Programming Guide

Agilent Technologies
E4428C/38C ESG Signal Generators

This guide applies to the following signal generator models:

E4428C ESG Analog Signal Generator

E4438C ESG 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/esg

Agilent Technologies

Manufacturing Part Number: E4400-90505

Printed in USA

August 2005

Notice

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**.

1 Getting Started

This chapter provides the following major sections:

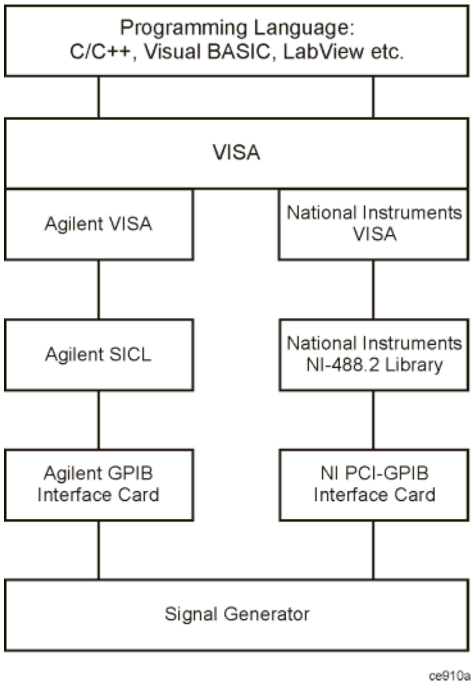
Introduction to Remote Operation

ESG 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 I/O library and programming language, can be used to remotely control your signal generator. Figure 1-1 uses the GPIB as an example of the relationships between the interface, I/O libraries, programming language, and signal generator.

Figure 1-1 Software / Hardware Layers


Programming Language:
C/C++, Visual BASIC, LabView etc.
VISA
Agilent VISA
National Instruments VISA
Agilent SICL
National Instruments NI-488.2 Library
Agilent GPIB Interface Card
NI PCI-GPIB Interface Card
Signal Generator
ce910a

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 4.

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 different 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 I/O library functions to remotely control the signal generator. Common programming languages include:

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

Java is a U.S. trademark of Sun Microsystems, Inc.
Visual Basic is a registered trademark of Microsoft Corporation

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, 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 some of the interface cards available. See the Agilent website, *www.agilent.com* for details on GPIB interface cards that are available.

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

Interface Card	Operating System	I/O Library	Languages	Backplane/BUS	Max I/O (kB/sec)	Buffering
Agilent 82341C for ISA bus computers	Windows 95/98/NT/ 2000®	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

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

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

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

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	I/O Library	Languages	Backplane/BUS	Max I/O (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 I/O Libraries for GPIB

The I/O libraries are included with your GPIB interface card. These libraries can also be downloaded from the National Instruments website or the Agilent website. Refer to "I/O Libraries" on page 3 for information on I/O libraries. The following is a discussion on these libraries.

VISA	VISA is an I/O library used to develop I/O 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 I/O libraries; NI-488.2 and SICL respectively. It is best to use the Agilent VISA library with the Agilent GPIB interface card or NI-VISA with the NI PCI-GPIB interface card.
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 LAN** > **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 **Enter**.
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

NI-VISA is a registered trademark of National Instruments Corporation

4. Verifying GPIB Functionality

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

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 I/O 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 documentation for more information.

Microsoft is a registered trademark of Microsoft Corporation.

Abort Function

The Agilent BASIC function `ABORT` and the other listed I/O 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.

Table 1-5

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

Remote Function

The Agilent BASIC function `REMOTE` and the other listed I/O 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.

Table 1-6

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

Agilent BASIC	The `REMOTE 719` function disables the front panel operation of all keys with the exception of the **Local** 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 `id` parameter, into remote mode and disables the front panel keys. Pressing the **Local** key on the signal generator front panel restores manual operation. The parameter id is the session identifier.

Local Lockout Function

The Agilent BASIC function `LOCAL LOCKOUT` and the other listed I/O 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.

Table 1-7

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

Agilent BASIC

The `LOCAL LOCKOUT` function disables all front-panel signal generator keys. Return to local control can occur only with a hard on/off, when the `LOCAL` command is sent or if the **Preset** 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 `id` parameter, into remote mode with local lockout. The parameter `id` 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.

Table 1-8

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

Agilent BASIC	The `LOCAL 719` 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 `ud` parameter in the parameter list is the interface or device descriptor.
SICL	The Agilent SICL function puts the signal generator into Local operation; enabling front panel key operation. The `id` parameter identifies the session.

Clear Function

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

Table 1-9

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

Agilent BASIC	The `CLEAR 719` 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 viClear 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 I/O buffers. The `id` parameter identifies the session.

Output Function

The Agilent BASIC I/O function `OUTPUT` and the other listed I/O library functions put the signal generator into a listen mode and prepare it to receive ASCII data, typically SCPI commands.

Table 1-10

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 I/O libraries use similar functions to read data from the signal generator.

Table 1-11

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, www.ieee.org, for details on these standards.

The signal generator supports the following LAN interface protocols:

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

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

1. Selecting I/O Libraries for LAN

The TELNET and FTP protocols do not require I/O libraries to be installed on your computer. However, to write programs to control your signal generator, 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 VISA	VISA is an I/O library used to develop I/O 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.

2. Setting Up the LAN Interface

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

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

Manual Configuration

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

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

3. Use the labeled text softkeys, or numeric keypad, or both to enter the desired hostname.

   To erase the current hostname, press **Editing Keys** > **Clear Text**.
4. Press **Enter**.
5. Press **LAN Config Manual DHCP** to **Manual**.
6. 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, you must also enter the subnet mask and default gateway. See your system administrator to obtain the appropriate values.

7. 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

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

NOTE If the DHCP server uses dynamic DNS to link the hostname with the assigned IP address, the hostname may be used in place of the IP address. Otherwise, the hostname is not usable and you may skip steps 2 through 4.

2. Press **Hostname**.

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

3. Use the labeled text softkeys, or numeric keypad, or both to enter the desired hostname.

   To erase the current hostname, press **Editing Keys** > **Clear Text**.

4. Press **Enter**.
5. Press **LAN Config Manual DHCP** to **DHCP**.
6. Press the **Proceed With Reconfiguration** softkey and then the **Confirm Change (Instrument will Reboot)** softkey.

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

3. 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 1-12.

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:

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

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

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

UNIX is a registered trademark of the Open Group
MS-DOS is a registered trademark of Microsoft Corporation

Table 1-12 Ping Responses

Normal Response for UNIX	A normal response to the ping command will be a total of 9 or 10 packets received with a minimal average round-trip time. The minimal average will be different from network to network. LAN traffic will cause the round-trip time to vary widely.
Normal Response for DOS or Windows	A normal response to the ping command will be a total of 9 or 10 packets received if 10 echo requests were specified.
Error Messages	If error messages appear, then check the command syntax before continuing with troubleshooting. If the syntax is correct, resolve the error messages using your network documentation or by consulting your network administrator. If an unknown host error message appears, try using the IP address instead of the hostname. Also, verify that the host name and IP address for the signal generator have been registered by your IT administrator. Check that the hostname and IP address are correctly entered in the node names database. To do this, enter `the nslookup` <`hostname`> command from the command prompt.
No Response	If there is no response from a ping, no packets were received. Check that the typed address or hostname matches the IP address or hostname assigned to the signal generator in the System **Utility** > **GPIB/RS-232 LAN** > **LAN Setup** menu. Ping each node along the route between your workstation and the signal generator, starting with your workstation. If a node doesn’t respond, contact your IT administrator. If the signal generator still does not respond to ping, you should suspect a hardware problem.
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 VXI-11

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

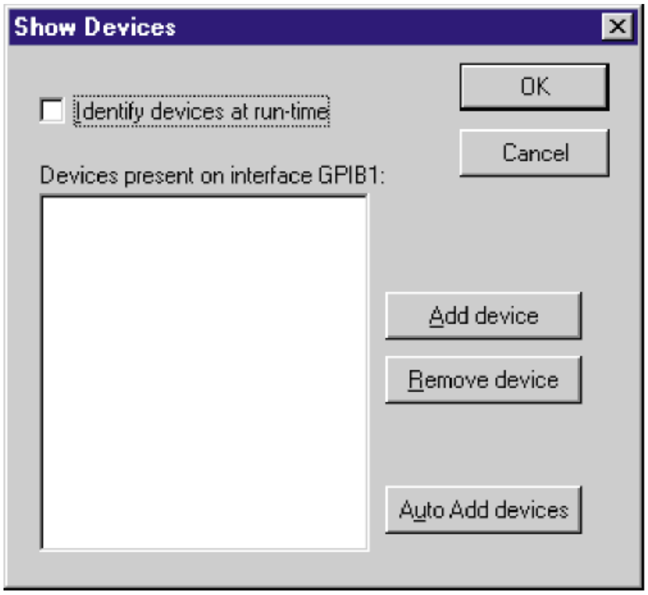
Configuring for VXI-11

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

Use the I/O Config program to configure the LAN client. Once the computer is configured for a LAN client, you can use the VXI-11 protocol and the VISA library to send SCPI commands to the signal generator over the LAN interface. Example programs for this protocol are included in “LAN Programming Examples” on page 80 of this programming guide.

NOTE For Agilent I/O library version J.01.0100, the “Identify devices at run-time” check box must be unchecked. Refer to Figure 1-2.

Figure 1-2 **Show Devices Form**


Show Devices
OK
Cancel
Identify devices at run-time
Devices present on interface GPIB1:
Add device
Remove device
Auto Add devices

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 ®operating system uses a command prompt style interface for the Telnet client. Refer to the Figure 1-5 on page 25 for an example of this interface.

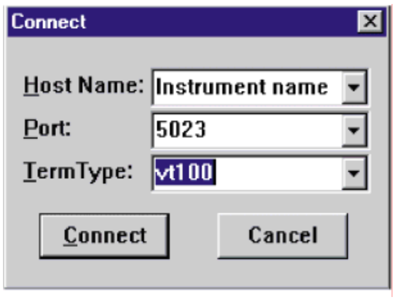
Using Telnet and MS-DOS Command Prompt

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

Windows 2000 is a registered trademark of Microsoft Corporation.

Figure 1-3



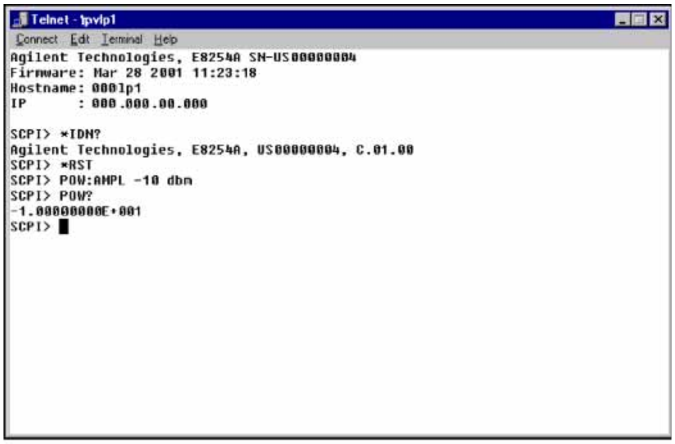
Host Name: Instrument name
Port: 5023
TermType: vt100

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 24.
7. To signal device clear, press `Ctrl-C` on your keyboard.
8. Select **Exit** from the **Connect** menu and type `exit` at the command prompt to end the Telnet session.

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

1. On your PC, click **Start** > **Run**.
2. Type `telnet` then click the **OK** button. The Telnet connection screen will be displayed.
3. Click on the **Connect** menu then select **Remote System**. A connection form is displayed. See Figure 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 24.
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

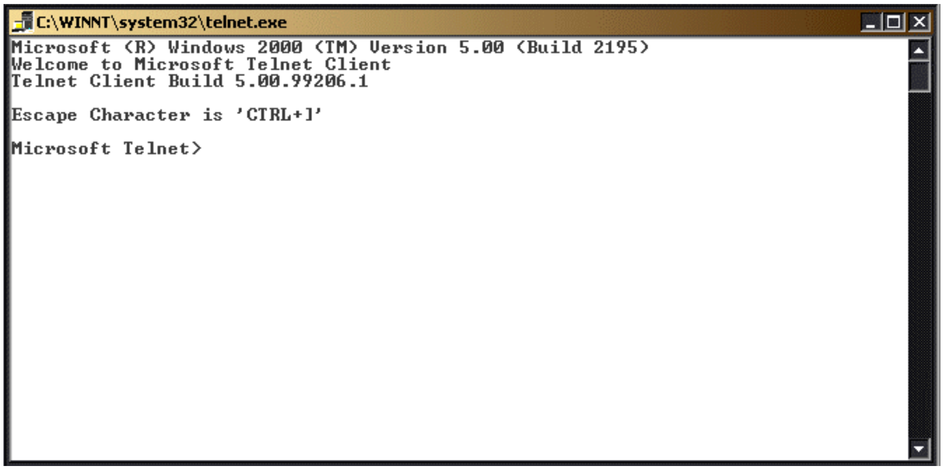


```
Agilent Technologies, E8254A SN-US00000004
Firmware: Mar 28 2001 11:23:18
Hostname: 0001p1
IP       : 000.000.00.000

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 25.
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 24.
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



```
Microsoft (R) Windows 2000 (TM) Version 5.00 (Build 2195)
Welcome to Microsoft Telnet Client
Telnet Client Build 5.00.99206.1

Escape Character is 'CTRL+]'

Microsoft Telnet>
```

The Standard UNIX Telnet Command

Synopsis

`telnet` [host [port]]

Description

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

Options and Parameters

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

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

Unix Telnet Example

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

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

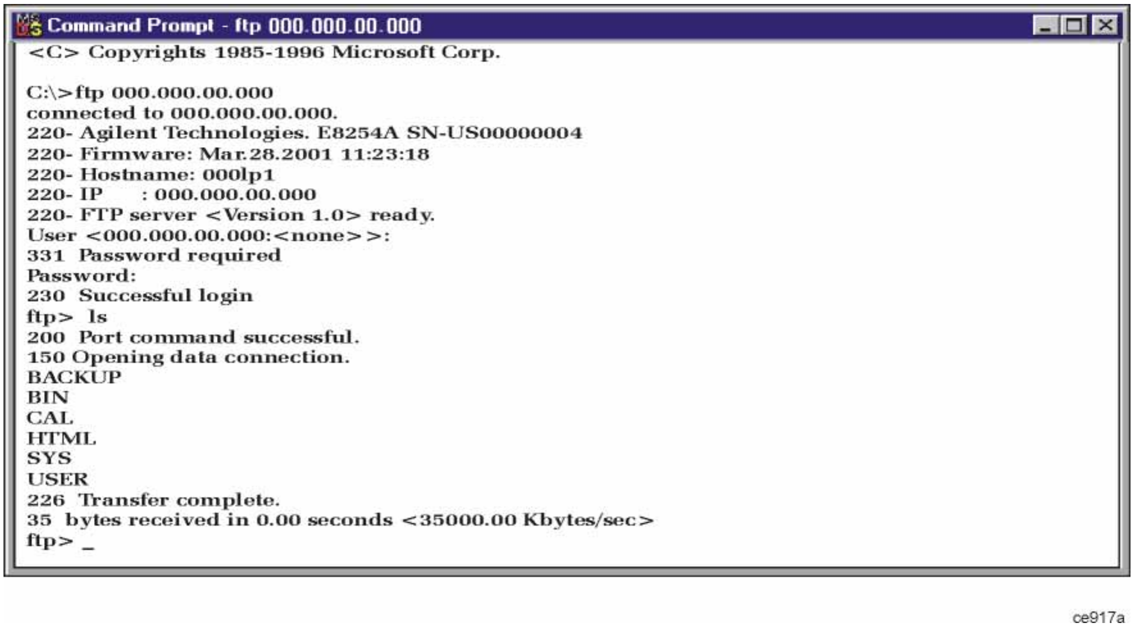
```
$ telnet myinstrument 5023

Trying....

Connected to signal generator

Escape character is ‘^]’.

Agilent Technologies, E44xx SN-US00000001

Firmware:

Hostname: your instrument

IP :xxx.xx.xxx.xxx

SCPI>
```

Using FTP

FTP allows users to transfer files between the signal generator and any computer connected to the LAN. For example, you can use FTP to download instrument screen images to a computer. When logged onto the signal generator with the FTP command, the signal generator’s file structure can be accessed. Figure 1-6 shows the FTP interface and lists the directories in the signal generator’s user level directory.

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

Figure 1-6 FTP Screen



```
<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> _
```

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

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

   `ftp` < `IP address` > or < `hostname` >
3. At the user name prompt, press enter.
4. At the password prompt, press enter.

   You are now in the signal generator’s user directory. Typing help at the command prompt will show you the FTP commands that are available on your system.
5. Type `quit` or `bye` to end your FTP session.
6. Type `exit` to end the command prompt session.

Using RS-232

The RS-232 serial interface can be used to communicate with the signal generator. The RS-232 connection is standard on most PCs and can be connected to the signal generator’s rear-panel connector using the cable described in Table 1-13 on page 29. 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 I/O Libraries for RS-232

The I/O libraries can be downloaded from the National Instrument website, www.ni.com, or Agilent’s website, www.agilent.com. The following is a discussion on these libraries.

Agilent BASIC	The Agilent BASIC language has an extensive I/O 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 I/O library used to develop I/O 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 I/O libraries used to communicate over the RS-232; NI-488.2 and SICL respectively.
NI-488.2	NI-488.2 I/O 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 AXILLARY INTERFACE connector. Refer to Table 1-13 for RS-232 cable information.

Table 1-13 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 and perform the following steps:

1. On the PC click **Start** > **Programs** > **Accessories** > **HyperTerminal**.
2. Select **HyperTerminal**.
3. Enter a name for the session in the text box and select an icon.
4. Select COM1 (COM2 can be used if COM1 is unavailable).
5. In the COM1 (or COM2, if selected) properties, set the following parameters:
   - Bits per second: 9600 must match signal generator's baud rate; On the signal generator Select **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 programmatically or download files to the signal generator, you *must* enable RTS-CTS (hardware) flow control on the controller. Note that only the RTS line is currently used.

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

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

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

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

Character Format Parameters

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

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

If You Have Problems

1. Verify that the baud rate, parity, and stop bits are the same for the computer and signal generator.
2. Verify that the RS-232 cable is identical to the cable specified in Table 1-13.
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.

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 33, 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 User’s Guide and “Waveform Memory” on page 184 and for FTP, see “Using FTP” on page 26 and “FTP Procedures” on page 191.

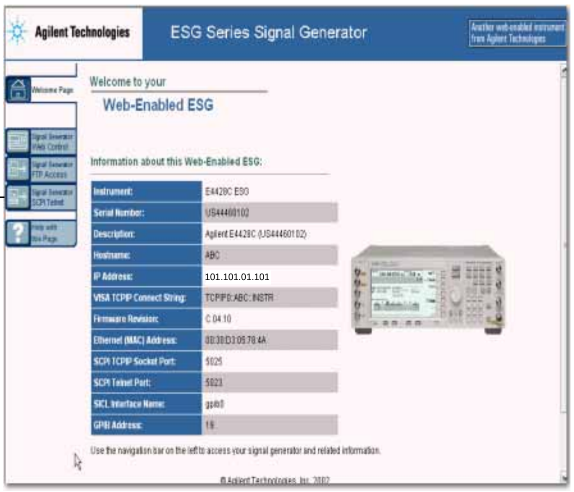
The Web Server service is compatible with the latest version of the Microsoft© Internet Explorer web browser.

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.


Agilent Technologies
ESG Series Signal Generator
Another web-enabled instrument from Agilent Technologies
Welcome Page
Signal Generator Web Control
Signal Generator FTP Access
Signal Generator SCPI Telnet
Help with this Page

Welcome to your
Web-Enabled ESG

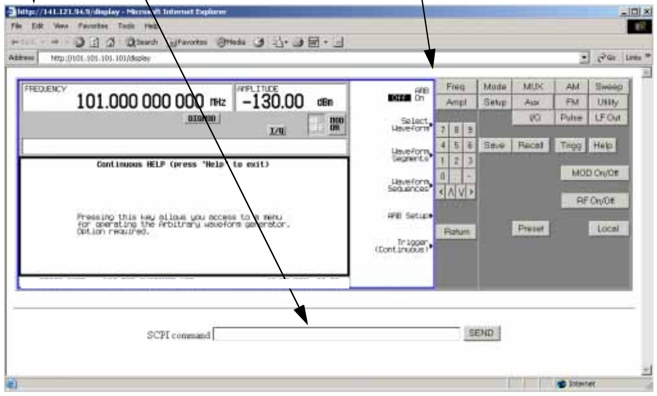
Information about this Web-Enabled ESG:

Instrument:	E4428C ESG
Serial Number:	US44460102
Description:	Agilent E4428C (US44460102)
Hostname:	ABC
IP Address:	101.101.01.101
VISA TCPIP Connect String:	TCPIP0::ABC::INSTR
Firmware Revision:	C.04.10
Ethernet (MAC) Address:	00:30:D3:05:78:4A
SCPI TCPIP Socket Port:	5025
SCPI Telnet Port:	5023
SICL Interface Name:	gpib0
GPIB Address:	19

Use the navigation bar on the left to access your signal generator and related information.

© Agilent Technologies, Inc. 2002

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.

Microsoft is a registered trademark of Microsoft Corp.

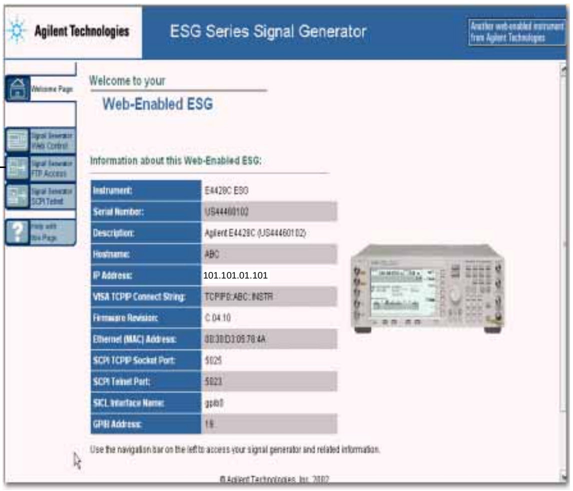
3. In the web browser address field, enter the signal generator’s IP (internet protocol) address. For example, *http://101.101.01.101* (where *101.101.01.101* is the signal generator’s IP address).

   The IP address can change depending on the LAN configuration (see “Using LAN” on page 16).

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.


Agilent Technologies
ESG Series Signal Generator
Another web-enabled instrument from Agilent Technologies
Welcome Page
Signal Generator Web Control
Signal Generator FTP Access
Signal Generator SCPI Telnet
Help with this Page

Welcome to your
Web-Enabled ESG

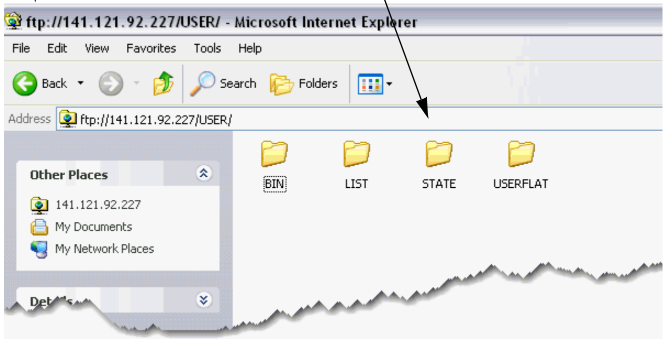
Information about this Web-Enabled ESG:

Instrument:	E4428C ESG
Serial Number:	US44460102
Description:	Agilent E4428C (US44460102)
Hostname:	ABC
IP Address:	101.101.01.101
VISA TCPIP Connect String:	TCPIP0::ABC::INSTR
Firmware Revision:	C.04.10
Ethernet (MAC) Address:	00:30:D3:05:78:4A
SCPI TCPIP Socket Port:	5025
SCPI Telnet Port:	5023
SICL Interface Name:	gpib0
GPIB Address:	19

Use the navigation bar on the left to access your signal generator and related information.

© Agilent Technologies, Inc. 2002

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 *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 `SYSTem:ERRor[:NEXT]?`
Clearing the Queue	Power up Send a `*CLS` command Read last item in the queue
Unresolved Errors	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"`

Errors that must be resolved. For example, unlock.

Error Message File

A complete list of error messages is provided in the file *errormesages.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

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.

The example programs are also available on the ESG Documentation CD-ROM, allowing 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 personal computer (PC) with the following configuration:

- Pentium® processor

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

- Windows NT 4.0 operating system
- C/C++ programming language with the Microsoft Visual C++ 6.0 IDE
- National Instruments PCI- GPIB interface card or Agilent GPIB interface card
- National Instruments VISA Library or Agilent VISA library
- COM1 or COM2 serial port available
- LAN interface card

The 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.

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

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 or National Instrument's 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 the VXI-11 SCPI service.

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 ESG Documentation CD-ROM, enabling you to cut and paste the examples into your project.

Running C# Programming Examples

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.

Visual Basic is a registered trademark of Microsoft corporation

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 ESG 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.
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

Before Using the Examples

If the Agilent GPIB interface card is used, then the Agilent VISA library should be installed along with Agilent SICL. If the National Instruments PCI-GPIB interface card is used, the NI-VISA library along with the NI-488.2 library should be installed. Refer to “2. Selecting I/O Libraries for GPIB” on page 9 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 simple program causes the signal generator to perform an instrument reset. The SCPI command `*RST` places the signal generator into a pre-defined state and the remote annunciator (R) appears on the front panel display.

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

```
10    !******************************************************************************
20    !
30    !  PROGRAM NAME:         basicex1.txt
40    !
50    !  PROGRAM DESCRIPTION:  This program verifies that the GPIB connections and
60    !                        interface are functional.
70    !
80    !  Connect a controller to the signal generator using a GPIB cable.
90    !
100   !
110   !  CLEAR and RESET the controller and type in the following commands and then
120   !  RUN the program:
130   !
140   !******************************************************************************
150   !
160   Sig_gen=719    ! Declares a variable to hold the signal generator's address
170   LOCAL Sig_gen  ! Places the signal generator into Local mode
180   CLEAR Sig_gen  ! Clears any pending data I/O and resets the parser
190   REMOTE 719     ! Puts the signal generator into remote mode
200   CLEAR SCREEN   ! Clears the controllers display
210   REMOTE 719
220   OUTPUT Sig_gen;"*RST"  ! Places the signal generator into a defined state
230   PRINT "The signal generator should now be in REMOTE."
240   PRINT
250   PRINT "Verify that the remote [R] annunciator is on.  Press the `Local' key, "
260   PRINT "on the front panel to return the signal generator to local control."
270   PRINT
```

```
280   PRINT "Press RUN to start again."
290   END   ! Program ends
```

Interface Check Using NI-488.2 and C++

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

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

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

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

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

int main(void)

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

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

  return 0;

}
```

Interface Check using VISA and C

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

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

```
10    !************************************************************************
20    !
30    !  PROGRAM NAME:         basicex2.txt
```

```
40    !
50    !  PROGRAM DESCRIPTION:  In REMOTE mode, access to the signal generators
60    !                        functional front panel keys are disabled except for
70    !                        the Local and Contrast keys.  The LOCAL LOCKOUT
80    !                        command will disable the Local key.
90    !                        The LOCAL command, executed from the controller, is then
100   !                        the only way to return the signal generator to front panel,
110   !                        Local, control.
120   !*************************************************************************
130   Sig_gen=719       ! Declares a variable to hold signal generator address
140   CLEAR Sig_gen     ! Resets signal generator parser and clears any output
150   LOCAL Sig_gen     ! Places the signal generator in local mode
160   REMOTE Sig_gen    ! Places the signal generator in remote mode
170   CLEAR SCREEN      ! Clears the controllers display
180   OUTPUT Sig_gen;"*RST"       ! Places the signal generator in a defined state
190   ! The following print statements are user prompts
200   PRINT "The signal generator should now be in remote."
210   PRINT "Verify that the 'R' and 'L' annunciators are visable"
220   PRINT ".......... Press Continue"
230   PAUSE
240   LOCAL LOCKOUT 7   ! Puts the signal generator in LOCAL LOCKOUT mode
250   PRINT             ! Prints user prompt messages
260   PRINT "Signal generator should now be in LOCAL LOCKOUT mode."
270   PRINT
280   PRINT "Verify that all keys including `Local' (except Contrast keys) have no effect."
290   PRINT
300   PRINT ".......... Press Continue"
310   PAUSE
320   PRINT
330   LOCAL 7           ! Returns signal generator to Local control
340   ! The following print statements are user prompts
350   PRINT "Signal generator should now be in Local mode."
```

```
360   PRINT
370   PRINT "Verify that the signal generator's front-panel keyboard is functional."
380   PRINT
390   PRINT "To re-start this program press RUN."
400   END
```

Local Lockout Using NI-488.2 and C++

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

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

```
10     !*********************************************************************************

20     !

30     !  PROGRAM NAME:         basicex3.txt

40     !

50     !  PROGRAM DESCRIPTION:  In this example, query commands are used with response
```

```
60    !                        data formats.
70    !
80    !  CLEAR and RESET the controller and RUN the following program:
90    !
100   !*****************************************************************************
110   !
120   DIM A$[10],C$[100],D$[10]   ! Declares variables to hold string response data
130   INTEGER B                   ! Declares variable to hold integer response data
140   Sig_gen=719                 ! Declares variable to hold signal generator address
150   LOCAL Sig_gen               ! Puts signal generator in Local mode
160   CLEAR Sig_gen               ! Resets parser and clears any pending output
170   CLEAR SCREEN                ! Clears the controller’s display
180   OUTPUT Sig_gen;"*RST"       ! Puts signal generator into a defined state
190   OUTPUT Sig_gen;"FREQ:CW?"   ! Querys the signal generator CW frequency setting
200   ENTER Sig_gen;F             ! Enter the CW frequency setting
210   ! Print frequency setting to the controller display
220   PRINT "Present source CW frequency is: ";F/1.E+6;"MHz"
230   PRINT
240   OUTPUT Sig_gen;"POW:AMPL?"  ! Querys the signal generator power level
250   ENTER Sig_gen;W             ! Enter the power level
260   ! Print power level to the controller display
270   PRINT "Current power setting is: ";W;"dBM"
280   PRINT
290   OUTPUT Sig_gen;"FREQ:MODE?" ! Querys the signal generator for frequency mode
300   ENTER Sig_gen;A$            ! Enter in the mode: CW, Fixed or List
310   ! Print frequency mode to the controller display
320   PRINT "Source's frequency mode is: ";A$
330   PRINT
340   OUTPUT Sig_gen;"OUTP OFF"   ! Turns signal generator RF state off
350   OUTPUT Sig_gen;"OUTP?"      ! Querys the operating state of the signal generator
360   ENTER Sig_gen;B             ! Enter in the state (0 for off)
370   ! Print the on/off state of the signal generator to the controller display
```

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

Queries Using NI-488.2 and C++

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

The following program example is available on the ESG 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. Launch Microsoft Visual C++ 6.0, add the required files, and enter the following code into your .cpp source file.

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

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

// PROGRAM FILE NAME:visaex3.cpp

//

// PROGRAM DESCRIPTION:This example demonstrates the use of query commands. The signal

// generator can be queried for conditions and instrument states. These commands are of

// the type "*IDN?"; the question mark indicates a query.

//

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


#include <visa.h>

#include "StdAfx.h"

#include <iostream>

#include <conio.h>

#include <stdlib.h>

using namespace std;



void main ()

{

ViSession defaultRM, vi;     // Declares variables of type ViSession

                             // for instrument communication

ViStatus viStatus = 0;       // Declares a variable of type ViStatus

                             // for GPIB verifications

char rdBuffer [256];         // Declares variable to hold string data
```

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

```
viPrintf(vi, "OUTP OFF\n");     // Turns source RF state off

viPrintf(vi, "OUTP?\n");        // Querys the signal generator's RF state

viScanf(vi, "%1i", &num);       // Reads the response (integer value)

                                // Prints the on/off RF state

 if (num > 0 ) {

printf("Source RF state is : on\n");

}else{

printf("Source RF state is : off\n");

}

                                  // Close the sessions

viClose(vi);

viClose(defaultRM);

}
```

Generating a CW Signal Using VISA and C

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

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

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

// PROGRAM FILE NAME:   visaex4.cpp

//

// PROGRAM DESCRIPTION: This example demonstrates query commands. The signal generator

// frequency and power level.

// The RF state of the signal generator is turn on and then the state is queried. The

// response will indicate that the RF state is on. The RF state is then turned off and

// queried. The response should indicate that the RF state is off. The query results are

// printed to the to the display window.

//

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


#include "StdAfx.h"
```

```
#include <visa.h>
#include <iostream>
#include <stdlib.h>
#include <conio.h>

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

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

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

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

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

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

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

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

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

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

void main ()
{
 ViSession defaultRM, vi;           // Declares variables of type ViSession
                                   // for instrument communication
ViStatus viStatus = 0;             // Declares a variable of type ViStatus
                                   // for GPIB verifications
                                   // Initialize VISA session
viStatus=viOpenDefaultRM(&defaultRM);
                               // open session to gpib device at address 19
```

```
viStatus=viOpen(defaultRM, "GPIB::19::INSTR", VI_NULL, VI_NULL, &vi);
if(viStatus){                        // If problems, then prompt user
    printf("Could not open ViSession!\n");
    printf("Check instruments and connections\n");
    printf("\n");
    exit(0);}

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

viPrintf(vi, "*RST\n");              // Resets the signal generator
viPrintf(vi, "FM:SOUR EXT2\n");      // Sets EXT 2 source for FM
viPrintf(vi, "FM:EXT2:COUP AC\n");   // Sets FM path 2 coupling to AC
viPrintf(vi, "FM:DEV 20 kHz\n");     // Sets FM path 2 deviation to 20 kHz
viPrintf(vi, "FREQ 700 MHz\n");      // Sets carrier frequency to 700 MHz
viPrintf(vi, "POW:AMPL -2.5 dBm\n"); // Sets the power level to -2.5 dBm
viPrintf(vi, "FM:STAT ON\n");        // Turns on frequency modulation
viPrintf(vi, "OUTP:STAT ON\n");      // Turns on RF output
                                     // Print user information
printf("Power level : -2.5 dBm\n");
printf("FM state : on\n");
printf("RF output : on\n");
printf("Carrier Frequency : 700 MHZ\n");
printf("Deviation : 20 kHZ\n");
printf("EXT2 and AC coupling are selected\n");
printf("\n");                        // Prints a carrage return
                                     // Close the sessions
viClose(vi);
viClose(defaultRM);
```

```
}
```

Generating an Internal 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. Launch Microsoft Visual C++ 6.0, add the required files, and enter the following code into your .cpp source file.

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

```
//****************************************************************************************
// PROGRAM FILE NAME:visaex6.cpp
//
// PROGRAM DESCRIPION: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 carrage return
                    // Close the sessions
viClose(vi);
viClose(defaultRM);
}
```

Generating a Step-Swept Signal Using VISA and C

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

The following program example is available on the ESG 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);
}
```

Generating a Swept Signal Using VISA and Visual C++

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

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

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

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

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

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

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

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

stat = viPrintf( inst, "*RST\n" );
// set power level for -10dBm
stat = viPrintf(inst, "POW -10DBM\n");
// set the start and stop frequency for the sweep
stat = viPrintf(inst, "FREQ:START 1GHZ\n");
stat = viPrintf(inst, "FREQ:STOP 2GHZ\n");
// setup dwell per point
stat = viPrintf(inst, "SWEEP:DWELL %e\n", dwell);
// setup number of points
stat = viPrintf(inst, "SWEEP:POINTS %d\n", npoints);

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

// set continuous trigger mode off
stat = viPrintf(inst, "INIT:CONT OFF\n");
```

```
    // turn list sweep on
    stat = viPrintf(inst, "FREQ:MODE LIST\n");

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

intCounter = intCounter-1;

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

}
```

Saving and Recalling States Using VISA and C

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

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

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

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

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

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

Reading the Data Questionable Status Register Using VISA and C

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

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

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

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

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

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

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

```
viStatus=viOpen(defaultRM, "GPIB::19::INSTR", VI_NULL, VI_NULL, &vi);

if(viStatus){                   // If problems, then prompt user

printf("Could not open ViSession!\n");

printf("Check instruments and connections\n");

printf("\n");

exit(0);}

printf("\n");

viClear(vi);// Clears the signal generator

// Prints user information

printf("Programming example to demonstrate reading the signal generator's
                         Status Byte\n");

printf("\n");

printf("Manually set up the sig gen for an unleveled output condition:\n");

printf("* Set signal generator output amplitude to +20 dBm\n");

printf("* Set frequency to maximum value\n");

printf("* Turn On signal generator's RF Output\n");

printf("* Check signal generator's display for the UNLEVEL annuniator\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 it’s operation, or detects a failure, then a Service Request can be generated. The computer will respond to the SRQ and, depending on the code, can perform some other operation or notify the user of failures or other conditions.

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

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

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

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


#include "visa.h"
```

```
#include <stdio.h>

#include "StdAfx.h"

#include "windows.h"

#include <conio.h>


#define  MAX_CNT 1024


int sweep=1;  // End of sweeep flag


/* Prototypes */


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


int main ()

{

ViSession defaultRM, vi;// Declares variables of type ViSession

// for instrument communication

ViStatus viStatus = 0;// Declares a variable of type ViStatus

                      // for GPIB verifications

char rdBuffer[MAX_CNT];// Declare a block of memory data


viStatus=viOpenDefaultRM(&defaultRM);// Initialize VISA session

if(viStatus < VI_SUCCESS){// If problems, then prompt user

printf("ERROR initializing VISA... exiting\n");

printf("\n");

return -1;}

                              // Open session to gpib device at address 19

viStatus=viOpen(defaultRM, "GPIB::19::INSTR", VI_NULL, VI_NULL, &vi);

if(viStatus){                 // If problems then prompt user

printf("ERROR: Could not open communication with
                                         instrument\n");

printf("\n");
return -1;}


viClear(vi);                  // Clears the signal generator

viPrintf(vi, "*RST\n");       // Resets signal generator

                              // Print program header and information

printf("** End of Sweep Service Request **\n");

printf("\n");

printf("The signal generator will be set up for a step sweep mode
      operation.\n");

printf("An '*' will be printed while the instrument is sweeping. The end of
                      \n");

printf("sweep will be indicated by an SRQ on the GPIB and the program will
                         end.\n");

printf("\n");

printf("Press Enter to continue\n");

printf("\n");

getch();


viPrintf(vi, "*CLS\n");// Clears signal generator status byte

viPrintf(vi, "STAT:OPER:NTR 8\n");// Sets the Operation Status Group  // Negative
Transition Filter to indicate a  // negative transition in Bit 3 (Sweeping)

// which will set a corresponding event in   // the Operation Event Register. This occurs
 // at the end of a sweep.

viPrintf(vi, "STAT:OPER:PTR 0\n");// Sets the Operation Status Group  // Positive
Transition Filter so that no

// positive transition on Bit 3 affects the  // Operation Event Register. The positive  //
transition occurs at the start of a sweep.

viPrintf(vi, "STAT:OPER:ENAB 8\n");// Enables Operation Status Event Bit 3  // to report
the event to Status Byte  // Register Summary Bit 7.

viPrintf(vi, "*SRE 128\n");// Enables Status Byte Register Summary Bit 7

// The next line of code indicates the  // function to call on an event

viStatus = viInstallHandler(vi, VI_EVENT_SERVICE_REQ, interupt, rdBuffer);

// The next line of code enables the  // detection of an event

viStatus = viEnableEvent(vi, VI_EVENT_SERVICE_REQ, VI_HNDLR, VI_NULL);
```

```
viPrintf(vi, "FREQ:MODE LIST\n");// Sets frequency mode to list

viPrintf(vi, "LIST:TYPE STEP\n");// Sets sweep to step

viPrintf(vi, "LIST:TRIG:SOUR IMM\n");// Immediately trigger the sweep

viPrintf(vi, "LIST:MODE AUTO\n");// Sets mode for the list sweep

viPrintf(vi, "FREQ:STAR 40 MHZ\n"); // Start frequency set to 40 MHz

viPrintf(vi, "FREQ:STOP 900 MHZ\n");// Stop frequency set to 900 MHz

viPrintf(vi, "SWE:POIN 25\n");// Set number of points for the step sweep

viPrintf(vi, "SWE:DWEL .5 S\n");// Allow .5 sec dwell at each point

viPrintf(vi, "INIT:CONT OFF\n");// Set up for single sweep

viPrintf(vi, "TRIG:SOUR IMM\n");// Triggers the sweep

viPrintf(vi, "INIT\n");  // Takes a single sweep

printf("\n");

// While the instrument is sweeping have the

// program busy with printing to the display.

// The Sleep function, defined in the header

// file windows.h, will pause the program

// operation for .5 seconds

while (sweep==1){

printf("*");

Sleep(500);}

printf("\n");

// The following lines of code will stop the

// events and close down the session


viStatus = viDisableEvent(vi, VI_ALL_ENABLED_EVENTS,VI_ALL_MECH);

viStatus = viUninstallHandler(vi, VI_EVENT_SERVICE_REQ, interupt,
                                               rdBuffer);

viStatus = viClose(vi);

viStatus = viClose(defaultRM);

return 0;


}
```

```
// The following function is called when an SRQ event occurs. Code specific to your

// requirements would be entered in the body of the function.


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

{

ViStatus status;

ViUInt16 stb;


 status = viReadSTB(vi, &stb);// Reads the Status Byte

sweep=0;// Sets the flag to stop the '*' printing

printf("\n");// Print user information

printf("An SRQ, indicating end of sweep has occurred\n");

viClose(event);// Closes the event

return VI_SUCCESS;

}
```

LAN Programming Examples

The LAN programming examples in this section demonstrate the use of VXI-11 and Sockets LAN to control the signal generator. For details on using FTP and TELNET refer to “Using FTP” on page 26 and “Using Telnet LAN” on page 22 of this guide.

Before Using the Examples

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

VXI-11 Programing

The signal generator supports the VXI-11 standard for instrument communication over the LAN interface. Agilent IO Libraries support the VXI-11 standard and must be installed on your computer before using the VXI-11 protocol. Refer to “Using VXI-11” on page 20 of this Programming Guide for information on configuring and using the VXI-11 protocol.

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

VXI-11 Programming Using SICL and C

The following program uses the VXI-11 protocol and SICL to control the signal generator. The signal generator is set to a 1 GHz CW frequency and then queried for its ID string. Before running this code, you must set up the interface using the Agilent IO Libraries IO Config utility.

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

```
//****************************************************************************************
//
// PROGRAM NAME:vxisicl.cpp
//
// PROGRAM DESCRIPTION:Sample test program using SICL and the VXI-11 protocol
```

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

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

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

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

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

                // Open SICL instrument handle using VXI-11 protocol

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

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

return 0;
}
```

VXI-11 Programming Using VISA and C

The following program uses the VXI-11 protocol and the VISA library to control the signal generator. The signal generator is set to a 1 GHz CW frequency and queried for its ID string. Before running this code, you must set up the interface using the Agilent IO Libraries IO Config utility.

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

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

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

#define MAX_COUNT 200

int main (void)

{
```

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

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

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


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

```
                                         // Read the power level
status = viRead(instr, (ViBuf)buffer, MAX_COUNT, &retCount);
buffer[retCount]= '\0';                  // Indicate the end of the string
printf("Power level = ");                // Print header to the screen
printf(buffer);                          // Print the queried power level
printf("\n");
status = viClose(instr);                 // Close down the system
status = viClose(defaultRM);
return 0;
}
```

Sockets LAN Programming using C

The program listing shown in "Setting Parameters and Sending Queries Using Sockets and C" on page 88 consists of two files; lanio.c and getopt.c. The lanio.c file has two main functions; `int main()` and an `int main1()`.

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

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

This program is available on the ESG Documentation CD-ROM as lanio.c

Sockets on UNIX

In UNIX, LAN communication via sockets is very similar to reading or writing a file. The only difference is the `openSocket()` routine, which uses a few network library routines to create the TCP/IP network connection. Once this connection is created, the `standard fread()` and `fwrite()` routines are used for network communication. The following steps outline the process:

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

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

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

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

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

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

Sockets on Windows

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

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

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

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

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

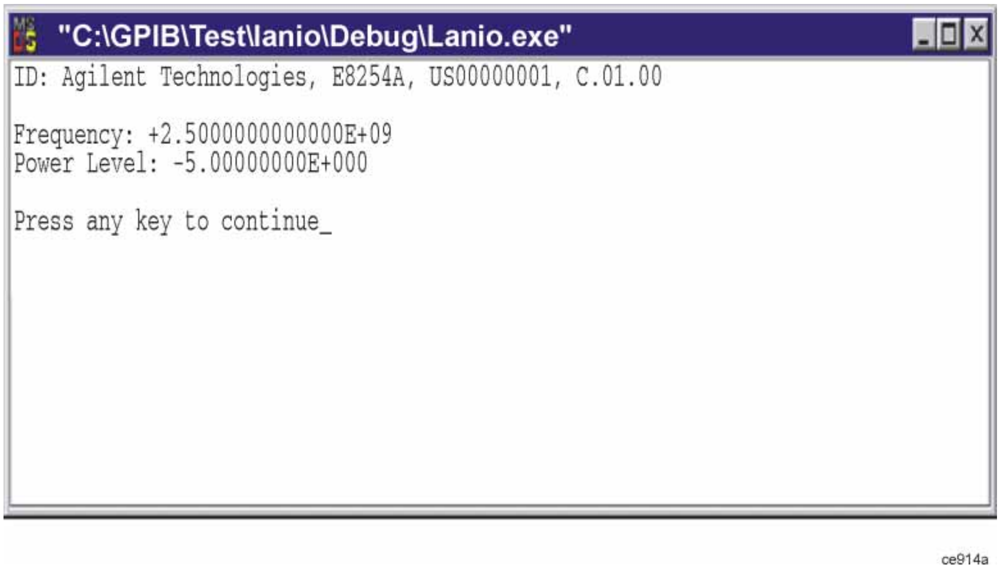
Programming Using main1() Function.

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

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

The program will run and display the results as shown in Figure 2-1.

Figure 2-1 Program Output Screen



```
ID: Agilent Technologies, E8254A, US00000001, C.01.00

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

Press any key to continue_
```

Setting Parameters and Sending Queries Using Sockets and C

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

```
/***************************************************************************
 *  $Header: lanio.c 04/24/01
 *  $Revision: 1.1 $
 *  $Date: 10/24/01
 *  PROGRAM NAME:   lanio.c
 *
 *  $Description:     Functions to talk to an Agilent signal generator
 *                    via TCP/IP.  Uses command-line arguments.
 *
 *                    A TCP/IP connection to port 5025 is established and
 *                    the resultant file descriptor is used to "talk" to the
 *                    instrument using regular socket I/O mechanisms. $
 *
 *
 *
 *  Examples:
 *
 *   Query the signal generator frequency:
 *        lanio xx.xxx.xx.x 'FREQ?'
 *
 *   Query the signal generator power level:
 *        lanio xx.xxx.xx.x  'POW?'
 *
 *   Check for errors (gets one error):
 *        lanio xx.xxx.xx.x  'syst:err?'
 *
 *   Send a list of commands from a file, and number them:
 *        cat scpi_cmds | lanio -n xx.xxx.xx.x
 *
 ****************************************************************************
 *
 *  This program compiles and runs under
 *     - HP-UX 10.20 (UNIX), using HP cc or gcc:
 *           + cc -Aa    -O -o lanio  lanio.c
 *           + gcc -Wall -O -o lanio  lanio.c
 *
 *     - Windows 95, using Microsoft Visual C++ 4.0 Standard Edition
 *     - Windows NT 3.51, using Microsoft Visual C++ 4.0
 *           + Be sure to add  WSOCK32.LIB  to your list of libraries!
 *           + Compile both lanio.c and getopt.c
 *           + Consider re-naming the files to lanio.cpp and getopt.cpp
 *
 *  Considerations:
 *     - On UNIX systems, file I/O can be used on network sockets.
 *       This makes programming very convenient, since routines like
 *       getc(), fgets(), fscanf() and fprintf() can be used.  These
 *       routines typically use the lower level read() and write() calls.
 *
 *     - In the Windows environment, file operations such as read(), write(),
 *       and close() cannot be assumed to work correctly when applied to
 *       sockets.  Instead, the functions send() and recv() MUST be used.
 *****************************************************************************/

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

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

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

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

#ifdef WINSOCK

#include <windows.h>

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

#else                       /* UNIX with BSD sockets */

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

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

   typedef  int SOCKET;

#endif /* WINSOCK */

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

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

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



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



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

    err = WSAStartup(wVersionRequested, &wsaData);

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

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



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



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

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

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

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

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

    return s;
}



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

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

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

    return NO_CMD_ERROR;
}


/*************************************************************************
 * recv_line(): similar to fgets(), but uses recv()
 *************************************************************************/
char * recv_line(SOCKET sock, char * result, int maxLength)
{
```

```
#ifdef WINSOCK
    int cur_length = 0;
    int count;
    char * ptr = result;
    int err = 1;

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

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

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

    }

    *ptr = '\0';

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



/*************************************************************************
 *
 > $Function: queryInstrument$
 *
 * $Description:  send a SCPI command to the instrument, return a response.$
 *
 * $Parameters:  $
 *     (FILE *) . . . . . . . . . file pointer associated with TCP/IP socket.
 *     (const char *command)  . . SCPI command string.
 *     (char *result) . . . . . . where to put the result.
 *     (size_t) maxLength . . . . maximum size of result array in bytes.
 *
 * $Return:  (long) . . . . . . . The number of bytes in result buffer.
 *
 * $Errors:   returns 0 if anything goes wrong. $
 *
 ***************************************************************************/
long queryInstrument(SOCKET sock,
                      const char *command, char *result, size_t maxLength)
{
    long ch;
    char tmp_buf[8];
    long resultBytes = 0;
    int command_err;
    int count;

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



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

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

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

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

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

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

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

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

```
            *result = (char)ch;
            if (recv_line(sock, result+1, maxLength-1) == NULL) return 0;

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

    return resultBytes;
}




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

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

}


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

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

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

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

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

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

    basename = strrchr(argv[0], '/');
    if (basename != NULL)
        basename++ ;
    else
        basename = argv[0];

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

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

            if (optind > argc)
            {
                usage(basename);
                exit(1);
            }
        }
    }
    else
    {
        /* no hostname! */
        usage(basename);
        exit(1);
    }

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

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

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

    if (strlen(command) > 0)
    {
    /********************************************************
    /* if the command has a '?' in it, use queryInstrument. */
    /* otherwise, simply send the command.                  */
    /********************************************************/
```

```
        if ( isQuery(command) )
        {
            long bufBytes;
            bufBytes = queryInstrument(instSock, command,
                                       charBuf, INPUT_BUF_SIZE);
            if (!quiet)
            {
                fwrite(charBuf, bufBytes, 1, stdout);
                fwrite("\n", 1, 1, stdout) ;
                fflush(stdout);
            }
        }
        else
        {
            commandInstrument(instSock, command);
        }
    }
    else
    {
        /* read a line from <stdin> */
        while ( gets(charBuf) != NULL )
        {
            if ( !strlen(charBuf) )
                continue ;

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

            strcat(charBuf, "\n");

            if (!quiet)
            {
```

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

    if ( isQuery(charBuf) )
    {
        long bufBytes;

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

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

```
            if (number) number++;
        }
    }

    if (show_errs) {
        showErrors(instSock);
    }

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

    return 0;
}

/* End of lanio.cpp  *




/***************************************************************************/
/* $Function: main1$                                                       */
/* $Description: Output a series of SCPI commands to the signal generator */
/*              Send query results to stdout. $                            */
/*                                                                         */
/* $Return:  (int) . . . non-zero if an error occurs                       */
/*                                                                         */
/***************************************************************************/
/* Rename this int main1() function to int main(). Re-compile and the     */
/* execute the program                                                     */
```

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


int main1()
{


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



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


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


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


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



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

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

 getopt(3C)                                                      getopt(3C)

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

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

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

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

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

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


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

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

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

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

    optarg = NULL;

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

```
                optind++;

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

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

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

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

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

```
    return(c);
}
```

Sockets LAN Programming Using PERL

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

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

Setting the Power Level and Sending Queries Using PERL

The following program example is available on the ESG Documentation CD-ROM as perl.txt.

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

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

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

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

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

Sockets LAN Programming Using Java

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

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

Generating a CW Signal Using Java

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

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


import java.io.*;
import java.net.*;
class ScpiSockTest
{
    public static void main(String[] args)
    {
       String instrumentName = "xxxxx";         // Put instrument hostname here
try
        {
       Socket t = new Socket(instrumentName,5025);  // Connect to instrument
                                                            // Setup read/write mechanism
            BufferedWriter out =
            new BufferedWriter(
            new OutputStreamWriter(t.getOutputStream()));
            BufferedReader in =
            new BufferedReader(
            new InputStreamReader(t.getInputStream()));
            System.out.println("Setting frequency to 1 GHz...");
            out.write("freq 1GHz\n");               // Sets frequency
            out.flush();
            System.out.println("Waiting for source to settle...");
            out.write("*opc?\n");                   // Waits for completion
            out.flush();
            String opcResponse = in.readLine();
            if (!opcResponse.equals("1"))
             {
              System.err.println("Invalid response to '*OPC?'!");
              System.exit(1);
             }
       System.out.println("Retrieving instrument ID...");
       out.write("*idn?\n");                        // Querys the id string
       out.flush();
       String idnResponse = in.readLine();          // Reads the id string
                                                    // Prints the id string
       System.out.println("Instrument ID: " + idnResponse);
       }
       catch (IOException e)
       {
       System.out.println("Error" + e);
     }
   }
}
```

RS-232 Programming Examples

- “Interface Check Using Agilent BASIC” on page 118
- “Interface Check Using VISA and C” on page 119
- “Queries Using Agilent BASIC” on page 121
- “Queries Using VISA and C” on page 122

Before Using the Examples

On the signal generator select the following settings:

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

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 28 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 31 for more help.

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

```
//****************************************************************************
//
// PROGRAM NAME:       rs232ex2.cpp
//
// PROGRAM DESCRIPTION: This code example uses the RS-232 serial interface to control
// the signal generator.
```

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



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

#define MAX_COUNT 200

int main (void)
{

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

status = viOpenDefaultRM(&defaultRM);// Initializes the system
// Open communication with Serial Port 2
status = viOpen(defaultRM, "ASRL2::INSTR", VI_NULL, VI_NULL, &instr);
if(status){// If problems, then prompt user
printf("Could not open ViSession!\n");
printf("Check instruments and connections\n");
printf("\n");
exit(0);}
                                        // Set timeout for 5 seconds
viSetAttribute(instr, VI_ATTR_TMO_VALUE, 5000);
// Asks for sig gen ID string
 status = viWrite(instr, (ViBuf)"*IDN?\n", 6, &retCount);

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

3 Programming the Status Register System

This chapter provides the following major sections:

Overview

NOTE Some of the status bits and register groups do not apply to the E4428C:

- Standard Operation Condition Register bits (see Table 3-5 on page 142)
- Baseband Operation Status Group
- Data Questionable Condition Register bits (see Table 3-7 on page 148)
- Data Questionable Power Condition Register bit (see Table 3-8 on page 152)
- Data Questionable Frequency Condition Register bit (see Table 3-9 on page 155)
- Data Questionable Calibration Condition Register bit (see Table 3-11 on page 161)
- Data Questionable Bert Status Group

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

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

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

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

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

The lower level status registers are grouped according to their functionality. For example, the Data Quest. Frequency Status Group consists of five registers. This chapter may refer to a group as a register so that the cumbersome correct description is avoided. For example, the Standard Operation Status Group's Condition Register can be referred to as the Standard Operation Status register. Refer to "Status Groups" on page 138 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 commands.

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

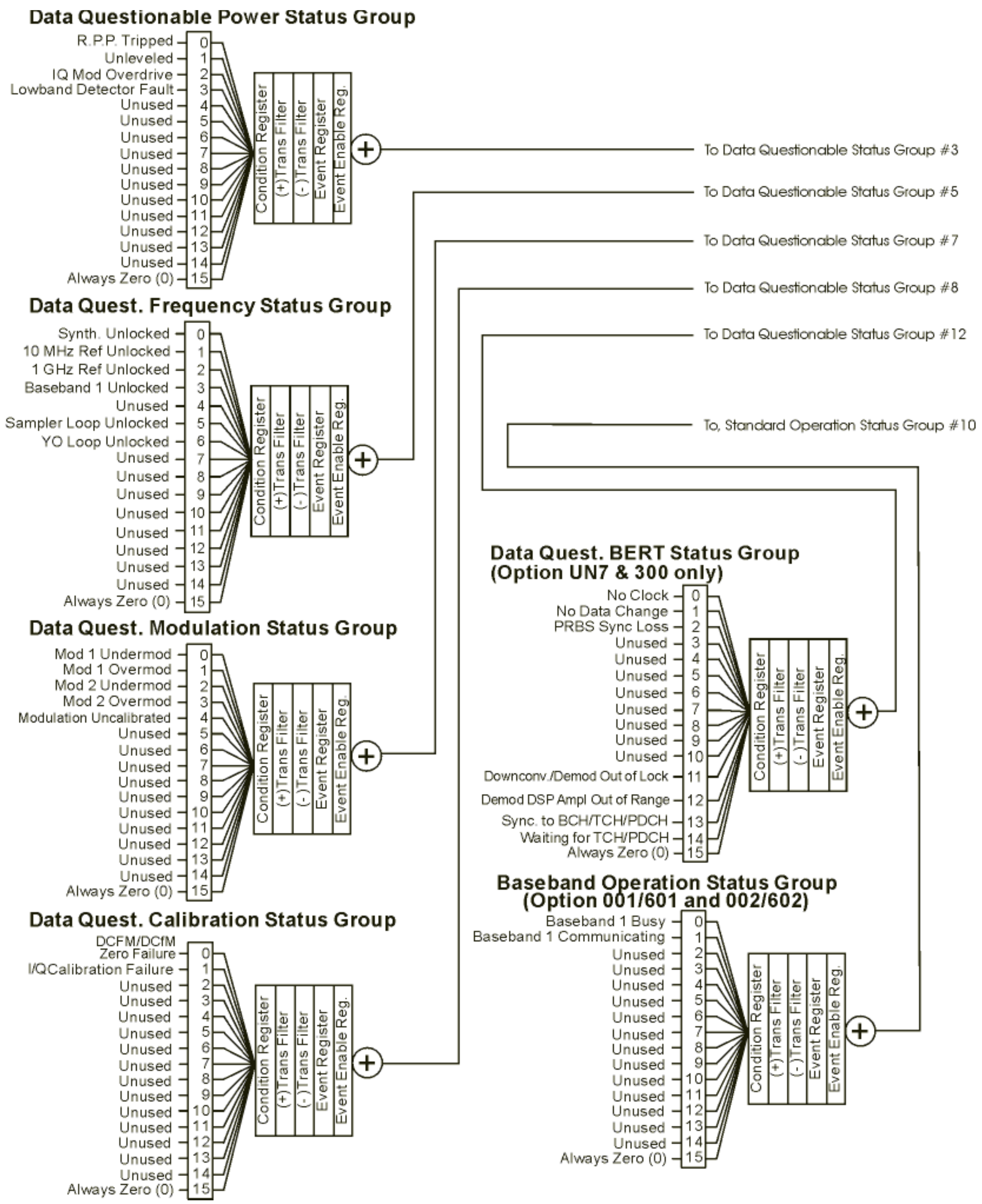
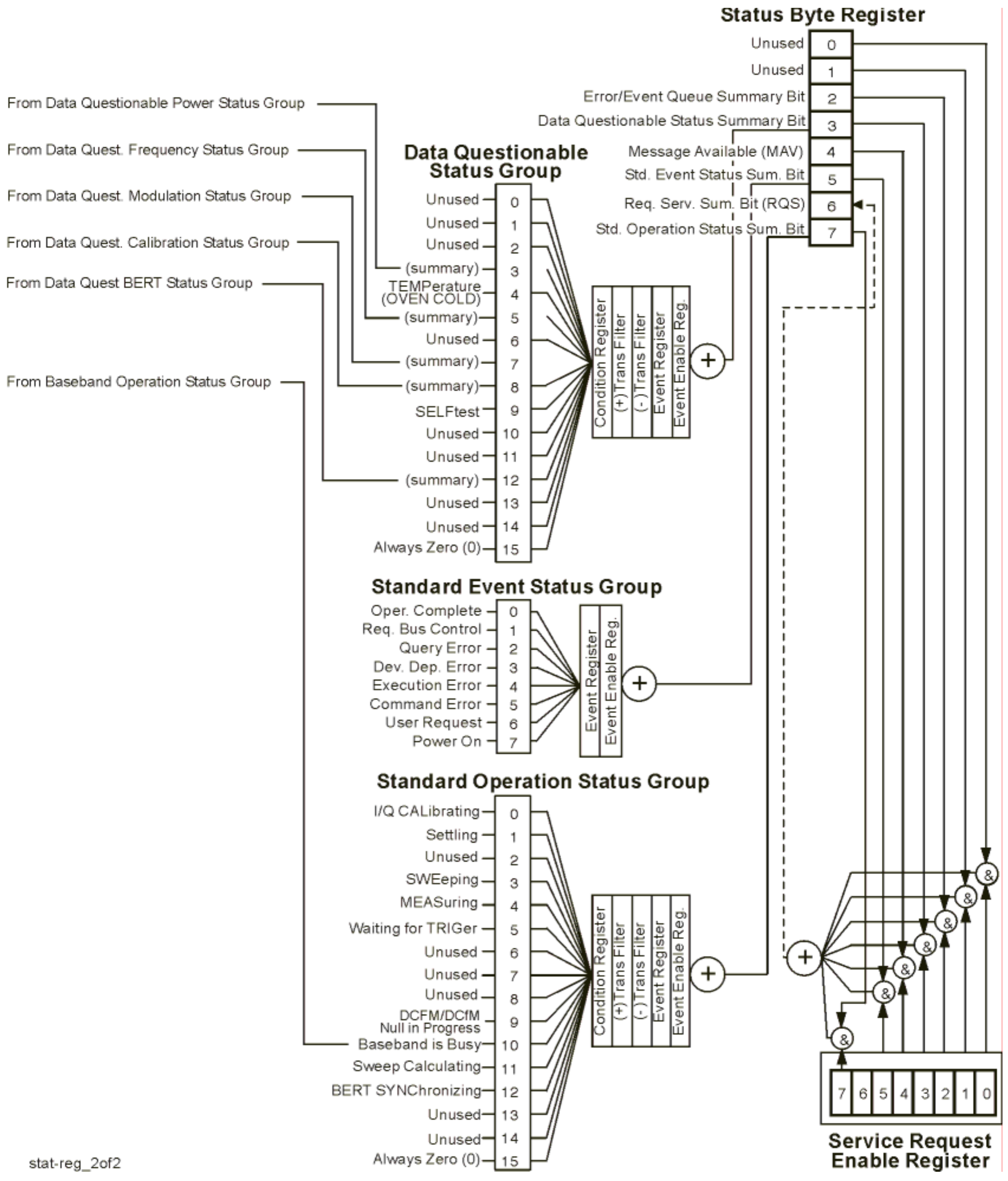

Data Questionable Power Status Group
R.P.P. Tripped
Unleveled
IQ Mod Overdrive
Lowband Detector Fault
Unused
Always Zero (0)
Condition Register
(+)Trans Filter
(-)Trans Filter
Event Register
Event Enable Reg.
To Data Questionable Status Group #3
To Data Questionable Status Group #5
To Data Questionable Status Group #7
To Data Questionable Status Group #8
To Data Questionable Status Group #12
To, Standard Operation Status Group #10
Data Quest. Frequency Status Group
Synth. Unlocked
10 MHz Ref Unlocked
1 GHz Ref Unlocked
Baseband 1 Unlocked
Sampler Loop Unlocked
YO Loop Unlocked
Data Quest. Modulation Status Group
Mod 1 Undermod
Mod 1 Overmod
Mod 2 Undermod
Mod 2 Overmod
Modulation Uncalibrated
Data Quest. Calibration Status Group
DCFM/DCfM Zero Failure
I/QCalibration Failure
Data Quest. BERT Status Group (Option UN7 & 300 only)
No Clock
No Data Change
PRBS Sync Loss
Downconv./Demod Out of Lock
Demod DSP Ampl Out of Range
Sync. to BCH/TCH/PDCH
Waiting for TCH/PDCH
Baseband Operation Status Group (Option 001/601 and 002/602)
Baseband 1 Busy
Baseband 1 Communicating

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 127 or Figure 3-2 on page 128 for register location and names.
2. Send the unique SCPI query that reads that register.
3. Examine the bit to see if the condition has changed.

Determining What to Monitor

You can monitor the following conditions:

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

Monitoring Current Signal Generator Hardware and Firmware Status

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

Monitoring Whether a Condition (Bit) has Changed

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

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

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 127 or Figure 3-2 on page 128 for bit positions and values.

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

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

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

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

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

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

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

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

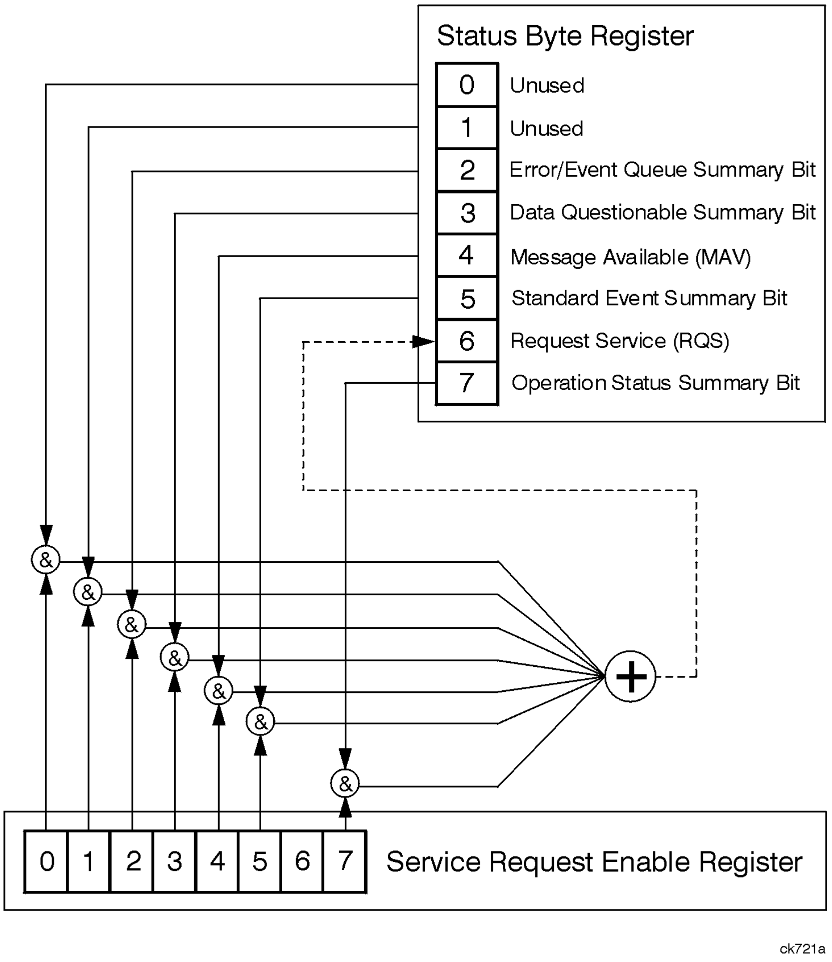
:STATus:PRESet presets all transition filters, non-IEEE 488.2 enable registers, and error/event queue enable registers. (Refer to Table 3-2.)

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:PTRansition	32767
:STATus:QUEStionable:ENABle	0
:STATus:QUEStionable:NTRansition	0
:STATus:QUEStionable:PTRansition	32767
:STATus:QUEStionable:FREQuency:ENABle	32767
:STATus:QUEStionable:FREQuency:NTRansition	32767
:STATus:QUEStionable:FREQuency:PTRansition	32767
:STATus:QUEStionable:MODulation:ENABle	32767
:STATus:QUEStionable:MODulation:NTRansition	32767
:STATus:QUEStionable:MODulation:PTRansition	32767
:STATus:QUEStionable:POWer:ENABle	32767
:STATus:QUEStionable:POWer:NTRansition	32767
:STATus:QUEStionable:POWer:PTRansition	32767
:STATus:QUEStionable:BERT:ENABle	32767
:STATus:QUEStionable:BERT:NTRansition	32767
:STATus:QUEStionable:BERT:PTRansition	32767

Status Byte Group

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


Status Byte Register
0 Unused
1 Unused
2 Error/Event Queue Summary Bit
3 Data Questionable Summary Bit
4 Message Available (MAV)
5 Standard Event Summary Bit
6 Request Service (RQS)
7 Operation Status Summary Bit
&
+
0 1 2 3 4 5 6 7
Service Request Enable Register
ck721a

Status Byte Register

Table 3-3 Status Byte Register Bits

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

Query: *STB?

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

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

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

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

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

Service Request Enable Register

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

`*SRE <data>`	`<data>` is the sum of the decimal values of the bits you want to enable except bit 6. Bit 6 cannot be enabled on this register. Refer to Figure 3-1 on page 127 or Figure 3-2 on page 128.
Example:	To enable bits 7 and 5 to trigger a service request when either corresponding status group register summary bit sets to 1. Send the command `*SRE 160` (128 + 32).
Query:	`*SRE?`
Response:	The decimal value of the sum of the bits previously enabled with the `*SRE <data>` command.

Status Groups

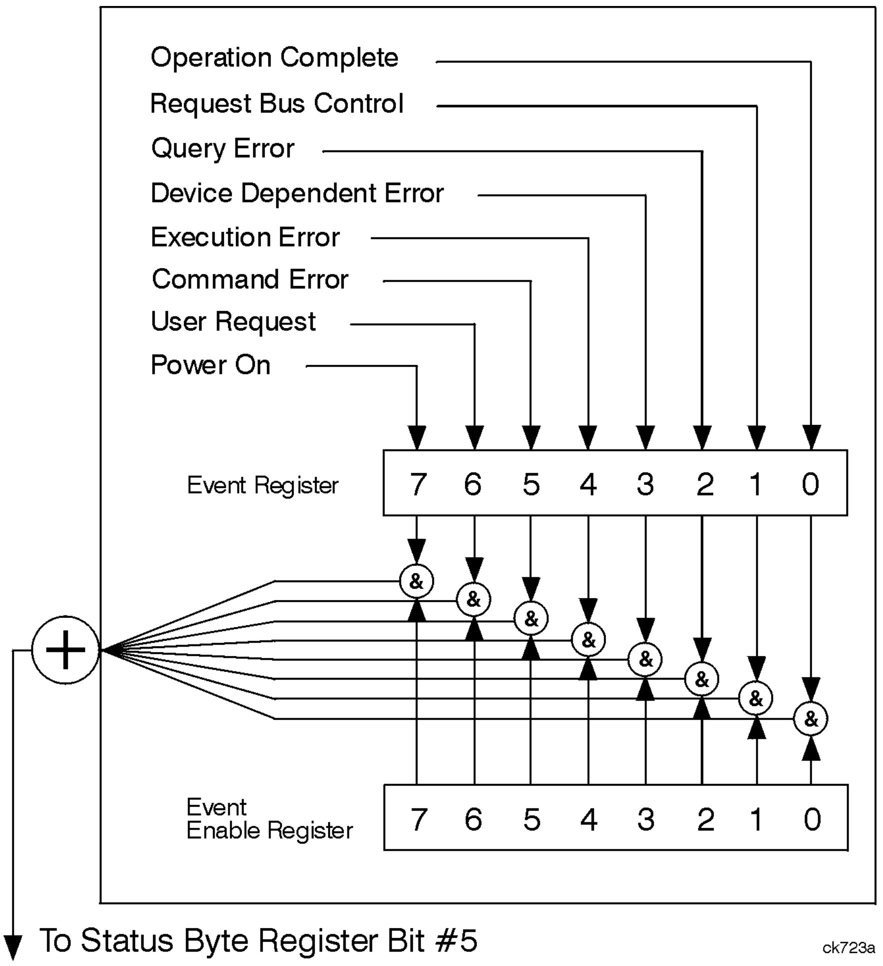
The Standard Operation Status Group and the Data Questionable Status Group consist of the registers listed below. The Standard Event Status Group is similar but does *not* have negative or positive transition filters or a condition register.

Condition Register	A condition register continuously monitors the hardware and firmware status of the signal generator. There is no latching or buffering for a condition register; it is updated in real time.
Negative Transition Filter	A negative transition filter specifies the bits in the condition register that will set corresponding bits in the event register when the condition bit changes from 1 to 0.
Positive Transition Filter	A positive transition filter specifies the bits in the condition register that will set corresponding bits in the event register when the condition bit changes from 0 to 1.
Event Register	An event register latches transition events from the condition register as specified by the positive and negative transition filters. Once the bits in the event register are set, they remain set until cleared by either querying the register contents or sending the `*CLS` command.
Event Enable Register	An enable register specifies the bits in the event register that generate the summary bit. The signal generator logically ANDs corresponding bits in the event and enable registers and ORs all the resulting bits to produce a summary bit. Summary bits are, in turn, used by the Status Byte Register.

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

Standard Event Status Group

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



Standard Event Status Register

Table 3-4 Standard Event Status Register Bits

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

Query:	`*ESR?`
Response:	The *decimal* sum of the bits set to 1
Example:	The decimal value 136 is returned. The decimal sum = 128 (bit 7) + 8 (bit 3).

Standard Event Status Enable Register

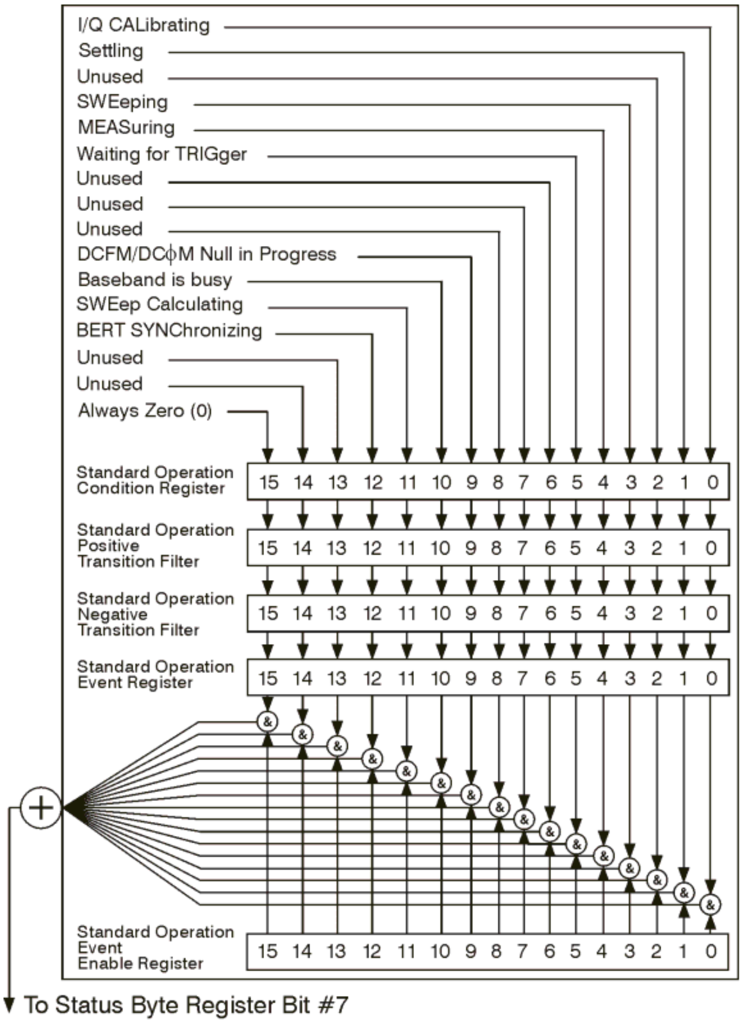
The Standard Event Status Enable Register lets you choose which bits in the Standard Event Status Register set the summary bit (bit 5 of the Status Byte Register) to 1.

`*ESE <data>`	`<data>` is the sum of the decimal values of the bits you want to enable.
Example:	To enable bit 7 and bit 6 so that whenever either of those bits are set to 1, the Standard Event Status summary bit of the Status Byte Register is set to 1. Send the command `*ESE 192` (128 + 64).
Query:	`*ESE?`
Response:	Decimal value of the sum of the bits previously enabled with the `*ESE <data>` command.

Standard Operation Status Group

NOTE Some of the bits in this status group do not apply to the E4428C and will return a zero when queried. See Table 3-5 on page 142 for more information.

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


I/Q CALibrating
Settling
Unused
SWEeping
MEASuring
Waiting for TRIGger
Unused
Unused
Unused
DCFM/DCΦM Null in Progress
Baseband is busy
SWEep Calculating
BERT SYNChronizing
Unused
Unused
Always Zero (0)
Standard Operation Condition Register
Standard Operation Positive Transition Filter
Standard Operation Negative Transition Filter
Standard Operation Event Register
Standard Operation Event Enable Register
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
To Status Byte Register Bit #7

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[a]	**I/Q Calibrating**. A 1 in this position indicates an I/Q calibration is in process.
1	**Settling**. A 1 in this bit position indicates that the signal generator is settling.
2	**Unused**. This bit position is set to 0.
3	**Sweeping**. A 1 in this bit position indicates that a sweep is in progress.
4[a]	**Measuring**. A1 in this bit position indicates that a bit error rate test is in progress
5	**Waiting for Trigger**. 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	**Unused**. These bits are always set to 0.
9	**DCFM/DCϕM Null in Progress**. A 1 in this bit position indicates that the signal generator is currently performing a DCFM/DCΦM zero calibration.
10[a]	**Baseband is Busy**. A 1 in this bit position indicates that the baseband generator is communicating or processing. This is a summary bit. See the “Baseband Operation Status Group” on page 144 for more information.
11	**Sweep Calculating**. A 1 in this bit position indicates that the signal generator is currently doing the necessary pre-sweep calculations.
12[a]	**BERT Synchronizing**. A 1 in this bit position is set while the BERT is synchronizing to ‘BCH’, then ‘TCH’ and then to ‘PRBS’.
12, 13, 14	**Unused**. These bits are always set to 0.
15	**Always 0**.

a. On the E4428C, this bit is set to 0.

Query:	`STATus:OPERation:CONDition?`
Response:	The *decimal* sum of the bits set to 1
Example:	The decimal value 520 is returned. The decimal sum = 512 (bit 9) + 8 (bit 3).

Standard Operation Transition Filters (negative and positive)

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

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

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

Standard Operation Event Register

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

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

Standard Operation Event Enable Register

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

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

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

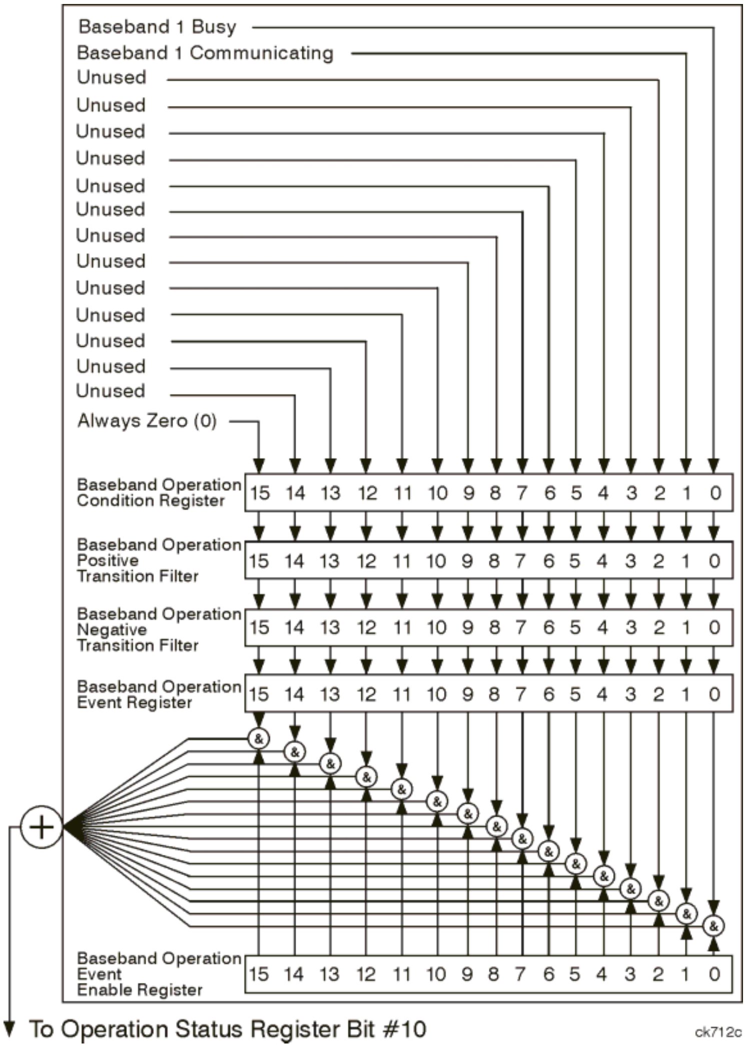
Query: `STATus:OPERation:ENABle?`

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

Baseband Operation Status Group

NOTE This status group does not apply to the E4428C, and if queried will return a zero.

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 1 Busy
Baseband 1 Communicating
Unused
Unused
Unused
Unused
Unused
Unused
Unused
Unused
Unused
Unused
Unused
Unused
Unused
Always Zero (0)
Baseband Operation Condition Register
Baseband Operation Positive Transition Filter
Baseband Operation Negative Transition Filter
Baseband Operation Event Register
Baseband Operation Event Enable Register
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
&
To Operation Status Register Bit #10
ck712c

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	**Baseband 1 Busy**. 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	**Unused**. This bit position is set to 0.
15	**Always 0**.

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 content of that register.

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

Baseband Operation Event Enable Register

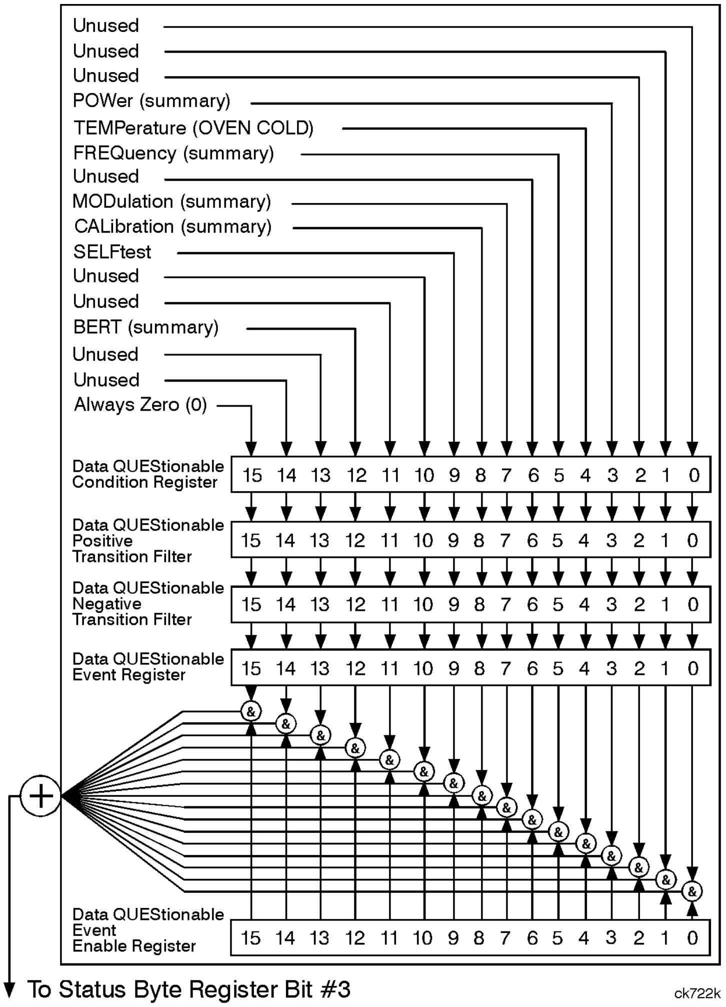
The Baseband Operation Event Enable Register lets you choose which bits in the Baseband Operation Event Register can set the summary bit (bit 7 of the Status Byte Register).

Command:	`STATus:OPERation:BASeband:ENABle <value>`, where `<value>` is the sum of the decimal values of the bits you want to enable.
Example:	To 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 Status Byte Register is set to 1. Send the command `STAT:OPER: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 Some of the bits in this status group do not apply to the E4428C and will return a zero when queried. See Table 3-7 on page 148 for more information.

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



Data Questionable Condition Register

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

Table 3-7 Data Questionable Condition Register Bits

Bit	Description
0, 1, 2	**Unused**. These bits are always set to 0.
3	**Power (summary)**. This is a summary bit taken from the QUEStionable:POWer register. A 1 in this bit position indicates that one of the following may have happened: The ALC (Automatic Leveling Control) is unable to maintain a leveled RF output power (i.e., ALC is UNLEVELED), the reverse power protection circuit has been tripped. See the “Data Questionable Power Status Group” on page 151 for more information.
4	**Temperature (OVEN COLD)**. A 1 in this bit position indicates that the internal reference oscillator (reference oven) is cold.
5	**Frequency (summary)**. This is a summary bit taken from the QUEStionable:FREQuency register. A 1 in this bit position indicates that one of the following may have happened: synthesizer PLL unlocked, 10 MHz reference VCO PLL unlocked, 1 GHz reference unlocked, sampler, YO loop unlocked or baseband 1 unlocked. For more information, see the “Data Questionable Frequency Status Group” on page 154.
6	**Unused**. This bit is always set to 0.
7	**Modulation (summary)**. This is a summary bit taken from the QUEStionable:MODulation register. A 1 in this bit position indicates that one of the following may have happened: modulation source 1 underrange, modulation source 1 overrange, modulation source 2 underrange, modulation source 2 overrange, modulation uncalibrated. See the “Data Questionable Modulation Status Group” on page 157 for more information.
8[a]	**Calibration (summary)**. 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 “Data Questionable Calibration Status Group” on page 160 for more information.
9	**Self Test**. 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, 11	**Unused**. These bits are always set to 0.
12[b]	**BERT (summary)**. This is a summary bit taken from the QUEStionable:BERT register. A 1 in this bit position indicates that one of the following occurred: no BCH/TCH synchronization, no data change, no clock input, PRBS not synchronized, demod/DSP unlocked or demod unleveled. See the “Data Questionable BERT Status Group” on page 163 for more information.

Table 3-7 **Data Questionable Condition Register Bits**

Bit	Description
13, 14	**Unused**. These bits are set to 0.
15	**Always 0**.

a. On the E4428C, this bit applies only to the DCFM/DCΦM calibration.
b. On the E4428C, this bit is set to 0.

Query: `STATus:QUEStionable:CONDition?`

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

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

Data Questionable Transition Filters (negative and positive)

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

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

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

Data Questionable Event Register

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

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

Data Questionable Event Enable Register

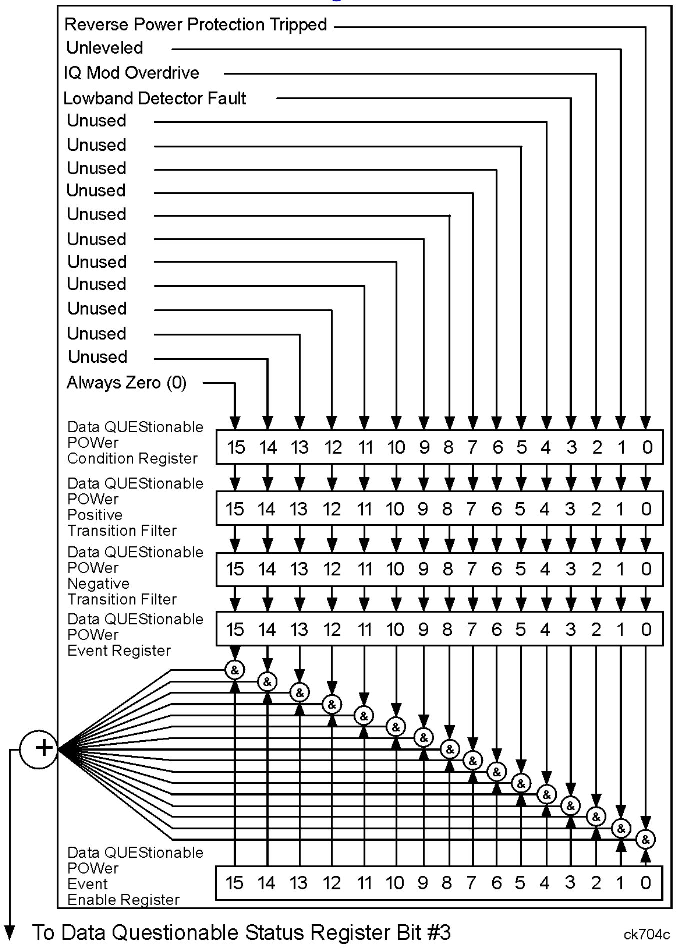
The Data Questionable Event Enable Register lets you choose which bits in the Data Questionable Event Register set the summary bit (bit 3 of the Status Byte Register) to 1.

Command:	`STATus:QUEStionable:ENABle <value>` 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 are set to 1, the Data Questionable Status summary bit of the Status Byte Register is set to 1. Send the command `STAT:QUES:ENAB 520` (512 + 8).
Query:	`STATus:QUEStionable:ENABle?`
Response:	Decimal value of the sum of the bits previously enabled with the `STATus:QUEStionable:ENABle <value>` command.

Data Questionable Power Status Group

NOTE There are two conditions when a bit from this status group does not apply and returns a zero when queried. For more information, see Table 3-8 on page 152.

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.


Reverse Power Protection Tripped
Unleveled
IQ Mod Overdrive
Lowband Detector Fault
Unused
Unused
Unused
Unused
Unused
Unused
Unused
Unused
Unused
Unused
Unused
Always Zero (0)
Data QUEStionable POWer Condition Register
Data QUEStionable POWer Positive Transition Filter
Data QUEStionable POWer Negative Transition Filter
Data QUEStionable POWer Event Register
Data QUEStionable POWer Event Enable Register
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
&
+
To Data Questionable Status Register Bit #3
ck704c

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[a]	**Reverse Power Protection Tripped**. A 1 in this bit position indicates that the reverse power protection (RPP) circuit has been tripped. There is no output in this state. Any conditions that may have caused the problem should be corrected. The RPP circuit can be reset by sending the remote SCPI command: OUTput:PROTection:CLEar. Resetting the RPP circuit bit, will reset this bit to 0.
1	**Unleveled**. A 1 in this bit indicates that the output leveling loop is unable to set the output power.
2[b]	**IQ Mod Overdrive**. A 1 in this bit indicates that the signal level into the IQ modulator is too large.
3	**Lowband Detector Fault**. A 1 in this bit indicates that the lowband detector heater circuit has failed.
2–14	**Unused**. These bits are always set to 0.
15	**Always 0.**

a. On the E4428C/38C with Option 506, this bit is set to 0.
b. On the E4428C, this bit is set to 0.

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

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

Data Questionable Power Transition Filters (negative and positive)

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

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

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

Data Questionable Power Event Register

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

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

Data Questionable Power Event Enable Register

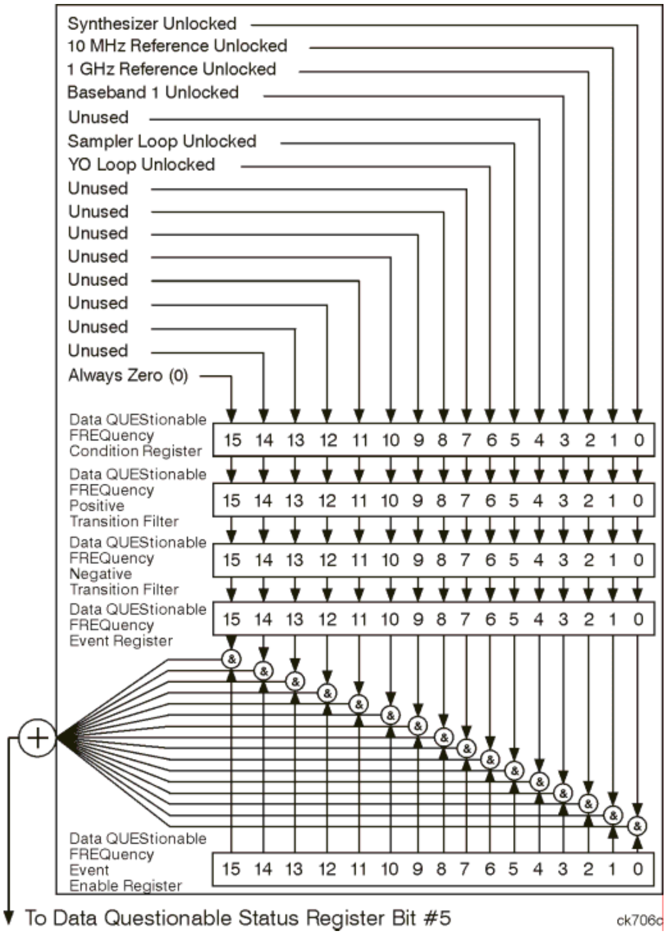
The Data Questionable Power Event Enable Register lets you choose which bits in the Data Questionable Power Event Register set the summary bit (bit 3 of the Data Questionable Condition Register) to 1.

Command:	`STATus:QUEStionable:POWer:ENABle <value>` 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 are set to 1, the Data Questionable Power summary bit of the Data Questionable Condition Register is set to 1. Send the command `STAT:QUES:POW:ENAB 520` (512 + 8).
Query:	`STATus:QUEStionable:POWer:ENABle?`
Response:	Decimal value of the sum of the bits previously enabled with the `STATus:QUEStionable:POWer:ENABle <value>` command.

Data Questionable Frequency Status Group

NOTE A bit in this status group does not apply to the E4428C and will return a zero when queried. See Table 3-9 on page 155 for more information.

The Data Questionable Frequency Status Group is used to determine the specific event that set bit 5 in the Data Questionable Condition Register. This group consists of the Data Questionable Frequency Condition Register, the Data Questionable Frequency Transition Filters (negative and positive), the Data Questionable Frequency Event Register, and the Data Questionable Frequency Event Enable Register.


Synthesizer Unlocked
10 MHz Reference Unlocked
1 GHz Reference Unlocked
Baseband 1 Unlocked
Unused
Sampler Loop Unlocked
YO Loop Unlocked
Unused
Unused
Unused
Unused
Unused
Unused
Unused
Unused
Always Zero (0)
Data QUEStionable FREQuency Condition Register
Data QUEStionable FREQuency Positive Transition Filter
Data QUEStionable FREQuency Negative Transition Filter
Data QUEStionable FREQuency Event Register
Data QUEStionable FREQuency Event Enable Register
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
To Data Questionable Status Register Bit #5
ck706c

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

a. On the E4428C, 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]?`

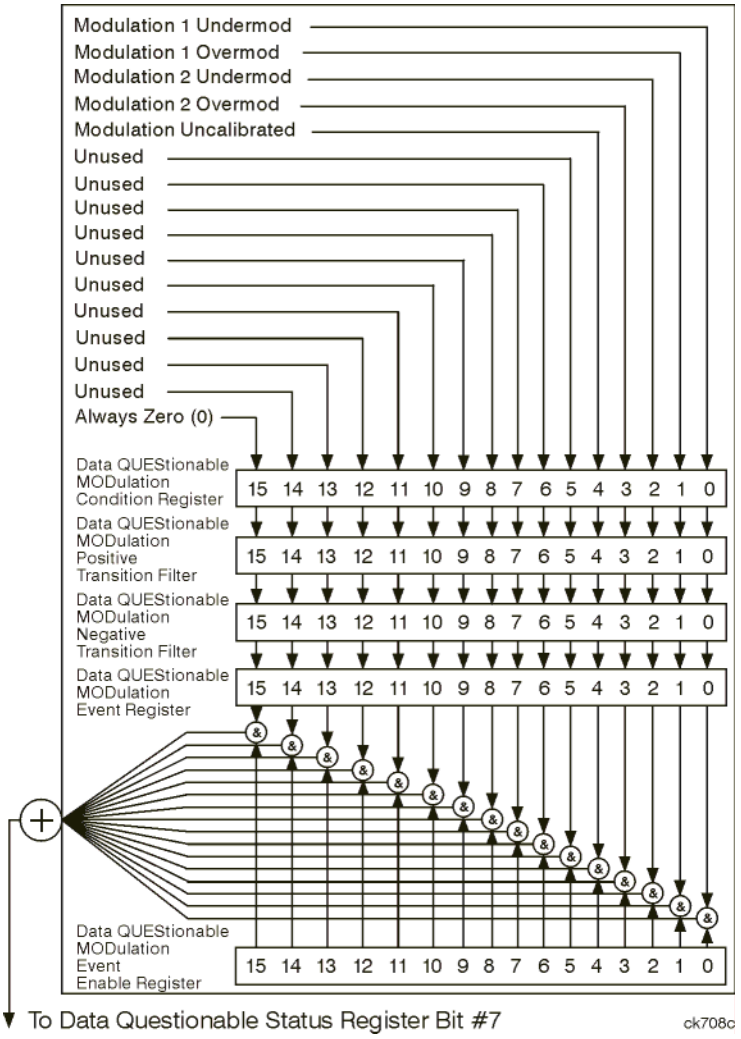
Data Questionable Frequency Event Enable Register

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 are set to 1, the Data Questionable Frequency summary bit of the Data Questionable Condition Register is set to 1. Send the command `STAT:QUES:FREQ:ENAB 520` (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.


Modulation 1 Undermod
Modulation 1 Overmod
Modulation 2 Undermod
Modulation 2 Overmod
Modulation Uncalibrated
Unused
Unused
Unused
Unused
Unused
Unused
Unused
Unused
Unused
Unused
Always Zero (0)
Data QUEStionable MODulation Condition Register
Data QUEStionable MODulation Positive Transition Filter
Data QUEStionable MODulation Negative Transition Filter
Data QUEStionable MODulation Event Register
Data QUEStionable MODulation Event Enable Register
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
&
+
To Data Questionable Status Register Bit #7
ck708c

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	**Modulation 1 Undermod**. A 1 in this bit indicates that the External 1 input, ac coupling on, is less than 0.97 volts.
1	**Modulation 1 Overmod**. A 1 in this bit indicates that the External 1 input, ac coupling on, is more than 1.03 volts.
2	**Modulation 2 Undermod**. A 1 in this bit indicates that the External 2 input, ac coupling on, is less than 0.97 volts.
3	**Modulation 2 Overmod**. A 1 in this bit indicates that the External 2 input, ac coupling on, is more than 1.03 volts.
4	**Modulation Uncalibrated**. A 1 in this bit indicates that modulation is uncalibrated.
5–14	**Unused**. This bit is always set to 0.
15	**Always 0**.

Query: `STATus:QUEStionable:MODulation:CONDition?`

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

Data Questionable Modulation Transition Filters (negative and positive)

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

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

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

Data Questionable Modulation Event Register

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

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

Data Questionable Modulation Event Enable Register

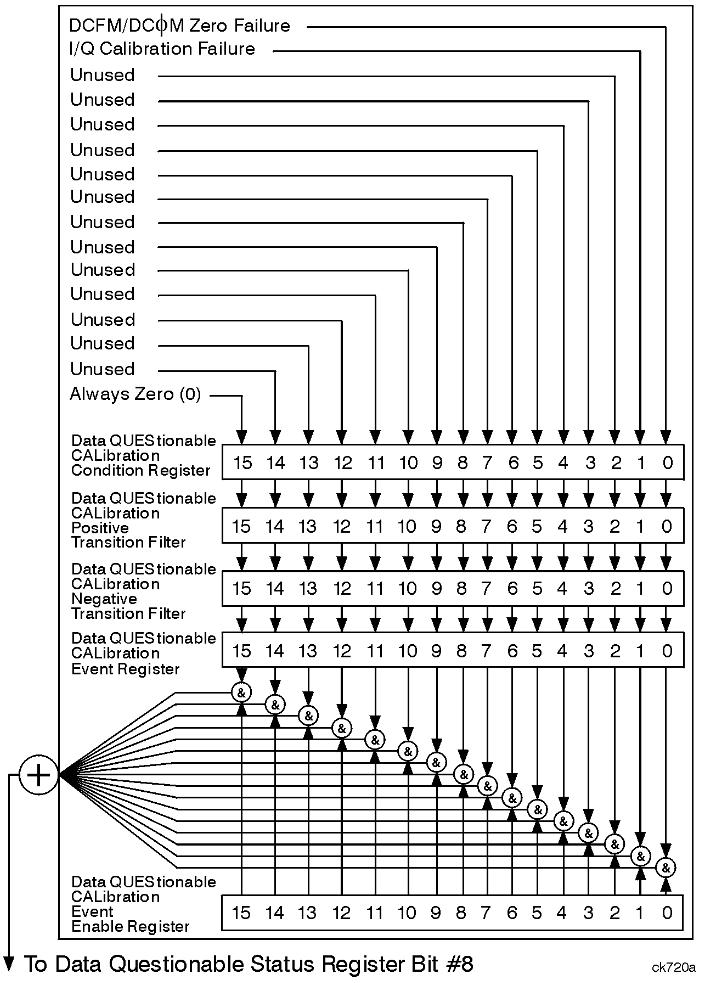
The Data Questionable Modulation Event Enable Register lets you choose which bits in the Data Questionable Modulation Event Register set the summary bit (bit 7 of the Data Questionable Condition Register) to 1.

Command:	`STATus:QUEStionable:MODulation:ENABle <value>` 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 are set to 1, the Data Questionable Modulation summary bit of the Data Questionable Condition Register is set to 1. Send the command `STAT:QUES:MOD:ENAB 520` (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 A bit in this status group does not apply to the E4428C and will return a zero when queried. See Table 3-11 on page 161 for more information.

The Data Questionable Calibration Status Group is used to determine the specific event that set bit 8 in the Data Questionable Condition Register. This group consists of the Data Questionable Calibration Condition Register, the Data Questionable Calibration Transition Filters (negative and positive), the Data Questionable Calibration Event Register, and the Data Questionable Calibration Event Enable Register.



Data Questionable Calibration Condition Register

The Data Questionable Calibration Condition Register continuously monitors the calibration status of the signal generator. Condition registers are read only.

Table 3-11 Data Questionable Calibration Condition Register Bits

Bit	Description
0	**DCFM/DCΦM Zero Failure**. 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 has no validity until the condition of this bit is 0.
1[a]	**I/Q Calibration Failure**. A 1 in this bit indicates that the I/Q modulation calibration experienced a failure.
2–14	**Unused**. These bits are always set to 0.
15	**Always 0**.

a. On the E4428C, 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

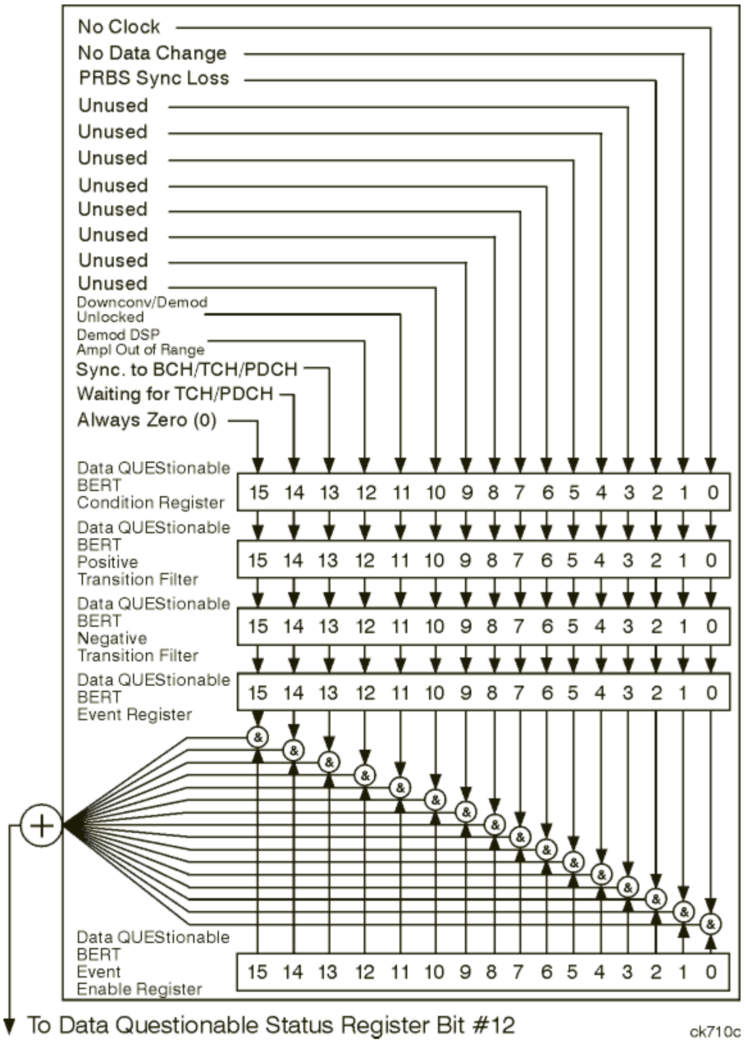
The Data Questionable Calibration Event Enable Register lets you choose which bits in the Data Questionable Calibration Event Register set the summary bit (bit 8 of the Data Questionable Condition register) to 1.

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 are set to 1, the Data Questionable Calibration summary bit of the Data Questionable Condition Register is set to 1. Send the command `STAT:QUES:CAL:ENAB 520` (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.

Data Questionable BERT Status Group

NOTE This status group does not apply to the E4428C, and if queried will return a zero.

The Data Questionable BERT Status Group is used to determine the specific event that set bit 12 in the Data Questionable Condition Register. The Data Questionable Status group consists of the Data Questionable BERT Condition Register, the Data Questionable BERT Transition Filters (negative and positive), the Data Questionable BERT Event Register, and the Data Questionable BERT Event Enable Register.


No Clock
No Data Change
PRBS Sync Loss
Unused
Unused
Unused
Unused
Unused
Unused
Unused
Unused
Downconv/Demod Unlocked
Demod DSP Ampl Out of Range
Sync. to BCH/TCH/PDCH
Waiting for TCH/PDCH
Always Zero (0)
Data QUEStionable BERT Condition Register
Data QUEStionable BERT Positive Transition Filter
Data QUEStionable BERT Negative Transition Filter
Data QUEStionable BERT Event Register
Data QUEStionable BERT Event Enable Register
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
&
+
To Data Questionable Status Register Bit #12
ck710c

Data Questionable BERT Condition Register

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

Table 3-12 Data Questionable BERT Condition Register Bits

Bit	Description
0	**No Clock**. A 1 in this bit indicates no clock input for more than 3 seconds.
1	**No Data Change**. A 1 in this bit indicates no data change occurred during the last 200 clock signals.
2	**PRBS Sync Loss**. A 1 is set while PRBS synchronization is not established. *RST sets the bit to zero.
3–10	**Unused**. These bits are always set to 0.
11	**Down conv. / Demod Unlocked**. A 1 in this bit indicates that either the demodulator or the down converter is out of lock.
12	**Demod DSP Ampl out of range**. A 1 in this bit indicates the demodulator amplitude is out of range. The *RST command will set this bit to zero (0).
13	**Sync. to BCH/TCH/PDCH**. If the synchronization source is BCH, a 1 in this bit indicates BCH synchronization is not established it does not indicate the TCH/PDCH synchronization status. If the sync source is TCH or PDCH, a 1 in this bit indicates that TCH or PDCH synchronization is not established. *RST sets the bit to zero.
14	**Waiting for TCH/PDCH**. A 1 in this bit indicates that a TCH or PDCH midamble has not been received. This bit is set when bit 13 is set. The bit is also set when the TCH or PDCH synchronization was once locked and then lost (in this case the front panel displays “WAITING FOR TCH (or PDCH)”. *RST set the bit to zero.
15	**Always 0.**

Query: `STATus:QUEStionable:BERT:CONDition?`

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

Data Questionable BERT Transition Filters (negative and positive)

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

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

Data Questionable BERT Event Register

The Data Questionable BERT 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:BERT[:EVENt]?`

Data Questionable BERT Event Enable Register

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

Command: `STATus:QUEStionable:BERT: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 are set to 1, the Data Questionable BERT summary bit of the Data Questionable Condition Register is set to 1. Send the command `STAT:QUES:BERT:ENAB 520` (512 + 8).

Query: `STATus:QUEStionable:BERT:ENABle?`

Response: Decimal value of the sum of the bits previously enabled with the `STATus:QUEStionable:BERT: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

NOTE Creating and downloading waveform data is available only in E4438C ESG Vector Signal Generators with Option 001/601 or 002/602.

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 waveforms 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
- PSG/ESG Download Assistant for use only with MATLAB®
  www.agilent.com/find/downloadassistant
- N7622A Signal Studio Toolkit
  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

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

- 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 178.

For more information on waveform data, see "Understanding Waveform Data" on page 170.

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:

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

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 (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.

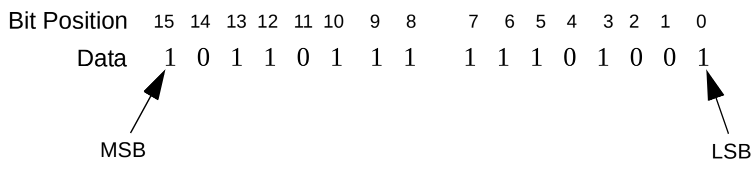
1 = 0001	2 = 0010	3 = 0011	4 = 0100	5 = 0101
6 = 0110	7 = 0111	8 = 1000	9 = 1001	A = 1010
B = 1011	C = 1100	D = 1101	E = 1110	F = 1111

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.



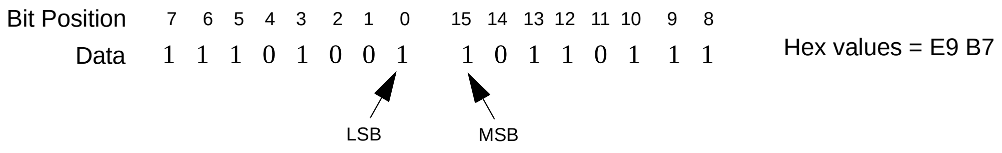
Because we are using 2-bytes of data, the MSB appears in the second byte.

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.

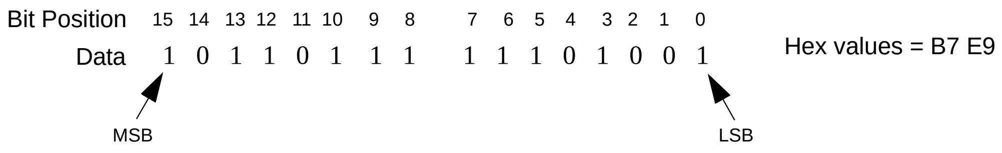
Little Endian Order

The lowest order byte that contains bits 0–7 comes first.



Big Endian Order

The highest order byte that contains bits 8–15 comes first.



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© processors and its clones use little endian. Sun™ 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.

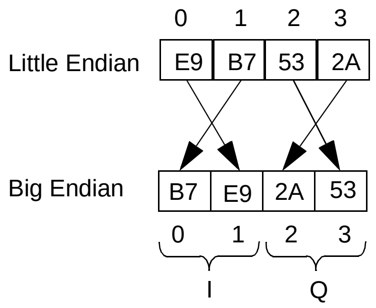
The byte order describes how the system processor stores integer values as binary data in memory. If you output data from a little endian system to a text file (ASCII text), the values are the same as viewed from a big endian system. The order only becomes important when you use the data in binary format, 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 171.

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.

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.



16-bit integer values (2 bytes = 1 integer value)

I data = bytes 0 and 1
Q data = bytes 2 and 3

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.

Voltage	DAC Range	Input Range	Binary Data	Hex Data
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 176.

Using E443xB ESG DAC Input Values

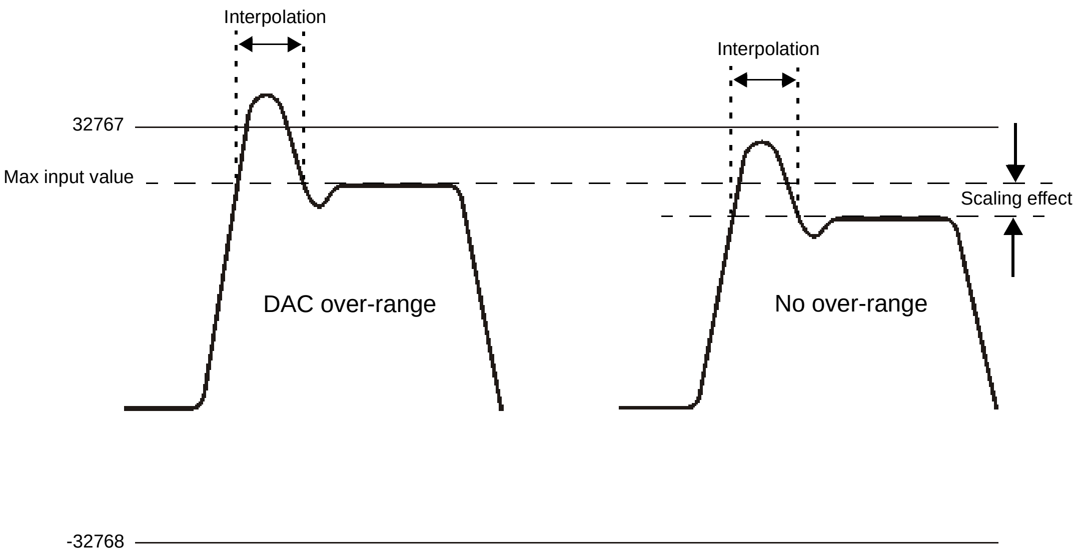
The signal generator's input values differ from those of the earlier E443xB ESG models. For the E443xB models, the input values are all positive (unsigned) and the data is contained within 14 bits plus 2 bits for markers. This means that the E443xB DAC has a smaller range:

- 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 211.

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.

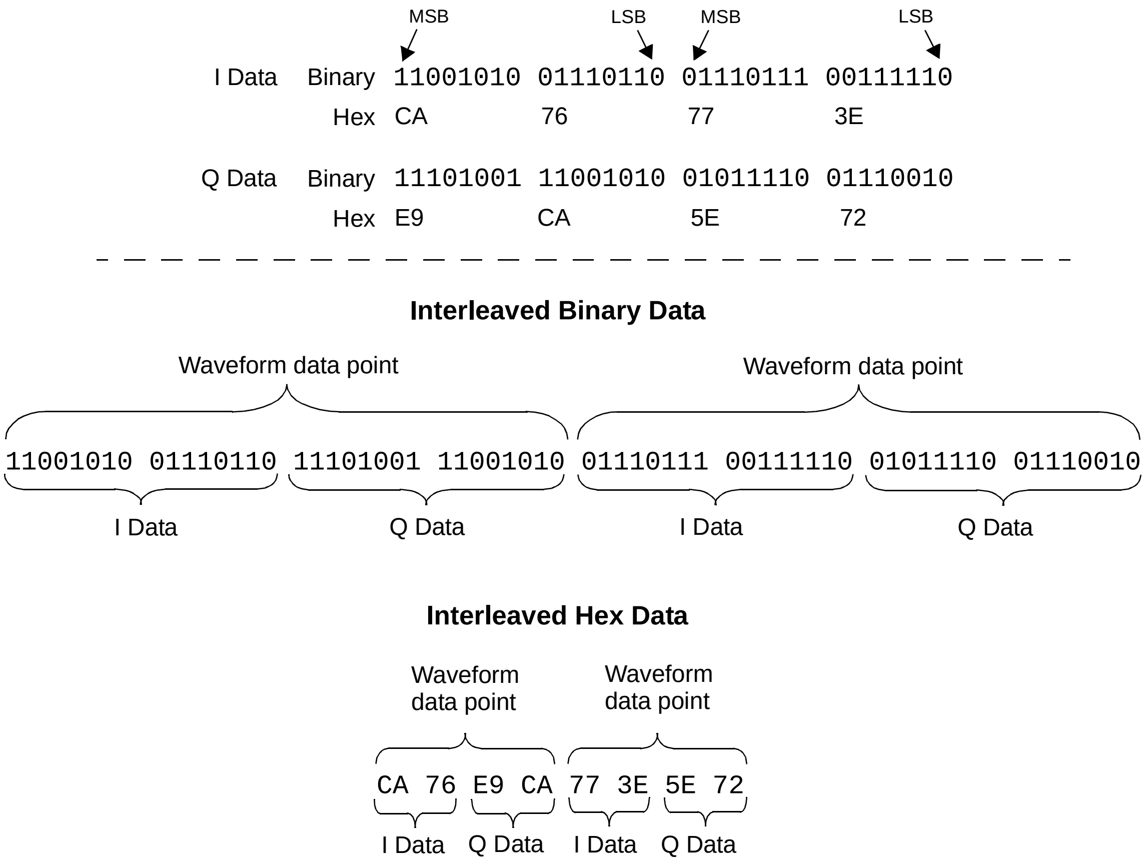
```
  01011100 10011110
+ 10100011 01100001
  00000000 00000000
```

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 211.

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.



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 186.

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 *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.

4 3 2 1 Marker Number Position

Marker Byte $\underbrace{\texttt{0000}}_{\text{Reserved}}\ \texttt{1 0 1 1}$

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
00000010: 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
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
00000040: 05 05 05 05 05 05 05 05 05 05 05 05 05 05 05 05
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
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
```

`0f` = All markers on

`01` = Marker 1 on

`05` = Markers 1 and 3 on

`04` = Marker 3 on

`00` = No active markers

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 *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		1byte (8 bits)		5 bytes
(16 bits)	(16 bits)		Bits 4–7 reserved—Bits 0–3 set		

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 181). When you store waveforms, the signal generator saves changes to the waveform file, marker file, and file header.

Waveform Phase Continuity

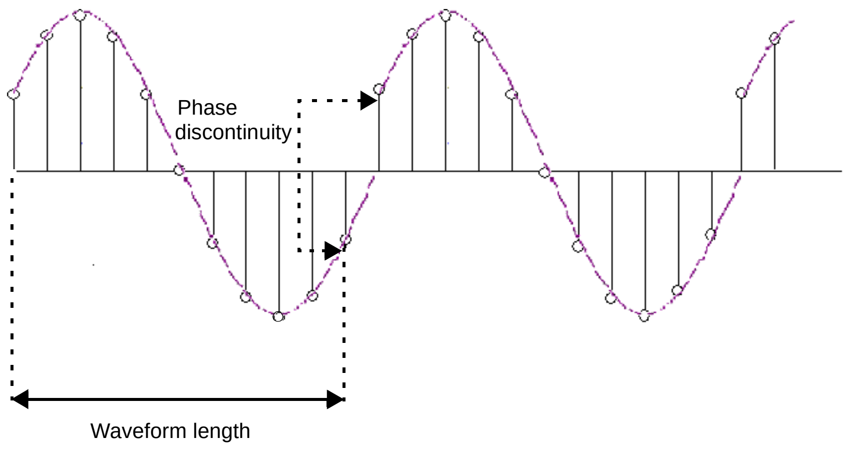
Phase Discontinuity, Distortion, and Spectral Regrowth

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

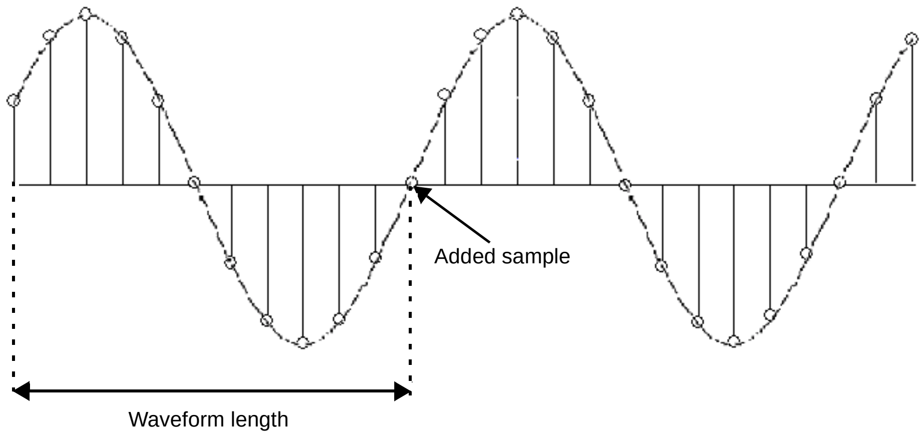
You can easily avoid phase discontinuities for periodic waveforms by simulating an integer number of cycles when you create your waveform segment.

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 Nth 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.

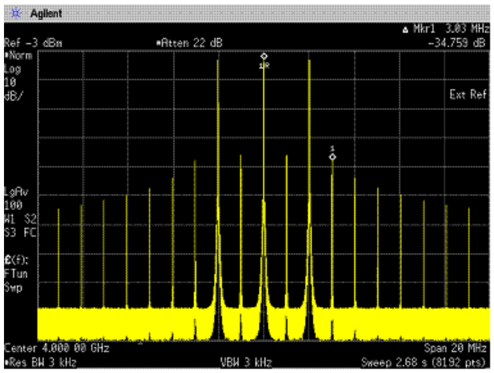
However, if the period of the waveform exceeds the waveform playback memory available in the arbitrary waveform generator, a periodic phase discontinuity could be unavoidable. N5110B Baseband Studio for Waveform Capture and Playback alleviates this concern because it does not rely on the signal generator 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



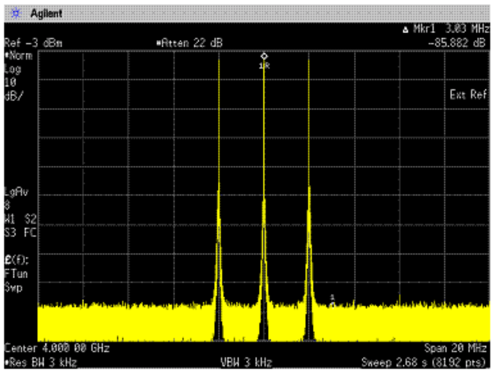
The following figures illustrate the influence a single sample can have. The generated 3-tone test signal requires 100 samples in the waveform to maintain periodicity for all three tones. The measurement on the left shows the effect of using the first 99 samples rather than all 100 samples. Notice all 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.

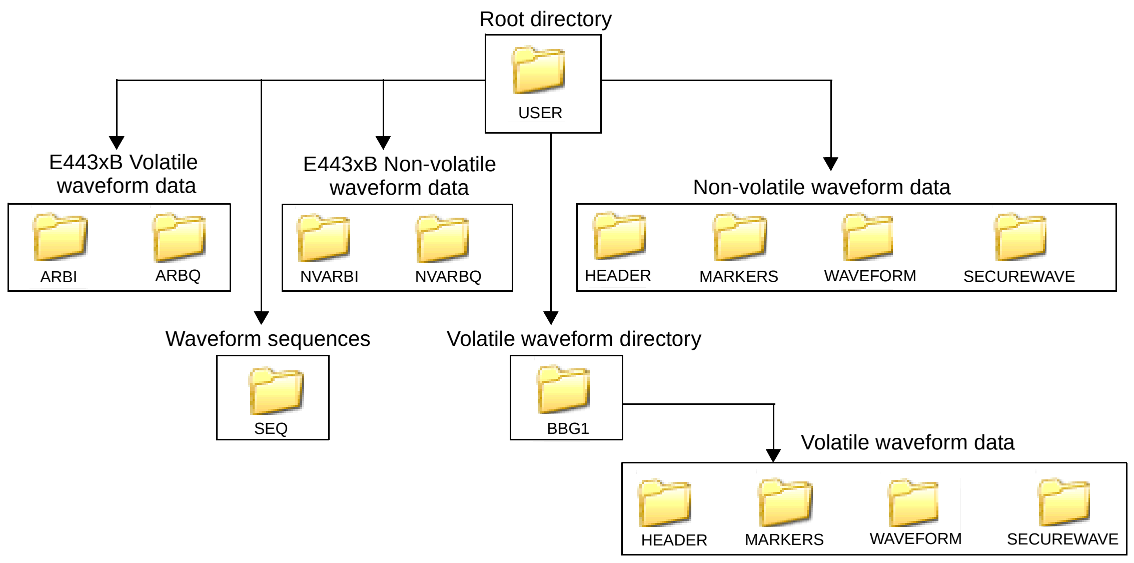
Volatile — Random access memory that does not survive cycling of the signal generator power. This memory is commonly referred to as waveform memory (WFM1) or waveform playback memory. To play back waveforms, they must reside in volatile memory. The following file types share this memory:

- 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 × 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 x 1024 bytes = 3072 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 (001/601 and 002/602) contain the waveform playback memory. The following tables show the maximum available memory.

Volatile (WFM1) Memory	
Option	**Size**
001/601 (BBG)	8 MSa (40 MB)
002 (BBG)	32 MSa (160 MB)
602 (BBG)	64 MSa (320 MB)

Non-Volatile (NVWFM) Memory	
Option	**Size**
Standard	3 MSa (15 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 178.

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

NOTE You can *not* extract files created with ESG firmware revisions prior to C.03.10.

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)

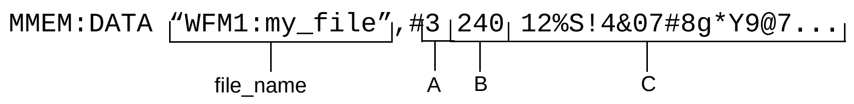
SCPI Command Line Structure

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...
```

`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 188
- Table 4-2, “Downloading Encrypted Files for No Extraction,” on page 189
- Table 4-3, “Downloading Unencrypted Files for Extraction,” on page 189
- Table 4-4, “Downloading Encrypted Files for Extraction,” on page 190
- Table 4-5, “Extracting Encrypted Waveform Data,” on page 191

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/bbg1/waveform/<file_name>",<blockdata>`
	`MMEM:DATA "user/bbg1/markers/<file_name>",<blockdata>`
	`MMEM:DATA "user/bbg1/header/<file_name>",<blockdata>`

Table 4-1 Downloading Unencrypted Files for No Extraction

Download Method / Memory Type	Command Syntax Options

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/bbg1/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/bbg1/waveform/file_name",<blockdata>`
	`MEM:DATA:UNPRotected "/user/bbg1/markers/file_name",<blockdata>`
	`MEM:DATA:UNPRotected "/user/bbg1/header/file_name",<blockdata>`
	`MEM:DATA:UNPRotected "WFM1:file_name",<blockdata>`
	`MEM:DATA:UNPRotected "MKR1:file_name",<blockdata>`
	`MEM:DATA:UNPRotected "HDR1:file_name",<blockdata>`
	`MEM:DATA:UNPRotected "file_name@WFM1",<blockdata>`
	`MEM:DATA:UNPRotected "file_name@MKR1",<blockdata>`
	`MEM:DATA:UNPRotected "file_name@HDR1",<blockdata>`

Table 4-3 Downloading Unencrypted Files for Extraction

Download Method / Memory Type	Command Syntax Options

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[1]	`put <file_name> /user/bbg1/waveform/<file_name>`
	`put <file_name> /user/bbg1/markers/<file_name>`

FTP/non-volatile memory[1]	`put <file_name> /user/waveform/<file_name>`
	`put <file_name> /user/markers/<file_name>`

1 See "FTP Procedures" on page 191.

Table 4-4 Downloading Encrypted Files for Extraction

Download Method / Memory Type	Command Syntax Options

SCPI/volatile memory	`MEM:DATA:UNPRotected "/user/bbg1/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[1]	`put <file_name> /user/bbg1/securewave/<file_name>`
FTP/non-volatile memory[1]	`put <file_name> /user/securewave/<file_name>`

1 See "FTP Procedures" on page 191.

Table 4-5 Extracting Encrypted Waveform Data

Download Method / Memory Type	Command Syntax Options

SCPI/volatile memory	`MMEM:DATA? "/user/bbg1/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[1]	`get /user/bbg1/securewave/<file_name>`
FTP/non-volatile memory[1]	`get /user/securewave/<file_name>`

1 See FTP Procedures.

FTP Procedures

There are three ways to FTP files:

- use Microsoft's ® Internet Explorer FTP feature
- use the signal generator's internal web server (ESG firmware ≥ C.03.76)
- 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

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

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 32.

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.

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 214.

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 168. To learn about I/Q data for the signal generator, see "Understanding Waveform Data" on page 170.

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 235 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 188 and "Downloading Waveform Data" on page 200.

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 "Waveform Memory" on page 184.
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 *short*, 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	Create a loop to do the following: • Generate and scale the I data (DAC values). This example uses a simple sine equation, where 2*3.14 equals one waveform cycle. Change the equation to fit your application. — The array pointer, *index*, increments from 0–499, creating 500 I data points over one period of the sine waveform. — 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 “Scaling DAC Values” on page 174. **NOTE** 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. • Generate and scale the Q data (DAC value). This example uses a simple cosine equation, where 2*3.14 equals one waveform cycle. Change the equation to fit your application. — The array pointer, *index*, increments from 0–499, creating 500 Q data points over one period of the cosine waveform. — 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 “Scaling DAC Values” on page 174.

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, *iq.txt* is the file name and **ofile* 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 ("*/temp/iq.txt*")
13	Open the text file in *write* 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:

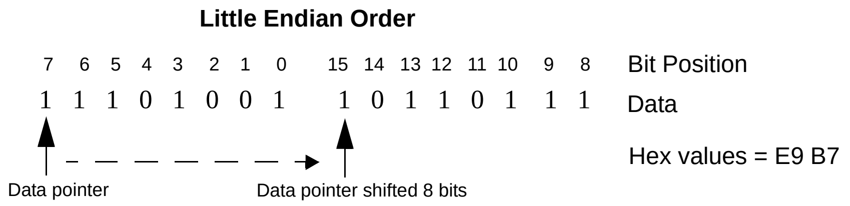
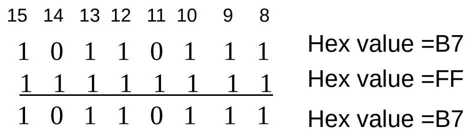
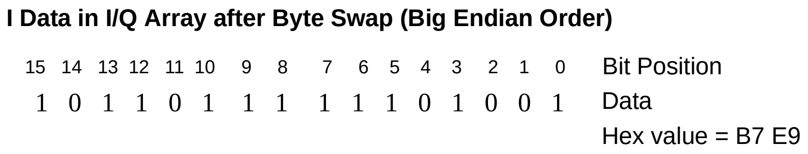
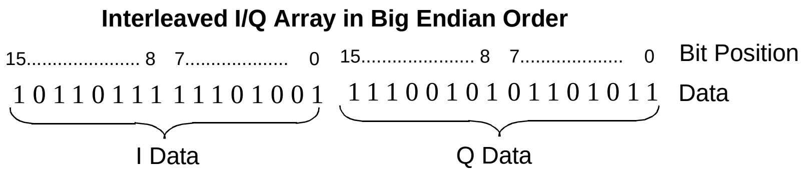
- Interleaving and byte swapping I and Q data for little endian order
- Interleaving I and Q data for big endian order

For more information on byte order, see “Little Endian and Big Endian (Byte Order)” on page 171.

Line Code—Interleaving and Byte Swapping for Little Endian Order

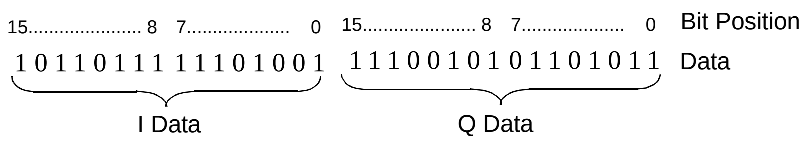
```
20    char iqbuffer[NUMSAMPLES*4];
21    for(index=0; index<numsamples; index++)
22    {
23    short ivalue = idata[index];
24    short qvalue = qdata[index];
25    iqbuffer[index*4]   = (ivalue >> 8) & 0xFF;
26    iqbuffer[index*4+1] = ivalue & 0xFF;
27    iqbuffer[index*4+2] = (qvalue >> 8) & 0xFF;
28    iqbuffer[index*4+3] = qvalue & 0xFF;
29    }
30    return 0;
```

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.
21–29	Create a loop to do the following: • Save the current I data array value to a variable. **NOTE** In rare instances, a compiler may define *short* as larger than 16 bits. If this condition exists, replace *short* with the appropriate object or label that defines a 16-bit integer. • Save the current Q data array value to a variable. • Swap the low bytes (bits 0–7) of the data with the high bytes of the data (done for both

Line	Code Description—Interleaving and Byte Swapping for Little Endian Order
21–29	the I and Q data), and interleave the I and Q data. — shift the data pointer right 8 bits to the beginning of the high byte (*ivalue >> 8*)  Little Endian Order 7 6 5 4 3 2 1 0 15 14 13 12 11 10 9 8 Bit Position 1 1 1 0 1 0 0 1 1 0 1 1 0 1 1 1 Data Hex values = E9 B7 Data pointer Data pointer shifted 8 bits — *AND* (boolean) the high I byte with 0xFF to make the high I byte the value to store in the IQ array—(*ivalue >> 8*) *& 0xFF*  15 14 13 12 11 10 9 8 1 0 1 1 0 1 1 1 Hex value =B7 1 1 1 1 1 1 1 1 Hex value =FF 1 0 1 1 0 1 1 1 Hex value =B7 — *AND* (boolean) the low I byte with 0xFF (*ivalue & 0xFF*) to make the low I byte the value to store in the I/Q array location just after the high byte [*index* * 4 + 1]  I Data in I/Q Array after Byte Swap (Big Endian Order) 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 Bit Position 1 0 1 1 0 1 1 1 1 1 1 0 1 0 0 1 Data Hex value = B7 E9 — 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 [*index* * 4 + n].  Interleaved I/Q Array in Big Endian Order 15...................... 8 7.................... 0 15...................... 8 7.................... 0 Bit Position 1 0 1 1 0 1 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

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	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. **NOTE** In rare instances, a compiler may define *short* as larger than 16 bits. If this condition exists, replace *short* with the appropriate object or label that defines a 16-bit integer.
21–25	Create a loop to do the following: • Store the I data values to the I/Q array location [*index**2]. • Store the Q data values to the I/Q array location [*index**2+1]. **Interleaved I/Q Array in Big Endian Order**  15..................... 8 7................... 0 15..................... 8 7................... 0 Bit Position 1 0 1 1 0 1 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 203.

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 193.

To download data from simulation software environments, it is typically easier to use one of the free download utilities (described on page 210), 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 210).

If you do not need the level of detail this section provides, you can find complete programming examples in "Programming Examples" on page 214. Prior to downloading the I/Q data, ensure that it conforms to the data requirements shown on page 168. To learn about I/Q data for the signal generator, see "Understanding Waveform Data" on page 170. For creating waveform data, see "Creating Waveform Data" on page 193.

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 210.

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      io = agt_newconnection('tcpip','IP address');
       %io = agt_newconnection('gpib',<primary address>,<secondary
2      address>);
3      [status,status_description,query_result] = agt_query(io,'*idn?');
4      if status == -1
5      display ‘fail to connect to the signal generator’;
       end;
```

Line	Code Description—Open a Connection Session with the Signal Generator
1	Sets up a structure (indicated above by *io*) used by subsequent function calls to establish a LAN connection to the signal generator. • *agt_newconnection()* is the function of Agilent Download Assistant used in MATLAB to build a connection to the signal generator. • If you are using GPIB to connect to the signal generator, provide the board, primary address, and secondary address: *io = agt_newconnection('gpib',0,19);* Change the GPIB address based on your instrument setting.
2	Send a query to the signal generator to verify the connection. • *agt_query()* is an Agilent Download Assistant function that sends a query to the signal generator. • If signal generator receives the query `*idn?`, *status* returns a zero and *query_result* returns the signal generator’s model number, serial number, and firmware version.
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     [status, status_description] = agt_waveformload(io, IQwave,
      'waveformfile1', 2000, 'no_play','norm_scale');
7     if status == -1
8     display ‘fail to download to the signal generator’;
9     end;
```

Line	Code Description—Download the I / Q data
6	Download the I/Q waveform data to the signal generator by using the function call (*agt_waveformload*) 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. Notice that with this function, you can perform the following actions: • download complex I/Q data • name the file (optional argument) • set the sample rate (optional argument) 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. • start or not start waveform playback after downloading the data (optional argument) Use either the argument *play* or the argument *no_play*. • whether to normalize and scale the I/Q data (optional argument) If you normalize and scale the data within the body of the code, then use *no_normscale*, but if you need to normalize and scale the data, use *norm_scale*. This normalizes the waveform data to the DAC values and then scales the data to 70% of the DAC values. • download marker data (optional argument) If there is no marker data, the signal generator creates a default marker file, all marker set to zero. To verify the waveform data download, see “Loading, Playing, and Verifying a Downloaded Waveform” on page 207.
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 235.

For information on creating I/Q waveform data, refer to "Creating Waveform Data" on page 193.

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	Assign the signal generator's LAN hostname, IP address, or GPIB address to a character string. • This example uses the Agilent IO library's *iopen()* SICL function to establish a LAN connection with the signal generator. The input argument, *lan[hostname or IP address]* 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: "*lan[999.137.240.9]*" • 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 "*gpib0,19*". • For the detailed information about the parameters of the SICL function *iopen()*, refer to the online "*Agilent SICL User's Guide for Windows*."

Line	Code Description—Open a Connection Session
2	Open a connection session with the signal generator to download the generated I/Q data. The SICL function *iopen()* is from the Agilent IO library and creates a session that returns an identifier to *id*. • If *iopen()* succeeds in establishing a connection, the function returns a valid session *id*. The valid session *id* is not viewable, and can only be used by other SICL functions. • If *iopen()* generates an error before making the connection, the session identifier is set to zero. This occurs if the connection fails. • To use this function in C++, you must include the standard header #include <sicl.h> before the main() function.
3–7	If *id* = 0, the program prints out the error message and exits the program.

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);
14   iwrite(id, cmd, strlen(cmd), 0, 0);
15   iwrite(id, iqbuffer, bytesToSend, 0, 0);
16   iwrite(id, "\n", 1, 1, 0);
```

Line	Code Description—Download the I/Q data
8	Define an integer variable (*bytesToSend*) to store the number of bytes to send to the signal generator.

Line	Code Description—Download the I/Q data
9	Calculate the total number of bytes, and store the value in the integer variable defined in line 8. In this code, *numsamples* 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 *numsamples* by four. This is shown in the following example: numsamples = 500 waveform points numsamples × 4 = 2000 (four bytes per point) bytesToSend = 2000 (numsamples × 4) For information on setting the number of waveform points, see “1. Create I and Q data.” on page 194.
10	Create a string large enough to hold the *bytesToSend* value as characters. In this code, string *s* is set to 20 bytes (20 characters—one character equals one byte)
11	Create a string and set its length (*cmd*[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 *bytesToSend* in string *s*. For example, if bytesToSend = 2000; *s* = ”2000”
13	Store the SCPI command syntax and parameters in the string *cmd*. The SCPI command prepares the signal generator to accept the data. • *sprintf()* is a standard function in C++, which writes string data to a string variable. • *strlen()* is a standard function in C++, which returns length of a string. • If *bytesToSend* = 2000, then *s* = “2000”, *strlen(s)* = 4, so *cmd* = :MEM:DATA ”WFM1:FILE1\” #42000.
14	Send the SCPI command stored in the string *cmd* to the signal generator, which is represented by the session *id*. • *iwrite()* is a SICL function in Agilent IO library, which writes the data (block data) specified in the string *cmd* to the signal generator (*id*). • The third argument of *iwrite()*, *strlen(cmd)*, 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. • The fourth argument of iwrite(), 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.

Line	Code Description—Download the I/Q data
15	Send the generated waveform data stored in the I/Q array *(iqbuffer)* to the signal generator. • *iwrite()* sends the data specified in *iqbuffer* to the signal generator (session identifier specified in *id*). • The third argument of *iwrite()*, *bytesToSend*, contains the length of the *iqbuffer* in bytes. In this example, it is 2000. • The fourth argument of *iwrite()*, 0, means there is no END indicator in the data. In many programming languages, there are two methods to send SCPI commands and data: — Method 1 where the program stops the data download when it encounters the first zero (END indicator) in the data. — 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. 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.
16	Send the terminating carriage (\n) as the last byte of the waveform data. • *iwrite()* writes the data “\n” to the signal generator (session identifier specified in *id)*. • The third argument of *iwrite()*, 1, sends one byte to the signal generator. • The fourth argument of *iwrite()*, 1, is the END indicator, which the program uses to terminate the data download. To verify the waveform data download, see “Loading, Playing, and Verifying a Downloaded Waveform” on page 207.

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>"
```

NOTE 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.

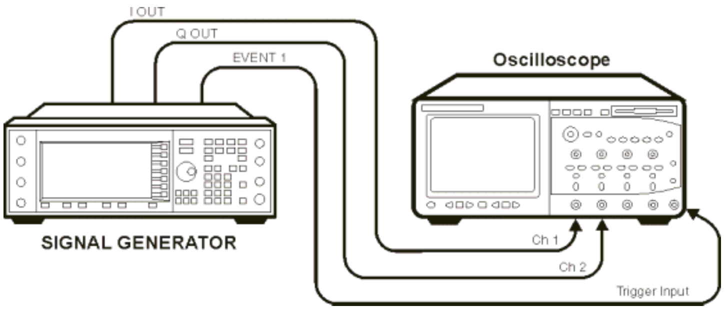
   Remotely send the following SCPI commands:

```
[: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 207.

   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 193 and "Downloading Waveform Data" on page 200. For information on the waveform data requirements, see "Waveform Data Requirements" on page 168.

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*

  This software provides a graphical interface for downloading files.

- Agilent IntuiLink for PSG/ESG Signal Generators: *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*

  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[1]		
Downloads complex MATLAB waveform data			X
Downloads MATLAB files (.mat)	X		
Downloads unencrypted interleaved 16-bit I/Q files [2]	X	X	
Interleaves and downloads earlier 14-bit E443xB I and Q files [2]	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

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

2 ASCII or binary format.

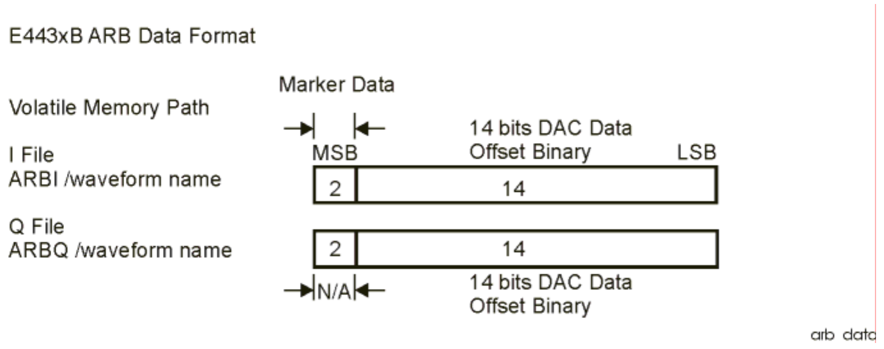
Downloading E443xB Signal Generator Files

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 178 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 178. If you create new waveform files for the signal generator, use the format shown in "Waveform Data Requirements" on page 168.



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/`

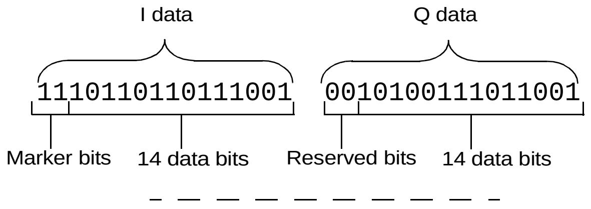
Non-Volatile Memory Storage Locations

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

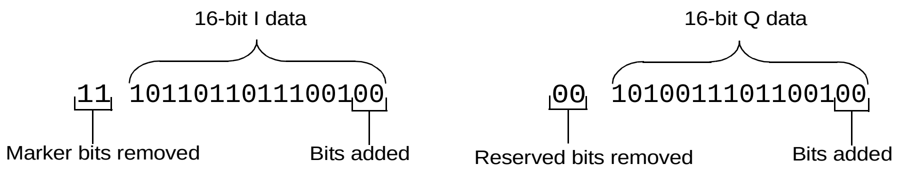
Loading files into the above directories (volatile or non-volatile memory) does not actually store them in those directories. Instead, these directories function as "pipes" to the format translator. The signal generator performs the following functions on the E443xB data:

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

E443xB 14-Bit Data

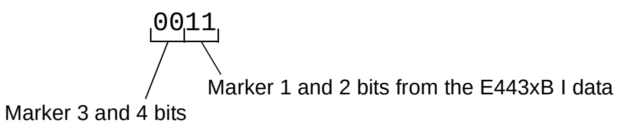


Left Shifts and Adds Zeros—Removes Marker and Reserved Bits (E4438C 16-Bit Data Format)



- Creates a maker file and places the marker information, bits 14 and 15 of the E443xB I data, into the marker file for markers one and two. Markers three and four, within the new marker file, are set to zero (off).

Places the I Marker Bits into the E4438C Marker File



- Interleaves the 16-bit I and Q data creating one I/Q file.
- Creates a file header with all parameters set to unspecified (factory default file header setting).

SCPI Commands

Use the following commands to download E443xB waveform files into the signal generator.

NOTE Before downloading waveform data into volatile memory, turn off the Dual ARB player by pressing **Mode** > **Dual ARB** > **ARB Off On** until Off is highlighted or send the SCPI command `[:SOURce]:RADio:ARB[:STATe] OFF`.

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 215
- "Creating and Storing I/Q Data—Little Endian Order" on page 220
- "Creating and Downloading I/Q Data—Big and Little Endian Order" on page 222
- "Importing and Downloading I/Q Data—Big Endian Order" on page 227
- "Importing and Downloading Using VISA—Big Endian Order" on page 230
- "Importing, Byte Swapping, Interleaving, and Downloading I and Q Data—Big and Little Endian Order" on page 235

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 Waveform" on page 243 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 refered 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\\EsgTestFile”;
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 =”gpib0,19”;

// 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 sampes 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 precentage
        // 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 \”WFM1: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 CodeWarrier 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 <sicl.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 =”gpib0,19”;

// 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\\BurstA1I.bin”;
    char *qfile = “c:\\SignalGenerator\\data\\BurstA1Q.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, 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(ifile, “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 Example

Creating and Storing I/Q Waveform

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 215 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 refered 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 acual bit pattern is that of signed short integers and
%       that is how the ESG/PSG interprets them.
filename = ‘C:\Temp\EsgTestFile’;
[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);
```

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 191).

```
'*************************************************************************************
' 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 complemant, 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 247 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 187 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:

- "Downloading Waveform Data Using HP BASIC for Windows®" on page 255
- "Downloading Waveform Data Using HP BASIC for UNIX" on page 258
- "Downloading E443xB Waveform Data Using HP BASIC for Windows" on page 261
- "Downloading E443xB Waveform Data Using HP Basic for UNIX" on page 263

Downloading Waveform Data Using HP BASIC for Windows®

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 @ESG 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
```

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

```
110   PRINT "Data Generated"
120   Nbytes=4*Num_points
130   ASSIGN @Esg TO 719
140   ASSIGN @Esgb TO 719;FORMAT MSB FIRST
150   Nbytes$=VAL$(Nbytes)
160   Ndigits=LEN(Nbytes$)
170   Ndigits$=VAL$(Ndigits)
180   WAIT 1
190   OUTPUT @Esg USING "#,K";":MMEM:DATA ""WFM1:data_file"",#"
200   OUTPUT @Esg USING "#,K";Ndigits$
210   OUTPUT @Esg USING "#,K";Nbytes$
220   WAIT 1
230   OUTPUT @Esgb;Int_array(*)
240   OUTPUT @Esg;END
250   ASSIGN @Esg TO *
260   ASSIGN @Esgb 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.

Program Comments (Continued)

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 `ESGb` 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.

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 @ESG 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 @Esg TO 719;FORMAT ON
140   ASSIGN @Esgb TO 719;FORMAT OFF
150   Nbytes$=VAL$(Nbytes)
160   Ndigits=LEN(Nbytes$)
170   Ndigits$=VAL$(Ndigits)
180   WAIT 1
190   OUTPUT @Esg USING "#,K";":MMEM:DATA ""WFM1:data_file"",#"
200   OUTPUT @Esg USING "#,K";Ndigits$
```

```
210    OUTPUT @Esg USING "#,K";Nbytes$
220    WAIT 1
230    OUTPUT @Esgb;Int_array(*)
240    WAIT 2
241    OUTPUT @Esg;END
250    ASSIGN @Esg TO *
260    ASSIGN @Esgb 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.

Program Comments (Continued)

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 `ESGb` 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.

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 @ESG USING "#,K";":MMEM:DATA ""NVARBI:testfile"", #"

and replace line 130 with:

210 OUTPUT @ESG 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 @Esg TO 719
120    !ASSIGN @Esgb TO 719;FORMAT MSB FIRST
130    Nbytes$=VAL$(Nbytes)
140    Ndigits=LEN(Nbytes$)
150    Ndigits$=VAL$(Ndigits)
160    OUTPUT @Esg USING "#,K";":MMEM:DATA ""ARBI:file_name_1"",#"
```

```
170    OUTPUT @Esg USING "#,K";Ndigits$
180    OUTPUT @Esg USING "#,K";Nbytes$
190    OUTPUT @Esgb;Iwfm_data(*)
200    OUTPUT @Esg;END
210    OUTPUT @Esg USING "#,K";":MMEM:DATA ""ARBQ:file_name_1"",#"
220    OUTPUT @Esg USING "#,K";Ndigits$
230    OUTPUT @Esg USING "#,K";Nbytes$
240    OUTPUT @Esgb;Qwfm_data(*)
250    OUTPUT @Esg;END
260    ASSIGN @Esg TO *
270    ASSIGN @Esgb 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 table on page 256 for program comments.

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 @Esg TO 719;FORMAT ON
120    ASSIGN @Esgb TO 719;FORMAT OFF
130    Nbytes$=VAL$(Nbytes)
140    Ndigits=LEN(Nbytes$)
150    Ndigits$=VAL$(Ndigits)
160    OUTPUT @Esg USING "#,K";":MMEM:DATA ""ARBI:file_name_1"",#"
170    OUTPUT @Esg USING "#,K";Ndigits$
180    OUTPUT @Esg USING "#,K";Nbytes$
190    OUTPUT @Esgb;Iwfm_data(*)
200    OUTPUT @Esg;END
210    OUTPUT @Esg USING "#,K";":MMEM:DATA ""ARBQ:file_name_1"",#"
```

```
220   OUTPUT @Esg USING "#,K";Ndigits$
230   OUTPUT @Esg USING "#,K";Nbytes$
240   OUTPUT @Esgb;Qwfm_data(*)
250   OUTPUT @Esg;END
260   ASSIGN @Esg TO *
270   ASSIGN @Esgb 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 table on page 259 for program comments.

Troubleshooting Waveform Files

Symptom	Possible Cause
ERROR 224, Text file busy	Attempting to download a waveform that has the same name as the waveform currently being played by the signal generator. To solve the problem, either change the name of the waveform being downloaded or turn off the ARB.
ERROR 628, DAC over range	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. To solve the problem, scale or reduce the DAC input values. For more information, see “DAC Input Values” on page 173.
ERROR 629, File format invalid	The signal generator requires a minimum of 60 samples to build a waveform and the same number of I and Q data points.
ERROR -321, Out of memory	There is not enough space in the ARB memory for the waveform file being downloaded. To solve the problem, either reduce the file size of the waveform file or delete unnecessary files from ARB memory.
No RF Output	The marker RF blanking function may be active. To check for and turn RF blanking off, press **Mode** > **Dual ARB** > **ARB Setup** > **Marker Utilities** > **Marker Routing** > **Pulse/RF Blank** > **None**. This problem occurs when the file header contains unspecified settings and a previously played waveform used the marker RF blanking function. For more information on the marker functions, see the *User’s Guide*.
Undesired output signal	Check for the following: • The data was downloaded in little endian order. See “Little Endian and Big Endian (Byte Order)” on page 171 for more information. • The waveform contains an odd number of samples. An odd number of samples can cause waveform discontinuity. See “Waveform Phase Continuity” on page 181 for more information.

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

NOTE This feature is available only in E4438C ESG Vector Signal Generators with Option 001/601 or 002/602.

The signal generator accepts user file data downloads. The files can be in either binary or bit format, each consisting of 8-bit bytes. Both file types are stored in the signal generator's non-volatile memory.

- In binary format the data is in multiples of 8 bits; all 8 bits of a 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.

Framed and Unframed Data Types

There are two modes that can be used: framed mode and pattern mode (unframed).

- In framed mode, user file data is inserted into the data fields of an existing or user-defined, custom framed digital modulation format, such as DECT, PHS, or TETRA.

  The signal generator's firmware generates the required framing structure and inserts user file data into the data field(s) of the selected format. For more information, see "User Files as Data Source for Framed Transmission" on page 270.

NOTE Unlike pattern RAM (PRAM) downloads to memory, user files contain "data field" information only. The control data bits required for files downloaded directly into PRAM are not required for user file data.

- In pattern 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 each data bit into waveform memory, and sets 31 additional control bits depending upon 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; where each bit is represented by a 32-bit word.

  If the bit rate exceeds 50 Mbps, the user data is written to memory one symbol per 32-bit word, rather than one bit per 32-bit word. This is generally referred to as *parallel* mode.

Bit Memory and Binary Memory

User files can be downloaded to bit memory or binary memory. Bit memory accepts data in integer number of bits, up to the maximum available memory. The data length in bytes for files downloaded into bit memory is equal to the number of significant bits plus 7, divided by 8, then rounded down to the nearest integer plus 8 bytes for the file header. You must have enough bytes to contain the bits you specify. If the number of bits is not a multiple of 8, the least significant bits of the last byte will be ignored.

Bit memory provides more versatility and is the preferred memory for user file downloads.

Binary memory requires data formatted in 8-bit bytes. Files stored or downloaded to binary memory are converted to bit files prior to editing in the bit file editor. Afterward, these modified files from binary memory are stored in bit memory as bit files.

Data Requirements

1. Data must be in binary format.

   SCPI specifies the data in 8-bit bytes.

2. Bit length must be a multiple of the data-field length of the active format.

   Also, the bit length of a user file must be a multiple of the data-field length of the active format in order to completely fill the frame’s data field without leaving a remainder.

   Remaining data is truncated by the signal generator’s firmware and is therefore not present in the resulting waveform at the RF output.

3. Bit length must be a multiple of 8 (binary downloads only).

   SCPI specifies data in 8-bit bytes, and the binary memory stores data in 8-bit bytes.
   If the length (in bits) of the original data pattern is not a multiple of 8, you may need to:

   - add additional bits to complete the ASCII character,
   - 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, or
   - use a bit file and download to bit memory instead.

Data Limitations

Maximum selectable file sizes are directly proportional to the available memory space and the signal generator’s pattern RAM (volatile memory) size. To determine the maximum user file size, you must consider the following:

- framing overhead
- pattern RAM storage size (Option 001/601 = 800 kB, Option 002 = 3.2 MB, or Option 602 = 6.4 MB)

  The maximum memory for bit and binary user data is less than the maximum memory for PRAM data.
- available memory

You may have to delete files from memory before downloading larger files. For more information on signal generator memory, see “Waveform Memory” on page 184.

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) actually utilize the same storage media.

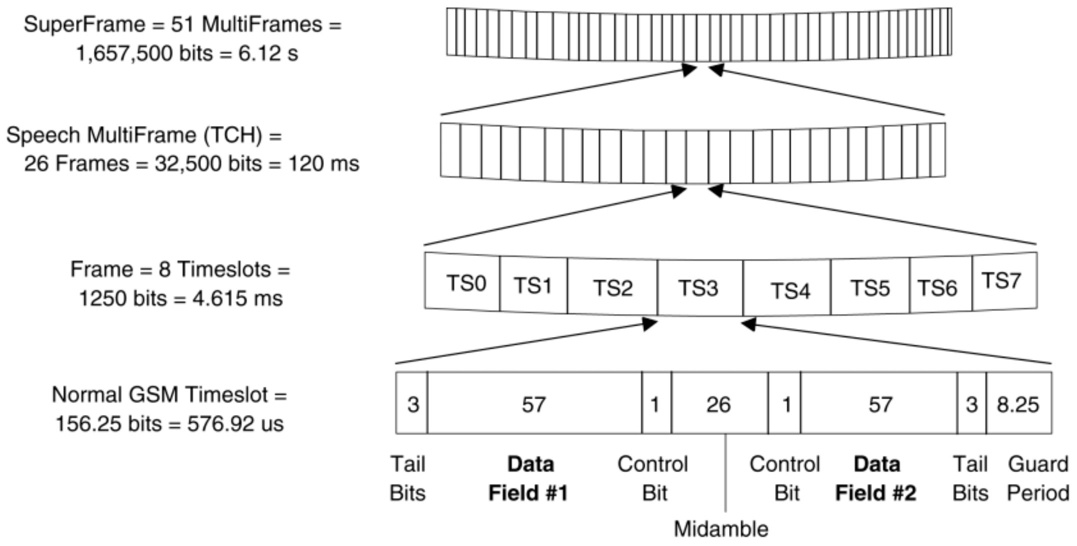
Data Volatility

The signal generator provides two data storage areas: volatile waveform memory (WFM1) and non-volatile memory (NVWFM). Data stored in volatile waveform memory cannot be recovered if it is overwritten or if the power is cycled. Data stored in non-volatile memory, however, remains until you delete the file. The Option 005 signal generator’s hard disk provides 5 GB of non-volatile storage. Signal generators without Option 005 provide 15 MB of non-volatile storage.

User Files as Data Source for Framed Transmission

Specifying a user file as the data source for a framed transmission provides you with an easy method to multiplex real data into internally generated TDMA framing. The user file will fill the data fields of the active timeslot in the first frame, and continue to fill the same timeslot of successive frames as long as there is more data in the file. This functionality allows a communications system designer to download and modulate proprietary data sequences, specific PN sequences, or simulate multiframe transmission, such as those specified by some mobile communications protocols. As the example in the following figure shows, a GSM multiframe transmission requires 26 frames for speech.

Figure 5-1 GSM Multiframe Transmission


SuperFrame = 51 MultiFrames = 1,657,500 bits = 6.12 s
Speech MultiFrame (TCH) = 26 Frames = 32,500 bits = 120 ms
Frame = 8 Timeslots = 1250 bits = 4.615 ms
TS0
TS1
TS2
TS3
TS4
TS5
TS6
TS7
Normal GSM Timeslot = 156.25 bits = 576.92 us
3
57
1
26
1
57
3
8.25
Tail Bits
Data Field #1
Control Bit
Midamble
Control Bit
Data Field #2
Tail Bits
Guard Period

When a user file is selected as the data source for a framed transmission, the signal generator's firmware loads PRAM with the framing protocol of the active TDMA format. For all addresses corresponding to active (on) timeslots, burst bits are set to 1 and data bits are set with the contents of the user file for the data fields of the timeslot. Other bits are set according to the configuration selected. For inactive (off) timeslots, burst control bits are set to 0, and data is "unspecified." Pattern reset is set to 1 for the last byte in PRAM, causing the pattern to repeat after the last byte is read.

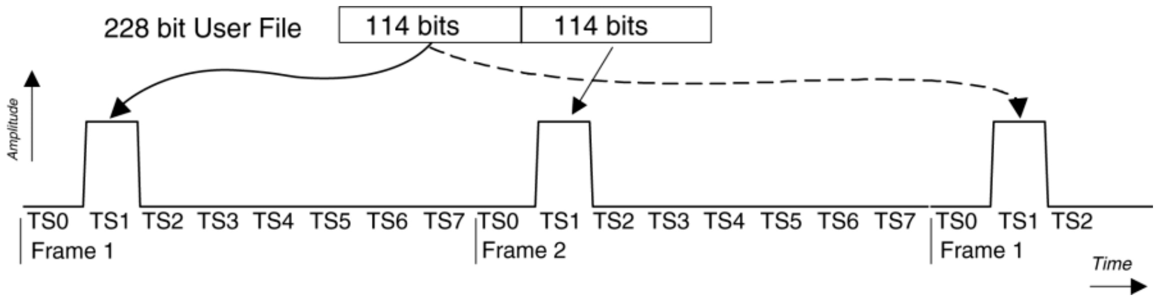
NOTE The data in PRAM is static. Firmware writes to PRAM once for the configuration selected and the hardware reads this data repeatedly. Firmware overwrites the volatile PRAM memory to reflect the desired configuration only when the data source or mode (digital communications format) is changed.

Take for example, transmitting a 228-bit user file for timeslot #1 (TS1) in a normal GSM transmission. Per the standard, a GSM normal channel is 156.25-bits long, with two 57-bit data fields (114 bits total per timeslot), and 42 bits for control or signalling purposes.

NOTE Compliant with the GSM standard, which specifies 156.25-bit timeslots, the signal generator uses 156-bit timeslots and adds an extra guard bit every fourth timeslot.

The 7 remaining timeslots in the GSM frame are off. The user file will completely fill timeslot #1 in two consecutive frames, and will then repeat. See Figure 5-2.

Figure 5-2 Mapping User File Data to a Single Timeslot


228 bit User File
114 bits
114 bits
Amplitude
TS0 TS1 TS2 TS3 TS4 TS5 TS6 TS7 TS0 TS1 TS2 TS3 TS4 TS5 TS6 TS7 TS0 TS1 TS2
Frame 1
Frame 2
Frame 1
Time

For this protocol configuration, the signal generator’s firmware loads PRAM with the bits defined in the following table.

Frame	Timeslot	PRAM Word Offset	Data Bits	Burst Bits	Pattern Reset Bit
1	0	0 -155	0/1 (don’t care)	0 (off)	0 (off)
1	1 (on)	156 - 311	set by GSM standard (42 bits) & first 114 bits of user file	1 (on)	0
1	2	312 - 467	0/1 (don’t care)	0	0
1	3	468 - 624	0/1 (don’t care)	0	0
1	4	625 - 780	0/1 (don’t care)	0	0
1	5	781 - 936	0/1 (don’t care)	0	0
1	6	937 - 1092	0/1 (don’t care)	0	0
1	7	1093 - 1249	0/1 (don’t care)	0	0
2	0	1250 - 1405	0/1 (don’t care)	0	0
2	1 (on)	1406 - 1561	set by GSM standard (42 bits) & remaining bits of user file	1 (on)	0
2	2 through 6	1562 - 2342	0/1 (don’t care)	0	0 (off)
2	7	2343 - 2499	0/1 (don’t care)	0	0 (1 in offset 2499 only)

Event 1 output is set to 0 or 1 depending on the sync out selection, which enables the Event 1 output at either the beginning of the frame, beginning of a specific timeslot, or at all timeslots.

Because timeslots are configured and enabled within the signal generator, a user file can be individually assigned to one or more timeslots. A timeslot cannot have more than one data source (PN sequence or user file) specified for it. The amount of user file data that can be mapped into hardware memory depends on both the amount of PRAM available on the baseband generator, and the number and size of each frame. The amount of PRAM required for a framed transmission is calculated as follows:

PRAM storage required (measured in 32-bit words) =
size of normal GSM timeslot $\times$ timeslots per frame $\times$ speech multiframe(TCH) $\times$ superframe

size of normal GSM timeslot = 156.25 bits

timeslots per frame = 8 timeslots.

speech multiframe(TCH) = 26 frames

superframe = 51 speech multiframes

For example, to calculate the number of bytes to generate a superframe for GSM:

$= 156.25 \times 8 \times 26 \times 51$

= 1,657,5000 32-bit words = 6,630,000 bytes.

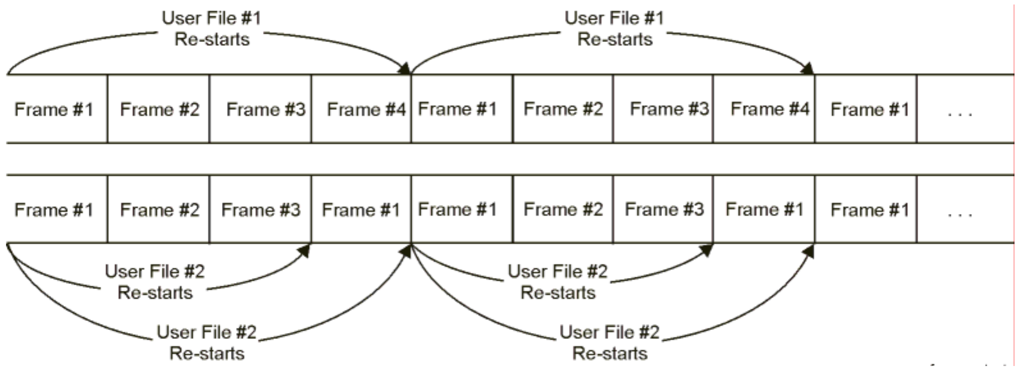
Multiple User Files Selected as Data Sources for Different Timeslots

If two or more user files are selected for a framed transmission, the amount of PRAM required is determined by the user file that generates the largest number of frames. In order to generate continuously repeating data patterns, each user file must be long enough to completely fill an integer number of timeslots. In addition, all user files must meet the “multiple of 8 bits” and “enough PRAM memory” requirements to be correctly modulated.

For example, user file #1 contains 114 bits and fills the data fields of a normal GSM timeslot, and user file #2 contains 148 bits for a custom GSM timeslot. In order to correctly transmit these data patterns as continuously repeating user files without discontinuities, both data patterns must be repeated four times. Therefore, user file #1 contains 456 bits, and user file 2 contains 592 bits. Each user file will then create exactly four frames in pattern RAM.

When two or more user files generate different numbers of complete frames, the user files will repeat on different cycles. All user files will restart when the user file that generates the largest number of frames repeats. For example, user file #1 needs four frames to completely transmit its data, and user file #2 needs only three. User file #2 will repeat after the third frame, and again when user file #1 repeats. See Figure 5-3. If these were integer multiples of each other, both user files would be continuous, and user file #2 would repeat after two frames.

Figure 5-3 Repeating Different Length User Files



Downloading User File Data

This section includes information that explains how to download user file data. It includes data requirements and limitations, preliminary setup, SCPI commands and sample command lines for both downloads to bit memory and binary memory.

Data Requirements and Limitations Summary

1. Data must be binary.
2. Bit length must be a multiple of the data-field length of the active TDMA format.
3. User file size is limited by the available memory.
4. When designing user files, you must consider the signal generator’s PRAM storage size (Option 001/601 = 800 kB, Option 002 = 3.2 MB, or Option 602 = 6.4 MB), framing overhead, and available memory.

   The maximum memory for bit and binary user data is less than the maximum memory for PRAM data.
5. For downloads to binary memory, bit length must be a multiple of 8; SCPI specifies the data in 8-bit bytes.

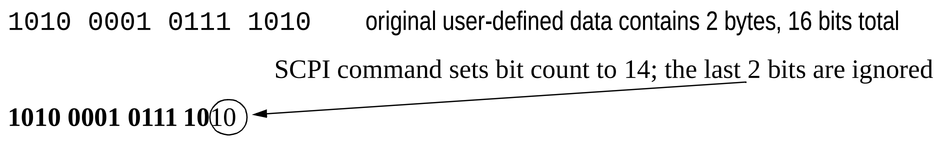
No preliminary setup is required for user file downloads.

Bit Memory Downloads

Bit memory accepts data in any integer number of bits, up to the maximum available memory. The data length in bytes for files downloaded to bit memory is equal to the number of significant bits plus 7, divided by 8, then rounded down to the nearest integer plus 8 bytes for the file header. Each file has a 16-byte header associated with it.

You must have enough bytes to contain the bits you specify. If the number of bits is not a multiple of 8, the least significant bits of the last byte will be 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.


1010 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
1010 0001 0111 1010

Bit memory provides more versatility and is preferred for user file downloads.

SCPI Commands

Send the following command to download the user file data into the signal generator’s bit memory catalog.

```
:MEMory:DATA:BIT "<file_name>",<bit count>,<data block>
```

Example

```
:MEMory:DATA:BIT "<file_name>",16,#12Qz
```

`file_name`	provides the user file name as it will appear in the signal generator’s bit memory catalog
`16`	states the number of bits to download
`#`	indicates the start of the data block
`1`	states the number of decimal digits that follows this number that defines the number of data bytes in the data block
`2`	denotes the number of data bytes in the data block, which follows
`Qz`	the ASCII representation of the 16 bits of data (data block) to be downloaded to the signal generator

Querying the Waveform Data

Use the following SCPI command to query user file data from the bit memory catalog:

```
:MEMory:DATA:BIT? "<file_name>"
```

The output format is the same as the input format.

Binary Memory Downloads

Binary memory requires data formatted in 8-bit bytes. Files stored or downloaded to binary memory are converted to bit files prior to editing in the Bit File Editor. Afterward, these modified files from binary memory are stored in bit memory as bit files. Bit memory is the 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 memory. The variable `<file_name>` denotes the name that will be associated with the downloaded user file stored in the signal generator.

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 binary memory:

```
:MMEM:DATA? "file_name"
```

The output format is the same as the input format.

Selecting Downloaded User Files as the Transmitted Data

Unframed Data

The following front panel key presses or remote commands will select the desired user file from the catalog of user files as a continuous stream of unframed data for the active TDMA format or for a custom modulation.

Via the front panel:

1. For a TDMA format, press **Mode** > **Real Time TDMA** > ***desired format*** > **Data** > **User File**.

   For custom modulation, press **Mode** > **Custom** > **Real Time I / Q Baseband** > **Data** > **User File**.
2. Highlight the desired file in the catalog of user files.
3. Press **Select File** > ***desired format*** **Off On** or **Custom Off On** to On.

Via the remote interface:

The following commands activate the desired TDMA format:

```
[:SOURce]:RADio:<desired format>:DATA "BIT:<file_name>"
```

```
[:SOURce]:RADio:<desired format>[:STATe] On
```

The following commands activate the custom modulation format:

```
[:SOURce]:RADio:CUSTom:DATA "BIT:<file_name>"
```

```
[:SOURce]:RADio:CUSTom[:STATe] On
```

NOTE To select a user file from binary memory, send the same commands shown in the above examples without `BIT:` preceding the file name. For example:

```
[:SOURce]:RADio:<desired format>:DATA "<file_name>"
```

Framed Data

The following front panel key presses or remote commands will select the desired user file from the catalog of user files as a continuous stream of framed data for the active TDMA format.

Via the front panel:

1. Press **Mode** > **Real Time TDMA** > ***desired format*** > **Data Format Pattern Framed** > **Configure Timeslots** > **Configure (current active timeslot)** > **Data** > **User File**.
2. Highlight the desired file in the catalog of user files.
3. Press **Select File**
4. To activate the TDMA format, press **Mode** > **Real Time TDMA** > ***desired format*** > toggle the format on.

Via the remote interface:

The following SCPI commands select and activate the user file as framed data for an NADC uplink traffic channel in timeslot 1. The same command syntax is used for other data transmission formats.

```
[:SOURce]:RADio:NADC:DATA "BIT:<file_name>"
```

```
[:SOURce]:RADio:NADC[:STATe] On
```

The following commands load the data and activate the NADC modulation format:

```
[:SOURce]:RADio:NADC:SLOT1:UTCHannel:DATA "BIT:<file_name>"
```

```
[:SOURce]:RADio:NADC[:STATe] On
```

Modulating and Activating the Carrier

The following settings can be performed from the front panel or by using remote commands to modulate the carrier and turn on the RF output.

Via the front panel:

1. Set the carrier frequency to 2.5 GHz.
2. Set the carrier amplitude to −10.0 dBm.
3. Modulate the carrier.
4. Activate the RF output.

Via the remote interface:

```
[:SOURce]:FREQuency:FIXed 2.5GHZ
[:SOURce]:POWer[:LEVel][:IMMediate][:AMPLitude] -10.0DBM
:OUTPut:MODulation[:STATe] ON
:OUTPut[:STATe] ON
```

FIR Filter Coefficient Downloads

NOTE This feature is available only in E4438C ESG Vector Signal Generators with Option 001/601 or 002/602.

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

There are two requirements for user-defined FIR filter coefficient files:

1. Data must be in ASCII format.

   The signal generator processes FIR filter coefficients as floating point numbers.

2. 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 285.

Data Limitations

Filter lengths of up to 1024 taps (coefficients) are allowed. 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 Real Time I/Q Baseband FIR filter files are limited to 1024 taps, 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 sampling period (Δt) 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, the GSM symbol rate is 270.83 ksps. With an oversample ratio of 4, the sampling rate is 1083.32 kHz and Δt (inverse of FS) is 923.088 nsec.

Downloading FIR Filter Coefficient Data

The ESG stores the FIR files in the FIR (/USER/FIR) directory, which utilizes non-volatile memory. 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 will download a typical set of FIR filter coefficient values and name 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, 0.000001,...`	represent FIR filter coefficients.

Selecting a Downloaded User FIR Filter as the Active Filter

FIR Filter Data for TDMA Format

The following front panel key presses or remote commands will select user FIR filter data as the active filter for a TDMA modulation format.

Via the front panel:

1. Press **Mode** > **Real Time TDMA** > ***desired format*** > **Modify Standard** > **Filter** > **Select** > **User FIR**
2. Highlight the desired file in the catalog of FIR files.
3. Press **Select File**.

To activate the TDMA format press **Mode** > **Real Time TDMA** > ***desired format*** and toggle the format on.

Via the remote interface:

```
[:SOURce]:RADio:<desired format>:FILTer "<file_name>"
```

This command selects the user FIR filter, specified by the file name, as the active filter for the TDMA modulation format. After selecting the file, activate the TDMA format with the following command:

```
[:SOURce]:RADio:<desired format>[:STATe] On
```

FIR Filter Data for Custom Modulation

The following front panel key presses or remote commands will select user FIR filter data as the active filter for a custom modulation format.

Via the front panel:

1. Press **Mode** > **Custom** > **Real Time IQ Baseband > Filter** > **Select** > **User FIR**
2. Highlight the desired file in the catalog of FIR files.
3. Press **Select File**.

To activate the custom modulation, press **Mode** > **Custom** > **Real Time IQ Baseband > Custom Off On** and toggle to on.

Via the remote interface:

```
[:SOURce]:RADio:CUSTom:FILTer "<file_name>"
```

This command selects the user FIR filter, specified by the file name, as the active filter for the custom modulation format. After selecting the file, activate the TDMA format with the following command:

```
[:SOURce]:RADio:CUSTom[:STATe] On
```

FIR Filter Data for CDMA and W-CDMA Modulation

The following front panel key presses or remote commands will select user FIR filter data as the active filter for a CDMA modulation format. The process is very similar for W-CDMA.

Via the front panel:

1. Press **Mode** > **CDMA** > **Arb IS-95A > CDMA Define** > **Filter** > **Select** > **User FIR**
2. Highlight the desired file in the catalog of FIR files.
3. Press **Select File**.

To activate the CDMA modulation, press **Mode** > **CDMA** > **Arb IS-95A > CDMA Off On** to On.

Via the remote interface:

```
[:SOURce]:RADio:<desired format>:ARB:FILTer "<file_name>"
```

This command selects the User FIR filter, specified by the file name, as the active filter for the CDMA or W-CDMA modulation format. After selecting the file, activate the CDMA or W-CDMA format with the following command:

```
[:SOURce]:RADio:<desired format>:ARB[:STATe] On
```

Modulating and Activating the Carrier

The following front panel key presses or remote commands will set the carrier frequency, power, turn on the modulation, and turn on the RF output.

Via the front panel:

1. Press **Frequency** > **2.5** > **GHz**. Sets the signal generator frequency to 2.5 Ghz.
2. Press **Amplitude** > **–10** > **dBm**. Sets the signal generator power to –10 dBm.
3. Press **Mod On/Off** until the display annunciator reads `MOD ON`.
4. 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
   ```

Downloads Directly into Pattern RAM (PRAM)

NOTE This feature is available only in E4438C ESG Vector Signal Generators with Option 001/601 or 002/602.

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 such as MATLAB or MathCad can generate data which can be downloaded directly into PRAM in either a list format or a block format.

Direct downloads to PRAM provides complete control over bursting, which is especially helpful for designing experimental or proprietary framing schemes.

This section contains information that will help you transfer user-generated data from a system controller to the signal generator’s PRAM. It explains how to download data directly into PRAM and modulate the carrier signal with the data.

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.) by the front panel interface or by remote-command interface.

Data Limitations

Total (data bits plus control bits) download size limitations to PRAM (volatile memory) are 8 MB with Option 001/601, 32 MB with Option 002, and 64 MB with Option 602. However the signal generator shares this memory with other file types, so the actual available memory varies depending on the files currently residing in volatile memory. Each downloaded byte for PRAM uses 4 bytes of storage.

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) actually utilize the same storage media.

A data PRAM 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.

The signal generator provides two data storage areas: volatile waveform memory (WFM1) and non-volatile memory (NVWFM). Data stored in volatile waveform memory cannot be recovered if it is overwritten or if the power is cycled. Data stored in non-volatile memory, however, remains until you delete the file. The

Option 005 signal generator’s hard disk provides 5 GB of non-volatile storage. Signal generators without Option 005 provide 15 MB of non-volatile storage. For more information on signal generator memory, see “Waveform Memory” on page 184.

Downloading in List Format

Because of parsing, list data format downloads are *significantly* slower than block format downloads.

Data Requirements and Limitations Summary

1. Data must be 8-bit unsigned integers, from 0 to 255.

   This requirement is necessary as list format downloads are parsed prior to being loaded into PRAM.

2. 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 8-bit bytes. Each byte contains 1 bit of data field information, and 7 bits of control information associated with the data field bit.

Table 5-1 PRAM Data Byte

Bit	Function	Value	Comments
0	Data	0/1	This bit is the data to be modulated. This bit is “unspecified” when burst (bit 2) is set to 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 the RF output on continuously. For framed data, this bit is set to 1 for *on* timeslots and 0 for *off* timeslots.
3	Reserved	0	Always 0
4	Reserved	1	Always 1
5	Reserved	0	Always 0
6	Event 1 Output	0/1	Setting this bit to 1 causes a level transition at the EVENT 1 BNC connector. This can be used for many functions. For example, as a marker output to trigger external hardware when the data pattern has restarted, or to create a data-synchronous pulse train by toggling this bit in alternate addresses.
7	Pattern Reset	0/1	Set to 0 = continue to next sequential memory address. Set to 1 = end of memory and restart memory playback. This bit is set to 0 for all bytes except the last address of PRAM. For the last address (byte) of PRAM, it is set to 1 to restart the pattern.

Preliminary Setup

It is important to set up the digital communications format before downloading data. This allows the signal generator to define the modulation format, filter, and data clock.

CAUTION Activating the digital communications format after the data has been downloaded to PRAM may corrupt the downloaded data.

Via the front panel:

To set up the TDMA format, press **Mode** > ***desired format*** and toggle the format on.

To adjust symbol rate, filtering, or other parameters, press the appropriate softkey and adjust the value.

To set up the custom modulation format, press **Mode** > **Custom** and toggle the format on.

Via the remote interface:

For TDMA formats, send the following SCPI commands:

```
[:SOURce]:RADio:<desired format>[:STATe] ON
[:SOURce]:RADio:<desired format>:BURSt[:STATe] ON
[:SOURce]:BURSt:SOURce INT
```

For custom modulation, send:

```
[:SOURce]:RADio:CUSTOm[:STATe] ON
```

To adjust symbol rate, filtering, or other parameters, send the appropriate SCPI command.

SCPI Command to Download Data in List Format

```
:MEMory:DATA:PRAM:FILE:LIST "<file_name>",<uint8>[,<uint8>,<...>]
```

This command downloads the list-formatted data directly into PRAM. The variable `<uint8>` is any of the valid 8-bit unsigned integer values between 0 and 255, as specified by Table 5-1 on page 284. Note that each value corresponds to a unique byte/address in PRAM.

Sample Command Line

For example, to burst a FIX4 data pattern of “1100” five times, then turn the burst off for 32 data periods (assuming a 1-bit/symbol modulation format), the command is:

```
:MEMory:DATA:PRAM:FILE:LIST "<newFile>",85,21,21,20,20,21,21,20,20,21,21,
20,20,21,21,20,20,21,21,20,20,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,
16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,144
```

`newFile`	name of the PRAM file as it will appear in waveform memory

`85`	enables event 1 trigger signifying the beginning of the data pattern
`21`	signifies data = 1, burst = on (1)
`20`	signifies data = 0, burst = on (1)
`16`	signifies data = unspecified, burst = off (0)
`144`	signifies data = unspecified, burst = off (0), pattern repeat = on (1)

Downloading in Block Format

NOTE Because there is no parsing involved, block data format downloads are *significantly* faster than list format downloads.

Data Requirements and Limitations Summary

1. Data must be in binary form.

   This requirement is necessary as the baseband generator reads binary data from the data generator.

2. 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 8-bit bytes. Each byte contains 1-bit of data field information, and 7-bits of control information associated with the data field bit. See Table 5-1 on page 284 for the required data and control bits.

Because a waveform containing 16 megabits of data for subsequent modulation must also contain another 112 megabits of control information, a file this size (16 MB) requires a signal generator with Option 002 (32 MB) or 602 (64 MB). The largest amount of data (modulation data and control data) for a waveform in an Option 001/601 signal generator is approximately 8 megabits, which provides only enough memory for 56 megabits of control data (64 megabits = 8 MB of memory).

Preliminary Setup

It is important to set up the digital communications format before downloading data. This allows the signal generator to define the modulation format, filter, and data clock.

CAUTION Activating the digital communications format after the data has been downloaded to PRAM may corrupt the downloaded data.

Via the front panel:

To set up the TDMA format, press **Mode** > ***desired format*** and toggle the format on.

To set up a custom modulation format, press **Mode** > **Custom** and toggle the format on.

To adjust symbol rate, filtering, or other parameters, press the appropriate softkey and adjust the value.

Via the remote interface:

For TDMA formats, send the following SCPI command:

```
[:SOURce]:RADio:<desired format>[:STATe] ON
```

For custom modulation, send:

```
[:SOURce]:RADio:CUSTom[:STATe] ON
```

To adjust symbol rate, filtering, or other parameters, send the appropriate SCPI command.

SCPI Command to Download Data in Block Format

```
:MEMory:DATA:PRAM:FILE:BLOCk "<filename>",<datablock>
```

This command downloads the block-formatted data directly into pattern RAM. In the following sample command line, the datablock is designated as #ABC.

Sample Command Line

```
:MEMory:DATA:PRAM:FILE:BLOCk "<file_name>",#ABC
```

`<file_name>`	name of the PRAM file as it will appear in waveform 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 1

```
:MEMory:DATA:PRAM:FILE:BLOCk "<new_File>",#2161@2S@g4u&07!89*7
```

`<new_File>`	name of the PRAM file as it will appear in waveform memory
`#`	indicates the start of the data block
`2`	defines the number of decimal digits to follow in “B”.
`16`	denotes how many bytes of data are to follow.

`1@2S@g4u&07!89*7`	the ASCII representation of the binary data that is 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 the PRAM Waveform

After the waveform data is written to PRAM, the data pattern can be viewed using an oscilloscope. There is approximately a 12-symbol delay 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 *E4428C/38C ESG Signal Generators User's Guide* and *E4428C/38C ESG 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 *E4428C/38C ESG 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
```

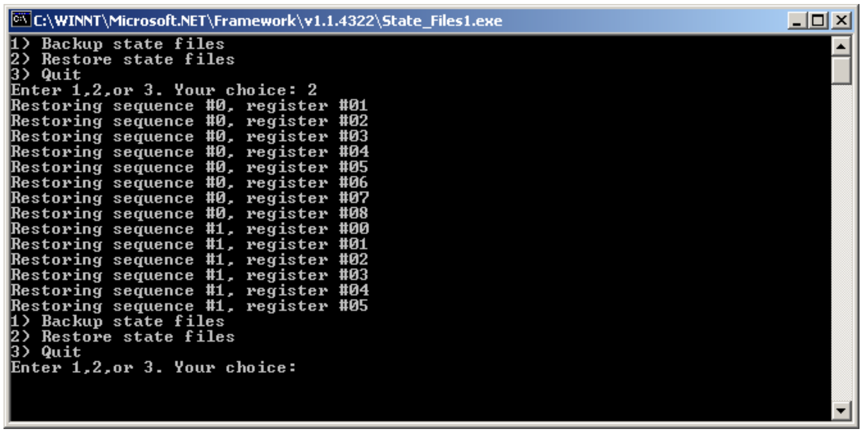
Save and Recall Programming Example

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.



```
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

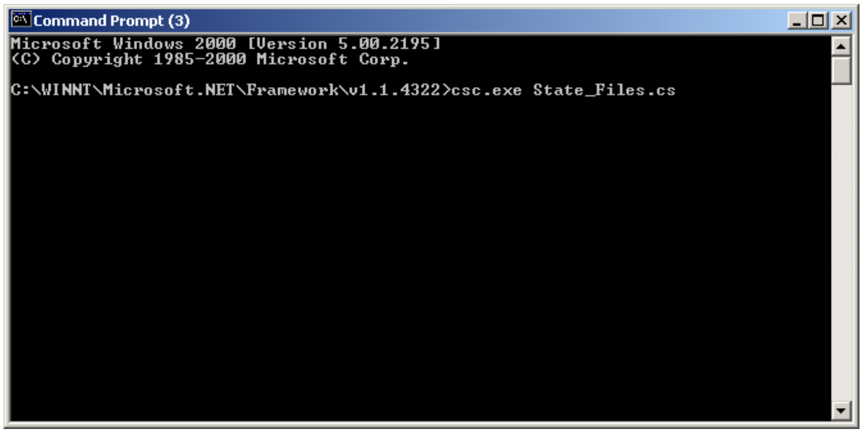
The Microsoft .NET Framework is a platform for creating Web Services and applications. There are three components of the .NET Framework: the common language runtime, class libraries, and Active Server Pages, called ASP.NET. Refer to the Microsoft website for more information on the .NET Framework.

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 ESG'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.



```
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\WINNT\Microsoft.NET\Framework\v1.1.4322>csc.exe State_Files.cs
```

The State_Files.cs program is listed below. You can copy this program from the examples directory on the ESG CD–ROM E4400–90501.

```
//****************************************************************************************
// FileName: State_Files.cs
//
// This C# example code saves and recalls signal generator instrument states. The saved
// instrument state files are written to the local computer directory computer where the
// State_Files.exe is located. This is a console application that uses DLL importing to
// allow for calls to the unmanaged Agilent IO Library VISA DLL.
//
// The Agilent VISA library must be installed on your computer for this example to run.
// Important: Replace the visaOpenString with the IP address for your signal generator.
//
//****************************************************************************************
```

```
using System;
using System.IO;
using System.Text;
using System.Runtime.InteropServices;
using System.Collections;
using System.Text.RegularExpressions;

namespace State_Files

{
    class MainApp
    {
        // Replace the visaOpenString variable with your instrument's address.

        static public string visaOpenString = "TCPIP0::000.000.000.000"; //"GPIB0::19";
      //"TCPIP0::esg3::INSTR";



public const uint DEFAULT_TIMEOUT = 30 * 1000;// Instrument timeout 30 seconds.
        public const int MAX_READ_DEVICE_STRING = 1024; // Buffer for string data reads.
      public const int TRANSFER_BLOCK_SIZE = 4096;// Buffer for byte data.

        // The main entry point for the application.

        [STAThread]

static void Main(string[] args)
        {

    uint defaultRM;// Open the default VISA resource manager
    if (VisaInterop.OpenDefaultRM(out defaultRM) == 0) // If no errors, proceed.
      {
```

```
    uint device;

    // Open the specified VISA device: the signal generator

if (VisaInterop.Open(defaultRM, visaOpenString,VisaAccessMode.NoLock,

                  DEFAULT_TIMEOUT, out device) == 0)

   // if no errors proceed.

  {

  bool quit = false;

    while (!quit)// Get user input
     {

     Console.Write("1) Backup state files\n" +

                   "2) Restore state files\n" +

                   "3) Quit\nEnter 1,2,or 3. Your choice: ");

        string choice = Console.ReadLine();
      switch (choice)

           {

            case "1":
            {
          BackupInstrumentState(device);  // Write instrument state
          break;                   // files to the computer
            }

                       case "2":

             {

           RestoreInstrumentState(device); // Read instrument state

           break;// files to the ESG

             }

         case "3":

             {

          quit = true;

          break;

             }

          default:

             {

          break;

             }
```

```
                  }
                }
             VisaInterop.Close(device);// Close the device
                }
              else
                {
              Console.WriteLine("Unable to open " + visaOpenString);
             }
                 VisaInterop.Close(defaultRM);   // Close the default resource manager
             }
              else
                {
           Console.WriteLine("Unable to open the VISA resource manager");
             }
           }

         /* This method restores all the sequence/register state files located in
          the local directory (identified by a ".STA" file name extension)
          to the signal generator.*/

static public void RestoreInstrumentState(uint device)
   {
    DirectoryInfo di = new DirectoryInfo(".");// Instantiate object class
    FileInfo[] rgFiles = di.GetFiles("*.STA");  // Get the state files
    foreach(FileInfo fi in rgFiles)
      {
      Match m = Regex.Match(fi.Name, @"^(\d)_(\d\d)");
      if (m.Success)
        {
        string sequence = m.Groups[1].ToString();
        string register = m.Groups[2].ToString();
        Console.WriteLine("Restoring sequence #" + sequence +
                                      ", register #" + register);

/* Save the target instrument's current state to the specified sequence/
register pair. This ensures the index file has an entry for the specified
sequence/register pair. This workaround will not be necessary in future
revisions of firmware.*/

        WriteDevice(device,"*SAV " + register + ", " + sequence + "\n",
                                true); // << on SAME line!
        // Overwrite the newly created state file with the state
        // file that is being restored.
        WriteDevice(device, "MEM:DATA \"/USER/STATE/" + m.ToString() + "\",",
                                false); // << on SAME line!
        WriteFileBlock(device, fi.Name);
        WriteDevice(device, "\n", true);
                 }
              }
        }

/* This method reads out all the sequence/register state files from the signal
generator and stores them in your computer's local directory with a ".STA"
extension */

static public void BackupInstrumentState(uint device)
   {
   // Get the memory catalog for the state directory
      WriteDevice(device, "MEM:CAT:STAT?\n", false);
      string catalog = ReadDevice(device);
      /* Match the catalog listing for state files which are named
         (sequence#)_(register#)  e.g.  0_01, 1_01, 2_05*/
      Match m = Regex.Match(catalog, "\"(\\d_\\d\\d),");
      while (m.Success)
```

```
         {
        // Grab the matched filename from the regular expresssion
        string nextFile = m.Groups[1].ToString();
       // Retrieve the file and store with a .STA extension
       // in the current directory
       Console.WriteLine("Retrieving state file: " + nextFile);
         WriteDevice(device, "MEM:DATA? \"/USER/STATE/" + nextFile + "\"\n", true);
       ReadFileBlock(device, nextFile + ".STA");
       // Clear newline
       ReadDevice(device);
       // Advance to next match in catalog string
       m = m.NextMatch();
        }
   }

/*  This method writes an ASCII text string (SCPI command) to the signal generator.
If the bool "sendEnd" is true, the END line character will be sent at the
conclusion of the write. If "sendEnd is false the END line will not be sent.*/

static public void WriteDevice(uint device, string scpiCmd, bool sendEnd)
   {
    byte[] buf = Encoding.ASCII.GetBytes(scpiCmd);
    if (!sendEnd) // Do not send the END line character
      {
     VisaInterop.SetAttribute(device, VisaAttribute.SendEndEnable, 0);
      }
    uint retCount;
    VisaInterop.Write(device, buf, (uint)buf.Length, out retCount);
    if (!sendEnd) // Set the bool sendEnd true.
      {
     VisaInterop.SetAttribute(device, VisaAttribute.SendEndEnable, 1);
      }
   }

// This method reads an ASCII string from the specified device
static public string ReadDevice(uint device)
   {
    string retValue = "";
    byte[] buf = new byte[MAX_READ_DEVICE_STRING]; // 1024 bytes maximum read
    uint retCount;
    if (VisaInterop.Read(device, buf, (uint)buf.Length -1, out retCount) == 0)
      {
      retValue = Encoding.ASCII.GetString(buf, 0, (int)retCount);
      }
    return retValue;
   }

/* The following method reads a SCPI definite block from the signal generator
and writes the contents to a file on your computer. The trailing
newline character is NOT consumed by the read.*/

static public void ReadFileBlock(uint device, string fileName)
   {
    // Create the new, empty data file.
    FileStream fs = new FileStream(fileName, FileMode.Create);
    // Read the definite block header:  #{lengthDataLength}{dataLength}
    uint retCount = 0;
    byte[] buf = new byte[10];
    VisaInterop.Read(device, buf, 2, out retCount);
    VisaInterop.Read(device, buf, (uint)(buf[1]-'0'), out retCount);
     uint fileSize = UInt32.Parse(Encoding.ASCII.GetString(buf, 0, (int)retCount));
    // Read the file block from the signal generator
    byte[] readBuf = new byte[TRANSFER_BLOCK_SIZE];
    uint bytesRemaining = fileSize;
    while (bytesRemaining != 0)
      {
       uint bytesToRead = (bytesRemaining < TRANSFER_BLOCK_SIZE) ?
       bytesRemaining : TRANSFER_BLOCK_SIZE;
       VisaInterop.Read(device, readBuf, bytesToRead, out retCount);
       fs.Write(readBuf, 0, (int)retCount);
       bytesRemaining -= retCount;
      }
    // Done with file
    fs.Close();
   }

/* The following method writes the contents of the specified file to the
specified file in the form of a SCPI definite block.  A newline is
NOT appended to the block and END is not sent at the conclusion of the
write.*/

static public void WriteFileBlock(uint device, string fileName)
   {
    // Make sure that the file exists, otherwise sends a null block
    if (File.Exists(fileName))
     {
     FileStream fs = new FileStream(fileName, FileMode.Open);
     // Send the definite block header: #{lengthDataLength}{dataLength}
     string fileSize = fs.Length.ToString();
     string fileSizeLength = fileSize.Length.ToString();
     WriteDevice(device, "#" + fileSizeLength + fileSize, false);
     // Don't set END at the end of writes
     VisaInterop.SetAttribute(device, VisaAttribute.SendEndEnable, 0);
     // Write the file block to the signal generator
     byte[] readBuf = new byte[TRANSFER_BLOCK_SIZE];
     int numRead = 0;
     uint retCount = 0;
     while ((numRead = fs.Read(readBuf, 0, TRANSFER_BLOCK_SIZE)) != 0)
        {
        VisaInterop.Write(device, readBuf, (uint)numRead, out retCount);
        }
       // Go ahead and set END on writes
     VisaInterop.SetAttribute(device, VisaAttribute.SendEndEnable, 1);
     // Done with file
     fs.Close();
     }
     else
     {
     // Send an empty definite block
     WriteDevice(device, "#10", false);
     }
    }
   }

// Declaration of VISA device access constants
public enum VisaAccessMode
    {
        NoLock = 0,
        ExclusiveLock = 1,
        SharedLock = 2,
        LoadConfig = 4
    }

// Declaration of VISA attribute constants
public enum VisaAttribute
    {
        SendEndEnable = 0x3FFF0016,
        TimeoutValue  = 0x3FFF001A
    }

// This class provides a way to call the unmanaged Agilent IO Library VISA C
// functions from the C# application

public class VisaInterop
    {
        [DllImport("agvisa32.dll", EntryPoint="viClear")]
        public static extern int Clear(uint session);

        [DllImport("agvisa32.dll", EntryPoint="viClose")]
        public static extern int Close(uint session);

        [DllImport("agvisa32.dll", EntryPoint="viFindNext")]
        public static extern int FindNext(uint findList, byte[] desc);

        [DllImport("agvisa32.dll", EntryPoint="viFindRsrc")]
        public static extern int FindRsrc(
            uint session,
            string expr,
            out uint findList,
            out uint retCnt,
            byte[] desc);

        [DllImport("agvisa32.dll", EntryPoint="viGetAttribute")]
public static extern int GetAttribute(uint vi, VisaAttribute attribute, out uint
attrState);

        [DllImport("agvisa32.dll", EntryPoint="viOpen")]
        public static extern int Open(
            uint session,
            string rsrcName,
            VisaAccessMode accessMode,
            uint timeout,
            out uint vi);

        [DllImport("agvisa32.dll", EntryPoint="viOpenDefaultRM")]
        public static extern int OpenDefaultRM(out uint session);

        [DllImport("agvisa32.dll", EntryPoint="viRead")]
        public static extern int Read(
            uint session,
            byte[] buf,
            uint count,
            out uint retCount);

        [DllImport("agvisa32.dll", EntryPoint="viSetAttribute")]
public static extern int SetAttribute(uint vi, VisaAttribute attribute, uint attrState);

        [DllImport("agvisa32.dll", EntryPoint="viStatusDesc")]
        public static extern int StatusDesc(uint vi, int status, byte[] desc);

        [DllImport("agvisa32.dll", EntryPoint="viWrite")]
        public static extern int Write(
            uint session,
            byte[] buf,
            uint count,
            out uint retCount);
    }
}
```

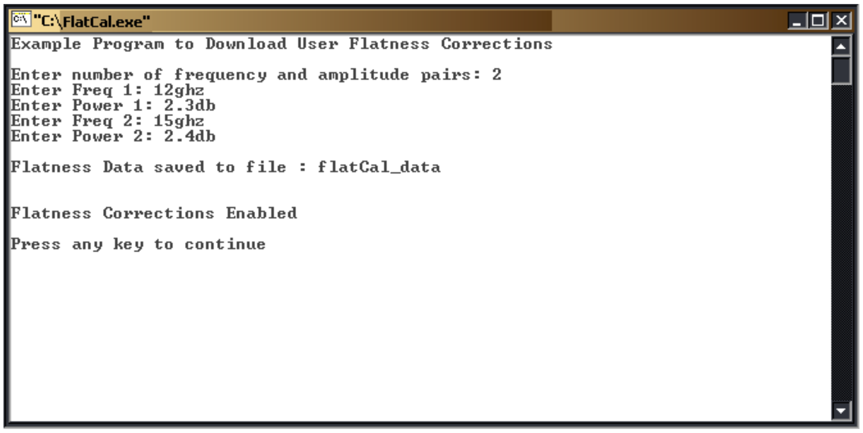
Download User Flatness Corrections Using C++ and VISA

This sample program uses C++ and the VISA libraries to download user–flatness correction values to the signal generator. The program uses the LAN interface but can be adapted to use the GPIB interface by changing the address string in the program.

You must include header files and resource files for library functions needed to run this program. Refer to "Running C/C++ Programming Examples" on page 39 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 5-4 FlatCal Console Application



```
Example Program to Download User Flatness Corrections

Enter number of frequency and amplitude pairs: 2
Enter Freq 1: 12ghz
Enter Power 1: 2.3db
Enter Freq 2: 15ghz
Enter Power 2: 2.4db

Flatness Data saved to file : flatCal_data


Flatness Corrections Enabled

Press any key to continue
```

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:

Each section contains the following troubleshooting information:

- a list of symptoms and possible causes of typical problems encountered while downloading data to the signal generator
- reminders regarding special considerations, file requirements, and data limitations
- tips on creating data, transferring data, data application and memory usage

User File Download Problems

Table 5-2 User FIR File Download Trouble - Symptoms and Causes

Symptom	Possible Cause
No data modulated	Not enough data to fill a single timeslot. If a user file does not completely fill a single timeslot, the firmware will not load any data into the timeslot. For example, if a timeslot’s data field should contain 114 bits, and only 100 bits are provided in the user file, no data will be loaded into the data field of the timeslot. Therefore, no data will be detected at the RF output.
At RF output, some data modulated, some data missing	Data does not completely fill an integer number of timeslots. If a user file fills the data fields of more than one timeslot in a continuously repeating framed transmission, the user file will be restarted after the last timeslot containing completely filled data fields. For example, if the user file contains enough data to fill the data fields of 3.5 timeslots, firmware will load 3 timeslots with data and restart the user file after the third timeslot. The last 0.5 timeslot worth of data will never be modulated.

Data Requirement Reminders

To avoid user-file data download problems, the following conditions *must* be met:

1. The user file selected must entirely fill the data field of each timeslot.
2. For binary memory downloads, the user file must be a multiple of 8 bits, so that it can be represented in ASCII characters.
3. Available PRAM must be large enough to support both the data field bits and the framing bits.

Requirement for Continuous User File Data Transmission

“Full Data Field” Requirements

If a user file does not *completely* fill a single timeslot, the firmware does not load *any* data into that timeslot. For example, if a timeslot’s data field should contain 114 bits, and only 100 bits are provided in the user file, no data is loaded into the timeslot data field, and no data is transmitted at the RF output.

To solve this problem, add bits to the user file until it completely fills the data field of the active protocol.

“Integer Number of Timeslots” Requirement for Multiple-Timeslots

If a user file fills the data fields of more than one timeslot in a continuously repeating framed transmission, the user file is restarted after the last timeslot containing completely filled data fields. For example, if the user file contains enough data to fill the data fields of 3.5 timeslots, firmware loads 3 timeslots with data and restart the user file after the third timeslot. The last 0.5 timeslot worth of data is never modulated.

To solve this problem, add or subtract bits from the user file until it completely fills an integer number of timeslots

“Multiple-of-8-Bits” Requirement

For downloads to binary memory, user file data must be downloaded in multiples of 8 bits, since SCPI specifies data in 8-bit bytes. Therefore, if the original data pattern’s length is not a multiple of 8, you may need to:

- Add additional bits to complete the ASCII character
- replicate the data pattern to generate a continuously repeating pattern with no discontinuity
- truncate the remaining bits

NOTE The “multiple-of-8-bits” data length requirement (for binary memory downloads) is in *addition* to the requirement of completely filling the data field of an integer number of timeslots.

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. Additionally, the external data can be multiplexed into internally generated framing

User FIR Filter Coefficient File Download Problems

Table 5-3 User FIR File Download Trouble - Symptoms and Causes

Symptom	Possible Cause
ERROR -321, Out of memory	There is not enough memory available for the FIR coefficient file being downloaded. To solve the problem, either reduce the file size of the FIR file or delete unnecessary files from memory.
ERROR -223, Too much data	User FIR filter has too many symbols. 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.

Data Requirement Reminders

To avoid user FIR filter coefficient data download problems, the following conditions *must* be met:

1. Data must be in ASCII format.
2. Downloads must be in list format.
3. Filters containing more symbols than the hardware allows (64 for Real Time and 512 for ARB) will not be selectable for that configuration.

Direct PRAM Download Problems

Table 5-4 Direct-to-PRAM Download Trouble - Symptoms and Causes

Symptom	Possible Cause
The transmitted pattern is interspersed with random, unwanted data.	Pattern reset bit not set. Insure that the pattern reset bit (bit 7, value 128) is set on the last byte of your downloaded data.
ERROR -223, Too much data	PRAM download exceeds the size of PRAM memory. Either use a smaller pattern or get more memory by ordering the appropriate hardware option.

Data Requirement Reminders

To avoid direct-download-to-PRAM problems, the following conditions *must* be met:

1. The data must be in binary form.
2. For every bit of modulation data (bit 0), you must provide 7 bits of control information (bits 1-7).

Table 5-5 PRAM Byte Information

Bit	Function	Value	Comments
0	Data	0/1	This bit is the data to be modulated. This bit is “unspecified” when burst (bit 2) is set to 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 the RF output on continuously. For framed data, this bit is set to 1 for *on* timeslots and 0 for *off* timeslots.
3	Reserved	0	Always 0
4	Reserved	1	Always 1
5	Reserved	0	Always 0

Table 5-5 PRAM Byte Information

Bit	Function	Value	Comments
6	Event 1 Output	0/1	Setting this bit to 1 causes a level transition at the EVENT 1 BNC connector. This can be used for many functions. For example, as a marker output to trigger external hardware when the data pattern has restarted, or to create a data-synchronous pulse train by toggling this bit in alternate addresses.
7	Pattern Reset	0/1	Set to 0 = continue to next sequential memory address. Set to 1 = end of memory and restart memory playback. This bit is set to 0 for all bytes except the last address of PRAM. For the last address (byte) of PRAM, it is set to 1 to restart the pattern.

Index

Index