** until the RF ON annunciator appears on the display.

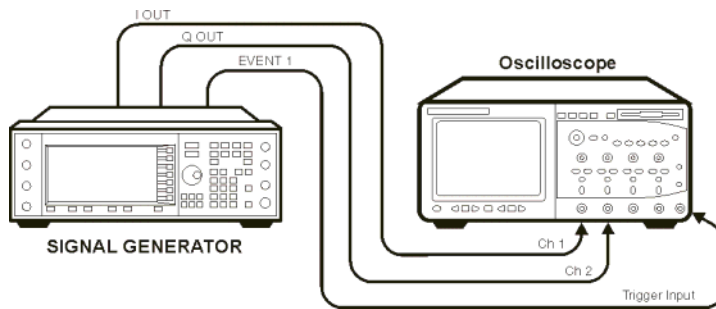
Remotely send the following SCPI commands:

```
[ :SOURce]:RADio:ARB[:STATe] ON
:OUTPut:MODulation[:STATe] ON
:OUTPut[:STATe] ON
```

## Verifying the Waveform

Perform this procedure after completing the steps in the previous procedure, [Playing the Waveform](#).

1. Connect the signal generator to an oscilloscope as shown in the figure.



2. Set an active marker point on the first waveform point for marker one.
  - From the front panel:
    - a. Press **ARB Setup > Marker Utilities > Set Markers**.
    - b. Highlight the same waveform selected in [“Playing the Waveform” on page 164](#).
    - c. Press **Set Markers > Marker 1 2 3 4** to 1.
    - d. Press **Set Markers Off All Points > Set Marker on First Point**.
  - Remotely send the following SCPI commands:
    - a. `[ :SOURce]:RADio:ARB:MARKer:CLear:ALL "WFM1:<file_name>",1`
    - b. `[ :SOURce]:RADio:ARB:MARKer:[SET]"WFM1:<file_name>",1,1,1,0.`
3. Compare the oscilloscope display to the plot of the I and Q data from the text file you created when you generated the data.

If the oscilloscope display, and the I and Q data plots differ, recheck your code. For detailed information on programmatically creating and downloading waveform data, see [“Creating Waveform Data” on page 152](#) and [“Downloading Waveform Data” on page 157](#). For information on the waveform data requirements, see [“Waveform Data Requirements” on page 132](#).

## Using the Download Utilities

Agilent provides free download utilities to download waveform data into the signal generator. The table in this section describes the capabilities of three such utilities.

For more information and to install the utilities, refer to the following URLs:

- Agilent Signal Studio Toolkit: [www.agilent.com/find/signalstudio](http://www.agilent.com/find/signalstudio)  
This software provides a graphical interface for downloading files.
- Agilent IntuiLink for PSG/ESG Signal Generators: [www.agilent.com/find/intuilink](http://www.agilent.com/find/intuilink)  
This software places icons in the Microsoft Excel and Word toolbar. Use the icons to connect to the signal generator and open a window for downloading files.
- PSG/ESG Download Assistant: [www.agilent.com/find/downloadassistant](http://www.agilent.com/find/downloadassistant)  
This software provides functions for the MATLAB environment to download waveform data.

Features	Agilent Signal Studio Toolkit	Agilent IntuiLink	PSG/ESG Download Assistant
Downloads encrypted waveform files	X		
Downloads Signal Studio waveform files	X <sup>a</sup>		
Downloads complex MATLAB waveform data			X
Downloads MATLAB files (.mat)	X		
Downloads unencrypted interleaved 16-bit I/Q files <sup>b</sup>	X	X	
Interleaves and downloads earlier 14-bit E443xB I and Q files <sup>b</sup>	X	X	
Swaps bytes for little endian order		X	
Downloads user-created marker files	X	X	X
Performs scaling	X	X	X
Starts waveform play back	X		X
Sends SCPI Commands and Queries	X		X
Builds a waveform sequence	X		X

a. Some Signal Studio products let you create and export waveform files to a PC. Signal Studio Toolkit downloads the exported files.

b. ASCII or binary format.

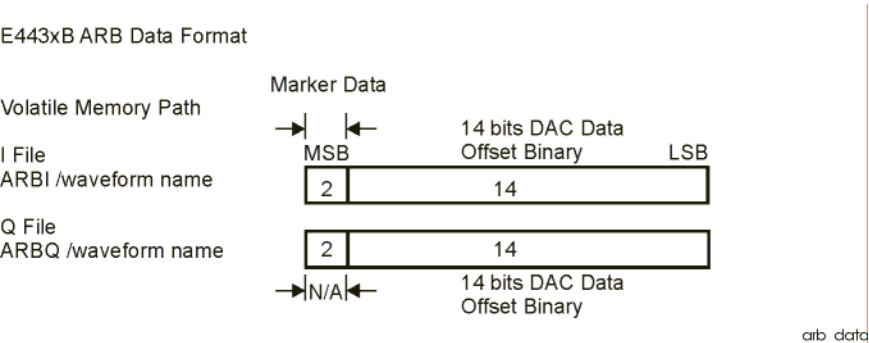
## Downloading E443xB Signal Generator Files

To download earlier E443xB model I and Q files, use the same SCPI commands as if downloading files to an E443xB signal generator. The signal generator automatically converts the E443xB files to the proper file format as described in “Waveform Structure” on page 139 and stores them in the signal generator’s memory. This conversion process causes the signal generator to take more time to download the earlier file format. To minimize the time to convert earlier E443xB files to the proper file format, store E443xB file downloads to volatile memory, and then transfer them over to non-volatile (NVWFM) memory.

**NOTE** You cannot extract waveform data downloaded as E443xB files.

E443xB Data Format

The following diagram describes the data format for the E443xB waveform files. This file structure can be compared with the new style file format shown in “Waveform Structure” on page 139. If you create new waveform files for the signal generator, use the format shown in “Waveform Data Requirements” on page 132.



Storage Locations for E443xB ARB files

Place waveforms in either volatile memory or non-volatile memory. The signal generator supports the E443xB directory structure for waveform file downloads.

Volatile Memory Storage Locations

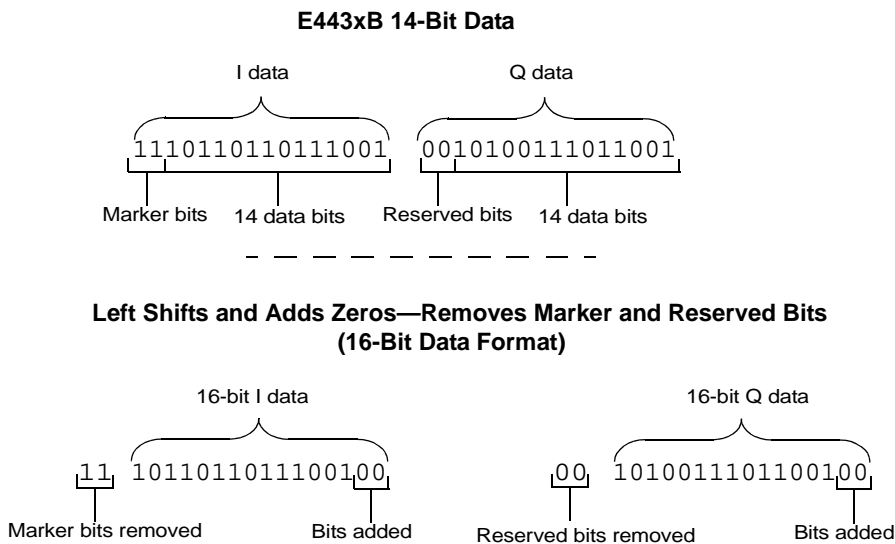
- /user/arbi/
- /user/arbq/

Non-Volatile Memory Storage Locations

- /user/nvarbi/
- /user/nvarbq/

Loading files into the above directories (volatile or non-volatile memory) does not actually store them in those directories. Instead, these directories function as “pipes” to the format translator. The signal generator performs the following functions on the E443xB data:

- Converts the 14-bit I and Q data into 16-bit data (the format required by the signal generator). Left shifts the data and appends two bits (zeros) before the least significant bit.





Extraction Method/ Memory Type	Command Syntax Options
SCPI/ volatile memory	:MMEM:DATA "ARBI:<file_name>", <I waveform block data> :MMEM:DATA "ARBQ:<file_name>", <Q waveform data>
SCPI/ non-volatile memory	:MMEM:DATA "NVARBI:<file_name>", <I waveform block data> :MMEM:DATA "NVARBQ:<file_name>", <Q waveform block data>

The variables <I waveform block data> and <Q waveform block data> represents data in the E443xB file format. The string variable <file\_name> is the name of the I and Q data file. After downloading the data, the signal generator associates a file header and marker file with the I/Q data file.

## Programming Examples

The programming examples use GPIB or LAN interfaces and are written in the following languages:

- C++
- MATLAB
- Visual Basic
- HP Basic

See [Chapter 1](#) of this programming guide for information on interfaces and I/O libraries.

The example programs are also available on the signal generator Documentation CD-ROM, which allows you to cut and paste the examples into an editor.

## C++ Programming Examples

This section contains the following programming examples:

- “Creating and Storing Offset I/Q Data—Big and Little Endian Order” on page 170
- “Creating and Storing I/Q Data—Little Endian Order” on page 174
- “Creating and Downloading I/Q Data—Big and Little Endian Order” on page 176
- “Importing and Downloading I/Q Data—Big Endian Order” on page 180
- “Importing and Downloading Using VISA—Big Endian Order” on page 183
- “Importing, Byte Swapping, Interleaving, and Downloading I and Q Data—Big and Little Endian Order” on page 187

## Creating and Storing Offset I/Q Data—Big and Little Endian Order

On the documentation CD, this programming example's name is "*offset\_iq\_c++.txt*."

This C++ programming example (compiled using Microsoft Visual C++ 6.0) follows the same coding algorithm as the MATLAB programming example "[Creating and Storing I/Q Data](#)" on page 194 and performs the following functions:

- error checking
- data creation
- data normalization
- data scaling
- I/Q signal offset from the carrier (single sideband suppressed carrier signal)
- byte swapping and interleaving for little endian order data
- I and Q interleaving for big endian order data
- binary data file storing to a PC or workstation
- reversal of the data formatting process (byte swapping, interleaving, and normalizing the data)

After creating the binary file, you can use FTP, one of the download utilities, or one of the C++ download programming examples to download the file to the signal generator.

```
// This C++ example shows how to
// 1.) Create a simple IQ waveform
// 2.) Save the waveform into the ESG/PSG Internal Arb format
//      This format is the for the E4438C, E8267C, E8267D
//      This format will not work with the ESG E443xB
// 3.) Load the internal Arb format file into an array

#include <stdio.h>
#include <string.h>
#include <math.h>

const int POINTS = 1000; // Size of waveform
const char *computer = "PCWIN";

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

// 1.) Create Simple IQ Signal *****
// This signal is a single tone on the upper
// side of the carrier and is usually referred to as
// a Single Side Band Suppressed Carrier (SSBSC) signal.
// It is nothing more than a cosine wavefomm in I
// and a sine waveform in Q.

int points = POINTS; // Number of points in the waveform
int cycles = 101; // Determines the frequency offset from the carrier
```

```

double Iwave[POINTS]; // I waveform
double Qwave[POINTS]; // Q waveform
short int waveform[2*POINTS]; // Holds interleaved I/Q data
double maxAmp = 0; // Used to Normalize waveform data
double minAmp = 0; // Used to Normalize waveform data
double scale = 1;
char buf; // Used for byte swapping
char *pChar; // Used for byte swapping
bool PC = true; // Set flag as appropriate

double phaseInc = 2.0 * 3.141592654 * cycles / points;
double phase = 0;
int i = 0;
for( i=0; i<points; i++ )
{
    phase = i * phaseInc;
    Iwave[i] = cos(phase);
    Qwave[i] = sin(phase);
}

// 2.) Save waveform in internal format *****
// Convert the I and Q data into the internal arb format
// The internal arb format is a single waveform containing interleaved IQ
// data. The I/Q data is signed short integers (16 bits).
// The data has values scaled between +-32767 where
//   DAC Value   Description
//   32767       Maximum positive value of the DAC
//   0           Zero out of the DAC
//  -32767       Maximum negative value of the DAC
// The internal arb expects the data bytes to be in Big Endian format.
// This is opposite of how short integers are saved on a PC (Little Endian).
// For this reason the data bytes are swapped before being saved.

// Find the Maximum amplitude in I and Q to normalize the data between +-1
maxAmp = Iwave[0];
minAmp = Iwave[0];
for( i=0; i<points; i++)
{
    if( maxAmp < Iwave[i] )
        maxAmp = Iwave[i];
    else if( minAmp > Iwave[i] )
        minAmp = Iwave[i];
}

```

```
    if( maxAmp < Qwave[i] )
        maxAmp = Qwave[i];
    else if( minAmp > Qwave[i] )
        minAmp = Qwave[i];
}
maxAmp = fabs(maxAmp);
minAmp = fabs(minAmp);
if( minAmp > maxAmp )
    maxAmp = minAmp;

// Convert to short integers and interleave I/Q data
scale = 32767 / maxAmp;      // Watch out for divide by zero.
for( i=0; i<points; i++)
{
    waveform[2*i] = (short)floor(Iwave[i]*scale + 0.5);
    waveform[2*i+1] = (short)floor(Qwave[i]*scale + 0.5);
}
// If on a PC swap the bytes to Big Endian
if( strcmp(computer,"PCWIN") == 0 )
//if( PC )
{
    pChar = (char *)&waveform[0];    // Character pointer to short int data
    for( i=0; i<2*points; i++ )
    {
        buf = *pChar;
        *pChar = *(pChar+1);
        *(pChar+1) = buf;
        pChar+= 2;
    }
}
// Save the data to a file
// Use FTP or one of the download assistants to download the file to the
// signal generator
char *filename = "C:\\Temp\\PsgTestFile";
FILE *stream = NULL;
stream = fopen(filename, "w+b");// Open the file
if (stream==NULL) perror ("Cannot Open File");
int numwritten = fwrite( (void *)waveform, sizeof( short ), points*2, stream );
fclose(stream);// Close the file

// 3.) Load the internal Arb format file *****
// This process is just the reverse of saving the waveform
```

```
// Read in waveform as unsigned short integers.
// Swap the bytes as necessary
// Normalize between +-1
// De-interleave the I/Q Data
// Open the file and load the internal format data
stream = fopen(filename, "r+b");// Open the file
if (stream==NULL) perror ("Cannot Open File");
int numread = fread( (void *)waveform, sizeof( short ), points*2, stream );
fclose(stream);// Close the file
// If on a PC swap the bytes back to Little Endian
if( strcmp(computer,"PCWIN") == 0 )
{
    pChar = (char *)&waveform[0];    // Character pointer to short int data
    for( i=0; i<2*points; i++ )
    {
        buf = *pChar;
        *pChar = *(pChar+1);
        *(pChar+1) = buf;
        pChar+= 2;
    }
}
// Normalize De-Interleave the IQ data
double IwaveIn[POINTS];
double QwaveIn[POINTS];
for( i=0; i<points; i++)
{
    IwaveIn[i] = waveform[2*i] / 32767.0;
    QwaveIn[i] = waveform[2*i+1] / 32767.0;
}
return 0;
}
```

### Creating and Storing I/Q Data—Little Endian Order

On the documentation CD, this programming example's name is "*CreateStore\_Data\_c++.txt*."

This C++ programming example (compiled using Metrowerks CodeWarrior 3.0) performs the following functions:

- error checking
- data creation
- byte swapping and interleaving for little endian order data
- binary data file storing to a PC or workstation

After creating the binary file, you can use FTP, one of the download utilities, or one of the C++ download programming examples to download the file to the signal generator.

```
#include <iostream>
#include <fstream>
#include <math.h>
#include <stdlib.h>

using namespace std;

int main ( void )
{
    ofstream out_stream;          // write the I/Q data to a file
    const unsigned int SAMPLES =200;    // number of sample pairs in the waveform
    const short AMPLITUDE = 32000;      // amplitude between 0 and full scale dac value
    const double two_pi = 6.2831853;

    //allocate buffer for waveform
    short* iqData = new short[2*SAMPLES]; // need two bytes for each integer
    if (!iqData)
    {
        cout << "Could not allocate data buffer." << endl;
        return 1;
    }

    out_stream.open("IQ_data");// create a data file
    if (out_stream.fail())
    {
        cout << "Input file opening failed" << endl;
        exit(1);
    }

    //generate the sample data for I and Q. The I channel will have a sine
    //wave and the Q channel will a cosine wave.
```

```

for (int i=0; i<SAMPLES; ++i)
{
    iqData[2*i] = AMPLITUDE * sin(two_pi*i/(float)SAMPLES);
    iqData[2*i+1] = AMPLITUDE * cos(two_pi*i/(float)SAMPLES);
}
// make sure bytes are in the order MSB(most significant byte) first. (PC only).

char* cptr = (char*)iqData; // cast the integer values to characters

for (int i=0; i<(4*SAMPLES); i+=2) // 4*SAMPLES
{
    char temp = cptr[i]; // swap LSB and MSB bytes
    cptr[i]=cptr[i+1];
    cptr[i+1]=temp;
}

// now write the buffer to a file

    out_stream.write((char*)iqData, 4*SAMPLES);
return 0;
}

```

## Creating and Downloading I/Q Data—Big and Little Endian Order

On the documentation CD, this programming example's name is "*CreateDwnLd\_Data\_c++.txt*."

This C++ programming example (compiled using Microsoft Visual C++ 6.0) performs the following functions:

- error checking
- data creation
- data scaling
- text file creation for viewing and debugging data
- byte swapping and interleaving for little endian order data
- interleaving for big endian order data
- data saving to an array (data block)
- data block download to the signal generator

```
// This C++ program is an example of creating and scaling
// I and Q data, and then downloading the data into the
// signal generator as an interleaved I/Q file.
// This example uses a sine and cosine wave as the I/Q
// data.
//
// Include the standard headers for SICL programming
#include <sicl.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

// Choose a GPIB, LAN, or RS-232 connection
char* instOpenString = "lan[galqaDhcp1]";
//char* instOpenString = "gpi0,19";

// Pick some maximum number of samples, based on the
// amount of memory in your computer and the signal generator.
const int NUMSAMPLES=500;

int main(int argc, char* argv[])
{
    // Create a text file to view the waveform
    // prior to downloading it to the signal generator.
    // This verifies that the data looks correct.

    char *ofile = "c:\\temp\\iq.txt";
```



```
// Create arrays to hold the I and Q data

int idata[NUMSAMPLES];
int qdata[NUMSAMPLES];

// save the number of samples into numsamples
int numsamples = NUMSAMPLES;

// Fill the I and Q buffers with the sample data
for(int index=0; index<numsamples; index++)
{
    // Create the I and Q data for the number of waveform
    // points and Scale the data (20000 * ...) as a percentage
    // of the DAC full scale (-32768 to 32767). This example
    // scales to approximately 70% of full scale.
    idata[index]=23000 * sin((4*3.14*index)/numsamples);
    qdata[index]=23000 * cos((4*3.14*index)/numsamples);
}

// Print the I and Q values to a text file. View the data
// to see if its correct and if needed, plot the data in a
// spreadsheet to help spot any problems.
FILE *outfile = fopen(ofile, "w");

if (outfile==NULL) perror ("Error opening file to write");
for(index=0; index<numsamples; index++)
{
    fprintf(outfile, "%d, %d\n", idata[index], qdata[index]);
}
fclose(outfile);

// Little endian order data, use the character array and for loop.
// If big endian order, comment out this character array and for loop,
// and use the next loop (Big Endian order data).

// We need a buffer to interleave the I and Q data.
// 4 bytes to account for 2 I bytes and 2 Q bytes.

char iqbuffer[NUMSAMPLES*4];

// Interleave I and Q, and swap bytes from little
// endian order to big endian order.
for(index=0; index<numsamples; index++)
{
```

```

    int ivalue = idata[index];
    int qvalue = qdata[index];
    iqbuffer[index*4]   = (ivalue >> 8) & 0xFF; // high byte of i
    iqbuffer[index*4+1] = ivalue & 0xFF;        // low byte of i
    iqbuffer[index*4+2] = (qvalue >> 8) & 0xFF; // high byte of q
    iqbuffer[index*4+3] = qvalue & 0xFF;        // low byte of q
}

// Big Endian order data, uncomment the following lines of code.
// Interleave the I and Q data.

// short iqbuffer[NUMSAMPLES*2];           // Big endian order, uncomment this line
// for(index=0; index<numsamples; index++) // Big endian order, uncomment this line
// {                                         // Big endian order, uncomment this line
//     iqbuffer[index*2]   = idata[index]; // Big endian order, uncomment this line
//     iqbuffer[index*2+1] = qdata[index]; // Big endian order, uncomment this line
// }                                         // Big endian order, uncomment this line

// Open a connection to write to the instrument
INST id=iopen(instOpenString);
if (!id)
{
    fprintf(stderr, "iopen failed (%s)\n", instOpenString);
    return -1;
}

// Declare variables to hold portions of the SCPI command
int bytesToSend;
char s[20];
char cmd[200];

bytesToSend = numsamples*4;           // calculate the number of bytes
sprintf(s, "%d", bytesToSend); // create a string s with that number of bytes

// The SCPI command has four parts.
// Part 1 = :MEM:DATA "filename",#
// Part 2 = length of Part 3 when written to a string
// Part 3 = length of the data in bytes. This is in s from above.
// Part 4 = the buffer of data

// Build parts 1, 2, and 3 for the I and Q data.
sprintf(cmd, ":MEM:DATA \"WF1:FILE1\", #d%d", strlen(s), bytesToSend);

```

```
// Send parts 1, 2, and 3
iwrite(id, cmd, strlen(cmd), 0, 0);

// Send part 4. Be careful to use the correct command here. In many
// programming languages, there are two methods to send SCPI commands:
// Method 1 = stop at the first '0' in the data
// Method 2 = send a fixed number of bytes, ignoring '0' in the data.
// You must find and use the correct command for Method 2.
iwrite(id, iqbuffer, bytesToSend, 0, 0);
// Send a terminating carriage return
iwrite(id, "\n", 1, 1, 0);

printf("Loaded file using the E4438C, E8267C and E8267D format\n");
return 0;
}
```

## Importing and Downloading I/Q Data—Big Endian Order

On the documentation CD, this programming example's name is "*impDwnLd\_c++.txt*."

This C++ programming example (compiled using Metrowerks CodeWarrior 3.0) assumes that the data is in big endian order and performs the following functions:

- error checking
- binary file importing from the PC or workstation.
- binary file download to the signal generator.

```
// Description: Send a file in blocks of data to a signal generator
//
#include <sic1.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// ATTENTION:
// - Configure these three lines appropriately for your instrument
//   and use before compiling and running
//
char* instOpenString = "gpib7,19"; //for LAN replace with "lan[<hostname or IP address>]"
const char* localSrcFile = "D:\\home\\TEST_WAVE"; //enter file location on PC/workstation
const char* instDestFile = "/USER/BBG1/WAVEFORM/TEST_WAVE"; //for non-volatile memory
                                                             //remove BBG1 from file path

// Size of the copy buffer
const int BUFFER_SIZE = 100*1024;

int
main()
{
    INST id=iopen(instOpenString);
    if (!id)
    {
        fprintf(stderr, "iopen failed (%s)\n", instOpenString);
        return -1;
    }

    FILE* file = fopen(localSrcFile, "rb");
    if (!file)
    {
        fprintf(stderr, "Could not open file: %s\n", localSrcFile);
        return 0;
    }
}
```

```

if( fseek( file, 0, SEEK_END ) < 0 )
{
    fprintf(stderr, "Cannot seek to the end of file.\n" );
    return 0;
}

long lenToSend = ftell(file);
printf("File size = %d\n", lenToSend);

if (fseek(file, 0, SEEK_SET) < 0)
{
    fprintf(stderr, "Cannot seek to the start of file.\n");
    return 0;
}

char* buf = new char[BUFFER_SIZE];
if (buf && lenToSend)
{
    // Prepare and send the SCPI command header
    char s[20];
    sprintf(s, "%d", lenToSend);
    int lenLen = strlen(s);
    char s2[256];
    sprintf(s2, "mmem:data \"%s\", %#d%d", instDestFile, lenLen, lenToSend);
    iwrite(id, s2, strlen(s2), 0, 0);

    // Send file in BUFFER_SIZE chunks
    long numRead;
    do
    {
        numRead = fread(buf, sizeof(char), BUFFER_SIZE, file);
        iwrite(id, buf, numRead, 0, 0);
    } while (numRead == BUFFER_SIZE);

    // Send the terminating newline and EOM
    iwrite(id, "\n", 1, 1, 0);

    delete [] buf;
}
else
{

```

```
        fprintf(stderr, "Could not allocate memory for copy buffer\n");  
    }  
  
    fclose(file);  
    iclose(id);  
    return 0;  
}
```

## Importing and Downloading Using VISA—Big Endian Order

On the documentation CD, this programming example's name is "*DownLoad\_Visa\_c++.txt*."

This C++ programming example (compiled using Microsoft Visual C++ 6.0) assumes that the data is in big endian order and performs the following functions:

- error checking
- binary file importing from the PC or workstation
- binary file download to the signal generator's non-volatile memory

To load the waveform data to volatile (WFM1) memory, change the `instDestfile` declaration to: "USER/BBG1/WAVEFORM/".

```
//*****
// PROGRAM NAME:Download_Visa_c++.cpp
//
// PROGRAM DESCRIPTION:Sample test program to download ARB waveform data. Send a
// file in chunks of ascii data to the signal generator.
//
// NOTE: You must have the Agilent IO Libraries installed to run this program.
//
// This example uses the LAN/TCPIP to download a file to the baseband generator's
// non-volatile memory. The program allocates a memory buffer on the PC or
// workstation of 102400 bytes (100*1024 bytes). The actual size of the buffer is
// limited by the memory on your PC or workstation, so the buffer size can be
// increased or decreased to meet your system limitations.
//
// While this program uses the LAN/TCPIP to download a waveform file into
// non-volatile memory, it can be modified to store files in volatile memory
// WFM1 using GPIB by setting the instrOpenString = "TCPIP0::xxx.xxx.xxx.xxx::INSTR"
// declaration with "GPIB::19::INSTR"
//
// The program also includes some error checking to alert you when problems arise
// while trying to download files. This includes checking to see if the file exists.
//*****
// IMPORTANT: Replace the xxx.xxx.xxx.xxx IP address in the instOpenString declaration
// in the code below with the IP address of your signal generator. (or you can use the
// instrument's hostname). Replace the localSrcFile and instDestFile directory paths
// as needed.
//*****

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

```
//  
// IMPORTANT:  
//   Configure the following three lines correctly before compiling and running  
  
char* instOpenString = "TCPIP0:xxx.xxx.xxx.xxx:INSTR"; // your instrument's IP address  
  
const char* localSrcFile = "\\Files\\IQ_DataC";  
  
const char* instDestFile = "/USER/WAVEFORM/IQ_DataC";  
  
const int BUFFER_SIZE = 100*1024; // Size of the copy buffer  
  
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;  
    }  
  
    FILE* file = fopen(localSrcFile, "rb"); // Open local source file for binary reading  
  
    if (!file) // If any errors display the error and exit the program  
    {  
        fprintf(stderr, "Could not open file: %s\n", localSrcFile);  
return 0;  
    }  
  
    if( fseek( file, 0, SEEK_END ) < 0 )  
    {  
        fprintf(stderr, "Cannot lseek to the end of file.\n" );  
        return 0;  
    }  
}
```



```

long lenToSend = ftell(file); // Number of bytes in the file

printf("File size = %d\n", lenToSend);

if (fseek(file, 0, SEEK_SET) < 0)
{
    fprintf(stderr, "Cannot lseek to the start of file.\n");
    return 0;
}

unsigned char* buf = new unsigned char[BUFFER_SIZE]; // Allocate char buffer memory

if (buf && lenToSend)
{
    // Do not send the EOI (end of instruction) terminator on any write except the
    // last one

    viSetAttribute( vi, VI_ATTR_SEND_END_EN, 0 );

    // Prepare and send the SCPI command header

    char s[20];
    sprintf(s, "%d", lenToSend);

    int lenLen = strlen(s);
    unsigned char s2[256];

    // Write the command mmem:data and the header. The number lenLen represents the
    // number of bytes and the actual number of bytes is the variable lenToSend

    sprintf((char*)s2, "mmem:data \"%s\", #d%d", instDestFile, lenLen, lenToSend);

    // Send the command and header to the signal generator

    viWrite(vi, s2, strlen((char*)s2), 0);

    long numRead;

    // Send file in BUFFER_SIZE chunks to the signal generator

    do

```

```
{
    numRead = fread(buf, sizeof(char), BUFFER_SIZE, file);

    viWrite(vi, buf, numRead, 0);

} while (numRead == BUFFER_SIZE);

// Send the terminating newline and EOI

viSetAttribute( vi, VI_ATTR_SEND_END_EN, 1 );

char* newLine = "\n";

viWrite(vi, (unsigned char*)newLine, 1, 0);

delete [] buf;
}
else
{
    fprintf(stderr, "Could not allocate memory for copy buffer\n");
}

fclose(file);
viClose(vi);
viClose(defaultRM);

return 0;
}
```

### Importing, Byte Swapping, Interleaving, and Downloading I and Q Data—Big and Little Endian Order

On the documentation CD, this programming example's name is "*impDwnLd2\_c++.txt*."

This C++ programming example (compiled using Microsoft Visual C++ 6.0) performs the following functions:

- error checking
- binary file importing (earlier E443xB or current model signal generators)
- byte swapping and interleaving for little endian order data
- data interleaving for big endian order data
- data scaling
- binary file download for earlier E443xB data or current signal generator formatted data

```
// This C++ program is an example of loading I and Q
// data into an E443xB, E4438C, E8267C, or E8267D signal
// generator.
//
// It reads the I and Q data from a binary data file
// and then writes the data to the instrument.

// Include the standard headers for SICL programming
#include <sicl.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// Choose a GPIB, LAN, or RS-232 connection
char* instOpenString = "gpi0,19";

// Pick some maximum number of samples, based on the
// amount of memory in your computer and your waveforms.
const int MAXSAMPLES=50000;

int main(int argc, char* argv[])
{
    // These are the I and Q input files.
    // Some compilers will allow '/' in the directory
    // names. Older compilers might need '\\' in the
    // directory names. It depends on your operating system
    // and compiler.
    char *ifile = "c:\\SignalGenerator\\data\\BurstAlI.bin";
    char *qfile = "c:\\SignalGenerator\\data\\BurstAlQ.bin";
```

```
// This is a text file to which we will write the
// I and Q data just for debugging purposes. It is
// a good programming practice to check your data
// in this way before attempting to write it to
// the instrument.
char *ofile = "c:\\SignalGenerator\\data\\iq.txt";

// Create arrays to hold the I and Q data
int idata[MAXSAMPLES];
int qdata[MAXSAMPLES];

// Often we must modify, scale, or offset the data
// before loading it into the instrument. These
// buffers are used for that purpose. Since each
// sample is 16 bits, and a character only holds
// 8 bits, we must make these arrays twice as long
// as the I and Q data arrays.
char ibuffer[MAXSAMPLES*2];
char qbuffer[MAXSAMPLES*2];

// For the E4438C or E8267C/67D, we might also need to interleave
// the I and Q data. This buffer is used for that
// purpose. In this case, this buffer must hold
// both I and Q data so it needs to be four times
// as big as the data arrays.
char iqbuffer[MAXSAMPLES*4];

// Declare variables which will be used later
bool done;
FILE *infile;
int index, numsamples, i1, i2, ivalue;

// In this example, we'll assume the data files have
// the I and Q data in binary form as unsigned 16 bit integers.
// This next block reads those binary files. If your
// data is in some other format, then replace this block
// with appropriate code for reading your format.
// First read I values
done = false;
index = 0;
infile = fopen(infile, "rb");
if (infile==NULL) perror ("Error opening file to read");
```

```

while(!done)
{
    i1 = fgetc(infile); // read the first byte
    if(i1==EOF) break;
    i2 = fgetc(infile); // read the next byte
    if(i2==EOF) break;
    ivalue=i1+i2*256;    // put the two bytes together
    // note that the above format is for a little endian
    // processor such as Intel. Reverse the order for
    // a big endian processor such as Motorola, HP, or Sun
    idata[index++]=ivalue;
    if(index==MAXSAMPLES) break;
}
fclose(infile);

// Then read Q values
index = 0;
infile = fopen(qfile, "rb");
if (infile==NULL) perror ("Error opening file to read");
while(!done)
{
    i1 = fgetc(infile); // read the first byte
    if(i1==EOF) break;
    i2 = fgetc(infile); // read the next byte
    if(i2==EOF) break;
    ivalue=i1+i2*256;    // put the two bytes together
    // note that the above format is for a little endian
    // processor such as Intel. Reverse the order for
    // a big endian processor such as Motorola, HP, or Sun
    qdata[index++]=ivalue;
    if(index==MAXSAMPLES) break;
}
fclose(infile);

// Remember the number of samples which were read from the file.
numsamples = index;

// Print the I and Q values to a text file. If you are
// having trouble, look in the file and see if your I and
// Q data looks correct. Plot the data from this file if
// that helps you to diagnose the problem.
FILE *outfile = fopen(ofile, "w");

```

```
if (outfile==NULL) perror ("Error opening file to write");
for(index=0; index<numsamples; index++)
{
    fprintf(outfile, "%d, %d\n", idata[index], qdata[index]);
}
fclose(outfile);

// The E443xB, E4438C, E8267C or E8267D all use big-endian
// processors. If your software is running on a little-endian
// processor such as Intel, then you will need to swap the
// bytes in the data before sending it to the signal generator.

// The arrays ibuffer and qbuffer are used to hold the data
// after any byte swapping, shifting or scaling.

// In this example, we'll assume that the data is in the format
// of the E443xB without markers. In other words, the data
// is in the range 0-16383.
// 0 gives negative full-scale output
// 8192 gives 0 V output
// 16383 gives positive full-scale output
// If this is not the scaling of your data, then you will need
// to scale your data appropriately in the next two blocks.

// ibuffer and qbuffer will hold the data in the E443xB format.
// No scaling is needed, however we need to swap the byte order
// on a little endian computer. Remove the byte swapping
// if you are using a big endian computer.
for(index=0; index<numsamples; index++)
{
    int ivalue = idata[index];
    int qvalue = qdata[index];
    ibuffer[index*2]   = (ivalue >> 8) & 0xFF; // high byte of i
    ibuffer[index*2+1] = ivalue & 0xFF;         // low byte of i
    qbuffer[index*2]   = (qvalue >> 8) & 0xFF; // high byte of q
    qbuffer[index*2+1] = qvalue & 0xFF;         // low byte of q
}

// iqbuffer will hold the data in the E4438C, E8267C, E8267D
// format. In this format, the I and Q data is interleaved.
// The data is in the range -32768 to 32767.
// -32768 gives negative full-scale output
```

```
//      0 gives 0 V output
//      32767 gives positive full-scale output
// From these ranges, it appears you should offset the
// data by 8192 and scale it by 4. However, due to the
// interpolators in these products, it is better to scale
// the data by a number less than four. Commonly a good
// choice is 70% of 4 which is 2.8.
// By default, the signal generator scales data to 70%
// If you scale the data here, you may want to change the
// signal generator scaling to 100%
// Also we need to swap the byte order on a little endian
// computer. This code also works for big endian order data
// since it swaps bytes based on the order.
for(index=0; index<numsamples; index++)
{
    int iscaled = 2.8*(idata[index]-8192); // shift and scale
    int qscaled = 2.8*(qdata[index]-8192); // shift and scale
    iqbuffer[index*4]  = (iscaled >> 8) & 0xFF; // high byte of i
    iqbuffer[index*4+1] = iscaled & 0xFF;        // low byte of i
    iqbuffer[index*4+2] = (qscaled >> 8) & 0xFF; // high byte of q
    iqbuffer[index*4+3] = qscaled & 0xFF;        // low byte of q
}

// Open a connection to write to the instrument
INST id=iopen(instOpenString);
if (!id)
{
    fprintf(stderr, "iopen failed (%s)\n", instOpenString);
    return -1;
}

// Declare variables which will be used later
int bytesToSend;
char s[20];
char cmd[200];

// The E4438C, E8267C and E8267D accept the E443xB format.
// so we can use this next section on any of these signal generators.
// However the E443xB format only uses 14 bits.

bytesToSend = numsamples*2; // calculate the number of bytes
sprintf(s, "%d", bytesToSend); // create a string s with that number of bytes
```

```
// The SCPI command has four parts.
// Part 1 = :MEM:DATA "filename",
// Part 2 = length of Part 3 when written to a string
// Part 3 = length of the data in bytes. This is in s from above.
// Part 4 = the buffer of data

// Build parts 1, 2, and 3 for the I data.
sprintf(cmd, ":MEM:DATA \"ARBI:FILE1\", %#d%d", strlen(s), bytesToSend);
// Send parts 1, 2, and 3
iwrite(id, cmd, strlen(cmd), 0, 0);
// Send part 4. Be careful to use the correct command here. In many
// programming languages, there are two methods to send SCPI commands:
// Method 1 = stop at the first '0' in the data
// Method 2 = send a fixed number of bytes, ignoring '0' in the data.
// You must find and use the correct command for Method 2.
iwrite(id, ibuffer, bytesToSend, 0, 0);
// Send a terminating carriage return
iwrite(id, "\n", 1, 1, 0);

// Identical to the section above, except for the Q data.
sprintf(cmd, ":MEM:DATA \"ARBQ:FILE1\", %#d%d", strlen(s), bytesToSend);
iwrite(id, cmd, strlen(cmd), 0, 0);
iwrite(id, qbuffer, bytesToSend, 0, 0);
iwrite(id, "\n", 1, 1, 0);

printf("Loaded FILE1 using the E443xB format\n");

// The E4438C, E8267C and E8267D have a newer faster format which
// allows 16 bits to be used. However this format is not accepted in
// the E443xB. Therefore do not use this next section for the E443xB.

printf("Note: Loading FILE2 on a E443xB will cause \"ERROR: 208, I/O error\"\n");

// Identical to the I and Q sections above except
// a) The I and Q data are interleaved
// b) The buffer of I+Q is twice as long as the I buffer was.
// c) The SCPI command uses WFM1 instead of ARBI and ARBQ.
bytesToSend = numsamples*4;
sprintf(s, "%d", bytesToSend);
sprintf(cmd, ":mem:data \"WFM1:FILE2\", %#d%d", strlen(s), bytesToSend);
iwrite(id, cmd, strlen(cmd), 0, 0);
```



```
iwrite(id, iqbuffer, bytesToSend, 0, 0);  
iwrite(id, "\n", 1, 1, 0);  
printf("Loaded FILE2 using the E4438C, E8267C and E8267D format\n");  
return 0;  
}
```

## MATLAB Programming Examples

This section contains the following programming examples:

- “Creating and Storing I/Q Data” on page 194
- “Creating and Downloading a Pulse” on page 197

### Creating and Storing I/Q Data

On the documentation CD, this programming example’s name is “*offset\_iq\_ml.m.*”

This MATLAB programming example follows the same coding algorithm as the C++ programming example “Creating and Storing Offset I/Q Data–Big and Little Endian Order” on page 170 and performs the following functions:

- error checking
- data creation
- data normalization
- data scaling
- I/Q signal offset from the carrier (single sideband suppressed carrier signal)
- byte swapping and interleaving for little endian order data
- I and Q interleaving for big endian order data
- binary data file storing to a PC or workstation
- reversal of the data formatting process (byte swapping, interleaving, and normalizing the data)

```
function main
% Using MatLab this example shows how to
% 1.) Create a simple IQ waveform
% 2.) Save the waveform into the ESG/PSG Internal Arb format
%     This format is for the E4438C, E8267C, and E8267D
%     This format will not work with the earlier E443xB ESG
% 3.) Load the internal Arb format file into a MatLab array

% 1.) Create Simple IQ Signal *****
% This signal is a single tone on the upper
% side of the carrier and is usually referred to as
% a Single Side Band Suppressed Carrier (SSBSC) signal.
% It is nothing more than a cosine wavefomm in I
% and a sine waveform in Q.
%
points = 1000;      % Number of points in the waveform
cycles = 101;      % Determines the frequency offset from the carrier

phaseInc = 2*pi*cycles/points;
phase = phaseInc * [0:points-1];

Iwave = cos(phase);
```

```
Qwave = sin(phase);

% 2.) Save waveform in internal format *****
% Convert the I and Q data into the internal arb format
% The internal arb format is a single waveform containing interleaved IQ
% data. The I/Q data is signed short integers (16 bits).
% The data has values scaled between +-32767 where
%   DAC Value   Description
%   32767       Maximum positive value of the DAC
%   0           Zero out of the DAC
%  -32767       Maximum negative value of the DAC
% The internal arb expects the data bytes to be in Big Endian format.
% This is opposite of how short integers are saved on a PC (Little Endian).
% For this reason the data bytes are swapped before being saved.

% Interleave the IQ data
waveform(1:2:2*points) = Iwave;
waveform(2:2:2*points) = Qwave;
%[Iwave;Qwave];
%waveform = waveform(:)';

% Normalize the data between +-1
waveform = waveform / max(abs(waveform)); % Watch out for divide by zero.

% Scale to use full range of the DAC
waveform = round(waveform * 32767); % Data is now effectively signed short integer values

% waveform = round(waveform * (32767 / max(abs(waveform)))); % More efficient than previous two
% steps!

% PRESERVE THE BIT PATTERN but convert the waveform to
% unsigned short integers so the bytes can be swapped.
% Note: Can't swap the bytes of signed short integers in MatLab.
waveform = uint16(mod(65536 + waveform,65536)); %

% If on a PC swap the bytes to Big Endian
if strcmp( computer, 'PCWIN' )
    waveform = bitor(bitshift(waveform,-8),bitshift(waveform,8));
end

% Save the data to a file
% Note: The waveform is saved as unsigned short integers. However,
%       the actual bit pattern is that of signed short integers and
```

```
%      that is how the ESG/PSG interprets them.
filename = 'C:\Temp\PsgTestFile';
[FID, message] = fopen(filename,'w');% Open a file to write data
if FID == -1 error('Cannot Open File'); end
fwrite(FID,waveform,'unsigned short');% write to the file
fclose(FID);                          % close the file

% 3.) Load the internal Arb format file *****
% This process is just the reverse of saving the waveform
% Read in waveform as unsigned short integers.
% Swap the bytes as necessary
% Convert to signed integers then normalize between +-1
% De-interleave the I/Q Data

% Open the file and load the internal format data
[FID, message] = fopen(filename,'r');% Open file to read data
if FID == -1 error('Cannot Open File'); end
[internalWave,n] = fread(FID, 'uint16');% read the IQ file
fclose(FID);% close the file

internalWave = internalWave';    % Conver from column array to row array

% If on a PC swap the bytes back to Little Endian
if strcmp( computer, 'PCWIN' )    % Put the bytes into the correct order
    internalWave= bitor(bitshift(internalWave,-8),bitshift(bitand(internalWave,255),8));
end

% convert unsigned to signed representation
internalWave = double(internalWave);
tmp = (internalWave > 32767.0) * 65536;
iqWave = (internalWave - tmp) ./ 32767; % and normalize the data

% De-Interleave the IQ data
IwaveIn = iqWave(1:2:n);
QwaveIn = iqWave(2:2:n);
```

## Creating and Downloading a Pulse

On the documentation CD, this programming example's name is "*pulsepat.m*."

This MATLAB programming example performs the following functions:

- I and Q data creation for 10 pulses
- marker file creation
- data scaling
- downloading using PSG/ESG Download Assistant functions (see ["Using the Download Utilities" on page 166](#) for more information)

```
% Script file: pulsepat.m
%
% Purpose:
%To calculate and download an arbitrary waveform file that simulates a
%simple antenna scan pulse pattern to the PSG vector signal generator.
%
% Define Variables:
% n -- counting variable (no units)
% t -- time (seconds)
% rise -- raised cosine pulse rise-time definition (samples)
% on -- pulse on-time definition (samples)
% fall -- raised cosine pulse fall-time definition (samples)
% i -- in-phase modulation signal
% q -- quadrature modulation signal

n=4; % defines the number of points in the rise-time and fall-time
t=-1:2/n:1-2/n; % number of points translated to time
rise=(1+sin(t*pi/2))/2; % defines the pulse rise-time shape
on=ones(1,120); % defines the pulse on-time characteristics
fall=(1+sin(-t*pi/2))/2; % defines the pulse fall-time shape
off=zeros(1,896); % defines the pulse off-time characteristics

% arrange the i-samples and scale the amplitude to simulate an antenna scan
% pattern comprised of 10 pulses
i = .707*[rise on fall off...
[.9*[rise on fall off]]...
[.8*[rise on fall off]]...
[.7*[rise on fall off]]...
[.6*[rise on fall off]]...
[.5*[rise on fall off]]...
[.4*[rise on fall off]]...
[.3*[rise on fall off]]...
[.2*[rise on fall off]]...
[.1*[rise on fall off]]];

% set the q-samples to all zeroes
q = zeros(1,10240);
```

```
% define a composite iq matrix for download to the PSG using the
% PSG/ESG Download Assistant
IQData = [i + (j * q)];

% define a marker matrix and activate a marker to indicate the beginning of the waveform
Markers = zeros(2,length(IQData));      % fill marker array with zero, i.e no markers set
Markers(1,1) = 1;                        % set marker to first point of playback

% make a new connection to the PSG over the GPIB interface
io = agt_newconnection('gpib',0,19);

% verify that communication with the PSG has been established
[status, status_description, query_result] = agt_query(io, '*idn?');
if (status < 0) return; end

% set the carrier frequency and power level on the PSG using the PSG Download Assistant
[status, status_description] = agt_sendcommand(io, 'SOURce:FREQuency 20000000000');
[status, status_description] = agt_sendcommand(io, 'POWer 0');

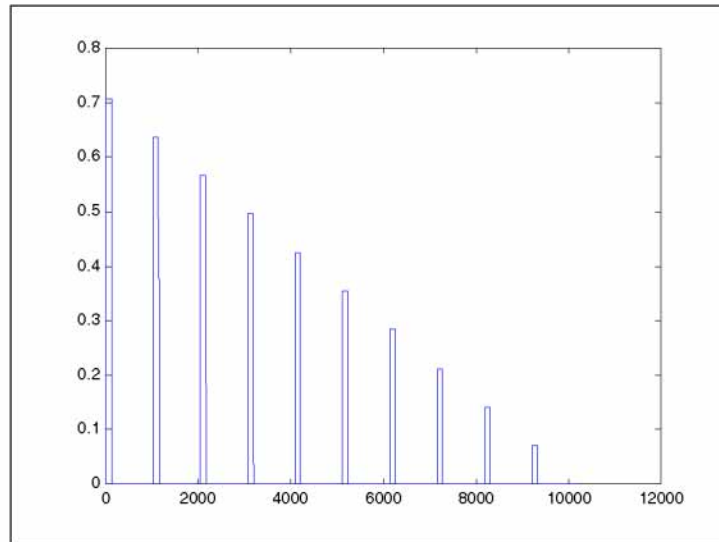
% define the ARB sample clock for playback
sampclk = 400000000;

% download the iq waveform to the PSG baseband generator for playback
[status, status_description] = agt_waveformload(io, IQData, 'pulsepat', sampclk, 'play',
'no_normscale', Markers);

% turn on RF output power
[status, status_description ] = agt_sendcommand( io, 'OUTPut:STATe ON' )
```

You can test your program by performing a simulated plot of the in-phase modulation signal in Matlab (see [Figure 4-1 on page 199](#)). To do this, enter `plot (i)` at the Matlab command prompt.

Figure 4-1 Simulated Plot of In-Phase Signal



The following additional Matlab M-file pulse programming examples are also available on the *Documentation CD-ROM*:

barker.m	This programming example calculates and downloads an arbitrary waveform file that simulates a simple 7-bit barker RADAR signal to the PSG vector signal generator.
chirp.m	This programming example calculates and downloads an arbitrary waveform file that simulates a simple compressed pulse RADAR signal using linear FM chirp to the PSG vector signal generator.
FM.m	This programming example calculates and downloads an arbitrary waveform file that simulates a single tone FM signal with a rate of 6 KHz, deviation of $\pm 14.3$ KH, Bessel null of dev/rate=2.404 to the PSG vector signal generator.
nchirp.m	This programming example calculates and downloads an arbitrary waveform file that simulates a simple compressed pulse RADAR signal using non-linear FM chirp to the PSG vector signal generator.
pulse.m	This programming example calculates and downloads an arbitrary waveform file that simulates a simple pulse signal to the PSG vector signal generator.
pulsedroop.m	This programming example calculates and downloads an arbitrary waveform file that simulates a simple pulse signal with pulse droop to the PSG vector signal generator.

## Visual Basic Programming Examples

### Creating I/Q Data—Little Endian Order

On the documentation CD, this programming example's name is "*Create\_IQData\_vb.txt*."

This Visual Basic programming example, using Microsoft Visual Basic 6.0, uses little endian order data, and performs the following functions:

- error checking
- I an Q integer array creation
- I an Q data interleaving
- byte swapping to convert to big endian order
- binary data file storing to a PC or workstation

Once the file is created, you can download the file to the signal generator using FTP (see "[FTP Procedures](#)" on page 150).

```
*****
' Program Name: Create_IQData
' Program Description: This program creates a sine and cosine wave using 200 I/Q data
' samples. Each I and Q value is represented by a 2 byte integer. The sample points are
' calculated, scaled using the AMPLITUDE constant of 32767, and then stored in an array
' named iq_data. The AMPLITUDE scaling allows for full range I/Q modulator DAC values.
' Data must be in 2's complement, MSB/LSB big-endian format. If your PC uses LSB/MSB
' format, then the integer bytes must be swapped. This program converts the integer
' array values to hex data types and then swaps the byte positions before saving the
' data to the IQ_DataVB file.
*****

Private Sub Create_IQData()
Dim index As Integer
Dim AMPLITUDE As Integer
Dim pi As Double
Dim loByte As Byte
Dim hiByte As Byte
Dim loHex As String
Dim hiHex As String
Dim strSrc As String
Dim numPoints As Integer
Dim FileHandle As Integer
Dim data As Byte
Dim iq_data() As Byte
Dim strFilename As String

strFilename = "C:\IQ_DataVB"

Const SAMPLES = 200      ' Number of sample PAIRS of I and Q integers for the waveform
```



```

AMPLITUDE = 32767      ' Scale the amplitude for full range of the signal generators
                        ' I/Q modulator DAC

pi = 3.141592

Dim intIQ_Data(0 To 2 * SAMPLES - 1) 'Array for I and Q integers: 400
ReDim iq_data(0 To (4 * SAMPLES - 1)) 'Need MSB and LSB bytes for each integer value: 800

'Create an integer array of I/Q pairs

For index = 0 To (SAMPLES - 1)
    intIQ_Data(2 * index) = CInt(AMPLITUDE * Sin(2 * pi * index / SAMPLES))
    intIQ_Data(2 * index + 1) = CInt(AMPLITUDE * Cos(2 * pi * index / SAMPLES))
Next index

'Convert each integer value to a hex string and then write into the iq_data byte array
'MSB, LSB ordered
For index = 0 To (2 * SAMPLES - 1)
    strSrc = Hex(intIQ_Data(index)) 'convert the integer to a hex value

    If Len(strSrc) <> 4 Then
        strSrc = String(4 - Len(strSrc), "0") & strSrc 'Convert to hex format i.e "800F"
    End If                                           'Pad with 0's if needed to get 4
                                                    'characters i.e '0' to "0000"

    hiHex = Mid$(strSrc, 1, 2) 'Get the first two hex values (MSB)
    loHex = Mid$(strSrc, 3, 2) 'Get the next two hex values (LSB)
    loByte = CByte("&H" & loHex) 'Convert to byte data type LSB
    hiByte = CByte("&H" & hiHex) 'Convert to byte data type MSB

    iq_data(2 * index) = hiByte 'MSB into first byte
    iq_data(2 * index + 1) = loByte 'LSB into second byte

Next index

'Now write the data to the file

FileHandle = FreeFile() 'Get a file number

numPoints = UBound(iq_data) 'Get the number of bytes in the file

Open strFilename For Binary Access Write As #FileHandle Len = numPoints + 1

```

```
On Error GoTo file_error

    For index = 0 To (numPoints)
        data = iq_data(index)
        Put #FileHandle, index + 1, data 'Write the I/Q data to the file
    Next index

Close #FileHandle

Call MsgBox("Data written to file " & strFilename, vbOKOnly, "Download")

Exit Sub

file_error:
    MsgBox Err.Description
    Close #FileHandle

End Sub
```

### Downloading I/Q Data

On the documentation CD, this programming example's name is "*Download\_File\_vb.txt*."

This Visual Basic programming example, using Microsoft Visual Basic 6.0, downloads the file created in "[Creating I/Q Data—Little Endian Order](#)" on page 200 into non-volatile memory using a LAN connection. To use GPIB, replace the instOpenString object declaration with "GPIB::19::INSTR". To download the data into volatile memory, change the instDestfile declaration to "USER/BBG1/WAVEFORM/".

---

**NOTE** The example program listed here uses the VISA COM I/O API, which includes the WriteIEEEBlock method. This method eliminates the need to format the download command with arbitrary block information such as defining number of bytes and byte numbers. Refer to "[SCPI Command Line Structure](#)" on page 147 for more information.

---

This program also includes some error checking to alert you when problems arise while trying to download files. This includes checking to see if the file exists.

```
' *****
' Program Name: Download_File
' Program Description: This program uses Microsoft Visual Basic 6.0 and the Agilent
' VISA COM I/O Library to download a waveform file to the signal generator.
'
' The program downloads a file (the previously created 'IQ_DataVB' file) to the signal
' generator. Refer to the Programming Guide for information on binary
' data requirements for file downloads. The waveform data 'IQ_DataVB' is
' downloaded to the signal generator's non-volatile memory(NVWFM)
' " /USER/WAVEFORM/IQ_DataVB". For volatile memory(WFM1) download to the
```

```
' " /USER/BBG1/WAVEFORM/IQ_DataVB" directory.
'
' You must reference the Agilent VISA COM Resource Manager and VISA COM 1.0 Type
' Library in your Visual Basic project in the Project/References menu.
' The VISA COM 1.0 Type Library, corresponds to VISACOM.tlb and the Agilent
' VISA COM Resource Manager, corresponds to AgtRM.DLL.
' The VISA COM 488.2 Formatted I/O 1.0, corresponds to the BasicFormattedIO.dll
' Use a statement such as "Dim Instr As VisaComLib.FormattedIO488" to
' create the formatted I/O reference and use
' "Set Instr = New VisaComLib.FormattedIO488" to create the actual object.
' *****
' IMPORTANT: Use the TCPIP address of your signal generator in the rm.Open
' declaraion. If you are using the GPIB interface in your project use "GPIB::19::INSTR"
' in the rm.Open declaration.
' *****

Private Sub Download_File()
' The following four lines declare IO objects and instantiate them.
Dim rm As VisaComLib.ResourceManager
Set rm = New AgilentRMLib.SRMCLs
Dim SigGen As VisaComLib.FormattedIO488
Set SigGen = New VisaComLib.FormattedIO488

' NOTE: Use the IP address of your signal generator in the rm.Open declaration
Set SigGen.IO = rm.Open("TCPIP0::000.000.000.000")

Dim data As Byte
Dim iq_data() As Byte
Dim FileHandle As Integer
Dim numPoints As Integer
Dim index As Integer
Dim Header As String
Dim response As String
Dim hiByte As String
Dim loByte As String
Dim strFilename As String

strFilename = "C:\IQ_DataVB" 'File Name and location on PC
                        'Data will be saved to the signal generator's NVWFM
                        '\USER/WAVEFORM/IQ_DataVB directory.

FileHandle = FreeFile()
```

```
On Error GoTo errorhandler

With SigGen
    'Set up the signal generator to accept a download
    .IO.Timeout = 5000    'Timeout 50 seconds
    .WriteString "*RST"    'Reset the signal generator.
End With

numPoints = (FileLen(strFilename))    'Get number of bytes in the file: 800 bytes

ReDim iq_data(0 To numPoints - 1)    'Dimension the iq_data array to the
                                     'size of the IQ_DataVB file: 800 bytes

Open strFilename For Binary Access Read As #FileHandle    'Open the file for binary read
On Error GoTo file_error

For index = 0 To (numPoints - 1)    'Write the IQ_DataVB data to the iq_data array
    Get #FileHandle, index + 1, data    '(index+1) is the record number
    iq_data(index) = data
Next index

    Close #FileHandle    'Close the file

'Write the command to the Header string. NOTE: syntax
Header = "MEM:DATA " & "/USER/WAVEFORM/IQ_DataVB" & ", "

'Now write the data to the signal generator's non-volatile memory (NVWFM)

SigGen.WriteIEEEBlock Header, iq_data

SigGen.WriteString "*OPC?"    'Wait for the operation to complete
response = SigGen.ReadString    'Signal generator reponse to the OPC? query
Call MsgBox("Data downloaded to the signal generator", vbOKOnly, "Download")
Exit Sub

errorhandler:
    MsgBox Err.Description, vbExclamation, "Error Occurred", Err.HelpFile, Err.HelpContext
Exit Sub

file_error:
    Call MsgBox(Err.Description, vbOKOnly)    'Display any error message
    Close #FileHandle

End Sub
```

## HP Basic Programming Examples

This section contains the following programming examples:

- “Creating and Downloading Waveform Data Using HP BASIC for Windows®” on page 205
- “Creating and Downloading Waveform Data Using HP BASIC for UNIX” on page 208
- “Creating and Downloading E443xB Waveform Data Using HP BASIC for Windows” on page 210
- “Creating and Downloading E443xB Waveform Data Using HP Basic for UNIX” on page 212

### Creating and Downloading Waveform Data Using HP BASIC for Windows®<sup>1</sup>

On the documentation CD, this programming example’s name is “*hpbasicWin.txt*.”

The following program will download a waveform using HP Basic for Windows into volatile ARB memory. The waveform generated by this program is the same as the default SINE\_TEST\_WFM waveform file available in the signal generator’s waveform memory. This code is similar to the code shown for BASIC for UNIX but there is a formatting difference in line 130 and line 140.

To download into non-volatile memory, replace line 190 with:

```
190 OUTPUT @PSG USING "#,K";MMEM:DATA ""NVWFM:testfile"", #"
```

As discussed at the beginning of this section, I and Q waveform data is interleaved into one file in 2’s compliment form and a marker file is associated with this I/Q waveform file.

In the Output commands, USING “#,K” formats the data. The pound symbol (#) suppresses the automatic EOL (End of Line) output. This allows multiple output commands to be concatenated as if they were a single output. The “K” instructs HP Basic to output the following numbers or strings in the default format.

```
10 ! RE-SAVE "BASIC_Win_file"
20 Num_points=200
30 ALLOCATE INTEGER Int_array(1:Num_points*2)
40 DEG
50 FOR I=1 TO Num_points*2 STEP 2
60     Int_array(I)=INT(32767*(SIN(I*360/Num_points)))
70 NEXT I
80 FOR I=2 TO Num_points*2 STEP 2
90     Int_array(I)=INT(32767*(COS(I*360/Num_points)))
100 NEXT I
110 PRINT "Data Generated"
120 Nbytes=4*Num_points
130 ASSIGN @Psg TO 719
140 ASSIGN @Psgb TO 719;FORMAT MSB FIRST
150 Nbytes$=VAL$(Nbytes)
160 Ndigits=LEN(Nbytes$)
```

---

1. Windows and MS Windows are U.S registered trademarks of Microsoft Corporation.

```

170  Ndigits$=VAL$(Ndigits)
180  WAIT 1
190  OUTPUT @Psg USING "#,K";":MMEM:DATA " "WFM1:data_file" ",#"
200  OUTPUT @Psg USING "#,K";Ndigits$
210  OUTPUT @Psg USING "#,K";Nbytes$
220  WAIT 1
230  OUTPUT @Psgb;Int_array(*)
240  OUTPUT @Psg;END
250  ASSIGN @Psg TO *
260  ASSIGN @Psgb TO *
270  PRINT
280  PRINT "*END*"
290  END

```

### Program Comments

10:	Program file name
20:	Sets the number of points in the waveform.
30:	Allocates integer data array for I and Q waveform points.
40:	Sets HP BASIC to use degrees for cosine and sine functions.
50:	Sets up first loop for I waveform points.
60:	Calculate and interleave I waveform points.
70:	End of loop
80	Sets up second loop for Q waveform points.
90:	Calculate and interleave Q waveform points.
100:	End of loop.
120:	Calculates number of bytes in I/Q waveform.
130:	Opens an I/O path to the signal generator using GPIB. 7 is the address of the GPIB card in the computer, and 19 is the address of the signal generator. This I/O path is used to send ASCII data to the signal generator.
140:	Opens an I/O path for sending binary data to the signal generator.
150:	Creates an ASCII string representation of the number of bytes in the waveform.
160 to 170:	Finds the number of digits in Nbytes.
190:	Sends the first part of the SCPI command, MEM:DATA along with the name of the file, data_file, that will receive the waveform data. The name, data_file, will appear in the signal generator's memory catalog.

**Program Comments (Continued)**

200 to 210:	Sends the rest of the ASCII header.
230:	Sends the binary data. Note that <code>PSGb</code> is the binary I/O path.
240:	Sends an End-of-Line to terminate the transmission.
250 to 260:	Closes the connections to the signal generator.
290:	End the program.

## Creating and Downloading Waveform Data Using HP BASIC for UNIX

On the documentation CD, this programming example's name is "*hpbasicUx.txt*."

The following program shows you how to download waveforms using HP Basic for UNIX. The code is similar to that shown for HP BASIC for Windows, but there is a formatting difference in line 130 and line 140.

To download into non-volatile memory, replace line 190 with:

```
190 OUTPUT @PSG USING "#,K";":MMEM:DATA ""NVWFM:testfile"", #"
```

As discussed at the beginning of this section, I and Q waveform data is interleaved into one file in 2's compliment form and a marker file is associated with this I/Q waveform file.

In the Output commands, USING "#,K" formats the data. The pound symbol (#) suppresses the automatic EOL (End of Line) output. This allows multiple output commands to be concatenated as if they were a single output. The "K" instructs HP BASIC to output the following numbers or strings in the default format.

```
10 ! RE-SAVE "UNIX_file"
20 Num_points=200
30 ALLOCATE INTEGER Int_array(1:Num_points*2)
40 DEG
50 FOR I=1 TO Num_points*2 STEP 2
60     Int_array(I)=INT(32767*(SIN(I*360/Num_points)))
70 NEXT I
80 FOR I=2 TO Num_points*2 STEP 2
90     Int_array(I)=INT(32767*(COS(I*360/Num_points)))
100 NEXT I
110 PRINT "Data generated "
120 Nbytes=4*Num_points
130 ASSIGN @Psg TO 719;FORMAT ON
140 ASSIGN @Psgb TO 719;FORMAT OFF
150 Nbytes$=VAL$(Nbytes)
160 Ndigits=LEN(Nbytes$)
170 Ndigits$=VAL$(Ndigits)
180 WAIT 1
190 OUTPUT @Psg USING "#,K";":MMEM:DATA ""WFM1:data_file"",#"
200 OUTPUT @Psg USING "#,K";Ndigits$
210 OUTPUT @Psg USING "#,K";Nbytes$
220 WAIT 1
230 OUTPUT @Psgb;Int_array(*)
240 WAIT 2
241 OUTPUT @Psg;END
250 ASSIGN @Psg TO *
260 ASSIGN @Psgb TO *
270 PRINT
280 PRINT "**END**"
```



290    END

### Program Comments

10:	Program file name
20:	Sets the number of points in the waveform.
30:	Allocates integer data array for I and Q waveform points.
40:	Sets HP BASIC to use degrees for cosine and sine functions.
50:	Sets up first loop for I waveform points.
60:	Calculate and interleave I waveform points.
70:	End of loop
80	Sets up second loop for Q waveform points.
90:	Calculate and interleave Q waveform points.
100:	End of loop.
120:	Calculates number of bytes in I/Q waveform.
130:	Opens an I/O path to the signal generator using GPIB. 7 is the address of the GPIB card in the computer, and 19 is the address of the signal generator. This I/O path is used to send ASCII data to the signal generator.
140:	Opens an I/O path for sending binary data to the signal generator.
150:	Creates an ASCII string representation of the number of bytes in the waveform.
160 to 170:	Finds the number of digits in Nbytes.
190:	Sends the first part of the SCPI command, MEM:DATA along with the name of the file, data_file, that will receive the waveform data. The name, data_file, will appear in the signal generator's memory catalog.
200 to 210:	Sends the rest of the ASCII header.
230:	Sends the binary data. Note that PSGB is the binary I/O path.
240:	Sends an End-of-Line to terminate the transmission.
250 to 260:	Closes the connections to the signal generator.
290:	End the program.

## Creating and Downloading E443xB Waveform Data Using HP BASIC for Windows

On the documentation CD, this programming example's name is "*hpbasicWin2.txt*."

The following program shows you how to download waveforms using HP Basic for Windows into volatile ARB memory. This program is similar to the following program example as well as the previous examples. The difference between BASIC for UNIX and BASIC for Windows is the way the formatting, for the most significant bit (MSB) on lines 110 and 120, is handled.

To download into non-volatile ARB memory, replace line 80 with:

```
160 OUTPUT @PSG USING "#,K";":MMEM:DATA ""NVARBI:testfile"", #"
```

and replace line 130 with:

```
210 OUTPUT @PSG USING "#,K";":MMEM:DATA ""NVARBQ:testfile"", #"
```

First, the I waveform data is put into an array of integers called `Iwfm_data` and the Q waveform data is put into an array of integers called `Qwfm_data`. The variable `Nbytes` is set to equal the number of bytes in the I waveform data. This should be twice the number of integers in `Iwfm_data`, since an integer is 2 bytes. Input integers must be between 0 and 16383.

In the Output commands, USING "`#,K`" formats the data. The pound symbol (`#`) suppresses the automatic EOL (End of Line) output. This allows multiple output commands to be concatenated as if they were a single output. The "`K`" instructs HP Basic to output the following numbers or strings in the default format.

```
10      ! RE-SAVE "ARB_IQ_Win_file"
20      Num_points=200
30      ALLOCATE INTEGER Iwfm_data(1:Num_points),Qwfm_data(1:Num_points)
40      DEG
50      FOR I=1 TO Num_points
60          Iwfm_data(I)=INT(8191*(SIN(I*360/Num_points))+8192)
70          Qwfm_data(I)=INT(8191*(COS(I*360/Num_points))+8192)
80      NEXT I
90      PRINT "Data Generated"
100     Nbytes=2*Num_points
110     ASSIGN @Psg TO 719
120     !ASSIGN @Psgb TO 719;FORMAT MSB FIRST
130     Nbytes$=VAL$(Nbytes)
140     Ndigits=LEN(Nbytes$)
150     Ndigits$=VAL$(Ndigits)
160     OUTPUT @Psg USING "#,K";":MMEM:DATA ""ARBI:file_name_1"",#,"
170     OUTPUT @Psg USING "#,K";Ndigits$
180     OUTPUT @Psg USING "#,K";Nbytes$
190     OUTPUT @Psgb;Iwfm_data(*)
200     OUTPUT @Psg;END
210     OUTPUT @Psg USING "#,K";":MMEM:DATA ""ARBQ:file_name_1"",#,"
220     OUTPUT @Psg USING "#,K";Ndigits$
230     OUTPUT @Psg USING "#,K";Nbytes$
240     OUTPUT @Psgb;Qwfm_data(*)
```

```

250  OUTPUT @Psg;END
260  ASSIGN @Psg TO *
270  ASSIGN @Psgb TO *
280  PRINT
290  PRINT "**END*"
300  END

```

#### Program Comments

10:	Program file name.
20	Sets the number of points in the waveform.
30:	Defines arrays for I and Q waveform points. Sets them to be integer arrays.
40:	Sets HP BASIC to use degrees for cosine and sine functions.
50:	Sets up loop to calculate waveform points.
60:	Calculates I waveform points.
70:	Calculates Q waveform points.
80:	End of loop.
160 and 210:	The I and Q waveform files have the same name
90 to 300:	See the <a href="#">table on page 206</a> for program comments.

## Creating and Downloading E443xB Waveform Data Using HP Basic for UNIX

On the documentation CD, this programming example's name is "*hpbasicUx2.txt*."

The following program shows you how to download waveforms using HP BASIC for UNIX. It is similar to the previous program example. The difference is the way the formatting for the most significant bit (MSB) on lines is handled.

First, the I waveform data is put into an array of integers called *Iwfm\_data* and the Q waveform data is put into an array of integers called *Qwfm\_data*. The variable *Nbytes* is set to equal the number of bytes in the I waveform data. This should be twice the number of integers in *Iwfm\_data*, since an integer is represented 2 bytes. Input integers must be between 0 and 16383.

In the Output commands, USING "#,K" formats the data. The pound symbol (#) suppresses the automatic EOL (End of Line) output. This allows multiple output commands to be concatenated as if they were a single output. The "K" instructs HP BASIC to output the following numbers or strings in the default format.

```
10      ! RE-SAVE "ARB_IQ_file"
20      Num_points=200
30      ALLOCATE INTEGER Iwfm_data(1:Num_points),Qwfm_data(1:Num_points)
40      DEG
50      FOR I=1 TO Num_points
60          Iwfm_data(I)=INT(8191*(SIN(I*360/Num_points))+8192)
70          Qwfm_data(I)=INT(8191*(COS(I*360/Num_points))+8192)
80      NEXT I
90      PRINT "Data Generated"
100     Nbytes=2*Num_points
110     ASSIGN @Psg TO 719;FORMAT ON
120     ASSIGN @Psgb TO 719;FORMAT OFF
130     Nbytes$=VAL$(Nbytes)
140     Ndigits=LEN(Nbytes$)
150     Ndigits$=VAL$(Ndigits)
160     OUTPUT @Psg USING "#,K";":MMEM:DATA " "ARBI:file_name_1","","#
170     OUTPUT @Psg USING "#,K";Ndigits$
180     OUTPUT @Psg USING "#,K";Nbytes$
190     OUTPUT @Psgb;Iwfm_data(*)
200     OUTPUT @Psg;END
210     OUTPUT @Psg USING "#,K";":MMEM:DATA " "ARBQ:file_name_1","","#
220     OUTPUT @Psg USING "#,K";Ndigits$
230     OUTPUT @Psg USING "#,K";Nbytes$
240     OUTPUT @Psgb;Qwfm_data(*)
250     OUTPUT @Psg;END
260     ASSIGN @Psg TO *
270     ASSIGN @Psgb TO *
280     PRINT
290     PRINT "**END**"
```

300    END

### Program Comments

10:	Program file name.
20	Sets the number of points in the waveform.
30:	Defines arrays for I and Q waveform points. Sets them to be integer arrays.
40:	Sets HP BASIC to use degrees for cosine and sine functions.
50:	Sets up loop to calculate waveform points.
60:	Calculates I waveform points.
70:	Calculates Q waveform points.
80:	End of loop.
160 and 210:	The I and Q waveform files have the same name
90 to 300	See the <a href="#">table on page 209</a> for program comments.

## Troubleshooting Waveform Files

Symptom	Possible Cause
ERROR 224, Text file busy	<p>Attempting to download a waveform that has the same name as the waveform currently being played by the signal generator.</p> <p>To solve the problem, either change the name of the waveform being downloaded or turn off the ARB.</p>
ERROR 628, DAC over range	<p>The amplitude of the signal exceeds the DAC input range. The typical causes are unforeseen overshoot (DAC values within range) or the input values exceed the DAC range.</p> <p>To solve the problem, scale or reduce the DAC input values. For more information, see <a href="#">“DAC Input Values” on page 135</a>.</p>
ERROR 629, File format invalid	<p>The signal generator requires a minimum of 60 samples to build a waveform and the same number of I and Q data points.</p>
ERROR -321, Out of memory	<p>There is not enough space in the ARB memory for the waveform file being downloaded.</p> <p>To solve the problem, either reduce the file size of the waveform file or delete unnecessary files from ARB memory.</p>
No RF Output	<p>The marker RF blanking function may be active. To check for and turn RF blanking off, press <b>Mode &gt; Dual ARB &gt; ARB Setup &gt; Marker Utilities &gt; Marker Routing &gt; Pulse/RF Blank &gt; None</b>. This problem occurs when the file header contains unspecified settings and a previously played waveform used the marker RF blanking function.</p> <p>For more information on the marker functions, see the <i>ES257D/67D PSG Signal Generators User's Guide</i>.</p>
Undesired output signal	<p>Check for the following:</p> <ul style="list-style-type: none"> <li>• The data was downloaded in little endian order. See <a href="#">“Little Endian and Big Endian (Byte Order)” on page 134</a> for more information.</li> <li>• The waveform contains an odd number of samples. An odd number of samples can cause waveform discontinuity. See <a href="#">“Waveform Phase Continuity” on page 142</a> for more information.</li> </ul>

---

## 5 Creating and Downloading User-Data Files

This chapter explains the requirements and process of downloading user-data and contains the following sections:

- “User Bit/Binary File Data Downloads” on page 215
- “FIR Filter Coefficients Downloads” on page 219
- “Downloads Directly into Pattern RAM (PRAM)” on page 221
- “Save and Recall Instrument State Files” on page 225
- “Download User Flatness Corrections Using C++ and VISA” on page 235
- “Data Transfer Troubleshooting” on page 239

### User Bit/Binary File Data Downloads

---

**NOTE** This feature is available only in E8257D PSG vector signal generators with Option 601 or 602.

---

The signal generator accepts user file data downloads. The files can be in either binary or bit format with the data represented as bytes. Both file types can be stored in the signal generator’s non-volatile memory (see “Waveform Memory” on page 144 for more information on memory).

- In binary format all 8 bits of the byte are taken as data and used.
- In bit format the number of bits in the file is known and the non-data bits in the last byte are discarded.

After downloading the files, they can be selected as the transmitting data source. This section contains information on transferring user file data from a PC to the signal generator. It explains how to download user files into the signal generator’s memory and modulate the carrier signal with those files.

When a file is selected for use in Real-time Custom mode, the file is modulated as a continuous, unframed stream of data, according to the modulation type, symbol rate, and filtering associated with the selected format.

When a user file is selected as the data source, the signal generator’s firmware loads the data into waveform memory, and sets the other control bits depending on the operating mode, regardless of whether framed or unframed transmission is selected. In this manner, user files are mapped into waveform memory bit-by-bit; one bit per 32 bit control word.

Unlike pattern RAM (PRAM) downloads (see page 221), user files contain “data field” information only. The control data bits required for files downloaded directly into PRAM are not required for user files.

## Data Requirements and Limitations

1. Data must be in binary format. SCPI specifies data represented in bytes.
2. For binary downloads, bit length must be a multiple of 8.

SCPI specifies that data is represented in bytes and the binary memory stores data as bytes. If the length (in bits) of the original data pattern is not a multiple of 8, you may need to perform one of the following actions:

- add additional bits to complete the byte
  - replicate the data pattern without discontinuity until the total length is a multiple of 8 bits
  - truncate and discard bits until you reach a string length that is a multiple of 8
  - use a bit file and download to bit memory instead
3. Download size limitations are directly proportional to the available memory space, and the signal generator's pattern RAM storage size (Option 601 = 800 kB or Option 602 = 6.4 MB).

The maximum memory for bit and binary user data is less than the maximum memory for PRAM data. You may have to delete files from memory before downloading larger files.

If the data fields absolutely must be continuous data streams, and the size of the data exceeds the available PRAM, then real-time data and synchronization can be supplied by an external data source to the front-panel DATA, DATA CLOCK, and SYMBOL SYNC connectors.

---

**NOTE** References to *pattern RAM* (PRAM) are for descriptive purposes only and relate to the manner in which the memory is being used. PRAM and volatile waveform memory (WFM1) actually utilize the same storage media.

---

## Bit and Binary Directories

User files can be downloaded to a bit (/user/bit/) or binary (/user/bin/) directory in either volatile or non-volatile memory.

---

**NOTE** File names are limited to 23 characters.

---

### Bit Directory Downloads

The bit directory (/user/bit/) accepts data in integer number of bits, up to the maximum available memory. For additional information on signal generator memory, see [“Waveform Memory” on page 144](#).

The data length in bytes for files downloaded to bit memory is equal to the number of significant bits plus seven, divided by eight, then rounded down to the nearest integer. Each file has a 16-byte header associated with it.

There must be enough bytes to contain the specified number of bits. If the number of bits is not a multiple of 8, the least significant bits of the last byte are ignored.

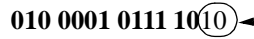
For example, specifying 14 bits of a 16-bit string using the command :MEMory:DATA:BIT "file\_name",14,#12Qz results in the last 2 bits being ignored. See the following figure.



010 0001 0111 1010      original user-defined data contains 2 bytes, 16 bits total

SCPI command sets bit count to 14; the last 2 bits are ignored

010 0001 0111 1010




---

**NOTE**    A bit directory provides more versatility, and is the preferred memory location for user file downloads.

---

### SCPI Commands

Send the following command to download the user file data into the signal generator's bit directory:

```
:MEMory:DATA:BIT "<file_name>",<bit count>,<datablock>
```

#### Example

```
:MEMory:DATA:BIT "file_name",16,#16Qz$0&5
```

file\_name            provides the user file name as it will appear in the signal generator's bit catalog

16                    Number of bits used.

#1                    1 decimal digits will be used to define the number of data bytes.

6                    6 bytes of data will follow

Qz\$0&5              the ASCII representation of the 48 bits of data that are downloaded to the signal generator, however not all ASCII values are printable

### Querying the Waveform Data

Use the following SCPI command to query the user bit data file from a binary directory:

```
:MEMory:DATA:BIT? "<file_name>"
```

The output format is the same as the input format.

### Binary Directory Downloads

The binary directory (/user/bin/) requires that data be formatted in bytes. Files stored or downloaded to a binary directory are converted to bit files prior to editing in the bit file editor, after which they are stored in a bit directory as bit files. A bit directory is preferred for user file downloads.

## SCPI Commands

```
:MMEM:DATA "<file_name>", <data_block>
```

Send this command to download the user file data into the signal generator's binary directory. The variable <file\_name> denotes the name of the downloaded user file stored in the signal generator.

---

**NOTE** The command syntax and downloading method for arbitrary block data depends on the programming language and software libraries you are using.

If you are using Agilent Technologies' VISA COM I/O Library (available in the Agilent IO Libraries for Windows Version M.01.01.04), you can use the `WriteIEEEBlock` method, which does not use the #ABC formatting parameter. Refer to [“Visual Basic Programming Examples” on page 200](#) for an example of this method.

---

## Sample Command Line

```
:MMEM:DATA "file_name", #ABC
```

file_name	the name of the user file stored in the signal generator's memory
#	indicates the start of the data block
A	the number of decimal digits to follow in B
B	a decimal number specifying the number of data bytes in C
C	the binary user file data

## Example

```
:MMEM:DATA "<file_name>",#2151&2S?4g@07p!897
```

<file_name>	provides the user file name as it will appear in the signal generator's binary memory catalog
#	indicates the start of the data block
2	defines the number of decimal digits to follow in “B”
15	denotes how many bytes of data are to follow
1&2S?4g@07p!897	the ASCII representation of the binary data that is downloaded to the signal generator, however not all ASCII values are printable

## Querying the Waveform Data

Use the following SCPI command line to query user file data from a binary memory location:

```
:MMEM:DATA? "file_name"
```

The output format is the same as the input format and includes the file length and file size information.

## Selecting Downloaded User Files as the Transmitted Data

Use the following steps to select the desired user file from the catalog of user files as a continuous stream of unframed data for a custom modulation.

Via the front panel:

1. For custom modulation, press **Mode > Custom > Real Time I/Q Baseband > Data > User File.** and highlight the desired file in the catalog.

```
[ :SOURce]:RADio:CUSTom:DATA "BIT:<file_name>" or  
[ :SOURce]:RADio:CUSTom:DATA "<file_name>" for binary files
```

2. Highlight the desired file in the catalog of user files.
3. Press **Select File > Custom Off On** to **On**.

```
[ :SOURce]:RADio:CUSTom[:STATe] On
```

4. Modulate and activate the carrier:

- a. Set the carrier frequency.

```
[ :SOURce]:FREQuency:FIXed 2.5GHZ
```

- b. Set the carrier amplitude.

```
[ :SOURce]:POWer[:LEVel][:IMMediate][:AMPLitude] -10.0DBM
```

- c. Turn on modulation.

```
:OUTPut:MODulation[:STATe] ON
```

- d. Turn on the RF output.

```
:OUTPut[:STATe] ON
```

## FIR Filter Coefficients Downloads

---

**NOTE** This feature is available only in E8257D PSG vector signal generators with Option 601 or 602.

---

The signal generator accepts finite impulse response (FIR) filter coefficient downloads. After downloading the coefficients, these user-defined FIR filter coefficient values can be selected as the filtering mechanism for the active digital communications standard.

### Data Requirements and Limitations

- Data must be in ASCII format. The signal generator processes FIR filter coefficients as floating point numbers.
- Data must be in List format. FIR filter coefficient data is processed as a list by the signal generator's firmware. See ["Sample Command Line" on page 223](#).
- Filters containing more symbols than the hardware allows are not selectable for that configuration.

The Real Time I/Q Baseband FIR filter files are limited to 1024 taps (coefficients), 64 symbols, and a 16-times oversample ratio. FIR filter files with more than 64 symbols cannot be used.

The ARB Waveform Generator FIR filter files are limited to 512 taps and 512 symbols.

- The oversample ratio (OSR) is the number of filter taps per symbol. Oversample ratios from 1 through 32 are possible. The maximum combination of OSR and symbols allowed is 32 symbols with an OSR of 32.
- The sampling period ( $\Delta t$ ) is equal to the inverse of the sampling rate (FS). The sampling rate is equal to the symbol rate multiplied by the oversample ratio. For example, for a symbol rate of 270.83 ksp/s, if the oversample ratio is 4, the sampling rate is 1083.32 kHz and  $\Delta t$  (inverse of FS) is 923.088 ns.

## Downloading FIR Filter Coefficients

Use the following SCPI command line to download FIR filter coefficients from the PC to the signal generator's FIR memory:

```
:MEMory:DATA:FIR "<file_name>","osr,coefficient{,coefficient}
```

Use the following SCPI command line to query list data from FIR memory:

```
:MEMory:DATA:FIR? "<file_name>"
```

### Sample Command Line

The following SCPI command downloads a set of FIR filter coefficient values (the values are for a Gaussian filter) and names the file "FIR1":

```
:MEMory:DATA:FIR "FIR1",4,0,0,0,0,0,0.000001,0.000012,0.000132,0.001101,  
0.006743,0.030588,0.103676,0.265790,0.523849,0.809508,1,1,0.809508,0.523849,  
0.265790,0.103676,0.030588,0.006743,0.001101,0.000132,0.000012,0.000001,0,  
0,0,0,0
```

FIR1                      assigns the name FIR1 to the associated OSR (over sample ratio) and coefficient values. The file is then represented with this name in the FIR File catalog.

4                         specifies the oversample ratio.

0,0,0,0,0,0,  
0.000001,...            represent FIR filter coefficients.

## Selecting a Downloaded User FIR Filter as the Active Filter

### Using FIR Filter Data for Custom Modulation

Use the following steps to select user FIR filter data as the active filter for a custom modulation format.

Press **Mode > Custom >**

For the Real Time I/Q Baseband mode:

- Press **Real Time I/Q Baseband > Filter > Select > User Fir > (Highlight File) > Select File**  
Press **Mode Setup > Custom On**

Via the remote interface:

```
[ :SOURce]:RADio:CUSTom:FILTer "<file_name>"  
[ :SOURce]:RADio:CUSTom[:STATe] On
```

For the Arb Waveform Generator mode:

- Press **Arb Waveform Generator > Digital Mod Define > Filter > Select > User Fir > (Highlight File) > Select File**  
Press **Mode Setup > Digital Modulation On**  
Via the remote interface:  
[:SOURce]:RADio:DMODulation:ARB:FILTer "<file\_name>"  
[:SOURce]:RADio:DMODulation:ARB[:STATe] On

## Downloads Directly into Pattern RAM (PRAM)

---

**NOTE** This feature is available only in E8257D PSG vector signal generators with Option 601 or 602.

---

**NOTE** References to *pattern RAM* (PRAM) are for descriptive purposes only, relating to the manner in which the memory is being used. PRAM and volatile waveform memory (WFM1) use the same memory.

---

Typically, the signal generator's firmware generates the required data and framing structure and loads this data into Pattern RAM (PRAM). The data is read by the baseband generator, which in turn is input to the I/Q modulator. The signal generator can also accept data downloads directly into PRAM from a computer. Programs created with applications such as MATLAB<sup>®1</sup> or MathCad<sup>®2</sup> can generate data which can be downloaded directly into PRAM in either a list format or a block format. Direct downloads to PRAM allow complete control over bursting, which is especially helpful for designing experimental or proprietary framing schemes.

The signal generator's baseband generator assembly builds modulation schemes by reading data stored in PRAM and constructing framing protocols according to the data patterns present. PRAM data can be manipulated (types of protocols changed, standard protocols modified or customized, etc.) using either the front panel interface, or remote commands.

## Preliminary Setup

---

**CAUTION** Set up the digital communications format *before* downloading data. This enables the signal generator to define the modulation format, filter, and data clock. Activating the digital communications format after the data has been downloaded to PRAM can corrupt the downloaded data.

---



---

1. ~~MATLAB~~ is a registered trademark of MathWorks, Inc.

2. Mathcad is a registered trademark of Mathsoft Engineering & Education Inc.

## Data Requirements and Limitations

### 1. Data format:

*List Format:* Because list format downloads are parsed before they are loaded into PRAM, data must be 8-bit, unsigned integers, from 0 to 255.

*Block Format:* Because the baseband generator reads binary data from the data generator, data must be in binary form.

### 2. Total (data bits plus control bits) download size limitations are 8 MB with Option 601 and 64 MB with Option 602. Each sample for PRAM uses 4 bytes of data.

A data pattern file containing 8 megabits of modulation data must contain another 56 megabits of control information. A file of this size requires 8 MB of memory.

### 3. For every bit of modulation data (bit 0), you must provide 7 bits of control information (bits 1-7).

The signal generator processes data in bytes. Each byte contains 1 bit of “data field” information, and seven bits of control information associated with the data field bit. See the following table for the required data and control bits.

Bit	Function	Value	Comments
0	Data	0/1	The data to be modulated; “unspecified” when burst (bit 2) = 0.
1	Reserved	0	Always 0.
2	Burst	0/1	Set to 1 = RF on. Set to 0 = RF off. For non-bursted, non-TDMA systems, this bit is set to 1 for all memory locations, leaving RF output on continuously. For framed data, this bit is set to 1 for <i>on</i> timeslots and 0 for <i>off</i> timeslots
3	Reserved	0	Always 0.
4	Reserved	1	Always 1.
5	Reserved	0	Always 0.
6	Event 1 Output	0/1	Set to 1 = a level transition at the EVENT 1 BNC connector. Use examples: as a marker output to trigger external hardware when data pattern restarts; toggling in alternate addresses to create a data-synchronous pulse train.
7	Pattern Reset	0/1	Set to 0 = continue to next sequential memory address. Set to 1 = end of memory and restart memory playback. Set to 0 for all bytes except last address of PRAM, where 1 restarts pattern.



### Sample Command Line

A sample command line:

```
:MEMory:DATA:PRAM:FILE:BLOCK "<file name>",#ABC
```

# indicates the start of the data block.  
A the number of decimal digits to follow in *B*.  
B a decimal number specifying the number of data bytes in *C*.  
C the binary data.

### Example 1

```
:MEMory:DATA:PRAM:FILE:BLOCK "<new_file>",#1912s4%7*9!
```

<new\_File> name of the PRAM file as it will appear in waveform memory  
# indicates the start of the data block.  
1 1 decimal digits to follow.  
9 9 bytes of data to follow.  
12s4%7\*9! the ASCII representation of the binary data downloaded to the signal generator, however not all ASCII values are printable

## Modulating and Activating the Carrier

The following section explains how to modulate the carrier with the data downloaded to PRAM, first from the front panel interface, and then via remote SCPI commands.

### Via the Front Panel

1. Set the carrier frequency to 2.5 Ghz (Frequency > 2.5 > GHz).
2. Set the carrier amplitude -10.0 dBm (Amplitude > -10 > dBm).
3. Turn modulation on (press **Mod On/Off** until the display annunciator reads MOD ON).
4. Activate the RF output (press **RF On/Off** until the display annunciator reads RF ON).

### Via the Remote Interface

Send the following SCPI commands to modulate and activate the carrier.

1. Set the carrier frequency to 2.5 Ghz:  
[:SOURce]:FREQuency:FIXed 2.5GHZ
2. Set the carrier power to -10.0 dBm:  
[:SOURce]:POWER[:LEVel][:IMMediate][:AMPLitude] -10.0DBM
3. Activate the modulation:  
:OUTPut:MODulation[:STATe] ON
4. Activate the RF output:  
:OUTPut[:STATe] ON



## Viewing a PRAM Waveform

After the waveform data is written to PRAM, the data pattern can be viewed using an oscilloscope. There is delay (approximately 12-symbols) between a state change in the burst bit and the corresponding effect at the RF out. This delay varies with symbol rate and filter settings, and requires compensation to advance the burst bit in the downloaded PRAM file.

## 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 *E8257D/67D PSG Signal Generators User's Guide* and *E8257D/67D PSG Signal Generators Key Reference* 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 *E8257D/67D PSG Signal Generators 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
```

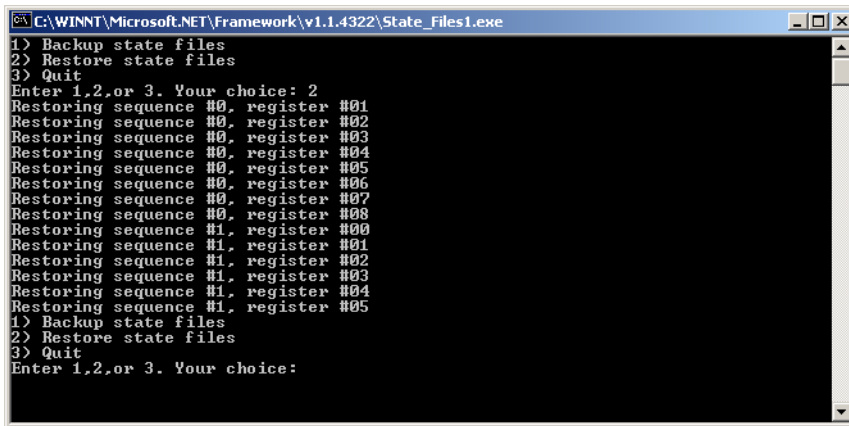
## Save and Recall Programming Example

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

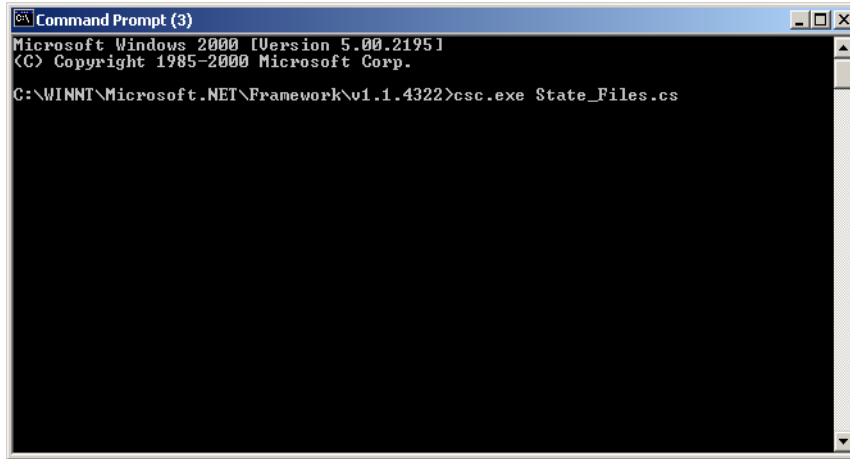
The Microsoft .NET Framework is a platform for creating Web Services and applications. There are three components of the .NET Framework: the common language runtime, class libraries, and Active Server Pages, called ASP.NET. Refer to the Microsoft website for more information on the .NET Framework.

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 PSG'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.



The `State_Files.cs` program is listed below. You can copy this program from the examples directory on the PSG CD-ROM E8251-90351.

```
//*****
// 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::psg3::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 PSG
        }
    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\\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);
}
}
```

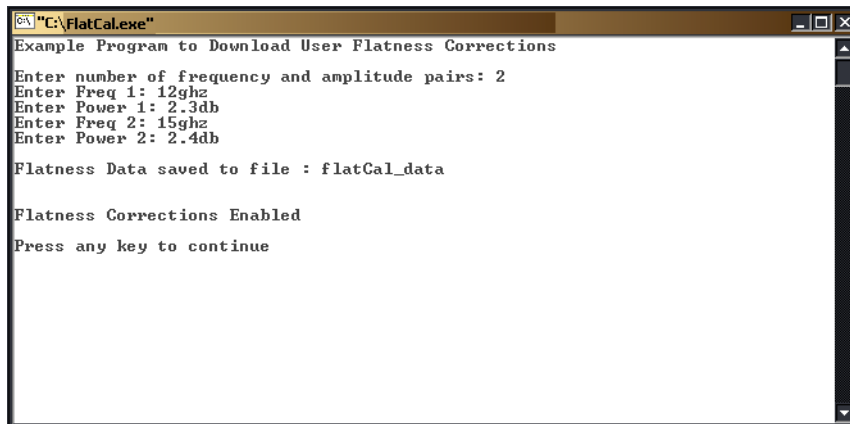
## Download User Flatness Corrections Using C++ and VISA

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/C++ Programming Examples” on page 32](#) 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 via 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 PSG'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 PSG
char c = ','; //comma separator for SCPI command
for (int j=0; j< num; j++) //download pairs to the PSG
{
    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 PSG 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 PSG
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 PSG
//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;
}
```

## Data Transfer Troubleshooting

---

**NOTE** This feature is available only in E4438C ESG Vector Signal Generators with Option 001/601 or 002/602.

---

This section is divided by the following data transfer method:

[“User Bit/Binary File Download Problems” on page 239](#)

[“User FIR Filter Coefficient File Download Problems” on page 240](#)

[“Direct PRAM File Download Problems” on page 240](#)

### User Bit/Binary File Download Problems

Download problems can be caused by data corruption or communication protocol errors.

#### Data Corruption

If the signal generator displays an error message after a file download is attempted:

1. Check to see that the SCPI download command and syntax is correct.
2. Check that the end of line character is sent.
3. Review the [“Data Requirements and Limitations” on page 216](#). The signal generator uses binary data data.

#### Communication Errors

If the signal generator does not respond to communication attempts:

1. Check to see that the SCPI download command and syntax is correct.
2. Refer to the [“Using VISA Assistant” on page 16](#) section in this guide.

#### Using Externally Generated, Real-Time Data for Large Files

The data fields absolutely must be continuous data streams, and the size of the data exceeds the available PRAM, real-time data and synchronization can be supplied by an external data source to the front-panel DATA, DATA CLOCK, and SYMBOL SYNC connectors. This data can be continuously transmitted, or can be framed by supplying a data-synchronous burst pulse to the EXT1 INPUT connector on the front panel.

## User FIR Filter Coefficient File Download Problems

Symptom	Possible Cause
ERROR -321, Out of memory	<p>There is not enough memory available for the FIR coefficient file being downloaded.</p> <p>To solve the problem, either reduce the file size of the FIR file or delete unnecessary files from memory.</p>
ERROR -223, Too much data	<p>User FIR filter has too many symbols.</p> <p>Real Time cannot use a filter that has more than 64 symbols (512 symbols maximum for ARB). You may have specified an incorrect oversample ratio in the filter table editor.</p>

---

**NOTE** Review [“Data Requirements and Limitations”](#) on page 219.

---

## Direct PRAM File Download Problems

Symptom	Possible Cause
The transmitted pattern is interspersed with random, unwanted data.	<p>Pattern reset bit not set.</p> <p>Ensure that the pattern reset bit (bit 7, value 128) is set on the last byte of your downloaded data.</p>
ERROR -223, Too much data	<p>PRAM download exceeds the size of PRAM memory.</p> <p>Either use a smaller pattern or get more memory by ordering the appropriate hardware option.</p>

---

**NOTE** Review [“Data Requirements and Limitations”](#) on page 222.

---



**Symbols**

.NET framework, 225

**Numerics**

2's complement data format, 138

8757D pass-thru commands, using, 60

**A**

abort function, 8

add device, 4

address

    GPIB address, 7

    IP address, 13

Agilent

    BASIC, 35

    SICL, 34

    VISA, 6, 12, 24, 34

Agilent Connection Expert, 7

Agilent GPIB card, 5

Agilent IO Config, 17, 63

Agilent IO Libraries, 3

Agilent IO libraries, 6, 12

Agilent IO Libraries Suite, 3, 15, 63

Agilent IO library, 12

Agilent Signal Studio, 166

Agilent Signal Studio Toolkit, 132

Agilent VISA COM Resource Manager 1.0, 33

ARB waveform file downloads

    data requirements, 132

    download utilities, 132, 166

    playing downloaded waveforms, 164

ASCII

    characters, 216

    data, 11

    format, 219

**B**

baseband operation

    status group, 112

Baseband Studio for Waveform Capture and

    Playback, 143

BASIC, 34, 35, 38, 40

BASIC commands

    ABORT, 8

    CLEAR, 10

    ENTER, 11

    LOCAL, 10

    LOCAL LOCKOUT, 9

    OUTPUT, 11

    REMOTE, 9

Berkeley sockets, 18

big endian and little endian (byte order), 134

    changing byte order, 135

    interleaving and byte swapping, 155

big-endian, 200

binary, 222

binary downloads, 216, 217

binary format, 215

bit format

    description, 215

    downloading, 216

bit status monitor, 100

bit values, 99

bits and bytes, 132

block format, 223

byte order

    byte swapping, 135

    changing byte order, 135

    interleaving I/Q data, 155

    little endian and big endian, 134

**C**

C#, 226

C++ programming examples, 169

C/C++, include files, 32

CLEAR, 10

CLS command, 102

coefficients, 219

command prompt, 14, 87

commands

    abort, 8

    clear, 10

    enter, 11

    local, 10

    local lockout, 9

    output, 11

    remote, 9

computer interfaces, 2

condition registers, description, 106

---

# Index

connection expert, 3  
connection wizard, 3  
control data bits, 215  
controller, 8  
creating and downloading waveform files, 131  
creating waveform data, 152  
    saving to a text file for review, 154  
csc.exe, 225

## D

DAC input values, 135  
data  
    directories, 216  
data encryption, 147  
data format  
    E443xB signal generator, 167  
data questionable filters  
    calibration transition, 128  
    frequency transition, 122  
    modulation transition, 125  
    power transition, 119  
    transition, 116  
data questionable groups  
    calibration status, 127  
    frequency status, 121  
    modulation status, 124  
    power status, 118  
    status, 115  
data questionable registers  
    calibration condition, 128  
    calibration event, 128  
    calibration event enable, 129  
    condition, 116  
    event, 117  
    event enable, 117  
    frequency condition, 122  
    frequency event, 122  
    frequency event enable, 123  
    modulation condition, 125  
    modulation event, 125  
    modulation event enable, 126  
    power condition, 119  
    power event, 119  
    power event enable, 120  
data requirements, 132, 216, 219, 221

decryption, 147  
DHCP, 13, 14, 27  
directories  
    binary, 216  
    bit, 216  
DNS, 13, 15  
DOS command prompt, 19  
download  
    utilities  
        Agilent Signal Studio Toolkit, 132  
        differences, 166  
        IntuiLink for PSG/ESG Signal Generators, 132  
        PSG/ESG Download Assistant, 132  
    waveform data, 131, 157  
        advanced programming languages, 160  
        commands, 146  
        E443xB signal generator files, 136, 166  
        encrypted files for extraction, 150  
        encrypted files for no extraction, 149  
        ftp procedures, 150  
        memory locations, 147  
        playing waveforms, 164  
        simulation software, 158  
        unencrypted files for extraction, 149  
        unencrypted files for no extraction, 148  
download user flatness, 225  
downloading  
    C++, 169  
    HP Basic, 205  
    MATLAB, 197  
    user files, 215  
    Visual Basic, 202  
dynamic host communication protocol, 13

## E

E443xB files, 166, 187  
    downloading, 168  
    formatting, 136, 167  
    storing, 167  
E443xB programming examples, 205  
edit visa config, 4  
EnableRemote, 9  
enabling SRQ, 101  
encryption, 146, 147  
    downloading for extraction, 150

- downloading for no extraction, 149
- extracting waveform data, 150
- end-of-file indicator, 148
- enter, 11
- errors, 15, 29
- ESE commands, 102
- even number of samples, 141
- event registers, description, 106
- example programs, 169
  - C++, 169
  - E443xB files, 187, 205
  - HP Basic, 205
  - MATLAB, 194
  - Visual Basic, 200
- examples
  - bit directory, downloading to, 216
  - FIR filter coefficients, downloading, 220
  - GPIO programming, 34
  - Java, 87
  - LAN programming, 63
  - pass-thru commands, 60
  - PERL, 63
  - PRAM, 223
  - PRAM downloads, 223
  - requirements, 32
  - RS-232 programming, 89
  - save and recall, 226
  - serial interface, 89
  - Telnet, 22
  - user files as transmitted data, 219
- extract waveform data, 146, 149–150

## F

- files
  - decryption, 147
  - download utilities, 166
  - downloading, 215
  - encryption, 146, 147
  - error messages, 29
  - extraction commands and file paths, 148
  - header information, 140, 147
  - including, 32
  - managing, 215
  - transfer methods, 147
  - transferring, 22

- using, 215
- waveform structure, 140
- filters
  - See also* transition filters
  - negative transition, description, 106
  - positive transition, description, 106
- FIR filter coefficients, downloading, 219
- firmware status, monitoring, 99
- flatness corrections, 235
- FTP
  - LAN, 12
  - using, 22
- ftp, 147
  - commands for downloading and extracting files, 149–150
  - procedures for downloading files, 150
  - web server information, 27
  - web server procedure, 151
- function statements, 8

## G

- Getting Started wizard, 7

## GPIO

- address, 7, 63
- cables, 7
- configuration, 7
- controller, 8
- function statements, 8
- interface card (PC), 5, 6
- interface card, installing, 5
- interface, description, 2
- IO libraries, 6
- listener, 8
- on UNIX, 6
- program examples, 34
- remote operation, 1
- talker, 8
- using, 5
- verifying operation, 7

## H

- hardware status, monitoring, 99
- headers, 32
- hexadecimal data, 200

---

# Index

hostname, [13](#), [63](#)

HP Basic programming examples, [205](#)

HyperTerminal, [25](#)

## I

I/Q data

- creating with advanced programming languages, [152](#)

- encryption, [146](#), [147](#)

- interleaving, [138](#), [155](#)

  - big endian and little endian, [155](#)

  - byte swapping, [155](#)

- memory locations, [145](#), [156](#)

- saving to a text file for review, [154](#)

- scaling, [136](#)

- waveform structure, [141](#)

iabort, [9](#)

ibloc, [10](#)

ibstop, [9](#)

ibwrt, [11](#)

iclear, [11](#)

IEEE, [5](#)

IEEE 488.2 common commands, [102](#)

igpibIlo, [10](#)

include files, [235](#)

input values, DAC, [135](#)

instrument communication, [4](#)

instrument state files, [225](#)

instrument status, monitoring, [95](#)

interactive IO, [15](#)

interactive io, [3](#)

interface card, GPIB (PC), [5](#), [6](#)

interfaces, computer, [2](#)

interleaving, *See* I/Q data, [138](#)

IntuiLink for PSG/ESG Signal Generators, [135](#), [166](#)

IO Config, [4](#), [7](#), [16](#), [63](#)

io config, [3](#)

IO interface, [4](#)

IO libraries, [1](#), [3](#), [6](#), [7](#), [15](#), [16](#), [24](#), [31](#)

IP

- address, [13](#)

- sockets LAN, [18](#)

iremote, [9](#)

## J

Java program example, [87](#)

## L

LAN

- client, [63](#)

- configuring, [13](#)

- DHCP, [13](#)

- DHCP configuration, [14](#)

- end-of-file indicator, [148](#)

- establishing a connection, [158](#), [160](#)

- FTP, [12](#)

- hostname, [13](#)

- interface, description, [2](#)

- IO libraries, [12](#)

- IP address, [13](#)

- Java, [87](#)

- overview, [12](#)

- PERL, [86](#)

- Ping

  - errors, [15](#)

- program examples, [63](#)

- remote operation, [1](#)

- sockets, [12](#), [63](#)

- sockets LAN, [12](#)

- sockets using C, [64](#)

- Telnet, [12](#), [18](#)

- verifying operation, [14](#)

- VXI-11, [12](#), [63](#)

LAN config, [16](#)

lan configuration, [27](#)

LAN services setup, [14](#)

languages, programming, [5](#)

libraries, [3](#)

libraries, IO, [24](#)

list format, [223](#)

list, error messages, [29](#)

listener, [8](#)

little endian and big endian, [134](#)

- changing byte order, [155](#)

- interleaving and byte swapping, [155](#)

local, [10](#)

local echo telnet, [21](#)

local lockout, [9](#)

LSB and MSB, [133](#)

LSB/MSB, [200](#)

## M

manual operation, [9](#)

marker file, [140](#), [147](#)

### MATLAB

download utility, [166](#)

downloading data, [158](#)

programming examples, [194](#)

### memory

allocation, [145](#)

defined, [144](#)

locations, [144](#)

non-volatile (NVWFM), [147](#)

size, [146](#)

volatile (WFM1), [147](#)

MSB and LSB, [133](#)

MS-DOS command prompt, [14](#), [19](#), [22](#)

## N

### National Instruments

GPIO interface card, [6](#)

NI-488.2, [34](#)

NI-488.2 include files, [32](#)

PCI-GPIB interface requirements, [34](#)

VISA, [6](#), [24](#)

negative transition filter, description, [106](#)

net framework, [225](#)

### NI-488.2

EnableRemote, [9](#)

GPIO, [6](#)

iblcr, [11](#)

ibloc, [10](#)

ibrd, [12](#)

ibstop, [9](#)

ibwrt, [11](#)

RS-232, [24](#)

SetRWLS, [10](#)

non-volatile memory, [144](#), [147](#)

memory allocation, [145](#)

NVWFM, [22](#)

## O

OPC commands, [102](#)

OSR oversample ratio, [219](#)

output command, [11](#)

## P

parser, [10](#)

pass-thru commands, [60](#)

pattern RAM, downloading, [221](#)

pc, [200](#)

PCI-GPIB, [34](#)

PERL, [63](#)

phase discontinuity, [142](#)

avoiding, [143](#)

Baseband Studio for Waveform Capture and Playback, [143](#)

samples, [143](#)

phase distortion, [142](#)

### Ping

errors, [15](#)

program, [14](#)

ping responses, [15](#)

polling method (status registers), [100](#)

positive transition filter, description, [106](#)

### PRAM

comparison to user files, [215](#)

data requirements, [222](#)

description note, [216](#)

downloading

block format, [223](#)

list format, [223](#)

### PRAM downloads

modulating and activating the carrier, [224](#)

troubleshooting, [240](#)

pre-compiled headers, [32](#)

### problems

PRAM downloads, [240](#)

user bit/binary downloads, [239](#)

user FIR filter downloads, [240](#)

programming examples, [169](#)

BASIC, [34](#), [35](#), [38](#), [40](#)

C#, [226](#)

C++, [169](#)

E443xB files, [187](#), [205](#)

---

# Index

GPIB, 34  
HP Basic, 205  
Java, 63  
LAN, 63  
MATLAB, 194  
NI-488.2 and C++, 34, 36, 39, 41  
pass-thru commands, 60  
PERL, 63  
requirements, 32  
sockets and C, 63  
using, 31  
VISA and C, 34, 37, 43, 45  
Visual Basic, 200, 202  
VXI-11, 63  
programming languages, 5, 31  
    byte swapping for little endian order, 155  
    creating waveform data, 152  
    downloading waveform data, 157  
PSG/ESG Download Assistant, 166

**Q**  
query waveform data, 223  
queue, error, 29

**R**  
ramp sweep, using pass-thru commands, 60  
recall states, 225  
register system overview, 95  
registers  
    *See also* status registers  
    condition, description, 106  
    data questionable calibration condition, 128  
    data questionable calibration event, 128  
    data questionable calibration event enable, 129  
    data questionable condition, 116  
    data questionable event, 117  
    data questionable event enable, 117  
    data questionable frequency condition, 122  
    data questionable frequency event, 122  
    data questionable frequency event enable, 123  
    data questionable modulation condition, 125  
    data questionable modulation event, 125  
    data questionable modulation event enable, 126  
    data questionable power condition, 119

    data questionable power event, 119  
    data questionable power event enable, 120  
    in status groups (descriptions), 106  
    overall system, 97, 98  
    standard event status, 107  
    standard event status enable, 108  
    standard operation condition, 109, 113  
    standard operation event, 111, 113  
    standard operation event enable, 111, 114  
    status byte, 105  
remote annunciator, 89  
remote function, 9  
remote operation  
    interfaces, 1  
requirements, waveform data, 132  
RS-232  
    address, 89  
    baud rate, 24  
    cable, 24  
    configuration, 24  
    echo, 24  
    format parameters, 26  
    interface, 24  
    interface, description, 3  
    IO libraries, 24  
    overview, 23  
    program examples, 89  
    remote operation, 1  
    settings, baud rate, 89  
    verifying operation, 25

**S**  
samples  
    even number, 141  
    waveform, 141  
sampling period, 219  
save and recall, 225  
scaling I/Q data, 136  
SCPI, 27, 64  
    error queue, 29  
    register model, 96  
    service, 18  
SCPI commands  
    command line structure, 147  
    download E443xB files, 168

- encrypted files, [149](#), [150](#)
- end-of-file indicator, [148](#)
- extraction, [146](#), [148](#), [149](#), [150](#)
- no extraction, [148](#), [149](#)
- playing downloaded waveforms, [164](#)
- PRAM downloads (modulating and activating the carrier), [224](#)
- sockets LAN, [18](#)
- status registers, [102](#)
- unencrypted files, [148](#), [149](#)
- SCPI file transfer methods, [147](#)
- securewave directory, [147](#)
  - downloading encrypted files, [150](#)
  - extracting waveform data, [150](#)
- serial interface
  - description, [3](#)
  - remote operation, [1](#)
- service request, [101](#)
- service request enable register, [105](#)
- service request method
  - status registers, [101](#)
- service request method, using, [101](#)
- SetRWLS, [10](#)
- SICL
  - GPIB, [6](#)
  - iabort, [9](#)
  - iclear, [11](#)
  - igpiblo, [10](#)
  - iprintf, [11](#)
  - iremote, [9](#)
  - iscanf, [12](#)
  - LAN, [12](#)
  - library, [34](#)
  - RS-232, [24](#)
- signal generator
  - monitoring status, [95](#)
- Signal Studio Toolkit, [132](#), [166](#)
- simulation software, [158](#)
- sockets, [64](#)
  - example, [66](#)
  - examples, [63](#)
  - Java, [87](#)
  - LAN, [18](#), [63](#), [64](#)
  - PERL, [86](#)
  - UNIX, [64](#)
  - Windows, [65](#)
- sockets SCPI, [64](#)
- software libraries, IO, [3](#)
- SRE, [102](#), [105](#)
- SRQ method (status registers), [101](#)
- standard event
  - status enable register, [108](#)
  - status group, [106](#)
  - status register, [107](#)
- standard operation
  - condition register, [109](#), [113](#)
  - event enable register, [111](#), [114](#)
  - event register, [111](#), [113](#)
  - status group, [108](#)
  - transition filters, [110](#), [113](#)
- state files, [225](#)
- status byte
  - group, [104](#)
  - overall register system, [97](#), [98](#)
  - register, [105](#)
  - register bits, [105](#)
- status groups
  - baseband operation, [112](#)
  - data questionable, [115](#)
  - data questionable calibration, [127](#)
  - data questionable frequency, [121](#)
  - data questionable modulation, [124](#)
  - data questionable power, [118](#)
  - registers, [106](#)
  - standard event, [106](#)
  - standard operation, [108](#)
  - status byte, [104](#)
- status registers
  - See also* registers
  - accessing information, [99](#)
  - bit values, [99](#)
  - hierarchy, [96](#)
  - in status groups, [106](#)
  - monitor, [100](#)
  - overall system, [97](#), [98](#)
  - programming, [95](#)
  - SCPI commands, [102](#)
  - SCPI model, [96](#)
  - setting and querying, [102](#)
  - standard event, [107](#)

---

# Index

- standard event status enable, [108](#)
- system overview, [95](#)
- using, [99](#)

STB command, [102](#)

symbol rate, [219](#)

system requirements

- C/C++ examples, [32](#)

## T

talker, [8](#)

taps, [219](#)

TCP/IP, [27](#)

TCPIP, [4](#), [16](#), [18](#), [63](#)

Telnet

- DOS command prompt, [19](#)
- example, [22](#)
- PC, [19](#)
- sockets LAN, [18](#)
- UNIX, [21](#)
- using, [18](#)
- Windows 2000, [20](#)

Toolkit, Signal Studio, [132](#), [166](#)

transition filters

- See also* filters
- data questionable, [116](#)
- data questionable calibration, [128](#)
- data questionable frequency, [122](#)
- data questionable modulation, [125](#)
- data questionable power, [119](#)
- description, [106](#)
- standard operation, [110](#), [113](#)

troubleshooting

- ping response errors, [15](#)
- PRAM downloads, [240](#)
- RS-232, [26](#)
- user bit/binary downloads, [239](#)
- user file downloads, [239](#)
- user FIR filter downloads, [240](#)

## U

unencrypted files

- downloading for extraction, [149](#)
- downloading for no extraction, [148](#)

user file creation and download, [215](#)

user files, downloading, [215](#)

user flatness, [225](#), [235](#)

## V

viPrintf, [235](#)

VISA

- description, [6](#)
- include files, [32](#)
- LAN, [12](#)
- library, [12](#), [34](#), [200](#)
- RS-232, [24](#)
- viClear, [10](#)
- viPrintf, [11](#)
- viScanf, [11](#)
- viTerminate, [8](#)
- VXI-11, [63](#)

VISA Assistant, [4](#), [16](#)

VISA assistant, [15](#)

Visa Assistant, [7](#)

VISA COM IO Library, [33](#)

VISA LAN client, [16](#)

visa.h, [235](#)

Visual Basic

- IDE, [33](#)
- references, [33](#)

Visual Basic programming examples, [200](#)

viWrite, [235](#)

volatile memory, [144](#), [147](#)

- memory allocation, [145](#)

VXI-11, [12](#), [17](#), [63](#)

## W

waveform data

- 2's complement data format, [138](#)
- bits and bytes, [132](#)
- byte order, [135](#)
- byte swapping, [135](#)
- commands for downloading and extracting, [146–152](#)
- creating, [152](#)
- DAC input values, [135](#)
- data requirements, [132](#)
- encryption, [146–150](#)
- explained, [132](#)



- I and Q interleaving, [138](#)
- LSB and MSB, [133](#)
- saving to a text file for review, [154](#)
- waveform downloads
  - memory, [144](#)
    - allocation, [145](#)
    - size, [146](#)
    - volatile and non-volatile, [144](#)
  - samples, [141](#)
  - structure, [141](#)
  - troubleshooting files, [214](#)
  - using advanced programming languages, [160](#)
  - using download utilities, [166](#)
  - using HP BASIC, [205–212](#)
  - using simulation software, [158](#)
  - with Visual Basic 6.0, [202](#)
- waveform generation
  - using C++, [169](#)
  - using HP Basic, [205](#)
  - using MATLAB, [194](#)
  - using Visual Basic 6.0, [200](#)
- web server, [27](#)
- WFM1, [22](#)
- Windows 2000, [20](#)
- Windows NT, [3](#), [7](#)
- Winsock, [18](#)
- WriteIEEEBlock, [202](#)

