# Programming Guide

## Agilent Technologies
## ESG Vector Signal Generator

**This guide applies to the signal generator models listed below. 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 this page) with the latest revision, which can be downloaded from the website shown below.**

**E4438C Vector Signal Generator**

***www.agilent.com/find/signalgenerators***


**Part Number: E4400-90505**

**Printed in USA**

**January 2003**

# Notice

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

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

# 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**.

# Contents

# Contents

# 1 Getting Started

This chapter provides the following major sections:

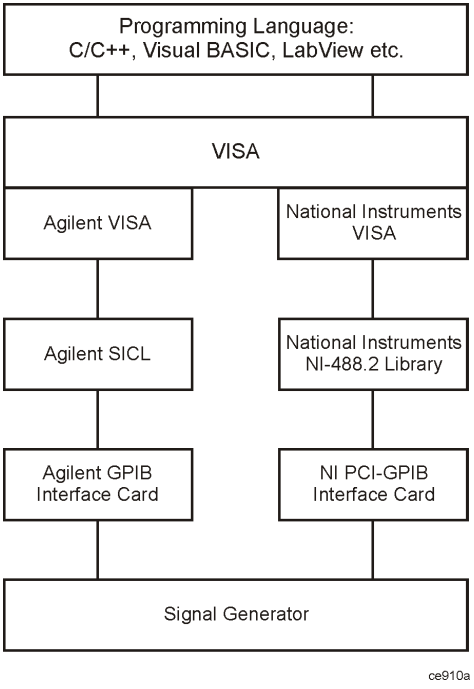
# Introduction to Remote Operation

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**



## 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:

- VMEbus Extensions for Instrumentation (VXI) as defined in VXI-11
- 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. An I/O library must be installed on your computer before writing any programs to control the signal generator.

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

## 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 ®

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 interface cards.

**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. 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.)

NI-VISA is a registered trademark of National Instruments Corporation

**Table 1-4 Agilent GPIB Cables**

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

## 4. Verifying GPIB Functionality

Use the 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.

### 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.

Microsoft is a registered trademark of Microsoft Corporation.

### 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 LAN interface and LAN-connected computer using one of several LAN interface protocols. The LAN allows instruments to be connected together and controlled by a LAN-based computer. LAN and its associated interface operations are defined in the IEEE 802.2 standard. See the IEEE website for more details.

The signal generator supports the following LAN interface protocols:

- VXI-11
- 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 I/O libraries can 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, an IP address must be assigned to the signal generator and the signal generator connected to the LAN. Your system administrator can issue a hostname, IP address, default gateway, and subnet mask for the signal generator.

1. Press **Utility** > **GPIB/RS-232 LAN** > **LAN Setup**.
2. Press **Hostname**.
3. Use the labeled text softkeys and/or numeric keypad to enter the desired hostname.

   To erase the current hostname, press **Editing Keys** > **Clear Text**.
4. Press **Enter**.
5. Press **IP Address** and enter a desired address.

   Use the left and right arrow keys to move the cursor. Use the up and down arrow keys, front panel knob, or numeric keypad to enter an IP address. To erase the current IP address, press the **Clear Text** softkey.

---

**NOTE** To remotely access the signal generator from a different LAN subnet, you must also enter the subnet mask and default gateway. See your system administrator to obtain the appropriate values.

---

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

   This action assigns a hostname and IP address (as well as a gateway and subnet mask, if these have been configured) to the signal generator. The hostname, IP address, gateway and subnet mask are not affected by an instrument preset or by a power cycle.
7. Connect the signal generator to the LAN using a 10BASE-T LAN cable.

## 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 64 10
```

where `hostname` is your instruments name and 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
```

where `hostname` is your instruments name and 10 is the number of echo requests. Type `ping` at the command prompt for details on the ping command.

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.<br><br>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.<br><br>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.<br><br>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.<br><br>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 64 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**

## 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

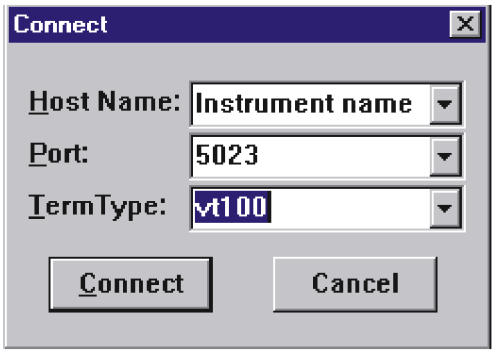
TELNET provides a means of communicating with the signal generator over the LAN. The TELNET client, run on a LAN connected computer, will create a login session on the signal generator. A connection, established between computer and signal generator, generates a user interface display screen with `SCPI>`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

### Using TELNET and MS-DOS Command Prompt

1. On the PC click **Start** > **Programs** > **Command Prompt**.
2. At the command prompt, type in `telnet`.
3. Press enter. The TELNET display screen will be displayed.
4. Click on the **Connect** menu then select **Remote System**. A connection form will be displayed. Refer to Figure 1-3.
5. Enter the hostname, port number, and TermType then click Connect. Refer to Figure 1-3.
   - 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 23.
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.

**Figure 1-3 Connect Form**

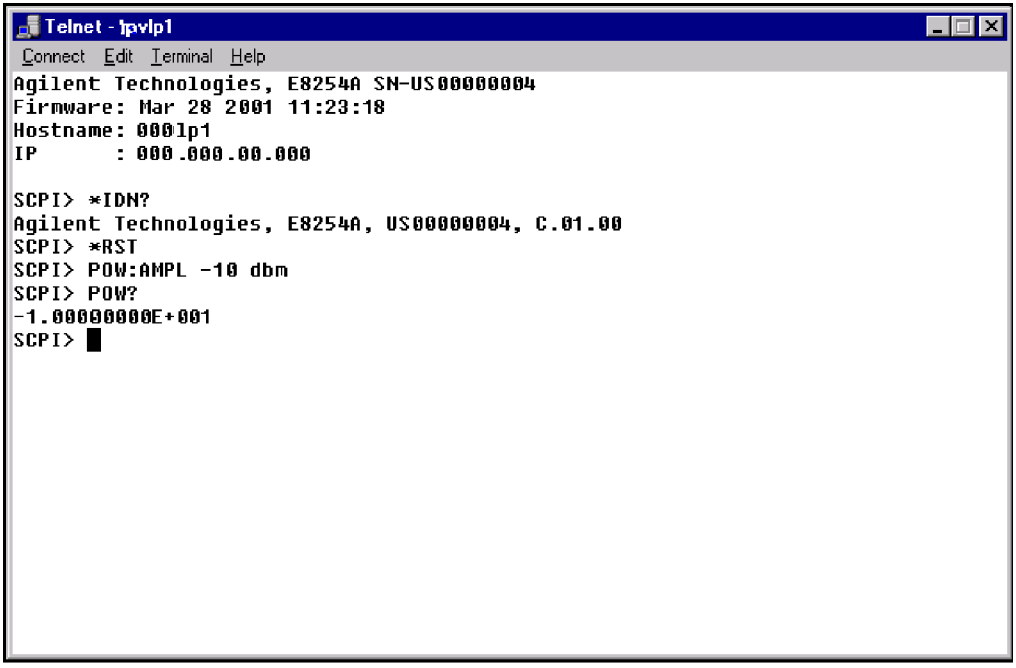



cd923a

### 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 will be displayed. Refer to Figure 1-3.
4. Enter the hostname, port number, and TermType then click Connect. Refer to Figure 1-3.
   - 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.
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**




ce918a

### 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 `5023`, 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:

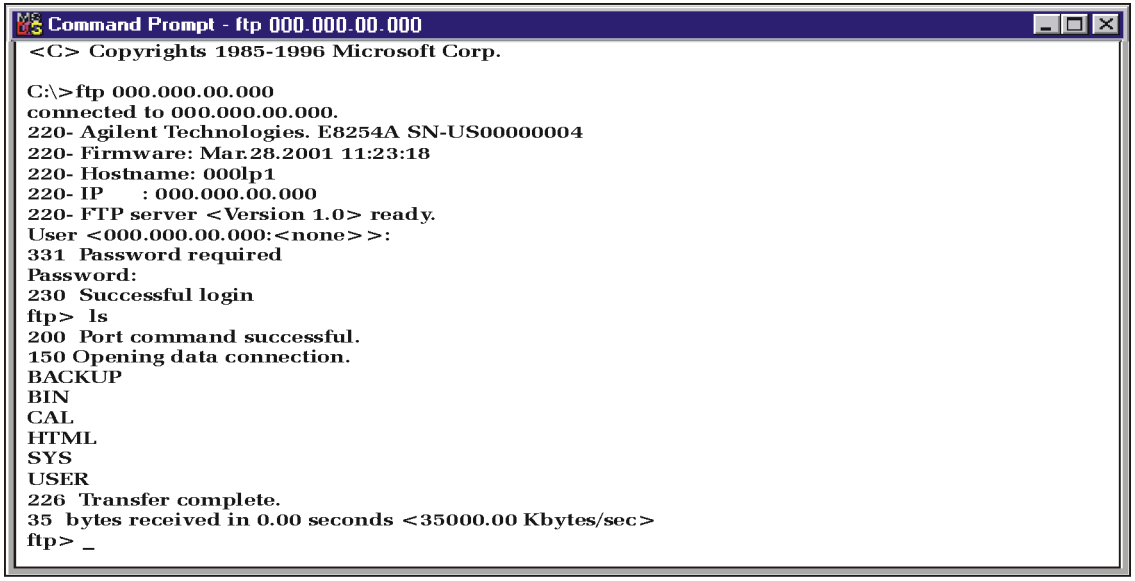
```
$ telnet myinstrument 5023

Trying....

Connected to signal generator

Escape character is '^]'.

Agilent Technologies, E4438C 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 1-5 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 1-5 FTP Screen**




ce917a

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 28. 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.

# 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

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
- Windows NT 4.0 operating system

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

- 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.

# 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 7 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);
}
```

## 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 25 and "Using TELNET LAN" on page 21 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 18 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 72 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.

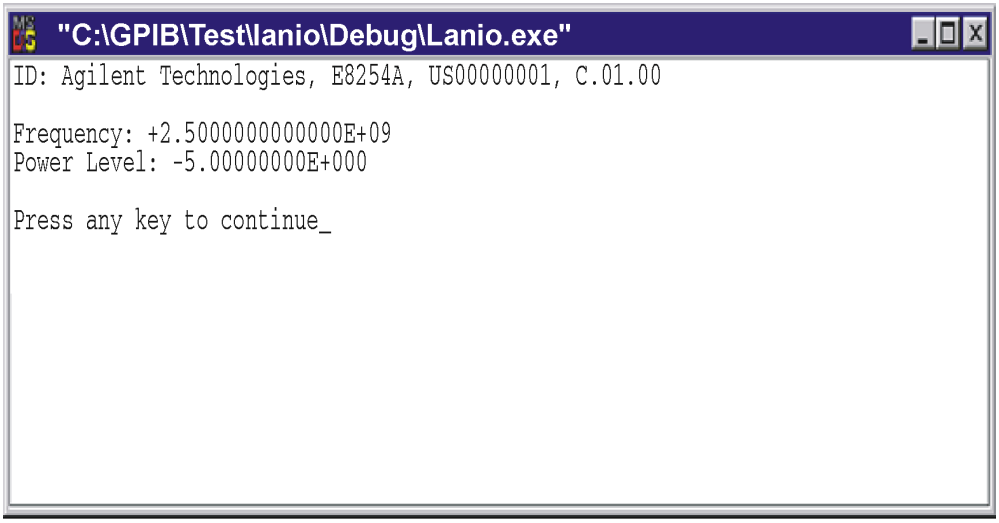
2. Click **Rebuild All** from **Build** menu. Then Click **Execute Lanio.exe**. The Debug window will appear with a prompt "Press any key to continue." This indicates that the program has compiled and can be used to send commands to the signal generator.
3. Click **Start**, click **Programs**, then click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




ce914a

### 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



## 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 27 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 30 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)
{

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

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

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

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

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

# 3 Programming the Status Register System

This chapter provides the following major sections:

# Overview

During remote operation, you may need to monitor the status of the signal generator for error conditions or status changes. The signal generator's error queue can be read with the SCPI query :SYSTem:ERRor? (Refer to ":ERRor[:NEXT]" in the SCPI command reference guide) to see if any errors have occurred. An alternative method uses the signal generator's status register system to monitor error conditions and/or condition changes.

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

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

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

The SCPI register model of the status system has multiple registers that are arranged in a hierarchical order. The lower-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 115 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)**

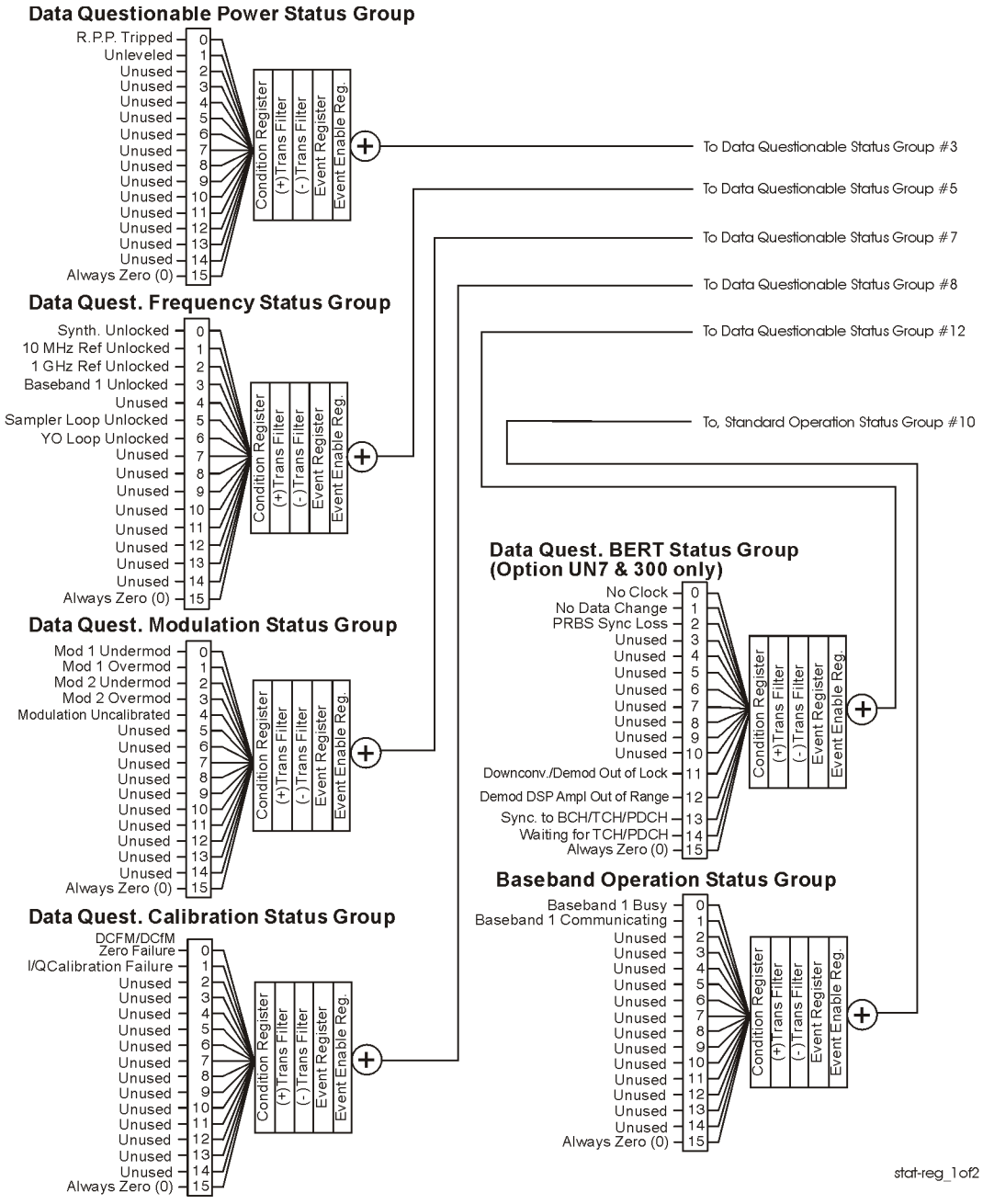
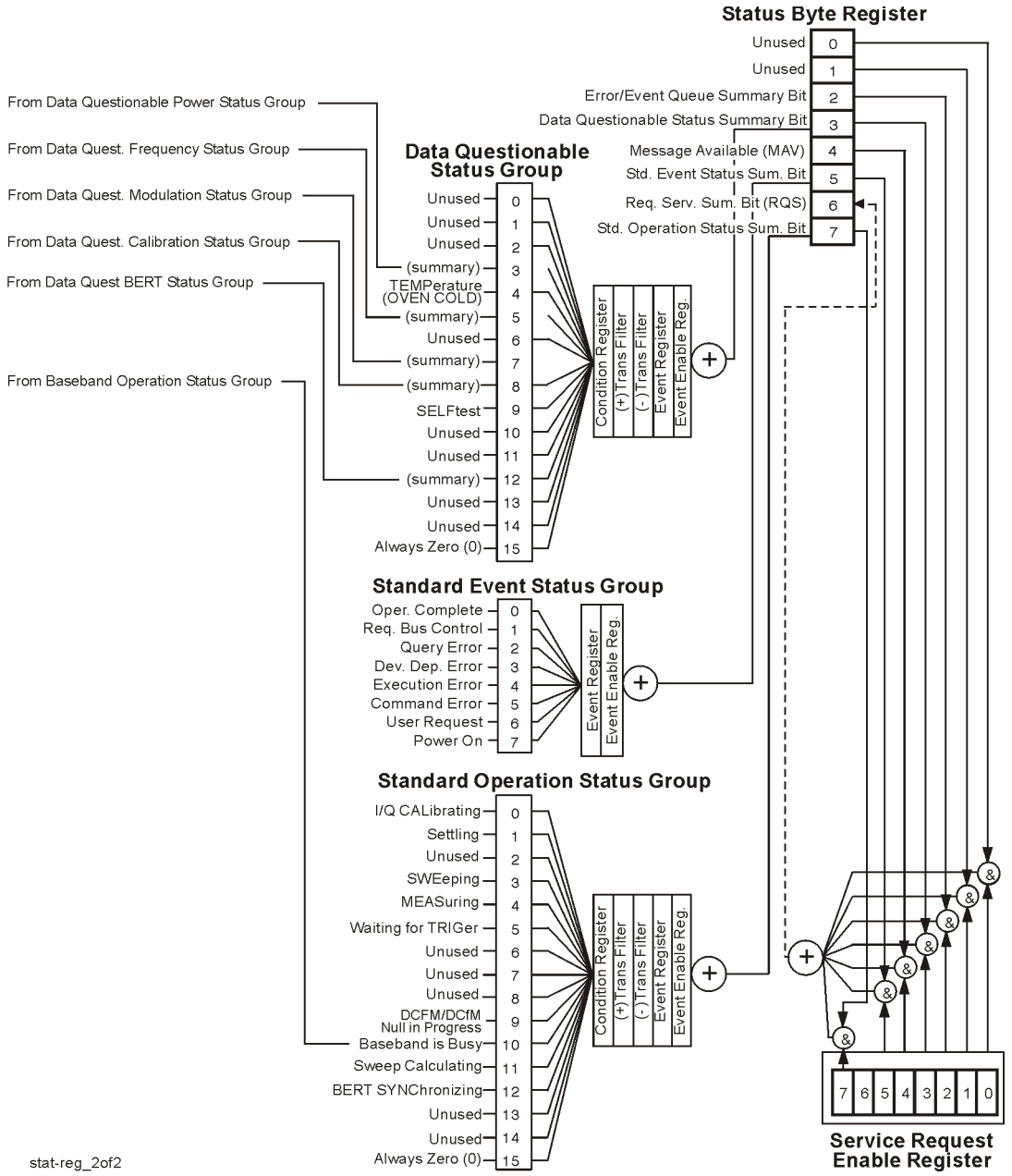


**Figure 3-2 The Overall Status Byte Register System (2 of 2)**



# 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 103 or Figure 3-2 on page 104 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 103 or Figure 3-2 on page 104 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.

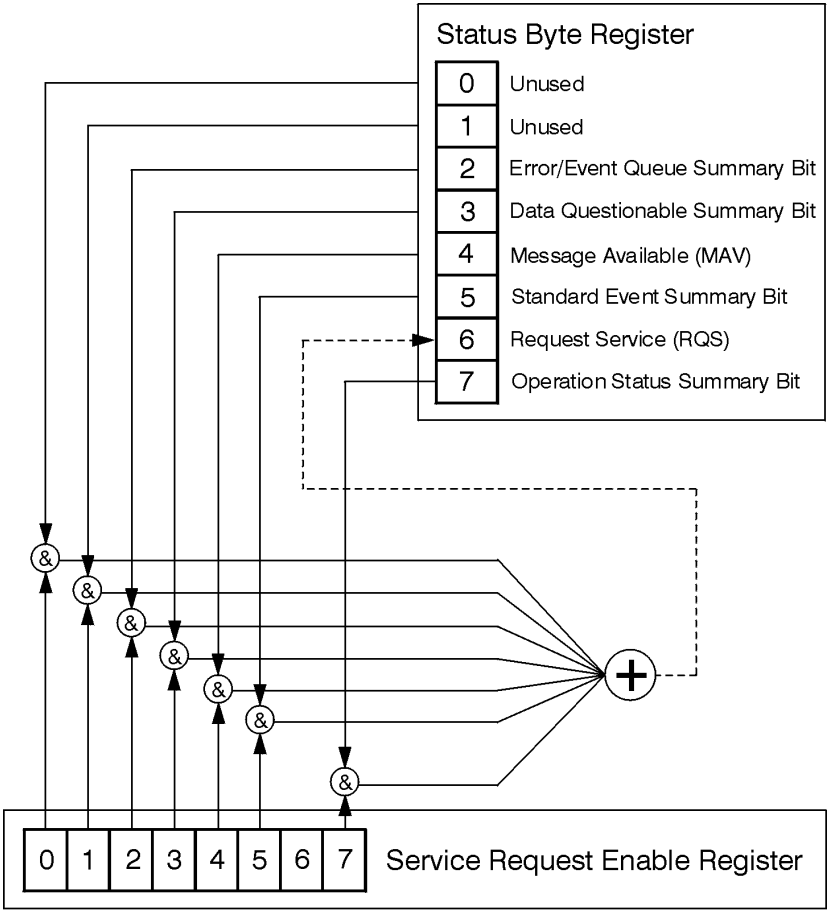
:STATus:PRESet presets all transition filters, non-IEEE 488.2 enable registers, and error/event queue enable registers. (Refer to Table 3-2.)

**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

**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 103 or Figure 3-2 on page 104. |
| 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

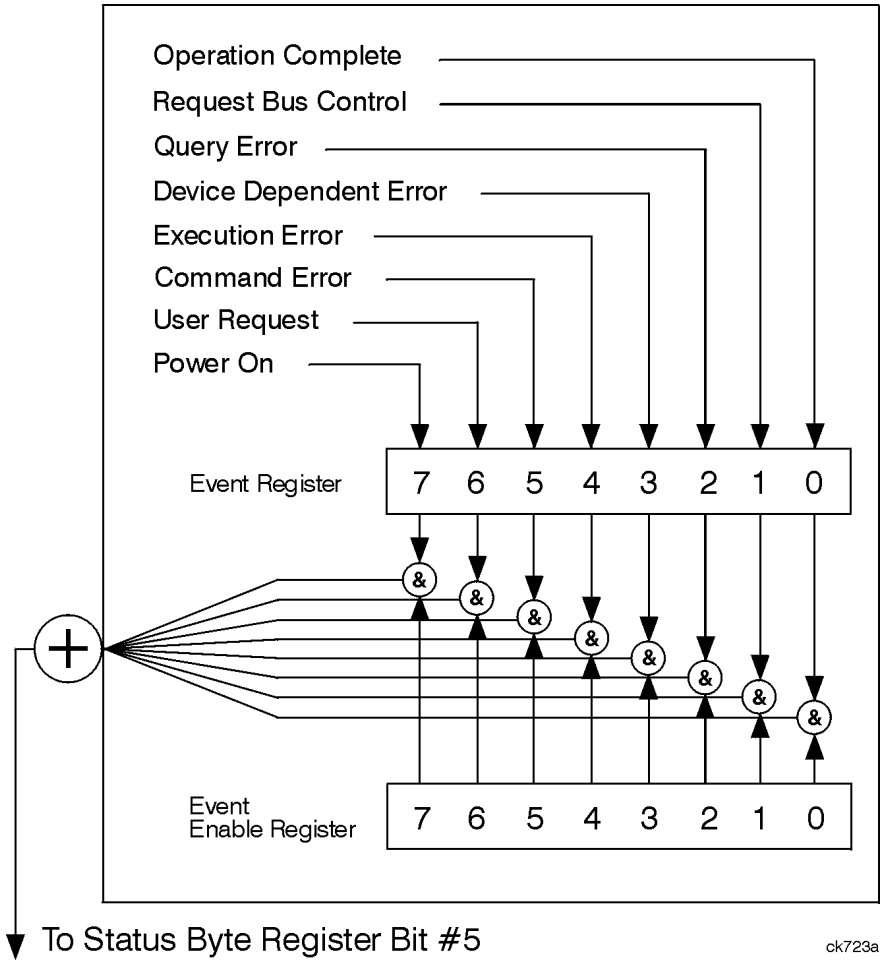
The Standard Operation Status Group and the Data Questionable Status Group consist of the registers listed below. The Standard Event Status Group is similar but does *not* have negative or positive transition filters or a condition register.

| | |
|---|---|
| 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).

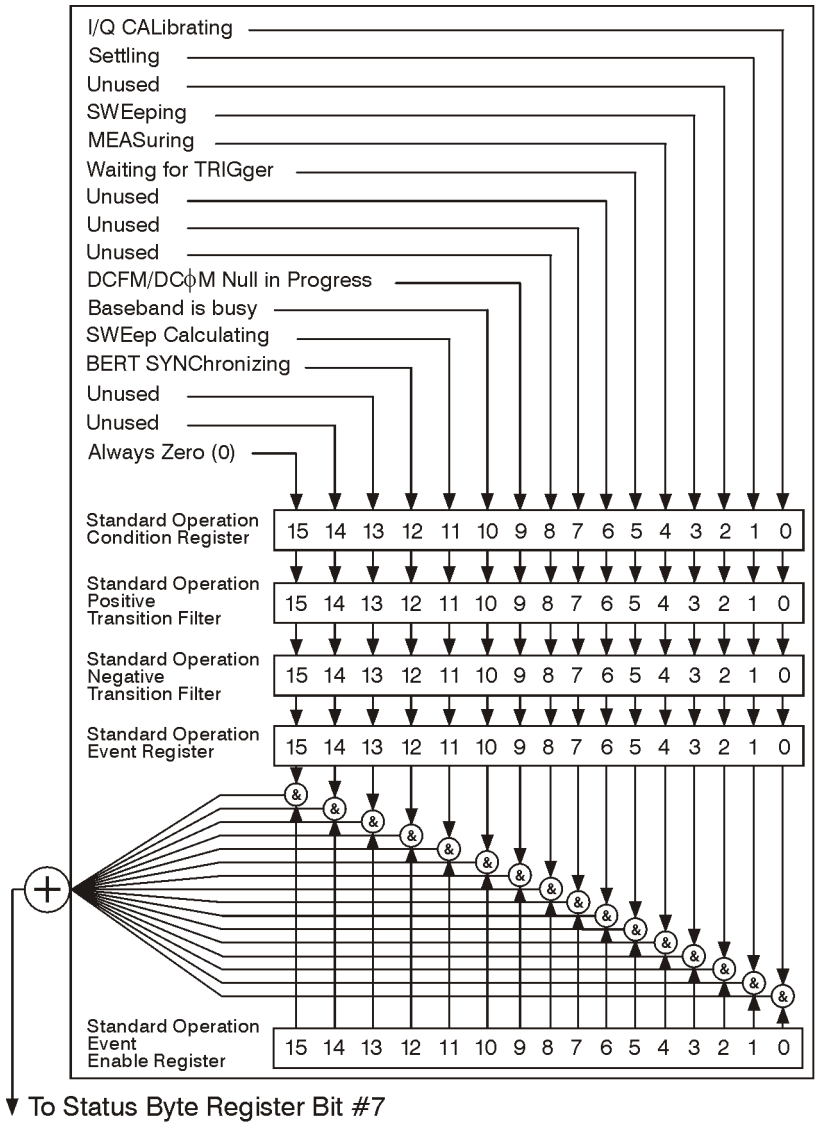
### Standard Event Status Enable Register

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 is set to 1, the Standard Event Status summary bit of the Status Byte Register is set to 1. Send the command `*ESE 192` (128 + 64). |
| Query: | `*ESE?` |
| Response: | Decimal value of the sum of the bits previously enabled with the `*ESE <data>` command. |

## Standard Operation Status Group

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.




ck702c

### 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 | **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 | **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 | **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 122 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 | **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**. |

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.

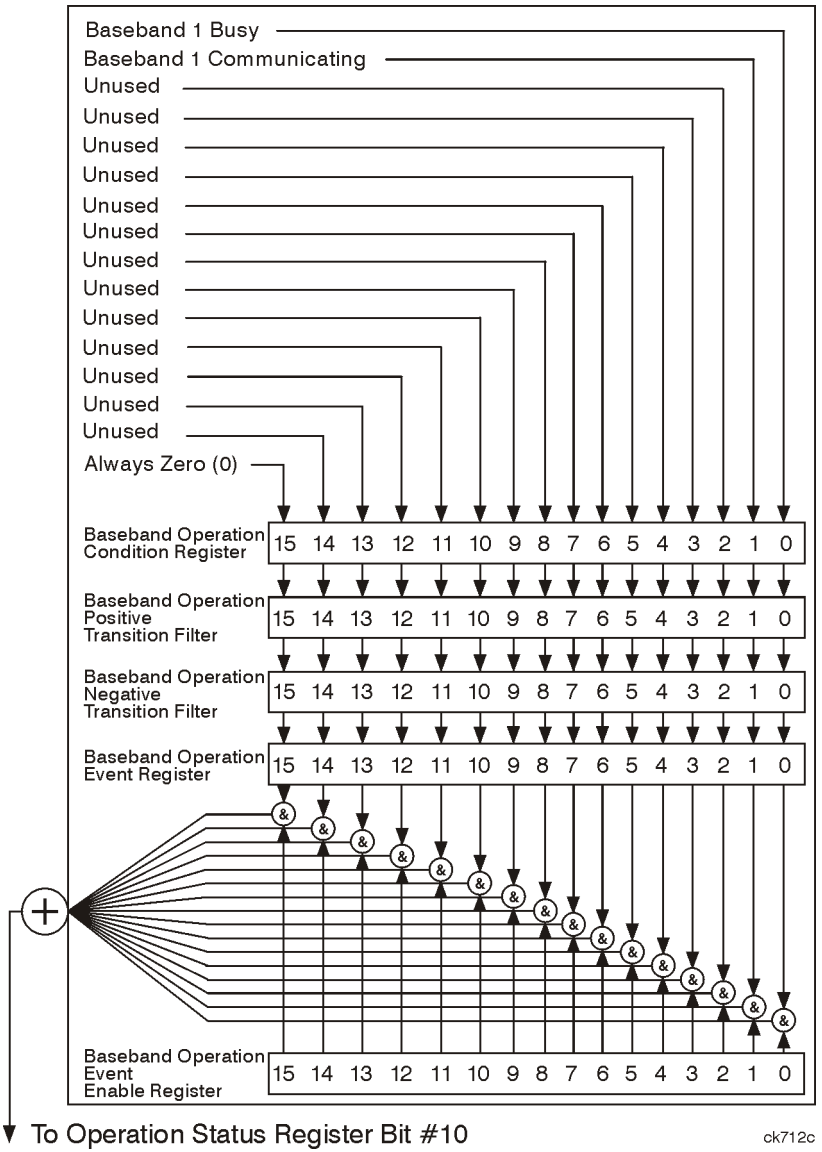
Example: To enable bit 9 and bit 3 so that whenever either of those bits is set to 1, the Standard Operation Status summary bit of the Status Byte Register is set to 1. Send the command `STAT:OPER:ENAB 520` (512 + 8).

Query: `STATus:OPERation:ENABle?`

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

## Baseband Operation Status Group

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



### Baseband Operation Condition Register

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

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

| Bit | Description |
|---|---|
| 0 | **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.

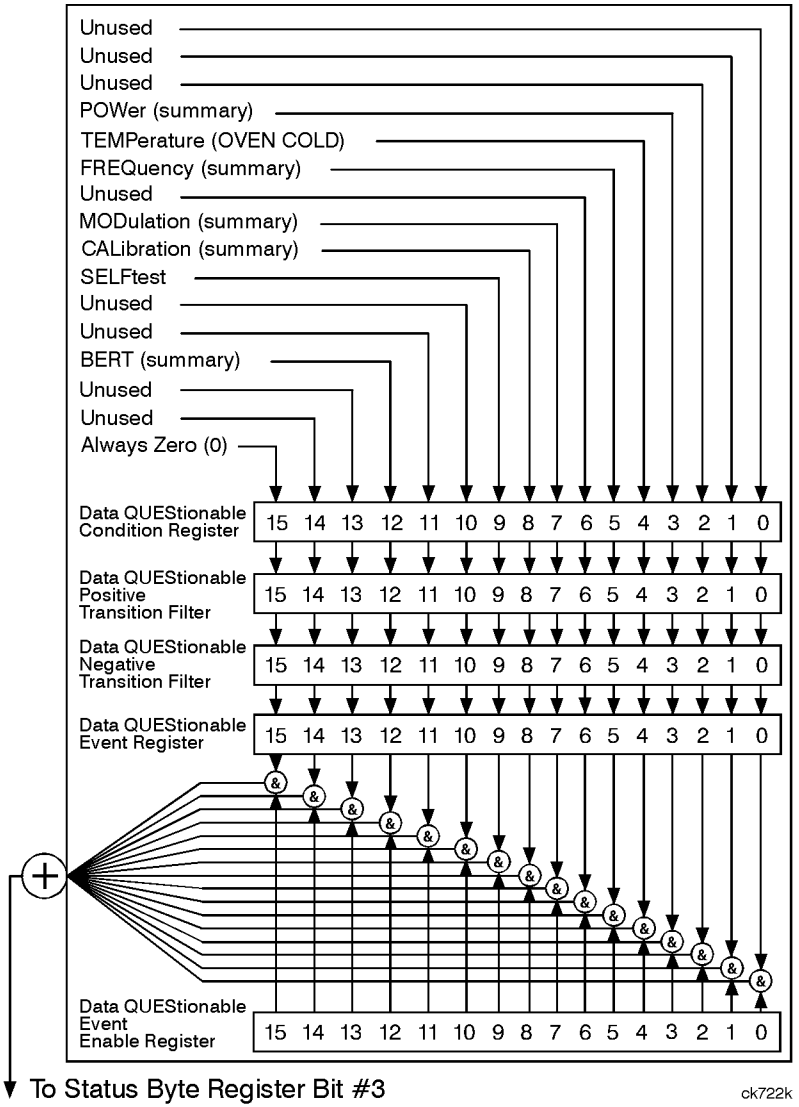
Example: To enable bit 0 and bit 1 so that whenever either of those bits is set to 1, the Baseband Operation Status summary bit of the Status Byte Register is set to 1. Send the command `STAT:OPER:ENAB 520` (512 + 8).

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

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 129 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 132. |
| 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 135 for more information. |
| 8 | **Calibration (summary)**. This is a summary bit taken from the QUEStionable:CALibration register. A 1 in this bit position indicates that 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 138 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. |

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

| Bit | Description |
|---|---|
| 12 | **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 141 for more information. |
| 13, 14 | **Unused**. These bits are set to 0. |
| 15 | **Always 0**. |

Query: `STATus:QUEStionable:CONDition?`

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

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

### Data Questionable Transition Filters (negative and positive)

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

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

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

### Data Questionable Event Register

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

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

### Data Questionable Event Enable Register

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

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

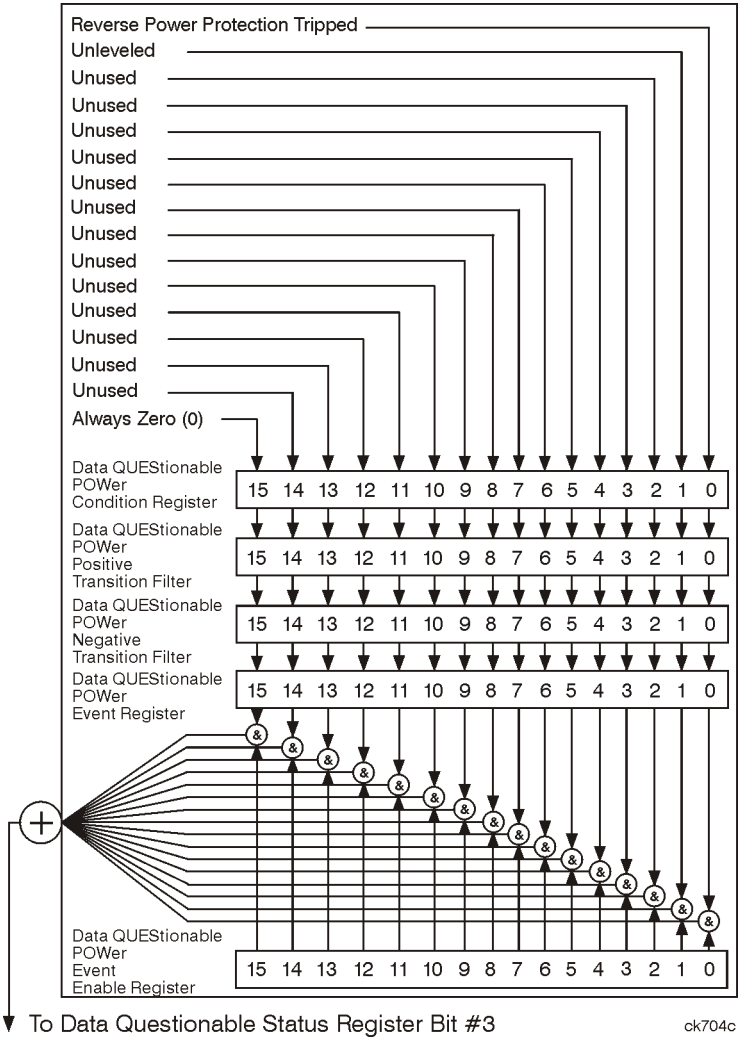
Example: Enable bit 9 and bit 3 so that whenever either of those bits is set to 1, the Data Questionable Status summary bit of the Status Byte Register is set to 1. Send the command `STAT:QUES:ENAB 520` (512 + 8).

Query: `STATus:QUEStionable:ENABle?`

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

## Data Questionable Power Status Group

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



### Data Questionable Power Condition Register

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

**Table 3-8 Data Questionable Power Condition Register Bits**

| Bit | Description |
|---|---|
| 0 | **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. This bit is always set to 0. |
| 1 | **Unleveled**. A 1 in this bit indicates that the output leveling loop is unable to set the output power. |
| 2–14 | **Unused**. These bits are always set to 0. |
| 15 | **Always 0.** |

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

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

### Data Questionable Power Transition Filters (negative and positive)

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

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

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

### Data Questionable Power Event Register

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

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

### Data Questionable Power Event Enable Register

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

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

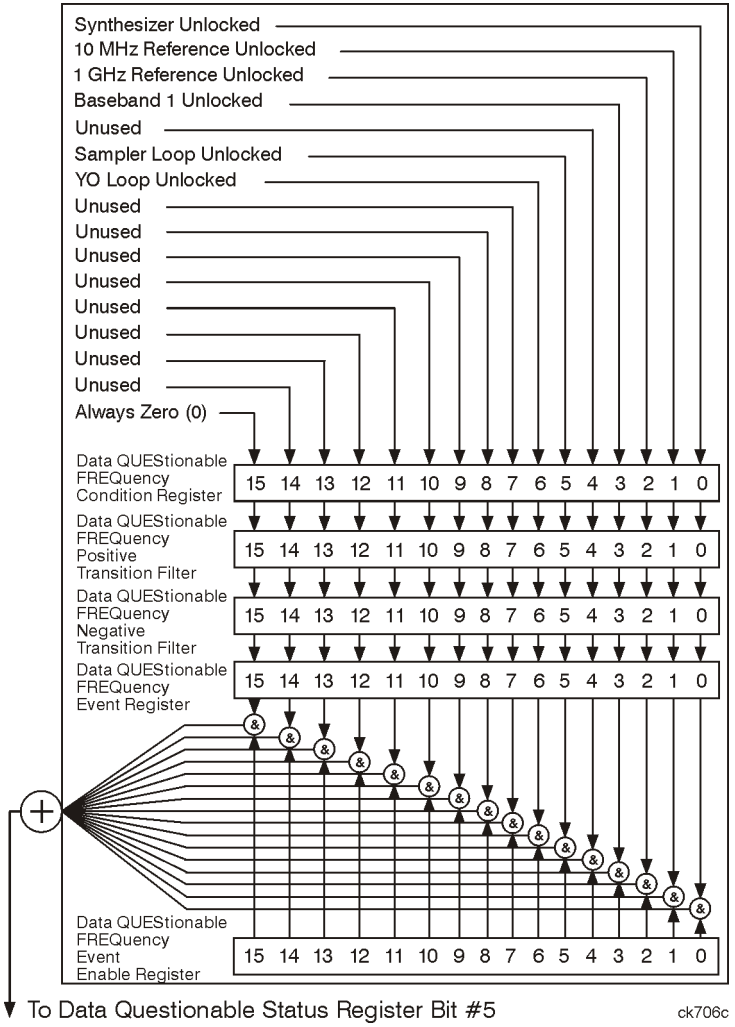
Example: Enable bit 9 and bit 3 so that whenever either of those bits is set to 1, the Data Questionable Power summary bit of the Data Questionable Condition Register is set to 1. Send the command `STAT:QUES:POW:ENAB 520` (512 + 8).

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

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



### Data Questionable Frequency Condition Register

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

**Table 3-9 Data Questionable Frequency Condition Register Bits**

| Bit | Description |
|---|---|
| 0 | **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 | **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**. |

Query: `STATus:QUEStionable:FREQuency:CONDition?`

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

### Data Questionable Frequency Transition Filters (negative and positive)

Specifies which types of bit state changes in the condition register set corresponding bits in the event register. Changes can be positive (0 to 1) or negative (1 to 0).

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

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

### Data Questionable Frequency Event Register

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

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

### Data Questionable Frequency Event Enable Register

Lets you choose which bits in the Data Questionable Frequency Event Register set the summary bit (bit 5 of the Data Questionable Condition Register) to 1.

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

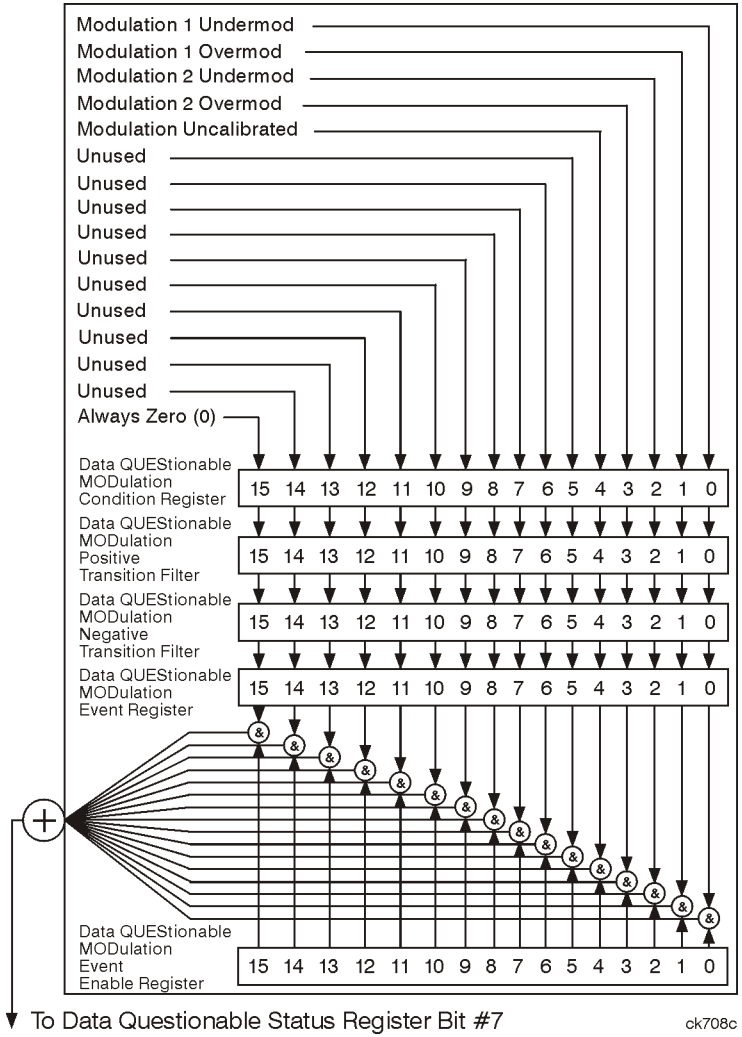
Example: Enable bit 9 and bit 3 so that whenever either of those bits is set to 1, the Data Questionable Frequency summary bit of the Data Questionable Condition Register is set to 1. Send the command `STAT:QUES:FREQ:ENAB 520` (512 + 8).

Query: `STATus:QUEStionable:FREQuency:ENABle?`

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

## Data Questionable Modulation Status Group

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



### Data Questionable Modulation Condition Register

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

**Table 3-10 Data Questionable Modulation Condition Register Bits**

| Bit | Description |
|---|---|
| 0 | **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.

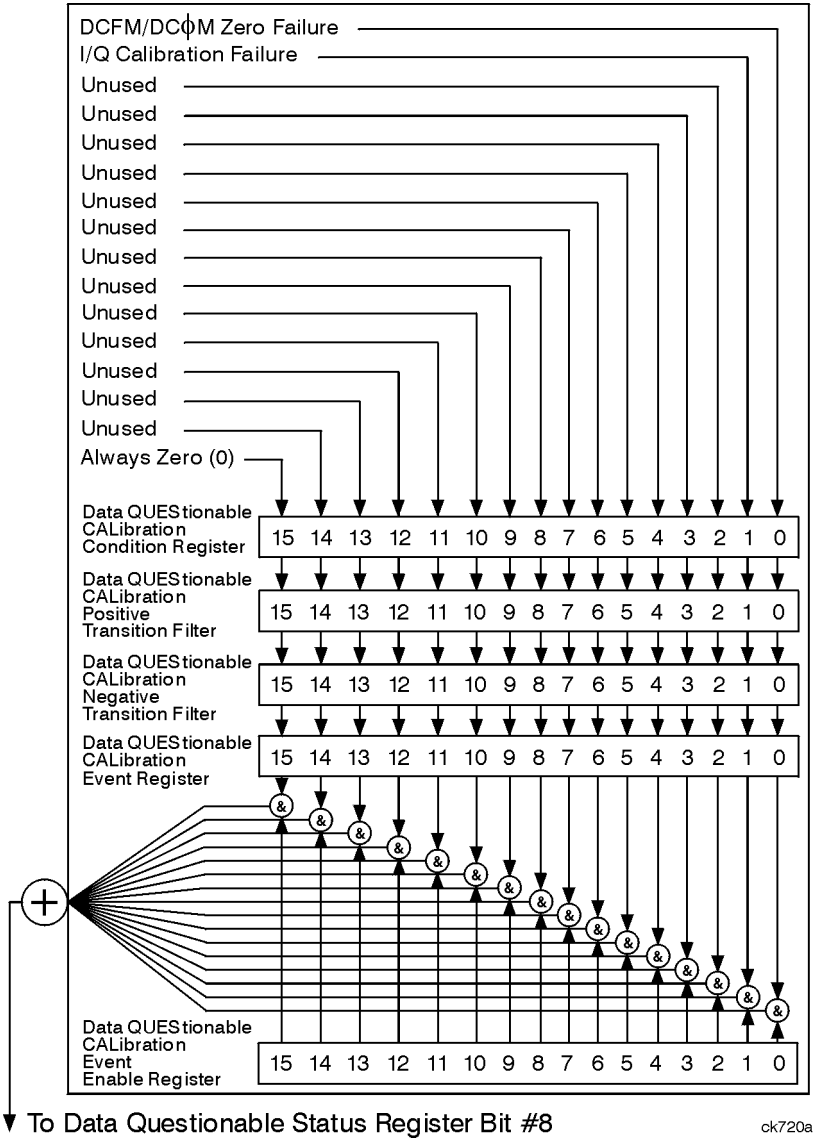
Example: Enable bit 9 and bit 3 so that whenever either of those bits is set to 1, the Data Questionable Modulation summary bit of the Data Questionable Condition Register is set to 1. Send the command `STAT:QUES:MOD:ENAB 520` (512 + 8).

Query: `STATus:QUEStionable:MODulation:ENABle?`

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

## Data Questionable Calibration Status Group

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 | **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**. |

Query: `STATus:QUEStionable:CALibration:CONDition?`

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

### Data Questionable Calibration Transition Filters (negative and positive)

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

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

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

### Data Questionable Calibration Event Register

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

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

### Data Questionable Calibration Event Enable Register

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

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

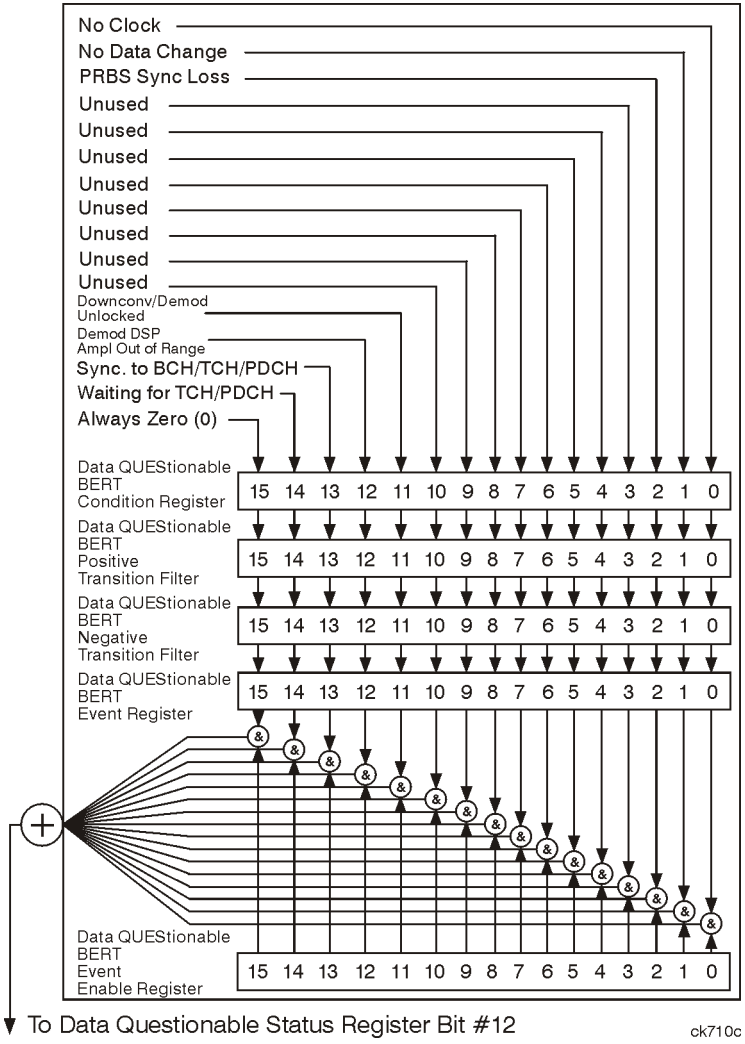
Example: Enable bit 9 and bit 3 so that whenever either of those bits is set to 1, the Data Questionable Calibration summary bit of the Data Questionable Condition Register is set to 1. Send the command `STAT:QUES:CAL:ENAB 520` (512 + 8).

Query: `STATus:QUEStionable:CALibration:ENABle?`

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

## Data Questionable BERT Status Group

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.



### 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 is 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 Downloading and Using Files

Computer generated data can be downloaded into the signal generator. Depending on the options present, the signal generator can accept ARB waveform data, user file data, FIR filter coefficient data, and data downloads directly to waveform memory.

This section explains signal generator memory, and the different waveform download methods:



**NOTE** The procedures in this chapter were written with the assumption that you are familiar with the signal generator's front panel controls and softkey menus. If you are not, please refer to the User's Guide.

# ARB Waveform Data Downloads

The signal generator accepts I/Q waveform data downloads. After downloading the data file into non-volatile memory, the file can be loaded into volatile memory and then played. These user-defined I/Q waveforms can also be sequenced together with other waveforms and played as part of a waveform sequence.

**NOTE** The signal generator can use waveform files developed for the E443xB model signal generators. Refer to "Downloading E443xB Signal Generator Files" on page 160 for information on how to download these file types.

The I/Q waveform data is used to drive the I and Q ports of the I/Q modulator. The waveform data is described using 16-bit I and 16-bit Q integer values in 2's complement format. The I and Q data values are interleaved, creating a single I/Q waveform data file. The 2-byte I integer and 2-byte Q integer values, along with a marker byte make up one sample and one point is one pair of I/Q values. There are five bytes of data for every sample as shown in Table 4-1.

The signal generator uses a marker file that is always associated with an I/Q waveform file. If you do not create a marker file for the I/Q waveform file then the signal generator will automatically create one. This automatically generated default marker file consists of all zeros. The marker data drives the signal generator's EVENT output connectors.

- Marker bit 1 drives EVENT 1 (Rear-panel BNC)
- Marker bit 2 drives EVENT 2 (Rear-panel BNC)
- Marker bit 3 drives EVENT 3 (Rear-panel Auxiliary D-Connector pin 19)
- Marker bit 4 drives EVENT 4 (Rear-panel Auxiliary D-Connector pin 18)

**NOTE** The default marker file is automatically created when no user-defined marker file is provided. The creation is done when the I/Q waveform file is loaded into volatile WFM1 (waveform memory) prior to playing. If the default marker file is used, toggle the **Pulse/RF Blank (None)** softkey to None.

The marker file consists of 8-bit samples with each sample having four marker bits and four unused bits. The result is that the I/Q file will have four times as many bytes as the marker file. See Table 4-1 for more detail on the file structure.

The signal generator uses this two-file format when generating waveform data. More details are given in the following sections of this chapter.

**Table 4-1**

| I/Q Data File Structure | | | |
|---|---|---|---|
| **Sample 1** | | **Sample 2** | |
| I 16 bits | Q 16 bits | I 16 bits | Q 16 bits |
| 2 bytes | 2 bytes | 2 bytes | 2 bytes |
| **Marker File Structure** | | | |
| 4 bits unused MSB | $M_0$ 4 bits LSB | 4 bits unused MSB | $M_1$ 4 bits LSB |
| 1 byte | | 1 byte | |

## Bit-value and Output Power

Bit-value and output-power:

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

## Types of Arbitrary Waveform Generator Memory

Waveform data can be saved to volatile memory (called waveform memory or WFM1) and non-volatile memory (called NVWFM memory). The data in waveform memory is lost whenever the signal generator's line power is cycled. If Option 005 is not installed, then the signal generator provides approximately 3 Msamples of non-volatile memory. With Option 005 there is approximately 1 Gsamples of NVWFM memory available.

Waveforms stored in NVWFM memory must be moved to volatile memory in order to be sequenced and played.

### Waveform Data Storage Path

- Volatile memory - waveform memory

I/Q waveform data is stored in the signal generator's `/user/bbg1/waveform/` directory. The associated marker and header data files, if provided, are stored in the `/user/bbg1/markers/` and `/user/bbg1/header/`directories. This is volatile memory and the contents will be lost when the signal generator is turned off.

- Non-volatile memory - NVWFM memory

If the Option 005 is installed then I/Q data is stored in the signal generator's `/user/waveform/` directory. The associated marker data file, if provided, is stored in the `/user/markers/`directory and the header data file is stored in the `/user/header/`directory. This is non-volatile memory and the contents will not be lost when the signal generator is turned off.

## Data Requirements

I/Q waveform data downloads have the following requirements:

- Data must be in signed, 2's complement format.
- Data must be in 2-byte integers.

  Two bytes are needed to express 16-bit waveforms. The signal generator accepts the most significant byte (MSB) first.
- Input data must be between -32768 and 32767.

  This range is based on the input specifications of the 16-bit DAC used to create the analog voltages for the I/Q modulator.
- Each I/Q waveform must contain at least 60 samples to play in the waveform sequencer (one sample equals one pair of I/Q values and markers). An error message, "`File format invalid`", is displayed if this requirement is not met. The file format is discussed in greater detail in the following sections.
- Each I/Q waveform must contain an even number of samples to play in the waveform sequencer. An error message, "`File format invalid`", is displayed if this requirement is not met. The file format is discussed in greater detail in the following sections.
- A marker file is always associated with an I/Q waveform file. An empty (all zeros) default marker file will be created if a marker file is not provided by the user.
- The user-defined marker file and I/Q waveform data file *must* have the same name in the signal generator.

## File Structure and Memory

For volatile waveform memory (WFM1), there are approximately eight Msamples (32 Msamples with Option 002) of memory allocated in 1024-byte segments. For non-volatile memory (NVWFM), Option 005 provides approximately 1 Gsamples of storage. Signal generators without Option 005 provide 3 Msamples of NVWFM storage.

A waveform file must have a minimum of 60 samples of data. Each sample equals one I/Q pair of values, represented by four bytes of data, along with markers, represented by a single byte of data. A 60 sample waveform file will occupy at 1024 bytes of waveform memory.

If a waveform file is too large to fit into a 1024-byte memory segment, additional memory space is allocated in multiples of 1024 bytes. For example, a waveform represented by 500 samples is allocated to a 4096-byte memory segment (500 samples x 5 bytes).

Total memory usage may be much more than the sum of the samples that make up waveform files. Many small waveform files can use large amounts of memory.

## Downloading Waveforms

Two files, a waveform data file and the associated marker file, are downloaded into waveform memory before being sequenced and played. The waveform data file can be loaded into the signal generator's waveform or NVWFM memory using the following 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). Refer to "Downloads Using FTP" on page 152 for information on FTP.

### Sample Command Line

A sample command line using SCPI:

SCPI command, `<Arbitrary Block Data>`

The `<Arbitrary Block Data>` is defined in the IEEE std. 488.2-1992 section 7.7.6. The following is an example or the format as used to download waveform data to the signal generator:

```
:MMEM:DATA "WFM1:<file_name>",#ABC
```

| | |
|---|---|
| `<file_name>` | the name of the waveform file stored in the signal generator. |
| `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 waveform data. |

**NOTE** If sockets is used to send data to the signal generator, you must provide an end-of- file indicator. Use the following command to download waveform data: MEM:DATA <WFM1:`file_name`>,`#0`<`data`> NL^END

**Example 1**

```
:MMEM:DATA "WFM1 :FILENAME", #3 240 ( 240 bytes of data )
                             A  B          C
```

`FILENAME` the waveform file name as it will appear in the signal generator's waveform memory catalog

`#3` defines the number of decimal digits to follow in B. This variable is represented by A in the sample command line.

`240` denotes how many bytes of data are to follow. This variable is represented by B in the sample command line.

`(240 bytes of data)` the binary waveform data order for each 2-byte sample is defined as MSB (most significant byte) first and LSB (least significant byte) last. The waveform must have at least 60 samples of data. Each sample (I/Q data) is represented by 4 bytes, 2 bytes for the I sample and 2 bytes for the Q sample. In the example above the data, 240 bytes, represents 60 samples of data.

**Example 2**

```
:MMEM:DATA "WFM1:file_name",#1912S407897
```

`file_name` the waveform file name as it will appear in the signal generator's waveform memory catalog.

`#1` defines the number of decimal digits to follow in "B".

`9` denotes how many bytes of data are to follow.

`12S407897` the ASCII representation of the data that is downloaded to the signal generator. This variable is represented by C in the sample command line.

### Downloads to Waveform Memory

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

`MMEM:DATA "WFM1:<file_name>",#ABC` for the waveform data file.
`MMEM:DATA "MKR1:<file_name>",#ABC` for the markers file.
`MMEM:DATA "HDR1:<file_name>",#ABC` for the header file.

The full directory path name can be specified in the command line. The following SCPI commands are equivalent to the previous commands:

`MMEM:DATA "/USER/BBG1/WAVEFORM/<file_name>",#ABC` for the waveform data file.
`MMEM:DATA "/USER/BBG1/MARKERS/<file_name>",#ABC` for the markers file.
`MMEM:DATA "/USER/BBG1/HEADER/<file_name>",#ABC` for the header file.

### Downloads to NVWFM Memory:

To download files to NVWFM (non-volatile memory), using the GPIB or the LAN interface, use the following SCPI commands:

`MMEM:DATA "NVWFM:<file_name>",#ABC` for the waveform file.
`MMEM:DATA "NVMKR:<file_name>",#ABC` for the markers file.
`MMEM:DATA "NVHDR:<file_name>",#ABC` for the header file.

The full directory path name can be specified in the command line. The following SCPI commands are equivalent to the previous commands:

`MMEM:DATA "/USER/WAVEFORM/<file_name>",#ABC` for the waveform file.
`MMEM:DATA "/USER/MARKERS/<file_name>",#ABC` for the markers file.
`MMEM:DATA "/USER/HEADER/<file_name>",#ABC` for the header file.

**Downloads Using FTP** To download files to NVWFM memory (non-volatile memory), using the file transfer protocol (FTP) over the LAN interface, perform the following steps.

1. From the PC Command Prompt, or Unix command line, change the directory to the directory where the file to be downloaded is located.
2. From the PC Command Prompt or Unix command line type `ftp` *instrument name*. Where *instrument name* is the hostname for the signal generator or the signal generator's IP address.
3. At the `User:` prompt, in the ftp window, press the Enter key (no entry is required).
4. At the `Password:` prompt, in the ftp window, press the Enter key (no entry is required).

5. At the `ftp` prompt type:
   `put <file_name> /USER/WAVEFORM/<file_name_1>`

   where `<file_name>` is the name of the file to download and `<file_name_1>` the name designator for the signal generator's `/USER/WAVEFORM/` directory.

   If you have a marker file associated with the data file, use the following command to download it to the signal generator:
   put `<marker file_name> /USER/MARKERS/<file_name_1>`

   where `<marker file_name>` is the name of the file to download and `<file_name_1>` the name designator for the file in the signal generator's `/USER/MARKERS/` directory. Marker files and the associated I/Q waveform file 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

## Example Programs

**Waveform Generation Using C++** The following program (`Metrowerks CodeWarrior 3.0`) creates an I/Q waveform and writes the data to a file on your PC. Once the file is created, you can use the file transfer protocol (FTP) to download the waveform data to the signal generator. Refer to "Downloads Using FTP" on page 152 for more information on FTP.

```
#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;
}
```

**Waveform Downloading Using HP BASIC for Windows™** 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 80 with:

210 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
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. |
| 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. |

**Waveform Downloading Using HP BASIC for UNIX** 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 45 and line 50.

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. |
| 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. |

## Playing a Downloaded Waveform

The following procedure shows you how to load and play a downloaded waveform file, using front panel key presses and remote commands.

1. Select the downloaded waveform file in non-volatile waveform memory (NVWFM) and load it into volatile waveform memory (WFM1). The file consists of both I/Q and marker file data, and requires a SCPI command to load remotely.

   Via the front panel:

   a. Press **Mode** > **Dual ARB** > **Select Waveform** > **Waveform Segments** > **Load Store** until Load is highlighted.

   b. Highlight the waveform file in the NVWFM catalog using the arrow keys or front panel knob.

   c. Press **Load Segment From NVWFM Memory**. If the waveform is not highlighted, use the cursor to highlight the waveform.

   Via the remote interface, send any one of the following SCPI commands:

   ```
   :MEMory:COPY[NAME]"<NVWFM:file_name>","<WFM1:file_name>"
   :MEMory:COPY[NAME]"<NVMKR:file_name>","<MKR1:file_name>"
   :MEMory:COPY[NAME]"<NVHDR:file_name>","<HDR1:file_name>"
   ```

**NOTE** When copying a waveform, marker or header file from volatile and non-volatile memory, the waveform and the associated marker and header files are copied. Conversely, when a waveform file is deleted, the associated marker and header files are deleted. It is not necessary to send a separate command to copy or delete the marker and header files.

2. Select the downloaded waveform file in volatile waveform memory for playback.

   Via the front panel:

   a. Press **Return** > **Select Waveform.**

   b. Highlight selection.

   c. Press **Select Waveform.**

   To select a segment via the remote interface, send the following SCPI command:

   ```
   [:SOURce}:RADio:ARB:WAVeform "WFM1:<file_name>"
   ```

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

Via 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.

Via the remote interface, send the following SCPI commands:

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

## Downloading E443xB Signal Generator Files

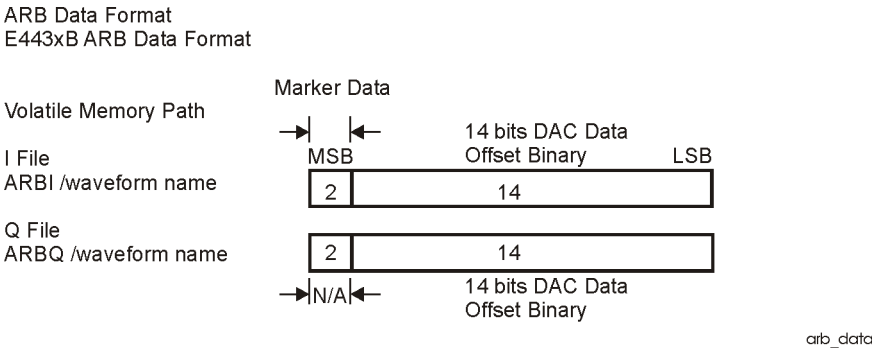
Download the E443xB type files to the signal generator exactly as if downloading files to a E443xB signal generator.

Downloaded E443xB waveform type files will automatically be converted to the new file format, as described on page 146, and stored into the signal generator's memory.

The file conversion process takes more time than downloading files that are already in the new file format. Store E443xB file downloads to waveform memory and then transfer them over to NVWFM memory to avoid the time required to convert these file types.

### 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 Table 4-1 on page 147. If new waveform files are created for the signal generator, use the new style format.



### Storage Locations for ARB files

Waveforms can be stored to either volatile waveform memory or the non-volatile NVARB memory. The signal generator supports the E443xB directory structure for waveform file storage.

- For the E443xB style 14-bit waveforms the following storage locations are used:

1. For non-volatile, NVARB memory, the directory locations are `/user/nvarbi/` and `/user/nvarbq`.
2. For volatile waveform memory the directory locations are `/user/arbi/` and `/user/arbq/`.

Loading files into the above directories does not actually store them in those directories. Instead, these directories function as “pipes” to the format translator. The E443xB files are translated into 16-bit versions (by appending 0’s to the least significant bit (LSB) and interleaving the data) and stored in the regular waveform directories along with other new style waveform files.

Although the you can see the E443xB file names in the `/arbi`, `/arbq` and `nvarbi/nvarbq` directories, these names are really pointers. Refer to “Types of Arbitrary Waveform Generator Memory” on page 147 for more information on the new style directory structure.

### SCPI Commands

The signal generator automatically generates a marker file for downloaded E443xB waveform files. The following commands will download E443xB waveform files into the signal generator.

### Downloads to Waveform Memory:

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

```
:MMEM:DATA "ARBI:<file_name>", <I waveform data>
:MMEM:DATA "ARBQ:<file_name>", <Q waveform data>
```

These commands download E443xB I/Q waveform data into the signal generator's waveform memory. The <`I waveform data`> and <`Q waveform data`> is the file format as described above. The string variable `<file_name>` is the name of the waveform data file. The signal generator will associate a marker file with the data file.

### Downloads to Non-Volatile Memory:

```
:MMEM:DATA "NVARBI:<file_name>", <I waveform data>
:MMEM:DATA "NVARBQ:<file_name>", <Q waveform data>
```

These commands download E443xB I/Q waveform data into the signal generator's non-volatile memory. The string variable `<file_name>` is the name of the data file. The signal generator will associate a marker file with the data file when the file is moved to waveform memory prior to playing.

### Example Programs

**Waveform Downloading Using HP BASIC for Windows™** 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:

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

and replace line 130 with:

130 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 156 for program comments. |

**Waveform Downloading Using HP BASIC for UNIX** 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 158 for program comments. |

# User Bit/Binary File Data Downloads

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.

- 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 170.

**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 the data into waveform memory, and sets the other seven 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 symbol-by-symbol; one symbol equals one byte and occupies one address in memory.

### Bit Memory and Binary Memory

User files can be downloaded to the 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 seven, divided by eight, then rounded down to the nearest integer. 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.

---

**NOTE** Not all binary values are ASCII characters that can be printed. In fact, only ASCII characters corresponding to decimal values 32 through 126 are printable keyboard characters. Typically, the ASCII character corresponding to an 8-bit pattern is not printable.

Because of this, the program written to download and upload user files *must correctly convert* the binary data into 8-bit ASCII characters.

---

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

Download size limitations are directly proportional to the available memory space and the signal generator's pattern RAM size (Option 001= 1 Mbyte, Option 002 = 4 Mbyte). To determine the maximum user file size, you must consider the following:

- framing overhead
- pattern RAM size (1 Mbyte or 8 Mbyte)
- available memory

You may have to delete files from memory before downloading larger files.

## Data Volatility

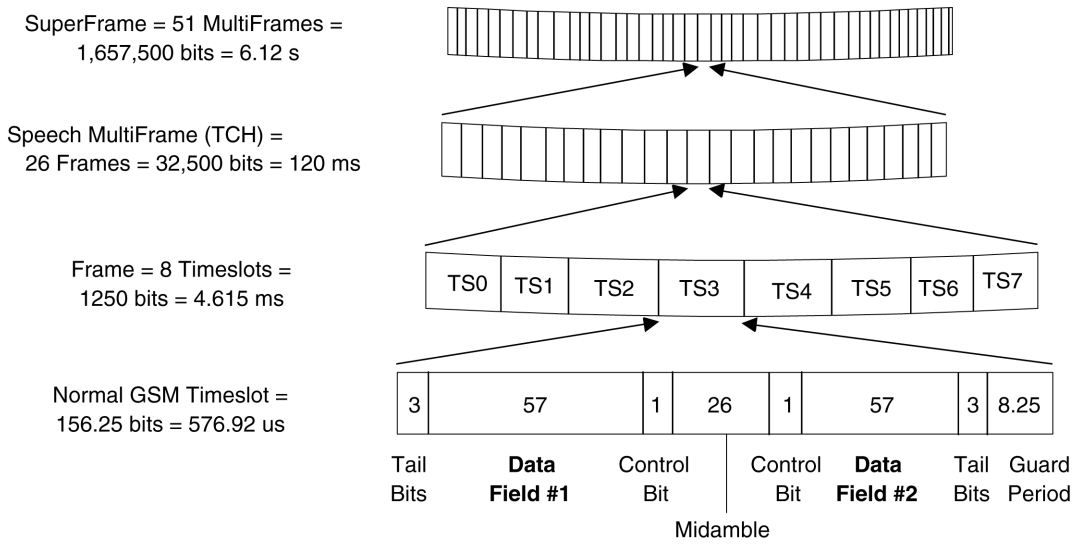
The signal generator provides two data storage areas: volatile waveform memory (WFM1) and non-volatile memory (NVWFM). Data stored in volatile waveform memory cannot be recovered if it is overwritten or if the power is cycled. Data stored in non-volatile memory, however, remains until you delete the file. The Option 005 signal generator's hard disk provides 1 Gsamples of non-volatile storage. Signal generators without Option 005 provide 3 Msamples of non-volatile storage.

**NOTE** References to pattern RAM (PRAM) are for descriptive purposes only. PRAM equates to volatile waveform memory (WFM1).

## 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 4-1 GSM Multiframe Transmission**




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.

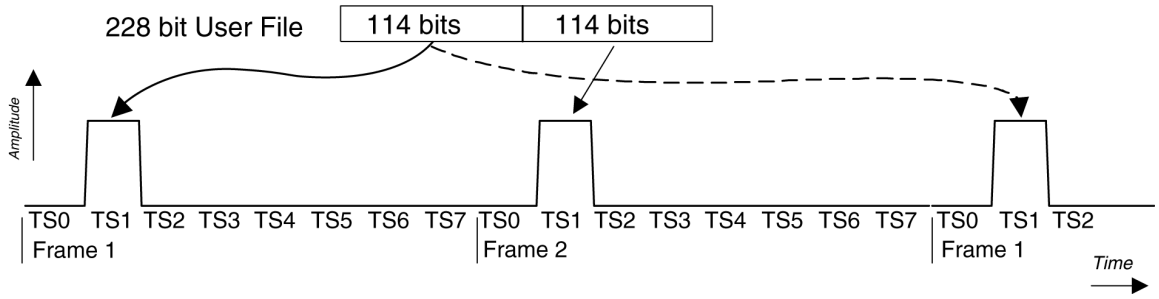
**NOTE** The data in PRAM is static. Firmware writes to PRAM once for the configuration selected and the hardware reads this data repeatedly. Firmware overwrites the volatile PRAM memory to reflect the desired configuration only when the data source or mode (digital communications format) is changed.

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 seven 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 4-2.

**Figure 4-2 Mapping User File Data to a Single Timeslot**



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

| **Frame** | **Timeslot** | PRAM Address | **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 address 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 required =
size of normal GSM timeslot × timeslots per frame × speech multiframe(TCH) × superframe

size of normal GSM timeslot = 156.25 bits

timeslots per frame = eight 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 bytes.

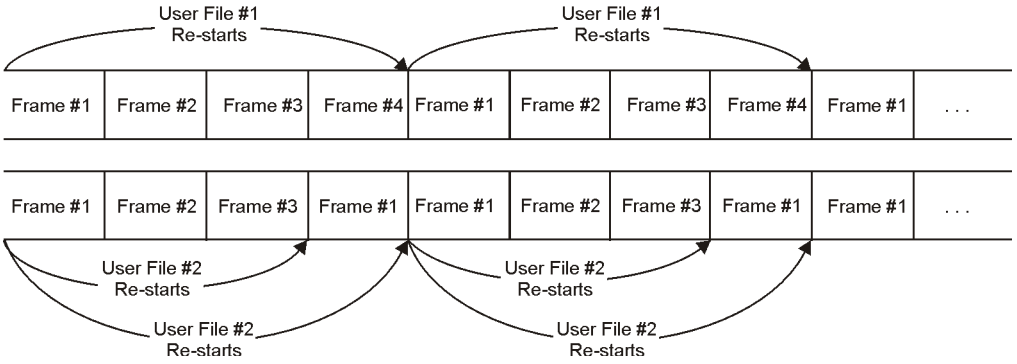
## Multiple User Files Selected as Data Sources for Different Timeslots

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 4-3. If these were integer multiples of each other, both user files would be continuous, and user file #2 would repeat after two frames.

**Figure 4-3 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 size (8 Msample or 32 Msample), framing overhead, and available memory.
5. For downloads to binary memory, bit length must be a multiple of eight; SCPI specifies the data in 8-bit bytes.

---

**NOTE** Not all binary values are ASCII characters that can be printed. Only ASCII characters corresponding to decimal values 32 through 126 are printable keyboard characters. Typically, the ASCII character corresponding to an 8-bit pattern is not printable.

Because of this, the program written to download and upload user files *must correctly convert* the binary data into 8-bit characters.

---

### Preliminary Setup

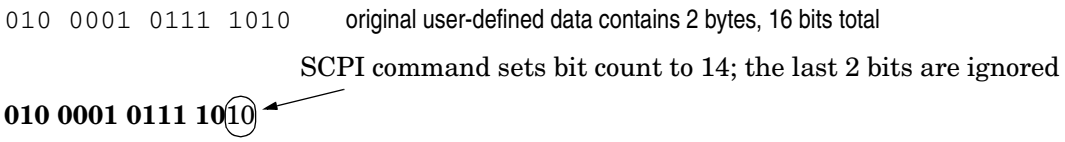
No preliminary setup is required for user file downloads.

### 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 seven, divided by eight, then rounded down to the nearest integer. 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 a16-bit string using the command `:MEMory:DATA:BIT "file_name", 14, #12Qz` results in the last 2 bits being ignored. See the following figure.




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.

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

**Example** `:MEMory:DATA:BIT "file_name", 16, #12Qz`

| | |
|---|---|
| `file_name` | provides the user file name as it will appear in the signal generator's binary memory catalog |
| `#1` | defines the number of decimal digits to follow in "B" |
| `2` | denotes how many bytes of data are to follow |
| `Qz` | the ASCII representation of the 16 bits of data that are downloaded to the signal generator. This variable is represented by "C" in the sample command line |

**Querying the Waveform Data** Use the following SCPI command to query user file data from binary memory:

```
: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>", <datablock>`

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 |
| `#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", #1912S407897`

| | |
|---|---|
| `file_name` | provides the user file name as it will appear in the signal generator's binary memory catalog |
| `#1` | defines the number of decimal digits to follow in "B" |
| `9` | denotes how many bytes of data are to follow |
| `12S407897` | the ASCII representation of the data that is downloaded to the signal generator. This variable is represented by C in the sample command line |

**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:SLOT1:UTCHannel:DATA "BIT:<file_name>"
```

`[:SOURce]:RADio:NADC[:STATe] On` activates the NADC format.

## Modulating and Activating the Carrier

The following front panel key presses or remote commands will 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

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 185.

## 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 ($\Delta t$) is equal to the inverse of the sampling rate (FS). The sampling rate is equal to the symbol rate multiplied by the oversample ratio. For example, the GSM symbol rate is 270.83 ksps. With an oversample ratio of 4, the sampling rate is 1083.32 kHz and $\Delta t$ (inverse of FS) is 923.088 nsec.

## Data Volatility

The signal generator provides two data storage areas: volatile waveform memory (WFM1) and non-volatile memory (NVWFM). FIR filter coefficients stored in volatile waveform memory cannot be recovered if overwritten or if the power is cycled. Coefficients stored in non-volatile memory, however, remain until you delete the file. The Option 005 signal generator's hard disk provides 1 Gsamples of non-volatile storage. Signal generators without Option 005 provide 3 Msamples of non-volatile storage.

## Downloading FIR Filter Coefficient Data

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,`<br>`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:

`[:SOURce]:FREQuency:FIXed 2.5GHZ` sets the carrier frequency to 2.15 GHz.

`[:SOURce]:POWer[:LEVel][:IMMediate][:AMPLitude] -10.0DBM` sets the carrier amplitude to -10.0 dBm.

`:OUTPut:MODulation[:STATe] ON` modulates the carrier.

`:OUTPut[:STATe] ON` activates the RF output.

# Downloads Directly into Pattern RAM (PRAM)

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 Metlab™ 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 allow you 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.

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

## Data Limitations

Total (data bits plus control bits) download size limitations are 8 Mbytes or 32 Mbytes with Option 002. Each sample for PRAM uses 4 bytes of data.

A data pattern file containing 8 Mbits of modulation data must contain another 56 Mbits of control information. A file of this size requires 8 Mbytes of memory; the largest amount of modulation data for a waveform in the signal generator without Option 002.

## 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 1 Gsamples of non-volatile storage. Signal generators without Option 005 provide 3 Msamples of non-volatile storage.

**NOTE** References to pattern RAM (PRAM) are for descriptive purposes only. PRAM equates to volatile waveform memory (WFM1).

## Downloading in List Format

**NOTE** 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 seven bits of control information associated with the data field bit. See Table 4-1 for the required data and control bits.

Total (data bits plus control bits) download size limitations are 8 Mbytes or 32 Mbytes for Option 002.

### 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. 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 the 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 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:LIST <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 4-1. 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:LIST 85,21,20,20,21,21,20,20,21,21,20,20,21,21,20,20,21,
21,20,20,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,
16,16,16,16,16,16,16,16,16,144
```

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

### Querying the Waveform Data

Use the following SCPI command line to determine whether there is a user-defined pattern in the PRAM:

```
:MEMory:DATA:PRAM?
```

## Downloading in Block Format

**NOTE** Because there is no parsing, block data format downloads are 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 seven bits of control information associated with the data field bit. See Table 4-1 for the required data and control bits.

Total (data bits plus control bits) download size limitations are 8 Mbytes or 32 Mbytes for Option 002.

Because a waveform containing 16 Mbit of data for subsequent modulation must also contain another 112 Mbits of control information, a file this size requires a signal generator with Option 002, which provides 32 Mbytes of pattern RAM. The largest amount of modulation data for a waveform in an Option 001 signal generator is approximately 8 Mbits, which leaves enough room for the required 56 Mbits of control bits.

### Preliminary Setup

Before downloading data, set up the digital communications format to enable the signal generator to define the modulation format, filter, and data clock. Activating the digital communications format after data downloads to PRAM can corrupt the 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:BLOCk <datablock>`

This command downloads the block-formatted data directly into pattern RAM.

### Sample Command Line

A sample command line:

`:MEMory:DATA:PRAM:BLOCk #ABC`

| | |
|---|---|
| `#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:BLOCk #1912S407897`

| | |
|---|---|
| `#1` | defines the number of decimal digits to follow in “B”. |
| `9` | denotes how many bytes of data are to follow. |
| `12S407897` | is the ASCII representation of the data downloaded to the signal generator. This variable is represented by C in the sample command line. |

**NOTE** Not all binary values can be printed as ASCII characters. In fact, only ASCII characters corresponding to decimal values 32 to 126 are printable keyboard characters. The above example was chosen for simplicity. Typically, the binary value corresponding to your 8-bit pattern is not printable.

Therefore, the program written to download and upload user files *must correctly convert* between binary and the visible representation of the data sequence.

## 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.

# Data Transfer Troubleshooting

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

## Direct PRAM Download Problems

**Table 4-2 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.<br><br>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.<br><br>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 seven bits of control information (bits 1-7).

| 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.<br>Set to 0 = RF off.<br>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.<br>Set to 1 = end of memory and restart memory playback.<br>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. |

## User File Download Problems

**Table 4-3 User FIR File Download Trouble - Symptoms and Causes**

| Symptom | Possible Cause |
|---|---|
| No data modulated | Not enough data to fill a single timeslot.<br><br>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,<br>some data modulated,<br>some data missing | Data does not completely fill an integer number of timeslots.<br><br>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.

---

The following method can be used to compute the number of data pattern repetitions required in order to form a continuous data stream.

In this example, a modified PN9, 511-bit data pattern is to be applied as the data source for a 114-bit data field in a GSM Normal timeslot.

Set up a spreadsheet containing:

A = number of repetitions of the original data pattern
B = user file length = number of repetitions × original data pattern length
C = Number of characters = user file length ÷ 8 (8 bits-per-character)
D = number of frames = user file length ÷ timeslot data field size (114)
E = total required PRAM = number of frames × number of bits-per-frame (1250 for GSM)

| **A**<br><br>**Number of reps** | **B**<br><br>**Data Pattern Length × Repetitions** | **C**<br><br>**Number of Characters (B ÷ 8)** | **D**<br><br>**Number of frames needed to end on a timeslot boundary (B ÷ timeslot data field size)** | **E**<br><br>**Total PRAM required (D × number of bits-per-frame)** |
|---|---|---|---|---|
| 1 | 511 | 63.88 | 4.48 | 5,603.07 |
| 2 | 1,022 | 127.75 | 8.96 | 11,206.14 |
| 3 | 1,533 | 191.63 | 13.45 | 16,809.21 |
| 4 | 2,044 | 255.50 | 17.93 | 22,412.28 |
| 5 | 2,555 | 319.38 | 22.41 | 28,015.35 |
| 6 | 3,066 | 383.25 | 26.89 | 33,618.42 |
| 7 | 3,577 | 447.13 | 31.38 | 39,221.49 |
| 8 | 4,088 | 511 | 35.86 | 44,824.56 |
| 9 | 4,599 | 574.88 | 40.34 | 50,427.63 |
| ... | ... | ... | ... | ... |
| 455 | 232,505 | 29,063.13 | 2,039.52 | 2,549,396.92 |
| 456 | 233.016 | 29,127 | 2,044 | 2,555,000 |

The first row where both columns C and D are integers (the shaded row at the bottom of the table) is the minimum number of repetitions required to transmit the user file without discontinuity. In this example, in order to correctly generate the modified PN9 and download it to a user file, the user file must contain 456 repetitions of the 511-bit pattern. 233,016 total bits will be downloaded to the signal generator, for a total of 29,127 characters.

**"Pattern RAM Memory Depth" Requirement** It is possible to exhaust the available PRAM with a large, continuous user file transmitted across a large number of frames.

In the previous example, selecting the 233,016-bit user file as the data source for the GSM Normal timeslot will cause the firmware to compute 2,044 frames of data, filling 2,555,000 bytes of PRAM depth. Option 002 (4 Mbyte PRAM) is required for this configuration. Trying to load this data on an Option 001 (1 Mbyte PRAM) signal generator will cause an error, because there is not enough PRAM to hold the required data.

If PN11 was used instead of PN9, 456 repetitions of the data pattern would require a 933,432-bit user file, requiring 8,188 frames and 10,235,000 bytes of PRAM. Because the size of this data exceeds the limits of Option 002, you would need to supply a file this size via the external DATA connector.

**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

The 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 the internally generated framing

## User FIR Filter Coefficient File Download Problems

**Table 4-4 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.<br><br>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.<br><br>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.

## ARB Waveform Data Download Problems

**Table 4-5 I/Q Waveform Data Download Trouble - Symptoms and Causes**

| 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.<br><br>To solve the problem, either change the name of the waveform being downloaded or turn off the ARB. |
| ERROR -321, Out of memory. | There is not enough space in the ARB memory for the waveform file being downloaded.<br><br>To solve the problem, either reduce the file size of the waveform file or delete unnecessary files from ARB memory. |
| No RF Output | If no user marker file is provided then a default marker file containing all zeros is created. If the signal generator's Mrk 2 to RF Blank softkey is set to on, the RF will be blanked. Go to MODE > Dual ARB > ARB Setup and toggle Mrk 2 to RF to off. |

### Data Requirement Reminders

To avoid I/Q waveform data download problems, the following six conditions *must* be met:

1. Data must be in signed, 2's complement (binary) format.
2. Data must ordered MSB first to LSB last. Each I/Q sample has 4 bytes of data.
3. Input integers must be between –32768 and 32767.
4. Each I and Q waveform file must have at least 60 samples.

# Index

# Index