Appendix G

Application Interface Library

G.1  VISA Library
4. Programming with VISA ..........................................................................................37
VISA Resources and Attributes ........................................................................39
VISA Resources .................................................................................................39
VISA Attributes ..................................................................................................40
Using Sessions ....................................................................................................41
 Including the VISA Declarations File (C/C++) ..................................................41
 Adding the visa32.bas File (Visual Basic) ..........................................................41
 Opening a Session ...............................................................................................41
 Addressing a Session .........................................................................................44
 Closing a Session ...............................................................................................46
 Searching for Resources ......................................................................................47
 Sending I/O Commands ..........................................................................................49
 Types of I/O ...........................................................................................................49
 Using Formatted I/O ...........................................................................................49
 Using Non-Formatted I/O ...................................................................................59
 Using Events and Handlers ....................................................................................62
 Events and Attributes ..........................................................................................62
 Using the Callback Method ..................................................................................69
 Using the Queuing Method ....................................................................................77
 Trapping Errors ....................................................................................................82
 Trapping Errors ....................................................................................................82
 Exception Events ..................................................................................................83
 Using Locks ...........................................................................................................87

5. Programming via GPIB and VXI .........................................................................93
GPIB and VXI Interfaces Overview ......................................................................95
 General Interface Information ..............................................................................95
 GPIB Interfaces Overview ...................................................................................96
 VXI Interfaces Overview .....................................................................................98
 GPIB-VXI Interfaces Overview ...........................................................................100
 Using High-Level Memory Functions ..................................................................102
 Programming the Registers ..................................................................................102
 High-Level Memory Functions Examples .............................................................104
 Using Low-Level Memory Functions ...................................................................107
 Programming the Registers ..................................................................................107
 Low-Level Memory Functions Examples ..............................................................109
 Using Low/High-Level Memory I/O Methods .......................................................112
 Using Low-Level viPeek/viPoke ..........................................................................112
 Using High-level viIn/viOut .................................................................................113
 Using High-level viMoveIn/viMoveOut ...............................................................113
Using the Memory Access Resource......................................... 117
Memory I/O Services .......................................................... 117
MEMACC Attribute Descriptions ........................................ 120
Using VXI-Specific Attributes .................................................... 123
Using the Map Address as a Pointer ........................................ 123
Setting the VXI Trigger Line ............................................... 125

6. Programming via LAN ............................................................ 127
LAN Interfaces Overview ........................................................... 129
LAN Hardware Architecture ................................................ 129
LAN Software Architecture ................................................. 131
LAN Client Interface Overview ........................................... 133
VISA LAN Client Interface Overview .................................. 136
LAN Server Interface Overview ........................................ 140
Communicating with GPIB Devices via LAN ....................... 141
Addressing a Session ......................................................... 141
Using Timeouts over LAN .................................................. 143
LAN Signal Handling on HP-UX ......................................... 145

7. VISA Language Reference .............................................................. 147
VISA Functions Overview .......................................................... 149
VISA Functions by Interface/Resource ................................ 149
VISA Functions by Type ..................................................... 153
viAssertIntrSignal ....................................................................... 158
viAssertTrigger ........................................................................... 160
viAssertUtilSignal ...................................................................... 163
viBufRead .................................................................................. 165
viBufWrite .................................................................................. 167
viClear ........................................................................................ 169
viClose ...................................................................................... 171
viDisableEvent ........................................................................... 173
viDiscardEvents ........................................................................ 176
viEnableEvent .......................................................................... 179
viEventHandler .......................................................................... 184
viFindNext ................................................................................. 189
viFindRsrc ................................................................................ 190
viFlush ....................................................................................... 195
viGetAttribute .......................................................................... 197
viGpibCommand ........................................................................ 199
viGpibControlATN ..................................................................... 201
viGpibControlREN ..................................................................... 203
viGpibPassControl ..................................................................... 205
viGpibSendIFC .......................................................................... 207
viIn8, viIn16, and viIn32 ............................................................ 208
viInstallHandler .......................................................................... 211
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>viLock</td>
<td>213</td>
</tr>
<tr>
<td>viMapAddress</td>
<td>217</td>
</tr>
<tr>
<td>viMapTrigger</td>
<td>220</td>
</tr>
<tr>
<td>viMemAlloc</td>
<td>223</td>
</tr>
<tr>
<td>viMemFree</td>
<td>225</td>
</tr>
<tr>
<td>viMove</td>
<td>226</td>
</tr>
<tr>
<td>viMoveAsync</td>
<td>229</td>
</tr>
<tr>
<td>viMoveIn8, viMoveIn16, and viMoveIn32</td>
<td>233</td>
</tr>
<tr>
<td>viMoveOut8, viMoveOut16, and viMoveOut32</td>
<td>236</td>
</tr>
<tr>
<td>viOpen</td>
<td>239</td>
</tr>
<tr>
<td>viOpenDefaultRM</td>
<td>243</td>
</tr>
<tr>
<td>viOut8, viOut16, and viOut32</td>
<td>245</td>
</tr>
<tr>
<td>viParseRsrc</td>
<td>248</td>
</tr>
<tr>
<td>viPeek8, viPeek16, and viPeek32</td>
<td>251</td>
</tr>
<tr>
<td>viPoke8, viPoke16, and viPoke32</td>
<td>252</td>
</tr>
<tr>
<td>viPrintf</td>
<td>253</td>
</tr>
<tr>
<td>viQueryf</td>
<td>262</td>
</tr>
<tr>
<td>viRead</td>
<td>264</td>
</tr>
<tr>
<td>viReadAsync</td>
<td>267</td>
</tr>
<tr>
<td>viReadSTB</td>
<td>269</td>
</tr>
<tr>
<td>viReadToFile</td>
<td>271</td>
</tr>
<tr>
<td>viScanf</td>
<td>274</td>
</tr>
<tr>
<td>viSetAttribute</td>
<td>284</td>
</tr>
<tr>
<td>viSetBuf</td>
<td>286</td>
</tr>
<tr>
<td>viSPrintf</td>
<td>288</td>
</tr>
<tr>
<td>viSScanf</td>
<td>290</td>
</tr>
<tr>
<td>viStatusDesc</td>
<td>292</td>
</tr>
<tr>
<td>viTerminate</td>
<td>293</td>
</tr>
<tr>
<td>viUninstallHandler</td>
<td>295</td>
</tr>
<tr>
<td>viUnlock</td>
<td>297</td>
</tr>
<tr>
<td>viUnmapAddress</td>
<td>298</td>
</tr>
<tr>
<td>viUnmapTrigger</td>
<td>299</td>
</tr>
<tr>
<td>viVPrintf</td>
<td>301</td>
</tr>
<tr>
<td>viVQueryf</td>
<td>303</td>
</tr>
<tr>
<td>viVScanf</td>
<td>305</td>
</tr>
<tr>
<td>viVSPrintf</td>
<td>307</td>
</tr>
<tr>
<td>viVSScanf</td>
<td>309</td>
</tr>
<tr>
<td>viVxiCommandQuery</td>
<td>311</td>
</tr>
<tr>
<td>viWaitOnEvent</td>
<td>314</td>
</tr>
<tr>
<td>viWrite</td>
<td>314</td>
</tr>
<tr>
<td>viWriteAsync</td>
<td>320</td>
</tr>
<tr>
<td>viWriteFromFile</td>
<td>322</td>
</tr>
<tr>
<td>viWriteFromFile</td>
<td>324</td>
</tr>
</tbody>
</table>
A. VISA Library Information .............................................................. 327
   VISA Type Definitions ................................................................ 329
   VISA Error Codes (Alphabetical) .............................................. 332
   VISA Error Codes (by Function) ............................................... 336
   VISA Directories Information .................................................. 368
       Windows Directory Structure .............................................. 368
       Editing the VISA Configuration ........................................... 370

B. VISA Resource Classes ................................................................. 373
   Resource Classes Overview...................................................... 375
       Resource Classes vs. Interface Types ..................................... 375
       Interface Types vs. Resource Classes .................................... 376
       Resource Class Descriptions ................................................ 376
   Instrument Control (INSTR) Resource....................................... 377
       INSTR Resource Overview ................................................ 377
       INSTR Resource Attributes ............................................. 378
       INSTR Resource Attribute Descriptions .............................. 384
       INSTR Resource Events .................................................. 391
       INSTR Resource Operations ............................................ 393
   Memory Access (MEMACC) Resource...................................... 395
       MEMACC Resource Overview ........................................... 395
       MEMACC Resource Attributes ........................................ 396
       MEMACC Resource Attribute Descriptions ......................... 398
       MEMACC Resource Events ............................................. 400
       MEMACC Resource Operations ....................................... 401
   GPIB Bus Interface (INTFC) Resource.................................... 402
       INTFC Resource Overview .............................................. 402
       INTFC Resource Attributes ............................................ 402
       INTFC Resource Attribute Descriptions ............................... 404
       INTFC Resource Events .................................................. 406
       INTFC Resource Operations ............................................ 408
   VXI Mainframe Backplane (BACKPLANE) Resource.................. 409
       BACKPLANE Resource Overview ....................................... 409
       BACKPLANE Resource Attributes ..................................... 410
       BACKPLANE Resource Attribute Descriptions ..................... 411
       BACKPLANE Resource Events .......................................... 412
       BACKPLANE Resource Operations .................................... 412
   Servant Device-Side (SERVANT) Resource .............................. 413
       SERVANT Resource Overview ......................................... 413
       SERVANT Resource Attributes ....................................... 414
       SERVANT Resource Attribute Descriptions ......................... 415
       SERVANT Resource Events ............................................. 417
       SERVANT Resource Operations ....................................... 419
Contents 8

TCPIP Socket (SOCKET) Resource ..........................................420
  SOCKET Resource Overview .............................................420
  SOCKET Resource Attributes ..........................................420
  SOCKET Resource Attribute Descriptions ..........................422
  SOCKET Resource Event ...................................................423
  SOCKET Resource Operations ..........................................424

Glossary ..............................................................................................425

Index ......................................................................................................431
Notice

The information contained in this document is subject to change without notice.

Agilent Technologies shall not be liable for any errors contained in this document. *Agilent Technologies makes no warranties of any kind with regard to this document, whether express or implied. Agilent Technologies specifically disclaims the implied warranties of merchantability and fitness for a particular purpose.* Agilent Technologies shall not be liable for any direct, indirect, special, incidental, or consequential damages, whether based on contract, tort, or any other legal theory, in connection with the furnishing of this document or the use of the information in this document.

Warranty Information

A copy of the specific warranty terms applicable to your Agilent Technologies product and replacement parts can be obtained from Agilent Technologies, Inc.

U.S. Government Restricted Rights

The Software and Documentation have been developed entirely at private expense. They are delivered and licensed as "commercial computer software" as defined in DFARS 252.227-7013 (Oct 1988), DFARS 252.211-7015 (May 1991) or DFARS 252.227-7014 (Jun 1995), as a "commercial item" as defined in FAR 2.101(a), or as "Restricted computer software" as defined in FAR 52.227-19 (Jun 1987) (or any equivalent agency regulation or contract clause), whichever is applicable. You have only those rights provided for such Software and Documentation by the applicable FAR or DFARS clause or the Agilent standard software agreement for the product involved.
Trademark Information

Microsoft®, Windows ® 95, Windows ® 98, Windows ® Me, Windows ® 2000, and Windows NT® are U.S. registered trademarks of Microsoft Corporation. All other brand and product names are trademarks or registered trademarks of their respective companies.

Printing History

Edition 1 - May 1996
Edition 2 - September 1996
Edition 3 - February 1998
Edition 4 - July 2000
Edition 5 - July 2001

Copyright Information

Agilent Technologies VISA User’s Guide
Edition 5
Copyright © 1984 -1988 Sun Microsystems, Inc.
All rights reserved.
Introduction
Introduction

This Agilent Technologies VISA User’s Guide describes the Agilent Virtual Instrument Software Architecture (VISA) library and shows how to use it to develop instrument drivers and I/O applications on Windows 95, Windows 98, Windows Me, Windows NT 4.0, and Windows 2000, and on HP-UX version 10.20. This chapter includes:

- What’s in This Guide?
- VISA Overview

NOTE

Before you can use VISA, you must install and configure VISA on your computer. See Agilent IO Libraries Installation and Configuration Guide for Windows for installation on Windows systems. See Agilent IO Libraries Installation and Configuration for HP-UX for installation on HP-UX systems.

This guide shows programming techniques using C/C++ and Visual Basic. Since VISA and SICL are different libraries, using VISA functions and SICL functions in the same I/O application is not supported. Unless indicated, Windows NT refers to Windows NT 4.0.
What’s in This Guide?

- **Chapter 1 - Introduction** describes the contents of this guide, provides an overview of VISA, and shows how to contact Agilent Technologies.

- **Chapter 2 - Building a VISA Application in Windows** describes how to build a VISA application in a Windows environment. An example program is provided to help you get started programming with VISA.

- **Chapter 3 - Building a VISA Application in HP-UX** describes how to build a VISA application in the HP-UX environment. An example program is provided to help you get started programming with VISA.

- **Chapter 4 - Programming with VISA** describes the basics of VISA and lists some example programs. The chapter also includes information on creating sessions, using formatted I/O, events, etc.

- **Chapter 5 - Programming via GPIB and VXI** gives guidelines to use VISA to communicate over the GPIB, GPIB-VXI, and VXI interfaces to instruments.

- **Chapter 6 - Programming via LAN** gives guidelines to use VISA to communicate over a LAN (Local Area Network) to instruments.

- **Chapter 7 - VISA Language Reference** provides an alphabetical reference of supported VISA functions.

- **Appendix A - VISA Library Information** lists VISA data types and their definitions, VISA error codes, and VISA directory information.

- **Appendix B - VISA Resource Classes** describes the six VISA Resource Classes, including attributes, events, and operations.

- **Glossary** includes a glossary of terms and their definitions.
VISA Overview

VISA is a part of the Agilent IO Libraries. The Agilent IO Libraries consists of two libraries: Agilent Virtual Instrument Software Architecture (VISA) and Agilent Standard Instrument Control Library (SICL). This guide describes VISA for supported Windows and HP-UX environments.

For information on using SICL in Windows, see the Agilent SICL User’s Guide for Windows. For information on using SICL in HP-UX, see the Agilent Standard Instrument Control Library User’s Guide for HP-UX. For information on the Agilent IO Libraries, see the Agilent IO Libraries Installation and Configuration Guide.

Using VISA and SICL

Agilent Virtual Instrument Software Architecture (VISA) is an IO library designed according to the VXIplug&play System Alliance that allows software developed from different vendors to run on the same system.

Use VISA if you want to use VXIplug&play instrument drivers in your applications, or if you want the I/O applications or instrument drivers that you develop to be compliant with VXIplug&play standards. If you are using new instruments or are developing new I/O applications or instrument drivers, we recommend you use Agilent VISA.

Agilent Standard Instrument Control Library (SICL) is an I/O library developed by Agilent that is portable across many I/O interfaces and systems. You can use Agilent SICL if you have been using SICL and want to remain compatible with software currently implemented in SICL.
VISA Support

Agilent VISA is an I/O library that can be used to develop I/O applications and instrument drivers that comply with the VXIplug&play standards. Applications and instrument drivers developed with VISA can execute on VXIplug&play system frameworks that have the VISA I/O layer. Therefore, software from different vendors can be used together on the same system.

VISA Support on Windows

This 32-bit version of VISA is supported on Windows 95, Windows 98, Windows Me, Windows NT, and Windows 2000. (Support for the 16-bit version of VISA was removed in version H.01.00 of the Agilent IO Libraries.) C, C++, and Visual Basic are supported on all these Windows versions.

For Windows, VISA is supported on the GPIB, VXI, GPIB-VXI, Serial (RS-232), and LAN interfaces. VISA for the VXI interface on Windows NT is shipped with the Agilent Embedded VXI Controller product only. LAN support from within VISA occurs via an address translation such that a GPIB interface can be accessed remotely over a computer network.

VISA Support on HP-UX

VISA is supported on the GPIB, VXI, GPIB-VXI, and LAN interfaces on HP-UX version 10.20. LAN support from within VISA occurs via an address translation such that a GPIB interface can be accessed remotely over a computer network.

VISA Users

VISA has two specific types of users. The first type is the instrumentation end user who wants to use VXIplug&play instrument drivers in his or her applications. The second type of user is the instrument driver or I/O application developer who wants to be compliant with VXIplug&play standards.

Software development using VISA is intended for instrument I/O and C/C++ or Visual Basic programmers who are familiar with the Windows 95, Windows 98, Windows Me, Windows 2000, Windows NT, or HP-UX environment. To perform VISA installation and configuration on Windows NT or HP-UX, you must have system administration privileges on the Windows NT system or super-user (root) privileges on the HP-UX system.
VISA Overview

VISA Documentation

This table shows associated documentation you can use when programming with Agilent VISA in the Windows or HP-UX environment.

<table>
<thead>
<tr>
<th>Document</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agilent IO Libraries Installation and Configuration Guide for Windows</td>
<td>Shows how to install, configure, and maintain the Agilent IO Libraries on Windows.</td>
</tr>
<tr>
<td>Agilent IO Libraries Installation and Configuration Guide for HP-UX</td>
<td>Shows how to install, configure, and maintain the Agilent IO Libraries on HP-UX.</td>
</tr>
<tr>
<td>VISA Online Help</td>
<td>Information is provided in the form of Windows Help.</td>
</tr>
<tr>
<td>VISA Example Programs</td>
<td>Example programs are provided online to help you develop VISA applications.</td>
</tr>
<tr>
<td>VXIplug&amp;play System Alliance VISA Library Specification 4.3</td>
<td>Specifications for VISA.</td>
</tr>
<tr>
<td>VXIbus Consortium specifications (when using VISA over LAN)</td>
<td>TCP/IP Instrument Protocol Specification - VXI-11, Rev. 1.0</td>
</tr>
<tr>
<td></td>
<td>TCP/IP-VXIbus Interface Specification - VXI-11.1, Rev. 1.0</td>
</tr>
<tr>
<td></td>
<td>TCP/IP-IEEE 488.1 Interface Specification - VXI-11.2, Rev. 1.0</td>
</tr>
<tr>
<td></td>
<td>TCP/IP-IEEE 488.2 Instrument Interface Specification - VXI-11.3, Rev. 1.0</td>
</tr>
</tbody>
</table>

Contacting Agilent

- In the USA and Canada, you can reach Agilent Technologies at these telephone numbers:
  USA: 1-800-452-4844
  Canada: 1-877-894-4414

- Outside the USA and Canada, contact your country’s Agilent support organization. A list of contact information for other countries is available on the Agilent web site:
  http://www.agilent.com/find/assist
Building a VISA Application in Windows
Building a VISA Application in Windows

This chapter gives guidelines to build a VISA application in a Windows environment. The chapter contains the following sections:

- Building a VISA Program (C/C++)
- Building a VISA Program (Visual Basic)
- Logging Error Messages
Building a VISA Program (C/C++)

This section gives guidelines to build VISA programs using C/C++ language, including:

- Compiling and Linking VISA Programs (C/C++)
- Example VISA Program (C/C++)

Compiling and Linking VISA Programs (C/C++)

This section provides a summary of important compiler-specific considerations for several C/C++ compiler products when developing Win32 applications.

Linking to VISA Libraries

Your application must link to one of the VISA import libraries as follows, assuming default installation directories.

- VISA on Windows 95, Windows 98, or Windows Me:
  
  - C:\Program Files\VISA\WIN95\LIB\MSC\VISA32.LIB (Microsoft compilers)
  - C:\Program Files\VISA\WIN95\LIB\BC\VISA32.LIB (Borland compilers)

- VISA on Windows NT or Windows 2000:
  
  - C:\Program Files\VISA\WINNT\LIB\MSC\VISA32.LIB (Microsoft compilers)
  - C:\Program Files\VISA\WINNT\LIB\BC\VISA32.LIB (Borland compilers)

Microsoft Visual C++ Version 6.0 Compilers

1. Select Project|Update All Dependencies from the menu.
2. Select Project|Settings from the menu and click the C/C++ button.
3. Select Code Generation from the Category list box and select Multi-Threaded using DLL from the Use Run-Time Libraries list box. (VISA requires these definitions for Win32.) Click OK to close the dialog boxes.
Building a VISA Application in Windows

Building a VISA Program (C/C++)

4 Select **Project | Settings** from the menu. Click the **Link** button and add *visa32.lib* to the **Object/Library Modules** list box. Optionally, you may add the library directly to your project file. Click **OK** to close the dialog boxes.

5 You may want to add the include file and library file search paths. They are set by:
   - Select **Tools | Options** from the menu.
   - Click the **Directories** button to set the include file path.
   - Select **Include Files** from the **Show Directories For** list box.
   - Click the **Add** button and type one of the following:
     - `C:\Program Files\VISA\WIN95\INCLUDE` OR
     - `C:\Program Files\VISA\WINNT\INCLUDE`.

6 Select **Library Files** from the **Show Directories For** list box.

7 Click the **Add** button and type one of the following:
   - `C:\Program Files\VISA\WIN95\LIB\MSC` OR
   - `C:\Program Files\VISA\WINNT\LIB\MSC`.

Borland C++
Version 4.0
Compilers

You may want to add the include file and library file search paths. They are set under the **Options | Project** menu selection. Double-click **Directories** from the **Topics** list box and add one of the following:

   - `C:\Program Files\VISA\WIN95\INCLUDE`
   - `C:\Program Files\VISA\WIN95\LIB\BC`

   OR

   - `C:\Program Files\VISA\WINNT\INCLUDE`
   - `C:\Program Files\VISA\WINNT\LIB\BC`
Example VISA Program (C/C++)

This section lists an example program called idn that queries a GPIB instrument for its identification string. This example assumes a Win32 Console Application using Microsoft or Borland C/C++ compilers on Windows.

- For VISA on Windows 95, Windows 98, and Windows Me, the idn example files are in \Program Files\VISA\WIN95\AGVISA\SAMPLES.
- For VISA on Windows NT or Windows 2000, the idn example files are in \Program Files\VISA\WINNT\AGVISA\SAMPLES.

Example C++ Program Source Code

The source file idn.c follows. An explanation of the various function calls in the example is provided directly after the program listing. If the program runs correctly, the following is an example of the output if connected to a 54601A oscilloscope. If the program does not run, see the Event Viewer for a list of run-time errors.

```
HEWLETT-PACKARD,54601A,0,1.7
/*idn.c
   This example program queries a GPIB device for an identification string and prints the results. Note that you must change the address. */

#include <visa.h>
#include <stdio.h>

void main () { 

ViSession defaultRM, vi;
char buf [256] = {0};

/* Open session to GPIB device at address 22 */
viOpenDefaultRM(&defaultRM);
viOpen(defaultRM, "GPIB0::22::INSTR", VI_NULL, VI_NULL, &vi);

/* Initialize device */
viPrintf(vi, "*RST\n");
/* Send an *IDN? string to the device */
viPrintf(vi, "*IDN?\n");
```

Building a VISA Application in Windows

Building a VISA Program (C/C++)

Example C/C++ Program Contents

A summary of the VISA function calls used in the example C/C++ program follows. For a more detailed explanation of VISA functionality, see Chapter 4 - Programming With VISA. See Chapter 7 - VISA Language Reference for more detailed information on these VISA function calls.

<table>
<thead>
<tr>
<th>Function(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>visa.h</td>
<td>This file is included at the beginning of the file to provide the function prototypes and constants defined by VISA.</td>
</tr>
<tr>
<td>ViSession</td>
<td>The ViSession is a VISA data type. Each object that will establish a communication channel must be defined as ViSession.</td>
</tr>
<tr>
<td>viOpenDefaultRM</td>
<td>You must first open a session with the default resource manager with the viOpenDefaultRM function. This function will initialize the default resource manager and return a pointer to that resource manager session.</td>
</tr>
<tr>
<td>viOpen</td>
<td>This function establishes a communication channel with the device specified. A session identifier that can be used with other VISA functions is returned. This call must be made for each device you will be using.</td>
</tr>
<tr>
<td>viPrintf and viScanf</td>
<td>These are the VISA formatted I/O functions that are patterned after those used in the C programming language. The viPrintf call sends the IEEE 488.2 *RST command to the instrument and puts it in a known state. The viPrintf call is used again to query for the device identification (*IDN?). The viScanf call is then used to read the results.</td>
</tr>
<tr>
<td>viClose</td>
<td>This function must be used to close each session. When you close a device session, all data structures that had been allocated for the session will be deallocated. When you close the default manager session, all sessions opened using that default manager session will be closed.</td>
</tr>
</tbody>
</table>
Building a VISA Program (Visual Basic)

This section gives guidelines to build a VISA program in the Visual Basic language, including:

- Visual Basic Programming Considerations
- Example VISA Program (Visual Basic)

Visual Basic Programming Considerations

Some considerations for programming in Visual Basic follow.

Required Module for a Visual Basic VISA Program

Before you can use VISA specific functions, your application must add the visa32.bas VISA Visual Basic module found in one of the following directories (assuming default installation directories). For Windows 2000/NT, C:\Program Files\VISA\winnt\include\. For Windows 95/98/Me, C:\Program Files\VISA\winnt\include\.

Installing the visa32.bas File

To install visa32.bas:

1. Select Project | Add Module from the menu
2. Select the Existing tab
3. Browse and select the visa32.bas file from applicable directory
4. Click the Open button

VISA Limitations in Visual Basic

VISA functions return a status code which indicates success or failure of the function. The only indication of an error is the value of returned status code. The VB Error variable is not set by any VISA function. Thus, you cannot use the 'ON ERROR' construct in VB or the value of the VB Error variable to catch VISA function errors.

VISA cannot callback to a VB function. Thus, you can only use the VI_QUEUE mechanism in viEnableEvent. There is no way to install a VISA event handler in VB.

VISA functions that take a variable number of parameters (viPrintf, viSscanf, viQueryf) are not callable from VB. Use the corresponding viVPrintf, viVSscanf and viVQueryf functions instead.

You cannot pass variables of type Variant to VISA functions. If you attempt this, the Visual Basic program will probably crash with a 'General Protection Fault' or an 'Access Violation'.
The functions `viVPrintf`, `viVscanf` and `viVqueryf` can be called from VB, but there are restrictions on the format conversions that can be used. Only one format conversion command can be specified in a format string (a format conversion command begins with the `%` character).

For example, the following is invalid:

```vbnet
status = viVPrintf(vi, "%lf%d" + Chr$(10), ...)  
```

Instead, you must make one call for each format conversion command, as shown in the following example:

```vbnet
status = viVPrintf(vi, "%lf" + Chr$(10), dbl_value)  
status = viVPrintf(vi, "%d" + Chr$(10), int_value)  
```

When reading to or writing from a numeric array, you must specify the first element of a numeric array as the `params` parameter. This passes the address of the first array element to the function. For example, the following code declares an array of 50 floating point numbers and then calls `viVPrintf` to write from the array.

```vbnet
Dim flt_array(50) As Double  
status = viVPrintf(id, "%,50f", dbl_array(0))  
```

When reading in a string value with `viVScanf` or `viVQueryf`, you must pass a fixed length string as the `params` parameter. To declare a fixed length string, instead of using the normal variable length declaration:

```vbnet
Dim strVal as String  
```

use the following declaration, where 40 is the fixed length.

```vbnet
Dim strVal as String * 40  
```
Example VISA Program (Visual Basic)

This section lists an example program called *idn* that queries a GPIB instrument for its identification string. This example builds a Standard EXE application for WIN32 programs using the Visual Basic 6.0 programming language.

For VISA on Windows 95, Windows 98, or Windows Me, the *idn* example files are in `C:\Program Files\VISA\WIN95\AGVISA\SAMPLES\vb\idn`. For VISA on Windows NT or Windows 2000, the *idn* example files are in `C:\Program Files\VISA\WINNT\AGVISA\SAMPLES\vb\idn`.

**Steps to Run the Program**

The steps to build and run the *idn* example program follow.

1. Connect an instrument to a GPIB interface that is compatible with IEEE 488.2.

2. Start the Visual Basic 6.0 application.

3. Start a new Visual Basic Standard EXE project. VB 6.0 will open a new Project1 project with a blank Form, Form1.

4. From the menu, select **Project** | **Add Module**, select the **Existing** tab, and browse to the *idn* directory.

5. The *idn* example files are located in directory `vb\samples\idn`. Select the file *idn.bas* and click **Open**. Since the Main( ) subroutine is executed when the program is run without requiring user interaction with a Form, Form1 may be deleted if desired. To do this, right-click Form1 in the Project Explorer window and select **Remove Form1**.

6. VISA applications in Visual Basic require the VISA Visual Basic (VB) declaration file *visa32.bas* in your VB project. This file contains the VISA function definitions and constant declarations needed to make VISA calls from Visual Basic.

**NOTE**

This example assumes you are building a new project (no *vbp* file exists for project). If you do not want to build the project from scratch, from the menu select **File** | **Open Project...** and select and open the *idn.vbp* file and skip to Step 9.
To add this module to your project in VB 6.0, from the menu select *Project | Add Module*, select the *Existing* tab, browse to the directory containing the VB Declaration file, select *visa32.bas*, and click *Open*.

The name and location of the VB declaration file depends on which operating system is used. Assuming the 'standard' VISA directory \Program Files\Visa or the 'standard' \VXIpnp directory \VXIpnp, the *visa32.bas* file can be located in one of these directories:

- \winnt\include\visa32.bas (Windows NT/2000)
- \win95\include\visa32.bas (Windows 95/98/Me)

At this point, the Visual Basic project can be run and debugged. You will need to change the VISA Interface Name and address in the code to match your device's configuration.

If you want to compile to an executable file, from the menu select *File | Make idn.exe...* and press *Open*. This will create *idn.exe* in the *idn* directory.

---

Example Program Source Code

An explanation of the various function calls in the example is provided after the program listing. If the program runs correctly, the following is an example of the output in a Message Box if connected to a 54601A oscilloscope.

```
HEWLETT-PACKARD,54601A,0,1.7
```

If the program does not run, see the Event Viewer for a list of run-time errors. The source file *idn.bas* follows.

```vbnet
Option Explicit
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
' idn.bas
' This example program queries a GPIB device for an identification
' string and prints the results. Note that you may have to change the
' VISA Interface Name and address for your device from "GPIB0" and "22",
' respectively.
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
Sub Main()
    Dim defrm As Long 'Session to Default Resource Manager
    Dim vi As Long 'Session to instrument
    Dim strRes As String * 200 'Fixed length string to hold results
```
' Open the default resource manager session
Call viOpenDefaultRM(defrm)

' Open the session to the resource
' The "GPIOB0" parameter is the VISA Interface name to a GPIB
' instrument as defined in
' Start | Programs | Agilent IO Libraries | IO Config
' Change this name to what you have defined your VISA Interface.
' "GPIOB0::22::INSTR" is the address string for the device.
' this address will be the same as seen in:
' Start | Programs | Agilent IO Libraries | VISA Assistant
' after the VISA Interface Name is defined in IO Config)
Call viOpen(defrm, "GPIOB0::22::INSTR", 0, 0, vi)

' Initialize device
Call viVPrintf(vi, "*RST" + Chr$(10), 0)

' Ask for the device's *IDN string.
Call viVPrintf(vi, "*IDN?" + Chr$(10), 0)

' Read the results as a string.
Call viVScanf(vi, "%t", strRes)

' Display the results
MsgBox "Result is: " + strRes, vbOKOnly, "*IDN? Result"

' Close the vi session and the resource manager session
Call viClose(vi)
Call viClose(defrm)
End Sub
Example Program Contents

A summary of the VISA function calls used in the example Visual Basic program follows. For a more detailed explanation of VISA functionality, see Chapter 4 - Programming with VISA. See Chapter 7 - VISA Language Reference for more detailed information on these VISA function calls.

<table>
<thead>
<tr>
<th>Function(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>viOpenDefaultRM</td>
<td>You must first open a session with the default resource manager with the <code>viOpenDefaultRM</code> function. This function will initialize the default resource manager and return a pointer (<code>defrm</code>) to that resource manager session.</td>
</tr>
<tr>
<td>viOpen</td>
<td>This function establishes a communication channel with the device specified. A session identifier (<code>vi</code>) that can be used with other VISA functions is returned. This call must be made for each device you will be using.</td>
</tr>
<tr>
<td>viVPrintf and viVScanf</td>
<td>These are the VISA formatted I/O functions. The <code>viVPrintf</code> call sends the IEEE 488.2 *RST command to the instrument (plus a linefeed character) and puts it in a known state. The <code>viVPrintf</code> call is used again to query for the device identification (*IDN?). The <code>viVScanf</code> call is then used to read the results (<code>strRes</code>) that are displayed in a Message Box.</td>
</tr>
<tr>
<td>viClose</td>
<td>This function must be used to close each session. When you close a device session, all data structures that had been allocated for the session will be deallocated. When you close the default manager session, all sessions opened using that default manager session will be closed.</td>
</tr>
</tbody>
</table>
Logging Error Messages

When developing or debugging your VISA application, you may want to view internal VISA messages while your application is running. You can do this by using the Message Viewer utility (for Windows 95/98/Me), the Event Viewer utility (for Windows 2000/NT), or the Debug Window (for Windows 95/98/2000/Me/NT). There are three choices for VISA logging:

- **Off** (default) for best performance
- **Event Viewer/Message Viewer**
- **Debug Window**

**Using the Event Viewer**

For Windows 2000 or Windows NT, the Event Viewer utility provides a way to view internal VISA error messages during application execution. Some of these internal messages do not represent programming errors and are actually error messages from VISA which are being handled internally by VISA. The process to use the Event Viewer is:

- Enable VISA logging from the Agilent IO Libraries Control, click **VISA Logging | Event Viewer**.
- Run your VISA program.
- View VISA error messages by running the Event Viewer. From the Agilent IO Libraries Control, click **Run Event Viewer**. VISA error messages will appear in the application log of the Event Viewer utility.

**Using the Message Viewer**

For Windows 95, Windows 98, or Windows Me, the Message Viewer utility provides a way to view internal VISA error messages during application execution. Some of these internal messages do not represent programming errors and are actually error messages from VISA which are being handled internally by VISA.

The Message Viewer utility must be run BEFORE you run your VISA application. However, the utility will receive messages while minimized. This utility also provides menu selections for saving the logged messages to a file and for clearing the message buffer.
Building a VISA Application in Windows

Logging Error Messages

The process to use the **Message Viewer** is:

- Enable VISA logging from the Agilent IO Libraries Control, click **VISA Logging | Message Viewer**.

- Start the **Message Viewer**. From the Agilent IO Libraries Control, click **Run Message Viewer**.

- Run your VISA program.

- View error messages in the **Message Viewer** window.

Using the Debug Window

- When VISA logging is directed to the **Debug Window**, VISA writes logging messages using the Win32 API call `OutputDebugString()`. The most common use for this feature is when debugging your VISA program using an application such as Microsoft Visual Studio. In this case, VISA messages will appear in the Visual Studio output window. The process to use the **Debug Window** is:

  - Enable VISA logging from the Agilent IO Libraries Control. Click **VISA Logging | Debug Window**.

  - Run your VISA program from Microsoft Visual Studio (or equivalent application).

  - View error messages in the Visual Studio (or equivalent) output window.
Building a VISA Application in HP-UX
Building a VISA Application in HP-UX

This chapter gives guidelines to build a VISA application on HP-UX version 10.20 or later. The chapter contains the following sections:

- Building a VISA Program in HP-UX
- Using Online Help
Building a VISA Program in HP-UX

This section lists and example program called `idn` that queries a GPIB instrument for its identification string. The `idn` example program is located in the following subdirectory:

```
opt/vxipnp/hpux/hpvisa/share/examples
```

Example Source Code

The source file `idn.c` follows. An explanation of the various function calls in the example is provided directly after the program listing.

```
/*idn.c
This program queries a GPIB device for an ID string and prints the results. Note that you must change the address. */

#include <visa.h>
#include <stdio.h>

void main () {

ViSession defaultRM, vi;
char buf [256] = {0};

/* Open session to GPIB device at address 22 */
viOpenDefaultRM(&defaultRM);
viOpen(defaultRM, "GPIB0::24::INSTR", VI_NULL, VI_NULL, &vi);

/* Initialize device */
viPrintf(vi, "*RST\n");

/* Send an *IDN? string to the device */
viPrintf(vi, "*IDN?\n");

/* Read results */
viScanf(vi, "%t", buf);

/* Print results */
printf ("Instrument identification string: %s\n", buf);

/* Close sessions */
viClose(vi);
viClose(defaultRM);
}
```
Example Program Contents

A summary of the VISA function calls used in the example program follows. For a more detailed explanation of VISA functionality, see Chapter 4 - Programming with VISA. See Chapter 7 - VISA Language Reference for more detailed information on these VISA calls.

`visa.h`. This file is included at the beginning of the file to provide the function prototypes and constants defined by VISA.

`ViSession`. The `ViSession` is a VISA data type. Each object that will establish a communication channel must be defined as `ViSession`.

`viOpenDefaultRM`. You must first open a session with the default resource manager with the `viOpenDefaultRM` function. This function will initialize the default resource manager and return a pointer to that resource manager session.

`viOpen`. This function establishes a communication channel with the device specified. A session identifier that can be used with other VISA functions is returned. This call must be made for each device you will be using.

`viPrintf` and `viScanf`. These are the VISA formatted I/O functions that are patterned after those used in the C programming language. The `viPrintf` call sends the IEEE 488.2 *RST command to the instrument and puts it in a known state. The `viPrintf` call is used again to query for the device identification (*IDN?). The `viScanf` call is then used to read the results.

`viClose`. This function must be used to close each session. When you close a device session, all data structures that had been allocated for the session will be deallocated. When you close the default manager session, all sessions opened using that default manager session will be closed.

Running the Example Program

To run the `idn` example program, type the program name at the command prompt. For example:

```plaintext
idn
```

If the program run correctly, the following is an example of the output if connected to a 54601A oscilloscope:

```plaintext
Hewlett-Packard,54601A,0,1.7
```
If you have problems running the `idn` example program, first check to make sure the device address specified in your program is correct. If the program still does not run, check the I/O configuration. See the Agilent I/O Libraries Installation and Configuration Guide for HP-UX for information on I/O configuration.

**Compiling and Linking a VISA Program**

You can create your VISA applications in ANSI C or C++. When compiling and linking a C program that uses VISA, use the `-lvisa` command line option to link in the VISA library routines. The following example creates the `idn` executable file:

```
cc -Aa -o idn idn.c -lvisa
```

- The `-Aa` option indicates ANSI C
- The `-o` option creates an executable file called `idn`
- The `-l` option links in the VISA library

**Logging Error Messages**

To view any VISA internal errors that may occur on HP-UX, edit the `/etc/opt/vxipnp/hpux/hpvisa/hpvisa.ini` file. Change the `ErrorLog=` line in this file to the following:

```
ErrorLog=true
```

The error messages, if any, will be then be printed to `stderr`. 
Using Online Help

Online help for VISA on HP-UX is provided with Bristol Technology's HyperHelp Viewer, or in the form of HP-UX manual pages (man pages), as explained in the following subsections.

Using the HyperHelp Viewer

The Bristol Technology HyperHelp Viewer allows you to view the VISA functions online. To start the HyperHelp Viewer with the VISA help file, type:

```
hyperhelp/opt/hyperhelp/visahelp.hlp
```

When you start the Viewer, you can also specify any of the following options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-k keyword</code></td>
<td>Opens the Viewer and searches for the specified keyword.</td>
</tr>
<tr>
<td><code>-p partial_keyword</code></td>
<td>Opens the Viewer and searches for a specific partial keyword.</td>
</tr>
<tr>
<td><code>-s viewmode</code></td>
<td>Opens the Viewer in the specified viewmode. If 1 is specified as the viewmode, the Viewer is shared by all applications. If 0 is specified, a separate Viewer is opened for each application (default).</td>
</tr>
<tr>
<td><code>-display display</code></td>
<td>Opens the Viewer on the specified display.</td>
</tr>
</tbody>
</table>

Using HP-UX Manual Pages

To use manual pages, type the HP-UX man command followed by the VISA function name:

```
man function
```

The following are examples of selecting online help on VISA functions:

```
man viPrintf
man viScanf
man viPeek
```
Programming with VISA
Programming with VISA

This chapter describes how to program with VISA. The basics of VISA are described, including formatted I/O, events and handlers, attributes, and locking. Example programs are also provided and can be found in the SAMPLES subdirectory on Windows environments or in the examples subdirectory on HP-UX.

See Appendix A - VISA Library Information for the specific location of the example programs on your operating system. For specific details on VISA functions, see Chapter 7 - VISA Language Reference. This chapter contains the following sections:

- VISA Resources and Attributes
- Using Sessions
- Sending I/O Commands
- Using Events and Handlers
- Trapping Errors
- Using Locks
Programming with VISA
VISA Resources and Attributes

VISA Resources and Attributes

This section introduces VISA resources and attributes, including:

- VISA Resources
- VISA Attributes

VISA Resources

In VISA, a resource is defined as any device (such as a voltmeter) with which VISA can provide communication. VISA defines six resource classes that a complete VISA system, fully compliant with the VXIplug&play Systems Alliance specification, can implement. Each resource class includes:

- Attributes to determine the state of a resource or session or to set a resource or session to a specified state.
- Events for communication with applications.
- Operations (functions) that can be used for the resource class.

A summary description of each resource class supported by Agilent VISA follows. See Appendix B - VISA Resource Classes for a description of the attributes, events, and operations for each resource class.

**NOTE**

Although the Servant Device-Side (SERVANT) resource is defined by the VISA specification, the SERVANT resource is not supported by Agilent VISA. See Appendix B - VISA Resource Classes for a description of the SERVANT resource.

<table>
<thead>
<tr>
<th>Resource Class</th>
<th>Interface Types</th>
<th>Resource Class Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instrument Control (INSTR)</td>
<td>Generic, GPIB, GPIB-VXI, Serial, TCP/IP, VXI</td>
<td>Device operations (reading, writing, triggering, etc.).</td>
</tr>
<tr>
<td>GPIB Bus Interface (INTFC)</td>
<td>Generic, GPIB</td>
<td>Raw GPIB interface operations (reading, writing, triggering, etc.).</td>
</tr>
<tr>
<td>Memory Access (MEMACC)</td>
<td>Generic, GPIB-VXI, VXI</td>
<td>Address space of a memory-mapped bus such as the VXIbus.</td>
</tr>
</tbody>
</table>
Programming with VISA

VISA Resources and Attributes

<table>
<thead>
<tr>
<th>Resource Class</th>
<th>Interface Types</th>
<th>Resource Class Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VXI Mainframe Backplane (BACKPLANE)</td>
<td>Generic, GPIB-VXI, VXI (GPIB-VXI BACKPLANE not supported)</td>
<td>VXI-defined operations and properties of each backplane (or chassis) in a VXIbus system.</td>
</tr>
<tr>
<td>Servant Device-Side Resource (SERVANT)</td>
<td>GPIB, VXI, TCPIP (not supported)</td>
<td>Operations and properties of the capabilities of a device and a device's view of the system in which it exists.</td>
</tr>
<tr>
<td>TCPIP Socket (SOCKET)</td>
<td>Generic, TCPIP</td>
<td>Operations and properties of a raw network socket connection using TCPIP.</td>
</tr>
</tbody>
</table>

VISA Attributes

Attributes are associated with resources or sessions. You can use attributes to determine the state of a resource or session or to set a resource or session to a specified state.

For example, you can use the `viGetAttribute` function to read the state of an attribute for a specified session, event context, or find list. There are read only (RO) and read/write (RW) attributes. Use the `viSetAttribute` function to modify the state of a read/write attribute for a specified session, event context, or find list.

The pointer passed to `viGetAttribute` must point to the exact type required for that attribute: `ViUInt16`, `ViInt32`, etc. For example, when reading an attribute state that returns a `ViUInt16`, you must declare a variable of that type and use it for the returned data. If `ViString` is returned, you must allocate an array and pass a pointer to that array for the returned data.

Example: Reading a VISA Attribute

This example reads the state of the `VI_ATTR_TERMCHAR_EN` attribute and changes it if it is not true.

```c
ViBoolean state, newstate;
newstate=VI_TRUE;
viGetAttribute(vi, VI_ATTR_TERMCHAR_EN, &state);
if (state err !=VI_TRUE) viSetAttribute(vi,
    VI_ATTR_TERMCHAR_EN, newstate);
```
Using Sessions

This section shows how to use VISA sessions, including:

- Including the VISA Declarations File (C/C++)
- Adding the visa32.bas File (Visual Basic)
- Opening a Session to a Resource
- Addressing a Session
- Closing a Session
- Searching for Resources

Including the VISA Declarations File (C/C++)

For C and C++ programs, you must include the `visa.h` header file at the beginning of every file that contains VISA function calls:

```c
#include "visa.h"
```

This header file contains the VISA function prototypes and the definitions for all VISA constants and error codes. The `visa.h` header file also includes the `visatype.h` header file.

The `visatype.h` header file defines most of the VISA types. The VISA types are used throughout VISA to specify data types used in the functions. For example, the `viOpenDefaultRM` function requires a pointer to a parameter of type `ViSession`. If you find `ViSession` in the `visatype.h` header file, you will find that `ViSession` is eventually typed as an unsigned long. VISA types are also listed in Appendix A - VISA System Information.

Adding the visa32.bas File (Visual Basic)

You must add the `visa32.bas` Basic Module file to your Visual Basic Project. The `visa32.bas` file contains the VISA function prototypes and definitions for all VISA constants and error codes.

Opening a Session

A `session` is a channel of communication. Sessions must first be opened on the default resource manager, and then for each resource you will be using.

- A `resource manager session` is used to initialize the VISA system. It is a parent session that knows about all the opened sessions. A resource manager session must be opened before any other session can be opened.
A resource session is used to communicate with a resource on an interface. A session must be opened for each resource you will be using. When you use a session you can communicate without worrying about the type of interface to which it is connected. This insulation makes applications more robust and portable across interfaces.

Resource Manager Sessions

There are two parts to opening a communications session with a specific resource. First, you must open a session to the default resource manager with the `viOpenDefaultRM` function. The first call to this function initializes the default resource manager and returns a session to that resource manager session. You only need to open the default manager session once. However, subsequent calls to `viOpenDefaultRM` returns a unique session to the same default resource manager resource.

Resource Sessions

Next, you open a session with a specific resource with the `viOpen` function. This function uses the session returned from `viOpenDefaultRM` and returns its own session to identify the resource session. The following shows the function syntax:

```c
viOpenDefaultRM(sesn);
viOpen(sesn, rsrcName, accessMode, timeout, vi);
```

The session returned from `viOpenDefaultRM` must be used in the `sesn` parameter of the `viOpen` function. The `viOpen` function then uses that session and the resource address specified in the `rsrcName` parameter to open a resource session. The `vi` parameter in `viOpen` returns a session identifier that can be used with other VISA functions.

Your program may have several sessions open at the same time by creating multiple session identifiers by calling the `viOpen` function multiple times. The following table summarizes the parameters in the previous function calls.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sesn</code></td>
<td>A session returned from the <code>viOpenDefaultRM</code> function that identifies the resource manager session.</td>
</tr>
<tr>
<td><code>rsrcName</code></td>
<td>A unique symbolic name of the resource (resource address).</td>
</tr>
</tbody>
</table>
Example: Opening a Resource Session

This example shows one way of opening resource sessions with a GPIB multimeter and a GPIB-VXI scanner. The example first opens a session with the default resource manager. The session returned from the resource manager and a resource address is then used to open a session with the GPIB device at address 22. That session will now be identified as \textit{dmm} when using other VISA functions.

The session returned from the resource manager is then used again with another resource address to open a session with the GPIB-VXI device at primary address 9 and VXI logical address 24. That session will now be identified as \textit{scanner} when using other VISA functions. See "Addressing a Session" for information on addressing particular devices.

\verbatim{c++}
ViSession defaultRM, dmm, scanner;
.
viOpenDefaultRM(&defaultRM);
viOpen(defaultRM, "GPIB0::22::INSTR", VI_NULL,
   VI_NULL, &dmm);
viOpen(defaultRM, "GPIB-VXI0::24::INSTR", VI_NULL,
   VI_NULL, &scanner);
.
viClose(scanner);
viClose(dmm);
viClose(defaultRM);
\endverbatim

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{accessMode}</td>
<td>Specifies the modes by which the resource is to be accessed. The value \texttt{VI_EXCLUSIVE_LOCK} is used to acquire an exclusive lock immediately upon opening a session. If a lock cannot be acquired, the session is closed and an error is returned. The \texttt{VI_LOAD_CONFIG} value is used to configure attributes specified by some external configuration utility. If this value is not used, the session uses the default values provided by this specification. Multiple access modes can be used simultaneously by specifying a &quot;bit-wise OR&quot; of the values. (Must use \texttt{VI_NULL} in VISA 1.0.).</td>
</tr>
<tr>
<td>\textit{timeout}</td>
<td>If the \textit{accessMode} parameter requires a lock, this parameter specifies the absolute time period (in milliseconds) that the resource waits to get unlocked before this operation returns an error. Otherwise, this parameter is ignored. (Must use \texttt{VI_NULL} in VISA 1.0.)</td>
</tr>
<tr>
<td>\textit{vi}</td>
<td>This is a pointer to the session identifier for this particular resource session. This pointer will be used to identify this resource session when using other VISA functions.</td>
</tr>
</tbody>
</table>
Programming with VISA
Using Sessions

Addressing a Session

As shown in the previous section, the \texttt{rsrcName} parameter in the \texttt{viOpen} function is used to identify a specific resource. This parameter consists of the VISA interface name and the resource address. The interface name is determined when you run the VISA configuration utility. This name is usually the interface type followed by a number.

The following table illustrates the format of the \texttt{rsrcName} for different interface types. \texttt{INSTR} is an optional parameter that indicates that you are communicating with a resource that is of type \texttt{INSTR}, meaning instrument. The keywords are:

- \texttt{ASRL} establishes communication with asynchronous serial devices.
- \texttt{GPIB} establishes communication with GPIB devices or interfaces.
- \texttt{GPIB-VXI} is used for GPIB-VXI controllers.
- \texttt{TCPIP} establishes communication with LAN instruments.
- \texttt{VXI} is used for VXI instruments.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Typical Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASRL</td>
<td>\texttt{ASRL[board]} [::\texttt{INSTR}]</td>
</tr>
<tr>
<td>GPIB</td>
<td>\texttt{GPIB[board]} ::primary address [::secondary address] [::\texttt{INSTR}]</td>
</tr>
<tr>
<td>GPIB</td>
<td>\texttt{GPIB[board]} ::\texttt{INTFC}</td>
</tr>
<tr>
<td>GPIB-VXI</td>
<td>\texttt{GPIB-VXI[board]} ::VXI logical address [::\texttt{INSTR}]</td>
</tr>
<tr>
<td>GPIB-VXI</td>
<td>\texttt{GPIB-VXI[board]} ::\texttt{MEMACC}</td>
</tr>
<tr>
<td>GPIB-VXI</td>
<td>\texttt{GPIB-VXI[board]} [::VXI logical address] ::\texttt{BACKPLANE}</td>
</tr>
<tr>
<td>TCPIP</td>
<td>\texttt{TCPIP[board]} ::host address [::LAN device name] ::\texttt{INSTR}</td>
</tr>
<tr>
<td>TCPIP</td>
<td>\texttt{TCPIP[board]} ::host address ::port ::\texttt{SOCKET}</td>
</tr>
<tr>
<td>VXI</td>
<td>\texttt{VXI[board]} ::VXI logical address [::\texttt{INSTR}]</td>
</tr>
<tr>
<td>VXI</td>
<td>\texttt{VXI[board]} ::\texttt{MEMACC}</td>
</tr>
<tr>
<td>VXI</td>
<td>\texttt{VXI[board]} [::VXI logical address] ::\texttt{BACKPLANE}</td>
</tr>
</tbody>
</table>
The following table describes the parameters used above.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>board</td>
<td>This optional parameter is used if you have more than one interface of the same type. The default value for <code>board</code> is 0.</td>
</tr>
<tr>
<td>host address</td>
<td>The IP address (in dotted decimal notation) or the name of the host computer/gateway.</td>
</tr>
<tr>
<td>LAN device name</td>
<td>The assigned name for a LAN device. The default is <code>inst()</code>.</td>
</tr>
<tr>
<td>port</td>
<td>The port number to use for a TCP/IP Socket connection.</td>
</tr>
<tr>
<td>primary address</td>
<td>This is the primary address of the GPIB device.</td>
</tr>
<tr>
<td>secondary address</td>
<td>This optional parameter is the secondary address of the GPIB device. If no <code>secondary address</code> is specified, none is assumed.</td>
</tr>
<tr>
<td>VXI logical address</td>
<td>This is the logical address of the VXI instrument.</td>
</tr>
</tbody>
</table>

Some examples of valid symbolic names follow.

<table>
<thead>
<tr>
<th>Address String</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VXI0::1::INSTR</td>
<td>A VXI device at logical address 1 in VXI interface VXI0.</td>
</tr>
<tr>
<td>GPIB-VXI::9::INSTR</td>
<td>A VXI device at logical address 9 in a GPIB-VXI controlled VXI system.</td>
</tr>
<tr>
<td>GPIB::1::0::INSTR</td>
<td>A GPIB device at primary address 1 and secondary address 0 in GPIB interface 0.</td>
</tr>
<tr>
<td>ASRL1::INSTR</td>
<td>A serial device located on port 1.</td>
</tr>
<tr>
<td>VXI::MEMACC</td>
<td>Board-level register access to the VXI interface.</td>
</tr>
<tr>
<td>GPIB-VXI1::MEMACC</td>
<td>Board-level register access to GPIB-VXI interface number 1.</td>
</tr>
<tr>
<td>GPIB2::INTFC</td>
<td>Interface or raw resource for GPIB interface 2.</td>
</tr>
</tbody>
</table>
Example: Opening a Session

This example shows one way to open a resource session with the GPIB device at primary address 23.

```c
ViSession defaultRM, vi;
.
.viOpenDefaultRM(&defaultRM);
.viOpen(defaultRM, "GPIB0::23::INSTR", VI_NULL,
    VI_NULL,&vi);
.
.viClose(vi);
.viClose(defaultRM);
```

Closing a Session

The `viClose` function must be used to close each session. You can close the specific resource session, which will free all data structures that had been allocated for the session. If you close the default resource manager session, all sessions opened using that resource manager session will be closed.

Since system resources are also used when searching for resources (viFindRsrc), the `viClose` function needs to be called to free up find lists. See "Searching for Resources" for more information on closing find lists.
Searching for Resources

When you open the default resource manager, you are opening a parent session that knows about all the other resources in the system. Since the resource manager session knows about all resources, it has the ability to search for specific resources and open sessions to these resources. You can, for example, search an interface for devices and open a session with one of the devices found.

Use the `viFindRsrc` function to search an interface for device resources. This function finds matches and returns the number of matches found and a handle to the resources found. If there are more matches, use the `viFindNext` function with the handle returned from `viFindRsrc` to get the next match:

```c
viFindRsrc(sesn, expr, findList, retcnt, instrDesc);
```

```c
viFindNext(findList, instrDesc);
```

```c
viClose(findList);
```

Where the parameters are defined as follows.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sesn</code></td>
<td>The resource manager session.</td>
</tr>
<tr>
<td><code>expr</code></td>
<td>The expression that identifies what to search (see table that follows).</td>
</tr>
<tr>
<td><code>findList</code></td>
<td>A handle that identifies this search. This handle will then be used as an input to the <code>viFindNext</code> function when finding the next match.</td>
</tr>
<tr>
<td><code>retcnt</code></td>
<td>A pointer to the number of matches found.</td>
</tr>
<tr>
<td><code>instrDesc</code></td>
<td>A pointer to a string identifying the location of the match. Note that you must allocate storage for this string.</td>
</tr>
</tbody>
</table>

The handle returned from `viFindRsrc` should be closed to free up all the system resources associated with the search. To close the find object, pass the `findList` to the `viClose` function.
Use the `expr` parameter of the `viFindRsrc` function to specify the interface to search. You can search for devices on the specified interface. Use the following table to determine what to use for your `expr` parameter.

NOTE
Because VISA interprets strings as regular expressions, the string `GPIB?*INSTR` applies to both GPIB and GPIB-VXI devices.

<table>
<thead>
<tr>
<th>Interface</th>
<th><code>expr</code> Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIB</td>
<td><code>GPIB[0-9]*::?*INSTR</code></td>
</tr>
<tr>
<td>VXI</td>
<td><code>VXI?*INSTR</code></td>
</tr>
<tr>
<td>GPIB-VXI</td>
<td><code>GPIB-VXI?*INSTR</code></td>
</tr>
<tr>
<td>GPIB and GPIB-VXI</td>
<td><code>GPIB?*INSTR</code></td>
</tr>
<tr>
<td>All VXI</td>
<td><code>?*VXI[0-9]*::?*INSTR</code></td>
</tr>
<tr>
<td>ASRL</td>
<td><code>ASRL[0-9]*::?*INSTR</code></td>
</tr>
<tr>
<td>All</td>
<td><code>?*INSTR</code></td>
</tr>
</tbody>
</table>

Example: Searching VXI Interface for Resources

This example searches the VXI interface for resources. The number of matches found is returned in `nmatches`, and `matches` points to the string that contains the matches found. The first call returns the first match found, the second call returns the second match found, etc. `VI_FIND_BUFLEN` is defined in the `visa.h` declarations file.

```c
ViChar buffer [VI_FIND_BUFLEN];
ViRsrc matches=buffer;
ViUInt32 nmatches;
ViFindList list;
.
viFindRsrc(defaultRM, "VXI?*INSTR", &list, &nmatches, matches);
.
viFindNext(list, matches);
.
viClose(list);
```
Sending I/O Commands

This section gives guidelines to send I/O commands, including:

- Types of I/O
- Using Formatted I/O
- Using Non-Formatted I/O

Types of I/O

Once you have established a communications session with a device, you can start communicating with that device using VISA’s I/O routines. VISA provides both formatted and non-formatted I/O routines.

- **Formatted I/O** converts mixed types of data under the control of a format string. The data is buffered, thus optimizing interface traffic.

- **Non-formatted I/O** sends or receives raw data to or from a device. With non-formatted I/O, no format or conversion of the data is performed. Thus, if formatted data is required, it must be done by the user.

You can choose between VISA’s formatted and non-formatted I/O routines. However, since the non-formatted I/O performs the low-level I/O, you should not mix formatted I/O and non-formatted I/O in the same session. See the following sections for descriptions and examples using formatted I/O and non-formatted I/O in VISA.

Using Formatted I/O

The VISA formatted I/O mechanism is similar to the C `stdio` mechanism. The VISA formatted I/O functions are `viPrintf`, `viQueryf`, and `viScanf`. There are also two non-buffered and non-formatted I/O functions that synchronously transfer data, called `viRead` and `viWrite` and two that asynchronously transfer data, called `viReadAsync` and `viWriteAsync`.

These are raw I/O functions and do not intermix with the formatted I/O functions. See “Using Non-Formatted I/O” in this chapter. See Chapter 7 - VISA Language Reference for more information on how data is converted under the control of the format string.
Programming with VISA

Sending I/O Commands

Formatted I/O Functions

As noted, the VISA formatted I/O functions are \texttt{viPrintf}, \texttt{viQueryf}, and \texttt{viScanf}.

- The \texttt{viPrintf} functions format according to the format string and send data to a device. The \texttt{viPrintf} function sends separate \texttt{arg} parameters, while the \texttt{viVPrintf} function sends a list of parameters in \texttt{params}:

\begin{verbatim}
viPrintf(vi, writeFmt[, arg1][, arg2][, ...]);
viVPrintf(vi, writeFmt, params);
\end{verbatim}

- The \texttt{viScanf} functions receive and convert data according to the format string. The \texttt{viScanf} function receives separate \texttt{arg} parameters, while the \texttt{viVScanf} function receives a list of parameters in \texttt{params}:

\begin{verbatim}
viScanf(vi, readFmt[, arg1][, arg2][, ...]);
viVScanf(vi, readFmt, params);
\end{verbatim}

- The \texttt{viQueryf} functions format and send data to a device and then immediately receive and convert the response data. Hence, the \texttt{viQueryf} function is a combination of the \texttt{viPrintf} and \texttt{viScanf} functions. Similarly, the \texttt{viVQueryf} function is a combination of the \texttt{viVPrintf} and \texttt{viVScanf} functions. The \texttt{viQueryf} function sends and receives separate \texttt{arg} parameters, while the \texttt{viVQueryf} function sends and receives a list of parameters in \texttt{params}:

\begin{verbatim}
viQueryf(vi, writeFmt, readFmt[, arg1][, arg2][, ...]);
viVQueryf(vi, writeFmt, readFmt, params);
\end{verbatim}

Formatted I/O Conversion

The formatted I/O functions convert data under the control of the format string. The format string specifies how the argument is converted before it is input or output. The format specifier sequence consists of a \texttt{\%} (percent) followed by an optional modifier(s), followed by a format code.

\begin{verbatim}
\%[modifiers]format code
\end{verbatim}

Zero or more modifiers may be used to change the meaning of the format code. Modifiers are only used when sending or receiving formatted I/O. To send formatted I/O, the asterisk (*) can be used to indicate that the number is taken from the next argument.
However, when the asterisk is used when receiving formatted I/O, it indicates that the assignment is suppressed and the parameter is discarded. Use the pound sign (#) when receiving formatted I/O to indicate that an extra argument is used. The following are supported modifiers. See the `viPrintf` function in Chapter 7 - VISA Language Reference for additional enhanced modifiers (@1, @2, @3, @H, @Q, or @B).

- **Field Width.** Field width is an optional integer that specifies how many characters are in the field. If the `viPrintf` or `viQueryf` (writeFmt) formatted data has fewer characters than specified in the field width, it will be padded on the left, or on the right if the – flag is present.

  You can use an asterisk (*) in place of the integer in `viPrintf` or `viQueryf` (writeFmt) to indicate that the integer is taken from the next argument. For the `viScanf` or `viQueryf` (readFmt) functions, you can use a # sign to indicate that the next argument is a reference to the field width.

  The field width modifier is only supported with `viPrintf` and `viQueryf` (writeFmt) format codes d, f, s, and `viScanf` and `viQueryf` (readFmt) format codes c, s, and [ ].

**Example: Using Field Width Modifier**

The following example pads `numb` to six characters and sends it to the session specified by `vi`:

```c
int numb = 61;
viPrintf(vi, "%6d\n", numb);
```

Inserts four spaces, for a total of 6 characters: 61

- **Precision.** Precision is an optional integer preceded by a period. This modifier is only used with the `viPrintf` and `viQueryf` (writeFmt) functions. The meaning of this argument is dependent on the conversion character used. You can use an asterisk (*) in place of the integer to indicate the integer is taken from the next argument.
This example converts `numb` so that there are only two digits to the right of the decimal point and sends it to the session specified by `vi`:

```c
float numb = 26.9345;
viPrintf(vi, "%f\n", numb);
```

Sends: `26.93`

**Argument Length Modifier.** The meaning of the optional argument length modifier `h`, `l`, `L`, `z`’’ or `Z` is dependent on the conversion character, as listed in the following table. Note that `z` and `Z` are not ANSI C standard modifiers.

<table>
<thead>
<tr>
<th>Argument Length Modifier</th>
<th>Format Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>h</code></td>
<td><code>d, b, B</code></td>
<td>Corresponding argument is a short integer or a reference to a short integer for <code>d</code>. For <code>b</code> or <code>B</code>, the argument is the location of a block of data or a reference to a data array. (<code>B</code> is only used with <code>viPrintf</code> or <code>viQueryf</code> (writeFmt).)</td>
</tr>
<tr>
<td><code>l</code></td>
<td><code>d, f, b, B</code></td>
<td>Corresponding argument is a long integer or a reference to a long integer for <code>d</code>. For <code>f</code>, the argument is a double float or a reference to a double float. For <code>b</code> or <code>B</code>, the argument is the location of a block of data or a reference to a data array. (<code>B</code> is only used with <code>viPrintf</code> or <code>viQueryf</code> (writeFmt).)</td>
</tr>
</tbody>
</table>
Programming with VISA

Sending I/O Commands

, Array Size. The comma operator is a format modifier that allows you to read or write a comma-separated list of numbers (only valid with \%d and \%f format codes). It is a comma followed by an integer. The integer indicates the number of elements in the array. The comma operator has the format of \,dd where \texttt{dd} is the number of elements to read or write.

For \texttt{viPrintf} or \texttt{viQueryf} (writeFmt), you can use an asterisk (\*) in place of the integer to indicate that the integer is taken from the next argument. For \texttt{viscanf} or \texttt{viQueryf} (readFmt), you can use a # sign to indicate that the next argument is a reference to the array size.

Example: Using Array Size Modifier

This example specifies a comma-separated list to be sent to the session specified by \texttt{vi}:

\begin{verbatim}
int list[5]={101,102,103,104,105};
viPrintf(vi, "%,5d\n", list);
\end{verbatim}

Sends: 101,102,103,104,105

■ Special Characters. Special formatting character sequences will send special characters. The following describes the special characters and what will be sent.

The format string for \texttt{viPrintf} and \texttt{viQueryf} (writeFmt) puts a special meaning on the newline character (\n). The newline character in the format string flushes the output buffer to the device.

<table>
<thead>
<tr>
<th>Argument Length Modifier</th>
<th>Format Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>f</td>
<td>Corresponding argument is a long double or a reference to a long double.</td>
</tr>
<tr>
<td>z</td>
<td>b, B</td>
<td>Corresponding argument is an array of floats or a reference to an array of floats. (B is only used with \texttt{viPrintf} or \texttt{viQueryf} (writeFmt).)</td>
</tr>
<tr>
<td>Z</td>
<td>b, B</td>
<td>Corresponding argument is an array of double floats or a reference to an array of double floats. (B is only used with \texttt{viPrintf} or \texttt{viQueryf} (writeFmt).)</td>
</tr>
</tbody>
</table>
All characters in the output buffer will be written to the device with an END indicator included with the last byte (the newline character). This means you can control at what point you want the data written to the device. If no newline character is included in the format string, the characters converted are stored in the output buffer. It will require another call to `viPrintf`, `viQueryf` (`writeFmt`), or `viFlush` to have those characters written to the device.

This can be very useful in queuing up data to send to a device. It can also raise I/O performance by doing a few large writes instead of several smaller writes. The * while using the `viScanf` functions acts as an assignment suppression character. The input is not assigned to any parameters and is discarded.

The grouping operator () in a regular expression has the highest precedence, the + and * operators in a regular expression have the next highest precedence after the grouping operator, and the or operator | in a regular expression has the lowest precedence. Some example expressions follow the table.

<table>
<thead>
<tr>
<th>Special Characters and Operators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>Matches any one character.</td>
</tr>
<tr>
<td>\</td>
<td>Makes the character that follows it an ordinary character instead of special character. For example, when a question mark follows a backslash (e.g., <code>\?</code>), it matches the '?' character instead of any one character.</td>
</tr>
<tr>
<td>[list]</td>
<td>Matches any one character from the enclosed list. A hyphen can be used to match a range of characters.</td>
</tr>
<tr>
<td>[^list]</td>
<td>Matches any character not in the enclosed list. A hyphen can be used to match a range of characters.</td>
</tr>
<tr>
<td>*</td>
<td>Matches 0 or more occurrences of the preceding character or expression.</td>
</tr>
<tr>
<td>+</td>
<td>Matches 1 or more occurrences of the preceding character or expression.</td>
</tr>
<tr>
<td>exp</td>
<td>exp</td>
</tr>
<tr>
<td>Special Characters and Operators</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>(exp)</strong></td>
<td>Grouping characters or expressions.</td>
</tr>
<tr>
<td><strong>&quot;</strong></td>
<td>Sends a blank space.</td>
</tr>
<tr>
<td><strong>\n</strong></td>
<td>Sends the ASCII line feed character. The END identifier will also be sent.</td>
</tr>
<tr>
<td><strong>\r</strong></td>
<td>Sends an ASCII carriage return character.</td>
</tr>
<tr>
<td><strong>\t</strong></td>
<td>Sends an ASCII TAB character.</td>
</tr>
<tr>
<td><strong>###</strong></td>
<td>Sends ASCII character specified by octal value.</td>
</tr>
<tr>
<td><strong>&quot;</strong></td>
<td>Sends the ASCII double quote character.</td>
</tr>
<tr>
<td>*<em>*</em></td>
<td>Sends a backslash character.</td>
</tr>
</tbody>
</table>

### Example Expression

<table>
<thead>
<tr>
<th>Example Expression</th>
<th>Sample Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIB?*INSTR</td>
<td>Matches GPIB0::2::INSTR, GPIB1::1::1::INSTR, and GPIB-VXI1::8::INSTR</td>
</tr>
<tr>
<td>GPIB[0-9]*::?*INSTR</td>
<td>Matches GPIB0::2::INSTR and GPIB1::1::1::INSTR but not GPIB-VXI1::8::INSTR</td>
</tr>
<tr>
<td>GPIB[0-9]::?*INSTR</td>
<td>Matches GPIB0::2::INSTR and GPIB1::1::1::INSTR but not GPIB12::8::INSTR</td>
</tr>
<tr>
<td>GPIB[^0]::?*INSTR</td>
<td>Matches GPIB1::1::1::INSTR but not GPIB0::2::INSTR or GPIB12::8::INSTR</td>
</tr>
<tr>
<td>VXI?*INSTR</td>
<td>Matches VXI0::1::INSTR but not GPIB-VXI0::1::INSTR</td>
</tr>
<tr>
<td>GPIB-VXI?*INSTR</td>
<td>Matches GPIB-VXI0::1::INSTR but not VXI0::1::INSTR</td>
</tr>
<tr>
<td>?<em>VXI[0-9]</em>::?*INSTR</td>
<td>Matches VXI0::1::INSTR and GPIB-VXI0::1::INSTR</td>
</tr>
<tr>
<td>ASRL[0-9]*::?*INSTR</td>
<td>Matches ASRL1::INSTR but not VXI0::5::INSTR</td>
</tr>
<tr>
<td>ASRL1+::INSTR</td>
<td>Matches ASRL1::INSTR and ASRL11::INSTR but not ASRL2::INSTR</td>
</tr>
</tbody>
</table>
# Programming with VISA

## Sending I/O Commands

<table>
<thead>
<tr>
<th>Example Expression</th>
<th>Sample Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>(GPIB</td>
<td>VXI)?*INSTR</td>
</tr>
<tr>
<td>(GPIB0</td>
<td>VXI0)::1::INSTR</td>
</tr>
<tr>
<td>?*INSTR</td>
<td>Matches all INSTR (device) resources</td>
</tr>
<tr>
<td>?<em>VXI[0-9]</em>::*?*MEMACC</td>
<td>Matches VXI0::MEMACC and GPIB-VXI1::MEMACC</td>
</tr>
<tr>
<td>VXI0::??</td>
<td>Matches VXI0::1::INSTR, VXI0::2::INSTR, and VXI0::MEMACC</td>
</tr>
<tr>
<td>??</td>
<td>Matches all resources</td>
</tr>
</tbody>
</table>

**Format Codes.** This table summarizes the format codes for sending and receiving formatted I/O.

<table>
<thead>
<tr>
<th>Format Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>viPrintf/viVPrintf and viQueryf/viVqueryf <em>(writeFmt)</em></td>
<td></td>
</tr>
<tr>
<td>d, i</td>
<td>Corresponding argument is an integer.</td>
</tr>
<tr>
<td>f</td>
<td>Corresponding argument is a double.</td>
</tr>
<tr>
<td>c</td>
<td>Corresponding argument is a character.</td>
</tr>
<tr>
<td>s</td>
<td>Corresponding argument is a pointer to a null terminated string.</td>
</tr>
<tr>
<td>%</td>
<td>Sends an ASCII percent (%) character.</td>
</tr>
<tr>
<td>o, u, x, X</td>
<td>Corresponding argument is an unsigned integer.</td>
</tr>
<tr>
<td>e, E, g, G</td>
<td>Corresponding argument is a double.</td>
</tr>
<tr>
<td>n</td>
<td>Corresponding argument is a pointer to an integer.</td>
</tr>
<tr>
<td>b, B</td>
<td>Corresponding argument is the location of a block of data.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>viPrintf/viVPrintf and viQueryf/viVqueryf <em>(readFmt)</em></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>d,i,n</td>
<td>Corresponding argument must be a pointer to an integer.</td>
</tr>
<tr>
<td>e,f,g</td>
<td>Corresponding argument must be a pointer to a float.</td>
</tr>
<tr>
<td>c</td>
<td>Corresponding argument is a pointer to a character sequence.</td>
</tr>
<tr>
<td>s,t,T</td>
<td>Corresponding argument is a pointer to a string.</td>
</tr>
<tr>
<td>o,u,x</td>
<td>Corresponding argument must be a pointer to an unsigned integer.</td>
</tr>
</tbody>
</table>
### Example: Receiving Data From a Session

This example receives data from the session specified by the `vi` parameter and converts the data to a string.

```
char data[180];
viScanf(vi, "%t", data);
```

### Formatted I/O Buffers

The VISA software maintains both a read and write buffer for formatted I/O operations. Occasionally, you may want to control the actions of these buffers. You can modify the size of the buffer using the `viSetBuf` function. See Chapter 7 - VISA Language Reference for more information on this function.

The write buffer is maintained by the `viPrintf` or `viQueryf` functions. The buffer queues characters to send to the device so that they are sent in large blocks, thus increasing performance. The write buffer automatically flushes when it sends a newline character from the format string. It may occasionally be flushed at other non-deterministic times, such as when the buffer fills.

When the write buffer flushes, it sends its contents to the device. If you set the `VI_ATTR_WR_BUF_OPER_MODE` attribute to `VI_FLUSH_ON_ACCESS`, the write buffer will also be flushed every time a `viPrintf` or `viQueryf` operation completes. See "VISA Attributes" in this chapter for information on setting VISA attributes.

The read buffer is maintained by the `viScanf` and `viQueryf` functions. It queues the data received from a device until it is needed by the format string. Flushing the read buffer destroys the data in the buffer and guarantees that the next call to `viScanf` or `viQueryf` reads data directly from the device rather than data that was previously queued.

If you set the `VI_ATTR_RD_BUF_OPER_MODE` attribute to `VI_FLUSH_ON_ACCESS`, the read buffer will be flushed every time a `viScanf` or `viQueryf` operation completes. See "VISA Attributes" in this chapter for information on setting VISA attributes.

### Format Codes

<table>
<thead>
<tr>
<th>Format Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[</td>
<td>Corresponding argument must be a character pointer.</td>
</tr>
<tr>
<td>b</td>
<td>Corresponding argument is a pointer to a data array.</td>
</tr>
</tbody>
</table>

---

Formatted I/O Buffers

This example receives data from the session specified by the `vi` parameter and converts the data to a string.

```
char data[180];
viScanf(vi, "%t", data);
```
You can manually flush the read and write buffers using the `viFlush` function. Flushing the read buffer also includes reading all pending response data from a device. If the device is still sending data, the flush process will continue to read data from the device until it receives an `END` indicator from the device.

Example: Sending and Receiving Formatted I/O

This C program example shows sending and receiving formatted I/O. The example opens a session with a GPIB device and sends a comma operator to send a comma-separated list. This example program is intended to show specific VISA functionality and does not include error trapping. Error trapping, however, is good programming practice and is recommended in your VISA applications. See "Trapping Errors" in this chapter.

This example program is installed on your system in the SAMPLES subdirectory on Windows environments or in the examples subdirectory on HP-UX. See Appendix A - VISA Library Information for locations of example programs on your operating system.

```c
#include <visa.h>
#include <stdio.h>

void main () {
    ViSession defaultRM, vi;
    double res;
    double list [2] = {1, 0.001};

    /* Open session to GPIB device at address 22 */
    viOpenDefaultRM(&defaultRM);
    viOpen(defaultRM, "GPIB0::22::INSTR", VI_NULL, VI_NULL, &vi);

    /* Initialize device */
    viPrintf(vi, "*RST\n");

    /* Set up device and send comma separated list */
    viPrintf(vi, "CALC:DBM:REF 50\n");
    viPrintf(vi, "MEAS:VOLT:AC? %,2f\n", list);
}```
Using Non-Formatted I/O

There are two non-buffered, non-formatted I/O functions that synchronously transfer data called `viRead` and `viWrite`. Also, there are two non-formatted I/O functions that asynchronously transfer data called `viReadAsync` and `viWriteAsync`. These are raw I/O functions and do not intermix with the formatted I/O functions.

Non-Formatted I/O Functions

The non-formatted I/O functions follow. For more information, see the `viRead`, `viWrite`, `viReadAsync`, `viWriteAsync`, and `viTerminate` functions in *Chapter 7 - VISA Language Reference*.

- **viRead.** The `viRead` function synchronously reads raw data from the session specified by the `vi` parameter and stores the results in the location where `buf` is pointing. Only one synchronous read operation can occur at any one time.

  ```c
  viRead(vi, buf, count, retCount)
  ```

- **viWrite.** The `viWrite` function synchronously sends the data pointed to by `buf` to the device specified by `vi`. Only one synchronous write operation can occur at any one time.

  ```c
  viWrite(vi, buf, count, retCount)
  ```

- **viReadAsync.** The `viReadAsync` function asynchronously reads raw data from the session specified by the `vi` parameter and stores the results in the location where `buf` is pointing. This operation normally returns before the transfer terminates. Thus, the operation returns `jobId`, which you can use with either `viTerminate` to abort the operation or with an I/O completion event to identify which asynchronous read operation completed.

  ```c
  viReadAsync(vi, buf, count, jobId)
  ```
Programming with VISA
Sending I/O Commands

viWriteAsync. The `viWriteAsync` function asynchronously sends the data pointed to by `buf` to the device specified by `vi`. This operation normally returns before the transfer terminates. Thus, the operation returns `jobId`, which you can use with either `viTerminate` to abort the operation or with an I/O completion event to identify which asynchronous write operation completed.

```c
viWriteAsync(vi, buf, count, jobId);
```

Example: Using Non-Formatted I/O Functions

This example program illustrates using non-formatted I/O functions to communicate with a GPIB device. This example program is intended to show specific VISA functionality and does not include error trapping. Error trapping, however, is good programming practice and is recommended in your VISA applications. See "Trapping Errors" in this chapter.

```c
/*nonfmtio.c
   This example program measures the AC voltage on a multimeter and prints the results. You may need to change the device address. */

#include <visa.h>
#include <stdio.h>

void main () {
    ViSession defaultRM, vi;
    char strres [20];
    unsigned long actual;

    /* Open session to GPIB device at address 22 */
    viOpenDefaultRM(&defaultRM);
    viOpen(defaultRM, "GPIB0::22::INSTR", VI_NULL, VI_NULL, &vi);

    /* Initialize device */
    viWrite(vi, (ViBuf)"*RST\n", 5, &actual);

    /* Set up device and take measurement */
    viWrite(vi, (ViBuf)"CALC:DBM:REF 50\n", 16, &actual);
    viWrite(vi, (ViBuf)"MEAS:VOLT:AC? 1, 0.001\n", 23, &actual);

    /* Read results */
    viRead(vi, (ViBuf)strres, 20, &actual);
```
/* NULL terminate the string */
strres[actual]=0;

/* Print results */
printf("Measurement Results: %s\n", strres);

/* Close session */
viClose(vi);
viClose(defaultRM);
Using Events and Handlers

This section gives guidelines to use events and handlers, including:

- Events and Attributes
- Using the Callback Method
- Using the Queuing Method

Events and Attributes

Events are special occurrences that require attention from your application. Event types include Service Requests (SRQs), interrupts, and hardware triggers. Events will not be delivered unless the appropriate events are enabled.

**NOTE**

VISA cannot callback to a Visual Basic function. Thus, you can only use the **queuing** mechanism in **viEnableEvent**. There is no way to install a VISA event handler in Visual Basic.

Event Notification

There are two ways you can receive notification that an event has occurred:

- Install an event handler with **viInstallhandler**, and enable one or several events with **viEnableEvent**. If the event was enabled with a handler, the specified event handler will be called when the specified event occurs. This is called a **callback**.

**NOTE**

VISA cannot callback to a Visual Basic function. This means that you can only use the **VI_QUEUE** mechanism in **viEnableEvent**. There is no way to install a VISA event handler in Visual Basic.

- Enable one or several events with **viEnableEvent** and call the **viWaitOnEvent** function. The **viWaitOnEvent** function will suspend the program execution until the specified event occurs or the specified timeout period is reached. This is called **queuing**.
The queuing and callback mechanisms are suitable for different programming styles. The queuing mechanism is generally useful for non-critical events that do not need immediate servicing. The callback mechanism is useful when immediate responses are needed. These mechanisms work independently of each other, so both can be enabled at the same time. By default, a session is not enabled to receive any events by either mechanism.

The `viEnableEvent` operation can be used to enable a session to respond to a specified event type using either the queuing mechanism, the callback mechanism, or both. Similarly, the `viDisableEvent` operation can be used to disable one or both mechanisms. Because the two methods work independently of each other, one can be enabled or disabled regardless of the current state of the other.

### Events That can be Enabled

The following table shows the events that are implemented for Agilent VISA for each resource class, where AP = Access Privilege, RO - Read Only, and RW = Read/Write. Note that some resource classes/events, such as the SERVANT class are not implemented by Agilent VISA and are not listed in the following tables.

Once the application has received an event, information about that event can be obtained by using the `viGetAttribute` function on that particular event context. Use the VISA `viReadSTB` function to read the status byte of the service request.

<table>
<thead>
<tr>
<th>Event Attribute</th>
<th>Description</th>
<th>AP</th>
<th>Data Type</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>VI_ATTR_EVENT_TYPE</code></td>
<td>Unique logical identifier of the event.</td>
<td>RO</td>
<td>ViEventType</td>
<td><code>VI_EVENT_SERVICE_REQ</code></td>
</tr>
</tbody>
</table>
Programming with VISA
Using Events and Handlers

**VI_EVENT_VXI_SIGP**
Notification that a VXIbus signal or VXIbus interrupt was received from the device.

<table>
<thead>
<tr>
<th>Event Attributes</th>
<th>Description</th>
<th>AP</th>
<th>Data Type</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>VI_ATTR_EVENT_TYPE</td>
<td>Unique logical identifier of the event.</td>
<td>RO</td>
<td>ViEventType</td>
<td>VI_EVENT_VXI_STOP</td>
</tr>
<tr>
<td>VI_ATTR_SIGP_STATUS_ID</td>
<td>The 16-bit Status/ID value retrieved during the IACK cycle or from the Signal register.</td>
<td>RO</td>
<td>ViUInt16</td>
<td>0 to FFFFh</td>
</tr>
</tbody>
</table>

**VI_EVENT_TRIG**
Notification that a trigger interrupt was received from the device. For VISA, the only triggers that can be sensed are VXI hardware triggers on the assertion edge (SYNC and ON trigger protocols only).

<table>
<thead>
<tr>
<th>Event Attributes</th>
<th>Description</th>
<th>AP</th>
<th>Data Type</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>VI_ATTR_EVENT_TYPE</td>
<td>Unique logical identifier of the event.</td>
<td>RO</td>
<td>ViEventType</td>
<td>VI_EVENT_TRIG</td>
</tr>
<tr>
<td>VI_ATTR_RECV_TRIG_ID</td>
<td>The identifier of the triggering mechanism on which the specified trigger event was received.</td>
<td>RO</td>
<td>ViInt16</td>
<td>VI_TRIG_TTL0 to VI_TRIG_TTL7; VI_TRIG_ECL0 to VI_TRIG_ECL1</td>
</tr>
</tbody>
</table>

* Agilent VISA can also return VI_TRIG_PANEL_IN (exception to the VISA Specification)

**VI_EVENT_IO_COMPLETION**
Notification that an asynchronous operation has completed.

<table>
<thead>
<tr>
<th>Event Attributes</th>
<th>Description</th>
<th>AP</th>
<th>Data Type</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>VI_ATTR_EVENT_TYPE</td>
<td>Unique logical identifier of the event.</td>
<td>RO</td>
<td>ViEventType</td>
<td>VI_EVENT_IO_COMPLETION</td>
</tr>
<tr>
<td>VI_ATTR_STATUS</td>
<td>Return code of the asynchronous I/O operation that has completed</td>
<td>RO</td>
<td>ViStatus</td>
<td>N/A</td>
</tr>
<tr>
<td>VI_ATTR_JOB_ID</td>
<td>Job ID of the asynchronous operation that has completed</td>
<td>RO</td>
<td>ViJobId</td>
<td>N/A</td>
</tr>
<tr>
<td>VI_ATTR_BUFFER</td>
<td>Address of a buffer that was used in an asynchronous operation.</td>
<td>RO</td>
<td>ViBuf</td>
<td>N/A</td>
</tr>
</tbody>
</table>
### VI_EVENT_IO_COMPLETION
Notification that an asynchronous operation has completed.

<table>
<thead>
<tr>
<th>Event Attribute</th>
<th>Description</th>
<th>AP</th>
<th>Data Type</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>VI_ATTR_RET_COUNT</td>
<td>Actual number of elements that were asynchronously transferred.</td>
<td>RO</td>
<td>ViUInt32</td>
<td>0 to FFFFFFFFh</td>
</tr>
<tr>
<td>VI_ATTR_OPER_NAME</td>
<td>Name of the operation generating the event.</td>
<td>RO</td>
<td>ViString</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Memory Access (MEMACC) Resource Event

<table>
<thead>
<tr>
<th>Event Attribute</th>
<th>Description</th>
<th>AP</th>
<th>Data Type</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>VI_ATTR_EVENT_TYPE</td>
<td>Unique logical identifier of the event.</td>
<td>RO</td>
<td>ViEventType</td>
<td>VI_EVENT_IO_COMPLETION</td>
</tr>
<tr>
<td>VI_ATTR_STATUS</td>
<td>Return code of the asynchronous I/O operation that has completed.</td>
<td>RO</td>
<td>ViStatus</td>
<td>N/A</td>
</tr>
<tr>
<td>VI_ATTR_JOB_ID</td>
<td>Job ID of the asynchronous operation that has completed.</td>
<td>RO</td>
<td>ViJobId</td>
<td>N/A</td>
</tr>
<tr>
<td>VI_ATTR_BUFFER</td>
<td>Address of a buffer that was used in an asynchronous operation.</td>
<td>RO</td>
<td>ViBuf</td>
<td>N/A</td>
</tr>
<tr>
<td>VI_ATTR_RET_COUNT</td>
<td>Actual number of elements that were asynchronously transferred.</td>
<td>RO</td>
<td>ViUInt32</td>
<td>0 to FFFFFFFFh</td>
</tr>
<tr>
<td>VI_ATTR_OPER_NAME</td>
<td>Name of the operation generating the event.</td>
<td>RO</td>
<td>ViString</td>
<td>N/A</td>
</tr>
</tbody>
</table>
### GPIB Bus Interface (INTFC) Resource Events

**VI_EVENT_GPIB_CIC**
Notification that the GPIB controller has gained or lost CIC (controller in charge) status

<table>
<thead>
<tr>
<th>Event Attribute</th>
<th>Description</th>
<th>AP</th>
<th>Data Type</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>VI_ATTR_EVENT_TYPE</td>
<td>Unique logical identifier of the event.</td>
<td>RO</td>
<td>ViEventType</td>
<td>VI_EVENT_GPIB_CIC</td>
</tr>
<tr>
<td>VI_ATTR_GPIB_RECV_CIC_STATE</td>
<td>Controller has become controller-in-charge.</td>
<td>RO</td>
<td>ViBoolean</td>
<td>VI_TRUE, VI_FALSE</td>
</tr>
</tbody>
</table>

**VI_EVENT_GPIB_TALK**
Notification that the GPIB controller has been addressed to talk

<table>
<thead>
<tr>
<th>Event Attribute</th>
<th>Description</th>
<th>AP</th>
<th>Data Type</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>VI_ATTR_EVENT_TYPE</td>
<td>Unique logical identifier of the event.</td>
<td>RO</td>
<td>ViEventType</td>
<td>VI_EVENT_GPIB_TALK</td>
</tr>
</tbody>
</table>

**VI_EVENT_GPIB_LISTEN**
Notification that the GPIB controller has been addressed to listen.

<table>
<thead>
<tr>
<th>Event Attribute</th>
<th>Description</th>
<th>AP</th>
<th>Data Type</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>VI_ATTR_EVENT_TYPE</td>
<td>Unique logical identifier of the event.</td>
<td>RO</td>
<td>ViEventType</td>
<td>VI_EVENT_GPIB_LISTEN</td>
</tr>
</tbody>
</table>

**VI_EVENT_CLEAR**
Notification that the GPIB controller has been sent a device clear message.

<table>
<thead>
<tr>
<th>Event Attribute</th>
<th>Description</th>
<th>AP</th>
<th>Data Type</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>VI_ATTR_EVENT_TYPE</td>
<td>Unique logical identifier of the event.</td>
<td>RO</td>
<td>ViEventType</td>
<td>VI_EVENT_CLEAR</td>
</tr>
</tbody>
</table>
### VI_EVENT_TRIGGER
Notification that a trigger interrupt was received from the interface.

<table>
<thead>
<tr>
<th>Event Attribute</th>
<th>Description</th>
<th>AP</th>
<th>Data Type</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>VI_ATTR_EVENT_TYPE</td>
<td>Unique logical identifier of the event.</td>
<td>RO</td>
<td>ViEventType</td>
<td>VI_EVENT_TRIG</td>
</tr>
<tr>
<td>VI_ATTR_RECV_TRIG_ID</td>
<td>The identifier of the triggering mechanism on which the specified trigger event was received.</td>
<td>RO</td>
<td>ViInt16</td>
<td>VI_TRIG_SW</td>
</tr>
</tbody>
</table>

### VI_EVENT_IO_COMPLETION
Notification that an asynchronous operation has completed.

<table>
<thead>
<tr>
<th>Event Attribute</th>
<th>Description</th>
<th>AP</th>
<th>Data Type</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>VI_ATTR_EVENT_TYPE</td>
<td>Unique logical identifier of the event.</td>
<td>RO</td>
<td>ViEventType</td>
<td>VI_EVENT_IO_COMPLETION</td>
</tr>
<tr>
<td>VI_ATTR_STATUS</td>
<td>Return code of the asynchronous I/O operation that has completed.</td>
<td>RO</td>
<td>ViStatus</td>
<td>N/A</td>
</tr>
<tr>
<td>VI_ATTR_JOB_ID</td>
<td>Job ID of the asynchronous operation that has completed.</td>
<td>RO</td>
<td>ViJobId</td>
<td>N/A</td>
</tr>
<tr>
<td>VI_ATTR_BUFFER</td>
<td>Address of buffer used in an asynchronous operation.</td>
<td>RO</td>
<td>ViBuf</td>
<td>N/A</td>
</tr>
<tr>
<td>VI_ATTR_RET_COUNT</td>
<td>Actual number of elements that were asynchronously transferred.</td>
<td>RO</td>
<td>ViUInt32</td>
<td>0 to FFFFFFFFh</td>
</tr>
<tr>
<td>VI_ATTR_OPER_NAME</td>
<td>The name of the operation generating the event.</td>
<td>RO</td>
<td>ViString</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Programming with VISA
Using Events and Handlers

VXI Mainframe Backplane (BACKPLANE) Resource Events

**VI_EVENT_TRIG**
Notification that a trigger interrupt was received from the backplane. For VISA, the only triggers that can be sensed are VXI hardware triggers on the assertion edge (SYNC and ON trigger protocols only).

<table>
<thead>
<tr>
<th>Event Attribute</th>
<th>Description</th>
<th>AP</th>
<th>Data Type</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>VI_ATTR_EVENT_TYPE</td>
<td>Unique logical identifier of the event.</td>
<td>RO</td>
<td>ViEventTypes</td>
<td>VI_EVENT_TRIG</td>
</tr>
<tr>
<td>VI_ATTR_RECV_TRIG_ID</td>
<td>The identifier of the triggering mechanism on which the specified trigger event was received.</td>
<td>RO</td>
<td>ViInt16</td>
<td>VI_TRIG_TTL0 to VI_TRIG_TTL7; VI_TRIG_ECL0 to VI_TRIG_ECL1</td>
</tr>
</tbody>
</table>

**VI_EVENT_VXI_VME_SYSFAIL**
Notification that the VXI/VME SYSFAIL\(^*\) line has been asserted.

<table>
<thead>
<tr>
<th>Event Attribute</th>
<th>Description</th>
<th>AP</th>
<th>Data Type</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>VI_ATTR_EVENT_TYPE</td>
<td>Unique logical identifier of the event.</td>
<td>RO</td>
<td>ViEventTypes</td>
<td>VI_EVENT_VXI_VME_SYSFAIL</td>
</tr>
</tbody>
</table>

**VI_EVENT_VXI_VME_SYSRESET**
Notification that the VXI/VME SYSRESET\(^*\) line has been reset.

<table>
<thead>
<tr>
<th>Event Attribute</th>
<th>Description</th>
<th>AP</th>
<th>Data Type</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>VI_ATTR_EVENT_TYPE</td>
<td>Unique logical identifier of the event.</td>
<td>RO</td>
<td>ViEventTypes</td>
<td>VI_EVENT_VXI_VME_SYSRESET</td>
</tr>
</tbody>
</table>

TCPIP Socket (SOCKET) Resource Event

**VI_EVENT_IO_COMPLETION**
Notification that an asynchronous operation has completed.

<table>
<thead>
<tr>
<th>Event Attributes</th>
<th>Description</th>
<th>AP</th>
<th>Data Type</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>VI_ATTR_EVENT_TYPE</td>
<td>Unique logical identifier of the event.</td>
<td>RO</td>
<td>ViEventTypes</td>
<td>VI_EVENT_IO_COMPLETION</td>
</tr>
<tr>
<td>VI_ATTR_STATUS</td>
<td>Return code of the asynchronous I/O operation that has completed</td>
<td>RO</td>
<td>ViStatus</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Example: Reading Event Attributes

Once you have decided which attribute to check, you can read the attribute using the `viGetAttribute` function. The following example shows one way you could check which trigger line fired when the `VI_EVENT_TRIG` event was delivered.

Note that the `context` parameter is either the event `context` passed to your event handler, or the `outcontext` specified when doing a `wait on event`. See “VISA Attributes” in this chapter for more information on reading attribute states.

```c
ViInt16 state;
.
.
viGetAttribute(context, VI_ATTR_RECV_TRIG_ID, &state);
```

Using the Callback Method

The callback method of event notification is used when an immediate response to an event is required. To use the callback method for receiving notification that an event has occurred, you must do the following. Then, when the enabled event occurs, the installed event handler is called.

- Install an event handler with the `viInstallHandler` function
- Enable one or several events with the `viEnableEvent` function
Programming with VISA
Using Events and Handlers

Example: Using the Callback Method
This example shows one way you can use the callback method.

```c
ViStatus _VI_FUNC my_handler (ViSession vi,
   ViEventType eventType, ViEvent context, ViAddr usrHandle) {

   /* your event handling code here */

   return VI_SUCCESS;
}
main(){
   ViSession vi;
   ViAddr addr=0;
   
   viInstallHandler(vi, VI_EVENT_SERVICE_REQ, my_handler,
      addr);
   viEnableEvent(vi, VI_EVENT_SERVICE_REQ, VI_HNDLR,
      VI_NULL);
   
   /* your code here */
   
   viDisableEvent(vi, VI_EVENT_SERVICE_REQ, VI_HNDLR);
   viUninstallHandler(vi, VI_EVENT_SERVICE_REQ, my_handler,
      addr);
   
}
```

Installing Handlers
VISA allows applications to install multiple handlers for an event type on the same session. Multiple handlers can be installed through multiple invocations of the `viInstallHandler` operation, where each invocation adds to the previous list of handlers.

If more than one handler is installed for an event type, each of the handlers is invoked on every occurrence of the specified event(s). VISA specifies that the handlers are invoked in Last In First Out (LIFO) order. Use the following function when installing an event handler:

```c
viInstallHandler(vi, eventType, handler, userHandle);
```

Where the parameters are defined as follows:
The userHandle parameter allows you to assign a value to be used with the handler on the specified session. Thus, you can install the same handler for the same event type on several sessions with different userHandle values. The same handler is called for the specified event type.

However, the value passed to userHandle is different. Therefore the handlers are uniquely identified by the combination of the handler and the userHandle. This may be useful when you need a different handling method depending on the userHandle.

Example: Installing an Event Handler

This example shows how to install an event handler to call my_handler when a Service Request occurs. Note that VI_EVENT_SERVICE_REQ must also be an enabled event with the viEnableEvent function for the service request event to be delivered.

```c
viInstallHandler(vi, VI_EVENT_SERVICE_REQ, my_handler, addr);
```

Use the viUninstallHandler function to uninstall a specific handler. Or you can use wildcards (VI_ANY_HNDLR in the handler parameter) to uninstall groups of handlers. See viUninstallHandler in Chapter 7 - VISA Language Reference for more details on this function.

Writing the Handler

The handler installed needs to be written by the programmer. The event handler typically reads an associated attribute and performs some sort of action. See the event handler in the example program later in this section.
Programming with VISA

Using Events and Handlers

Enabling Events

Before an event can be delivered, it must be enabled using the `viEnableEvent` function. This function causes the application to be notified when the enabled event has occurred. Where the parameters are:

\[
\text{viEnableEvent}(vi, \text{eventType}, \text{mechanism}, \text{context});
\]

Using `VI_QUEUE` in the `mechanism` parameter specifies a queuing method for the events to be handled. If you use both `VI_QUEUE` and one of the mechanisms listed above, notification of events will be sent to both locations. See the next subsection for information on the queuing method.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vi</code></td>
<td>The session on which the handler will be installed.</td>
</tr>
<tr>
<td><code>eventType</code></td>
<td>The type of event to enable.</td>
</tr>
<tr>
<td><code>mechanism</code></td>
<td>The mechanism by which the event will be enabled. It can be enabled in several different ways. You can use <code>VI_HNDLR</code> in this parameter to specify that the installed handler will be called when the event occurs. Use <code>VI_SUSPEND_HNDLR</code> in this parameter which puts the events in a queue and waits to call the installed handlers until <code>viEnableEvent</code> is called with <code>VI_HNDLR</code> specified in the <code>mechanism</code> parameter. When <code>viEnableEvent</code> is called with <code>VI_HNDLR</code> specified, the handler for each queued event will be called.</td>
</tr>
<tr>
<td><code>context</code></td>
<td>Not used in VISA 1.0. Use <code>VI_NULL</code>.</td>
</tr>
</tbody>
</table>

Example: Enabling a Hardware Trigger Event

This example illustrates enabling a hardware trigger event.

\[
\begin{align*}
\text{viInstallHandler}(vi, \text{VI_EVENT_TRIG}, \text{my\_handler}, &\text{&addr}); \\
\text{viEnableEvent}(vi, \text{VI\_EVENT\_TRIG}, \text{VI\_HNDLR}, \text{VI\_NULL});
\end{align*}
\]

The `VI\_HNDLR` mechanism specifies that the handler installed for `VI\_EVENT\_TRIG` will be called when a hardware trigger occurs.

If you specify `VI\_ALL\_ENABLE\_EVENTS` in the `eventType` parameter, all events that have previously been enabled on the specified session will be enabled for the `mechanism` specified in this function call.

Use the `viDisableEvent` function to stop servicing the event specified.
Example: Trigger Callback

This example program installs an event handler and enables the trigger event. When the event occurs, the installed event handler is called. This program is intended to show specific VISA functionality and does not include error trapping. Error trapping, however, is good programming practice and is recommended in your VISA applications. See "Trapping Errors" in this chapter.

This example program is installed on your system in the SAMPLES subdirectory on Windows environments or in the examples subdirectory on HP-UX. See Appendix A - VISA Library Information for locations of example programs on your operating system.

/* evnthdlr.c
   This example program illustrates installing an event handler to be called when a trigger interrupt occurs.
   Note that you may need to change the address. */

#include <visa.h>
#include <stdio.h>

/* trigger event handler */
ViStatus _VI_FUNCH myHdlr(ViSession vi, ViEventType eventType, ViEvent ctx, ViAddr userHdlr){

ViInt16 trigId;

/* make sure it is a trigger event */
if(eventType!=VI_EVENT_TRIG){
    /* Stray event, so ignore */
    return VI_SUCCESS;
}

/* print the event information */
printf("Trigger Event Occurred!\n");
printf("...Original Device Session = %ld\n", vi);

/* get the trigger that fired */
viGetAttribute(ctx, VIATTR_RECV_TRIG_ID, &trigId);
printf("Trigger that fired: ");
switch(trigId){
case VI_TRIG_TTL0:
    printf("TTL0");
    break;
default:
    printf("<other 0x%x>", trigId); 
    break;
}

/* evnthdlr.c
   This example program illustrates installing an event handler to be called when a trigger interrupt occurs.
   Note that you may need to change the address. */

#include <visa.h>
#include <stdio.h>

/* trigger event handler */
ViStatus _VI_FUNCH myHdlr(ViSession vi, ViEventType eventType, ViEvent ctx, ViAddr userHdlr){

ViInt16 trigId;

/* make sure it is a trigger event */
if(eventType!=VI_EVENT_TRIG){
    /* Stray event, so ignore */
    return VI_SUCCESS;
}

/* print the event information */
printf("Trigger Event Occurred!\n");
printf("...Original Device Session = %ld\n", vi);

/* get the trigger that fired */
viGetAttribute(ctx, VIATTR_RECV_TRIG_ID, &trigId);
printf("Trigger that fired: ");
switch(trigId){
case VI_TRIG_TTL0:
    printf("TTL0");
    break;
default:
    printf("<other 0x%x>", trigId); 
    break;
}
Programming with VISA

Using Events and Handlers

```c
printf("\n");
return VI_SUCCESS;
}

void main(){
  ViSession defaultRM, vi;

  /* open session to VXI device */
  viOpenDefaultRM(&defaultRM);
  viOpen(defaultRM, "VXI0::24::INSTR", VI_NULL, VI_NULL, &vi);

  /* select trigger line TTL0 */
  viSetAttribute(vi, VI_ATTR_TRIG_ID, VI_TRIG_TTL0);
  /* install the handler and enable it */
  viInstallHandler(vi, VI_EVENT_TRIG, myHdlr, (ViAddr)10);
  viEnableEvent(vi, VI_EVENT_TRIG, VI_HNDLR, VI_NULL);
  /* fire trigger line, twice */
  viAssertTrigger(vi, VI_TRIG_PROT_SYNC);
  viAssertTrigger(vi, VI_TRIG_PROT_SYNC);

  /* unenable and unistall the handler */
  viDisableEvent(vi, VI_EVENT_TRIG, VI_HNDLR);
  viUninstallHandler(vi, VI_EVENT_TRIG, myHdlr, (ViAddr)10);

  /* close the sessions */
  viClose(vi);
  viClose(defaultRM);
}
```

Example: SRQ Callback

This program installs an event handler and enables an SRQ event. When the event occurs, the installed event handler is called. This example program is intended to show specific VISA functionality and does not include error trapping. Error trapping, however, is good programming practice and is recommended in your VISA applications. See "Trapping Errors" in this chapter.

This program is installed on your system in the SAMPLES subdirectory on Windows environments or in the examples subdirectory on HP-UX. See Appendix A - VISA Library Information for locations of example programs on your operating system.
/* srqhdlr.c
   This example program illustrates installing an event
   handler to be called when an SRQ interrupt occurs.
   Note that you may need to change the address. */

#include <visa.h>
#include <stdio.h>
#if defined (_WIN32)
    #include <windows.h> /* for Sleep() */
    #define YIELD Sleep( 10 )
#else defined (_BORLANDC_)
    #include <windows.h> /* for Yield() */
    #define YIELD Yield()
#elif defined (_WINDOWS)
    #include <io.h>      /* for _wyield */
    #define YIELD   _wyield()
#else
    #include <unistd.h>
    #define YIELD sleep (1)
#endif

int srqOccurred;

/* trigger event handler */
ViStatus _VI_FUNCH mySrqHdlr(ViSession vi, ViEventType
    eventType, ViEvent ctx, ViAddr userHdlr){

    ViUInt16 statusByte;

    /* make sure it is an SRQ event */
    if(eventType!=VI_EVENT_SERVICE_REQ){
        /* Stray event, so ignore */
        printf("\nStray event of type 0x%lx\n", eventType

        return VI_SUCCESS;
    }

    /* print the event information */
    printf("\nSRQ Event Occurred!\n");
    printf("...Original Device Session = %ld\n", vi);

    /* get the status byte */
    viReadSTB(vi, &statusByte);
    printf("...Status byte is 0x%x\n", statusByte);

    srqOccurred = 1;
    return VI_SUCCESS;
Programming with VISA
Using Events and Handlers

```c
void main(){
    ViSession defaultRM, vi;
    long count;

    /* open session to message based VXI device */
    viOpenDefaultRM(&defaultRM);
    viOpen(defaultRM, "GPIB-VXI0::24::INSTR", VI_NULL, VI_NULL, &vi);

    /* Enable command error events */
    viPrintf(vi, "*ESE 32\n");

    /* Enable event register interrupts */
    viPrintf(vi, "*SRE 32\n");

    /* install the handler and enable it */
    viInstallHandler(vi, VI_EVENT_SERVICE_REQ, mySrqHdlr, (ViAddr)10);
    viEnableEvent(vi, VI_EVENT_SERVICE_REQ, VI_HNDLR, VI_NULL);

    srqOccurred = 0;

    /* Send a bogus command to the message based device to
    cause an SRQ. Note: 'IDN' causes the error -- 'IDN?' is the correct syntax */
    viPrintf(vi, "IDN\n");

    /* Wait a while for the SRQ to be generated and for the
    handler to be called. Print something while we wait */
    printf("Waiting for an SRQ to be generated .\n");
    for (count = 0; (count < 10) && (srqOccurred == 0); count++) {
        long count2 = 0;
        printf(".\n");
        while ( (count2++ < 100) && (srqOccurred == 0) ){
            YIELD;
        }
    }
    printf("\n");

    /* disable and uninstall the handler */
    viDisableEvent(vi, VI_EVENT_SERVICE_REQ, VI_HNDLR);
    viUninstallHandler(vi, VI_EVENT_SERVICE_REQ, mySrqHdlr,
```
Programming with VISA
Using Events and Handlers

Using the Queuing Method

The queuing method is generally used when an immediate response from your application is not needed. To use the queuing method for receiving notification that an event has occurred, you must do the following:

- Enable one or several events with the `viEnableEvent` function.
- When ready to query, use the `viWaitOnEvent` function to check for queued events.

If the specified event has occurred, the event information is retrieved and the program returns immediately. If the specified event has not occurred, the program suspends execution until a specified event occurs or until the specified timeout period is reached.

Example: Using the Queuing Method

This example program shows one way you can use the queuing method.

```c
main();
ViSession vi;
ViEventType eventType;
ViEvent event;
.
viEnableEvent(vi, VI_EVENT_SERVICE_REQ, VI_QUEUE, VI_NULL);
.
viWaitOnEvent(vi, VI_EVENT_SERVICE_REQ, VI_TMO_INFINITY, &eventType, &event);
.
viClose(event);
viDisableEvent(vi, VI_EVENT_SERVICE_REQ, VI_QUEUE);
}
```
Enabling Events

Before an event can be delivered, it must be enabled using the `viEnableEvent` function:

```c
viEnableEvent(vi, eventType, mechanism, context);
```

where the parameters are defined as follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vi</code></td>
<td>The session the handler will be installed on.</td>
</tr>
<tr>
<td><code>eventType</code></td>
<td>The type of event to enable.</td>
</tr>
<tr>
<td><code>mechanism</code></td>
<td>The mechanism by which the event will be enabled. Specify <code>VI_QUEUE</code> to use the queuing method.</td>
</tr>
<tr>
<td><code>context</code></td>
<td>Not used in VISA 1.0. Use <code>VI_NULL</code>.</td>
</tr>
</tbody>
</table>

When you use `VI_QUEUE` in the `mechanism` parameter, you are specifying that the events will be put into a queue. Then, when a `viWaitOnEvent` function is invoked, the program execution will suspend until the enabled event occurs or the timeout period specified is reached. If the event has already occurred, the `viWaitOnEvent` function will return immediately.

Example: Enabling a Hardware Trigger Event

This example illustrates enabling a hardware trigger event.

```c
viEnableEvent(vi, VI_EVENT_TRIG, VI_QUEUE, VI_NULL);
```

The `VI_QUEUE` mechanism specifies that when an event occurs, it will go into a queue. If you specify `VI_ALL_ENABLE_EVENTS` in the `eventType` parameter, all events that have previously been enabled on the specified session will be enabled for the `mechanism` specified in this function call. Use the `viDisableEvent` function to stop servicing the event specified.

Wait on the Event

When using the `viWaitOnEvent` function, specify the session, the event type to wait for, and the timeout period to wait:

```c
viWaitOnEvent(vi, inEventType, timeout, outEventType, outContext);
```

The event must have previously been enabled with `VI_QUEUE` specified as the `mechanism` parameter.
Example: Wait on Event for SRQ

This example shows how to install a wait on event for service requests.

```c
    viEnableEvent(vi, VI_EVENT_SERVICE_REQ, VI_QUEUE, VI_NULL);
    viWaitOnEvent(vi, VI_EVENT_SERVICE_REQ,
                  VI_TMO_INFINITE,
                  &eventType, &event);

    viDisableEvent(vi, VI_EVENT_SERVICE_REQ, VI_QUEUE);
```

Every time a wait on event is invoked, an event context object is created. Specifying `VI_TMO_INFINITE` in the `timeout` parameter indicates that the program execution will suspend indefinitely until the event occurs. To clear the event queue for a specified event type, use the `viDiscardEvents` function.

Example: Trigger Event Queuing

This program enables the trigger event in a queuing mode. When the `viWaitOnEvent` function is called, the program will suspend operation until the trigger line is fired or the timeout period is reached. Since the trigger lines were already fired and the events were put into a queue, the function will return and print the trigger line that fired.

This program is intended to show specific VISA functionality and does not include error trapping. Error trapping, however, is good programming practice and is recommended in your VISA applications. See “Trapping Errors” in this chapter.

This example program is installed on your system in the `SAMPLES` subdirectory on Windows environments or in the `examples` subdirectory on HP-UX. See `Appendix A - VISA Library Information` for locations of example programs on your operating system.

```c
/* evntqueu.c 
   This example program illustrates enabling an event queue using viWaitOnEvent. Note that you must change the device address. */

#include <visa.h>
#include <stdio.h>

void main(){
    ViSession defaultRM,vi;
    ViEventType eventType;
    ViEvent eventVi;
    ViStatus err;
```
ViInt16 trigId;

/* open session to VXI device */
viOpenDefaultRM(&defaultRM);
viOpen(defaultRM, "VXI0::24::INSTR", VI_NULL, VI_NULL,
&vi);

/* select trigger line TTL0 */
viSetAttribute(vi, VI_ATTR_TRIG_ID, VI_TRIG_TTL0);

/* enable the event */
viEnableEvent(vi, VI_EVENT_TRIG, VI_QUEUE, VI_NULL);

/* fire trigger line, twice */
viAssertTrigger(vi, VI_TRIG_PROT_SYNC);
viAssertTrigger(vi, VI_TRIG_PROT_SYNC);

/* Wait for the event to occur */
err=viWaitOnEvent(vi, VI_EVENT_TRIG, 10000, &eventType,
&eventVi);
if(err==VI_ERROR_TMO){
    printf("Timeout Occurred! Event not received.\n");
    return;
}

/* print the event information */
printf("Trigger Event Occurred!\n");
printf("...Original Device Session = %ld\n", vi);

/* get trigger that fired */
viGetAttribute(eventVi, VI_ATTR_RECV_TRIG_ID, &trigId);
printf("Trigger that fired: ");
switch(trigId){
    case VI_TRIG_TTL0:
        printf("TTL0");
        break;
    default:
        printf("<other 0x%x>", trigId);
        break;
}
printf("\n");

/* close the context before continuing */
viClose(eventVi);
/* get second event */
err=viWaitOnEvent(vi, VI_EVENT_TRIG, 10000, &eventType,
&eventVi);
if(err==VI_ERROR_TMO){
    printf("Timeout Occurred! Event not received.\n");
    return;
}
printf("Got second event\n");

/* close the context before continuing */
viClose(eventVi);

/* disable event */
viDisableEvent(vi, VI_EVENT_TRIG, VI_QUEUE);

/* close the sessions */
viClose(vi);
viClose(defaultRM);
Programming with VISA

Trapping Errors

This section gives guidelines to trap errors, including:

- Trapping Errors
- Exception Events

Trapping Errors

The example programs in this guide show specific VISA functionality and do not include error trapping. Error trapping, however, is good programming practice and is recommended in all your VISA application programs. To trap VISA errors you must check for `VI_SUCCESS` after each VISA function call.

If you want to ignore WARNINGS, you can test to see if `err` is less than (`<`) `VI_SUCCESS`. Since WARNINGS are greater than `VI_SUCCESS` and ERRORS are less than `VI_SUCCESS`, `err_handler` would only be called when the function returns an ERROR. For example:

```c
if(err < VI_SUCCESS) err_handler(vi, err);
```

Example: Check for `VI_SUCCESS`

This example illustrates checking for `VI_SUCCESS`. If `VI_SUCCESS` is not returned, an error handler (written by the programmer) is called. This must be done with each VISA function call.

```c
ViStatus err;
.
.err=viPrintf(vi, "*RST\n");
if (err < VI_SUCCESS) err_handler(vi, err);
.
```

Example: Printing Error Code

The following error handler prints a user-readable string describing the error code passed to the function:

```c
void err_handler(ViSession vi, ViStatus err){
    char err_msg[1024]=\0;
    viStatusDesc (vi, err, err_msg);
    printf ("ERROR = %s\n", err_msg);
    return;
}
```
Example: Checking Instrument Errors

When programming instruments, it is good practice to check the instrument to ensure there are no instrument errors after each instrument function. This example uses a SCPI command to check a specific instrument for errors.

```c
void system_err()
{
    ViStatus err;
    char buf[1024]=\0; int err_no;

    err=viPrintf(vi, "SYSTEM:ERR?\\n");
    if (err < VI_SUCCESS) err_handler (vi, err);

    err=viScanf (vi, "%d%t", &err_no, &buf);
    if (err < VI_SUCCESS) err_handler (vi, err);

    while (err_no >0){
        printf ("Error Found: %d,%s\\n", err_no, buf);
        err=viScanf (vi, "%d%t", &err_no, &buf);
    }
    err=viFlush(vi, VI_READ_BUF);
    if (err < VI_SUCCESS) err_handler (vi, err);

    err=viFlush(vi, VI_WRITE_BUF);
    if (err < VI_SUCCESS) err_handler (vi, err);
}
```

Exception Events

An alternative to trapping VISA errors by checking the return status after each VISA call is to use the VISA exception event. On sessions where an exception event handler is installed and VI_EVENT_EXCEPTION is enabled, the exception event handler is called whenever an error occurs while executing an operation.

Exception Handling Model

The exception-handling model follows the event-handling model for callbacks and it uses the same operations as those used for general event handling. For example, an application calls `viInstallHandler` and `viEnableEvent` to enable exception events. The exception event is like any other event in VISA, except that the queueing and suspended handler mechanisms are not allowed.
Programming with VISA

Trapping Errors

When an error occurs for a session operation, the exception handler is executed synchronously. That is, the operation that caused the exception blocks until the exception handler completes its execution. The exception handler is executed in the context of the same thread that caused the exception event.

When invoked, the exception handler can check the error condition and instruct the exception operation to take a specific action. It can instruct the exception operation to continue normally (by returning `VI_SUCCESS`) or to not invoke any additional handlers in the case of handler nesting (by returning `VI_SUCCESS_NCHAIN`).

As noted, an exception operation blocks until the exception handler execution is completed. However, an exception handler sometimes may prefer to terminate the program prematurely without returning the control to the operation generating the exception. VISA does not preclude an application from using a platform-specific or language-specific exception handling mechanism from within the VISA exception handler.

For example, the C++ try/catch block can be used in an application in conjunction with the C++ throw mechanism from within the VISA exception handler. When using the C++ try/catch/throw or other exception-handling mechanisms, the control will not return to the VISA system. This has several important repercussions:

1. If multiple handlers were installed on the exception event, the handlers that were not invoked prior to the current handler will not be invoked for the current exception.

2. The exception context will not be deleted by the VISA system when a C++ exception is used. In this case, the application should delete the exception context as soon as the application has no more use for the context, before terminating the session. An application should use the `viClose` operation to delete the exception context.

3. Code in any operation (after calling an exception handler) may not be called if the handler does not return. For example, local allocations must be freed before invoking the exception handler, rather than after it.

One situation in which an exception event will not be generated is in the case of asynchronous operations. If the error is detected after the operation is posted (i.e., once the asynchronous portion has begun), the status is returned normally via the I/O completion event.
Programming with VISA
Trapping Errors

However, if an error occurs before the asynchronous portion begins (i.e., the
error is returned from the asynchronous operation itself), then the exception
event will still be raised. This deviation is due to the fact that asynchronous
operations already raise an event when they complete, and this I/O
completion event may occur in the context of a separate thread previously
unknown to the application. In summary, a single application event handler
can easily handle error conditions arising from both exception events and
failed asynchronous operations.

Using the
VI_EVENT_EXCEPTION Event

You can use the **VI_EVENT_EXCEPTION** event as notification that an error
condition has occurred during an operation invocation. The following table
describes the **VI_EVENT_EXCEPTION** event attributes.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Access Privilege</th>
<th>Data Type</th>
<th>Range</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>VI_ATTR_EVENT_TYPE</td>
<td>RO Global</td>
<td>ViEventType</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>VI_ATTR_STATUS</td>
<td>RO Global</td>
<td>ViStatus</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>VI_ATTR_OPER_NAME</td>
<td>RO Global</td>
<td>ViString</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Example: Exception Events

/* This is an example of how to use exception events
to trap VISA errors. An exception event handler must
be installed and exception events enabled on all
sessions where the exception handler is used.*/

```c
#include <stdio.h>
#include <visa.h>

ViStatus __stdcall myExceptionHandler (ViSession vi,
    ViEventType eventType,
    ViEvent context,
    ViAddr usrHandle
) {
    ViStatus exceptionErrNbr;
    char nameBuffer[256];
    ViString functionName = nameBuffer;
    char errStrBuffer[256];
    /* Get the error value from the exception context */
    viGetAttribute( context, VI_ATTR_STATUS,
                    &exceptionErrNbr );
    /* Get the function name from the exception context */
    viGetAttribute( context, VI_ATTR_OPER_NAME,
                    functionName );
```
errStrBuffer[0] = 0;
    viStatusDesc( vi, exceptionErrNbr, errStrBuffer );
    printf("ERROR: Exception Handler reports\n" "(\%s)\n",
            "VISA function '%s' failed with error 0x%lx\n",
            "functionName, exceptionErrNbr, errStrBuffer ");
    return VI_SUCCESS;
}

void main()
{
    ViStatus  status;
    ViSession drm;
    ViSession vi;
    ViAddr    myUserHandle = 0;

    status = viOpenDefaultRM( &drm );
    if ( status < VI_SUCCESS ) {
        printf( "ERROR: viOpenDefaultRM failed with error = 0x%lx\n", status );
        return;
    }
    /* Install the exception handler and enable events for it */
    status = viInstallHandler(drm, VI_EVENT_EXCEPTION,
                                myExceptionHandler, myUserHandle);
    if ( status < VI_SUCCESS ) {
        printf( "ERROR: viInstallHandler failed with error 0x%lx\n", status );
    }
    status = viEnableEvent(drm, VI_EVENT_EXCEPTION, VI_HNDLR,
                            VI_NULL);
    if ( status < VI_SUCCESS ) {
        printf( "ERROR: viEnableEvent failed with error 0x%lx\n", status );
    }
    /* Generate an error to demonstrate that the handler will be called */
    status = viOpen( drm, "badVisaName", NULL, NULL, &vi );
    if ( status < VI_SUCCESS ) {
        printf("ERROR: viOpen failed with error 0x%lx\n"
            "Exception Handler should have been called\n"
            "before this message was printed.\n",status );
    }
}
Using Locks

In VISA, applications can open multiple sessions to a VISA resource simultaneously. Applications can, therefore, access a VISA resource concurrently through different sessions. However, in certain cases, applications accessing a VISA resource may want to restrict other applications from accessing that resource.

Lock Functions

For example, when an application needs to perform successive write operations on a resource, the application may require that, during the sequence of writes, no other operation can be invoked through any other session to that resource. For such circumstances, VISA defines a locking mechanism that restricts access to resources.

The VISA locking mechanism enforces arbitration of accesses to VISA resources on a per-session basis. If a session locks a resource, operations invoked on the resource through other sessions either are serviced or are returned with an error, depending on the operation and the type of lock used.

If a VISA resource is not locked by any of its sessions, all sessions have full privilege to invoke any operation and update any global attributes. Sessions are not required to have locks to invoke operations or update global attributes. However, if some other session has already locked the resource, attempts to update global attributes or invoke certain operations will fail.

See descriptions of the individual VISA functions in Chapter 7 - VISA Language Reference to determine which would fail when a resource is locked.

viLock/viUnlock Functions

The VISA viLock function is used to acquire a lock on a resource.

\[ \text{viLock}(\text{vi}, \text{lockType}, \text{timeout}, \text{requestedKey}, \text{accessKey}); \]

The VI_ATTR_RSRC_LOCK_STATE attribute specifies the current locking state of the resource on the given session, which can be either VI_NO_LOCK, VI_EXCLUSIVE_LOCK, or VI_SHARED_LOCK.

The VISA viUnlock function is then used to release the lock on a resource. If a resource is locked and the current session does not have the lock, the error VI_ERROR_RSRC_LOCKED is returned.
VISA Lock Types

VISA defines two different types of locks: Exclusive Lock and Shared Lock.

- **Exclusive Lock** - A session can lock a VISA resource using the lock type `VI_EXCLUSIVE_LOCK` to get exclusive access privileges to the resource. This exclusive lock type excludes access to the resource from all other sessions.

  If a session has an exclusive lock, other sessions cannot modify global attributes or invoke operations on the resource. However, the other sessions can still get attributes.

- **Shared Lock** - A session can share a lock on a VISA resource with other sessions by using the lock type `VI_SHARED_LOCK`. Shared locks in VISA are similar to exclusive locks in terms of access privileges, but can still be shared between multiple sessions.

  If a session has a shared lock, other sessions that share the lock can also modify global attributes and invoke operations on the resource (of course, unless some other session has a previous exclusive lock on that resource). A session that does not share the lock will lack these capabilities.

Locking a resource restricts access from other sessions and, in the case where an exclusive lock is acquired, ensures that operations do not fail because other sessions have acquired a lock on that resource. Thus, locking a resource prevents other, subsequent sessions from acquiring an exclusive lock on that resource. Yet, when multiple sessions have acquired a shared lock, VISA allows one of the sessions to acquire an exclusive lock along with the shared lock it is holding.

Also, VISA supports nested locking. That is, a session can lock the same VISA resource multiple times (for the same lock type) via multiple invocations of the `viLock` function. In such a case, unlocking the resource requires an equal number of invocations of the `viUnlock` function. Nested locking is also explained in detail later in this section.

Some VISA operations may be permitted even when there is an exclusive lock on a resource, or some global attributes may not be read when there is any kind of lock on the resource. These exceptions, when applicable, are mentioned in the descriptions of the individual VISA functions and attributes.

See Chapter 7 - VISA Language Reference for descriptions of individual functions to determine which are applicable for locking and which are not restricted by locking.
This example shows a session gaining an exclusive lock to perform the `viPrintf` and `viScanf` VISA operations on a GPIB device. It then releases the lock via the `viUnlock` function.

/* lockexcl.c
   This example program queries a GPIB device for an identification string and prints the results. Note that you may need to change the address. */

#include <visa.h>
#include <stdio.h>

void main () {

ViSession defaultRM, vi;
char buf [256] = {0};

/* Open session to GPIB device at address 22 */
viOpenDefaultRM (&defaultRM);
viOpen (defaultRM, "GPIB0::22::INSTR", VI_NULL, VI_NULL, &vi);

/* Initialize device */
viPrintf (vi, "*RST
");

/* Make sure no other process or thread does anything to this resource between viPrintf and viScanf calls */
viLock (vi, VI_EXCLUSIVE_LOCK, 2000, VI_NULL, VI_NULL);

/* Send an *IDN? string to the device */
viPrintf (vi, "*IDN?
");

/* Read results */
viScanf (vi, "%t", &buf);
/* Unlock this session so other processes and threads can use it */
viUnlock (vi);

/* Print results */
printf("Instrument identification string: %s\n", buf);
/* Close session */
viClose (vi);
viClose (defaultRM);"
This example shows a session gaining a shared lock with the `accessKey` called `lockkey`. Other sessions can now use this `accessKey` in the `requestedKey` parameter of the `viLock` function to share access on the locked resource. This example then shows the original session acquiring an exclusive lock while maintaining its shared lock.

When the session holding the exclusive lock unlocks the resource via the `viUnlock` function, all the sessions sharing the lock again have all the access privileges associated with the shared lock.

```
/* lockshr.c */
This example program queries a GPIB device for an identification string and prints the results. Note that you must change the address. */

#include <visa.h>
#include <stdio.h>

void main () {
    ViSession defaultRM, vi;
    char buf [256] = {0};
    char lockkey [256] = {0};

    /* Open session to GPIB device at address 22 */
    viOpenDefaultRM (&defaultRM);
    viOpen (defaultRM, "GPIB0::22::INSTR",
VI_NULL,VI_NULL,&vi);

    /* acquire a shared lock so only this process and processes that we know about can access this resource */
    viLock (vi, VI_SHARED_LOCK, 2000, VI_NULL, lockkey);

    /* at this time, we can make 'lockkey' available to other processes that we know about. This can be done with shared memory or other inter-process communication methods. These other processes can then call "viLock(vi,VI_SHARED_LOCK, 2000, lockkey, lockkey)" and they will also have access to this resource. */

    /* Initialize device */
    viPrintf (vi, "*RST\n");
```
/* Make sure no other process or thread does anything to this resource between the viPrintf() and the viScanf() calls. Note: this also locks out the processes with which we shared our 'shared lock' key. */

viLock (vi, VI_EXCLUSIVE_LOCK, 2000, VI_NULL, VI_NULL);
/* Send an *IDN? string to the device */
viPrintf (vi, "*IDN?\n");

/* Read results */
viScanf (vi, "%t", &buf);

/* unlock this session so other processes and threads can use it */
viUnlock (vi);

/* Print results */
printf ("Instrument identification string: %s\n", buf);

/* release the shared lock also*/
viUnlock (vi);

/* Close session */
viClose (vi);
viClose (defaultRM);
}
Programming with VISA
Using Locks

Notes:
Programming via GPIB and VXI
VISA supports three interfaces you can use to access GPIB and VXI instruments: GPIB, VXI, and GPIB-VXI. This chapter provides information to program GPIB and VXI devices via the GPIB, VXI or GPIB-VXI interfaces, including:

- GPIB and VXI Interfaces Overview
- Using High-Level Memory Functions
- Using Low-Level Memory Functions
- Using High/Low-Level Memory I/O Methods
- Using the Memory Access Resource
- Using VXI-Specific Attributes

See Chapter 4 - Programming with VISA for general information on VISA programming for the GPIB, VXI, and GPIB-VXI interfaces. See Chapter 7 - VISA Language Reference for information on the specific VISA functions.
GPIB and VXI Interfaces Overview

This section provides an overview of the GPIB, GPIB-VXI, and VXI interfaces, including:

- General Interface Information
- GPIB Interfaces Overview
- VXI Interfaces Overview
- GPIB-VXI Interfaces Overview

General Interface Information

VISA supports three interfaces you can use to access instruments or devices: GPIB, VXI, and GPIB-VXI. The GPIB interface can be used to access VXI instruments via a Command Module. In addition, the VXI backplane can be directly accessed with the VXI or GPIB-VXI interfaces.

What is an IO Interface?

An IO interface can be defined as both a hardware interface and as a software interface. The IO Config utility is used to associate a unique interface name with a hardware interface. The IO Libraries use an Interface Name or Logical Unit Number to identify an interface. This information is passed in the parameter string of the viOpen function call in a VISA program.

IO Config assigns an Interface Name and Logical Unit Number to the interface hardware, and other necessary configuration values for an interface when the interface is configured. See the Agilent IO Libraries Installation and Configuration Guide for Windows for IO Config information.

VXI Device Types

When using GPIB-VXI or VXI interfaces to directly access the VXI backplane (in the VXI mainframe), you must know whether you are programming a message-based or a register-based VXI device (instrument).

A message-based VXI device has its own processor that allows it to interpret high-level commands such as Standard Commands for Programmable Instruments (SCPI). When using VISA, you can place the SCPI command within your VISA output function call. Then, the message-based device interprets the SCPI command. In this case you can use the VISA formatted I/O or non-formatted I/O functions and program the message-based device as you would a GPIB device.
Programming via GPIB and VXI

GPIB and VXI Interfaces Overview

However, if the message-based device has shared memory, you can access the device’s shared memory by doing register peeks and pokes. VISA provides two different methods you can use to program directly to the registers: high-level memory functions or low-level memory functions.

A register-based VXI device typically does not have a processor to interpret high-level commands. Therefore, the device must be programmed with register peeks and pokes directly to the device’s registers. VISA provides two different methods you can use to program register-based devices: high-level memory functions or low-level memory functions.

GPIB Interfaces Overview

As shown in the following figure, a typical GPIB interface consists of a Windows PC with one or more GPIB cards (PCI and/or ISA) cards installed in the PC and one or more GPIB instruments connected to the GPIB cards via GPIB cable. I/O communication between the PC and the instruments is via the GPIB cards and the GPIB cable. This figure shows GPIB instruments at addresses 3 and 5.
Example: GPIB (82350) Interface
The GPIB interface system in the following figure consists of a Windows PC with two 82350 GPIB cards connected to three GPIB instruments via GPIB cables. For this system, the IO Config utility has been used to assign GPIB card #1 a VISA name of “GPIB0” and to assign GPIB card #2 a VISA name of “GPIB1”. VISA addressing is as shown in the figure.

82350 GPIB Card #1
Windows PC
GPIB Cable
GPIB Instruments

VISA Name
“GPIB0”
“GPIB1”

VISA Addressing
viOpen (... "GPIB0::5::INSTR"...) Open IO path to GPIB instrument at address 5 using 82350 Card #1
viOpen (... "GPIB0::3::INSTR"...) Open IO path to GPIB instrument at address 3 using 82350 Card #1
viOpen (... "GPIB1::3::INSTR"...) Open IO path to GPIB instrument at address 3 using 82350 Card #2
VXI Interfaces Overview

As shown in the following figure, a typical VXI (E8491) interface consists of an E8491 PC Card in a Windows PC that is connected to an E8491B IEEE-1394 Module in a VXI mainframe via an IEEE-1394 to VXI cable. The VXI mainframe also includes one or more VXI instruments.

Example: VXI (E8491B) Interfaces

The VXI interface system in the following figure consists of a Windows PC with an E8491 PC card that connects to an E8491B IEEE-1394 to VXI Module in a VXI Mainframe. For this system, the three VXI instruments shown have logical addresses 8, 16, and 24. The IO Config utility has been used to assign the E8491 PC card a VISA name of “VXI0”. VISA addressing is as shown in the figure.

For information on the E8491B module, see the Agilent E8491B User’s Guide. For information on VXI instruments, see the applicable VXI Instrument User’s Guide.
GPIB and VXI Interfaces Overview

Interface VISA Name | Windows PC | IEEE-1394 to VXI | VXI Mainframe
---|---|---|---
VISA Name | "VXI0" | | |

VISA Addressing

viOpen (... "VXI0::24::INSTR"...)  Open IO path to VXI instrument at logical address 24 using E8491 PC Card and E8491 IEEE-1394 to VXI Module
GPIB-VXI Interfaces Overview

As shown in the following figure, a typical GPIB-VXI interface consists of a GPIB card (82350 or equivalent) in a Windows PC that is connected via a GPIB cable to an E1406A Command Module. The E1406A sends commands to the VXI instruments in a VXI mainframe. There is no direct access to the VXI backplane from the PC.

NOTE

For a GPIB-VXI interface, VISA uses a DLL supplied by the Command Module vendor to translate the VISA VXI calls to Command Module commands that are vendor-specific. The DLL required for Agilent/Hewlett-Packard Command Modules is installed by the Agilent IO Libraries Installer. This DLL is installed by default when Agilent VISA is installed.
Example: GPIB-VXI (E1406A) Interface

The GPIB-VXI interface system in the following figure consists of a Windows PC with an 82350 GPIB card that connects to an E1406A Command Module in a VXI Mainframe. The VXI mainframe includes one or more VXI instruments.

When the IO Libraries were installed, a GPIB-VXI driver with GPIB address 9 was also installed and the E1406A was configured for primary address 9 and logical address (LA) 0. The three VXI instruments shown have logical addresses 8, 16, and 24.

The IO Config utility has been used to assign the GPIB-VXI driver a VISA Name of “GPIB-VXI0” and to assign the 82350 GPIB card a VISA name of “GPIB0”. VISA addressing is as shown in the figure.

For information on the E1406A Command Module, see the Agilent E1406A Command Module User’s Guide. For information on VXI instruments, see the applicable VXI instrument User’s Guide.

### GPIB-VXI Interface (E1406A Command Module)

<table>
<thead>
<tr>
<th>Interface VISA Name</th>
<th>Windows PC</th>
<th>VXI Mainframe</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>GPIB-VXI0</em></td>
<td>82350 GPIB Card</td>
<td>Primary Address 9</td>
</tr>
<tr>
<td><em>GPIB0</em></td>
<td></td>
<td>E 14 0 6 A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LA 0 LA 8 LA 24 LA 16</td>
</tr>
<tr>
<td>VISA Addressing</td>
<td></td>
<td>viOpen (... &quot;GPIB-VXI0::24::INSTR&quot;...) Open IO path to VXI instrument at logical address 24 using 82350 GPIB Card and E1406A VXI Command Module at GPIB primary address 9</td>
</tr>
</tbody>
</table>
Using High-Level Memory Functions

High-level memory functions allow you to access memory on the interface through simple function calls. There is no need to map memory to a window. Instead, when high-level memory functions are used, memory mapping and direct register access are automatically done.

The tradeoff, however, is speed. High-level memory functions are easier to use. However, since these functions encompass mapping of memory space and direct register access, the associated overhead slows program execution time. If speed is required, use the low-level memory functions discussed in “Using Low-Level Memory Functions”.

Programming the Registers

High-level memory functions include the viIn and viOut functions for transferring 8-, 16-, or 32-bit values, as well as the viMoveIn and viMoveOut functions for transferring 8-, 16-, or 32-bit blocks of data into or out of local memory. You can therefore program using 8-, 16-, or 32-bit transfers.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>viIn8 (vi, space, offset, val8) ;</td>
<td>Reads 8 bits of data from the specified offset.</td>
</tr>
<tr>
<td>viIn16 (vi, space, offset, val16) ;</td>
<td>Reads 16 bits of data from the specified offset.</td>
</tr>
<tr>
<td>viIn32 (vi, space, offset, val32) ;</td>
<td>Reads 32 bits of data from the specified offset.</td>
</tr>
<tr>
<td>viOut8 (vi, space, offset, val8) ;</td>
<td>Writes 8 bits of data to the specified offset.</td>
</tr>
<tr>
<td>viOut16 (vi, space, offset, val16) ;</td>
<td>Writes 16 bits of data to the specified offset.</td>
</tr>
<tr>
<td>viOut32 (vi, space, offset, val32) ;</td>
<td>Writes 32 bits of data to the specified offset.</td>
</tr>
<tr>
<td>viMoveIn8 (vi, space, offset, length, buf8) ;</td>
<td>Moves an 8-bit block of data from the specified offset to local memory.</td>
</tr>
</tbody>
</table>
When using the `viIn` and `viOut` high-level memory functions to program to the device registers, all you need to specify is the session identifier, address space, and the offset of the register. Memory mapping is done for you. For example, in this function:

```c
viIn32(vi, space, offset, val32);
```

`vi` is the session identifier and `offset` is used to indicate the offset of the memory to be mapped. `offset` is relative to the location of this device's memory in the given address space. The `space` parameter determines which memory location to map the space. Valid `space` values are:

- **VI_A16_SPACE** - Maps in VXI/MXI A16 address space
- **VI_A24_SPACE** - Maps in VXI/MXI A24 address space
- **VI_A32_SPACE** - Maps in VXI/MXI A32 address space

The `val32` parameter is a pointer to where the data read will be stored. If, instead, you write to the registers via the `viOut32` function, the `val32` parameter is a pointer to the data to write to the specified registers. If the device specified by `vi` does not have memory in the specified address space, an error is returned. The following example uses `viIn16`:

```c
ViSession defaultRM, vi;
ViUInt16 value;

viOpenDefaultRM(&&defaultRM);
viOpen(defaultRM, "VXI::24", VI_NULL, VI_NULL, &vi);
viIn16(vi, VI_A16_SPACE, 0x100, &value);
```
Using `viMoveIn` and `viMoveOut` You can also use the `viMoveIn` and `viMoveOut` high-level memory functions to move blocks of data to or from local memory. Specifically, the `viMoveIn` function moves an 8-, 16-, or 32-bit block of data from the specified offset to local memory, and the `viMoveOut` functions moves an 8-, 16-, or 32-bit block of data from local memory to the specified offset. Again, the memory mapping is done for you.

For example, in this function:

```c
viMoveIn32(vi, space, offset, length, buf32);
```

`vi` is the session identifier and `offset` is used to indicate the offset of the memory to be mapped. `offset` is relative to the location of this device’s memory in the given address space. The `space` parameter determines which memory location to map the space and the `length` parameter specifies the number of elements to transfer (8-, 16-, or 32-bits).

The `buf32` parameter is a pointer to where the data read will be stored. If, instead, you write to the registers via the `viMoveOut32` function, the `buf32` parameter is a pointer to the data to write to the specified registers.

**High-Level Memory Functions Examples**

Two example programs follow that use the high-level memory functions to read the ID and Device Type registers of a device at the VXI logical address 24. The contents of the registers are then printed out.

The first program uses the VXI interface and the second program accesses the backplane with the GPIB-VXI interface. These two programs are identical except for the string passed to `viOpen`.

**Example: Using the VXI Interface (High-Level) Memory Functions**

This program uses high-level memory functions and the VXI interface to read the ID and Device Type registers of a device at VXI0::24.

```c
/* vxihl.c
   This example program uses the high-level memory functions to read the id and device type registers of the device at VXI0::24. Change this address if necessary. The register contents are then displayed.*/

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

void main () {
```
Example: Using the GPIB-VXI Interface (High-Level) Memory Functions

This program uses high-level memory functions and the GPIB-VXI interface to read the ID and Device Type registers of a device at GPIB-VXI0::24.

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

void main ()
{
    ViSession defaultRM, dmm;
    unsigned short id_reg, devtype_reg;

    /* Open session to VXI device at address 24 */
    viOpenDefaultRM(&defaultRM);
    viOpen(defaultRM, "VXI0::24::INSTR", VI_NULL, VI_NULL, &dmm);

    /* Read instrument id register contents */
    viIn16(dmm, VI_A16_SPACE, 0x00, &id_reg);

    /* Read device type register contents */
    viIn16(dmm, VI_A16_SPACE, 0x02, &devtype_reg);

    /* Print results */
    printf("ID Register = 0x%4X\n", id_reg);
    printf("Device Type Register = 0x%4X\n", devtype_reg);

    /* Close sessions */
    viClose(dmm);
    viClose(defaultRM);
}
```
Programming via GPIB and VXI
Using High-Level Memory Functions

unsigned short id_reg, devtype_reg;

/* Open session to VXI device at address 24 */
viOpenDefaultRM(&defaultRM);
viOpen(defaultRM, "GPIB-VXI0::24::INSTR",
       VI_NULL, VI_NULL, &dmm);

/* Read instrument id register contents */
viIn16(dmm, VI_A16_SPACE, 0x00, &id_reg);

/* Read device type register contents */
viIn16(dmm, VI_A16_SPACE, 0x02, &devtype_reg);

/* Print results */
printf("ID Register = 0x%4X\n", id_reg);
printf("Device Type Register = 0x%4X\n", devtype_reg);

/* Close sessions */
viClose(dmm);
viClose(defaultRM);
Using Low-Level Memory Functions

Low-level memory functions allow direct access to memory on the interface just as do high-level memory functions. However, with low-level memory function calls, you must map a range of addresses and directly access the registers with low-level memory functions, such as `viPeek32` and `viPoke32`.

There is more programming effort required when using low-level memory functions. However, the program execution speed can increase. Additionally, to increase program execution speed, the low-level memory functions do not return error codes.

Programming the Registers

When using the low-level memory functions for direct register access, you must first map a range of addresses using the `viMapAddress` function. Next, you can send a series of peeks and pokes using the `viPeek` and `viPoke` low-level memory functions. Then, you must free the address window using the `viUnmapAddress` function. A process you could use is:

1. Map memory space using `viMapAddress`.
2. Read and write to the register's contents using `viPeek32` and `viPoke32`.
3. Unmap the memory space using `viUnmapAddress`.

You can program the registers using low-level functions for 8-, 16-, or 32-bit transfers. This table summarizes the low-level memory functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>viMapAddress</code></td>
<td>Maps the specified memory space.</td>
</tr>
<tr>
<td><code>viPeek8</code></td>
<td>Reads 8 bits of data from address specified.</td>
</tr>
<tr>
<td><code>viPeek16</code></td>
<td>Reads 16 bits of data from address specified.</td>
</tr>
</tbody>
</table>
Mapping Memory Space

When using VISA to access the device's registers, you must map memory space into your process space. For a given session, you can have only one map at a time. To map space into your process, use the VISA `viMapAddress` function:

```c
viMapAddress(vi, mapSpace, mapBase, mapSize, access, suggested, &address);
```

This function maps space for the device specified by the `vi` session. `mapBase`, `mapSize`, and `suggested` are used to indicate the offset of the memory to be mapped, amount of memory to map, and a suggested starting location, respectively. `mapSpace` determines which memory location to map the space. The following are valid `mapSpace` choices:

- `VI_A16_SPACE` - Maps in VXI/MXI A16 address space
- `VI_A24_SPACE` - Maps in VXI/MXI A24 address space
- `VI_A32_SPACE` - Maps in VXI/MXI A32 address space

A pointer to the address space where the memory was mapped is returned in the `address` parameter. If the device specified by `vi` does not have memory in the specified address space, an error is returned. Some example `viMapAddress` function calls are:

```c
/* Maps to A32 address space */
viMapAddress(vi, VI_A32_SPACE, 0x000, 0x100, VI_FALSE, VI_NULL, &address);
/* Maps to A24 address space */
viMapAddress(vi, VI_A24_SPACE, 0x00, 0x80, VI_FALSE, VI_NULL, &address);
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>viPeek32(vi, addr, val32)</code></td>
<td>Reads 32 bits of data from address specified.</td>
</tr>
<tr>
<td><code>viPoke8(vi, addr, val8)</code></td>
<td>Writes 8 bits of data to address specified.</td>
</tr>
<tr>
<td><code>viPoke16(vi, addr, val16)</code></td>
<td>Writes 16 bits of data to address specified.</td>
</tr>
<tr>
<td><code>viPoke32(vi, addr, val32)</code></td>
<td>Writes 32 bits of data to address specified.</td>
</tr>
<tr>
<td><code>viUnmapAddress(vi)</code></td>
<td>Unmaps memory space previously mapped.</td>
</tr>
</tbody>
</table>
Chapter 5

Programming via GPIB and VXI

Using Low-Level Memory Functions

Reading and Writing to Device Registers

When you have mapped the memory space, use the VISA low-level memory functions to access the device’s registers. First, determine which device register you need to access. Then, you need to know the register’s offset. See the applicable instrument User manual for a description of the registers and register locations. You can then use this information and the VISA low-level functions to access the device registers.

Example: Using viPeek16

An example using viPeek16 follows.

```c
ViSession defaultRM, vi;
ViUInt16 value;
ViAddr address;
ViUInt16 value;

viOpenDefaultRM(&&defaultRM);
viOpen(defaultRM, "VXI::24::INSTR", VI_NULL, VI_NULL, &vi);
viMapAddress(vi, VI_A16_SPACE, 0x00, 0x04, VI_FALSE, VI_NULL, &address);
viPeek16(vi, addr, &value)
```

Unmapping Memory Space

Make sure you use the viUnmapAddress function to unmap the memory space when it is no longer needed. Unmapping memory space makes the window available for the system to reallocate.

Low-Level Memory Functions Examples

Two example programs follow that use the low-level memory functions to read the ID and Device Type registers of the device at VXI logical address 24. The contents of the registers are then printed out. The first program uses the VXI interface and the second program uses the GPIB-VXI interface to access the VXI backplane. These two programs are identical except for the string passed to viope.

Example: Using the VXI Interface (Low-Level) Memory Functions

This program uses low-level memory functions and the VXI interface to read the ID and Device Type registers of a device at VXI0::24.

```c
/*vxill.c

This example program uses the low-level memory functions to read the id and device type registers of the device at VXI0::24. Change this address if necessary. The register contents are then displayed.*/
```
#include <visa.h>
#include <stdlib.h>
#include <stdio.h>

void main () {
    ViSession defaultRM, dmm;
    ViAddr address;
    unsigned short id_reg, devtype_reg;

    /* Open session to VXI device at address 24 */
    viOpenDefaultRM(&defaultRM);
    viOpen(defaultRM, "VXI0::24::INSTR", VI_NULL,
            VI_NULL, &dmm);

    /* Map into memory space */
    viMapAddress(dmm, VI_A16_SPACE, 0x00, 0x10,
                  VI_FALSE,VI_NULL, &address);

    /* Read instrument id register contents */
    viPeek16(dmm, address, &id_reg);

    /* Read device type register contents */
    /* ViAddr is defined as a void so we must cast
       it to something else to do pointer arithmetic */
    viPeek16(dmm, (ViAddr)((ViUInt16 *)address + 0x01),
            &devtype_reg);

    /* Unmap memory space */
    viUnmapAddress(dmm);

    /* Print results */
    printf ("ID Register = 0x%4X\n", id_reg);
    printf ("Device Type Register = 0x%4X\n", devtype_reg);

    /* Close sessions */
    viClose(dmm);
    viClose(defaultRM);
}
This program uses low-level memory functions and the GPIB-VXI interface to read the ID and Device Type registers of a device at GPIB-VXI0::24.

/*gpibvxil.c
This example program uses the low-level memory functions to read the id and device type registers of the device at GPIB-VXI0::24. Change this address if necessary. Register contents are then displayed.*/

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

void main () {

  ViSession defaultRM, dmm;
  ViAddr address;
  unsigned short id_reg, devtype_reg;

  /* Open session to VXI device at address 24 */
  viOpenDefaultRM(&defaultRM);
  viOpen(defaultRM, "GPIB-VXI0::24::INSTR", VI_NULL, VI_NULL, &dmm);

  /* Map into memory space */
  viMapAddress(dmm, VI_A16_SPACE, 0x00, 0x10, VI_FALSE, VI_NULL, &address);

  /* Read instrument id register contents */
  viPeek16(dmm, address, &id_reg);

  /* Read device type register contents */
  /* ViAddr is defined as a void * so we must cast it to something else to do pointer arithmetic */
  viPeek16(dmm, (ViAddr)((ViUInt16 *)address + 0x01), &devtype_reg);

  /* Unmap memory space */
  viUnmapAddress(dmm);

  /* Print results */
  printf ("ID Register = 0x%4X\n", id_reg);
  printf ("Device Type Register = 0x%4X\n", devtype_reg);

  /* Close sessions */
  viClose(dmm);
  viClose(defaultRM);
}
Using Low/High-Level Memory I/O Methods

VISA supports three different memory I/O methods for accessing memory on the VXI backplane, as shown. All three of these access methods can be used to read and write VXI memory in the A16, A24, and A32 address spaces. The best method to use depends on the VISA program characteristics.

- **Low-level `viPeek/viPoke`**
  - `viMapAddress`
  - `viUnmapAddress`
  - `viPeek8, viPeek16, viPeek32`
  - `viPoke8, viPoke16, viPoke32`

- **High-level `viIn/viOut`**
  - `viIn8, viIn16, viIn32`
  - `viOut8, viOut16, viOut32`

- **High-level `viMoveIn/viMoveOut`**
  - `viMoveIn8, viMoveIn16, viMoveIn32`
  - `viMoveOut8, viMoveOut16, viMoveOut32`

### Using Low-Level `viPeek/viPoke`

Low-level `viPeek/viPoke` is the most efficient in programs that require repeated access to different addresses in the same memory space.

The advantages of low-level `viPeek/viPoke` are:

- Individual `viPeek/viPoke` calls are faster than `viIn/viOut` or `viMoveIn/viMoveOut` calls.
- Memory pointer may be directly dereferenced in some cases for the lowest possible overhead.

The disadvantages of low-level `viPeek/viPoke` are:

- `viMapAddress` call is required to set up mapping before `viPeek/viPoke` can be used.
- `viPeek/viPoke` calls do not return status codes.
- Only one active `viMapAddress` is allowed per `vi` session.
- There may be a limit to the number of simultaneous active `viMapAddress` calls per process or system.
Using High-level viIn/viOut

High-level viIn/viOut calls are best in situations where a few widely scattered memory access are required and speed is not a major consideration.

The advantages high-level viIn/viOut are:

- Simplest method to implement.
- No limit on number of active maps.
- A16, A24, and A32 memory access can be mixed in a single vi session.

The disadvantage of high-level viIn/viOut calls is that they are slower than viPeek/viPoke.

Using High-level viMoveIn/viMoveOut

High-level viMoveIn/viMoveOut calls provide the highest possible performance for transferring blocks of data to or from the VXI backplane. Although these calls have higher initial overhead than the viPeek/viPoke calls, they are optimized on each platform to provide the fastest possible transfer rate for large blocks of data.

For small blocks, the overhead associated with viMoveIn/viMoveOut may actually make these calls longer than an equivalent loop of viIn/viOut calls. The block size at which viMoveIn/viMoveOut becomes faster depends on the particular platform and processor speed.

The advantages of high-level viMoveIn/viMoveOut are:

- Simple to use.
- No limit on number of active maps.
- A16, A24, and A32 memory access can be mixed in a single vi session.
- Provides the best performance when transferring large blocks of data.
- Supports both block and FIFO mode.

The disadvantage of viMoveIn/viMoveOut calls is that they have higher initial overhead than viPeek/viPoke.
Example: Using VXI Memory I/O

This program demonstrates using various types of VXI memory I/O.

/* memio.c
   This example program demonstrates the use of various memory I/O methods in VISA. */

#include <visa.h>
#include <stdlib.h>
#include <stdio.h>
#define VXI_INST "VXI0::24::INSTR"

void main () {
    ViSession defaultRM, vi;
    ViAddr address;
    ViUInt16 accessMode;
    unsigned short *memPtr16;
    unsigned short id_reg;
    unsigned short devtype_reg;
    unsigned short memArray[2];

    viOpenDefaultRM (&defaultRM);
    viOpen (defaultRM, VXI_INST, VI_NULL, VI_NULL, &vi);

    /* Map into memory space */
    viMapAddress (vi, VI_A16_SPACE, 0x00, 0x10, VI_FALSE, VI_NULL, &address);

    /* Using viPeek */
    viPeek16 (vi, address, &id_reg);

    /* Read device type register contents */
    viPeek16 (vi, address, &devtype_reg);

    /* Read instrument id register contents */
    viAddr is defined as a (void *) so we must cast it to something else in order to do pointer arithmetic. */
viPeek16 (vi, (ViAddr)((ViUInt16 *)address + 0x01),
   &devtype_reg);

/* Print results */
printf (" viPeek16: ID Register = 0x%4X\n", id_reg);
printf (" viPeek16: Device Type Register = 0x%4X\n",
   devtype_reg);

/* Use direct memory dereferencing if supported */
viGetAttribute( vi, VI_ATTR_WIN_ACCESS, &accessMode );
if ( accessMode == VI_DEREF_ADDR ) {

   /* assign pointer to variable of correct type */
   memPtr16 = (unsigned short *)address;

   /* do the actual memory reads */
   id_reg = *(memPtr16);
   devtype_reg = *(memPtr16+1);

   /* Print results */
   printf ("dereference: ID Register = 0x%4X\n", id_reg);
   printf ("dereference: Device Type Register = 0x%4X\n",
         devtype_reg);
}

/* Unmap memory space */
viUnmapAddress (vi);

/*================================================
 High Level memory I/O = viIn16
================================================= */

/* Read instrument id register contents */
viIn16 (vi, VI_A16_SPACE, 0x00, &id_reg);

/* Read device type register contents */
viIn16 (vi, VI_A16_SPACE, 0x02, &devtype_reg);

/* Print results */
printf (" viIn16: ID Register = 0x%4X\n", id_reg);
printf (" viIn16: Device Type Register = 0x%4X\n",
   devtype_reg);
High Level block memory I/O = viMoveIn16

The viMoveIn/viMoveOut commands do both block read/write and FIFO read write. These commands offer the best performance for reading and writing large data blocks on the VXI backplane. For this example we are only moving 2 words at a time. Normally, these functions would be used to move much larger blocks of data.

If the value of VI_ATTR_SRC_INCREMENT is 1 (the default), viMoveIn does a block read. If the value of VI_ATTR_SRC_INCREMENT is 0, viMoveIn does a FIFO read. If the value of VI_ATTR_DEST_INCREMENT is 1 (the default), viMoveOut does a block write. If the value of VI_ATTR_DEST_INCREMENT is 0, viMoveOut does a FIFO write.

Demonstrate block read.
Read instrument id register and device type register into an array.*/
viMoveIn16 (vi, VI_A16_SPACE, 0x00, 2, memArray);

/* Print results */
printf (" viMoveIn16: ID Register = 0x%4X
", memArray[0]);
printf (" viMoveIn16: Device Type Register = 0x%4X
", memArray[1]);

/* Demonstrate FIFO read.
First set the source increment to 0 so we will repetitively read from the same memory location.*/
viSetAttribute( vi, VI_ATTR_SRC_INCREMENT, 0 );

/* Do a FIFO read of the Id Register */
viMoveIn16 (vi, VI_A16_SPACE, 0x00, 2, memArray);

/* Print results */
printf (" viMoveIn16: 1 ID Register = 0x%4X
", memArray[0]);
printf (" viMoveIn16: 2 ID Register = 0x%4X
", memArray[1]);

/* Close sessions */
viClose (vi);
viClose (defaultRM); }
Using the Memory Access Resource

For VISA 1.1 and later, the Memory Access (MEMACC) Resource type has been added to VXI and GPIB-VXI. VXI::MEMACC and GPIB-VXI::MEMACC allow access to all of the A16, A24, and A32 memory by providing the controller with access to arbitrary registers or memory addresses on memory-mapped buses.

The MEMACC resource, like any other resource, starts with the basic operations and attributes of other VISA resources. For example, modifying the state of an attribute is done via the operation `viSetAttribute` (see Appendix B - VISA Resource Classes for details).

Memory I/O Services

Memory I/O services include high-level memory I/O services and low-level memory I/O services.

High-Level Memory I/O Services
High-level Memory I/O services allow register-level access to the interfaces that support direct memory access, such as the VXIbus, VMEbus, MXIbus, or even VME or VXI memory through a system controlled by a GPIB-VXI controller. A resource exists for each interface to which the controller has access.

You can access memory on the interface bus through operations such as `viIn16` and `viOut16`. These operations encapsulate the map/unmap and peek/poke operations found in the low-level service. There is no need to explicitly map the memory to a window.

Low-Level Memory I/O Services
Low-level Memory I/O services also allow register-level access to the interfaces that support direct memory access. Before an application can use the low-level service on the interface bus, it must map a range of addresses using the operation `viMapAddress`.

Although the resource handles the allocation and operation of the window, the programmer must free the window via `viUnMapAddress` when finished. This makes the window available for the system to reallocate.
Example: MEMACC Resource Program

This program demonstrates one way to use the MEMACC resource to open the entire VXI A16 memory and then calculate an offset to address a specific device.

```c
/* peek16.c */
#include <stdio.h>
#include <stdlib.h>
#include <visa.h>
#define EXIT 1
#define NO_EXIT 0

/* This function simplifies checking for VISA errors. */
void checkError( ViSession vi, ViStatus status, char *errStr,
    int doexit){
    char buf[256];
    if (status >= VI_SUCCESS) return;
    buf[0] = 0;
    viStatusDesc( vi, status, buf );
    printf( "ERROR 0x%lx (%s)
'" , status, errStr, 
        buf );
    if ( doexit == EXIT )
        exit ( 1 );
}

void main() { ViSession drm;
    ViSession vi;
    ViUInt16 inData16 = 0;
    ViUInt16 peekData16 = 0;
    ViUInt8 *addr;
    ViUInt16 *addr16;
    ViStatus status;
    ViUInt16 offset;

    status = viOpenDefaultRM ( &drm );
    checkError( 0, status, "viOpenDefaultRM", EXIT );

    /* Open a session to the VXI MEMACC Resource*/
    status = viOpen( drm, "vxi0::memacc", VI_NULL, VI_NULL,
        &vi );
    checkError (0, status, "viOpen", EXIT );
```
/* Calculate the A16 offset of the VXI REgisters for the device at VXI logical address 8. */
offset = 0xc000 + 64 * 8;

/* Open a map to all of A16 memory space. */
status = viMapAddress(vi, VI_A16_SPACE, 0, 0x10000, VI_FALSE, 0, (ViPAddr *)&addr);
checkError( vi, status, "viMapAddress", EXIT );

/* Offset the address pointer retruned from viMapAddress for use with viPeek16. */
addr16 = (ViUInt16 *) (addr + offset);

/* Peek the contents of the card’s ID register (offset 0 from card’s base address. Note that viPeek does not return a status code. */
viPeek16( vi, addr16, &peekData16 );

/* Now use viIn16 and read the contents of the same register */
status = viIn16(vi, VI_A16_SPACE, (ViBusAddress)offset, &inData16 );
checkError(vi, status, "viIn16", NO_EXIT );

/* Print the results. */
printf( "inData16 : 0x%04hx\n", inData16 );
printf( "peekData16: 0x%04hx\n", peekData16 );

viClose( vi );
viClose (drm );
}
MEMACC Attribute Descriptions

### Generic MEMACC Attributes
The following Read Only attributes (VI_ATTR_TMO_VALUE is Read/Write) provide general interface information.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VI_ATTR_INTF_TYPE</td>
<td>Interface type of the given session.</td>
</tr>
<tr>
<td>VI_ATTR_INTF_NUM</td>
<td>Board number for the given interface.</td>
</tr>
<tr>
<td>VI_ATTR_TMO_VALUE</td>
<td>Minimum timeout value to use, in milliseconds. A timeout value of VI_TMO_IMMEDIATE means operation should never wait for the device to respond. A timeout value of VI_TMO_INFINITE disables the timeout mechanism.</td>
</tr>
<tr>
<td>VI_ATTR_INTF_INST_NAME</td>
<td>Human-readable text describing the given interface.</td>
</tr>
<tr>
<td>VI_ATTR_DMA_ALLOW_EN</td>
<td>Specifies whether I/O accesses should use DMA (VI_TRUE) or Programmed I/O (VI_FALSE).</td>
</tr>
</tbody>
</table>

### VXI and GPIB-VXI Specific MEMACC Attributes
The following attributes, most of which are read/write, provide memory window control information.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VI_ATTR_VXI_LA</td>
<td>Logical address of the local controller.</td>
</tr>
<tr>
<td>VI_ATTR_SRC_INCREMENT</td>
<td>Used in viMoveInxx operation to specify how much the source offset is to be incremented after every transfer. The default value is 1 and the viMoveInxx operation moves from consecutive elements. If this attribute is set to 0, the viMoveInxx operation will always read from the same element, essentially treating the source as a FIFO register.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>VI_ATTR_DEST_INCREMENT</td>
<td>Used in viMoveOutxx operation to specify how much the destination offset is to be incremented after every transfer. The default value is 1 and the viMoveOutxx operation moves into consecutive elements.</td>
</tr>
<tr>
<td></td>
<td>If this attribute is set to 0, the viMoveOutxx operation will always write to the same element, essentially treating the destination as a FIFO register.</td>
</tr>
<tr>
<td>VI_ATTR_WIN_ACCESS</td>
<td>Specifies modes in which the current window may be addressed: not currently mapped, through the viPeekxx or viPokexx operations only, or through operations and/or by directly de-referencing the address parameter as a pointer.</td>
</tr>
<tr>
<td>VI_ATTR_WIN_BASE_ADDR</td>
<td>Base address of the interface bus to which this window is mapped.</td>
</tr>
<tr>
<td>VI_ATTR_WIN_SIZE</td>
<td>Size of the region mapped to this window.</td>
</tr>
<tr>
<td>VI_ATTR_SRC_BYTE_ORDER</td>
<td>Specifies the byte order used in high-level access operations, such as viInxx and viMoveInxx, when reading from the source.</td>
</tr>
<tr>
<td>VI_ATTR_DEST_BYTE_ORDER</td>
<td>Specifies the byte order used in high-level access operations, such as viOutxx and viMoveOutxx, when writing to the destination.</td>
</tr>
<tr>
<td>VI_ATTR_WIN_BYTE_ORDER</td>
<td>Specifies the byte order used in low-level access operations, such as viMapAddress, viPeekxx, and viPokexx, when accessing the mapped window.</td>
</tr>
<tr>
<td>VI_ATTR_SRC_ACCESS_PRIV</td>
<td>Specifies the address modifier used in high-level access operations, such as viInxx and viMoveInxx, when reading from the source.</td>
</tr>
<tr>
<td>VI_ATTR_DEST_ACCESS_PRIV</td>
<td>Specifies address modifier used in high-level access operations such as viOutxx and viMoveOutxx, when writing to destination.</td>
</tr>
<tr>
<td>VI_ATTR_WIN_ACCESS_PRIV</td>
<td>Specifies address modifier used in low-level access operations, such as viMapAddress, viPeekxx, and viPokexx, when accessing the mapped window.</td>
</tr>
</tbody>
</table>
### GPIB-VXI Specific

The following Read Only attributes provide specific address information about GPIB hardware.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VI_ATTR_INTF_PARENT_NUM</td>
<td>Board number of the GPIB board to which the GPIB-VXI is attached.</td>
</tr>
<tr>
<td>VI_ATTR_GPIB_PRIMARY_ADDR</td>
<td>Primary address of the GPIB-VXI controller used by the session.</td>
</tr>
<tr>
<td>VI_ATTR_GPIB_SECONDARY_ADDR</td>
<td>Secondary address of the GPIB-VXI controller used by the session.</td>
</tr>
</tbody>
</table>

### MEMACC Attributes

The following Read Only events provide notification that an asynchronous operation has completed.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VI_ATTR_EVENT_TYPE</td>
<td>Unique logical identifier of the event.</td>
</tr>
<tr>
<td>VI_ATTR_STATUS</td>
<td>Return code of the asynchronous I/O operation that has completed.</td>
</tr>
<tr>
<td>VI_ATTR_JOB_ID</td>
<td>Job ID of the asynchronous I/O operation that has completed.</td>
</tr>
<tr>
<td>VI_ATTR_BUFFER</td>
<td>Address of a buffer used in an asynchronous operation.</td>
</tr>
<tr>
<td>VI_ATTR_RET_COUNT</td>
<td>Actual number of elements that were asynchronously transferred.</td>
</tr>
</tbody>
</table>
Using VXI-Specific Attributes

VXI specific attributes can be useful to determine the state of your VXI system. Attributes are read only and read/write. Read only attributes specify things such as the logical address of the VXI device and information about where your VXI device is mapped. This section shows how you might use some of the VXI specific attributes. See Appendix B - VISA Resource Classes for information on VISA attributes.

Using the Map Address as a Pointer

The `VI_ATTR_WIN_ACCESS` read-only attribute specifies how a window can be accessed. You can access a mapped window with the VISA low-level memory functions or with a C pointer if the address is de-referenced. To determine how to access the window, read the `VI_ATTR_WIN_ACCESS` attribute.

The `VI_ATTR_WIN_ACCESS` read-only attribute can be set to one of the following:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VI_NMAPPED</td>
<td>Specifies that the window is not mapped.</td>
</tr>
<tr>
<td>VI_USE_Opers</td>
<td>Specifies that the window is mapped and you can only use the low-level memory functions to access the data.</td>
</tr>
<tr>
<td>VI_DEREF_ADDR</td>
<td>Specifies that the window is mapped and has a de-referenced address. In this case you can use the low-level memory functions to access the data, or you can use a C pointer. Using a de-referenced C pointer will allow faster access to data.</td>
</tr>
</tbody>
</table>
G.2  SICL Library
Standard Instrument Control Library

User’s Guide
Notice

The information contained in this document is subject to change without notice.

Test & Measurement Systems Inc. (TAMS) shall not be liable for any errors contained in this document. *TAMS makes no warranties of any kind with regard to this document, whether express or implied. TAMS specifically disclaims the implied warranties of merchantability and fitness for a particular purpose.* TAMS shall not be liable for any direct, indirect, special, incidental, or consequential damages, whether based on contract, tort, or any other legal theory, in connection with the furnishing of this document or the use of the information in this document.

Warranty Information

A copy of the specific warranty terms applicable to your Test & Measurement Systems Inc. product and replacement parts can be obtained from your local Sales and Service Office.

U.S. Government Restricted Rights

The Software and Documentation have been developed entirely at private expense. They are delivered and licensed as “commercial computer software” as defined in DFARS 252.227-7013 (Oct 1988), DFARS 252.211-7015 (May 1991) or DFARS 252.227-7014 (Jun 1995), as a “commercial item” as defined in FAR 2.101(a), or as “Restricted computer software” as defined in FAR 52.227-19 (Jun 1987) (or any equivalent agency regulation or contract clause), whichever is applicable. You have only those rights provided for such Software and Documentation by the applicable FAR or DFARS clause or the TAMS standard software agreement for the product involved.
Printing History

This is the seventh edition of the *Standard Instrument Control Library User’s Guide (for HP-UX)*. Note: on previous editions the Reference section was actually a separate manual. On previous versions, the manual was operating system specific.

- Edition 1 - May 1994
- Edition 2 - September 1994
- Edition 3 - January 1995
- Edition 4 - November 1995
- Edition 6 - August 2001
- Edition 7 - August 2003
Contents

SICL User’s Guide
Edition 7

1. Introduction
   SICL Overview................................................................. 20
   SICL Features................................................................. 20
   SICL User................................................................. 21
   Related Documents..................................................... 22
   Other SICL Learning Products.............................. 22
   Other Documentation............................................... 23

2. Getting Started with SICL
   Reviewing a SICL Program................................. 27
   Compiling and Linking a SICL Program ................. 29
   Using Shared Libraries.................................. 29
   Running an SICL Program.......................... 31
   Getting Online Help................................. 32
   Using Manual Pages................................. 32
   Where to Go Next................................. 33

3. Using SICL
   Including the sicl.h Header File.......................... 37
   Opening a Communications Session..................... 38
   Device Sessions........................................ 39
   Interface Sessions..................................... 40
   Commander Sessions.................................. 41
   Sending I/O Commands........................................ 42
   Formatted I/O........................................... 42
   Non-Formatted I/O...................................... 52
   Using Asynchronous Events............................. 54
   SRQ Handlers............................................. 54
   Interrupt Handlers....................................... 55
   Temporarily Disabling/Enabling Asynchronous Events... 55
   Asynchronous Events and Unix Signals.................. 57
   Interrupt Handler Example............................. 59
   Using Error Handlers....................................... 61
4. Using SICL with GPIB

Creating a Communications Session
with GPIB ................................................................. 71

Communicating with GPIB Devices ............................................. 72
  Addressing GPIB Devices ...................................................... 72
  SICL Function Support with GPIB Device Sessions ..................... 74
  GPIB Device Session Example .............................................. 75

Communicating with GPIB Interfaces .......................................... 77
  Addressing GPIB Interfaces ................................................... 77
  SICL Function Support with
    GPIB Interface Sessions .................................................... 78
  GPIB Interface Session Examples .......................................... 79

Communicating with GPIB Commanders ........................................ 85
  Addressing GPIB Commanders ............................................... 85
  SICL Function Support with
    GPIB Commander Sessions ............................................... 87

Summary of GPIB Specific Functions ........................................... 88

5. Using SICL with GPIO

Creating a Communications Session
with GPIO ................................................................................. 91

Communicating with GPIO Interfaces ........................................ 92
  Addressing GPIO Interfaces .................................................. 92
  SICL Function Support with
    GPIO Interface Sessions .................................................... 93
  GPIO Interface Session Example ........................................... 95
  GPIO Interrupts Example ...................................................... 96

Summary of GPIO Specific Functions ........................................... 98
6. Using SICL with VXI/MXI

Creating a Communications Session with VXI/MXI ............................... 103
Communicating with VXI/MXI Devices .................................................. 104
  Message-Based Devices ................................................................ 105
  Register-Based Devices .............................................................. 109
Communicating with VXI/MXI Interfaces .......................................... 118
  Addressing VXI/MXI Interface Sessions ..................................... 118
  VXI/MXI Interface Session Example .......................................... 120
Communicating with VME Devices ..................................................... 121
  Declaring Resources .................................................................... 122
  Mapping VME Memory ............................................................... 123
  Reading and Writing to the Device Registers ................................ 125
  Unmapping Memory Space ......................................................... 125
  VME Interrupts ........................................................................... 125
  VME Example ............................................................................ 126
Looking at SICL Function Support with VXI/MXI ............................... 130
  Device Sessions ........................................................................... 130
  Interface Sessions ....................................................................... 131
Using SICL Trigger Lines ................................................................. 132
Using i?blockcopy for DMA Transfers ............................................. 135
Using VXI Specific Interrupts ............................................................ 138
  Processing VME Interrupts Example ........................................... 140
Summary of VXI/MXI Specific Functions .......................................... 141

7. Using SICL with RS-232

Creating a Communications Session with RS-232 ................................ 145
Communicating with RS-232 Devices .................................................. 146
  Addressing RS-232 Devices .......................................................... 146
  SICL Function Support with RS-232 Device Sessions .................... 148
  RS-232 Device Session Example .................................................. 149
Communicating with RS-232 Interfaces ............................................. 150
  Addressing RS-232 Interfaces ....................................................... 150
  SICL Function Support with RS-232 Interface Sessions ................. 151
  RS-232 Interface Session Example .............................................. 154
Summary of RS-232 Specific Functions ............................................ 156
8. Using SICL with LAN

Overview of SICL LAN ................................................................. 163
LAN Software Architecture ....................................................... 165
SICL LAN Server ........................................................................ 167
Considering LAN Configuration and Performance ...................... 168
Communicating with Devices over LAN ....................................... 169
LAN-gatewayed Sessions ......................................................... 169
LAN Interface Sessions .............................................................. 176
Using Timeouts with LAN .......................................................... 178
LAN Timeout Functions ............................................................. 178
Default LAN Timeout Values .................................................... 179
Timeout Configurations to Be Avoided ........................................ 182
Application Terminations and Timeouts ...................................... 183
Using Signal Handling with LAN ................................................ 184
SIGIO Signals ............................................................................... 184
SIGPIPE Signals ......................................................................... 185
Summary of LAN Specific Functions ......................................... 186

9. Troubleshooting Your SICL Program

Installing an Error Handler ....................................................... 189
Looking at Error Codes and Messages ....................................... 190
Troubleshooting SICL ............................................................... 192
Compile and Link Errors ........................................................... 192
Run-time Errors ........................................................................ 194
Troubleshooting SICL over LAN (Client and Server) ..................... 195
SICL LAN Client Problems and Possible Solutions .................... 197
SICL LAN Server Problems and Possible Solutions ................... 199
Troubleshooting SICL over RS-232 .......................................... 202
No Response from Instrument .................................................. 202
RS-232 Port Allocation and HP-UX termio Functions ................... 202
Data Received from Instrument is Garbled ............................... 203
Data Lost During Large Transfers ............................................. 203
Troubleshooting SICL over GPIO .............................................. 204
Bad Address (for iopen) ............................................................ 204
Operation Not Supported ........................................................ 205
No Device .................................................................................. 206
10. SICL Language Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IABORT</td>
<td>212</td>
</tr>
<tr>
<td>IBLOCKCOPY</td>
<td>213</td>
</tr>
<tr>
<td>IBLOCKMOVEX</td>
<td>215</td>
</tr>
<tr>
<td>ICAUSEERR</td>
<td>217</td>
</tr>
<tr>
<td>ICLEAR</td>
<td>218</td>
</tr>
<tr>
<td>ICLOSE</td>
<td>219</td>
</tr>
<tr>
<td>IDEREFPTR</td>
<td>220</td>
</tr>
<tr>
<td>IFLUSH</td>
<td>221</td>
</tr>
<tr>
<td>IFREAD</td>
<td>223</td>
</tr>
<tr>
<td>IFWRITE</td>
<td>225</td>
</tr>
<tr>
<td>IGETADDR</td>
<td>226</td>
</tr>
<tr>
<td>IGETDATA</td>
<td>227</td>
</tr>
<tr>
<td>IGETDEVADDR</td>
<td>228</td>
</tr>
<tr>
<td>IGETERRNO</td>
<td>229</td>
</tr>
<tr>
<td>IGETERRSTR</td>
<td>231</td>
</tr>
<tr>
<td>IGETGATEWAYTYPE</td>
<td>232</td>
</tr>
<tr>
<td>IGETINTFSESS</td>
<td>233</td>
</tr>
<tr>
<td>IGETINTFTYPE</td>
<td>234</td>
</tr>
<tr>
<td>IGETLOCKWAIT</td>
<td>235</td>
</tr>
<tr>
<td>IGETLU</td>
<td>236</td>
</tr>
<tr>
<td>IGETLUINFO</td>
<td>237</td>
</tr>
<tr>
<td>IGETLULIST</td>
<td>238</td>
</tr>
<tr>
<td>IGETONERROR</td>
<td>239</td>
</tr>
<tr>
<td>IGETONINTR</td>
<td>240</td>
</tr>
<tr>
<td>IGETONSIRQ</td>
<td>241</td>
</tr>
<tr>
<td>IGETSESTYPE</td>
<td>242</td>
</tr>
<tr>
<td>IGETTERMCHR</td>
<td>243</td>
</tr>
<tr>
<td>IGETTIMEOUT</td>
<td>244</td>
</tr>
<tr>
<td>IGPBATNCTL</td>
<td>245</td>
</tr>
<tr>
<td>IGPIBBUSADDR</td>
<td>246</td>
</tr>
<tr>
<td>IGPIBBUSSTATUS</td>
<td>247</td>
</tr>
<tr>
<td>IGPIBGETT1DELAY</td>
<td>249</td>
</tr>
</tbody>
</table>
A. The SICL Files

B. Updating HP-UX 9 SICL Applications
   Building SICL Applications on HP-UX 11i ........................................... 399
C. The SICL Utilities

iclear ................................................................. 403
ipeek .............................................................. 405
ipoke ............................................................... 406
iread ............................................................... 407
iwrite ............................................................. 408

D. Customizing your VXI/MXI System

Overview of VXI/MXI Configuration ........................................... 411
The VXI/MXI Resource Manager (ivxirm) .................................. 412
The VXI/MXI Configuration Files .............................................. 413
  The vximanuf.cf Configuration File ........................................ 414
  The vximodel.cf Configuration File ....................................... 414
  The dynamic.cf Configuration File ....................................... 414
  The vmedev.cf Configuration File ....................................... 415
  The irq.cf Configuration File ............................................ 415
  The cmdrsrvt.cf Configuration File .................................... 416
  The names.cf Configuration File ....................................... 416
  The oride.cf Configuration File ....................................... 416
  The ttltrig.cf Configuration File ..................................... 417
The iproc Utility (Initialization and SYSRESET) .......................... 418
Viewing the VXIbus System Configuration ................................. 421
VXI/MXI Configuration Utilities ............................................. 422
iproc ............................................................................. 423
ivxirm ............................................................................. 425
ivxisc ............................................................................. 428
Introduction
Introduction

Welcome to the Standard Instrument Control Library (SICL) User’s Guide. This manual describes how to use SICL. A getting started chapter steps you through the process of building and running a simple SICL program. The basics of SICL programming are covered in the following chapter, and later chapters describe how to use SICL with specific interfaces; GPIB, GPIO, VXI/MXI, RS-232, and, LAN. Also included is a complete SICL language Reference.

See the I/O Libraries Installation and Configuration Guide for detailed information on SICL installation and configuration.

This manual contains the following:

- **Chapter 2 - Getting Started with SICL** steps you through building and running a simple example program. This is a good place to start if you are a first-time SICL user.
- **Chapter 3 - Using SICL** describes the basics of SICL along with some detailed example programs. You can find information on communication sessions, addressing, error handling, and more.
- **Chapter 4 - Using SICL with GPIB** describes communicating over the GPIB interface. Example programs are also provided.
- **Chapter 5 - Using SICL with GPIO** describes how to communicate over the GPIO interface. Example programs are also provided.
- **Chapter 6 - Using SICL with VXI/MXI** provides detailed information about communicating over the VXIbus.
- **Chapter 7 - Using SICL with RS-232** describes how to communicate over the RS-232 interface. Example programs are also provided.
- **Chapter 8 - Using SICL with LAN** describes how to communicate over a LAN. Example programs are also provided.
- **Chapter 9 - Troubleshooting Your SICL Program** describes some of the most common SICL programming problems and provides hints to help you solve the problems.
- **Chapter 10 - SICL Language Reference** provides a complete description of all of the available SICL functions and C language syntax.
This guide also contains the following appendices:

- **Appendix A - The SICL Files** summarizes where the SICL files are located on your system.

- **Appendix B - Updating HP-UX 9 SICL Applications** describes how to update your SICL applications that were written on HP-UX 9 to work on HP-UX 11i.

- **Appendix C - The SICL Utilities** describes the SICL utilities that can be used to read and write to devices or interfaces from the command line.

- **Appendix D - Customizing your VXI/MXI System** documents how you can customize your VXI/MXI system. VXI/MXI configuration utilities are documented as well.

This guide also contains a Glossary of terms and their definitions, as well as an Index.
SICL Overview

SICL is a modular instrument communications library that works with a variety of computer architectures, I/O interfaces, and operating systems. Applications written in C or C++ using this library can be ported at the source code level from one system to another without, or with very few, changes.

SICL uses standard, commonly used functions to communicate over a wide variety of interfaces. For example, a program written to communicate with a particular instrument on a given interface can also communicate with an equivalent instrument on a different type of interface. This is possible because the commands are independent of the specific communications interface. SICL also provides commands to take advantage of the unique features of each type of interface, thus giving the programmer complete control over I/O communications.

SICL Features

SICL has several features that distinguish it from other I/O libraries:

- Portability
- Centralized error handling
- Formatted I/O
- Device, interface, and commander communications sessions
- Asynchronous event notification
SICL User

SICL is intended for instrument I/O and C/C++ programmers who are familiar with the HP-UX or Linux operating system. This manual does not attempt to teach the C programming language or instrument I/O concepts.
Related Documents

Other SICL Learning Products

- *I/O Libraries Installation and Configuration Guide* provides a detailed installation procedure with information on how to configure your system to run SICL.

- *SICL Online Help* is provided in the form of Unix manual pages (man pages).

- *SICL Example Programs* are provided in the `/opt/sicl/share/examples` directory. These examples are designed to help you develop your SICL applications more easily.
Other Documentation

• HP-UX 11i Learning Products (http://docs.hp.com/)
  - HP-UX 11i Installation and Update Guide
  - HP C/HP-UX Reference Manual
  - HP-UX Linker and Libraries User’s Guide
  - Software Distributor Administration Guide for HP-UX 11i

• Linux Learning Products
  - Redhat Linux Installation Guide
  - GCC Manuals (http://www.gnu.org/)
  - Linux Network Administrator’s Guide by Olaf Kirch (O’Reilly & Associates)

• VXI Interface Learning Products
  - TAMS 80100B PCI-VXI Controller Installation & Operations Instructions

• GPIO Interface Learning Products
  - TAMS PCI GPIO Card (71622/81622) Installation and Operations Instructions

• GPIB Interface Learning Products
  - TAMS PCI GPIB Card (70488/80488) Installation and Operations Instructions.
  - HP/Agilent E2078A User’s Guide.
  - Tutorial Description of the Hewlett-Packard Interface Bus (HP/IB)

• Series 700 RS-232 Interface Learning Products
  - The RS-232 Solution by Joe Campbell, SYBEX Publishing
Introduction

Related Documents

- LAN Learning Products
  - Networking Overview
  - Installing and Administering LAN/9000 Software
  - Administering ARPA Services

- LAN/GPIB Gateway Learning Products
  - TAMS 3010 LAN I/O Gateway Installation and Configuration Guide
  - HP/Agilent E2050 LAN/GPIB Gateway Installation and Configuration Guide

- VXIbus Consortium Specifications
  - The VMEbus Specification
  - The VMEbus Extensions for Instrumentation
  - TCP/IP Instrument Protocol Specification - VXI-11, Rev. 1.0
  - TCP/IP-VXIbus Interface Specification - VXI-11.1, Rev. 1.0
  - TCP/IP-IEEE 488.1 Interface Specification - VXI-11.2, Rev. 1.0
  - TCP/IP-IEEE 488.2 Instrument Interface Specification - VXI-11.3, Rev. 1.0
Getting Started with SICL
Getting Started with SICL

This chapter steps you through building and running your first SICL program. If you plan to develop SICL applications, go through this chapter to ensure you perform all the steps required to build and run a SICL program.

This chapter contains the following sections:

• Reviewing an SICL Program

• Compiling and Linking an SICL Program

• Running an SICL Program

• Getting Online Help

• Where to Go Next

If you need additional information on any of the SICL functions, see Chapter 10 for details.
Example programs are included in your SICL product to help you get started using SICL. Copies of the example programs are located in the /opt/sicl/share/examples directory.

The following is a simple C program that uses SCPI commands to query an GPIB instrument for its identification string and print the results.

```c
/* idn.c
 * The following program uses SICL to query an HPIB instrument for an identification string and prints the results. */
#include <stdio.h>
#include <sicl.h>              /* SICL header file */

/* Modify this line to reflect the address of your device */
#define DEVICE_ADDRESS "hpib,0"
void main()
{
    /* declare a device session id */
    INST id;
    char buf[256];

    /* error handler to exit if an error is detected */
    ionerror(I_ERROR_EXIT)

    /* open a device session with device at DEVICE_ADDRESS */
    id = iopen (DEVICE_ADDRESS);

    /* set timeout value to 1 sec */
    itimeout (id, 1000);

    /* send SCPI *RST command and prompt for id string*/
    iprintf (id, "*RST\n");
    ipromptf (id, "*IDN?\n", "%t", buf);

    /* print contents of buf */
    printf ("%s\n", buf);

    /* close device session */
    iclose (id);
}
```

Reviewing a SICL Program

Example programs are included in your SICL product to help you get started using SICL. Copies of the example programs are located in the /opt/sicl/share/examples directory.

The following is a simple C program that uses SCPI commands to query an GPIB instrument for its identification string and print the results.

```c
/* idn.c
 * The following program uses SICL to query an HPIB instrument for an identification string and prints the results. */
#include <stdio.h>
#include <sicl.h>              /* SICL header file */

/* Modify this line to reflect the address of your device */
#define DEVICE_ADDRESS "hpib,0"
void main()
{
    /* declare a device session id */
    INST id;
    char buf[256];

    /* error handler to exit if an error is detected */
    ionerror(I_ERROR_EXIT)

    /* open a device session with device at DEVICE_ADDRESS */
    id = iopen (DEVICE_ADDRESS);

    /* set timeout value to 1 sec */
    itimeout (id, 1000);

    /* send SCPI *RST command and prompt for id string*/
    iprintf (id, "*RST\n");
    ipromptf (id, "*IDN?\n", "%t", buf);

    /* print contents of buf */
    printf ("%s\n", buf);

    /* close device session */
    iclose (id);
}
```
Note

The newline character (\n) in the `iprintf` and `ipromptf` functions in the previous example flushes the output buffer to the device and appends an END indicator to the newline. Sometimes flushing is needed for the device, and it is good practice to include this after each instrument command. You can specify when the buffer is flushed with the SICL `isetbuf` function. See Chapter 10 for information on this SICL function.

The SICL example program includes the following:

**sicl.h** This header file must be included at the beginning of your program to provide the function prototypes and constants defined by SICL.

**DEVICE_ADDRESS** This constant is defined specifically for this example. It is used to specify the device address. This address is then used in the `iopen` function call.

**INST** This is a type definition defined by SICL. It is used to represent a unique identifier that describes a specific device or interface.

**ionerror** This is a SICL function that installs an error handler that is automatically called if any SICL calls result in an error. `I_ERROR_EXIT` specifies that the error message is printed out and the program exited.

**iopen** This SICL function creates a device session with the device attached to the address specified in **DEVICE_ADDRESS** constant.

**timeout** This function is called to set the length of time that SICL will wait for an instrument to respond. Different timeout values can be set for different sessions as needed.

**iprintf, ipromptf** These formatted I/O functions are patterned after those used in the C programming language. They support the standard ANSI C format string, plus added formats defined specifically for instrument I/O.

**iclose** This function closes the session with the specified device.

For more details on SICL features, see Chapter 3, "Using SICL." You can also see Chapter 10 for specifics about these SICL function calls.
Compiling and Linking a SICL Program

You can create your SICL applications in C, ANSI C, or C++. When compiling and linking a C program that uses SICL, use the `-lsicl` command line option to link in the SICL library routines. The following example creates the `idn` executable file on HP-UX 11i:

```
cc -Aa -o idn idn.c -lsicl
```

on Linux, use

```
gcc -o idn idn.c -lsicl
```

- The `-Aa` option specifies ANSI C on HP-UX
- The `-o` option creates an executable file called `idn`
- The `-l` option links in the shared SICL library.

If you are building an application that was originally built on HP-UX 9, or if you need to link with the SICL archive libraries on HP-UX 9, see Appendix B, "Updating HP-UX 9 SICL Applications."

Using Shared Libraries

If your program uses a shared library that calls SICL, you must explicitly link the SICL library routines even if your program does not call SICL functions. If any part of your program performs instrument I/O, you must link the SICL library routines.

The following example shows the process of creating a shared library that calls SICL and using it with an end program on HP-UX 11i:

```
cc -Aa +z -c library.c -lsicl
ld -b -o library.sl library.o
cc -Aa -o y.y library.sl -lsicl
```

on Linux, use

```
gcc -c library.c -lsicl
ld -shared -o library.so library.o
gcc -o y.y -L -llibrary -lsicl
```
Getting Started with SICL

Compiling and Linking a SICL Program

Note: If you fail to link the SICL library routines, you may get duplicate symbol errors when linking the end program or you may get undefined symbol errors memory fault (coredump) errors when you run the program.
Running an SICL Program

Execute your SICL program by typing the program name at the command prompt. For example:

```
 idn
```

When using an HP/Agilent 54601A Four Channel Oscilloscope, you should get something similar to the following:

```
Hewlett-Packard,54601A,0,1.7
```

If you have problems running the `idn` example program, first check to make sure the device address specified in your program is correct. If the program still doesn’t run, check the I/O configuration by running the `iosetup` utility. See the *I/O Libraries Installation and Configuration Guide* for information on running `iosetup`. 
Getting Online Help

Online help is offered in the form of Unix manual pages (\texttt{man} pages). You can get help on the following SICL functions:

- SICL function calls
- SICL utilities

Using Manual Pages

To use manual pages, type the Unix \texttt{man} command followed by the SICL function call or utility:

\begin{verbatim}
man name
\end{verbatim}

The following are examples of getting online help on SICL function calls and utilities: Examples of SICL function calls:

\begin{verbatim}
man iprintf
man ipromptf
man iread
\end{verbatim}

Examples of SICL utilities:

\begin{verbatim}
man ipeek
man iread
man ivxisc
\end{verbatim}
Where to Go Next

Once you have your SICL example program running, you can continue with Chapter 3, "Using SICL." Additionally, you should look at the chapters that describe how to use SICL with your particular I/O interface:

- Chapter 4 - "Using SICL with GPIB"
- Chapter 5 - "Using SICL with GPIO"
- Chapter 6 - "Using SICL with VXI/MXI"
- Chapter 7 - "Using SICL with RS-232"
- Chapter 8 - "Using SICL with LAN"

If you have any problems, see Chapter 9, "Troubleshooting Your SICL Program."
Chapter 2

Getting Started with SICL

Where to Go Next
Using SICL
Using SICL

This chapter first describes how to use SICL and some of the basic features, such as error handling and locking. Detailed example programs are also provided to help you understand how these features work. Copies of the example programs are located in the /opt/sicl/share/examples directory.

This chapter contains the following sections:

- Including the sicl.h Header File
- Opening a Communications Session
- Sending I/O Commands
- Using Asynchronous Events
- Using Error Handlers
- Using Locking

For specific details on SICL function calls, see Chapter 10.
Including the sicl.h Header File

You must include the sicl.h header file at the beginning of every file that contains SICL calls. This header file contains the SICL function prototypes and the definitions for all SICL constants and error codes:

```
#include <sicl.h>
```
Opening a Communications Session

A communications session is a channel of communication with a particular device, interface, or commander:

- A **device session** is used to communicate with a specific device connected to an interface. A device is a unit that receives commands from a controller. Typically a device is an instrument but could be a computer, a plotter, or a printer.
- An **interface session** is used to communicate with a specified interface. Interface sessions allow you to use interface specific functions (for example, `igpibsendcmd`).
- A **commander session** is used to communicate with the interface commander. Typically a commander session is used when a computer connected to the interface is acting like a device.

There are two parts to opening a communication session with a specific device, interface, or commander. First, you must create an instance of a SICL session by declaring a variable of type `INST`. Once the variable is declared, then you can open the communication channel by using the SICL `iopen` function:

```c
INST id; id = iopen (addr);
```

Where `id` is declared with the type `INST` and communicates to a device, interface, or commander. The `addr` parameter is a string expression which specifies a device session address, interface session address, or a commander session address. See the sections that follow for details on creating the different types of communications sessions.

Your program may have several sessions open at the same time by creating multiple `INST` identifiers with the `iopen` function. Use the SICL `iclose` function to close a channel of communication.
Device Sessions

A device session allows you direct access to a device without worrying about the type of interface to which it is connected. On GPIB, for example, you do not have to address a device to listen before sending data to it. This insulation makes applications more robust and portable across interfaces, and is recommended for most applications.

Device sessions are the recommended way of communicating using SICL. They provide the highest level of programming, best overall performance, and best portability.

Addressing Device Sessions

To create a device session, specify either the interface symbolic name or logical unit and a particular device’s address in the addr parameter of the iopen function. The interface symbolic name and logical unit are defined during the system configuration. See the I/O Libraries Installation and Configuration Guide for information on these values.

The logical unit is an integer corresponding to the interface. The device address generally consists of the symbolic name or logical unit and an integer that corresponds to the device’s address. It may also include a secondary address which is also an integer.

Note

Secondary addressing is not supported on the VXI and RS-232 interfaces.
The following are valid device addresses:

- **7,23**  
  Device at bus address 23 connected to an interface card at logical unit 7.

- **7,23,1**  
  Device at bus address 23, secondary address 1, connected to an interface card at logical unit 7.

- **hpib,23**  
  Device at bus address 23 and symbolic name hpib.

- **hpib2,23,1**  
  Device at bus address 23, secondary address 1, connected to a second GPIB interface with symbolic name hpib2.

- **vxi,128**  
  Device at logical address 128 and symbolic name vxi.

The following is an example of opening a device session with the GPIB device at bus address 23:

```
INST dmm;
  dmm = iopen("hpib,23");
```

More on addressing specific devices can be found in the interface-specific chapter (for example, “Using SICL with GPIB”) later in this manual.

### Interface Sessions

An interface session allows low-level control of the specified interface. There is a full set of interface-specific SICL functions for programming features that are specific to a particular interface type (GPIB, VXI, etc). This gives you full control of the activities on a given interface, but does make for less portable code.

**Addressing Interface Sessions**

To create an interface session, specify either the interface **symbolic name** or **logical unit** in the `addr` parameter of the iopen function. The `addr` parameter should be a string that includes either the symbolic name or logical unit corresponding to the interface you wish to control.

The **logical unit** is an integer that corresponds to a specific interface. The **symbolic name** is a string which uniquely describes the interface.
Using SICL
Opening a Communications Session

The following are valid interface addresses:

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Interface card at logical unit 7.</td>
</tr>
<tr>
<td>hpiib</td>
<td>GPIB interface with the symbolic name hpiib.</td>
</tr>
<tr>
<td>hpiib2</td>
<td>Second GPIB interface with the symbolic name hpiib2.</td>
</tr>
</tbody>
</table>

The following example opens an interface session with the GPIB interface:

```
INST dmm;
dmm = iopen("hpib");
```

More on addressing specific interfaces can be found in the interface-specific chapter (for example, "Using SICL with GPIB") later in this manual.

**Commander Sessions**

The commander session allows you to talk to the interface controller. Typically, the controller is the computer used to communicate with devices on the interface. However, when the controller is no longer the active controller, or passes control, commander sessions can be used to talk to the controller. In this mode, the interface is acting like a device on the interface (non-controller).

To create a commander session, specify either the interface symbolic name or logical unit followed by a comma and then the string cmdr in the iopen function. The interface symbolic name and logical unit are defined during the system configuration. See the I/O Libraries Installation and Configuration Guide for information on these values. The following are valid commander addresses:

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hpiib,cmdr</td>
<td>GPIB commander session.</td>
</tr>
<tr>
<td>7,cmdr</td>
<td>Commander session on interface at logical unit 7.</td>
</tr>
</tbody>
</table>

The following is an example of creating a commander session with the GPIB interface:

```
INST cmdr; cmdr = iopen("hpib,cmdr");
```
Sending I/O Commands

Once you have established a communications session with a device, interface, or commander, you can start communicating with that session using either formatted I/O or non-formatted I/O.

- **Formatted I/O** converts mixed types of data under the control of a format string. The data is buffered, thus optimizing interface traffic. The formatted I/O routines are geared towards instruments and are very efficient in I/O.

- **Non-formatted I/O** sends or receives raw data to or from a device, interface, or commander. With non-formatted I/O, no formatting or conversion of the data is performed. Thus, if formatted data is required, it must be done by the user.

See the following sections for a complete description and examples of using formatted I/O and non-formatted I/O.

**Formatted I/O**

The SICL formatted I/O mechanism is similar to the C `stdio` mechanism. SICL formatted I/O, however, is designed specifically for instrument communication and is optimized for IEEE 488.2 compatible instruments. The three main functions for formatted I/O are as follows:

- The `fprintf` function formats according to the `format` string and sends data to the session specified by `id`:

  ```c
  fprintf (id, format [,arg1][,arg2][,...]) ;
  ```

- The `scanf` function receives data from the session specified by `id` and converts the data according to the `format` string:

  ```c
  scanf(id, format [,arg1][,arg2][,...]) ;
  ```
• The `ipromptf` function formats data according to the `writefmt` string and sends data to the session specified by `id` and then immediately receives the data and converts it according to the `readfmt` string:

```c
ipromptf(id, writefmt, readfmt [,arg1][,arg2][,...]) ;
```

See Chapter 10 for more information on these functions.

The formatted I/O functions are buffered. There are two non-buffered and non-formatted I/O functions called `iread` and `iwrite`. See the "Non-formatted I/O" section later in this chapter. These are raw I/O functions and do not intermix with the formatted I/O functions.

If raw I/O must be mixed, use the `ifread/ifwrite` functions. They have the same parameters as `iread` and `iwrite`, but read or write raw data to or from the formatted I/O buffers. Refer to the "Formatted I/O Buffers" section later in this chapter for more details.

### Formatted I/O Conversion

The formatted I/O functions convert data under the control of the format string. The format string specifies how each argument is converted before it is input or output. The typical format string syntax is as follows:

```c
%(format flags) [field width] [.precision] [, array size]
[argument modifier]conversion character
```

See `iprintf`, `ipromptf`, and `iscanf` in Chapter 10 for more information on how data is converted under the control of the format string.

**Format Flags.** Zero or more flags may be used to modify the meaning of the conversion character. The format flags are only used when sending formatted I/O (`iprintf` and `ipromptf`). The following are supported format flags:
Format Flags for `iprintf` and `ipromptf`

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@1</td>
<td>Converts to a IEEE 488.2 NR1 number.</td>
</tr>
<tr>
<td>@2</td>
<td>Converts to a IEEE 488.2 NR2 number.</td>
</tr>
<tr>
<td>@3</td>
<td>Converts to a IEEE 488.2 NR3 number.</td>
</tr>
<tr>
<td>@H</td>
<td>Converts to a IEEE 488.2 hexadecimal number.</td>
</tr>
<tr>
<td>@Q</td>
<td>Converts to a IEEE 488.2 octal number.</td>
</tr>
<tr>
<td>@B</td>
<td>Converts to a IEEE 488.2 binary number.</td>
</tr>
<tr>
<td>+</td>
<td>Prefixes number with sign (+ or -).</td>
</tr>
<tr>
<td>–</td>
<td>Left justifies result.</td>
</tr>
<tr>
<td>space</td>
<td>Prefixes number with blank space if positive or with - if negative.</td>
</tr>
<tr>
<td>#</td>
<td>Use alternate form. For o conversion, print a leading zero. For x or X, a nonzero will have 0x or 0X as a prefix. For e, E, f, g, or G, the result will always have one digit on the right of the decimal point.</td>
</tr>
<tr>
<td>0</td>
<td>Causes the left pad character to be a zero for all numeric conversion types.</td>
</tr>
</tbody>
</table>

The following example converts `numb` into a IEEE 488.2 floating point number (NR2) and sends it to the session specified by `id`:

```c
int numb = 61; iprintf (id, "%@2d", numb);
Sends:  61.000000
```
Field Width. Field width is an optional integer that specifies the minimum number of characters in the field. If the formatted data has fewer characters than specified in the field width, it will be padded. The padded character is dependent on various flags. You can use an asterisk (*) in place of the integer to indicate that the integer is taken from the next argument.

The following example pads numb to six characters and sends it to the session specified by id:

```c
int numb = 61;
iprintf (id, "%-6d", numb);
```

Inserts four characters, for a total of six characters: 61

Precision. Precision is an optional integer that is preceded by a period. When used with conversion characters e, E, and f, the number of digits to the right of the decimal point is specified. For the d, i, o, u, x, and X conversion characters, the minimum number of digits to appear is specified. For the s and S conversion characters, the precision specifies the maximum number of characters to be read from the argument. This field is only used when sending formatted I/O (iprintf and ipromptf). You can use an asterisk (*) in place of the integer to indicate that the integer is taken from the next argument.

The following example converts numb so that there are only two digits to the right of the decimal point and sends it to the session specified by id:

```c
float numb = 26.9345;
iprintf (id, "%0.2f", numb);
```

Sends: 26.93
Using SICL

Sending I/O Commands

,Array Size. The comma operator is a format modifier which allows you to read or write a comma-separated list of numbers (only valid with %d and %f conversion characters). It is a comma followed by an integer. The integer indicates the number of elements in the array argument. The comma operator has the format of ,dcl where dcl is the number of elements to read or write.

The following example specifies a comma separated list to be sent to the session specified by id:

```c
int list[5]={101,102,103,104,105};
printf (id, "%,5d", list);
```

Sends: 101,102,103,104,105

Argument Modifier. The meaning of the optional argument modifier h, l, w, z, and Z is dependent on the conversion character:

<table>
<thead>
<tr>
<th>Argument Modifier</th>
<th>Conversion Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>d, i</td>
<td>Corresponding argument is a short integer.</td>
</tr>
<tr>
<td>h</td>
<td>f</td>
<td>Corresponding argument is a float for printf or a pointer to a float for scanf.</td>
</tr>
<tr>
<td>l</td>
<td>d, i</td>
<td>Corresponding argument is a long integer.</td>
</tr>
<tr>
<td>l</td>
<td>b, B</td>
<td>Corresponding argument is a pointer to an array of long integers.</td>
</tr>
<tr>
<td>l</td>
<td>f</td>
<td>Corresponding argument is a double for printf or a pointer to a double for scanf.</td>
</tr>
<tr>
<td>w</td>
<td>b, B</td>
<td>Corresponding argument is a pointer to an array of short integers.</td>
</tr>
<tr>
<td>z</td>
<td>b, B</td>
<td>Corresponding argument is pointer to an array of floats.</td>
</tr>
<tr>
<td>Z</td>
<td>b, B</td>
<td>Corresponding argument is a pointer to an array of doubles.</td>
</tr>
</tbody>
</table>
Conversion Characters. The conversion characters for sending and receiving formatted I/O are different. The following tables summarize the conversion characters for each:

**fprintf and ipromptf Conversion Characters**

<table>
<thead>
<tr>
<th>Conversion Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>d, i</td>
<td>Corresponding argument is an integer</td>
</tr>
<tr>
<td>f</td>
<td>Corresponding argument is a double.</td>
</tr>
<tr>
<td>b, B</td>
<td>Corresponding argument is a pointer to an arbitrary block of data.</td>
</tr>
<tr>
<td>c, C</td>
<td>Corresponding argument is a character.</td>
</tr>
<tr>
<td>t</td>
<td>Controls whether the END indicator is sent with each LF character in the format string.</td>
</tr>
<tr>
<td>s, S</td>
<td>Corresponding argument is a pointer to a null terminated string.</td>
</tr>
<tr>
<td>%</td>
<td>Sends an ASCII percent (%) character.</td>
</tr>
<tr>
<td>o, u, x, X</td>
<td>Corresponding argument is an unsigned integer.</td>
</tr>
<tr>
<td>e, E, g, G</td>
<td>Corresponding argument is a double.</td>
</tr>
<tr>
<td>n</td>
<td>Corresponding argument is a pointer to an integer.</td>
</tr>
<tr>
<td>f</td>
<td>Corresponding argument is a pointer to a FILE descriptor opened for reading.</td>
</tr>
</tbody>
</table>

The following example sends an arbitrary block of data to the session specified by the `id` parameter. The asterisk (*) is used to indicate that the number is taken from the next argument:

```c
long int size = 1024;
char data [1024];
.
.
iprintf (id, "%*b", size, data);
```

Sends 1024 characters of block data.
### iscanf and ipromptf Conversion Characters

<table>
<thead>
<tr>
<th>Conversion Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>d, i, n</td>
<td>Corresponding argument must be a pointer to an integer.</td>
</tr>
<tr>
<td>e, f, g</td>
<td>Corresponding argument must be a pointer to a float.</td>
</tr>
<tr>
<td>c</td>
<td>Corresponding argument is a pointer to a character sequence.</td>
</tr>
<tr>
<td>s, S, t</td>
<td>Corresponding argument is a pointer to a string.</td>
</tr>
<tr>
<td>o, u, x</td>
<td>Corresponding argument must be a pointer to an unsigned integer.</td>
</tr>
<tr>
<td>[</td>
<td>Corresponding argument must be a character pointer.</td>
</tr>
<tr>
<td>F</td>
<td>Corresponding argument is a pointer to a FILE descriptor opened for writing.</td>
</tr>
</tbody>
</table>

The following example reads characters up to the first white space character from the session specified by the id parameter and puts the characters into data:

```c
char data[180];
iscanf (id, "%s", data);
```
The following ANSI C example shows how to use the formatted I/O functions to send and receive data. This example opens an GPIB communications session with a Multimeter and sends a comma operator to send a comma separated list to the Multimeter. The \lf conversion characters are then used to receive a double back from the Multimeter.

```
#include <sicl.h>
#include <stdio.h>

main()
{
    INST dvm;
    double res;
    double list[2] = {1,0.001};
    char buf[80];

    /* Print message and terminate on error */
    ionerror (I_ERROR_EXIT);

    /* Open the multimeter session */
    dvm = iopen ("hpib,16");
    itimeout (dvm, 10000);

    /* Initialize dvm */
    iprintf (dvm, "*RST\n");

    /* Set up multimeter and send comma separated list */
    iprintf (dvm, "CALC:DBM:REF 50\n");
    iprintf (dvm, "MEAS:VOLT:AC? %,2lf\n", list);

    /* Read the results */
    iscanf (dvm,"%lf", &res);

    /* Print the results */
    printf ("Result is %f\n",res);

    /* Close the multimeter session */
    iclose (dvm);
}
```
Using SICL

Sending I/O Commands

**Format String**  The format string for `iprintf` puts a special meaning on the newline character (`\n`). The newline character in the format string flushes the output buffer. All characters in the output buffer will be written with an END indicator included with the last byte (the newline character). This means that you can control at what point you want the data written. If no newline character is included in the format string for an `iprintf` call, then the converted characters are stored in the output buffer. It will require another call to `iprintf` or a call to `iflush` to have those characters written. `iflush` only sends the data queued in the buffer, and not the END indicator as in `iprintf`. Note that newline characters output from an output parameter do not cause a flush; only newlines in the format string do.

This can be very useful in queuing up data to send to a device. It can also raise I/O performance by doing a few large writes instead of several smaller writes. This behavior can be changed by the `isetbuf` and `isetubuf` functions. See the next section, "Formatted I/O Buffers."

The format string for `iscanf` ignores most white-space characters. Newlines (`\n`) and carriage returns (`\r`), however, are treated just like normal characters in the format string, which *must* match the next non-white-space character read.

**Formatted I/O Buffers**  The SICL software maintains both a read and write buffer for formatted I/O operations. Occasionally, you may want to control the actions of these buffers.

The write buffer is maintained by the `iprintf` and the write portion of the `ipromptf` functions. It queues characters to send so that they are sent in large blocks, thus increasing performance. The write buffer automatically flushes when it sends a newline character from the format string (see the `%t` conversion character to change this feature). It also flushes immediately after the write portion of the `ipromptf` function. It may occasionally be flushed at other non-deterministic times, such as when the buffer fills. When the write buffer flushes, it sends its contents.

The read buffer is maintained by the `iscanf` and the read portion of the `ipromptf` functions. It queues the data received until it is needed by the format string. The read buffer is automatically flushed before the write portion of an `ipromptf`. Flushing the read buffer destroys the data in the buffer and guarantees that the next call to `iscanf` or `ipromptf` reads data directly rather than data that was previously queued.
Note

Flushing the read buffer also includes reading all pending response data from a device. If the device is still sending data, the flush process will continue to read data from the device until it receives an END indicator from the device.

See the **isetbuf** function for other options for buffering data.

### Overview of Formatted I/O

The following set of functions are related to formatted I/O:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ifread</strong></td>
<td>Obtains raw data directly from the read formatted I/O buffer. This is the same buffer that <em>iscanf</em> uses.</td>
</tr>
<tr>
<td><strong>ifwrite</strong></td>
<td>Writes raw data directly to the write formatted I/O buffer. This is the same buffer that <em>iprintf</em> uses.</td>
</tr>
<tr>
<td><strong>iprintf</strong></td>
<td>Converts data via a format string and writes the arguments appropriately.</td>
</tr>
<tr>
<td><strong>iscanf</strong></td>
<td>Reads data, converts this data via a format string, and assigns the values to your arguments.</td>
</tr>
<tr>
<td><strong>ipromptf</strong></td>
<td>Sends, then receives, data from a device/instrument. It also converts data via format strings that are identical to <em>iprintf</em> and <em>iscanf</em>. The advantage of this function is that the <em>iprintf</em> and <em>iscanf</em> parts are done together.</td>
</tr>
<tr>
<td><strong>iflush</strong></td>
<td>Flushes the formatted I/O read and write buffers. A flush of the read buffer means that any data in the buffer is lost. A flush of the write buffer means that any data in the buffer is written to the session’s target address.</td>
</tr>
<tr>
<td><strong>isetbuf</strong></td>
<td>Sets the size of the formatted I/O read and the write buffers. A size of zero (0) means no buffering. Note that if no buffering is used, performance can be severely affected.</td>
</tr>
<tr>
<td><strong>isetubuf</strong></td>
<td>Sets the read or the write buffer to your allocated buffer. The same buffer cannot be used for both reading and writing. Also you should be careful in using buffers that are automatically allocated.</td>
</tr>
</tbody>
</table>
Non-Formatted I/O

There are two non-buffered, non-formatted I/O functions called iread and iwrite. These are raw I/O functions and do not intermix with the formatted I/O functions. If raw I/O must be mixed, use the ifread and ifwrite functions. They have the same parameters as iread and iwrite, but read or write raw data to or from the formatted I/O buffers.

The non-formatted I/O functions are described as follows:

- The iread function reads raw data from the device or interface specified by the id parameter and stores the results in the location where buf is pointing:

  iread(id, buf, bufsize, reason, actualcnt);

- The iwrite function sends the data pointed to by buf to the interface or device specified by the id parameter:

  iwrite(id, buf, datalen, end, actualcnt);

See Chapter 10 for more information on these functions.
Non-formatted I/O Example

The following example illustrates using non-formatted I/O to communicate with a Multimeter over the GPIB interface. The SICL non-formatted I/O functions iwrite and iread are used for the communication. A similar example is used to illustrate formatted I/O later in this chapter.

```c
/* nonformatio.c
 This example program measures AC voltage on a multimeter and prints out the results */
#include <sicl.h>
#include <stdio.h>

main()
{
    INST dvm;
    char strres[20];

    /* Print message and terminate on error */
    ionerror (I_ERROR_EXIT);

    /* Open the multimeter session */
    dvm = iopen("hpib,16");
    itimeout (dvm, 10000);

    /* Initialize dvm */
    iwrite (dvm, "*RST\n", 5, 1, NULL);

    /* Set up multimeter and take measurement */
    iwrite (dvm,"CALC:DBM:REF 50\n", 16, 1, NULL);
    iwrite (dvm,"MEAS:VOLT:AC? 1, 0.001\n", 23, 1, NULL);

    /* Read measurements */
    iread (dvm, strres, 20, NULL, NULL);

    /* Print the results */
    printf("Result is %s\n", strres);

    /* Close the multimeter session */
    iclose(dvm);
}
```
Using Asynchronous Events

Asynchronous events are events that happen outside the control of your application. These events include Service Requests (SRQ) and interrupts. An SRQ is a notification that a device requires service. Any device can generate an SRQ. Both devices and interfaces can generate interrupts.

By default, creating a session enables asynchronous events. However, the library will not report any events to the application until the appropriate handlers are installed in your program.

SRQ Handlers

The ionsrq function installs an SRQ handler. The currently installed SRQ handler is called any time its corresponding device or interface generates an SRQ. If an interface is unable to determine which device on the interface generated the SRQ, all SRQ handlers assigned to that interface will be called.

Therefore, an SRQ handler cannot assume that its corresponding device generated an SRQ. The SRQ handler should use the ireadstb function to determine whether its device generated an SRQ. If two or more sessions refer to the same device, and have handlers installed, the handlers for each of the sessions are called.
Interrupt Handlers

Two distinct steps are required for an interrupt handler to be called. First, the interrupt handler must be installed. Second, the interrupt event or events need to be enabled. The `ionintr` function installs an interrupt handler. The `isetintr` function enables notification of the interrupt event or events.

An interrupt handler can be installed with no events enabled. Conversely, interrupt events can be enabled with no interrupt handler installed. Only when both an interrupt handler is installed and interrupt events are enabled will the interrupt handler be called.

Temporarily Disabling/Enabling Asynchronous Events

To temporarily prevent all SRQ and interrupt handlers from executing, use the `iintroff` function. This disables all asynchronous handlers for all sessions in the process.

To re-enable asynchronous SRQ and interrupt handlers previously disabled by `iintroff`, use the `iintron` function. This enables all asynchronous handlers for all sessions in the process, that had been previously enabled.

---

**Note**

These functions do not affect the `isetintr` values or the handlers (`ionsrq` or `ionintr`) in any way. See `ionintr` and `ionsrq` in Chapter 10.

Default is on.
Using SICL

Using Asynchronous Events

**Note**  It is possible to overflow SICL’s interrupt queue if too many interrupts are generated while notification is disabled.

Calls to `iintroff/iintron` may be nested, meaning that there must be an equal number of on’s and off’s. This means that calling the `iintron` function may not actually re-enable notification of interrupts.

Occasionally, you may want to suspend a process and wait until an event occurs that causes a handler to execute. The `iwaithdlr` function causes the process to suspend until either an enabled SRQ or interrupt condition occurs and the related handler executes. Once the handler completes its operation, this function returns and processing continues. For this function to work properly, your application *must* turn interrupts off before enabling asynchronous events (that is, use `iintroff`). The `iwaithdlr` function behaves as if interrupts are enabled. Interrupts are still disabled after the `iwaithdlr` function has completed. Only calls to `iintron` will re-enable interrupts.

**Note**  Interrupts must be disabled if you are using `iwaithdlr`. Use `iintroff` to disable notification of interrupts.

The reason for disabling notification of interrupts is that the interrupt may occur between the `isetintr` and `iwaithdlr` and, if you only expect one interrupt, it might come before the `iwaithdlr`. Notification may not occur, that is, the handler may not get called. This may or may not be the effect you desire.
Using SICL

Using Asynchronous Events

For example:

```c
...  
iintroff ();  
ionintr (vxi, trigger_handler);  
isetintr (vxi, I_INTR_TRIG, I_TRIG_TTL0 | I_TRIG_TTL7);  
...  
vxitrigon (vxi, I_TRIG_TTL0);  
while (!done)  
    iwaithdlr (0);  
intron ();  
...  
```

Asynchronous Events and Unix Signals

If you are using SICL LAN, see the "LAN and Signal Handling" section in Chapter 8, "Using SICL with LAN."

SICL hpib and vxi interfaces use an Unix signal to implement interrupts and SRQs. The default SICL signal is SIGUSR2. This signal is managed completely by the SICL library. Your application must avoid SICL’s signal completely. Do not attempt to mask it, send it, or install a handler for it.

If your application needs SIGUSR2 for some purpose other than SICL, you can instruct SICL to use a different signal. This is done with the isetsig function. The following example selects signal 29 for SICL use:

```c
isetsig(29);
```

If you use isetsig, you must call it before any other function in your program. Also, you must pick an alternate signal carefully to avoid conflicting with other Unix resources.
Using SICL
Using Asynchronous Events

Protecting I/O Calls Against Interrupts

It is standard Unix behavior for I/O calls like `iread` and `iprintf` to be interrupted when the process receives a signal. If your process is not expecting to receive signals, such I/O side effects will probably be masked by the other standard behavior of unexpected signals: death of your process. If you are expecting signals, you may not want them to abort SICL I/O operations.

This can be solved by blocking or ignoring any expected signals while doing I/O activity. After I/O is complete, the original signal action can be restored. The choice to block or ignore depends on the need of your application. Ignored signals are not queued; blocked signals have a one-deep queue and are acted on as soon as the block is removed.

The following programming segment shows signal blocking. SIGALARM and SIGINT are blocked during an `iscanf` call.

```c
/* temporarily block 2 signals */
old_mask = sigblock(sigmask (SIGINT) | sigmask (SIGALRM));

/* call protected I/O function */
iscanf (id, "%f", &mydata);

/* restore original signal mask */
sigsetmask (old_mask);
```
Interrupt Handler Example

The following is an ANSI C example that installs an interrupt handler and enables the interrupts on the VXI TTL trigger lines. When the TTL trigger line is asserted, the installed interrupt handler is called.

```c
/* interrupts.c
 * This is an example of the interrupt handling in SICL. This
 * program installs an interrupt handler and enables the
 * interrupts on trigger and waits for the interrupt. */
#include <sicl.h>
#include <stdio.h>
#include <unistd.h>

int intr = 0;

void trigger_handler (INST id, long reason, long secval) {
    /* indicate that the interrupt happened */
    intr = 1;
} /* end of trigger_handler */

main ()
{
    INST id;

    /* start child process to fire trigger line */
    if (fork()==0)
        child();

    ionerror (I_ERROR_EXIT);
    iintroff();

    id = iopen ("vxi");

    /* set the interrupt handler */
    ionintr (id, trigger_handler);

    /* what interrupts to handle (interrupt on ttl 0 or 7 firing) */
    isetintr (id, I_INTR_TRIG, I_TRIG_TTL0 | I_TRIG_TTL7);
```
Using SICL

Using Asynchronous Events

/* Wait for interrupt to happen (30 second timeout) */
iwaithdlr (30000);

if (intr == 1)
    printf ("Interrupt handler called.\n");
else
    printf ("ERROR: Interrupt handler not called.\n");

iclose (id);
}

cchild ()
{
    INST id;

    /* Let the parent get into iwaithdlr */
sleep (2);

    ionerror (I_ERROR_EXIT);

    id = iopen ("vxi");

    /* pulse TTL0 */
    ivxitrigon (id, I_TRIG_TTL0);
    ivxitrigoff (id, I_TRIG_TTL0);

    iclose (id);
    exit (0);
}
Using Error Handlers

When a SICL function call results in an error, it typically returns a special value such as a NULL pointer, or a non-zero error code. SICL provides a convenient mechanism for handling errors. SICL allows you to install an error handler for all SICL functions within an application.

It is important to note that error handlers are per-process, not per-session. That is, one handler will work for all sessions in a process. This allows your application to ignore the return value and simply permits the error procedure to detect errors and recover. The error handler is called before the function that generated the error completes.

The function ionerror is used to install an error handler. It is defined as follows:

```c
int ionerror (proc);
void (*proc)();
```

Where:

```c
void proc (id, error);
INST id;
int error;
```

The routine proc is the error handler and is called whenever a SICL error occurs. Two special reserved values of proc may be passed to the ionerror function:

- **I_ERROR_EXIT**
  - This value installs a special error handler which will print a diagnostic message and then terminate the process.

- **I_ERROR_NO_EXIT**
  - This value installs a special error handler which will print a diagnostic message and then allow the process to continue execution.

This mechanism has substantial advantages over other I/O libraries, because error handling code is located away from the center of your application. This makes the application easier to read and understand.
Using SICL
Using Error Handlers

Error Handler Example

Typically, in an application, error handling code is intermixed with the I/O code. However, with SICL error handling routines, no special error handling code is inserted between the I/O calls. Instead, a single line at the top (calling ionerror) installs an error handler that gets called any time a SICL call results in an error.

In this example a standard, system-defined error handler is installed that prints a diagnostic message and exits.

/* errhand.c
   This example demonstrates how a SICL error handler can be installed */

#include <sicl.h>
#include <stdio.h>

main ()
{
   INST dvm;
   double res;

   ionerror (I_ERROR_EXIT);
   dvm = iopen ("hpib,16");
   itimeout (dvm, 10000);
   fprintf (dvm, "%s\n", "MEAS:VOLT:DC?");
   iscanf (dvm, "%lf", &res);
   printf ("Result is %f\n", res);
   iclose (dvm);

   exit (0);
}
The following is an ANSI C example of writing and implementing your own error handler:

```c
/* errhand2.c
   This program shows how you can install your own error handler */

#include <sicl.h>
#include <stdio.h>

void err_handler (INST id, int error) {
    fprintf (stderr, "Error: %s
", igeterrstr (error));
    exit (1);
}

main () {
    INST dvm;
    double res;

    ierror (err_handler);
    dvm = iopen ("hpib,16");
    itimeout (dvm, 10000);
    iprintf (dvm, "%s
", "MEAS:VOLT:DC?");
    iscanf (dvm, "%lf", &res);
    printf ("Result is %f
", res);
    iclose (dvm);

    exit (0);
}
```

Now, if any of the SICL functions result in an error, your error routine will be called.

---

**Note**

If an error occurs in `iopen`, the `id` that is passed to the error handler may not be valid.
Using Locking

Because SICL allows multiple sessions on the same device or interface, the action of opening does not mean you have exclusive use. In some cases this is not an issue, but should be a consideration if you are concerned with program portability.

The SICL i_lock function is used to lock an interface or device. The SICL i_unlock function is used to unlock an interface or device.

Locks are performed on a per-session (device, interface, or commander) basis. If a session within a given process locks a device or interface, then that device or interface can only be accessed from that session.

Locks can be nested. The device or interface only becomes unlocked when the same number of unlocks are done as the number of locks. Doing an unlock without a lock returns the error I_ERR_NOLOCK.

What does it mean to lock? Locking an interface (from an interface session) restricts other device and interface sessions from accessing this interface. Locking a device restricts other device sessions from accessing this device; however, other interface sessions may continue to access the interface for this device. Locking a commander (from a commander session) restricts other commander sessions from accessing this commander.

Caution

It is possible for an interface session to access an interface which is serving a device locked from a device session. This interface access usually allows the interface session to address or reset any device on the interface. In such a case, data may be lost from the device session that was underway.

In particular, be aware that both the HP/Agilent Visual Engineering Environment (HP/Agilent VEE) and the TAMS BASIC applications use SICL interface sessions. Hence, I/O operations from either of these applications can supersede any device session that has a lock on a particular device. Use interface session locks in your own program if these applications may be running simultaneously with your program.
Not all SICL routines are affected by locks. Some routines that simply set or return session parameters never touch the interface hardware and therefore work without locks. Each function defined in Chapter 10 has a section, "Affected by functions," that lists the keyword LOCK if the function is affected by locks. Functions without this keyword are not affected.

Lock Actions

If a session tries to perform any SICL function that obeys locks on an interface or device that is currently locked by another session, the default action is to suspend the call until the lock is released or, if a timeout is set, until it times out.

This action can be changed with the isetlockwait function (see Chapter 10 for a full description). If the isetlockwait function is called with the flag parameter set to 0 (zero), the default action is changed. Rather than causing SICL functions to suspend, an error will be returned immediately.

To return to the default action, or to suspend and wait for an unlock, call the isetlockwait function with the flag set to any non-zero value.
Using SICL
Using Locking

Locking in a Multi-user Environment

In a multi-user/multi-process environment where devices are being shared, it is a good idea to use locking to help ensure exclusive use of a particular device or set of devices. (However, as explained in the previous section, "Using Locking," remember that an interface session can access a device locked from a device session.) In general, it is not friendly behavior to lock a device at the beginning of an application and unlock it at the end. This can result in deadlock or long waits by others who want to use the resource.

The recommended way to use locking is per transaction. Per transaction means that you lock before you setup the device, then unlock after all the desired data has been acquired. When sharing a device, you cannot assume the state of the device, so the beginning of each transaction should have any setup needed to configure the device or devices to be used.
Locking Example

The following example show how device locking can be used to grant exclusive access to a device by an application. This example uses an HP/Agilent 34401 Multimeter.

/* locking.c
   This example shows how device locking can be used to grant exclusive access to a device */

#include <sicl.h>
#include <stdio.h>
main() {
  INST dvm;

  char strres[20];

  /* Print message and terminate on error */
  ionerror (I_ERROR_EXIT);

  /* Open the multimeter session */
  dvm = iopen("hpib,16");
  itimeout (dvm, 10000);

  /* Lock the multimeter device to prevent access from other applications */
  ilock(dvm);

  /* Take a measurement */
  iwrite (dvm, "MEAS:VOLT:DC?\n", 14, 1, NULL);

  /* Read the results */
  iread (dvm, strres, 20, NULL, NULL);

  /* Release the multimeter device for use by others */
  iunlock(dvm);

  /* Print the results */
  printf("Result is %s\n", strres);

  /* Close the multimeter session */
  iclose(dvm);
}
Using SICL

Using Locking
Using SICL with GPIB
Using SICL with GPIB

The HPIB interface (Hewlett-Packard Interface Bus) is Hewlett-Packard’s implementation of the IEEE 488.1 Bus. Other IEEE 488 versions include GPIB (General Purpose Interface Bus) and IEEE Bus. GPIB and HPIB are both used in the discussions and examples in this chapter. The HPIB related SICL functions have the string GPIB embedded in the function name.

This chapter explains how to use SICL to communicate over GPIB. In order to communicate over GPIB, you must have loaded the GPIB fileset during the system installation. See the I/O Libraries Installation and Configuration Guide for information.

This chapter describes in detail how to open a communications session and communicate with GPIB devices, interfaces, or controllers. The example programs shown in this chapter are also provided in the /opt/sicl/share/examples directory.

This chapter contains the following sections:

• Creating a Communications Session with GPIB
• Communicating with GPIB Devices
• Communicating with GPIB Interfaces
• Communicating with GPIB Commanders
• Summary of GPIB Specific Functions
Creating a Communications Session with GPIB

Once you have determined that your GPIB system is setup and operating correctly, you may want to start programming with the SICL functions. First you must determine what type of communication session you need. The three types of communications sessions are device, interface, and commander.
Communicating with GPIB Devices

The device session allows you direct access to a device without worrying about the type of interface to which it is connected. The specifics of the interface are hidden from the user.

Addressing GPIB Devices

To create a device session, specify either the interface symbolic name or logical unit and a particular device’s address in the addr parameter of the iopen function. The interface symbolic name and logical unit are defined during the system configuration. See the I/O Libraries Installation and Configuration Guide for information on these values.

The following are example GPIB addresses for device sessions:

- `hpib,7`: A device address corresponding to the device at primary address 7 and symbolic name hpib.
- `hpib,3,2`: A device address corresponding to the device at primary address 3, secondary address 2, and symbolic name hpib.
- `hpib,9,0`: A device address corresponding to the device at primary address 9, secondary address 0, and symbolic name hpib.

Note: The above examples use the default symbolic name specified during the system configuration. If you want to change the name listed above, you must also change the symbolic name or logical unit specified during the configuration. The name used in your SICL program must match the logical unit or symbolic name specified in the system configuration. Other possible interface names are GPIB, gpib, HPIB, etc.
SICL supports both primary and secondary addressing on GPIB interfaces. Remember that the primary address must be between 0 and 30 and that the secondary address must be between 0 and 30. The primary and secondary addresses correspond to the GPIB primary and secondary addresses.

**Note** If you are using a GPIB Command Module to communicate with VXI devices, the secondary address must be specified to select a specific instrument in the cardcage. Secondary addresses of 0, 1, 2, . . . 31 correspond to VXI instruments at logical addresses of 0, 8, 16, . . . 248, respectively.

The following is an example of opening a device session with an GPIB device at bus address 16:

```c
INST dmm
dmm = iopen("hpib,16");
```
SICL Function Support with GPIB Device Sessions

The following describes how some SICL functions are implemented for GPIB device sessions.

\textit{iwrite} Causes all devices to untalk and unlisten. It then sends this controller’s talk address followed by unlisten and then the listen address of the corresponding device session. Then it sends the data over the bus.

\textit{iread} Causes all devices to untalk and unlisten. It sends an unlisten, then sends this controller’s listen address followed by the talk address of the corresponding device session. Then it reads the data from the bus.

\textit{ireadstb} Performs a GPIB serial poll (SPOLL).

\textit{itrigger} Performs an addressed GPIB group execute trigger (GET).

\textit{iclear} Performs a GPIB device clear (DCL) on the device corresponding to this session.

GPIB Device Session Interrupts

There are no device-specific interrupts for the GPIB interface.

GPIB Device Sessions and Service Requests

GPIB device sessions support Service Requests (SRQ). On the GPIB interface, when one device issues an SRQ, the library will inform all GPIB device sessions that have SRQ handlers installed. (See \texttt{ionsrq} in Chapter 10.) This is an artifact of how GPIB handles the SRQ line. The interface cannot distinguish which device requested service. Therefore, the library acts as if all devices require service. Your SRQ handler can retrieve the device’s \texttt{status byte} by using the \texttt{ireadstb} function. It is good practice to ensure that a device isn’t requesting service before leaving the SRQ handler. The easiest technique for this is to service all devices from one handler.

The data transfer functions work only when the GPIB interface is the Active Controller. Passing control to another GPIB device causes the interface to lose active control.
**GPIB Device Session Example**

The following example illustrates communicating with an GPIB device session. This example opens two GPIB communications sessions with VXI devices (through a VXI Command Module). Then a scan list is sent to a switch, and measurements are taken by the multimeter every time a switch is closed.
/* hpibdev.c
   This example program sends a scan list to a switch and
   while looping closes channels and takes measurements.*/
#include <sicl.h>
#include <stdio.h>

main()
{
    INST dvm;
    INST sw;

double res;
int i;

    /* Print message and terminate on error */
    ionerror (I_ERROR_EXIT);

    /* Open the multimeter and switch sessions */
dvm = iopen ("hpib,9,3");
sw = iopen ("hpib,9,14");
itimeout (dvm, 10000);
itimeout (sw, 10000);

    /*Set up trigger*/
    iprintf (sw, "TRIG:SOUR BUS\n");

    /*Set up scan list*/
    iprintf (sw,"SCAN (@100:103)\n");
    iprintf (sw,"INIT\n");

    for (i=1;i<=4;i++)
    {
        /* Take a measurement */
        iprintf (dvm,"MEAS:VOLT:DC?\n");

        /* Read the results */
        iscanf (dvm,"%lf", &res);

        /* Print the results */
        printf ("Result is %f\n", res);

        /*Trigger to close channel*/
        iprintf (sw, "TRIG\n");
    }

    /* Close the multimeter and switch sessions */
iclose (dvm);
iclose (sw);
}
Communicating with GPIB Interfaces

Interface sessions allow you direct low-level control of the interface. You must do all the bus maintenance for the interface. This also implies that you have considerable knowledge of the interface. Additionally, when using interface sessions, you need to use interface specific functions. The use of these functions means that the program can not be used on other interfaces and, therefore, becomes less portable.

Addressing GPIB Interfaces

To create an interface session on your GPIB system, specify either the interface symbolic name or logical unit in the `addr` parameter of the `iopen` function. The interface symbolic name and logical unit are defined during the system configuration. See the *I/O Libraries Installation and Configuration Guide* for information on these values.

The following are example GPIB interface addresses:

```
hpib An interface symbolic name.
hpib2 An interface symbolic name.
7 An interface logical unit.
```

**Note** The above examples use the default symbolic name specified during the system configuration. If you want to change the name listed above, you must also change the symbolic name or logical unit specified during the configuration. The name used in your SICL program must match the logical unit or symbolic name specified in the system configuration. Other possible interface names are GPIB, gpib, HPIB, IEEE488, etc.
Using SICL with GPIB

Communicating with GPIB Interfaces

The following example opens a interface session with the GPIB interface:

```c
INST hpin;
hpib = iopen ("hpib");
```

**SICL Function Support with GPIB Interface Sessions**

The following describes how some SICL functions are implemented for GPIB interface sessions.

- **iwrite** Sends the specified bytes directly to the interface without performing any bus addressing. The `iwrite` function always clears the ATN line before sending any bytes, thus ensuring that the GPIB interface sends the bytes as data, not command bytes.

- **iread** Reads the data directly from the interface without performing any bus addressing.

- **itrigger** Performs a GPIB group execute trigger (GET) without additional addressing. This function should be used with the `igpibsendcmd` to send an UNL followed by the device addresses. This will allow the `itrigger` function to be used to trigger multiple GPIB devices simultaneously.

  Passing the `I_TRIG_STD` value to the `ixtrig` routine also causes a broadcast GPIB group execute trigger (GET). There are no other valid values for the `ixtrig` function.

- **iclear** Performs a GPIB interface clear (pulses IFC and REN), which resets the interface.

**GPIB Interface Session Interrupts**

There are specific interface session interrupts that can be used. See `isetintr` in Chapter 10 for information on the interface session interrupts.

There are no device specific interrupts for the GPIB interface.
GPIB Interface Sessions and Service Requests

GPIB interface sessions support Service Requests (SRQ). On the GPIB interface, when one device issues an SRQ, the library will inform all GPIB interface sessions that have SRQ handlers installed. (See ionsrq in Chapter 10.) It is good practice to ensure that a device isn’t requesting service before leaving the SRQ handler. The easiest technique for this is to service all devices from one handler.

GPIB Interface Session Examples

Checking the Bus Status

The following example program is an ANSI C program that retrieves the GPIB interface bus status information and displays it for the user.
/* hpibstatus.c
   The following example retrieves and displays HPIB bus
   status information. */
#include <stdio.h>
#include <sicl.h>

main()
{
    INST id; /* session id */
    int rem; /* remote enable */
    int srq; /* service request */
    int ndac; /* not data accepted */
    int sysctlr; /* system controller */
    int actctlr; /* active controller */
    int talker; /* talker */
    int listener; /* listener */
    int addr; /* bus address */

    /* exit process if SICL error detected */
    ionerror(I_ERROR_EXIT);

    /* open HPIB interface session */
    id = iopen("hpib");
    itimeout (id, 10000);

    /* retrieve HPIB bus status */
    igpibbusstatus(id, I_GPIB_BUS_REM, &rem);
    igpibbusstatus(id, I_GPIB_BUS_SRQ, &srq);
    igpibbusstatus(id, I_GPIB_BUS_NDAC, &ndac);
    igpibbusstatus(id, I_GPIB_BUS_SYSCTLR, &sysctlr);
    igpibbusstatus(id, I_GPIB_BUS_ACTCTLR, &actctlr);
    igpibbusstatus(id, I_GPIB_BUS_TALKER, &talker);
    igpibbusstatus(id, I_GPIB_BUS_LISTENER, &listener);
    igpibbusstatus(id, I_GPIB_BUS_ADDR, &addr);

    /* display bus status */
    printf("%-5s%-5s%-5s%-5s%-5s%-5s%-5s%-5s%-5s%-5s%-5s%-5s%-5s%-5s%-5s%-5s%-5s
", "REM", "SRQ", "NDC", "SYS", "ACT", "TLK", "LTN", "ADDR");
    printf("%-2d%-5d%-5d%-5d%-5d%-5d%-5d%-5d%-5d%-5d%-5d%-5d
", rem, srq, ndac, sysctlr, actctlr, talker, listener, addr);
    return 0;
}
The following example program sets up two GPIB instruments over an interface session and has the instruments communicate with each other.

The 3 main parts of this program are as follows:

- Read the data from the scope (get_data).
- Print some statistics about the data (massage_data).
- Have the scope send the data to a printer (print_data).

/* hpibintr.c
 This program requires a 54601A digitizing oscilloscope or compatible) and a printer capable of printing in HP RASTER GRAPHICS STANDARD (e.g. thinkjet).
 This program will tell the scope to take a reading on channel 1, then send the data back to this program.
 Then some simple statistics about the data is printed.
 The program then tells the scope to send the data directly to the printer, illustrating how the controller does not have to be directly involved in an HPIB transaction.*/

#include <stdio.h>   /* used for printf() */
#include <stdlib.h>  /* used for exit() */
#include <sicl.h>    /* SICL header file */

/* defines */
define INTF_ADDR    "hpib"
define SCOPE_ADDR   INTF_ADDR ",7"

/* function prototypes */
void initialize (void);
void get_data (void);
void massage_data (void);
void print_data (void);
void cleanup (void);
void srq_hdlr (INST id);

/* global data */
float pre[10];
INST scope;
INST intf;
void main() {
    ionerror(I_ERROR_EXIT);
    scope = iopen(SCOPE_ADDR);
    intf = iopen(INTF_ADDR);
    initialize();
    get_data();
    massage_data();
    print_data();
    cleanup();
    iclose(scope);
    iclose(intf);
}

void initialize() {
    /* initialize the hpib interface and scope */
    iclear(intf);
    itimeout(scope, 5000);
    itimeout(intf, 5000);
    iclear(scope);
    igpibllo(intf);
}

void get_data() {
    short readings[5000];
    int count;

    /* setup scope to accept waveform data */
    iprintf(scope, "*RST\n");
    iprintf(scope, ":autoscale\n");

    /* setup up the waveform source */
    iprintf(scope, ":waveform:format word\n");

    /* input waveform preamble to controller */
    iprintf(scope, ":digitize channel1\n");
    iprintf(scope, ":waveform:preamble?\n");
    iscanf(scope, ",%10f", pre);

    /* command scope to send data */
    iprintf(scope, ":waveform:data?\n");
Using SICL with GPIB
Communicating with GPIB Interfaces

/* enter the data */
count = 5000;
iscanf(scope, "%#wb\n", &count, readings);
printf("received %d words\n", count);
}

void massage_data() {
  float vdiv;
  float off;
  float sdiv;
  float delay;
  char id_str[50];

  vdiv = 32 * pre[7];
  off = (128 - pre[9]) * pre[7] + pre[8];

  /* retrieve the scope’s ID string */
ipromptf(scope, "*IDN?\n", "%s", id_str);

  /* print the statistics about the data */
  printf("Oscilloscope ID: %s\n", id_str);
  printf("---------- Current settings ----------\n");
  printf("Volts/Div = %f V\n", vdiv);
  printf("Offset = %f V\n", off);
  printf("S/Div = %f S\n", sdiv);
  printf("Delay = %f S\n", delay);
}

void print_data() {
  unsigned char status;
  char cmd[5];

  /* tell the scope to SRQ on ‘operation complete’*/
iprintf(scope, "*SRE 32; *ESE 1\n");

  /* tell the scope to print */
iprintf(scope, "*:print?; *OPC\n");
Using SICL with GPIB
Communicating with GPIB Interfaces

/* tell scope to talk and printer to listen. The listen command is formed by adding 32 to the device address of the device to be a listener. The talk command is formed by adding 64 to the device address of the device to be a talker */
cmd[0] = 63;    /* 63 is unlisten */
cmd[1] = 32+1; /* printer is at address 1, make it a listener*/
cmd[2] = 64+7; /* scope is at address 7, make it a talker*/
cmd[3] = \0;   /* terminate the string */

igpibsendcmd(intf, cmd, 3);

/* set up our SRQ handler to be called when the scope finishes printing */
ionsrq(scope, srq_hdlr);

/* now, the ATN line must be set to FALSE */
igpibatnctl(intf, 0);

/* wait for SRQ before continuing program */
status = 0;
while(status == 0) {
    iwaithdlr(120000L);

    /* make sure it was the scope requesting service */
    ireadstb(scope, &status);
    status &= 64;
}

/* clear the status byte so the scope can assert SRQ again if needed. */
iprintf(scope, "*CLS\n");

void cleanup() {
    /* give local control back to the scope */
    ilocal(scope);
}

void srq_hdlr(INST id) {
    /* this handler does nothing. we will use iwaithdlr() in the code above to determine when the handler gets called. */
}
Communicating with GPIB Commanders

Commander sessions are intended for use on GPIB interfaces that are not active controller. In this mode, a computer that is not the controller is acting like a device on the GPIB bus. In a commander session, the data transfer routines work only when the GPIB interface is not active controller.

Addressing GPIB Commanders

To create a commander session on your GPIB interface, specify either the interface symbolic name or logical unit in the addr parameter followed by a comma and the string cmdr in the iopen function. The interface symbolic name and logical unit are defined during the system configuration. See the I/O Libraries Installation and Configuration Guide for information on these values.

The following are example GPIB addresses for commander sessions:

- `hpi,cmdr`  A commander session with the hpiib symbolic name.
- `hpi2,cmdr` A commander session with the hpiib2 symbolic name.
- `7,cmdr` A commander session with the interface at logical unit 7.

**Note** The above examples use the default symbolic name specified during the system configuration. If you want to change the name listed above, you must also change the symbolic name or logical unit specified during the configuration. The name used in your SICL program must match the logical unit or symbolic name specified in the system configuration. Other possible interface names are GPIB, gpib, HPIB, etc.
Using SICL with GPIB

Communicating with GPIB Commanders

The following example opens a commander session the GPIB interface:

```c
INST hpib;
hpib = iopen ("hpib,cmdr");
```
SICL Function Support with
GPIB Commander Sessions

The following describes how some SICL functions are implemented for
GPIB commander sessions.

**iwrite**

If the interface has been addressed to talk, the data
is written directly to the interface. If the interface has
not been addressed to talk, it will wait to be
addressed to talk before writing the data.

**iread**

If the interface has been addressed to listen, the
data is read directly from the interface. If the
interface has not been addressed to listen, it will wait
to be addressed to listen before reading the data.

**isetstb**

Sets the status value that will be returned on a
ireadstb call (i.e. when this device is Serial
Polled). Bit 6 of the status byte has a special
meaning. If bit 6 is set, the SRQ line will be set. If
bit 6 is clear, the SRQ line will be cleared.

There are specific commander session interrupts that can be used. See
isetintr in Chapter 10 for information on the commander session
interrupts.
Summary of GPIB Specific Functions

**Note** Using these GPIB interface specific functions means that the program cannot be used on other interfaces and, therefore, becomes less portable.

### SICL GPIB Functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>igpibatntcl</td>
<td>Sets or clears the ATN line</td>
</tr>
<tr>
<td>igpibbusaddr</td>
<td>Change bus address</td>
</tr>
<tr>
<td>igpibstatus</td>
<td>Return requested bus data</td>
</tr>
<tr>
<td>igpibgett1delay</td>
<td>Retrieves the T1 delay setting on the GPIB interface</td>
</tr>
<tr>
<td></td>
<td>Sets bus in Local Lockout Mode</td>
</tr>
<tr>
<td>igpibllo</td>
<td>Passes active control to specified address</td>
</tr>
<tr>
<td>igpibpassctl</td>
<td>Performs a parallel poll on the bus</td>
</tr>
<tr>
<td>igpibppoll</td>
<td>Configures device for PPOLL response</td>
</tr>
<tr>
<td>igpibppollconfig</td>
<td>Sets PPOLL state</td>
</tr>
<tr>
<td>igpibppollresp</td>
<td>Sets the T1 delay on the GPIB interface</td>
</tr>
<tr>
<td>igpibrenctl</td>
<td>Sends data with ATN line set</td>
</tr>
<tr>
<td>igpibsendcmd</td>
<td>Sets or clears the REN line</td>
</tr>
<tr>
<td>igpibsett1delay</td>
<td></td>
</tr>
</tbody>
</table>
Appendix H

Device Manuals

H.1 Function Generator

H.1.1 Quick Reference
Front-Panel Menu Reference

Use Recall Menu as a shortcut to recall the last command executed.

A: MODulation MENU


1: AM SHAPE
Selects the shape of the AM modulating waveform.

2: AM SOURCE
Enables or disables the internal AM modulating source.

3: FM SHAPE
Selects the shape of the FM modulating waveform.

4: BURSTCNT
Sets the number of cycles per burst (1 to 50,000 cycles).

5: BURST RATE
Sets the burst rate in Hz for an internal burst source.

6: BURST PHAS
Sets the starting phase angle of the burst (0 to 360 degrees).

7: BURST SRC
Selects an internal or external gate source for burst modulation.

8: FSK FREQ
Sets the FSK “hopp” frequency.

9: FSK RATE
Sets the internal FSK rate between the carrier and FSK frequency.

10: FSK SRC
Selects an internal or external source for the FSK rate.

B: SWP (Sweep) MENU

1: START F ⇒ 2: STOP F ⇒ 3: SWP TIME ⇒ 4: SWP MODE

1: START F
Sets the start frequency in Hz for sweeping.

2: STOP F
Sets the stop frequency in Hz for sweeping.

3: SWP TIME
Sets the repetition rate in seconds for sweeping.

4: SWP MODE
Selects linear or logarithmic sweeping.

C: EDIT MENU *


1: NEW ARB
Initiates a new arb waveform or loads the selected arb waveform.

[2: POINTS]
Sets the number of points in a new arb waveform (8 to 16,000 points).

[3: LINE EDIT]
Performs a linear interpolation between two points in the arb waveform.

[4: POINT EDIT]
Edits the individual points of the selected arb waveform.

[5: INVERT]
Inverts the selected arb waveform by changing the sign of each point.

[6: SAVE AS]
Saves the current arb waveform in non-volatile memory.

7: DELETE
Deletes the selected arb waveform from non-volatile memory.

* The commands enclosed in square brackets ([ ]) are “hidden” until you make a selection from the NEW ARB command to initiate a new edit session.

D: SYStem MENU

1: OUT TERM ⇒ 2: POWER ON ⇒ 3: COMMA ⇒ 4: REVISION

1: OUT TERM
Selects the output termination (50Ω or high impedance).

2: POWER ON
Enables or disables automatic power-up in power-down state “0”.

3: ERROR
Reroutes errors from the error queue (up to 20 errors).

4: TEST
Performs a complete self-test.

5: COMMA
Enables or disables a comma separator between digits on the display.

6: REVISION
Displays the function generator’s firmware revision codes.

E: Input / Output MENU

1: HPIB ADDR ⇒ 2: INTERFACE ⇒ 3: BAUD RATE ⇒ 4: PARITY ⇒ 5: LANGUAGE

1: HPIB ADDR
Sets the GPIB bus address (0 to 30).

2: INTERFACE
Selects the GPIB or RS-232 interface.

3: BAUD RATE
Selects the baud rate for RS-232 operation.

4: PARITY
Sets even, odd, or no parity for RS-232 operation.

5: LANGUAGE
Verifies the interface language: SCI.

F: CALibration MENU *

1: SECURED ⇒ 2: SECURED ⇒ 3: CAL COUNT ⇒ 4: MESSAGE

1: SECURED
The function generator is secured against calibration; enter code to unsecure.

[2: SECURED]
The function generator is unsecured for calibration; enter code to secure.

3: CAL COUNT
Reads the total number of times the function generator has been calibrated.

4: MESSAGE
Reads the calibration string (up to 11 characters) entered from remote.

* The commands enclosed in square brackets ([ ]) are “hidden” unless the function generator is UNSECURED for calibration.
The APPLy Commands

(see page 138 in User’s Guide)

APPLy
  :SINusoid [<frequency> [,<amplitude> [,<offset>]]]
  :SQUare [<frequency> [,<amplitude> [,<offset>]]]
  :TRIangle [<frequency> [,<amplitude> [,<offset>]]]
  :RAMP [<frequency> [,<amplitude> [,<offset>]]]
  :NOISe [<frequency>|DEF> [,<amplitude> [,<offset>]]]
  :DC [<frequency>|DEF> [,<amplitude>|DEF> [,<offset>]]]
  :USER [<frequency> [,<amplitude> [,<offset>]]]

APPLy?

Output Configuration Commands

(see page 145 in User’s Guide)

[SOURce:]
  FUNCTION:SHApe {SIN|SQU|TRI|RAMP|NOIS|DC|USER}
  FUNCTION:SHApe?

[SOURce:]
  FREQuency {<frequency>|MIN|MAX}
  FREQuency? [MIN|MAX]

[SOURce:]
  PULSe:DCYCle {<percent>|MIN|MAX}
  PULSe:DCYCle? [MIN|MAX]

[SOURce:]
  VOLTage {<amplitude>|MIN|MAX}
  VOLTage? [MIN|MAX]
  VOLTage:OFFSet {<offset>|MIN|MAX}
  VOLTage:OFFSet? [MIN|MAX]
  VOLTage:UNIT {VPP|VRMS|DBM|DEF}
  VOLTage:UNIT?

OUTPUT:LOAD {50|INF|MIN|MAX}
OUTPUT:LOAD? [MIN|MAX]

OUTPUT:SYNC {OFF|ON}
OUTPUT:SYNC?
Modulation Commands

(see page 154 in User’s Guide)

[SOURce:]
  AM:DEPTH {<depth in percent>|MIN|MAX}
  AM:DEPTH? [MIN|MAX]
  AM:INTernal:FUNCTION {SIN|SQU|TRI|RAMP|NOIS|USER}
  AM:INTernal:FUNCTION?
  AM:INTernal:FREQuency {<frequency>|MIN|MAX}
  AM:INTernal:FREQuency? [MIN|MAX]
  AM:SOURce {BOTH|EXT}
  AM:SOURce?
  AM:STATe {OFF|ON}
  AM:STATe?

[SOURce:]
  FM:DEViation {<peak deviation in Hz>|MIN|MAX}
  FM:DEViation? [MIN|MAX]
  FM:INTernal:FUNCTION {SIN|SQU|TRI|RAMP|NOIS|USER}
  FM:INTernal:FUNCTION?
  FM:INTernal:FREQuency {<frequency>|MIN|MAX}
  FM:INTernal:FREQuency? [MIN|MAX]
  FM:STATe {OFF|ON}
  FM:STATe?

[SOURce:]
  BM:NCYCles {<# cycles>|INF|MIN|MAX}
  BM:NCYCles? [MIN|MAX]
  BM:PHASe {<degrees>|MIN|MAX}
  BM:PHASe? [MIN|MAX]
  BM:INTernal:RATE {<frequency>|MIN|MAX}
  BM:INTernal:RATE? [MIN|MAX]
  BM:SOURce {INT|EXT}
  BM:SOURce?
  BM:STATe {OFF|ON}
  BM:STATe?

FSK Commands

(see page 167 in User’s Guide)

[SOURce:]
  FSKey:FREQuency {<frequency>|MIN|MAX}
  FSKey:FREQuency? [MIN|MAX]
  FSKey:INTernal:RATE {<rate in Hz>|MIN|MAX}
  FSKey:INTernal:RATE? [MIN|MAX]
  FSKey:SOURce {INT|EXT}
  FSKey:SOURce?
  FSKey:STATe {OFF|ON}
  FSKey:STATe?
Sweep Commands
(see page 170 in User’s Guide)

[SOURCE:]
FREQuency:STARt {<frequency>|MIN|MAX}
FREQuency:STARt? [MIN|MAX]
FREQuency:STOP {<frequency>|MIN|MAX}
FREQuency:STOP? [MIN|MAX]

[SOURCE:]
SWEep:SPACing {LIN|LOG}
SWEep:SPACing?
SWEep:TIME {<seconds>|MIN|MAX}
SWEep:TIME? [MIN|MAX]
SWEep:STATe {OFF|ON}
SWEep:STATe?

Arbitrary Waveform Commands
(see page 174 in User’s Guide)

[SOURCE:]
FUNCTION:USER {<arb name>|VOLATILE}
FUNCTION:USER?
FUNCTION:SHAPe USER
FUNCTION:SHAPe?

DATA VOLATILE, <value>, <value>, ...
DATA:DAC VOLATILE, {<binary block>|<value>,<value>, ...}

DATA:ATTRIBUTE:AVERage? [<arb name>]
DATA:ATTRIBUTE:CFACtor? [<arb name>]
DATA:ATTRIBUTE:POINTS? [<arb name>]
DATA:ATTRIBUTE:PTPeak? [<arb name>]

DATA:CATalog?
DATA:COPY <destination arb name> [, VOLATILE]

DATA:DELETE <arb name>
DATA:DELETE:ALL

DATA:NVOLatile:CATalog?
DATA:NVOLatile:FREE?

FORMat:BORDer {NORMAL|SWApPed} Specify Byte Order
FORMat:BORDer?
System-Related Commands

(see page 188 in User’s Guide)

DISPLAY {OFF|ON}
DISPLAY?

DISPLAY:TEXT <quoted string>
DISPLAY:TEXT?
DISPLAY:TEXT:CLEar

SYSTem:BEEPer

SYSTem:ERROR?

SYSTem:VERSION?

*IDN?

*RST

*TST?

*S保存 {0|1|2|3}  State 0 is the power-down state.

*RCL {0|1|2|3}  States 1, 2, and 3 are user-defined.

MEMory:STATE:DELete {0|1|2|3}

Triggering Commands

(see page 186 in User’s Guide)

TRIGger:SOURce {IMM|EXT|BUS}
TRIGger:SOURce?

*TRG

Status Reporting Commands

(see page 209 in User’s Guide)

SYSTem:ERROR?

*CLS

*ESE <enable value>

*ESE?

*ESR?

*OPC

*OPC?
Calibration Commands
(see page 193 in User's Guide)

CALibration?

CALibration:COUNt?

CALibration
  :SECure:CODE <new code>
  :SECure:STATe {OFF|ON},<code>
  :SECure:STATe?

CALibration:SETup <0|1|2|3| ... |84>
CALibration:SETup?

CALibration:STRing <quoted string>
CALibration:STRing?

CALibration:VALue <value>
CALibration:VALue?

SCPI Status System
(see page 201 in User's Guide)
IEEE-488.2 Common Commands
(see page 209 in User’s Guide)

*CLS
*ESE <enable value>
*ESE?
*ESR?
*IDN?
*OPC
*OPC?
*PSC {0|1}
*PSC?
*RST
*SAV {0|1|2|3}
*RCL {0|1|2|3}
*SRE <enable value>
*SRE?
*STB?
*TRG
*TST?
*WAI

RS-232 Interface Commands
(see page 200 in User’s Guide)

SYSTem:LOCal
SYSTem:REMote
SYSTem:RWLock

For RS-232 wiring and connection information, see page 195 in the User’s Guide.

Phase-Lock Commands (Option 001)
(see the 33120A Option 001 User’s and Service Guide)

PHASe:ADJust <radians>
PHASe:ADJust?
PHASe:REFerence
PHASe:UNLock:ERRor:STATe (OFF|ON)
PHASe:UNLock:ERRor:STATe?
OUTPut:TRIGger:IMMediate
OUTPut:TRIGger:STATe (OFF|ON)
OUTPut:TRIGger:STATe?
Simplified Programming Overview

Using the APPLy Command
The APPLy command provides the most straightforward method to program the function generator over the remote interface. For example, the following statement outputs a 3 Vpp sine wave at 5 kHz with a -2.5 volt offset:

"APPL:SIN 5 KHZ, 3.0 VPP, -2.5 V"

Using the Low-Level Commands
Although the APPLy commands provide the most straightforward method to program the function generator, the low-level commands give you more flexibility to change individual parameters. For example, the following statements output a 3 Vpp sine wave at 5 kHz with a -2.5 volt offset:

"FUNC:SHAP SIN"
"FREQ 5.0 KHZ"
"VOLT 3.0 VPP"
"VOLT:OFFS -2.5 V"

Reading a Query Response
Only the query commands (commands that end with “?”) will instruct the function generator to send a response message. Queries return either output values or internal instrument settings. For example, the following statements read the error queue and print the most recent error:

    dimension statement
    "SYST:ERR?"
    bus enter statement
    print statement

Selecting a Trigger Source
When burst modulation or frequency sweep is enabled, the function generator will accept an immediate internal trigger, a hardware trigger from the rear-panel Ext Trig terminal, or a software (bus) trigger. By default, the internal trigger source is selected. If you want the function generator to use the external source or a bus trigger, you must select that source. For example, the following statements output a 3-cycle burst each time the Ext Trig terminal receives the rising edge of a TTL pulse:

"BM:NCYC 3"
"TRIG:SOUR EXT"
"BM:STAT ON"
Error Messages

This is a partial listing of error messages. See chapter 5 in the User’s Guide for more information.

-102, “Syntax error” Check for blank space before or after a colon in command header, or before a comma.

-103, “Invalid separator” Check for a comma used instead of a colon, semicolon, or blank space – or a blank instead of a comma.

-108, “Parameter not allowed” Check for extra parameters in the command string.

-109, “Missing parameter” Check for omitted parameters in the command string.

-113, “Undefined header” Check the spelling of the command or you may have used an invalid command.

-221, “Settings conflict” The requested setting is in conflict with the present configuration.

-222, “Data out of range” Check for a numeric parameter value that is outside the valid range for the command.

-224, “Illegal parameter value” Check for an invalid discrete parameter choice for the command.


-350, “Too many errors” More than 20 errors have occurred.

-410, “Query INTERRUPTED” The output buffer contains data from a previous command (the previous data is not overwritten).

781, “Not enough memory to store new arb waveform” Up to four user-defined waveforms can be stored in non-volatile memory. Use DATA:DEL to delete downloaded waveforms.

783, “Arb waveform name too long” The arb name can contain up to 8 characters. The first character must be a letter (A-Z), but the remaining characters can be number (0-9) or “_”.

785, “Specified arb waveform does not exist” The arb name specified has not been downloaded into VOLATILE memory.

786, “Cannot delete a built-in arb waveform” You cannot delete the five built-in arb waveforms.

787, “Cannot delete the currently selected active arb waveform” You cannot delete the arb waveform that is currently being output.
The parameters marked with a bullet (•) are stored in **non-volatile** memory. The factory settings are shown.

### Output Configuration

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Power-On/Reset State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>Sine wave</td>
</tr>
<tr>
<td>Frequency</td>
<td>1 kHz</td>
</tr>
<tr>
<td>Amplitude (into 50 ohms)</td>
<td>100 mV peak-to-peak</td>
</tr>
<tr>
<td>Offset</td>
<td>0.00 Vdc</td>
</tr>
<tr>
<td>Output Units</td>
<td>Volts peak-to-peak</td>
</tr>
<tr>
<td>Output Termination</td>
<td>50 ohms</td>
</tr>
</tbody>
</table>

### Modulation

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Power-On/Reset State</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM Carrier Waveform</td>
<td>1 kHz Sine wave</td>
</tr>
<tr>
<td>AM Modulating Waveform</td>
<td>100 Hz Sine wave</td>
</tr>
<tr>
<td>AM Depth</td>
<td>100%</td>
</tr>
<tr>
<td>FM Carrier Waveform</td>
<td>1 kHz Sine wave</td>
</tr>
<tr>
<td>FM Modulating Waveform</td>
<td>10 Hz Sine wave</td>
</tr>
<tr>
<td>FM Peak Frequency Deviation</td>
<td>100 Hz</td>
</tr>
<tr>
<td>Burst Carrier Frequency</td>
<td>1 kHz Sine wave</td>
</tr>
<tr>
<td>Burst Count</td>
<td>1 cycle</td>
</tr>
<tr>
<td>Burst Rate</td>
<td>100 Hz</td>
</tr>
<tr>
<td>Burst Starting Phase</td>
<td>0 degrees</td>
</tr>
<tr>
<td>FSK Carrier Waveform</td>
<td>1 kHz Sine wave</td>
</tr>
<tr>
<td>FSK &quot;Hop&quot; Frequency</td>
<td>100 Hz Sine wave</td>
</tr>
<tr>
<td>FSK Rate</td>
<td>10 Hz</td>
</tr>
<tr>
<td>Modulation State</td>
<td>Off</td>
</tr>
<tr>
<td>Sweep Start / Stop Frequency</td>
<td>100 Hz / 1 kHz</td>
</tr>
<tr>
<td>Sweep Time</td>
<td>1 second</td>
</tr>
<tr>
<td>Sweep Mode</td>
<td>Linear</td>
</tr>
</tbody>
</table>

### System-Related Operations

- Power-Down Recall: Disabled
- Display Mode: On
- Comma Separators: On

### Triggering Operations

- Trigger Source: Internal

### Input/Output Configuration

- GPIB Address: 10
- Interface: GPIB (IEEE-488)
- Baud Rate: 9600 baud
- Parity: None (8 data bits)

### Calibration

- Calibration State: Secured

**NOTE:** The power-on state will be different if you have enabled the power-down storage mode. See “Power-Down Recall Mode” on page 109 for more information.
H.2 Oscilloscope
Programmer’s Guide

Publication number 54600-97032
April 2001

This guide contains programming information for the following Agilent oscilloscope models:
- 54600
- 54601
- 54602
- 54603
- 54610
- 54615
- 54616

For Safety information, Warranties, and Regulatory information, see the pages behind the Index.

© Copyright Agilent Technologies 1995-1996, 2001
All Rights Reserved

Agilent 54600-Series Oscilloscopes
Programming the Oscilloscope

When you attach an interface module to the rear of the Agilent 54600-Series Oscilloscopes, the oscilloscope becomes programmable. That is, you can hook a controller (such as a PC or workstation) to the oscilloscope, and write programs on that controller to automate oscilloscope setup and data capture. Both GPIB (also known as IEEE-488) and RS-232-C interfaces are available.

The following figure shows the basic structure of every program you will write for the oscilloscope.

1. **Initialize**
   - Program initialization defines and initializes variables, allocates memory, or tests system configuration.
   - Controller initialization ensures that the interface to the oscilloscope (either GPIB or RS-232) is properly setup and ready for data transfer.
   - Oscilloscope initialization sets the channel, trigger, timebase, and acquisition subsystems for the desired measurement.

2. **Capture**
3. **Analyze**
Capture
Once you initialize the oscilloscope, you can begin capturing data for measurement. Remember that while the oscilloscope is responding to commands from the controller, it is not performing acquisitions. Also, when you change the oscilloscope configuration, any data already captured is most likely invalid.

To collect data, you use the DIGITIZE command. This command clears the waveform buffers and starts the acquisition process. Acquisition continues until the criteria, such as number of averages, completion criteria, and number of points is satisfied. Once the criteria is satisfied, the acquisition process is stopped. The acquired data is displayed by the oscilloscope, and the captured data can be measured, stored in memory in the oscilloscope, or transferred to the controller for further analysis. Any additional commands sent while DIGITIZE is working are buffered until DIGITIZE is complete.

You could also start the oscilloscope running, then use a wait loop in your program to ensure that the oscilloscope has completed at least one acquisition before you make a measurement. This is not recommended, because the needed length of the wait loop may vary, causing your program to fail. DIGITIZE, on the other hand, ensures that data capture is complete. Also, DIGITIZE, when complete, stops the acquisition process, so that all measurements are on displayed data, not a constantly changing data set.

Analyze
After the oscilloscope has completed an acquisition, you can find out more about the data, either by using the oscilloscope measurements or by transferring the data to the controller for manipulation by your program. Built-in measurements include IEEE standard parametric measurements (such as Vpp, frequency, pulse width) or the positioning and reading of voltage and time markers.

Using the WAVEFORM commands, you can transfer the data to your controller for special analysis, if desired.
In This Book

The *Agilent 54600-Series Oscilloscopes Programmer’s Guide* is your introduction to programming the Agilent 54600-Series Oscilloscopes using an instrument controller. This book, with the online *Agilent 54600-Series Oscilloscopes Programmer’s Reference*, provides a comprehensive description of the oscilloscope’s programmatic interface. The *Programmer’s Reference* is supplied as a Microsoft Windows Help file on a 3.5" diskette.

To program the Agilent 54600-Series Oscilloscope, you need an interface module, such as the Agilent 54650A or 54651A. You also need an instrument controller that supports either the IEEE-488 or RS-232-C interface standards, and a programming language capable of communicating with these interfaces. You can also use the Agilent 54655A/56A Test Automation Module and the Agilent 54658A/59B Measurement/Storage Module.

**Chapter 1** gives a general overview of oscilloscope programming.

**Chapter 2** shows a simple program, explains its operation, and discusses considerations for data types.

**Chapter 3** discusses the general considerations for programming the instrument over an GPIB interface.

**Chapter 4** discusses the general considerations for programming the instrument over an RS-232-C interface.

**Chapter 5** describes conventions used in representing syntax of commands throughout this book and in the online *Agilent 54600-Series Oscilloscopes Programmer’s Reference*, and gives an overview of the command set.

**Chapter 6** discusses the oscilloscope status registers and how to use them in your programs.

**Chapter 7** tells how to install the *Agilent 54600-Series Oscilloscopes Programmer’s Reference* online help file in Microsoft Windows, and explains help file navigation.

**Chapter 8** lists all the commands and queries available for programming the oscilloscope.
For information on oscilloscope operation, see the *Agilent 54600-Series Oscilloscopes User and Service Guide*. For information on interface configuration, see the documentation for the oscilloscope and the interface card used in your controller (for example, the 82341C interface for IBM PC-compatible computers).
Contents

1 Introduction to Programming
   Talking to the Instrument  1–3
   Program Message Syntax  1–4
   Combining Commands from the Same Subsystem  1–7
   Duplicate Mnemonics  1–7
   Query Command  1–8
   Program Header Options  1–9
   Program Data Syntax Rules  1–10
   Program Message Terminator  1–12
   Selecting Multiple Subsystems  1–12

2 Programming Getting Started
   Initialization  2–3
   Autoscale  2–4
   Setting Up the Instrument  2–4
   Example Program  2–5
   Using the DlGitize Command  2–6
   Receiving Information from the Instrument  2–8
   String Variables  2–9
   Numeric Variables  2–10
   Definite-Length Block Response Data  2–11
   Multiple Queries  2–12
   Instrument Status  2–12

3 Programming over GPIB
   Interface Capabilities  3–3
   Command and data concepts  3–3
   Addressing  3–4
   Communicating over the bus  3–5
   Lockout  3–6
   Bus Commands  3–6

4 Programming over RS-232-C
   Interface Operation  4–3
   Cables  4–3
Minimum three-wire interface with software protocol  4–4
Extended interface with hardware handshake  4–5
Configuring the Interface  4–7
Interface Capabilities  4–8
Communicating over the RS-232-C bus  4–9
Lockout Command  4–10

5 Programming and Documentation Conventions
Command Set Organization  5–3
The Command Tree  5–6
Truncation Rules  5–10
Infinity Representation  5–11
Sequential and Overlapped Commands  5–11
Response Generation  5–11
Notation Conventions and Definitions  5–12
Program Examples  5–13

6 Status Reporting
Serial Poll  6–6

7 Installing and Using the Programmer’s Reference
To install the help file under Microsoft Windows  7–3
To get updated help and program files via the Internet  7–4
To start the help file  7–5
To navigate through the help file  7–6

8 Programmer’s Quick Reference
Conventions  8–3
Suffix Multipliers  8–3
Commands and Queries  8–4

Index
Introduction to Programming

Chapters 1 and 2 introduce the basics for remote programming of an oscilloscope. The programming instructions in this manual conform to the IEEE 488.2 Standard Digital Interface for Programmable Instrumentation. The programming instructions provide the means of remote control.

To program the Agilent 54600-series oscilloscope you must add either an GPIB (for example, Agilent 54650A) or RS-232-C (for example, Agilent 54651A) interface to the rear panel.

You can perform the following basic operations with a controller and an oscilloscope:

- Set up the instrument.
- Make measurements.
- Get data (waveform, measurements, configuration) from the oscilloscope.
- Send information (pixel image, configurations) to the oscilloscope.

Other tasks are accomplished by combining these basic functions.

Languages for Program Examples

The programming examples for individual commands in this manual are written in Agilent BASIC, C, or SICL C.
Talking to the Instrument

Computers acting as controllers communicate with the instrument by sending and receiving messages over a remote interface. Instructions for programming normally appear as ASCII character strings embedded inside the output statements of a “host” language available on your controller. The input statements of the host language are used to read in responses from the oscilloscope.

For example, Agilent BASIC uses the OUTPUT statement for sending commands and queries. After a query is sent, the response is usually read in using the ENTER statement.

Messages are placed on the bus using an output command and passing the device address, program message, and terminator. Passing the device address ensures that the program message is sent to the correct interface and instrument.

The following Agilent BASIC statement sends a command which sets the bandwidth limit of channel 1 on:

```
OUTPUT < device address > ;" :CHANNEL1:BWLIMIT ON" <terminator>
```

The < device address > represents the address of the device being programmed. Each of the other parts of the above statement are explained in the following pages.
Program Message Syntax

To program the instrument remotely, you must understand the command format and structure expected by the instrument. The IEEE 488.2 syntax rules govern how individual elements such as headers, separators, program data, and terminators may be grouped together to form complete instructions. Syntax definitions are also given to show how query responses are formatted. The figure below shows the main syntactical parts of a typical program statement.

Program Message Syntax

Output Command
The output command is entirely dependent on the programming language. Throughout this manual, Agilent BASIC is used in most examples of individual commands. If you are using other languages, you will need to find the equivalents of Agilent BASIC commands like OUTPUT, ENTER, and CLEAR in order to convert the examples. The instructions listed in this manual are always shown between quotation marks in the example programs.

Device Address
The location where the device address must be specified is also dependent on the programming language you are using. In some languages, this may be specified outside the output command. In Agilent BASIC, this is always specified after the keyword OUTPUT. The examples in this manual assume the oscilloscope is at device address 707. When writing programs, the address varies according to how the bus is configured.
Instructions

Instructions (both commands and queries) normally appear as a string embedded in a statement of your host language, such as BASIC, Pascal, or C. The only time a parameter is not meant to be expressed as a string is when the instruction’s syntax definition specifies <block data>, such as learnstring. There are only a few instructions which use block data.

Instructions are composed of two main parts:

- The header, which specifies the command or query to be sent.
- The program data, which provide additional information needed to clarify the meaning of the instruction.

Instruction Header

The instruction header is one or more mnemonics separated by colons (:) that represent the operation to be performed by the instrument. The command tree in chapter 5 illustrates how all the mnemonics can be joined together to form a complete header (see chapter 5, “Programming and Documentation Conventions”).

The example in figure 1 is a command. Queries are indicated by adding a question mark (?) to the end of the header. Many instructions can be used as either commands or queries, depending on whether or not you have included the question mark. The command and query forms of an instruction usually have different program data. Many queries do not use any program data.

White Space (Separator)

White space is used to separate the instruction header from the program data. If the instruction does not require any program data parameters, you do not need to include any white space. In this manual, white space is defined as one or more spaces. ASCII defines a space to be character 32 (in decimal).

Program Data

Program data are used to clarify the meaning of the command or query. They provide necessary information, such as whether a function should be on or off, or which waveform is to be displayed. Each instruction’s syntax definition shows the program data, as well as the values they accept. The section “Program Data Syntax Rules” in this chapter has all of the general rules about acceptable values.

When there is more than one data parameter, they are separated by commas (,). Spaces can be added around the commas to improve readability.
Header Types
There are three types of headers:

- Simple Command headers.
- Compound Command headers.
- Common Command headers.

**Simple Command Header**  Simple command headers contain a single mnemonic. AUTOSCALE and DIGITIZE are examples of simple command headers typically used in this instrument. The syntax is:

```plaintext
<program mnemonic><terminator>
```

Simple command headers must occur at the beginning of a program message; if not, they must be preceded by a colon.

When program data must be included with the simple command header (for example, `:DIGITIZE CHAN1`), white space is added to separate the data from the header. The syntax is:

```plaintext
<program mnemonic><separator><program data><terminator>
```

**Compound Command Header**  Compound command headers are a combination of two program mnemonics. The first mnemonic selects the subsystem, and the second mnemonic selects the function within that subsystem. The mnemonics within the compound message are separated by colons. For example:

To execute a single function within a subsystem:

```plaintext
:<subsystem>:<function><separator><program data><terminator>
```

(For example `:CHANNEL1:BWLIMIT ON`)

**Common Command Header**  Common command headers control IEEE 488.2 functions within the instrument (such as clear status). Their syntax is:

```plaintext
:*<command header><terminator>
```

No space or separator is allowed between the asterisk (*) and the command header. *CLS is an example of a common command header.
Combining Commands from the Same Subsystem

To execute more than one function within the same subsystem a semi-colon (;) is used to separate the functions:

```
:subsystem:operation<separator><parameter><terminator>
```

(For example :CHANNEL1:COUPLING DC;BWLIMIT ON)

Duplicate Mnemonics

Identical function mnemonics can be used for more than one subsystem. For example, the function mnemonic RANGE may be used to change the vertical range or to change the horizontal range:

```
:CHANNEL1:RANGE .4
```

sets the vertical range of channel 1 to 0.4 volts full scale.

```
:TIMEBASE:RANGE 1
```

sets the horizontal time base to 1 second full scale.

CHANNEL1 and TIMEBASE are subsystem selectors and determine which range is being modified.
Query Command

Command headers immediately followed by a question mark (?) are queries. After receiving a query, the instrument interrogates the requested function and places the answer in its output queue. The answer remains in the output queue until it is read or another command is issued. When read, the answer is transmitted across the bus to the designated listener (typically a controller). For example, the query :TIMEBASE:RANGE? places the current time base setting in the output queue. In Agilent BASIC, the controller input statement:

```
ENTER < device address > ;Range
```

passes the value across the bus to the controller and places it in the variable Range.

Query commands are used to find out how the instrument is currently configured. They are also used to get results of measurements made by the instrument. For example, the command :MEASURE:RISETIME? instructs the instrument to measure the rise time of your waveform and places the result in the output queue.

The output queue must be read before the next program message is sent. For example, when you send the query :MEASURE:RISETIME? you must follow that query with an input statement. In Agilent BASIC, this is usually done with an ENTER statement immediately followed by a variable name. This statement reads the result of the query and places the result in a specified variable.

---

**Read the Query Result First**

Sending another command or query before reading the result of a query causes the output buffer to be cleared and the current response to be lost. This also generates a query interrupted error in the error queue.
Program Header Options

Program headers can be sent using any combination of uppercase or lowercase ASCII characters. Instrument responses, however, are always returned in uppercase.

Program command and query headers may be sent in either long form (complete spelling), short form (abbreviated spelling), or any combination of long form and short form.

TIMEBASE:DELAY 1US  - long form
TIM:DEL 1US  - short form

Programs written in long form are easily read and are almost self-documenting. The short form syntax conserves the amount of controller memory needed for program storage and reduces the amount of I/O activity.

Command Syntax Programming Rules

The rules for the short form syntax are shown in chapter 5, “Programming and Documentation Conventions.”
Introduction to Programming

Program Data Syntax Rules

Program data is used to convey a variety of types of parameter information related to the command header. At least one space must separate the command header or query header from the program data.

<program mnemonic><separator><data><terminator>

When a program mnemonic or query has multiple program data a comma separates sequential program data.

<program mnemonic><separator><data>,<data><terminator>

For example, :MEASURE:TVOLT 1.0V,2 has two program data: 1.0V and 2.

There are two main types of program data which are used in commands: character and numeric program data.

Character Program Data

Character program data is used to convey parameter information as alpha or alphanumeric strings. For example, the :TIMEBASE:MODE command can be set to normal, delayed, XY, or ROLL. The character program data in this case may be NORMAL, DELAYED, XY, or ROLL. The command :TIMEBASE:MODE DELAYED sets the time base mode to delayed.

The available mnemonics for character program data are always included with the instruction’s syntax definition. When sending commands, either the long form or short form (if one exists) may be used. Upper-case and lower-case letters may be mixed freely. When receiving query responses, upper-case letters are used exclusively.

Numeric Program Data

Some command headers require program data to be expressed numerically. For example, :TIMEBASE:RANGE requires the desired full scale range to be expressed numerically.

For numeric program data, you have the option of using exponential notation or using suffix multipliers to indicate the numeric value. The following numbers are all equal:

\[ 28 = 0.28E2 = 280e-1 = 28000m = 0.028K = 28e-3K. \]

When a syntax definition specifies that a number is an integer, that means that the number should be whole. Any fractional part would be ignored, truncating the number. Numeric data parameters which accept fractional values are called real numbers.
All numbers are expected to be strings of ASCII characters. Thus, when sending the number 9, you would send a byte representing the ASCII code for the character “9” (which is 57). A three-digit number like 102 would take up three bytes (ASCII codes 49, 48, and 50). This is taken care of automatically when you include the entire instruction in a string.

**Embedded Strings**

Embedded strings contain groups of alphanumeric characters which are treated as a unit of data by the oscilloscope. For example, the line of text written to the advisory line of the instrument with the :SYSTEM:DSP command:

```
:SYSTEM:DSP"This is a message."
```

Embedded strings may be delimited with either single (’) or double (") quotes. These strings are case-sensitive and spaces act as legal characters just like any other character.
Program Message Terminator

The program instructions within a data message are executed after the program message terminator is received. The terminator may be either an NL (New Line) character, an EOI (End-Or-Identify) asserted in the GPIB interface, or a combination of the two. Asserting the EOI sets the EOI control line low on the last byte of the data message. The NL character is an ASCII linefeed (decimal 10).

<table>
<thead>
<tr>
<th>New Line Terminator Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>The NL (New Line) terminator has the same function as an EOS (End Of String) and EOT (End Of Text) terminator.</td>
</tr>
</tbody>
</table>

Selecting Multiple Subsystems

You can send multiple program commands and program queries for different subsystems on the same line by separating each command with a semicolon. The colon following the semicolon enables you to enter a new subsystem. For example:

```
<program mnemonic><data>;:<program mnemonic><data><terminator>
:CHANNEL1:RANGE 0.4;:TIMEBASE:RANGE 1
```

<table>
<thead>
<tr>
<th>Combining Compound and Simple Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple commands may be any combination of compound and simple commands.</td>
</tr>
</tbody>
</table>
Programming Getting Started

This chapter explains how to set up the instrument, how to retrieve setup information and measurement results, how to digitize a waveform, and how to pass data to the controller.

<table>
<thead>
<tr>
<th>Languages for Programming Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>The programming examples in this guide are written in Agilent BASIC, C, or SICL C.</td>
</tr>
</tbody>
</table>
Initialization

To make sure the bus and all appropriate interfaces are in a known state, begin every program with an initialization statement. Agilent BASIC provides a CLEAR command which clears the interface buffer:

```
CLEAR 707  ! initializes the interface of the instrument
```

When you are using GPIB, CLEAR also resets the oscilloscope’s parser. The parser is the program which reads in the instructions which you send it.

After clearing the interface, initialize the instrument to a preset state:

```
OUTPUT 707;"*RST"  ! initializes the instrument to a preset state.
```

### Information for Initializing the Instrument

The actual commands and syntax for initializing the instrument are discussed in the common commands section of the online *Agilent 54600-Series Oscilloscopes Programmer’s Reference*.

Refer to your controller manual and programming language reference manual for information on initializing the interface.
Autoscale

The AUTOSCALE feature performs a very useful function on unknown waveforms by setting up the vertical channel, time base, and trigger level of the instrument.

The syntax for the autoscale function is:

:AUTOSCALE<terminator>

Setting Up the Instrument

A typical oscilloscope setup would set the vertical range and offset voltage, the horizontal range, delay time, delay reference, trigger mode, trigger level, and slope. A typical example of the commands sent to the oscilloscope are:

:CHANNEL1:PROBE X10;RANGE 16;OFFSET 1.00<terminator>
:TIMEBASE:MODE NORMAL;RANGE 1E-3;DELAY 100E-6<terminator>

This example sets the time base at 1 ms full-scale (100 μs/div) with delay of 100 μs. Vertical is set to 16V full-scale (2 V/div) with center of screen at 1V and probe attenuation set to 10.
Example Program

This program demonstrates the basic command structure used to program the oscilloscope.

10 CLEAR 707 ! Initialize instrument interface
20 OUTPUT 707;"*RST" ! Initialize inst to preset state
30 OUTPUT 707;":TIMEBASE:RANGE 5E-4" ! Time base to 50 us/div
40 OUTPUT 707;":TIMEBASE:DELAY 0" ! Delay to zero
50 OUTPUT 707;":TIMEBASE:REFERENCE CENTER" ! Display reference at center
60 OUTPUT 707;":CHANNEL1:PROBE X10" ! Probe attenuation to 10:1
70 OUTPUT 707;":CHANNEL1:RANGE 1.6" ! Vertical range to 1.6 V full scale
80 OUTPUT 707;":CHANNEL1:OFFSET -.4" ! Offset to -0.4
90 OUTPUT 707;":CHANNEL1:COUPLING DC" ! Coupling to DC
100 OUTPUT 707;":TRIGGER:MODE NORMAL" ! Normal triggering
110 OUTPUT 707;":TRIGGER:LEVEL -.4" ! Trigger level to -0.4
120 OUTPUT 707;":TRIGGER:SLOPE POSITIVE" ! Trigger on positive slope
130 OUTPUT 707;":ACQUIRE:TYPE NORMAL" ! Normal acquisition
140 OUTPUT 707;":DISPLAY:GRID OFF" ! Grid off
150 END

- Line 10 initializes the instrument interface to a known state.
- Line 20 initializes the instrument to a preset state.
- Lines 30 through 50 set the time base mode to normal with the horizontal time at 50 μs/div with 0 s of delay referenced at the center of the graticule.
- Lines 60 through 90 set the vertical range to 1.6 volts full scale with center screen at -0.4 volts with 10:1 probe attenuation and DC coupling.
- Lines 100 through 120 configure the instrument to trigger at -0.4 volts with normal triggering.
- Line 130 configures the instrument for normal acquisition.
- Line 140 turns the grid off.
Using the DIGitize Command

The DIGitize command is a macro that captures data satisfying the specifications set up by the ACQuire subsystem. When the digitize process is complete, the acquisition is stopped. The captured data can then be measured by the instrument or transferred to the controller for further analysis. The captured data consists of two parts: the waveform data record and the preamble.

Ensure New Data is Collected

After changing the oscilloscope configuration, the waveform buffers are cleared. Before doing a measurement, the DIGitize command should be sent to the oscilloscope to ensure new data has been collected.

When you send the DIGitize command to the oscilloscope, the specified channel signal is digitized with the current ACQuire parameters. To obtain waveform data, you must specify the WAVEFORM parameters for the waveform data prior to sending the :WAVEFORM:DATA? query.

Set :TIMebase:MODE to NORMal when Using :DIGitize

:TIMebase:MODE must be set to NORMal to perform a :DIGitize or to perform any WAVEform subsystem query. A "Settings conflict" error message will be returned if these commands are executed when MODE is set to ROLL, XY, or DELayed. Sending the *RST (reset) command will also set the time base mode to normal.

The number of data points comprising a waveform varies according to the number requested in the ACQuire subsystem. The ACQuire subsystem determines the number of data points, type of acquisition, and number of averages used by the DIGitize command. This allows you to specify exactly what the digitized information contains.
The following program example shows a typical setup:
OUTPUT 707;":ACQUIRE:TYPE AVERAGE"<terminator>
OUTPUT 707;":ACQUIRE:COMPLETE 100"<terminator>
OUTPUT 707;":WAVEFORM:SOURCE CHANNEL1"<terminator>
OUTPUT 707;":WAVEFORM:FORMAT BYTE"<terminator>
OUTPUT 707;":ACQUIRE:COUNT 8"<terminator>
OUTPUT 707;":WAVEFORM:POINTS 500"<terminator>
OUTPUT 707;":DIGITIZE CHANNEL1"<terminator>
OUTPUT 707;":WAVEFORM:DATA?"<terminator>

This setup places the instrument into the averaged mode with eight averages. This means that when the DIGitize command is received, the command will execute until the signal has been averaged at least eight times.

After receiving the :WAVEFORM:DATA? query, the instrument will start passing the waveform information when addressed to talk.

Digitized waveforms are passed from the instrument to the controller by sending a numerical representation of each digitized point. The format of the numerical representation is controlled with the :WAVEFORM:FORMAT command and may be selected as BYTE, WORD, or ASCII.

The easiest method of transferring a digitized waveform depends on data structures, formatting available and I/O capabilities. You must scale the integers to determine the voltage value of each point. These integers are passed starting with the leftmost point on the instrument’s display. For more information, see the waveform subsystem commands and corresponding program code examples in the online *Agilent 54600-Series Oscilloscopes Programmer’s Reference*.

---

**Aborting a Digitize Operation Over GPIB**

When using GPIB, a digitize operation may be aborted by sending a Device Clear over the bus (CLEAR 707).
Receiving Information from the Instrument

After receiving a query (command header followed by a question mark), the instrument interrogates the requested function and places the answer in its output queue. The answer remains in the output queue until it is read or another command is issued. When read, the answer is transmitted across the interface to the designated listener (typically a controller). The input statement for receiving a response message from an instrument’s output queue typically has two parameters; the device address, and a format specification for handling the response message. For example, to read the result of the query command :CHANNEL1:COUPLING?, you would execute the Agilent BASIC statement:

```
ENTER <device address> ;Setting$
```

where <device address> represents the address of your device. This would enter the current setting for the channel one coupling in the string variable Setting$.

All results for queries sent in a program message must be read before another program message is sent. For example, when you send the query :MEASURE:RISETIME?, you must follow that query with an input statement. In Agilent BASIC, this is usually done with an ENTER statement.

Sending another command before reading the result of the query causes the output buffer to be cleared and the current response to be lost. This also causes an error to be placed in the error queue.

Executing an input statement before sending a query causes the controller to wait indefinitely.

The format specification for handling response messages is dependent on both the controller and the programming language.
String Variables

The output of the instrument may be numeric or character data depending on what is queried. Refer to the specific commands for the formats and types of data returned from queries.

Express String Variables Using Exact Syntax

In Agilent BASIC, string variables are case sensitive and must be expressed exactly the same each time they are used.

Address Varies According to Configuration

For the example programs in the help file, assume that the device being programmed is at device address 707. The actual address varies according to how you have configured the bus for your own application.

The following example shows the data being returned to a string variable:

```
10 DIM Rang$[30]
20 OUTPUT 707;":CHANNEL1:RANGE?"
30 ENTER 707;Rang$
40 PRINT Rang$
50 END
```

After running this program, the controller displays:

`+8.00000E-01`
Numeric Variables

The following example shows the data being returned to a numeric variable:

10 OUTPUT 707;":CHANNEL1:RANGE?"
20 ENTER 707;Rang
30 PRINT Rang
40 END

After running this program, the controller displays:

.8
Definite-Length Block Response Data

Definite-length block response data allows any type of device-dependent data to be transmitted over the system interface as a series of 8-bit binary data bytes. This is particularly useful for sending large quantities of data or 8-bit extended ASCII codes. The syntax is a pound sign ( # ) followed by a non-zero digit representing the number of digits in the decimal integer. After the non-zero digit is the decimal integer that states the number of 8-bit data bytes being sent. This is followed by the actual data.

For example, for transmitting 4000 bytes of data, the syntax would be:

```
#00004000<4000 bytes of data><terminator>
```

The “8” states the number of digits that follow, and “00004000” states the number of bytes to be transmitted.
Multiple Queries

You can send multiple queries to the instrument within a single program message, but you must also read them back within a single program message. This can be accomplished by either reading them back into a string variable or into multiple numeric variables. For example, you could read the result of the query :TIMEBASE:RANGE?;DELAY? into the string variable Results$ with the command:

```
ENTER 707;Results$
```

When you read the result of multiple queries into string variables, each response is separated by a semicolon. For example, the response of the query :TIMEBASE:RANGE?;DELAY? would be:

```
<range_value>; <delay_value>
```

Use the following program message to read the query :TIMEBASE:RANGE?;DELAY? into multiple numeric variables:

```
ENTER 707;Result1,Result2
```

Instrument Status

Status registers track the current status of the instrument. By checking the instrument status, you can find out whether an operation has been completed, whether the instrument is receiving triggers, and more. Chapter 6, “Status Reporting” explains how to check the status of the instrument.
Programming over GPIB
Programming over GPIB

This section describes the GPIB interface functions and some general concepts. In general, these functions are defined by IEEE 488.1. They deal with general interface management issues, as well as messages which can be sent over the interface as interface commands.

For more information on connecting the controller to the oscilloscope, see the documentation for the GPIB interface card you are using.
Interface Capabilities

The interface capabilities of the oscilloscope, as defined by IEEE 488.1, are SH1, AH1, T5, L4, SR1, RL1, PP0, DC1, DT1, C0, and E2.

Command and data concepts

The interface has two modes of operation:

- command mode
- data mode

The bus is in the command mode when the ATN line is true. The command mode is used to send talk and listen addresses and various bus commands, such as a group execute trigger (GET).

The bus is in the data mode when the ATN line is false. The data mode is used to convey device-dependent messages across the bus. The device-dependent messages include all of the instrument commands and responses.
Addressing

Set the instrument address by using the front panel controls on the oscilloscope after the GPIB interface has been installed on the rear panel of the oscilloscope.

1. Press **Print/Utility**, then press the **I/O Menu** softkey.

2. Press the **Inst Addr** softkey to select the instrument address. Increment the address by successively pressing the **Inst Addr** softkey. The address can also be incremented or decremented by turning the knob closest to the **Cursors** key.

- Each device on the GPIB resides at a particular address, ranging from 0 to 30.
- The active controller specifies which devices talk and which listen.
- An instrument may be talk addressed, listen addressed, or unaddressed by the controller.

If the controller addresses the instrument to talk, the instrument remains configured to talk until it receives an interface clear message (IFC), another instrument’s talk address (OTA), its own listen address (MLA), or a universal untalk command (UNT).

If the controller addresses the instrument to listen, the instrument remains configured to listen until it receives an interface clear message (IFC), its own talk address (MTA), or a universal unlisten command (UNL).
Communicating over the bus

Since GPIB can address multiple devices through the same interface card, the device address passed with the program message must include not only the correct interface select code, but also the correct instrument address.

**Interface Select Code (Selects Interface)**
Each interface card has a unique interface select code. This code is used by the controller to direct commands and communications to the proper interface. The default is typically “7” for GPIB controllers.

**Instrument Address (Selects Instrument)**
Each instrument on an GPIB must have a unique instrument address between decimal 0 and 30. The device address passed with the program message must include not only the correct instrument address, but also the correct interface select code.

DEVICE ADDRESS = (Interface Select Code * 100) + (Instrument Address)

For example, if the instrument address for the oscilloscope is 4 and the interface select code is 7, when the program message is passed, the routine performs its function on the instrument at device address 704.

For the oscilloscope, the instrument address is typically set to 0.

**Oscilloscope Device Address**
The examples in this manual and in the online Agilent 54600-Series Oscilloscopes Programmer’s Reference assume the oscilloscope is at device address 707.

See the documentation for your GPIB interface card for more information on select codes and addresses.
Lockout

You can use the SYSTem:LOCK ON command to disable front-panel control while a program is running. By default, the instrument accepts and executes bus commands, and the front panel is entirely active.

<table>
<thead>
<tr>
<th>Restore Front-Panel Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycling power also restores front panel control.</td>
</tr>
</tbody>
</table>

With GPIB, the instrument is placed in the lockout mode by sending the local lockout command (LLO). The instrument can be returned to local by sending the go-to-local command (GTL) to the instrument.

Bus Commands

The following commands are IEEE 488.1 bus commands (ATN true). IEEE 488.2 defines many of the actions which are taken when these commands are received by the instrument.

**Device Clear**
The device clear (DCL) or selected device clear (SDC) commands clear the input and output buffers, reset the parser, and clear any pending commands. If either of these commands is sent during a digitize operation, the digitize operation is aborted.

**Interface Clear (IFC)**
The interface clear (IFC) command halts all bus activity. This includes unaddressing all listeners and the talker, disabling serial poll on all devices, and returning control to the system controller.
Introduction

The Programmer's Quick Reference provides the commands and queries with their corresponding arguments and returned formats for the Agilent 54600-Series Oscilloscopes. The arguments for each command list the minimum argument required. The part of the command or query listed in uppercase letters refers to the short form of that command or query. The long form is the combination of the uppercase and lowercase letters. Any optional parameters are listed at the end of each parameter listing.

This quick reference lists commands for the following Agilent oscilloscope models:

54600
54601
54602
54603
54610
54615
54616
Conventions

The following conventions used in this guide include:

< > Indicates that words or characters enclosed in angular brackets symbolize a program code parameter or an GPIB command.

::= "is defined as." <A>::= <B> indicates that <A> can be replaced by <B> in any statement containing <A>.

| "or" Indicates a choice of one element from a list. For example, <A> | <B> indicates <A> or <B> but not both.

... Indicates that the element preceding the ellipses may be repeated one or more times.

[] Indicates that the bracketed items are optional.

{} Indicates that when items are enclosed by braces, one, and only one of the elements may be selected.

{N,...,P} Indicates selection of one integer between N and P inclusive.

Suffix Multipliers

The following suffix multipliers are available for arguments.

| EX ::= 1E18 | M ::= 1E-3 |
| PE ::= 1E15 | U ::= 1E-6 |
| T ::= 1E12  | N ::= 1E-9 |
| G ::= 1E9   | P ::= 1E-12 |
| MA ::= 1E6  | F ::= 1E-15 |
| K ::= 1E3   | A ::= 1E-18 |

For more information regarding specific commands or queries, please refer to the online Agilent 54600-Series Oscilloscopes Programmer’s Reference.
Commands and Queries

The following tables facilitate easy access to each command and query for the Agilent 54600-Series Oscilloscopes. The commands and queries are divided into separate categories with each entry alphabetized.

The arguments for each command list the minimum argument required. The part of the command or query listed in uppercase letters refers to the short form of that command or query. The long form is the combination of the uppercase and lowercase letters.

These commands also show specific information about how the command operates on a particular oscilloscope model. For additional information, refer to the online Agilent 54600-Series Oscilloscopes Programmer’s Reference.
<table>
<thead>
<tr>
<th>Command</th>
<th>Query</th>
<th>Options and Query Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>:ACQuire:COMPLETE &lt;complete_argument&gt;</td>
<td>:ACQuire:COMPLETE? &lt;complete_argument&gt;</td>
<td>:= 0 to 100; an integer in NR1 format</td>
</tr>
<tr>
<td>:ACQuire:COUNt &lt;count_argument&gt;</td>
<td>:ACQuire:COUNt? &lt;count_argument&gt;</td>
<td>:= 8, 64, or 256; an integer in NR1 format</td>
</tr>
<tr>
<td>n/a</td>
<td>:ACQuire:POINts?</td>
<td>For all models except 54615/16: 1 to 4000; an integer in NR1 format. For the 54615/16: 1 to 5000; an integer in NR1 format.</td>
</tr>
<tr>
<td>n/a</td>
<td>:ACQuire:SETup?</td>
<td>:ACQuire:TYPE(NORM</td>
</tr>
<tr>
<td>:ACQuire:TYPE &lt;acq_type&gt;</td>
<td>:ACQuire:TYPE? &lt;acq_type&gt;</td>
<td>:= {NORMal</td>
</tr>
<tr>
<td>:ASTore</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>:AUToscale</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>:BLANK &lt;display&gt;</td>
<td>n/a</td>
<td>&lt;display&gt; := {CHAN &lt;n&gt;</td>
</tr>
<tr>
<td>:CHANnel&lt;n&gt;:BWLimit (ON</td>
<td>OFF)</td>
<td>:CHANnel&lt;n&gt;:BWLimit? (ON</td>
</tr>
<tr>
<td>:CHANnel&lt;n&gt;:COUPling (AC</td>
<td>DC</td>
<td>GND)</td>
</tr>
<tr>
<td>:CHANnel&lt;n&gt;:INPut (FIFTy</td>
<td>ONEMeg)</td>
<td>:CHANnel&lt;n&gt;:INPut? (FIFTy</td>
</tr>
<tr>
<td>:CHANnel&lt;n&gt;:INVert (ON</td>
<td>OFF)</td>
<td>:CHANnel&lt;n&gt;:INVert? (ON</td>
</tr>
<tr>
<td>:CHANnel&lt;n&gt;:MATH (OFF</td>
<td>PLUS</td>
<td>SUBTract)</td>
</tr>
<tr>
<td>:CHANnel&lt;n&gt;:OFFSet &lt;offset_argument&gt;</td>
<td>:CHANnel&lt;n&gt;:OFFSet? &lt;offset_argument&gt;</td>
<td>:= offset value in volts in &lt;NR3&gt; format. &lt;n&gt; := 1 or 2; an integer in NR1 format for 54600/03/10/15/16 1, 2, 3 or 4; an integer in NR1 format for the 54601/02</td>
</tr>
<tr>
<td>:CHANnel&lt;n&gt;:PMODE (AUTO</td>
<td>MANual)</td>
<td>:CHANnel&lt;n&gt;:PMODE? (AUTO</td>
</tr>
<tr>
<td>Command</td>
<td>Query</td>
<td>Options and Query Returns</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------</td>
<td>-------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>:CHANnel&lt;n&gt;:PROBe &lt;attenuation&gt;</td>
<td>:CHANnel&lt;n&gt;:PROBe? &lt;attenuation&gt; := X1, X10, X100 for 51600/01/02/03 X1, X10, X20, X100 for the 54610/15/16 &lt;n&gt; ::= 1 or 2; an integer in NR1 format for 54600/03/10/15/16 1, 2, 3 or 4; an integer in NR1 format for the 54601/02</td>
<td></td>
</tr>
<tr>
<td>:CHANnel&lt;n&gt;:PROTect (OFF</td>
<td>ON)</td>
<td>:CHANnel&lt;n&gt;:PROTect? (OFF</td>
</tr>
<tr>
<td>:CHANnel&lt;n&gt;:RANGE &lt;range_argument&gt;</td>
<td>:CHANnel&lt;n&gt;:RANGE? &lt;range_argument&gt; ::= Full-scale range value for channels 1 or 2 in NR3 format, and [LOW</td>
<td>HIGH] for channels 3 or 4.</td>
</tr>
<tr>
<td>n/a</td>
<td>:CHANnel&lt;n&gt;:SETup?</td>
<td>For 54600/01/02/03 channels 1 and 2: CHANnel&lt;n&gt;:RANGE &lt;range&gt;; OFFSet &lt;offset&gt;; COUPling (AC</td>
</tr>
<tr>
<td>:CHANnel2:SKEW &lt;skew_argument&gt;</td>
<td>:CHANnel2:SKEW? &lt;skew_argument&gt; ::= the skew value in seconds in &lt;NR3&gt; format</td>
<td></td>
</tr>
<tr>
<td>:CHANnel&lt;n&gt;:VERNier (ON</td>
<td>OFF)</td>
<td>:CHANnel&lt;n&gt;:VERNier? (ON</td>
</tr>
<tr>
<td>*CLS</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>:DITHer (ON</td>
<td>OFF)</td>
<td>:DITHer? (ON</td>
</tr>
<tr>
<td>:DISPLAY:COLumn &lt;number&gt;</td>
<td>:DISPLAY:COLumn? &lt;number&gt; ::= 0 through 63; an integer in NR1 format</td>
<td></td>
</tr>
<tr>
<td>:DISPLAY:CONNect (ON</td>
<td>OFF)</td>
<td>:DISPLAY:CONNect? (ON</td>
</tr>
<tr>
<td>:DISPLAY:DATA &lt;binary block_data&gt;</td>
<td>:DISPLAY:DATA? &lt;binary block_data&gt; ::= 16256 bytes of data in IEEE 488.2 # format</td>
<td></td>
</tr>
</tbody>
</table>
**Programmer’s Quick Reference**

**Commands and Queries**

<table>
<thead>
<tr>
<th>Command</th>
<th>Query</th>
<th>Options and Query Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>:DISPlay:GRID (ON</td>
<td>OFF</td>
<td>SIMPle</td>
</tr>
<tr>
<td>:DISPlay:INVerse (ON</td>
<td>OFF)</td>
<td>:DISPlay:INVerse?</td>
</tr>
<tr>
<td>:DISPlay:LINE &lt;string&gt;</td>
<td>n/a</td>
<td>&lt;string&gt; := any series of ASCII characters enclosed in quotation marks</td>
</tr>
<tr>
<td>:DISPlay:PAlette &lt;palette_number&gt;</td>
<td>:DISPlay:PAlette?</td>
<td>&lt;palette_number&gt; ::= 0 through 6; an integer in NR1 format</td>
</tr>
</tbody>
</table>
| :DISPlay:PIXel <x>, <y>, <intensity> | :DISPlay:PIXel? <x>,<y> | For 54616C:  
  <x> ::= x coordinate of the pixel to be set; an integer (0 to 500) in NR1 format  
  <y> ::= y coordinate of the pixel to be set; an integer (0 to 275) in NR1 format  
  <intensity> (command) ::= an integer in NR1 format:  
    0 to clear pixel  
    1 to light pixel in autostore plane  
    2 to light pixel in graticule plane  
  <intensity> (query) ::= an integer in NR1 format:  
    0 for pixel off  
    1 for autostore and any text on  
    2 for any waveforms on  
    3 for autostore and any text on, and any waveform on  
For all other models:  
  <x> ::= x coordinate of the pixel to be set; an integer (0 to 511) in NR1 format  
  <y> ::= y coordinate of the pixel to be set; an integer (0 to 303) in NR1 format  
  <intensity> (command) ::= an integer in NR1 format:  
    0 to clear pixel  
    1 for half-bright  
    2 for full-bright  
    other value to clear pixel  
  <intensity> (query) ::= an integer in NR1 format:  
    0 for pixel off  
    1 for pixel with half-bright on  
    2 for pixel with full-bright on  
    3 for pixel with both half-bright and full-bright on |
| :DISPlay:ROW <row number> | :DISPlay:ROW? | <row number> ::= 1...20; an integer in NR1 format |
Programmer’s Quick Reference
Commands and Queries

<table>
<thead>
<tr>
<th>Command</th>
<th>Query</th>
<th>Options and Query Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>n/a</td>
<td>:DISPlay:SETup?</td>
<td>:DISPlay:ROW &lt;row_number&gt;; &lt;row_number&gt; ::= 1...20; an integer in NR1 format</td>
</tr>
<tr>
<td></td>
<td></td>
<td>COLUMN &lt;column_number&gt;; &lt;column_number&gt; ::= 0...63; an integer in NR1 format</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INVerse &lt;inverse&gt;; &lt;inverse&gt; ::= (ON</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GRID &lt;grid&gt;; &lt;grid&gt; ::= (ON</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SOURce &lt;source&gt;; &lt;source&gt; ::= (PMEMory1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CONNect &lt;connect_status&gt;; &lt;connect_status&gt; ::= (ON</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PALette &lt;palette_number&gt; (54616C only); &lt;palette_number&gt; ::= 0...6; an integer in NR1 format</td>
</tr>
<tr>
<td></td>
<td>:DISPlay:SOURce &lt;value&gt;</td>
<td>:DISPlay:SOURce?</td>
</tr>
<tr>
<td>n/a</td>
<td>:DISPlay:TEXT BLANK n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>*ESE</td>
<td>:ERASe &lt;value&gt;</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;value&gt; ::= (PMEMory1</td>
</tr>
<tr>
<td></td>
<td>*ESE?</td>
<td>&lt;mask_argument&gt; ::= 0...255; an integer in NR1 format</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit Weight Enables</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 128 NOT USED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 64 URQ - User Request</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 32 CME - Command Error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 16 EXE - Execution Error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 8 DDE - Device Dependent Error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 4 QYE - Query Error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 2 TRG - Trigger Query</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 1 OPC - Operation Complete</td>
</tr>
<tr>
<td>n/a</td>
<td>*ESR?</td>
<td>&lt;status&gt; ::= 0...255; an integer in NR1 format</td>
</tr>
<tr>
<td></td>
<td></td>
<td>:EXTernal:COUPling (DC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>:EXTernal:INPut (FIFTy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>:EXTernal:OFFSet &lt;offset_argument&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>:EXTernal:PMODe (AUTO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>:EXTernal:PROBe &lt;attenuation&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>:EXTernal:PROTect (OFF</td>
</tr>
</tbody>
</table>
## Programmer's Quick Reference

### Commands and Queries

<table>
<thead>
<tr>
<th>Command</th>
<th>Query</th>
<th>Options and Query Returns</th>
</tr>
</thead>
</table>
| n/a | :EXTernal:SETup? | For the 54610: EXTernal:OFFSet <offset_value>; COUPling {DC | AC | GND}; PROBe {X1 | X10 | X20 | X100}; PMODe {AUTO | MANual}; INPut {FIFTy | ONEMeg}; PROTect {OFF | ON}; SKEW <skew_value>
| | | For the 54615/16: EXTernal:COUPling {DC | AC | GND}; PROBe {X1 | X10 | X20 | X100}; PMODe {AUTO | MANual}; INPut {FIFTy | ONEMeg}; PROTect {OFF | ON}
| :EXTernal:SKEW <skew_value> | :EXTernal:SKEW? | <skew_value> ::= external trigger skew value in seconds in NR3 format |
| :FUNCTION2:CENTer <frequency> | :FUNCTION2:CENTer? | <frequency> ::= the current center frequency in NR3 format. The range of legal values is from 0 Hz to 10.00 GHz. |
| :FUNCTION2:MOVE (LEFT) | n/a | n/a |
| :FUNCTION<N>:OFFSet <offset> | :FUNCTION<N>:OFFSet? | <offset> ::= the value at center screen in NR3 format. The range of legal values is +/-10 times the current sensitivity of the selected function. <N> ::= 1 or 2 |
| :FUNCTION<N>:OPERation <operation> | :FUNCTION<N>:OPERation? | <operation> ::= {ADD | SUBTract | MULTiply} for :FUNCTION1:OPERation. {INTEGRate | DIFFerentiate | FFT} for :FUNCTION2:OPERation. <N> ::= 1 or 2 |
| :FUNCTION2:PEAKs? (FREQ1 | DB1 | FREQ2 | DB2) | :FUNCTION2:PEAKs? | <measurement> ::= {FREQ1 | DB1 | FREQ2 | DB2}. The measurement is the value of the peak specified in NR3 format. |
| :FUNCTION<N>:RANGE <range> | :FUNCTION<N>:RANGE? | <range> ::= the full-scale vertical axis value in NR3 format. The range for FUNCTION1 is 8E-6 to 8E+6. The range for the INTEGRate function is 8E-9 to 400E+3. The range for the DIFFerentiate function is 8E-6 to 1.6E11. The range for the FFT function is 8 to 400 dB/div. <N> ::= 1 or 2 |
| :FUNCTION2:REFERence <level> | :FUNCTION2:REFERence? | <level> ::= the current reference level in NR3 format. The range of legal values is from -160.0 dBV to +240.0 dBV in increments of 2.5 dBV. |
| :FUNCTION2:SOURce (CHANnel1 | CHANnel2 | FUNCTION1) | :FUNCTION2:SOURce? | <CHANnel1 | CHANnel2 | FUNCTION1>. The current reference level value is in NR3 format. The range of legal values is from -160.0 dBV to +240.0 dBV in increments of 2.5 dBV. |
| :FUNCTION2:SPAN <span> | :FUNCTION2:SPAN? | <span> ::= the current frequency span in NR3 format. Legal values are 1.221 Hz to 9.766 GHz |
| :FUNCTION<N>:VIEW (ON | OFF) | :FUNCTION<N>:VIEW? | <ON | OFF> <N> ::= 1 or 2 |
| :FUNCTION2:WINDOW (RECTangular | HANNing | FLATtop | EXPonent) | n/a | n/a |

---

8–9
## Programmer’s Quick Reference

### Commands and Queries

<table>
<thead>
<tr>
<th>Command</th>
<th>Query</th>
<th>Options and Query Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>n/a</td>
<td>*IDN?</td>
<td>HEWLETT-PACKARD,&lt;model&gt;, 0, X.X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;model&gt; ::= the model number of the instrument</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;X.X&gt; ::= the software revision of the instrument</td>
</tr>
<tr>
<td>n/a</td>
<td>*LRN?</td>
<td>&lt;learn_string&gt; ::= a maximum of 218 bytes of data in IEEE 488.2 # format</td>
</tr>
<tr>
<td>:MASK:CREATE</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>:MASK:HDATA</td>
<td>:MASK:HDATA?</td>
<td>&lt;header&gt; ::= block header that contains the ASCII characters #8000998 and is sent prior to the data.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;mask_data&gt; ::= 998 bytes of data that represent the currently selected mask template.</td>
</tr>
<tr>
<td>:MASK:DEST</td>
<td>:MASK:DESTination? (TRACe</td>
<td>PRINter)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(ON</td>
</tr>
<tr>
<td>:MASK:FAIL</td>
<td>:MASK:FAILmode? (IN</td>
<td>OUT)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(ON</td>
</tr>
<tr>
<td>:MASK:INCR</td>
<td>:MASK:INCRement (ON</td>
<td>OFF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(ON</td>
</tr>
<tr>
<td>:MASK:NUM</td>
<td>:MASK:NUMBer?</td>
<td>&lt;number&gt; ::= memory (1 or 2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:MASK:POST</td>
<td>:MASK:POSTfailure? (RUN</td>
<td>STOP)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(RUN</td>
</tr>
<tr>
<td>:MASK:SAVE</td>
<td>:MASK:SAVE?</td>
<td>(ON</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:MASK:STAT</td>
<td>:MASK:STATistics?</td>
<td>&lt;compares, failures, failure %&gt; ::= current number of mask tests performed number of failures detected percentage of failures</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:MASK:TEST</td>
<td>:MASK:TEST?</td>
<td>(ON</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:MASK:TOL</td>
<td>:MASK:TOLerance?</td>
<td>&lt;value&gt; ::= the tolerance used when creating a mask template. The entered value can be from 0.00 to 20.0 percent.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:MEAS:DEF</td>
<td>:MEASure:DEFine DELay</td>
<td>&lt;edgeN&gt; ::= the edge selection for channels 1 and 2. N is the selected edge number (1 to 5).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:MEAS:DEL</td>
<td>:MEASure:DELay?</td>
<td>&lt;return_value&gt; ::= floating point number delay time in seconds in NR3 format</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:MEAS:DUTY</td>
<td>:MEASure: DUTYcycle?</td>
<td>&lt;return_value&gt; ::= ratio of positive pulse width to period in NR3 format</td>
</tr>
</tbody>
</table>

8–10
<table>
<thead>
<tr>
<th>Command</th>
<th>Query</th>
<th>Options and Query Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>:MEASure:FALLtime</td>
<td>:MEASure:FALLtime?</td>
<td>&lt;return_value&gt; ::= time in seconds between the 10% and 90% voltage levels in NR3 format</td>
</tr>
<tr>
<td>:MEASure:FREQuency</td>
<td>:MEASure:FREQuency?</td>
<td>&lt;return_value&gt; ::= frequency in Hertz in NR3 format</td>
</tr>
<tr>
<td>:MEASure:LOWer &lt;voltage&gt;</td>
<td>:MEASure:LOWer?</td>
<td>&lt;voltage&gt; ::= the user-defined lower threshold in volts in NR3 format</td>
</tr>
<tr>
<td>:MEASure:NWIDth</td>
<td>:MEASure:NWIDth?</td>
<td>&lt;return_value&gt; ::= negative pulse width in seconds in NR3 format</td>
</tr>
<tr>
<td>:MEASure:OVERshoot</td>
<td>:MEASure:OVERshoot?</td>
<td>&lt;voltage&gt; ::= the percent of the overshoot of the selected waveform in NR3 format</td>
</tr>
<tr>
<td>:MEASure:PERiod</td>
<td>:MEASure:PERiod?</td>
<td>&lt;return_value&gt; ::= waveform period in seconds in NR3 format</td>
</tr>
<tr>
<td>:MEASure:PHASEs</td>
<td>:MEASure:PHASEs?</td>
<td>&lt;return_value&gt; ::= the phase angle value in degrees in NR3 format</td>
</tr>
<tr>
<td>:MEASure:PRESheEt</td>
<td>:MEASure:PRESheEt?</td>
<td>&lt;return_value&gt; ::= the percent of preshoot of the selected waveform in NR3 format</td>
</tr>
<tr>
<td>:MEASure:PSTArt &lt;value&gt;</td>
<td>:MEASure:PSTArt?</td>
<td>&lt;value&gt; ::= the relative position of time marker 1 in degrees in NR3 format</td>
</tr>
<tr>
<td>:MEASure:PSTOp &lt;value&gt;</td>
<td>:MEASure:PSTOp?</td>
<td>&lt;value&gt; ::= the relative position of time marker 2 in degrees in NR3 format</td>
</tr>
<tr>
<td>:MEASure:PWIDth</td>
<td>:MEASure:PWIDth?</td>
<td>&lt;return_value&gt; ::= width of positive pulse in seconds in NR3 format</td>
</tr>
<tr>
<td>:MEASure:RISEtime</td>
<td>:MEASure:RISEtime?</td>
<td>&lt;return_value&gt; ::= rise time in seconds in NR3 format</td>
</tr>
<tr>
<td>:MEASure:SCRatch n/a</td>
<td>:MEASure:SCRatch?</td>
<td>n/a</td>
</tr>
<tr>
<td>:MEASure:SET100 n/a</td>
<td>:MEASure:SET100?</td>
<td>n/a</td>
</tr>
<tr>
<td>:MEASure:SET360 n/a</td>
<td>:MEASure:SET360?</td>
<td>n/a</td>
</tr>
<tr>
<td>:MEASure:SHOW (ON I OFF)</td>
<td>:MEASure:SHOW?</td>
<td>(ON</td>
</tr>
<tr>
<td>:MEASure:SOURce CHANnel &lt;&lt; &gt;&gt;</td>
<td>:MEASure:SOURce?</td>
<td>&lt;&lt; &gt;&gt; ::= 1 or 2; an integer in NR1 format for 54600/03/10/15/16 1, 2, 3 or 4; an integer in NR1 format for the 54601/02</td>
</tr>
<tr>
<td>:MEASure:DELta?</td>
<td>:MEASure:DELta?</td>
<td>&lt;return_value&gt; ::= time difference in seconds between start and stop markers in NR3 format</td>
</tr>
<tr>
<td>:MEASure:THResholds (T1090</td>
<td>:MEASure:THResholds?</td>
<td>(T1090</td>
</tr>
<tr>
<td>:MEASure:TSTArt &lt;value&gt;</td>
<td>:MEASure:TSTArt?</td>
<td>&lt;value&gt; ::= time at the start marker in seconds in NR3 format</td>
</tr>
<tr>
<td>:MEASure:TSTOp &lt;value&gt;</td>
<td>:MEASure:TSTOp?</td>
<td>&lt;value&gt; ::= time at the stop marker in seconds in NR3 format</td>
</tr>
</tbody>
</table>
| n/a            | :MEASure:TVOLT <<volt_argument>, <slope><occurrence> | <volt_argument> ::= positive or negative voltage level that the waveform must cross.  
|                |                  | <slope> ::= direction of the waveform when <volt_argument> is crossed.                  |
|                |                  | <occurrence> ::= number of crossings to be reported.                                     |
|                |                  | <return_value> ::= time in seconds of specified voltage crossing in NR3 format          |
### Command Query Options and Query Returns

<table>
<thead>
<tr>
<th>Command</th>
<th>Query</th>
<th>Options and Query Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>:MEASure:UPPer</td>
<td>:MEASure:UPPer?</td>
<td>&lt;voltage&gt; ::= the user-defined upper threshold in volts in NR3 format</td>
</tr>
<tr>
<td>:MEASure:VPSTArt</td>
<td>:MEASure:VPSTArt?</td>
<td>&lt;value&gt; ::= the relative position of voltage marker 1 in percent in NR3 format</td>
</tr>
<tr>
<td>:MEASure:VTOP</td>
<td>:MEASure:VTOP?</td>
<td>&lt;return_value&gt; ::= voltage at the top of the waveform in NR3 format</td>
</tr>
<tr>
<td>:MEASure:VMAX</td>
<td>:MEASure:VMAX?</td>
<td>&lt;return_value&gt; ::= maximum voltage of the selected waveform in NR3 format</td>
</tr>
<tr>
<td>:MEASure:VPP</td>
<td>:MEASure:VPP?</td>
<td>&lt;return_value&gt; ::= voltage peak to peak in NR3 format</td>
</tr>
<tr>
<td>:MEASure:VRMS</td>
<td>:MEASure:VRMS?</td>
<td>&lt;return_value&gt; ::= calculated dc RMS voltage in NR3 format</td>
</tr>
<tr>
<td>:MEASure:VSTArt</td>
<td>:MEASure:VSTArt?</td>
<td>&lt;vstart_argument&gt; ::= voltage value for VMarker 1 in NR3 format</td>
</tr>
<tr>
<td>:MEASure:VSTOp</td>
<td>:MEASure:VSTOp?</td>
<td>&lt;vstop_argument&gt; ::= voltage value for VMarker 2 in NR3 format</td>
</tr>
<tr>
<td>:MEASure:VAVerage</td>
<td>:MEASure:VAVerage?</td>
<td>&lt;return_value&gt; ::= calculated average voltage in NR3 format</td>
</tr>
<tr>
<td>:MEASure:VBASe</td>
<td>:MEASure:VBASe?</td>
<td>&lt;base_voltage&gt; ::= voltage at the base of the selected waveform in NR3 format</td>
</tr>
<tr>
<td>n/a</td>
<td>:MEASure:VDELTa?</td>
<td>&lt;return_value&gt; ::= delta V value in volts in NR3 format</td>
</tr>
<tr>
<td>:MEASure:VMIN</td>
<td>:MEASure:VMIN?</td>
<td>&lt;return_value&gt; ::= minimum voltage of the selected waveform in NR3 format</td>
</tr>
<tr>
<td>:MEASure:VTIMe</td>
<td>:MEASure:VTIMe?</td>
<td>&lt;vtime_argument&gt; ::= displayed time from trigger in seconds in NR3 format</td>
</tr>
</tbody>
</table>

---

Programmer's Quick Reference

Commands and Queries

8–12
<table>
<thead>
<tr>
<th>Command</th>
<th>Query</th>
<th>Options and Query Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>:MENU &lt;integer&gt;</td>
<td>:MENU?</td>
<td>&lt;integer&gt;, ::= the following: Menu Number No menu selected 0 Channel 1 1 Channel 2 2 Channel 3 (54601/02) 3 External Trigger (54610/15/16) 3 Channel 4 (54601/02) 4 Math Trigger source 6 Trigger mode 7 Trigger slope 8 Main/delayed (horizontal) 9 Time measurements 10 Voltage measurements 11 Cursors 12 Trace 13 Setup 14 Display 15 Utility/Print. 16</td>
</tr>
<tr>
<td>:MERGe &lt;pixel memory&gt;</td>
<td>n/a</td>
<td>&lt;pixel memory&gt;, ::= {PMEMory1</td>
</tr>
<tr>
<td>*OPC</td>
<td>*OPC?</td>
<td>ASCII '1' is placed in the output queue when all pending device operations have completed.</td>
</tr>
<tr>
<td>n/a</td>
<td>*OPT?</td>
<td>n identifies the module and option pairing. X.X identifies the module software revision. Module: No Opt. 005 With Opt. 005 Basic Interface 0.X.X 50.X.X Test Automation 1.X.X 51.X.X Measurement/Storage 2.X.X 52.X.X</td>
</tr>
<tr>
<td>:PRINT? [enhancement]</td>
<td>n/a</td>
<td>[enhancement], ::= [HiRes, [PCLColor]] HiRes ::= contains both half-bright and full-bright display information PCLColor ::= color DeskJet selection only on 54616C</td>
</tr>
<tr>
<td>:RCL &lt;value&gt;</td>
<td>n/a</td>
<td>&lt;value&gt;, ::= {1</td>
</tr>
<tr>
<td>*RST</td>
<td>n/a</td>
<td>See reset values in the online Programmer’s Reference.</td>
</tr>
<tr>
<td>:RUN</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>*SAV &lt;value&gt;</td>
<td>n/a</td>
<td>&lt;value&gt;, ::= {1</td>
</tr>
<tr>
<td>:SEQUence:NEXT</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>:SEQUence:PREVious</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>:SEQUence:PROTect (ON</td>
<td>OFF)</td>
<td>:SEQUence:PROTect? &lt;protect&gt;, ::= (ON</td>
</tr>
<tr>
<td>:SEQUence:RESet</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>
## Programmer’s Quick Reference
### Commands and Queries

<table>
<thead>
<tr>
<th>Command</th>
<th>Query</th>
<th>Options and Query Returns</th>
</tr>
</thead>
</table>
| :SEQuence:SETup    | :SEQuence:SETup? | MASK ::= an individual mask sent to the setup string.  
STEP ::= an individual step sent to the setup string.  
<number> ::= the mask number or step number sent to the setup string.  
<header> ::= the type of setup to be sent or returned:  
For individual masks, ::= #800001000.  
For individual steps, ::= #80000244.  
For whole sequences, ::= #800064122.  
<setup_string> ::= the setup string to be sent:  
For individual masks, ::= 1000-byte string.  
For individual steps, ::= 244-byte string.  
For whole sequences, ::= 64122-byte string. |
| :SEQuence:STEP     | :SEQuence:STEP? | <number> ::= an integer from 1 to 100 in NR1 format |
| n/a                | :SEQuence:TEST? | <result> ::= an integer from 0 to 3 in NR1 format:  
0 = pass  
1 = fail minimum limit line  
2 = fail maximum limit line  
3 = fail both minimum and maximum limit lines |
| *SRE               | *SRE?      | <mask> ::= sum of all bits that are set, 0,...,255; an integer in NR1 format. <mask> ::= following values:  
|                   |            | Bit | Weight | Name | Condition |
|                   |            | 7   | 128    | Not Used |
|                   |            | 6   | 64     | RQS - Request Service |
|                   |            | 5   | 32     | ESB - Event Status Bit |
|                   |            | 4   | 16     | MAV - Message Available |
|                   |            | 3   | 8      | Not used |
|                   |            | 2   | 4      | Not used |
|                   |            | 1   | 2      | Not used |
|                   |            | 0   | 1      | Not used |
| n/a                | :STATus?   | (ON | OFF) |
|                   |            | <display> ::= (CHANnel <n> | PMEMory{1 | 2}) for the 54600/01/02/03/15/16  
(CHANnel <n> | PMEMory{1 | 2} | EXTer nal) for the 54610  
<n> ::= 1 or 2; an integer in NR1 format for the 54600/03/10/15/16  
1, 2, 3, or 4; an integer in NR1 format for the 54601/02 |
| n/a                | *STB?      | <value> ::= 0,...,255; an integer in NR1 format, as shown in the following:  
|                   |            | Bit | Weight | Name | Condition |
|                   |            | 7   | 128    | NOT USED |
|                   |            | 6   | 64     | RQS/MS | 0 = instrument has no reason for service  
1 = instrument is requesting service |
|                   |            | 5   | 32     | ESB | 0 = no event status conditions occurred  
1 = enabled event status condition occurred |
|                   |            | 4   | 16     | MAV | 0 = no output messages are ready  
1 = an output message is ready |
|                   |            | 3   | 8      | ---- | 0 = not used |
|                   |            | 2   | 4      | ---- | 0 = not used |
|                   |            | 1   | 2      | ---- | 0 = not used |
|                   |            | 0   | 1      | ---- | 0 = not used |
### Programmer’s Quick Reference

#### Commands and Queries

<table>
<thead>
<tr>
<th>Command</th>
<th>Query</th>
<th>Options and Query Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>:STOP</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>:SYSTem:DSP</td>
<td>&lt;string&gt;</td>
<td>string ::= quoted ASCII string</td>
</tr>
<tr>
<td>n/a</td>
<td>:SYSTem:ERRor?</td>
<td>error ::= an integer error code</td>
</tr>
<tr>
<td>n/a</td>
<td>:SYSTem:KEY</td>
<td>key_code ::= -1 to 16, or 19 to 50; an integer</td>
</tr>
<tr>
<td>n/a</td>
<td>:SYSTem:LOCK</td>
<td>value ::= {ON</td>
</tr>
<tr>
<td>n/a</td>
<td>:SYSTem:SETup?</td>
<td>setup_data ::= a maximum of 218 bytes of data in IEEE 488.2 # format.</td>
</tr>
<tr>
<td>:TER?</td>
<td>&lt;return_value&gt;</td>
<td>return_value ::= 0 or 1</td>
</tr>
<tr>
<td>:TIMebase:DELay</td>
<td>&lt;delay_value&gt;</td>
<td>delay_value ::= time from trigger to display reference in seconds. The display reference is left or center in NR3 format.</td>
</tr>
<tr>
<td>:TIMebase:MODE</td>
<td>&lt;value&gt;</td>
<td>value ::= (NORMal</td>
</tr>
<tr>
<td>:TIMebase:RANGe</td>
<td>&lt;range_value&gt;</td>
<td>range_value ::= the following values in NR3 format:</td>
</tr>
<tr>
<td>:TIMebase:REFerence</td>
<td>(LEFT</td>
<td>CENTer)</td>
</tr>
<tr>
<td>n/a</td>
<td>:TIMebase:SETup?</td>
<td>For all models except the 54615/16: TIMebase:MODE (NORM</td>
</tr>
<tr>
<td>:TIMebase:VERNier</td>
<td>(ON</td>
<td>OFF)</td>
</tr>
<tr>
<td>:TRACe:CLEAR</td>
<td>n/a</td>
<td>N ::= the trace memory number (1 to 100)</td>
</tr>
<tr>
<td>:TRACe:DATA</td>
<td>&lt;N&gt;,&lt;trace_data&gt;</td>
<td>&lt;N&gt; ::= the trace memory number (1 to 100). header ::= the 10-byte block header that contains the ASCII characters #8000nnnn and is sent prior to the data. (nnnn is the number of bytes in the data string.) trace_data ::= a maximum of 16,342 bytes of data, setup, and label information that represents the current trace.</td>
</tr>
<tr>
<td>Command</td>
<td>Query</td>
<td>Options and Query Returns</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-----------------------------------------------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>:TRACe:MODE &lt;N&gt; (ON</td>
<td>OFF)</td>
<td>:TRACe:MODE? &lt;N&gt;</td>
</tr>
<tr>
<td>:TRACe:SAVE &lt;N&gt;</td>
<td>n/a</td>
<td>&lt;N&gt; ::= the trace memory number (1 to 100).</td>
</tr>
<tr>
<td>*TRG</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>:TRIGger:COUPling (AC</td>
<td>DC)</td>
<td>:TRIGger:COUPling?</td>
</tr>
<tr>
<td>:TRIGger:FIELd (ALTernate</td>
<td>ONE</td>
<td>TWO</td>
</tr>
<tr>
<td>:TRIGger:HOLDoff &lt;holdoff_time&gt;</td>
<td>:TRIGger:HOLDoff?</td>
<td>&lt;holdoff_time&gt; ::= the holdoff time value in seconds in NR3 format.</td>
</tr>
<tr>
<td>:TRIGger:LEVel &lt;level_argument&gt;</td>
<td>:TRIGger:LEVel?</td>
<td>&lt;return_value&gt; ::= the trigger level in volts in NR3 format.</td>
</tr>
<tr>
<td>:TRIGger:LINE &lt;line_number&gt;</td>
<td>:TRIGger:LINE?</td>
<td>&lt;line_number&gt; ::= integer in NR1 format.</td>
</tr>
<tr>
<td>:TRIGger:MODE (AUTLevel</td>
<td>AUTO</td>
<td>NORMal</td>
</tr>
<tr>
<td>:TRIGger:NREJect (OFF</td>
<td>ON)</td>
<td>:TRIGger:NREJect?</td>
</tr>
<tr>
<td>:TRIGger:OPTMode (LINE</td>
<td>FIELD1</td>
<td>FIELD2</td>
</tr>
<tr>
<td>:TRIGger:POLarity (POSitive</td>
<td>NEGative)</td>
<td>:TRIGger:POLarity?</td>
</tr>
<tr>
<td>:TRIGger:REJect (OFF</td>
<td>LF</td>
<td>HF)</td>
</tr>
<tr>
<td>n/a</td>
<td>:TRIGger:SETup?</td>
<td>TRIG:MODE (AUTL</td>
</tr>
<tr>
<td>:TRIGger:SLOPe (NEGative</td>
<td>POSitive)</td>
<td>:TRIGger:SLOPe?</td>
</tr>
</tbody>
</table>

Programmer’s Quick Reference
Commands and Queries

8–16
<table>
<thead>
<tr>
<th>Command</th>
<th>Query</th>
<th>Options and Query Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>.TRIGger:SOURce &lt;source&gt;</td>
<td>.TRIGger:SOURce? &lt;source&gt; ::=</td>
<td>(CHANnel1</td>
</tr>
<tr>
<td>.TRIGger:STANdard (GENeric</td>
<td>NTSC</td>
<td>PAL</td>
</tr>
<tr>
<td>.TRIGger:TVHFrej (OFF</td>
<td>ON)</td>
<td>.TRIGger:TVHFrej? (OFF</td>
</tr>
<tr>
<td>.TRIGger:TVMode &lt;mode&gt;</td>
<td>.TRIGger:TVMode? &lt;mode&gt; ::=</td>
<td>(LINE</td>
</tr>
<tr>
<td>.TRIGger:VIR (ON</td>
<td>OFF)</td>
<td>.TRIGger:VIR? (ON</td>
</tr>
<tr>
<td>n/a</td>
<td>*TST? &lt;result&gt; ::= 0 or non-zero value; an integer in NR1 format 0 indicates the test passed. Non-zero indicates the test failed.</td>
<td></td>
</tr>
<tr>
<td>n/a</td>
<td>.VIEW &lt;display&gt; n/a &lt;display&gt; ::= (CHANnel &lt;n&gt;</td>
<td>PMEMory{1</td>
</tr>
<tr>
<td>n/a</td>
<td>*WAI n/a n/a</td>
<td></td>
</tr>
<tr>
<td>n/a</td>
<td>.WAVeform:BYTeorder &lt;value&gt; .WAVeform:BYTeorder? &lt;value&gt; ::= (LSBFirst</td>
<td>MSBFirst)</td>
</tr>
<tr>
<td>n/a</td>
<td>.WAVeform:DATA &lt;binary block data in # format&gt; .WAVeform:DATA? &lt;binary block length bytes&gt;, &lt;binary data&gt; For example, to transmit 4000 bytes of data, the syntax would be: #800004000&lt;4000 bytes of data&gt;&lt;NL&gt; 8 is the number of digits that follow 00004000 is the number of bytes to be transmitted &lt;4000 bytes of data&gt; is the actual data</td>
<td></td>
</tr>
<tr>
<td>n/a</td>
<td>.WAVeform:FORMat &lt;value&gt; .WAVeform:FORMat? &lt;value&gt; ::= (ASC</td>
<td>WORD</td>
</tr>
<tr>
<td>n/a</td>
<td>.WAVeform:POINts &lt;value&gt; .WAVeform:POINts? &lt;value&gt; ::= integer {100</td>
<td>200</td>
</tr>
</tbody>
</table>
## Programmer’s Quick Reference

### Commands and Queries

<table>
<thead>
<tr>
<th>Command</th>
<th>Query</th>
<th>Options and Query Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>n/a</td>
<td>:WAVeform:PREamble?</td>
<td>&lt;preamble_block&gt; ::= &lt;format NR1&gt;, &lt;type NR1&gt;, &lt;points NR1&gt;, &lt;count NR1&gt;, &lt;xincrement NR3&gt;, &lt;xorigin NR3&gt;, &lt;xreference NR1&gt;, &lt;yincrement NR3&gt;, &lt;yorigin NR3&gt;, &lt;yreference NR1&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;format&gt; ::= an integer in NR1 format: 0 for ASCII format, 1 for BYTE format, 2 for WORD format</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;type&gt; ::= an integer in NR1 format: 0 for AVERAGE type, 1 for NORMAL type, 2 for PEAK detect type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;count&gt; ::= an integer in NR1 format: 1, always 1 and is present for compatibility</td>
</tr>
<tr>
<td>n/a</td>
<td>:WAVeform:SOURce?</td>
<td>&lt;return_mode&gt; ::= {1</td>
</tr>
<tr>
<td>CHANnel &lt;n&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n/a</td>
<td>:WAVeform:TYPE?</td>
<td>&lt;return_mode&gt; ::= (NORMAL</td>
</tr>
<tr>
<td>n/a</td>
<td>:WAVeform:XINcrement?</td>
<td>&lt;return_value&gt; ::= x-increment value in the current preamble in NR3 format</td>
</tr>
<tr>
<td>n/a</td>
<td>:WAVeform:XORigin?</td>
<td>&lt;return_value&gt; ::= x-origin value in the current preamble in NR3 format</td>
</tr>
<tr>
<td>n/a</td>
<td>:WAVeform:XREFERENCE?</td>
<td>&lt;return_value&gt; ::= x-reference value in the current preamble in NR1 format</td>
</tr>
<tr>
<td>n/a</td>
<td>:WAVeform:YINcrement?</td>
<td>&lt;return_value&gt; ::= y-increment value in the current preamble in NR3 format</td>
</tr>
<tr>
<td>n/a</td>
<td>:WAVeform:YORigin?</td>
<td>&lt;return_value&gt; ::= y-origin value in the current preamble in NR3 format</td>
</tr>
<tr>
<td>n/a</td>
<td>:WAVeform:YREFERENCE?</td>
<td>&lt;return_value&gt; ::= y-reference value in the current preamble in NR1 format</td>
</tr>
</tbody>
</table>
H.3 MUX
Quick Reference Guide

SCPI Command Summary
The following conventions are used for SCPI command syntax for remote interface programming:

- Square brackets ([ ]) indicate optional keywords or parameters.
- Braces ({ }) enclose parameter choices within a command string.
- Triangle brackets (< >) enclose parameters for which you must substitute a value.
- A vertical bar (|) separates multiple parameter choices.

Rules for Using a Channel List
Many of the SCPI commands for the 34970A include a scan_list or ch_list parameter which allow you to specify one or more channels. The channel number has the form (@scc), where s is the slot number (100, 200, or 300) and cc is the channel number. You can specify a single channel, multiple channels, or a range of channels as shown below.

- The following command configures a scan list to include only channel 10 on the module in slot 300.
  
  ROUT:SCAN (@310)

- The following command configures a scan list to include multiple channels on the module in slot 200. The scan list now contains only channels 10, 12, and 15 (the scan list is redefined each time you send a new ROUTe:SCAN command).

  ROUT:SCAN (@210, 212, 215)

- The following command configures a scan list to include a range of channels. When you specify a range of channels, the range may contain invalid channels (they are ignored), but the first and last channel in the range must be valid. The scan list now contains channels 5 through 10 (slot 100) and channel 15 (slot 200).

  ROUT:SCAN (@105:110, 215)
Scan Measurement Commands
(see page 226 in the User’s Guide)

MEASure
:TEMPerature? {TCouple|RTD|FRTD|THERmistor|DEF}
    ,{(type) [DEF]},[1[,{(resolution) [MIN|MAX|DEF]}] ,[@<scan_list>]
:VOLtage:DC? [{<range> AUTO|MIN|MAX|DEF}
    [,,(resolution) [MIN|MAX|DEF]] ,[@<scan_list>]
:VOLtage:AC? [{<range> AUTO|MIN|MAX|DEF}
    [,,(resolution) [MIN|MAX|DEF]] ,[@<scan_list>]
:RESistance? [{<range> AUTO|MIN|MAX|DEF}
    [,,(resolution) [MIN|MAX|DEF]] ,[@<scan_list>]
:FREQuency? [{<range> AUTO|MIN|MAX|DEF}
    [,,(resolution) [MIN|MAX|DEF]] ,[@<scan_list>]
:PERiod? [{<range> AUTO|MIN|MAX|DEF}
    [,,(resolution) [MIN|MAX|DEF]] ,[@<scan_list>]
:TOTalize? {READ|RESET} ,[@<scan_list>]

Monitor Commands
(see page 237 in the User’s Guide)

ROUTE
:MONitor (@<channel>)
:MONitor?

ROUTE
:MONitor:STATe {OFF|ON}
:MONitor:STATe?

ROUTE:MONitor:DATA?

Scan Statistics Commands
(see page 233 in the User’s Guide)

CALCulate
:AVERage:MINimum? [@<ch_list>]
:AVErage:MINimum:TIME? [@<ch_list>]
:AVERage:MAXimum? [@<ch_list>]
:AVErage:MAXimum:TIME? [@<ch_list>]
:AVERage:AVERage? [@<ch_list>]
:AVERage:PTPeak? [@<ch_list>]
:AVERage:COUNt? [@<ch_list>]
:AVERage:CLEar [@<ch_list>]

DATA:LAST? [<num_rdgs>],[@<channel>]

This command redefines the scan list when executed.
Default parameters are shown in bold.
Scan Configuration Commands

(see page 226 in the User’s Guide)

ROUTE
  :SCAN (@<scan_list>)
  :SCAN?
  :SCAN:SIZE?

TRIGger
  :SOURce {BUS|IMMediate|EXTernal|ALARM1|ALARM2|ALARM3|ALARM4|TIMer}
  :SOURce?

TRIGger
  :TIMer <seconds>|MIN|MAX
  :TIMer?

TRIGger
  :COUNT <count>|MIN|MAX|INFinity
  :COUNT?

ROUTE
  :CHANnel:DELAY <seconds>[,( @<ch_list> )]
  :CHANnel:DELAY? [,( @<ch_list> )]
  :CHANnel:DELAY:AUTO OFF|ON[,( @<ch_list> )]
  :CHANnel:DELAY:AUTO? [,( @<ch_list> )]

FORMat
  :READing:ALARm OFF|ON
  :READing:ALARm?
  :READing:CHANnel OFF|ON
  :READing:CHANnel?
  :READing:TIME OFF|ON
  :READing:TIME?
  :READing:UNIT OFF|ON
  :READing:UNIT?

FORMat
  :READing:TIME:TYPE ABSolute|RELative
  :READing:TIME:TYPE?

ABORT
INITiate
READ?

Scan Memory Commands

(see page 235 in the User’s Guide)

DATA:POINts?
DATA:REMove? <num_rdgs>
SYSTEM:TIME:SCAN?
FETCh?
R? [<max_road>]

This command redefines the scan list when executed.
This command applies to all channels in the instrument (Global setting).
Default parameters are shown in bold.
Scanning With an External Instrument

(see page 239 in the User’s Guide)

ROUTE
  :SCAN (@<scan_list>)
  :SCAN?
  :SCAN:SIZE?

TRIGGER
  :SOURce {BUS|IMMediate|EXTernal|TIMer}
  :SOURce?

TRIGGER
  :TIMer [<seconds>|MIN|MAX]
  :TIMer?

TRIGGER
  :COUNt {<count>|MIN|MAX|INFinity}
  :COUNt?

ROUTE
  :CHANnel:DELAY <seconds>[,(<ch_list>)]
  :CHANnel:DELAY? [(<ch_list>)]

ROUTE
  :CHANnel:ADVance:SOURce {EXTernal|BUS|IMMediate}
  :CHANnel:ADVance:SOURce?

ROUTE
  :CHANnel:FWIRe {OFF|ON}[,(<ch_list>)]
  :CHANnel:FWIRe? [(<ch_list>)]

INSTRUMENT
  :DMM {OFF|ON}
  :DMM?
  :DMM:INSTalled?

This command redefines the scan list when executed.
This command applies to all channels in the instrument (Global setting). Default parameters are shown in bold.
Temperature Configuration Commands

(see page 219 in the User’s Guide)

6 CONFIGure
  :TEMPerature {TCouple|RTD|FRTD|THERmistor|DEF}
  ,{<type>|[DEF]}[,1],[,<resolution>|MIN|MAX|DEF}],[<scan_list>]
CONFIGure? [([<ch_list>])]

UNIT
  :TEMPerature {C|F|K},[<ch_list>]
  :TEMPerature? [([<ch_list>])]

[SENSe:]TEMPerature:TRANducer
  :TYPE {TCouple|RTD|FRTD|THERmistor|DEF},[<ch_list>]
  :TYPE? [([<ch_list>])]

[SENSe:]TEMPerature:TRANducer
  :TCouple:TYPE {B|E|J|K|N|R|S|T},[<ch_list>]
  :TCouple:CHECK {OFF|ON},[<ch_list>]
  :TCouple:CHECK? [([<ch_list>])]

[SENSe:]TEMPerature:TRANducer
  :TCouple:RJUNction:TYPE {INTernal|EXTernal|FIXed},[<ch_list>]
  :TCouple:RJUNction? [([<ch_list>])]

[SENSe:]TEMPerature:TRANducer
  :RTD:TYPE {85|91},[<ch_list>]
  :RTD:TYPE? [([<ch_list>])]
  :RTD:RESistance[:REFerence] <reference>,[<ch_list>]
  :RTD:RESistance[:REFerence]? [([<ch_list>])]

[SENSe:]TEMPerature:TRANducer
  :FRTD:TYPE {85|91},[<ch_list>]
  :FRTD:TYPE? [([<ch_list>])]
  :FRTD:RESistance[:REFerence] <reference>,[<ch_list>]
  :FRTD:RESistance[:REFerence]? [([<ch_list>])]

[SENSe:]TEMPerature:TRANducer
  :THERmistor:TYPE {2252|5000|10000},[<ch_list>]
  :THERmistor:TYPE? [([<ch_list>])]

[SENSe:]TEMPerature:NPLC {0.02|0.2|1|2|10|20|100|200}[,MIN|MAX|DEF],[<ch_list>]
  TEMPerature:NPLC? [([<ch_list>])]

5 This command redefines the scan list when executed.
   Default parameters are shown in bold.

5
Voltage Configuration Commands

(see page 223 in the User’s Guide)

```plaintext
CONFigure
 :VOLTage:DC [{<range>|AUTO|MIN|MAX|DEF},[,<resolution>|MIN|MAX|DEF}], (@<scan_list>)
CONFigure? [(@<ch_list>)]

[SENSe:]
VOLTage:DC:RANGe {<range>|MIN|MAX}[,(@<ch_list>)]
VOLTage:DC:RANGe? [{(0,<ch_list>)|MIN|MAX}]
VOLTage:DC:RANGe:AUTO {OFF|ON}[,(@<ch_list>)]
VOLTage:DC:RANGe:AUTO? [(@<ch_list>)]

[SENSe:]
VOLTage:DC:RESolution {<resolution>|MIN|MAX}[,(@<ch_list>)]
VOLTage:DC:RESolution? [{(0,<ch_list>)|MIN|MAX}]

[SENSe:]
VOLTage:DC:APERture {<time>|MIN|MAX}[,(@<ch_list>)]
VOLTage:DC:APERture? [{(0,<ch_list>)|MIN|MAX}]

[SENSe:]
VOLTage:DC:NPLC {0.02|0.2|1|2|10|20|100|200|MIN|MAX}[,(@<ch_list>)]
VOLTage:DC:NPLC? [{(0,<ch_list>)|MIN|MAX}]

INPut
:IMPedance:AUTO {OFF|ON}[,(@<ch_list>)]
:IMPedance:AUTO? [{(0,<ch_list>)}]

[SENSe:]
ZERO:AUTO {OFF|ONCE|ON}[,(@<ch_list>)]
ZERO:AUTO? [{(0,<ch_list>)}]

CONFigure
 :VOLTage:AC [{<range>|AUTO|MIN|MAX|DEF},[,<resolution>|MIN|MAX|DEF}], (@<scan_list>)
CONFigure? [(@<ch_list>)]

[SENSe:]
VOLTage:AC:RANGe {<range>|MIN|MAX}[,(@<ch_list>)]
VOLTage:AC:RANGe? [{(0,<ch_list>)|MIN|MAX}]
VOLTage:AC:RANGe:AUTO {OFF|ON}[,(@<ch_list>)]
VOLTage:AC:RANGe:AUTO? [(@<ch_list>)]

[SENSe:]
VOLTage:AC:BANDwidth {3|20|200|MIN|MAX}[,(@<ch_list>)]
VOLTage:AC:BANDwidth? [{(0,<ch_list>)|MIN|MAX}]
```

This command redefines the scan list when executed.
Default parameters are shown in bold.
Resistance Configuration Commands

(see page 224 in the User’s Guide)

configure
 :resistance [{<range>|auto|min|max|def},[,<resolution>|min|max|def}], (@<scan_list>)
 configure? [(@<ch_list>)]

[SENSe:]
resistance:range {<range>|min|max}[,(@<ch_list>)]
resistance:range? [((@<ch_list>)|min|max)]
resistance:range:auto {off|on}[,(@<ch_list>)]
resistance:range:auto? [((@<ch_list>)|min|max)]

[SENSe:]
resistance:resolution {<resolution>|min|max}[,(@<ch_list>)]
resistance:resolution? [((@<ch_list>)|min|max)]
resistance: aperture {<time>|min|max}[,(@<ch_list>)]
resistance: aperture? [((@<ch_list>)|min|max)]
resistance:nplc (0.02|0.2|1|2|10|20|100|200|min|max)[,(@<ch_list>)]
resistance:nplc? [((@<ch_list>)|min|max)]

[SENSe:]
resistance: ocompensated {off|on}[,(@<ch_list>)]
resistance: ocompensated? [((@<ch_list>)|off|on)]

configure
 :fresistance [{<range>|auto|min|max|def},[,<resolution>|min|max|def}], (@<scan_list>)
 configure? [(@<ch_list>)]

[SENSe:]
 fresistance:range {<range>|min|max}[,(@<ch_list>)]
 fresistance:range? [((@<ch_list>)|min|max)]
 fresistance:range:auto {off|on}[,(@<ch_list>)]
 fresistance:range:auto? [((@<ch_list>)|min|max)]

[SENSe:]
 fresistance:resolution {<resolution>|min|max}[,(@<ch_list>)]
 fresistance:resolution? [((@<ch_list>)|min|max)]
 fresistance: aperture {<time>|min|max}[,(@<ch_list>)]
 fresistance: aperture? [((@<ch_list>)|min|max)]
 fresistance:nplc (0.02|0.2|1|2|10|20|100|200|min|max)[,(@<ch_list>)]
 fresistance:nplc? [((@<ch_list>)|min|max)]

[SENSe:]
 fresistance: ocompensated {off|on}[,(@<ch_list>)]
 fresistance: ocompensated? [((@<ch_list>)|off|on)]

This command redefines the scan list when executed.
Default parameters are shown in bold.
Current Configuration Commands

(see page 224 in the User’s Guide)

Valid only on channels 21 and 22 on the 34901A multiplexer module.

**CONFigure**

:CURRENT:DC [{<range> | AUTO | MIN | MAX | DEF}
,
{<resolution> | MIN | MAX | DEF}],] (ch_list)

CONFigure? [(ch_list)]

[SENSe:]

CURRENT:DC:RANGe {<range> | MIN | MAX},] (ch_list)

CURRENT:DC:RANGe? [(ch_list)]

CURRENT:DC:RANGe:AUTO {OFF | ON},] (ch_list)

CURRENT:DC:RANGe:AUTO? [(ch_list)]

[SENSe:]

CURRENT:DC:RESolution {<resolution> | MIN | MAX},] (ch_list)

CURRENT:DC:RESolution? [(ch_list)]

CURRENT:DC:APERture {<time> | MIN | MAX},] (ch_list)

CURRENT:DC:APERture? [(ch_list)]

CURRENT:DC:NPLC {0.02 | 0.2 | 1 | 2 | 10 | 20 | 100 | 200 | MIN | MAX},] (ch_list)

CURRENT:DC:NPLC? [(ch_list)]

**CONFigure**

:CURRENT:AC [{<range> | AUTO | MIN | MAX | DEF}
,
{<resolution> | MIN | MAX | DEF}],] (ch_list)

CONFigure? [(ch_list)]

[SENSe:]

CURRENT:AC:RANGe {<range> | MIN | MAX},] (ch_list)

CURRENT:AC:RANGe? [(ch_list)]

CURRENT:AC:RANGe:AUTO {OFF | ON},] (ch_list)

CURRENT:AC:RANGe:AUTO? [(ch_list)]

[SENSe:]

CURRENT:AC:BANDwidth {3 | 20 | 200 | MIN | MAX},] (ch_list)

CURRENT:AC:BANDwidth? [(ch_list)]

This command redefines the scan list when executed.
Default parameters are shown in **bold**.
Frequency and Period Configuration Commands
(see page 214 in the User’s Guide)

Configure
  :FREQuency [{<range>|AUTO|MIN|MAX|DEF}
  [,<resolution>|MIN|MAX|DEF}],] (@<scan_list>)
  Configure? [(@<ch_list>)]

[SENSe:] FREQuency:VOLTage:RANGe<{<range>|MIN|MAX}[,(@<ch_list>)}
FREQuency:VOLTage:RANGe? [{(@<ch_list>)|MIN|MAX}]
FREQuency:VOLTage:RANGe:AUTO{OFF|ON},] (@<ch_list>)
FREQuency:VOLTage:RANGe:AUTO? [(@<ch_list>)]

[SENSe:] FREQuency:APERture {0.01|0.1|1|MIN|MAX}, (@<ch_list>)
FREQuency:APERture? [{(@<ch_list>)|MIN|MAX}]

[SENSe:] FREQuency:RANGe:LOWer{3|20|200|MIN|MAX}[,(@<ch_list>)}
FREQuency:RANGe:LOWer? [{(@<ch_list>)|MIN|MAX}]

Configure
  :PERiod [{<range>|AUTO|MIN|MAX|DEF}
  [,<resolution>|MIN|MAX|DEF}],] (@<scan_list>)
  Configure? [(@<ch_list>)]

[SENSe:] PERiod:VOLTage:RANGe<{<range>|MIN|MAX}[,(@<ch_list>)}
PERiod:VOLTage:RANGe? [{(@<ch_list>)|MIN|MAX}]
PERiod:VOLTage:RANGe:AUTO{OFF|ON},] (@<ch_list>)
PERiod:VOLTage:RANGe:AUTO? [(@<ch_list>)]

[SENSe:] PERiod:APERture {0.01|0.1|1|MIN|MAX}, (@<ch_list>)
PERiod:APERture? [{(@<ch_list>)|MIN|MAX}]

This command redefines the scan list when executed.
Default parameters are shown in **bold**.
Mx+B Scaling Commands
(see page 244 in the User’s Guide)

CALCulate
:SCALE:GAIN <gain>
:SCALE:GAIN? 
:SCALE:OFFSET <offset>
:SCALE:OFFSET? 
:SCALE:UNIT <quoted_string>
:SCALE:UNIT? 
CALCulate:SCALE:OFFSET:NULL
CALCulate
:SCALE:STATE {OFF|ON}
:SCALE:STATE? 

Alarm Limit Commands
(see page 247 in the User’s Guide)

OUTPUT
:ALARM{1|2|3|4}:SOURCE (@<ch_list>)
:ALARM{1|2|3|4}:SOURCE?
CALCulate
:LIMIT:UPPER <hi_limit>
:LIMIT:UPPER? 
:LIMIT:UPPER:STATE {OFF|ON}
:LIMIT:UPPER:STATE? 
CALCulate
:LIMIT:LOWER <lo_limit>
:LIMIT:LOWER? 
:LIMIT:LOWER:STATE {OFF|ON}
:LIMIT:LOWER:STATE? 

SYSTEM:ALARM?

OUTPUT
:ALARM:MODE {LATCH|TRACK}
:ALARM:MODE?
:ALARM:SLOPe {NEGative|POSitive}
:ALARM:SLOPe?

OUTPUT
:ALARM{1|2|3|4}:CLEAR
:ALARM:CLEAR:ALL

STATUS
:ALARM:CONDITION?
:ALARM:ENABLE <enable_value>
:ALARM:ENABLE?

<table>
<thead>
<tr>
<th>Ch 01 DIO (LSB)</th>
<th>Ch 02 DIO (MSB)</th>
<th>Ch 03 Totalizer</th>
<th>Ch 04 DAC</th>
<th>Ch 05 DAC</th>
</tr>
</thead>
</table>

CALCulate
:COMPARE:TYPE {EQUal|NEQual}
:COMPARE:TYPE? 
:COMPARE:DATA <data>
:COMPARE:DATA? 
:COMPARE:MASK <mask>
:COMPARE:MASK? 
:COMPARE:STATE {OFF|ON}
:COMPARE:STATE? 

This command applies to all channels in the instrument (Global setting).
Default parameters are shown in bold.
### Digital Input Commands
*(see page 255 in the User’s Guide)*

<table>
<thead>
<tr>
<th>Ch 01</th>
<th>Ch 02</th>
<th>Ch 03</th>
<th>Ch 04</th>
<th>Ch 05</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIO (LSB)</td>
<td>DIO (MSB)</td>
<td>Totalizer</td>
<td>DAC</td>
<td>DAC</td>
</tr>
</tbody>
</table>

**CONFigure:DIGital:BYTE** (@<scan_list>)

**CONFigure?** (;;) @<ch_list>

**[SENSe:]DIGital:DATA:**[@<BYTE WORD> (;;)] @<ch_list>

### Totalizer Commands
*(see page 256 in the User’s Guide)*

<table>
<thead>
<tr>
<th>Ch 01</th>
<th>Ch 02</th>
<th>Ch 03</th>
<th>Ch 04</th>
<th>Ch 05</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIO (LSB)</td>
<td>DIO (MSB)</td>
<td>Totalizer</td>
<td>DAC</td>
<td>DAC</td>
</tr>
</tbody>
</table>

**CONFigure:TOTalize** {READ|RRESet} (;; @<scan_list>)

**CONFigure?** (;;) @<ch_list>

**[SENSe:]TOTalize:**[@<TYPE> READ|RRESet> (;;)] @<ch_list>

**[SENSe:]TOTalize:SLOPe** {NEGative|POSitive} (;; @<ch_list>)

**[SENSe:]TOTalize:CLEAR:IMMediate** @<ch_list>

**[SENSe:]TOTalize:DATA** @<ch_list>

### Digital Output Commands
*(see page 258 in the User’s Guide)*

<table>
<thead>
<tr>
<th>Ch 01</th>
<th>Ch 02</th>
<th>Ch 03</th>
<th>Ch 04</th>
<th>Ch 05</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIO (LSB)</td>
<td>DIO (MSB)</td>
<td>Totalizer</td>
<td>DAC</td>
<td>DAC</td>
</tr>
</tbody>
</table>

**SOURce:**[@<DIGital:DATA>[@<BYTE WORD>]> (;; @<ch_list>)

**SOURce:DIGital:DATA**[@<BYTE WORD> (;;)] @<ch_list>

**SOURce:DIGital:STATE?** @<ch_list>

### DAC Output Commands
*(see page 258 in the User’s Guide)*

<table>
<thead>
<tr>
<th>Ch 01</th>
<th>Ch 02</th>
<th>Ch 03</th>
<th>Ch 04</th>
<th>Ch 05</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIO (LSB)</td>
<td>DIO (MSB)</td>
<td>Totalizer</td>
<td>DAC</td>
<td>DAC</td>
</tr>
</tbody>
</table>

**SOURce:**[@<VOLTage> <voltage> (;; @<ch_list>)

**SOURce:**[@<VOLTage> (;; @<ch_list>)

This command redefines the scan list when executed.
Default parameters are shown in **bold**.
**Switch Control Commands**  
*(see page 259 in the User’s Guide)*

ROUTE  
:CLOSE (@<ch_list>)  
:CLOSE:EXCLUSIVE (@<ch_list>)  
:CLOSE? (@<ch_list>)

ROUTE  
:OPEN (@<ch_list>)  
:OPEN? (@<ch_list>)

ROUTE:DONE?

SYSTEM:CPON {100|200|300|ALL}

**Scan Triggering Commands**  
*(see page 228 in the User’s Guide)*

TRIGGER  
:SOURce {BUS|IMMediate|EXITernal|ALARM1|ALARM2|ALARM3|ALARM4|TIMer}  
:SOURce?

TRIGGER  
:TIMer {<seconds>|MIN|MAX}  
:TIMer?

TRIGGER  
:COUNt {<count>|MIN|MAX|INfinity}  
:COUNt?

*TRG

INITiate

READ?

**State Storage Commands**  
*(see page 261 in the User’s Guide)*

*SAV {0|1|2|3|4|5}  
*RCL {0|1|2|3|4|5}  

MEMORY:STATE  
:NAME {1|2|3|4|5} [,<name>]  
:NAME? {1|2|3|4|5}

MEMORY:STATE:DELeTe {0|1|2|3|4|5}

MEMORY:STATE  
:RECall:AUTO {OFF|ON}  
:RECall:AUTO?

MEMORY:STATE:VALid? {0|1|2|3|4|5}

MEMORY:NSTates?

* This command applies to all channels in the instrument (Global setting).  
* Default parameters are shown in **bold**.
System-Related Commands
(see page 264 in the User’s Guide)

SYSTem
  :DATE <yyyy>,<mm>,<dd>
  :DATE?
  :TIME <hh>,<mm>,<ss.sss>
  :TIME?

FORMat
  :READing:TIME:TYPE {ABSolute|RELative}
  :READing:TIME:TYPE?

*IDN?

SYSTem:CTYPE? {100|200|300}

DIAgnostic
  :POKE:SLOT:DATA {100|200|300}, <quoted_string>
  :PEEK:SLOT:DATA? {100|200|300}

DISPlay {OFF|ON}
  DISPlay?

DISPlay
  :TEXT <quoted_string>
  :TEXT?

  :TEXT:CLEar

*RST

SYSTem:PRESet

SYSTem:CPON {100|200|300|ALL}

SYSTem:ERRor?

SYSTem:ALARm?

SYSTem:VERSion?

*TST?

Interface Configuration Commands
(see page 269 in the User’s Guide)

SYSTem:INTerface {GPIB|RS232}

SYSTem:LOCal

SYSTem:REMoTe

SYSTem:RWLock

Default parameters are shown in bold.
**Status System Commands**

*(see page 286 in the User’s Guide)*

*STB?
*SRE <enable_value>*
*SRS?

STATUs
  :QUEStionable:CONDition?
  :QUEStionable[:EVENt]?
  :QUEStionable:ENABle <enable_value>
  :QUEStionable:ENABle?

*ESR?
*RSE <enable_value>*
*RSE?

STATUs
  :ALARm:CONDition?
  :ALARm[:EVENt]?
  :ALARm:ENABLE <enable_value>
  :ALARm:ENABLE?

STATUs
  :OPERation:CONDition?
  :OPERation[:EVENt]?
  :OPERation:ENABLE <enable_value>
  :OPERation:ENABLE?

DATA:POINts
  :EVENt:THReshold <num_rdgs>
  :EVENt:THReshold?

*CLS

*PSC {0|1}

*PSC?

*OPC

**Calibration Commands**

*(see page 292 in the User’s Guide)*

CALibration?

CALibration:COUNT?

CALibration
  :SECure:CODE <new_code>
  :SECure:STATe {OFF|ON},<code>
  :SECure:STATe?

CALibration
  :STRING <quoted_string>
  :STRING?

CALibration
  :VALue <value>
  :VALue?
Service-Related Commands
(see page 294 in the User’s Guide)

INStrument
  :DMM {OFF|ON}
  :DMM?
  :DMM:INSTalled?

DIAGnostic
  :DMM:CYCles?
  :DMM:CYCles:CLEar {1|2|3}

DIAGnostic
  :RELay:CYCles? [@<ch_list>]
  :RELay:CYCles:CLEar [@<ch_list>]

*RST
SYSTem:PRESet
SYSTem:CPON {100|200|300|ALL}
SYSTem:VERSion?
*TST?

IEEE 488.2 Common Commands

*CLS
*ESR?
*ESE <enable_value>
*ESE?
*IDN?
*OPC
*OPC?
*PSC {0|1}
*PSC?
*RST
*SAV {0|1|2|3|4|5}
*RCL {0|1|2|3|4|5}

*STB?
*SRE <enable_value>
*SRE?
*TRG
*TST?

Default parameters are shown in **bold**.
**Agilent 34901A  20-Channel Multiplexer**
(see page 164 in the User’s Guide)

**Agilent 34902A  16-Channel Multiplexer**
(see page 166 in the User’s Guide)
Agilent 34903A 20-Channel Actuator
(see page 168 in the User’s Guide)

Agilent 34904A 4x8 Matrix
(see page 170 in the User’s Guide)
Agilent 34905A/6A Dual 4-Channel RF Multiplexers
(see page 172 in the User’s Guide)

Agilent 34908A 40-Channel Single-Ended Multiplexer
(see page 174 in the User’s Guide)
Agilent 34907A Multifunction Module
(see page 176 in the User’s Guide)
## Factory Reset State

The table below shows the state of the instrument after a FACTORY RESET from the Sto/Rcl menu or *RST command from the remote interface.

<table>
<thead>
<tr>
<th>Measurement Configuration</th>
<th>Factory Reset State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>DC Volts</td>
</tr>
<tr>
<td>Range</td>
<td>Autorange</td>
</tr>
<tr>
<td>Resolution</td>
<td>5½ digits</td>
</tr>
<tr>
<td>Integration Time</td>
<td>1 PLC</td>
</tr>
<tr>
<td>Input Resistance</td>
<td>10 MΩ (fixed for all DCV ranges)</td>
</tr>
<tr>
<td>Channel Delay</td>
<td>Automatic Delay</td>
</tr>
<tr>
<td>Totalizer Reset Mode</td>
<td>Count Not Reset When Read</td>
</tr>
<tr>
<td>Totalizer Edge Detect</td>
<td>Rising Edge</td>
</tr>
<tr>
<td>Scanning Operations</td>
<td></td>
</tr>
<tr>
<td>Scan List</td>
<td>Empty</td>
</tr>
<tr>
<td>Reading Memory</td>
<td>All Readings are Cleared</td>
</tr>
<tr>
<td>Min, Max, and Average</td>
<td>All Statistical Data is Cleared</td>
</tr>
<tr>
<td>Scan Interval Source</td>
<td>Immediate</td>
</tr>
<tr>
<td>Scan Interval</td>
<td>Front Panel = 10 Seconds</td>
</tr>
<tr>
<td>Scan Count</td>
<td>Remote = Immediate</td>
</tr>
<tr>
<td>Scan Reading Format</td>
<td>Front Panel = Continuous</td>
</tr>
<tr>
<td>Monitor in Progress</td>
<td>Remote = 1 Scan Sweep</td>
</tr>
<tr>
<td>Mx+B Scaling</td>
<td></td>
</tr>
<tr>
<td>Gain Factor (&quot;M&quot;)</td>
<td>1</td>
</tr>
<tr>
<td>Scale Factor (&quot;B&quot;)</td>
<td>0</td>
</tr>
<tr>
<td>Scale Label</td>
<td>Vdc</td>
</tr>
<tr>
<td>Alarm Limits</td>
<td></td>
</tr>
<tr>
<td>Alarm Queue</td>
<td>Not Cleared</td>
</tr>
<tr>
<td>Alarm State</td>
<td>Off</td>
</tr>
<tr>
<td>HI and LO Alarm Limits</td>
<td>0</td>
</tr>
<tr>
<td>Alarm Output</td>
<td>Alarm 1</td>
</tr>
<tr>
<td>Alarm Output Configuration</td>
<td>Latched Mode</td>
</tr>
<tr>
<td>Alarm Output State</td>
<td>Output Lines are Cleared</td>
</tr>
<tr>
<td>Alarm Output Slope</td>
<td>Fail = Low</td>
</tr>
<tr>
<td>Module Hardware</td>
<td></td>
</tr>
<tr>
<td>34901A, 34902A, 34908A</td>
<td>Reset: All Channels Open</td>
</tr>
<tr>
<td>34903A, 34904A</td>
<td>Reset: All Channels Open</td>
</tr>
<tr>
<td>34905A, 34906A</td>
<td>Reset: Channels s11 and s21 Selected</td>
</tr>
<tr>
<td>34907A</td>
<td>Reset: Both DIO Ports = Input, Count = 0, Both DACs = 0 Vdc</td>
</tr>
<tr>
<td>System-Related Operations</td>
<td></td>
</tr>
<tr>
<td>Display State</td>
<td>On</td>
</tr>
<tr>
<td>Error Queue</td>
<td>Errors Not Cleared</td>
</tr>
<tr>
<td>Stored States</td>
<td>No Change</td>
</tr>
</tbody>
</table>
H.4 Power Supply
5
Introduction to Programming

SCPI Commands 68
SCPI Messages 70
SCPI Conventions and Data Formats 72
SCPI Command Completion 74
Device Clear 74

This chapter contains a brief introduction to the SCPI Programming language. SCPI (Standard Commands for Programmable Instruments) is a programming language for controlling instrument functions over the GPIB.

SCPI provides instrument control with a standardized command syntax and style, as well as a standardized data interchange format for various classes of instruments.
SCPI Commands

SCPI has two types of commands, common and subsystem.

- Common commands generally control overall power system functions, such as reset, status, and synchronization. All common commands consist of a three-letter mnemonic preceded by an asterisk: *RST  *IDN?  *SRE 8

- Subsystem commands perform specific power system functions. They are organized into an inverted tree structure with the "root" at the top. The following figure shows a portion of a subsystem command tree, from which you access the commands located along the various paths.

### Multiple Commands in a Message

Multiple SCPI commands can be combined and sent as a single message with one message terminator. There are two important considerations when sending several commands within a single message:

- Use a semicolon to separate commands within a message.

- There is an implied header path that affects how commands are interpreted by the power system.

The header path can be thought of as a string that gets inserted before each command within a message. For the first command in a message, the header path is a null string. For each subsequent command the header path is defined as the characters that make up the headers of the previous command in the message up to and including the last colon separator. An example of a message with two commands is:

```
OUTPut:STATe ON,(@1);PROTection:CLEar (@1)
```

which shows the use of the semicolon separating the two commands, and also illustrates the header path concept. Note that with the second command, the leading header "OUTPut" was omitted because after the "OUTPut:STATe ON" command, the header path became...
defined as "OUTPut" and thus the instrument interpreted the second command as:

```
OUTPut:PROTection:CLEar (@1)
```

In fact, it would have been syntactically incorrect to include the "OUTP" explicitly in the second command, since the result after combining it with the header path would be:

```
OUTPut:OUTPut:PROTection:CLEar (@1)
```

which is incorrect.

### Moving Among Subsystems

In order to combine commands from different subsystems, you need to be able to reset the header path to a null string within a message. You do this by beginning the command with a colon (:), which discards any previous header path. For example, you could clear the output protection and check the status of the Operation Condition register in one message by using a root specifier as follows (the short form is used in the next two examples):

```
OUTP:PROT:CLE(@1);:STAT:OPER:COND?(@1)
```

The following message shows how to combine commands from different subsystems as well as within the same subsystem:

```
VOLT:LEV 7.5,(@1);PROT 10,(@1);:CURR:LEV 0.25,(@1)
```

Note the use of the optional header LEVel to maintain the correct path within the subsystems, and the use of the root specifier to move between subsystems.

### Including Common Commands

You can combine common commands with system commands in the same message. Treat the common command as a message unit by separating it with a semicolon (the message unit separator). Common commands do not affect the header path; you may insert them anywhere in the message.

```
OUTPut OFF,(@1);*RCL 1;OUTPut ON,(@1)
```

### Using Queries

Observe the following precautions with queries:

- Add a blank space between the query indicator (?) and any subsequent parameter such as a channel list.
- Allocate the proper number of variables for the returned data.
- Read back all the results of a query before sending another command to the power system. Otherwise, a Query Interrupted error will occur and the unreturned data will be lost.
Coupled Commands

When commands are coupled it means that the value sent by one command is affected by the settings of another command. The following commands are coupled:

- `[SOURce:]CURRent` and `[SOURce:]CURRent:RANGe`.
- `[SOURce:]VOLTage` and `[SOURce:]VOLTage:RANGe`.

If a range command is sent that places an output on a range with a lower maximum setting than the present level, an error is generated. This also occurs if a level is programmed with a value too large for the present range.

These types of errors can be avoided by sending the both level and range commands as a set, in the same SCPI message. For example,

```
CURRent 10, (@1); CURRent : RANGe 10, (@1) <NL>
```

will always be correct because the commands are not executed until the message terminator is received. Because the range and setting information is received as a set, no range/setting conflict occurs.

SCPI Messages

There are two types of SCPI messages, program and response.

- A **program message** consists of one or more properly formatted SCPI commands sent from the controller to the power system. The message, which may be sent at any time, requests the power system to perform some action.

- A **response message** consists of data in a specific SCPI format sent from the power system to the controller. The power system sends the message only when commanded by a program message "query."

The following figure illustrates the SCPI message structure.
The Message Unit

The simplest SCPI command is a single message unit consisting of a command header (or keyword) followed by a message terminator. The message unit may include a parameter after the header. The parameter can be numeric or a string.

```
ABORt<NL>
VOLTage 20, (@1)
```

Channel List Parameter

The channel parameter is required to address one or more channels. It has the following syntax:

```
(@<channel> [,<channel>] [,<channel>] [,<channel>])
```

You can also specify a range of sequential channels as follows:

```
<start_channel> : <end_channel>
```

For example, (@2) specifies channel 2 and (@1:3) specifies channels 1 through 3. A maximum of 4 channels may be specified through a combination of single channels and ranges. Query results are channel list order-sensitive. Results are returned in the order they are specified in the list.

**NOTE**

When adding a channel list parameter to a query, you must include a space character between the query indicator (?) and the channel list parameter. Otherwise error –103, Invalid separator will occur.

Headers

Headers, also referred to as keywords, are instructions recognized by the power system. Headers may be in the long form or in the short form. In the long form, the header is completely spelled out, such as VOLTAGE, STATUS, and DELAY. In the short form, the header has only the first three or four letters, such as VOLT, STAT, and DEL.

When the long form notation is used in this document, the capital letters indicate the equivalent short form. For example, MEASURE is the long form, and MEAS indicates the short form equivalent.

Query Indicator

Following a header with a question mark turns it into a query (VOLTage?, VOLTage:TRIGgered?). The ? is the query indicator. If a query contains a parameter, place the query indicator at the end of the last header, before the parameter.

```
VOLTage:TRIGgered? MAX, (@1)
```
Message Unit Separator

When two or more message units are combined into a compound message, separate the units with a semicolon.

\texttt{STATus:OPERation?(@1);QUEStionable?(@1)}

Root Specifier

When it precedes the first header of a message unit, the colon becomes the root specifier. It tells the command parser that this is the root or the top node of the command tree.

Message Terminator

A terminator informs SCPI that it has reached the end of a message. The following messages terminators are permitted:

- newline <NL>, which is ASCII decimal 10 or hex 0A.
- end or identify <END> (EOI with ATN false)
- both of the above <NL><END>
- also <CR><NL>

In the examples of this guide, there is an assumed message terminator at the end of each message.

SCPI Conventions and Data Formats

Conventions

The following SCPI conventions are used throughout this guide.

\textbf{Angle brackets < >} Items within angle brackets are parameter abbreviations. For example, \texttt{<NR1>} indicates a specific form of numerical data.

\textbf{Vertical bar |} Vertical bars separate alternative parameters. For example, \texttt{VOLT | CURR} indicates that either "VOLT" or "CURR" can be used as a parameter.

\textbf{Square Brackets [ ]} Items within square brackets are optional. The representation \texttt{[SOURce:]VOLTage} means that \texttt{SOURce:} may be omitted.

\textbf{Braces } \{ \} Braces indicate parameters that may be repeated zero or more times. It is used especially for showing arrays. The notation \texttt{<A>{<,B>}} shows that parameter "A" must be entered, while parameter "B" may be omitted or may be entered one or more times.

\textbf{Parentheses ( )} Items within parentheses are used in place of the usual parameter types to specify a channel list. The notation \texttt{(@1,3)} specifies a channel list that includes channels 1, 2, and 3. The notation \texttt{(@1,3)} specifies a channel list that includes only channels 1 and 3.
Data Formats

Data programmed or queried from the power system is ASCII. The data may be numerical or character string.

**Numeric Data Formats**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Response Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;NR1&gt;</td>
<td>Digits with an implied decimal point assumed at the right of the least-significant digit. Examples: 273</td>
</tr>
<tr>
<td>&lt;NR2&gt;</td>
<td>Digits with an explicit decimal point. Example: .0273</td>
</tr>
<tr>
<td>&lt;NR3&gt;</td>
<td>Digits with an explicit decimal point and an exponent. Example: 2.73E+2</td>
</tr>
</tbody>
</table>

**Parameter Formats**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Response Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;NRf&gt;</td>
<td>Extended format that includes &lt;NR1&gt;, &lt;NR2&gt; and &lt;NR3&gt;. Examples: 273 273. 2.73E2</td>
</tr>
<tr>
<td>&lt;NRf+&gt;</td>
<td>Expanded decimal format that includes &lt;NRf&gt; and MIN MAX. Examples: 273 273. 2.73E2 MAX. MIN and MAX are the minimum and maximum limit values that are implicit in the range specification for the parameter.</td>
</tr>
<tr>
<td>&lt;Bool&gt;</td>
<td>Boolean Data. They can be numeric (0, 1), or named (ON, OFF).</td>
</tr>
<tr>
<td>&lt;SPD&gt;</td>
<td>String program data. String parameters enclosed in single or double quotes.</td>
</tr>
</tbody>
</table>

**Suffixes and Multipliers**

<table>
<thead>
<tr>
<th>Class</th>
<th>Suffix</th>
<th>Unit</th>
<th>Unit with Multiplier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current</td>
<td>A</td>
<td>ampere</td>
<td>MA (milliampere)</td>
</tr>
<tr>
<td>Amplitude</td>
<td>V</td>
<td>volt</td>
<td>MV (millivolt)</td>
</tr>
<tr>
<td>Time</td>
<td>S</td>
<td>second</td>
<td>MS (millisecond)</td>
</tr>
</tbody>
</table>

**Common Multipliers**

- 1E3 K kilo
- 1E-3 M milli
- 1E-6 U micro

**Response Data Types**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Response Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;CRD&gt;</td>
<td>Character Response Data. Permits the return of character strings.</td>
</tr>
<tr>
<td>&lt;AARD&gt;</td>
<td>Arbitrary ASCII Response Data. Permits the return of undelimited 7-bit ASCII. This data type has an implied message terminator.</td>
</tr>
<tr>
<td>&lt;SRD&gt;</td>
<td>String Response Data. Returns string parameters enclosed in single or double quotes.</td>
</tr>
</tbody>
</table>
SCPI Command Completion

SCPI commands sent to the power system are processed either sequentially or in parallel. Sequential commands finish execution before a subsequent command begins. Parallel commands allow other commands to begin executing while the parallel command is still executing.

The following is a list of parallel commands. You should use some form of command synchronization as discussed in this section before assuming that these commands have completed.

```
OUTPUT:STATE INITIATE
VOLT OUTPUT:PROTECTION:CLEAR
CURR FUNC:MODE
```

The *WAI, *OPC, and *OPC? common commands provide different ways of indicating when all transmitted commands, including any parallel ones, have completed their operations. Some practical considerations for using these commands are as follows:

*WAI

This command prevents the power system from processing subsequent commands until all pending operations are completed. For example, the *WAI command can be used to make a current measurement after an output on command has completed:

```
OUTPUT ON,(@1);*WAI;:MEAS:CURR? (@1)
```

*OPC?

This command places a 1 in the Output Queue when all pending operations have completed. Because it requires your program to read the returned value before executing the next program statement, *OPC? can be used to cause the controller to wait for commands to complete before proceeding with its program.

*OPC

This command sets the OPC status bit when all pending operations have completed. Since your program can read this status bit on an interrupt basis, *OPC allows subsequent commands to be executed.

NOTE

The trigger subsystem must be in the Idle state for the status OPC bit to be true. As far as triggers are concerned, OPC is false whenever the trigger subsystem is in the Initiated state.

Device Clear

You can send a Device Clear at any time to abort a SCPI command that may be hanging up the GPIB interface. Device Clear clears the input and output buffers of the power system and prepares the power system to accept a new command string. The status registers, error queue, and all configuration states are left unchanged by Device Clear. The following statement shows how to send a device clear over the GPIB interface using Agilent BASIC:

```
CLEAR 705 IEEE-488 Device Clear
```
This section gives the syntax and parameters for all the IEEE 488.2 SCPI commands and the Common commands used by the power system. It is assumed that you are familiar with the material in chapter 5, which explains the terms, symbols, and syntactical structures used here and gives an introduction to programming. You should also be familiar with chapter 4, in order to understand how the power system functions.

**Subsystem commands** are specific to instrument functions. They can be a single command or a group of commands. The groups are comprised of commands that extend one or more levels below the root. The subsystem commands are arranged alphabetically according to the function they perform.

**Common commands** are defined by the IEEE 488.2 standard to perform common interface functions. They begin with an * and consist of three letters (command) or three letters and a ? (query). Common commands are grouped along with the subsystem commands according to the function they perform.
### Subsystem Commands

**NOTE** Some [optional] commands have been included for clarity. Not all commands apply to all models.

<table>
<thead>
<tr>
<th>SCPI Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ABORt</strong></td>
<td>Resets the measurement trigger system to the Idle state</td>
</tr>
<tr>
<td>:ACQuire (@chanlist)</td>
<td></td>
</tr>
<tr>
<td>:TRANsient (@chanlist)</td>
<td>Resets the transient trigger system to the Idle state</td>
</tr>
<tr>
<td><strong>CALibrate</strong></td>
<td>Calibrates the output current programming</td>
</tr>
<tr>
<td>:CURRent</td>
<td>Calibrates the current measurement</td>
</tr>
<tr>
<td>[:LEVel] &lt;NRf&gt;, (@channel)</td>
<td>Calibrates the peak current limit (Agilent N6751A/52A/61A/62A)</td>
</tr>
<tr>
<td>:MEASure &lt;NRf&gt;, (@channel)</td>
<td>Calibrates the current measurement</td>
</tr>
<tr>
<td>:PEAK (@channel)</td>
<td>Enters the calibration value</td>
</tr>
<tr>
<td><strong>DATA &lt;NRf&gt;</strong></td>
<td>Sets the calibration date</td>
</tr>
<tr>
<td><strong>DATE &lt;SPD&gt;, (@channel)</strong></td>
<td>Calibrates the current downprogrammer</td>
</tr>
<tr>
<td><strong>DPReg (@channel)</strong></td>
<td>Advances to the next calibration step</td>
</tr>
<tr>
<td>:LEVel P1</td>
<td>P2</td>
</tr>
<tr>
<td>:PASSword &lt;NRf&gt;</td>
<td>Saves the new cal constants in non-volatile memory</td>
</tr>
<tr>
<td>:SAVE</td>
<td>Enables/disables calibration mode</td>
</tr>
<tr>
<td>:STATE &lt;Bool&gt; [.&lt;NRf&gt;]</td>
<td>Calibrates the output voltage programming</td>
</tr>
<tr>
<td><strong>DATA &lt;NRf&gt;</strong></td>
<td>Calibrates common mode rejection ratio (N6751A/52A/61A/62A)</td>
</tr>
<tr>
<td><strong>VOLTage</strong></td>
<td>Calibrates the voltage measurement</td>
</tr>
<tr>
<td>[:LEVel] &lt;NRf&gt;, (@channel)</td>
<td>Selects 1-channel or 4-channel meter view</td>
</tr>
<tr>
<td>:CMRR (@channel)</td>
<td></td>
</tr>
<tr>
<td>:MEASure &lt;NRf&gt;, (@channel)</td>
<td></td>
</tr>
<tr>
<td>**DISPlay[:WINDow]:VIEW METER1</td>
<td>METER4**</td>
</tr>
<tr>
<td>**FETCh (Note 1)</td>
<td>MEASure**</td>
</tr>
<tr>
<td>[:SCALar]</td>
<td>Returns the average output voltage</td>
</tr>
<tr>
<td>:CURRent [:DC]؟ (@chanlist)</td>
<td>(Array commands only on Agilent N6761A/62A and Option 054)</td>
</tr>
<tr>
<td>:VOLTage [:DC]؟ (@chanlist)</td>
<td>Returns the instantaneous output current</td>
</tr>
<tr>
<td>:ARRay</td>
<td>Returns the instantaneous output voltage</td>
</tr>
<tr>
<td>:CURRent [:DC]؟ (@chanlist)</td>
<td></td>
</tr>
<tr>
<td>:VOLTage [:DC]؟ (@chanlist)</td>
<td></td>
</tr>
<tr>
<td><strong>INITiate</strong></td>
<td></td>
</tr>
<tr>
<td>[:IMMediate]</td>
<td>(Acquire command only on Agilent N6761A/62A and Option 054)</td>
</tr>
<tr>
<td>:ACQuire (@chanlist)</td>
<td>Enables the measurement system to receive triggers</td>
</tr>
<tr>
<td>:TRANsient (@chanlist)</td>
<td>Enables the output transient system to receive triggers</td>
</tr>
<tr>
<td>:CONTinuous</td>
<td>Enables/disables continuous transient triggers</td>
</tr>
<tr>
<td>:TRANsient &lt;Bool&gt;, (@chanlist)</td>
<td></td>
</tr>
<tr>
<td><strong>OUTPut</strong></td>
<td>Enables/disables the specified output channel(s)</td>
</tr>
<tr>
<td>[:STATe] &lt;Bool&gt;. [NORelay], (@chanlist)</td>
<td>Sets the output turn-off sequence delay</td>
</tr>
<tr>
<td>:DELaY</td>
<td>Sets the output turn-on sequence delay</td>
</tr>
<tr>
<td>:FALL &lt;NRf++&gt;, (@chanlist)</td>
<td></td>
</tr>
<tr>
<td>:RISE &lt;NRf++&gt;, (@chanlist)</td>
<td></td>
</tr>
<tr>
<td>:INHibit</td>
<td>Sets the remote inhibit input</td>
</tr>
<tr>
<td>:MODE LATChing</td>
<td>LIVE</td>
</tr>
<tr>
<td>SCPI Command</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>OUTPut (continued)</td>
<td></td>
</tr>
<tr>
<td>:PON</td>
<td>Programs the power-on state</td>
</tr>
<tr>
<td>:STATe RST</td>
<td>RCL0</td>
</tr>
<tr>
<td>:PROTection</td>
<td></td>
</tr>
<tr>
<td>:CLear</td>
<td>(@chanlist)</td>
</tr>
<tr>
<td>:COUPle</td>
<td>&lt;Bool&gt;</td>
</tr>
<tr>
<td>:DELay &lt;NRf+&gt;, (@chanlist)</td>
<td>Sets over-current protection programming delay</td>
</tr>
<tr>
<td>SENSE</td>
<td></td>
</tr>
<tr>
<td>:CURRent [:DC]</td>
<td></td>
</tr>
<tr>
<td>:RANGE [:UPPer] &lt;NRf+&gt;, (@chanlist)</td>
<td>Selects the current measurement range (Agilent N6761A/62A)</td>
</tr>
<tr>
<td>:FUNCTION &quot;VOLTage&quot;</td>
<td>&quot;CURRent&quot;, (@chanlist)</td>
</tr>
<tr>
<td>:SWEeep</td>
<td></td>
</tr>
<tr>
<td>:OFFSET:POINts &lt;NRf+&gt;, (@chanlist)</td>
<td></td>
</tr>
<tr>
<td>:POINts &lt;NRf+&gt;, (@chanlist)</td>
<td>Defines the number of data points in the measurement</td>
</tr>
<tr>
<td>:TINTerval &lt;NRf+&gt;, (@chanlist)</td>
<td>Sets the measurement sample interval</td>
</tr>
<tr>
<td>:VOLTage [:DC]</td>
<td></td>
</tr>
<tr>
<td>:RANGE [:UPPer] &lt;NRf+&gt;, (@chanlist)</td>
<td>Selects the voltage measurement range (Agilent N6761A/62A)</td>
</tr>
<tr>
<td>:WINDow [:TYPE] HANNing</td>
<td>RECTangular, (@chanlist)</td>
</tr>
<tr>
<td>[SOURce:]</td>
<td>CURRent</td>
</tr>
<tr>
<td>[:IMMediate][:AMPLitude] &lt;NRf+&gt;, (@chanlist)</td>
<td>Sets the output current</td>
</tr>
<tr>
<td>:TRIGgered [:AMPLitude] &lt;NRf+&gt;, (@chanlist)</td>
<td>Sets the triggered output current</td>
</tr>
<tr>
<td>:MODE FIXed</td>
<td>STEP</td>
</tr>
<tr>
<td>:PROTection</td>
<td></td>
</tr>
<tr>
<td>:STATe &lt;Bool&gt;, (@chanlist)</td>
<td>Enables/disables over-current protection on the selected output</td>
</tr>
<tr>
<td>:RANGE &lt;NRf+&gt;, (@chanlist)</td>
<td>Sets the output current range (Agilent N6761A/62A)</td>
</tr>
<tr>
<td>DIGItal</td>
<td></td>
</tr>
<tr>
<td>:INPut:DATA?</td>
<td>Reads the state of the digital port pins</td>
</tr>
<tr>
<td>:OUTPut:DATA &lt;NRf&gt;</td>
<td>Sets the digital port</td>
</tr>
<tr>
<td>:FUNCTION DIO</td>
<td>DINP</td>
</tr>
<tr>
<td>:POLarity POSitive</td>
<td>NEGative</td>
</tr>
<tr>
<td>:LIST</td>
<td></td>
</tr>
<tr>
<td>:COUNT &lt;NRf+&gt;</td>
<td>INFinity, (@chanlist)</td>
</tr>
<tr>
<td>:CURRent [:LEVEL] &lt;NRf&gt; {,&lt;NRf&gt;...}, (@chanlist)</td>
<td>Sets the current list</td>
</tr>
<tr>
<td>:POINts? (@chanlist)</td>
<td>Returns the number of current list points</td>
</tr>
<tr>
<td>:DWELI &lt;NRf&gt; {,&lt;NRf&gt;...}, (@chanlist)</td>
<td>Sets the list of dwell times</td>
</tr>
<tr>
<td>:POINts? (@chanlist)</td>
<td>Returns the number of dwell list points</td>
</tr>
<tr>
<td>:STEP ONCE</td>
<td>AUTO, (@chanlist)</td>
</tr>
<tr>
<td>:TERMinate</td>
<td></td>
</tr>
<tr>
<td>:LAST &lt;Bool&gt;, (@chanlist)</td>
<td>Sets the list termination mode</td>
</tr>
<tr>
<td>:OUTput</td>
<td></td>
</tr>
<tr>
<td>:BOSTep[:DATA] &lt;Bool&gt; {,&lt;Bool&gt;...}, (@chanlist)</td>
<td>Sets the steps to generate triggers at the Begin Of Step</td>
</tr>
<tr>
<td>:POINts? (@chanlist)</td>
<td>Returns the number of beginning of step list points</td>
</tr>
<tr>
<td>:EOSTep[:DATA] &lt;Bool&gt; {,&lt;Bool&gt;...}, (@chanlist)</td>
<td>Sets the steps to generate triggers at the End Of Step</td>
</tr>
<tr>
<td>:POINts? (@chanlist)</td>
<td>Returns the number of end of step list points</td>
</tr>
<tr>
<td>:VOLTage[:LEVEL] &lt;NRf&gt; {,&lt;NRf&gt;...}, (@chanlist)</td>
<td>Sets the voltage list</td>
</tr>
<tr>
<td>:POINts? (@chanlist)</td>
<td>Returns the number of voltage level points</td>
</tr>
<tr>
<td>STEP</td>
<td></td>
</tr>
<tr>
<td>:TOUTput &lt;Bool&gt;, (@chanlist)</td>
<td>Generate a trigger output on the voltage or current step transient</td>
</tr>
</tbody>
</table>
### SCPI Command Summary

<table>
<thead>
<tr>
<th>SCPI Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[SOURce:][ continued]</td>
<td></td>
</tr>
<tr>
<td>VOLTage</td>
<td></td>
</tr>
<tr>
<td>[:LEVel]</td>
<td>Sets the output voltage</td>
</tr>
<tr>
<td>: IMMediate [:AMPLitude] &lt;NRf&gt;, (@chanlist)</td>
<td>Sets the triggered output voltage</td>
</tr>
<tr>
<td>:TRIggered [:AMPLitude] &lt;NRf&gt;, (@chanlist)</td>
<td></td>
</tr>
<tr>
<td>:MODE FIXed</td>
<td>Sets the voltage trigger mode</td>
</tr>
<tr>
<td>:STEP</td>
<td></td>
</tr>
<tr>
<td>:LIST, (@chanlist)</td>
<td></td>
</tr>
<tr>
<td>:PROTection</td>
<td>Sets the over-voltage protection level</td>
</tr>
<tr>
<td>[:LEVel] &lt;NRf&gt;, (@chanlist)</td>
<td>Sets the output voltage range (Agilent N6761A/62A)</td>
</tr>
<tr>
<td>:RANGe &lt;NRf&gt;, (@chanlist)</td>
<td>Sets the output voltage slew rate</td>
</tr>
<tr>
<td>:SLEW &lt;NRf&gt;</td>
<td></td>
</tr>
<tr>
<td>INFinity, (@chanlist)</td>
<td></td>
</tr>
</tbody>
</table>

| STATus | |
| :OPERation | |
| [:EVENT]? (@chanlist) | Returns the value of the operation event register |
| :CONDition? (@chanlist) | Returns the value of the operation condition register |
| :ENABLE <NRf>, (@chanlist) | Enables specific bits in the Event register |
| :NTTransition <NRf>, (@chanlist) | Sets the Negative transition filter |
| :PTRTransition <NRf>, (@chanlist) | Sets the Positive transition filter |
| :PRESet | Presets all enable and transition registers to power-on |

| :QUESTIONable | |
| [:EVENT]? (@chanlist) | Returns the value of the questionable event register |
| :CONDition? (@chanlist) | Returns the value of the questionable condition register |
| :ENABLE <NRf>, (@chanlist) | Enables specific bits in the Event register |
| :NTTransition <NRf>, (@chanlist) | Sets the Negative transition filter |
| :PTRTransition <NRf>, (@chanlist) | Sets the Positive transition filter |

| SYSTem | |
| :CHANnel | |
| [:COUNT]? | Returns the number of output channels in a mainframe |
| :MODEL? (@chanlist) | Returns the model number of the selected channel |
| :OPTION? (@chanlist) | Returns the option installed in the selected channel |
| :SERial? (@chanlist) | Returns the serial number of the selected channel |

| :COMMunicate | |
| :RLSTate LOCal | Specifies the Remote/Local state of the instrument |
| REMote | |
| RWLock | |
| :TCPip:CONTrol? | Returns the control connection port number |
| :ERRor? | Returns the error number and error string |
| :GROup | |
| :CATalog? | Returns the groups that have been defined |
| :DEFine (@chanlist) | Group multiple channels together to create a single output |
| :DELete (channel) | Removes the specified channel from a group |
| :ALL | Ungroups all channels |
| :PASSword:FPANel:RESet | Resets the front panel lock password to zero |
| :REBoot | Returns the unit to its power-on state |
| :VERSion? | Returns the SCPI version number |

| TRIGger | |
| :ACQuire | (Acquire commands only on Agilent N6761A/62A and Option 054) |
| [:JMMediate] (@chanlist) | Triggers the measurement immediately |
| :SOURce BUS | Sets the measurement trigger source |
| PIN<pin> | |
| TRAN<chan>, (@chanlist) | |
| :TRANSient | Triggers the output immediately |
| [:JMMediate] (@chanlist) | Sets the output trigger source |
| :SOURce BUS | |
| PIN<pin> | |
| TRAN<chan>, (@chanlist) | |
# Common Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*CLS</td>
<td>Clear status</td>
<td>*RDT?</td>
<td>Return output channel descriptions</td>
</tr>
<tr>
<td>*ESE &lt;NRf&gt;</td>
<td>Standard event status enable</td>
<td>*RST</td>
<td>Reset</td>
</tr>
<tr>
<td>*ESE?</td>
<td>Return standard event status enable</td>
<td>*SAV &lt;NRf&gt;</td>
<td>Saves an instrument state</td>
</tr>
<tr>
<td>*ESR?</td>
<td>Return event status register</td>
<td>*SRE &lt;NRf&gt;</td>
<td>Set service request enable register</td>
</tr>
<tr>
<td>*IDN?</td>
<td>Return instrument identification</td>
<td>*SRE?</td>
<td>Return service request enable register</td>
</tr>
<tr>
<td>*OPC</td>
<td>Enable &quot;operation complete&quot; bit in ESR</td>
<td>*STB?</td>
<td>Return status byte</td>
</tr>
<tr>
<td>*OPC?</td>
<td>Return a &quot;1&quot; when operation complete</td>
<td>*TRG</td>
<td>Trigger</td>
</tr>
<tr>
<td>*OPT?</td>
<td>Return option number</td>
<td>*TST?</td>
<td>Performs self-test, then returns result</td>
</tr>
<tr>
<td>*RCL &lt;NRf&gt;</td>
<td>Recalls a saved instrument state</td>
<td>*WAI</td>
<td>Pauses additional command processing until all device commands are done</td>
</tr>
</tbody>
</table>

## *RST Settings

These settings are set by the *RST (Reset) command

<table>
<thead>
<tr>
<th>Calibration Function (Note 1)</th>
<th>Measurement (continued)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAL:STAT OFF</td>
<td>SENS:SWEP:OFFS:POIN 0</td>
</tr>
<tr>
<td>Current Function</td>
<td>SENS:SWEP:TINT 20.48E–6</td>
</tr>
<tr>
<td>[SOUR:]CURR 80 mA</td>
<td>SENS:VOLT:RANG MAX</td>
</tr>
<tr>
<td>[SOUR:]CURR:MODE FIX</td>
<td>SENS:WIND RECT</td>
</tr>
<tr>
<td>[SOUR:]CURR:PROT:STAT OFF</td>
<td>Output Function</td>
</tr>
<tr>
<td>[SOUR:]CURR:RANG MAX</td>
<td>OUTP OFF</td>
</tr>
<tr>
<td>[SOUR:]CURR:TRIG MIN</td>
<td>OUTP:DEL:FALL 0</td>
</tr>
<tr>
<td>Digital Function</td>
<td>OUTP:DEL:RISE 0</td>
</tr>
<tr>
<td>[SOUR:]DIG:OUTP:DATA 0</td>
<td>OUTP:PROT:COUP OFF</td>
</tr>
<tr>
<td>Display Function</td>
<td>OUTP:PROT:DEL 0.02</td>
</tr>
<tr>
<td>DISP:VIEW METERI</td>
<td>OUTP:REL OFF</td>
</tr>
<tr>
<td>List Function (Note 1)</td>
<td>Step Function</td>
</tr>
<tr>
<td>[SOUR:]LIST:COUN 1</td>
<td>[SOUR:]STEP:TOUT FALSE</td>
</tr>
<tr>
<td>[SOUR:]LIST:CURR MIN</td>
<td>Trigger Function</td>
</tr>
<tr>
<td>[SOUR:]LIST:DWEL 0.001</td>
<td>INIT:CONT:TRAN OFF</td>
</tr>
<tr>
<td>[SOUR:]LIST:STEP AUTO</td>
<td>TRIG:ACQ:SOUR BUS</td>
</tr>
<tr>
<td>[SOUR:]LIST:TERM:LAST OFF</td>
<td>TRIG:TRAN:SOUR BUS</td>
</tr>
<tr>
<td>[SOUR:]LIST:TOUT:BOST OFF</td>
<td>Voltage Function</td>
</tr>
<tr>
<td>[SOUR:]LIST:TOUT:EOST OFF</td>
<td>[SOUR:]VOLT MIN</td>
</tr>
<tr>
<td>[SOUR:]LIST:VOLT MIN</td>
<td>[SOUR:]VOLT:MODE FIX</td>
</tr>
<tr>
<td>Measurement Function</td>
<td>[SOUR:]VOLT:PROT:LEV MAX</td>
</tr>
<tr>
<td>SENS:CURR:RANG MAX</td>
<td>[SOUR:]VOLT:RANG MAX</td>
</tr>
<tr>
<td>SENS:FUNC &quot;VOLT&quot;</td>
<td>[SOUR:]VOLT:SLEW MAX</td>
</tr>
<tr>
<td>SENS:SWEP:POIN 1024</td>
<td>[SOUR:]VOLT:TRIG MIN</td>
</tr>
</tbody>
</table>

**Note 1** The calibration state and all list settings are not saved by the *SAV command.
The calibration subsystem lets you calibrate the power system. Only one channel can be calibrated at a time. Refer to Appendix B for details.

**NOTE**

If calibration mode has not been enabled with CALibrate:STATe, the calibration commands will generate an error. Use CALibrate:SAVE to save any changes, otherwise all changes will be lost when you exit calibration mode.

**CALibrate:CURRent[:LEVel] <value>, (@<channel>)**

This command initiates calibration of the output current. The value that you enter selects the range that is being calibrated.

**CALibrate:CURRent:MEASure <value>, (@<channel>)**

This command initiates calibration of the current measurement range. The value that you enter selects the range that is being calibrated.

**CALibrate:CURRent:PEAK (@<channel>)**

This command initiates calibration of the peak current limit.

**CALibrate:DATA <value>**

This command enters a calibration value that you obtain by reading an external meter. You must first select a calibration level (with CALibrate:LEVel) for the value being entered. Data values are expressed in base units - either volts or amperes, depending on which function is being calibrated.

**CALibrate:DATE <date>, (@<channel>)**

**CALibrate:DATE?**

This command stores the date that the power module was last calibrated. The calibration date is stored in nonvolatile memory. Enter any ASCII string up to 16 characters. The query returns the date.

**NOTE**

The firmware does not interpret the string format. The information is not used by the firmware. The command is only provided to store the calibration date.

**CALibrate:DPRog (@<channel>)**

This command initiates calibration of the current downprogrammer.
CALibrate:LEVel \{P1|P2|P3\}

This command is used to advance to the next level in the calibration. \textbf{P1} is the first calibration level; \textbf{P2} is the second level; \textbf{P3} is the third level.

\textbf{NOTE}

Some calibration sequences may require some settling time after sending CAL:LEV but before reading the data from the DVM and sending CAL:DATA.

CALibrate:PASSword <password>

This command lets you change the calibration password. A new password is automatically stored in nonvolatile memory and does not have to be stored with CALibrate:SAVE. If the password is set to 0, password protection is removed and the ability to enter calibration mode is unrestricted. The factory-default password 0 (zero).

CALibrate:SAVE

This command saves calibration constants in non-volatile memory after the calibration procedure has been completed. If calibration mode is exited by programming CALibrate:STATE OFF without first saving the new constants, the previous constants are restored.

CALibrate:STATE \{ON|OFF\} [,<password>]

This command enables or disables calibration mode. Calibration mode must be enabled for the power system to accept any calibration commands. The first parameter specifies the ON (1) or OFF (0) state. The second parameter is the password.

A numeric password is required if calibration mode is being enabled and the existing password is not 0. If the password is not entered or is incorrect, an error is generated and the calibration mode remains disabled. The query returns only the state, not the password.

The *RST value = OFF.

\textbf{NOTE}

When the calibration state is changed from enabled to disabled, new calibration constants are lost unless they have already been stored with CALibrate:SAVE.

CALibrate:VOLTage[:LEVel] <value>, (@<channel>)

This command initiates calibration of the output voltage. The value that you enter selects the range that is being calibrated.

CALibrate:VOLTage:CMRR (@<channel>)

This command initiates calibration of the voltage common mode rejection ratio.
**Display Subsystem**

**CALibrate:VOLTage:MEASure <value>, (@<channel>)**

This command initiates calibration of the voltage measurement range. The value that you enter selects the range that is being calibrated.

**Display Subsystem**

The display subsystem lets you control the front panel display.

**DISPlay[:WINDow]:VIEW {METER1|METER4}**

**DISPlay[:WINDow]:VIEW?**

This command selects the output channel view of the front panel display. METER1 displays one output channel. METER4 displays all output channels up to a maximum of four.

The *RST value = METER1.*
Measurement Subsystem

The measurement subsystem consists of Measure, Fetch, and Sense commands.

**Measure** commands measure the output voltage or current. They trigger the acquisition of new data before returning the reading. Measurements are performed by digitizing the instantaneous output voltage or current for a specified time interval, storing the results in a buffer, and calculating the average value. Use Measure commands when the measurement does not need to be synchronized with any other event.

**Fetch** commands return a reading computed from previously acquired data. If you take a voltage measurement, you can fetch only voltage data. If you take a current measurement, you can fetch only current data. Use Fetch commands when it is important that the measurement be synchronized with a triggered event.

**Sense** commands control the current measurement range, the bandwidth detector of the power system, and the data acquisition sequence.

Agilent Models N6761A and N6762A have simultaneous voltage and current measurement capability. In this case BOTH voltage and current are acquired, regardless of the parameter that is being measured. To return both values of a simultaneous measurement, first use the MEASure command to measure either the output voltage or current. Then use the FETCh command to return the other parameter.

**NOTE**

The FETCh:ARRay, MEASure:ARRay, and SENSE commands do not apply to all models (Refer to chapter 1, “Model Differences”).

FETCh:ARRay:CURRent[:DC]? (@<chanlist>)  
FETCh:ARRay:VOLTage[:DC]? (@<chanlist>)

MEASure:ARRay:CURRent[:DC]? (@<chanlist>)  
MEASure:ARRay:VOLTage[:DC]? (@<chanlist>)

These queries return an array containing the digitized output current in amperes or output voltage in volts. The data returned by the FETCh command is the result of the last measurement command or acquisition trigger. The data is valid until the next MEASure or INITiate command occurs.

The output voltage or current is digitized whenever a measurement command is sent or an acquisition trigger occurs. The sampling rate is set by SENSE:SWEep:TINTerval. The position of the trigger relative to the beginning of the data buffer is determined by SENSE:SWEep:OFFSet. The number of points returned is set by SENSE:SWEep:POINts.
**FETCh[:SCALar]:CURRent[:DC]? (@<chanlist>)**
**FETCh[:SCALar]:VOLTage[:DC]? (@<chanlist>)**

**MEASure[:SCALar]:CURRent[:DC]? (@<chanlist>)**
**MEASure[:SCALar]:VOLTage[:DC]? (@<chanlist>)**

These queries return the DC output current in amperes or output voltage in volts. The data returned by the FETCh command is the result of the last acquisition. The data is valid until the next MEASure or INITiate command occurs.

The output voltage or current is digitized whenever a measurement command is sent or an acquisition trigger occurs. The time interval is set by SENSE:SWEep:TINTerval. The position of the trigger relative to the beginning of the data buffer is determined by SENSE:SWEep:OFFSet. The number of points returned is set by SENSE:SWEep:POINts.

**SENSe:CURRent[:DC]:RANGe[:UPPer] {<value>|MIN|MAX}, (@<chanlist>)**
**SENSe:CURRent[:DC]:RANGe[:UPPer]? (@<chanlist>)**

This command selects a DC current measurement range on models that have multiple ranges. The value that you enter must be higher than the maximum current that you expect to measure. Units are in amperes. The instrument selects the range with the best resolution for the value entered. When queried, the returned value is the maximum DC current that can be measured on the range that is presently set.

Refer to Appendix A for the available ranges for each model.

The *RST value = the highest available range.*

**SENSe:FUNCTION {“VOLTage”|“CURRent”}, (@<chanlist>)**
**SENSe:FUNCTION? (@<chanlist>)**

This command selects a measurement function on models that do not have simultaneous voltage and current measurement capability. This command is required so that the acquisition system knows which measurement function to acquire when a measurement is triggered.

The *RST value = “VOLTage”.*

**SENSe:SWEep:OFFSet:POINts {<points>|MIN|MAX}, (@<chanlist>)**
**SENSe:SWEep:OFFSet:POINts? (@<chanlist>)**

This command defines the offset in a data sweep when an acquire trigger is used on models that have measurement controls. Programmed values can range from -4095 through 2,000,000,000 (2E9). Negative values represent data samples taken prior to the trigger. Positive values represent the delay after the trigger occurs but before the samples are acquired.

The *RST value = 0.*
SENSe:SWEep:POINts \{<points>|MIN|MAX\}, (@<chanlist>)
SENSe:SWEep:POINts? (@<chanlist>)

This command defines the number of points in a measurement on models that have measurement controls. Programmed values can range from 1 to 4096.

The *RST value = 1024.

SENSe:SWEep:TINTerval \{<interval>|MIN|MAX\}, (@<chanlist>)
SENSe:SWEep:TINTerval? (@<chanlist>)

This command defines the time period between samples in seconds on models that have measurement controls. Programmed values can range from 0.00002048 to 40000 seconds. Values are rounded to the nearest 20.48 microsecond increment.

The *RST value = 20.48 microseconds.

SENSe:VOLTage[:DC]:RANGe[:UPPer] \{<value>|MIN|MAX\}, (@<chanlist>)
SENSe:VOLTage[:DC]:RANGe[:UPPer]? (@<chanlist>)

This command selects a DC voltage measurement range on models that have multiple ranges. The programmed value must be the maximum voltage that you expect to measure. Units are in volts. The instrument selects the range with the best resolution for the value entered. When queried, the returned value is the maximum DC voltage that can be measured on the range that is presently set.

Refer to Appendix A for the available ranges for each model.

The *RST value = the highest available range.

SENSe:WINDow[:TYPE] \{HANNing|RECTangular\}, (@<chanlist>)
SENSe:WINDow[:TYPE]? (@<chanlist>)

This command sets the window function used in DC measurement calculations on models that have measurement controls. Select from:

- **HANNing**: A signal conditioning window that reduces errors in DC measurement calculations in the presence of periodic signals such as AC line ripple. The Hanning window multiplies each point in the sample by the function cosine^4.

- **RECTangular**: A window that returns measurement calculations with no signal conditioning.

Note that neither window function alters the instantaneous voltage or current data returned in the measurement array.

The *RST value = RECTangular.
Output Subsystem

The Output subsystem controls the output, power-on, protection, and relay functions.

OUTPut[:STATe] {ON|OFF}, [NORelay], (@<chanlist>)
OUTPut[:STATe]? (@<chanlist>)

This command enables or disables the specified output channel(s). The enabled state is ON (1); the disabled state is OFF (0). The state of a disabled output is a condition of zero output voltage and a zero source current. If output and sense relays are installed (Option 761), they will open when the output is disabled and close when the output is enabled. The query returns 0 if the output is off, and 1 if the output is on.

The NORelay optional parameter lets you turn the output state on or off and leave the state of the relays unchanged. When not specified, the relays open and close as the output is turned off and on.

Separate delays can be programmed for the off-to-on and the on-to-off transition using OUTPut:DELay:RISE and OUTput:DELay:FALL.

The *RST value = OFF.

NOTE
Because of internal circuit start-up procedures and any installed relay options, the output on command may take between 35 and 50 milliseconds to complete its function. Conversely, the output off command may take between 20 and 25 milliseconds to complete its function. To mitigate this built-in delay, you can program the output to zero volts rather than using the output on/off command.

OUTPut[:STATe]:DELay:FALL {<delay>|MIN|MAX}, (@<chanlist>)
OUTPut[:STATe]:DELay:FALL? (@<chanlist>)

This command sets the delay in seconds that the instrument waits before disabling the specified output. It affects on-to-off transitions including changes in the OUTPut:STATe as well as transitions due to changes in the voltage range or current range. It does NOT affect transitions to off caused by protection functions. Delay times can range from 0 to 1.023 seconds in increments of 1 millisecond.

This command allows multiple output channels to turn off in sequence. Each output will not turn off until its delay time has elapsed.

The *RST value = 0.

NOTE
Output channel turn-on and turn-off characteristics vary across the three module types - DC Power, Autoranging, and Precision (Refer to chapter 1, “Model Differences”). When several channels of the same module type are programmed by this command, output sequencing is precisely determined by the programmed delays.
However, when outputs of different module types are sequenced using this command, there may be an additional offset of a few milliseconds from one output to another. This offset is the same for each module type and is repeatable. Once you have characterized this offset, using an oscilloscope for example, you can adjust the programmed delays to compensate for the offset and give the desired output sequencing.

Outputs within the same module type can also have an offset if one model has output relays (Option 761) and another does not. These offsets are also repeatable and can be compensated for by adjusting the programmed delay values.

```
OUTPut[:STATE]:DELay:RISE {<delay>|MIN|MAX}, (@<chanlist>)
OUTPut[:STATE]:DELay:RISE? (@<chanlist>)
```

This command sets the delay in seconds that the instrument waits before enabling the specified output. It affects all off-to-on transitions including changes in the OUTPut:STATe as well as transitions due to OUTPut:PROTection:CLEar. Delay times can range from 0 to 1.023 seconds in increments of 1 millisecond.

This command allows multiple output channels to turn on in sequence. Each output will not turn on until its delay time has elapsed.

The *RST value = 0.

NOTE Refer to the note under OUTPut:DELay:FALL, which also applies to OUTPut:DELay:RISE.

```
OUTPut:INHibit:MODE {LATChing|LIVE|OFF}
OUTPut:INHibit:MODE?
```

This command selects the mode of operation of the Inhibit input (INH). The inhibit function shuts down ALL output channels in response to an external signal on the Inhibit input. If an output channel has been turned off by OUTPut:STATe, the inhibit function does not affect the output channel while it is in the OFF state. The Inhibit mode setting is stored in non-volatile memory.

The following modes can be selected:

- **LATChing** Causes a logic-true transition on the Inhibit input to disable all outputs. The outputs remain disabled until the Inhibit input is returned to logic-false and the latched INH status bit is cleared by sending the OUTP:PROT:CLE command or a protection clear command from the front panel.

- **LIVE** Allows the enabled outputs to follow the state of the Inhibit input. When the Inhibit input is true, the outputs are disabled. When the Inhibit input is false, the outputs are re-enabled.

- **OFF** The Inhibit input is ignored.
OUTPut:PON:STATE {RST|RCL0}
OUTPut:PON:STATE?

This command determines if the power-on state is set to the *RST (RST) state or the instrument state stored in memory location 0 (RCL0). The parameter is saved in non-volatile memory. Instrument states can be stored using the *SAV command.

Refer to *RST and *RCL under “System Commands” for more information.

OUTPut:PROTection:CLEAR (@<chanlist>)

This command clears the latched protection status that disables the output when an over-voltage, over-temperature, over-current, power-fail, or Inhibit status condition is detected. All conditions that generate the fault must be removed before the latched status can be cleared. The output is restored to the state it was in before the fault condition occurred.

NOTE

If a protection shutdown occurs during an output list, the list continues running even though the output is disabled. When the protection status is cleared and the output becomes enabled again, the output will be set to the values of the step that the list is presently at.

OUTPut:PROTection:COUPle {ON|OFF}
OUTPut:PROTection:COUPle?

This command enables/disables output coupling for protection faults. When enabled, ALL output channels are disabled when a protection fault occurs on any output channel. The enabled state is On (1); the disabled state is Off (0). When disabled, only the affected output channel is disabled when a protection fault is triggered.

The *RST value = OFF.

OUTPut:PROTection:DELay {<delay>|MIN|MAX}, (@<chanlist>)
OUTPut:PROTection:DELay? (@<chanlist>)

This command sets the over-current protection programming delay. This prevents momentary changes in status that can occur during reprogramming from triggering the over-current protection function. Programmed values can range from 0 to 255 milliseconds.

The *RST value = 20 ms.
Source Subsystem

The Source subsystem programs the current, digital, list, step, and voltage functions.

**NOTE**
The SOURce:CURRent:RANge, SOURce:VOLTage:RANge, and SOURce:LIST commands do not apply to all models (Refer to Chapter 1, “Model Differences”).

[SOURce:]CURRent[:LEVel][:IMMediate][:AMPLitude]{<value>|MIN|MAX}, (@<chanlist>)
[SOURce:]CURRent[:LEVel][:IMMediate][:AMPLitude]? (@<chanlist>)
[SOURce:]CURRent[:LEVel]:TRIGgered[:AMPLitude]{<value>|MIN|MAX}, (@<chanlist>)
[SOURce:]CURRent[:LEVel]:TRIGgered[:AMPLitude]? (@<chanlist>)

These commands set the immediate and the triggered current level of the output channel. The values are programmed in amperes. The immediate level is the output current setting. The triggered level is a stored value that is transferred to the output when a Step transient is triggered. This command is coupled with [SOURce:]CURRent:RANGe.

The *RST value = MIN.

[SOURce:]CURRent:MODE{FIXed|STEP|LIST}, (@<chanlist>)
[SOURce:]CURRent:MODE? (@<chanlist>)

These commands determine what happens to the output current when the transient system is initiated and triggered.

**FIXed**
The output voltage remains at the immediate value.

**STEP**
The output goes to the triggered level when a trigger occurs.

**LIST**
The output follows the programmed step value when a trigger occurs.

This function does not apply to all models (see Chapter 1, “Model Differences”).

The *RST value = FIXed.

[SOURce:]CURRent:PROTection:STATe{ON|OFF}, (@<chanlist>)
[SOURce:]CURRent:PROTection:STATe? (@<chanlist>)

This command enables or disables the over-current protection (OCP) function. The enabled state is On (1); the disabled state is Off (0). If the over-current protection function is enabled and the output goes into constant current operation, the output is disabled and the Questionable Condition status register OCP bit is set.

The current limit setting determines when the output channel goes into constant current operation. An over-current condition can be cleared with OUTPut:PROTection:CLEar after the cause of the condition is removed.

The *RST value = OFF.
This command only applies to models that have programmable ranges. Refer to Appendix A for the available ranges for each model.

This command sets the output current range. Units are in amperes. The instrument selects the range with the best resolution for the value entered. When queried, the returned value is the maximum DC current that can be output on the range that is presently set.

This command is coupled with the \texttt{[SOURce:]CURRent} command. This means that if a range command is sent that places an output on a range with a lower maximum current than the present current level, an error is generated. This also occurs if a current level is programmed with a value too large for the present range.

These types of errors can be avoided by sending the both level and range commands in the same SCPI message. When the range and setting information is received as a set, no range/setting conflict occurs.

The *RST value = the highest available range.

\textbf{NOTE}

If programming a range value causes a range change to occur while the output is enabled, the output will be temporarily disabled while the range switch occurs. The transition from on-to-off and then from off-to-on will also be delayed by the settings of OUTPut:DELay:FALL and OUTPut:DELay:RISE.

\texttt{[SOURce:]DIGital:INPut:DATA?}

This query reads the state of the digital control port. The query returns the state of pins 1 through 7 in bits 0 through 6 respectively.

\texttt{[SOURce:]DIGital:OUTPut:DATA \textless \texttt{value} \textgreater}

\texttt{[SOURce:]DIGital:OUTPut:DATA?}

This command sets the output data on the digital control port when that port is configured for Digital I/O operation. The port has seven signal pins and a digital ground pin. In the binary-weighted value that is written to the port, the pins are controlled according to the following bit assignments.

\begin{tabular}{|c|c|c|c|}
\hline
Pin & Bit & Pin & Bit \\
\hline
1 & 0 & 4 & 3 \\
2 & 1 & 5 & 4 \\
3 & 2 & 6 & 5 \\
7 & 6 &  &  \\
\hline
\end{tabular}

The query returns the last programmed value of the bits. To read the actual state of the pin, use \texttt{[SOURce:]DIGital:INPut:DATA?}
These commands set the functions of the digital port pins. The pin functions are saved in non-volatile memory.

**DIO** The pin is a general-purpose ground-referenced digital input/output. The output can be set with [SOURCE::DIGital:OUTPut:DATA <value>].

**DINPut** The pin is in digital input-only mode. The digital output data of the corresponding pin is ignored.

**TOUTput** The pin is configured as a trigger output. When configured as a trigger output, the pin will only generate output triggers if the Step or List transient system has been configured to generated trigger signals.

**TINPut** The pin is configured as a trigger input. When configured as a trigger input, the pin can be selected as a source of measurement and transient trigger signals.

**FAULt** Applies only to pin 1. Setting FAULt means that pin 1 functions as an isolated fault output. The fault signal is true when any output is in a protected state (from OCP, OVP, OT, PF, or INH). Note also that Pin 2 serves as the isolated common for pin 1. When pin 1 is set to the FAULt function, the instrument ignores any commands to program pin 2. Queries of pin 2 will return FAULt. If pin 1 is changed from FAULt to another function, pin 2 is set to DINPut.

**INHibit** Applies only to pin 3. When pin 3 is configured as an inhibit input, a true signal at the pin will disable all output channels.

These commands set the polarity of the digital port pins. The pin polarities are saved in non-volatile memory.

Setting a polarity to POSitive means that a logical true signal is a voltage high at the pin. Setting the polarity NEGative means that a logical true signal is a voltage low at the pin. For trigger inputs and outputs, POSitive means a rising edge; NEGative means a falling edge.
This command sets the number of times that the list is executed before it is completed. Applies only to models with list capability. The range is 1 through 256. Use INFINity to execute a list indefinitely. In this case, use ABORt:TRAnsient to stop the list.

The *RST value = 1.

This command specifies the current setting for each list step in amperes. Applies only to models with list capability. A comma-delimited list of up to 512 steps may be programmed.

The *RST value = 1 step with a value of MIN.

This query returns the number of points (steps) programmed in the current list. Applies only to models with list capability.

This command specifies the dwell time for each list step. Applies only to models with list capability. A comma-delimited list of up to 512 steps may be programmed. Dwell time is the time that the output will remain at a specific step. Dwell times can be programmed from 0 to 262.143 seconds with the following resolution:

<table>
<thead>
<tr>
<th>Range in seconds</th>
<th>Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 0.262143</td>
<td>1 microsecond</td>
</tr>
<tr>
<td>0 to 2.62143</td>
<td>10 microseconds</td>
</tr>
<tr>
<td>0 to 26.2143</td>
<td>100 microseconds</td>
</tr>
<tr>
<td>0 to 262.143</td>
<td>1 millisecond</td>
</tr>
</tbody>
</table>

At the end of the dwell time, the output state of the unit depends upon the LIST:STEP program settings. See LIST:STEP.

The order in which the values are entered determines the sequence when the list executes.

The *RST value = 1 step with a value of 0.001.

This query returns the number of points (steps) in the dwell list. Applies only to models with list capability.
[SOURce:]LIST:STEP {ONCE|AUTO}, (@<chanlist>)
[SOURce:]LIST:STEP? (@<chanlist>)

This command specifies how the list responds to triggers. Applies only to models with list capability.

**ONCE** Causes the output to remain at the present step until a trigger advances it to the next step. Triggers that arrive during the dwell time are ignored.

**AUTO** Causes the output to automatically advance to each step, after the receipt of an initial starting trigger. The steps are paced by the dwell list. As each dwell time elapses, the next step is immediately output.

The *RST value = AUTO.

[SOURce:]LIST:TERMinate:LAST {ON|OFF}, (@<chanlist>)
[SOURce:]LIST:TERMinate:LAST? (@<chanlist>)

This command determines the output value when the list terminates. Applies only to models with list capability. The state is either ON (1) or OFF (0). When ON, the output voltage or current remains at the value of the last list step. The value of the last voltage or current list step becomes the IMMediate value when the list completes. When OFF, and also when the list is aborted, the output returns to the settings it was at before the list started.

The *RST value = OFF.

[SOURce:]LIST:TOUTput:BOSTep[:DATA] {ON|OFF}{,{ON|OFF}}, (@<chanlist>)
[SOURce:]LIST:TOUTput:BOSTep[:DATA]? (@<chanlist>)

This command specifies which list steps generate a trigger out signal at the beginning of the list step (BOSTep). Applies only to models with list capability. A comma-delimited list of up to 512 steps may be programmed. The state is either ON (1) or OFF (0). A trigger is only generated when the state is set to ON.

The *RST value = 1 step with a value of OFF.

[SOURce:]LIST:TOUTput:EOSTep[:DATA] {ON|OFF}{,{ON|OFF}}, (@<chanlist>)
[SOURce:]LIST:TOUTput:EOSTep[:DATA]? (@<chanlist>)

This command specifies which list steps generate a trigger out signal at the end of the list step’s (EOSTep) dwell time. Applies only to models with list capability. A comma-delimited list of up to 512 steps may be programmed. The state is either ON (1) or OFF (0). A trigger is only generated when the state is set to ON.

The *RST value = 1 step with a value of OFF.

[SOURce:]LIST:VOLTage[:LEVel] <volt> {,<volt>}, (@<list>)
[SOURce:]LIST:VOLTage[:LEVel]? (@<chanlist>)

This command specifies the voltage setting for each list step in volts. Applies only to models with list capability. Up to 512 steps may be programmed. The values are separated by commas.

The *RST value = 1 step with a value of MIN.
[SOURce:]LIST:VOLTage:POINts? (@<chanlist>)

This query returns the number of points (steps) in the voltage list, not the point values. Applies only to models with list capability.

[SOURce:]STEP:TOUTput {ON|OFF}, (@<chanlist>)
[SOURce:]STEP:TOUTput? (@<chanlist>)

This command specifies whether an output trigger signal is generated when a transient voltage or current step occurs. The state is either ON (1) or OFF (0). A trigger is generated when the state is True.

The *RST value = OFF.

[SOURce:]VOLTage[:LEVel][:IMMediate][:AMPLitude] {<value>|MIN|MAX}, (@<chanlist>)
[SOURce:]VOLTage[:LEVel][:IMMediate][:AMPLitude]? (@<chanlist>)
[SOURce:]VOLTage[:LEVel]:TRIGgered[:AMPLitude] {<value>|MIN|MAX}, (@<chanlist>)
[SOURce:]VOLTage[:LEVel]:TRIGgered[:AMPLitude]? (@<chanlist>)

These commands set the immediate and the triggered voltage level of the output channel. The values are programmed in volts. The immediate level is the output voltage setting. The triggered level is a stored value that is transferred to the output when a Step transient is triggered. This command is coupled with [SOURce:]VOLTage:RANGe.

The *RST value = MIN.

[SOURce:]VOLTage:MODE {FIXed|STEP|LIST}, (@<chanlist>)
[SOURce:]VOLTage:MODE? (@<chanlist>)

These commands determine what happens to the output voltage when the transient system is initiated and triggered.

**FIXed**  The output voltage remains at the immediate value.

**STEP**  The output goes to the triggered level when a trigger occurs.

**LIST**  The output follows the programmed list step value when a trigger occurs.

This function does not apply to all models (see Chapter 1, “Model Differences”).

The *RST value = FIXed.

[SOURce:]VOLTage:PROTection:LEVel <value>|MIN|MAX}, (@<chanlist>)
[SOURce:]VOLTage:PROTection:LEVel? (@<chanlist>)

This command sets the over-voltage protection (OVP) level of the output channel. The values are programmed in volts. If the output voltage exceeds the OVP level, the output is disabled and the Questionable Condition status register OV bit is set. An over-voltage condition can be cleared with the Output Protection Clear command after the condition that caused the OVP trip is removed.

The *RST value = MAX.
[SOURce:]VOLTage:RANGe {<value>|MIN|MAX}, (@<chanlist>)
[SOURce:]VOLTage:RANGe? (@<chanlist>)

This command only applies to models that have programmable ranges. Refer to Appendix A for the available ranges for each model.

This command sets the output voltage range. Units are in volts. The instrument selects the range with the best resolution for the value that is entered. When queried, the returned value is the maximum voltage that can be output on the range that is presently set.

This command is coupled with the [SOURce:]VOLTage command. This means that if a range command is sent that places an output on range with a lower maximum voltage than the present voltage level, an error is generated. This also occurs if a voltage level is programmed with a value too large for the present range.

These types of errors can be avoided by sending the both level and range commands in the same SCPI message. When the range and setting information is received as a set, no range/setting conflict occurs.

The *RST value = the highest available range.

NOTE

If programming a range value causes a range change to occur while the output is enabled, the output will be temporarily disabled while the range switch occurs. The transition from on-to-off and then from off-to-on will also be delayed by the settings of OUTPut:DELay:FALL and OUTPut:DELay:RISE.

[SOURce:]VOLTage:SLEW[:IMMediate] {<value>|MIN|MAX|INF}, (@<chanlist>)
[SOURce:]VOLTage:SLEW[:IMMediate]? (@<chanlist>)

This command sets the voltage slew rate in volts per second. The slew rate setting affects all programmed voltage changes, including those due to the output state turning on or off. The slew rate can be set to any value between 0 and 9.9E37. For very large values, the slew rate will be limited by the analog performance of the output circuit. The keywords MAXimum or INFinity set the slew rate to maximum.

Internally, the slew rate is controlled by a 24-bit register. The slowest or minimum slew rate is a function of the full-scale voltage range. For a model with a 50 V range, the minimum slew rate is about 4.76 V/s. For other voltage ranges the minimum slew rate is proportional to this value, so for a model with a 5 V range the minimum slew rate is about 0.476 V/s. The unit accepts slew rates as low as 0 V/s, but values sent to the 24-bit register will be limited at 1 count.

The query returns the value that was sent, unless the value was less than the minimum slew rate, in which case the minimum value is returned. The LSB weight of the 24-bit register can be queried using VOLT:SLEW? MIN. The exact value varies slightly according to the voltage calibration.

The *RST value = 9.9E37.
Status Subsystem

Status register programming lets you determine the operating condition of the power system at any time. The power system has three groups of status registers; Operation, Questionable, and Standard Event. The Operation and Questionable status groups each consist of the Condition, Enable, and Event registers as well as NTR and PTR filters.

The Standard Event status group is also programmed using Common commands. Common commands control additional status functions such as the Service Request Enable and the Status Byte registers.

**Operation Status Group**

The Operation Status registers record signals that occur during normal operation. As shown below, the group consists of a Condition, PTR/NTR, Event, and Enable register. The outputs of the Operation Status register group are logically-ORed into the OPERation summary bit (7) of the Status Byte register.

**Questionable Status Group**

The Questionable Status registers record signals that indicate abnormal operation. As shown below, the group consists of the same register types as the Status Operation group. The outputs of the Questionable Status group are logically-ORed into the QUEstionable summary bit (3) of the Status Byte register.

**Standard Event Status Group**

The Standard Event registers are programmed by Common commands. The Standard Event event register latches events relating to communication status. It is a read-only register that is cleared when read. The Standard Event enable register functions similarly to the enable registers of the Operation and Questionable status groups.

**Status Byte Register**

This register summarizes the information from all other status groups as defined in the *IEEE 488.2 Standard Digital Interface for Programmable Instrumentation*.

**MSS and RQS Bits**

MSS is a real-time (unlatched) summary of all Status Byte register bits that are enabled by the Service Request Enable register. MSS is set whenever the power system has one or more reasons for requesting service. *STB? reads the MSS in bit position 6 of the response but does not clear any of the bits in the Status Byte register.

The RQS bit is a latched version of the MSS bit. Whenever the power system requests service, it sets the SRQ interrupt line true and latches RQS into bit 6 of the Status Byte register. When the controller
does a serial poll, RQS is cleared inside the register and returned in bit position 6 of the response. The remaining bits of the Status Byte register are not disturbed.

**MAV Bit and Output Queue**

The Output Queue is a first-in, first-out (FIFO) data register that stores power system-to-controller messages until the controller reads them. Whenever the queue holds one or more bytes, it sets the MAV bit (4) of the Status Byte register.
**STATus:PRESet**

This command sets all defined bits in the Status system’s PTR registers and clears all bits in the NTR and Enable registers.

<table>
<thead>
<tr>
<th>Operation Register</th>
<th>Questionable Register</th>
<th>Preset Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>STAT:OPER:ENAB</td>
<td>STAT:QUES:ENAB</td>
<td>0  all bits disabled</td>
</tr>
<tr>
<td>STAT:OPER:NTR</td>
<td>STAT:QUES:NTR</td>
<td>0  all bits disabled</td>
</tr>
<tr>
<td>STAT:OPER:PTR</td>
<td>STAT:QUES:PTR</td>
<td>31 all defined bits enabled</td>
</tr>
</tbody>
</table>

**STATus:OPERation[:EVENT]? (@<chanlist>)**

This query returns the value of the Operation Event register. The Event register is a read-only register, which stores (latches) all events that are passed by the Operation NTR and/or PTR filter. Reading the Operation Event register clears it. The bit configuration of the Operation status registers is as follows:

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>15-5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Value</td>
<td>–</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Bit Name</td>
<td>–</td>
<td>WTG-tran</td>
<td>WTG-meas</td>
<td>OFF</td>
<td>CC</td>
<td>CV</td>
</tr>
</tbody>
</table>

WTG-tran = The transient system is waiting for a trigger.
WTG-meas = The measurement system is waiting for a trigger
OFF = The output is programmed off
CC = The output is in constant current
CV = The output is in constant voltage

**STATus:OPERation:CONDition? (@<chanlist>)**

This query returns the value of the Operation Condition register. That is a read-only register, which holds the live (unlatched) operational status of the power system.

**STATus:OPERation:ENABLE <value>, (@<chanlist>)**

This command and its query set and read the value of the Operational Enable register. This register is a mask for enabling specific bits from the Operation Event register to set the operation summary bit (OPER) of the Status Byte register. This bit (bit 7) is the logical OR of all the Operational Event register bits that are enabled by the Status Operation Enable register.
STATus:OPERation:NTRansition <value>, (@<chanlist>)
STATus:OPERation:PTRansition <value>, (@<chanlist>)
STATus:OPERation:NTRansition? (@<chanlist>)
STATus:OPERation:PTRansition? (@<chanlist>)

These commands set and read the value of the Operation NTR (Negative-Transition) and PTR (Positive-Transition) registers. These registers serve as polarity filters between the Operation Condition and Operation Event registers to cause the following actions:

- When a bit in the Operation NTR register is set to 1, then a 1-to-0 transition of the corresponding bit in the Operation Condition register causes that bit in the Operation Event register to be set.
- When a bit of the Operation PTR register is set to 1, then a 0-to-1 transition of the corresponding bit in the Operation Condition register causes that bit in the Operation Event register to be set.
- If the same bits in both NTR and PTR registers are set to 1, then any transition of that bit at the Operation Condition register sets the corresponding bit in the Operation Event register.
- If the same bits in both NTR and PTR registers are set to 0, then no transition of that bit at the Operation Condition register can set the corresponding bit in the Operation Event register.

STATus:QUEStionable[:EVENt]? (@<chanlist>)

This query returns the value of the Questionable Event register. The Event register is a read-only register, which stores (latches) all events that are passed by the Questionable NTR and/or PTR filter. Reading the Questionable Event register clears it. The bit configuration of the Questionable status registers is as follows:

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>15-12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8-6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Value</td>
<td>–</td>
<td>2048</td>
<td>1024</td>
<td>512</td>
<td>–</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Bit Name</td>
<td>–</td>
<td>PROT</td>
<td>UNR</td>
<td>INH</td>
<td>–</td>
<td>CP–</td>
<td>OT</td>
<td>CP+</td>
<td>PF</td>
<td>OC</td>
<td>OV</td>
</tr>
</tbody>
</table>

PROT = The output has been disabled because it is coupled to a protection condition that occurred on another channel.
UNR = The output is unregulated
INH = The output is inhibited by an external signal
CP– = The output is limited by the negative power limit

OT = The over-temperature protection has tripped
CP+ = The output is limited by the positive power limit
PF = The output is disabled by the power-fail - which may be caused by a low-line or brownout condition on the AC line
OC = The output is disabled by the over-current protection
OV = The output is disabled by the over-voltage protection

STATus:QUEStionable:CONDition? (@<chanlist>)

This query returns the value of the Questionable Condition register. That is a read-only register, which holds the real-time (unlatched) questionable status of the power system.
STATus:QUEStionable:ENABle <value>, (@<chanlist>)
STATus:QUEStionable:ENABle? (@<chanlist>)

This command and its query set and read the value of the Questionable Enable register. This register is a mask for enabling specific bits from the Questionable Event register to set the questionable summary bit (QUES) of the Status Byte register. This bit (bit 3) is the logical OR of all the Questionable Event register bits that are enabled by the Questionable Status Enable register.

STATus:QUEStionable:NTRansiton <value>, (@<chanlist>)
STATus:QUEStionable:NTRansiton? (@<chanlist>)

STATus:QUEStionable:NTRansiton <value>, (@<chanlist>)
STATus:QUEStionable:NTRansiton? (@<chanlist>)

STATus:QUEStionable:PTRansiton <value>, (@<chanlist>)
STATus:QUEStionable:PTRansiton? (@<chanlist>)

These commands set or read the value of the Questionable NTR (Negative-Transition) and PTR (Positive-Transition) registers. These registers serve as polarity filters between the Questionable Condition and Questionable Event registers to cause the following actions:

- When a bit of the Questionable NTR register is set to 1, then a 1-to-0 transition of the corresponding bit of the Questionable Condition register causes that bit in the Questionable Event register to be set.

- When a bit of the Questionable PTR register is set to 1, then a 0-to-1 transition of the corresponding bit in the Questionable Condition register causes that bit in the Questionable Event register to be set.

- If the same bits in both NTR and PTR registers are set to 1, then any transition of that bit at the Questionable Condition register sets the corresponding bit in the Questionable Event register.

- If the same bits in both NTR and PTR registers are set to 0, then no transition of that bit at the Questionable Condition register can set the corresponding bit in the Questionable Event register.

*CLS

This command causes the following actions on the status system:

- Clears the Standard Event Status, Operation Status Event, and Questionable Status Event registers
- Clears the Status Byte and the Error Queue
- If *CLS immediately follows a program message terminator (<NL>), then the output queue and the MAV bit are also cleared.
*ESE

This command programs the Standard Event Status Enable register bits. The programming determines which events of the Standard Event Status Event register (see *ESR?) are allowed to set the ESB (Event Summary Bit) of the Status Byte register. A "1" in the bit position enables the corresponding event.

All of the enabled events of the Standard Event Status Event Register are logically ORed to cause the Event Summary Bit (ESB) of the Status Byte Register to be set. The query reads the Standard Event Status Enable register. The bit configuration of the Standard Event register is as follows:

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Value</td>
<td>128</td>
<td>-</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Bit Name</td>
<td>PON</td>
<td>-</td>
<td>CME</td>
<td>EXE</td>
<td>DDE</td>
<td>QUE</td>
<td>-</td>
<td>OPC</td>
</tr>
</tbody>
</table>

PON = Power-on has occurred
CME = Command error
EXE = Execution error
DDE = Device-dependent error
QUE = Query error
OPC = Operation complete

*ESR?

This query reads the Standard Event Status Event register. Reading the register clears it. The bit configuration is the same as the Standard Event Status Enable register (see *ESE).

*OPC

The command is mainly used for program synchronization. It causes the instrument to set the OPC bit (bit 0) of the Standard Event Status register when the instrument has completed all pending operations sent before the *OPC command. Pending operations are complete when:

- All commands sent before *OPC, including paralleled commands, have been completed. Most commands are sequential and are completed before the next command is executed. Commands that affect output voltage, current, or state, relays, and trigger actions are executed in parallel with subsequent commands. *OPC provides notification that all parallel commands have completed.

- All triggered actions are completed

*OPC does not prevent processing of subsequent commands, but the OPC bit will not be set until all pending operations are completed.

*OPC? causes the instrument to place an ASCII "1" in the Output Queue when all pending operations are completed. *OPC? does not suspend processing of commands.
**SRE**

This command sets the value of the Service Request Enable Register. This register determines which bits from the Status Byte Register are summed to set the Master Status Summary (MSS) bit and the Request for Service (RQS) summary bit. A 1 in any Service Request Enable Register bit position enables the corresponding Status Byte Register bit. All such enabled bits are then logically ORed to cause the MSS bit (bit 6) of the Status Byte Register to be set.

When the controller conducts a serial poll in response to SRQ, the RQS bit is cleared, but the MSS bit is not. When *SRE is cleared (by programming it with 0), the power system cannot generate an SRQ to the controller. The query returns the current state of *SRE.

**STB?**

This query reads the Status Byte register, which contains the status summary bits and the Output Queue MAV bit. Reading the Status Byte register does not clear it. The input summary bits are cleared when the appropriate event registers are read. The MAV bit is cleared at power-on, by *CLS, or when there is no more response data available.

A serial poll also returns the value of the Status Byte register, except that bit 6 returns Request for Service (RQS) instead of Master Status Summary (MSS). A serial poll clears RQS, but not MSS. When MSS is set, it indicates that the power system has one or more reasons for requesting service.

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1 – 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Value</td>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>–</td>
</tr>
<tr>
<td>Bit Name</td>
<td>OPER</td>
<td>MSS</td>
<td>ESB</td>
<td>MAV</td>
<td>QUES</td>
<td>ERR</td>
<td>–</td>
</tr>
</tbody>
</table>

OPER = Operation status summary  
MSS = Master status summary  
(RQS) = Request for service  
ESB = Event status byte summary  
MAV = Message available  
QUES = Questionable status summary  
ERR = Error queue not empty

**WAI**

This command instructs the power system not to process any further commands until all pending operations are completed. Pending operations are as defined under the *OPC command. *WAI can be aborted only by sending the power system a Device Clear command.
System Commands

System commands control system functions that are not directly related to output control, measurement, or status functions. Common commands are also used to control system functions.

**SYSTem:CHANnel[:COUNT]?**

This query returns the number of output channels in a mainframe.

**SYSTem:CHANnel:MODEL? (@<chanlist>)**

This query returns the model numbers of the selected output channels. Model numbers are comma-delimited.

**SYSTem:CHANnel:OPTION? (@<chanlist>)**

This query returns a list of options installed in each channel given in the channel list. The list of options for each channel is surrounded by double quotes. If there are no options installed in a channel, an empty pair of double quotes is returned.

**SYSTem:CHANnel:SERial? (@<chanlist>)**

This query returns the serial numbers of the selected output channels. Serial numbers are comma-delimited.

**SYSTem:COMMunicate:RLSTate {LOCal|REMoTe|RWLock}**

**SYSTem:COMMunicate:RLSTate? (@<chanlist>)**

This command configures the remote/local state of the instrument according to the following settings.

- **LOCal** The instrument is set to front panel and remote interface control.
- **REMoTe** The instrument is set to front panel and remote interface control.
- **RWLock** The front panel keys are disabled. The instrument can only be controlled via the remote interface. This programmable setting is completely independent from the front panel lock/unlock function that is available from the front panel menu.

The remote/local state can also be set by interface commands over the GPIB and some other I/O interfaces. When multiple remote programming interfaces are active, the interface with the most recently changed remote/local state determines the instrument’s remote/local state.

The remote/local state is unaffected by *RST or any SCPI commands other than SYSTem:COMMunicate:RLSTate. At power-on, the state is LOCal.

**SYSTem:COMMunicate:TCPiP:CONTrol?**

This query returns the control connection port number. This is used to open a control socket connection to the instrument.
**System Commands**

**SYSTem:ERRor?**

This query returns the next error number and its corresponding message string from the error queue. The queue is a FIFO (first-in, first-out) buffer that stores errors as they occur. As it is read, each error is removed from the queue. When all errors have been read, the query returns 0, “No error”. If more errors are accumulated than the queue can hold, the last error in the queue will be -350, “Too many errors” (see Appendix D for error codes).

**SYSTem:GROup:CATalog?**

This query returns information about channels that are grouped. The defined groups are enclosed in quotes. For example, the returned string “1,2,3”, “4” indicates that channels 1, 2, and 3 are grouped. Channel 4 is not grouped, as it appears by itself in the quote string.

**SYSTem:GROup:DEFine (@<chanlist>)**

This command defines a list of output channels as a paralleled group. This effectively creates a single output with higher current and power capability. You can group up to four channels per mainframe. Channels must have identical model numbers and options installed.

After the channels are wired in parallel and defined as a group, they can be addressed using any of the channel-specific SCPI commands by sending the channel number of the lowest channel in the group.

Group channel lists are stored in non-volatile memory and are unaffected by *RST or *RCL. But the group channel settings (voltage, current, etc.) are set and saved by *RST or *RCL.

This command deletes any previously saved states. However, for the group changes to take effect, you must also reboot the unit. Either cycle AC power or send the SYSTem:REBoot command.

**SYSTem:GROup:DELete <channel>**

This command removes the indicated channel from a group. It leaves the other channels in the group intact. When ungrouping a channel, you must also remove the parallel connections between the output and sense terminals of that channel.

This command deletes any previously saved states. However, for the group changes to take effect, you must also reboot the unit. Either cycle AC power or send the SYSTem:REBoot command.

**SYSTem:GROup:DELete:ALL**

This command restores a group of channels that have been grouped back to an ungrouped state. When ungrouping channels, you must also remove all paralleled connections between channels.

This command deletes any previously saved states. However, for the group changes to take effect, you must also reboot the unit. Either cycle AC power or send the SYSTem:REBoot command.
**SYSTem:PASSword:FPANel:RESet**

This command resets the front panel lockout password to the factory-shipped setting, which is zero (0). This command does not reset the calibration password.

**NOTE**

The front panel password can also be reset to 0 by setting an internal switch on the unit. This switch will also reset the calibration password to 0. Refer to Appendix B under “Calibration Switches” for more information.

**SYSTem:REBoot**

This command causes the instrument to reboot to its power-on state.

**SYSTem:VERSion?**

This query returns the SCPI version number to which the instrument complies. The returned value is of the form YYYY.V, where YYYY represents the year and V is the revision number for that year.

***IDN?**

This query requests the power system to identify itself. It returns a string of four fields separated by commas.

<table>
<thead>
<tr>
<th>Field Information</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Agilent Technologies</strong></td>
</tr>
<tr>
<td><strong>N67xxA</strong></td>
</tr>
<tr>
<td><strong>0</strong></td>
</tr>
<tr>
<td><strong>&lt;A&gt;.xx.xx</strong></td>
</tr>
</tbody>
</table>

***OPT?**

This query requests the mainframe to identify any installed options. A 0 indicates no options are installed.

***RCL <state>**

This command restores the power system to a state that was previously stored in memory locations 0 through 1 with the *SAV command. All instrument states are recalled except for the following:

- The trigger system is set to the Idle state by an implied ABORt command (this cancels any uncompleted trigger actions).
- Calibration is disabled by setting CALibration:STATe to OFF.
- All list settings are set to their *RST values.
- All status registers are set to their PRESet values.

**NOTE**

The device state stored in location 0 is automatically recalled at power turn-on when the Output Power-On state is set to RCL 0.
**System Commands**

**RDT?**

This query returns a description of the output channels installed in a mainframe. Semicolons separate multiple channel descriptions.

<table>
<thead>
<tr>
<th>Field</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAN &lt;c&gt;</td>
<td>Channel number</td>
</tr>
<tr>
<td>description</td>
<td>Description of the output channel</td>
</tr>
</tbody>
</table>

**RST**

This command resets the volatile memory of the power system to a factory-defined state (see “**RST Settings” at the beginning of this chapter).

*RST also forces the ABORt:ACQuire and ABORt:TRANsient commands. This cancels any measurement or output trigger actions presently in process, and resets the two WTG bits in the Status Operation Condition register.

**SAV <state>**

**CAUTION**

This command causes a write cycle to nonvolatile memory. Nonvolatile memory has a finite maximum number of write cycles. Programs that repeatedly cause write cycles to nonvolatile memory can eventually exceed the maximum number of write cycles and cause the memory to fail.

This command stores the present state of the power system to the specified location in non-volatile memory. Up to 2 states can be stored - in locations 0 and 1. Any state previously stored in the same location will be overwritten. Use the “RCL command to retrieve instrument states. Refer to “**RST Settings” at the beginning of this chapter for a list of instrument settings that can be saved.

If a particular state is desired at power-on, it should be stored in location 0. It will then be automatically recalled at power turn-on if the Output Power-On state is set to RCL0.

Note that list data (and the calibration state) is not saved as part of the *SAV operation. This means that all list data that is sent to the instrument will be lost when the power system is turned off.

Also, data saved in non-volatile memory, described in the Non-volatile Factory Settings table at the end of chapter 3, is not affected by the *SAV command.

**TST?**

This query causes the power system to do a self-test and report any errors. A 0 indicates the power system passed self-test. A 1 indicates one or more tests failed. Selftest errors are written to the error queue (see Appendix D). Note that *TST? also forces an *RST command.
Trigger Subsystem

The Trigger subsystem consists of the Abort, Initiate, and Trigger commands.

**Abort commands** cancel any triggered actions.

**Initiate commands** initialize the trigger system. This enables the trigger system to receive triggers.

**Trigger commands** control the remote triggering of the power system. They specify the trigger source for the transient and the measurement system and also generate software triggers.

`ABORt:ACQuire (@<chanlist>)`

`ABORt:TRANsient (@<chanlist>)`

These commands cancel any measurement or transient trigger actions presently in process. The two WTG bits in the Status Operation Condition register are also reset. These commands are executed at power-on and upon execution of `*RST`.

`INITiate[:IMMediate]:ACQuire (@<chanlist>)`

`INITiate[:IMMediate]:TRANsient (@<chanlist>)`

These commands control the enabling of both measurement and transient triggers. When a trigger is initiated, an event on a selected trigger source causes the specified triggering action to occur. If the trigger system is not initiated, all triggers are ignored.

`INITiate:CONTinuous:TRANsient {ON|OFF}, (@<chanlist>)`

`INITiate:CONTinuous:TRANsient? (@<chanlist>)`

This command continuously initiates the output trigger system. The enabled state is ON (1); the disabled state is OFF (0). With continuous triggering disabled, the output trigger system must be initiated for each trigger using the INITiate:TRANsient command.

The `*RST` value = OFF.

`TRIGger:ACQuire[:IMMediate] (@<chanlist>)`

This command sends an immediate trigger to the measurement system. When the trigger system is initiated, a measurement trigger causes the power system to measure the output voltage or current and store the results in a buffer. The measured quantity, voltage or current is specified by the SENSe:FUNCtion command.

`TRIGger:ACQuire:SOURce {BUS|PIN<pin>|TRANsient<chan>}, (@<chanlist>)`

`TRIGger:ACQuire:SOURce? (@<chanlist>)`

This command selects the trigger source for the measurement trigger system. The following trigger sources can be selected:
**TRIGGER SUBSYSTEM**

**BUS**
GPIB device trigger, *TRG, or <GET> (Group Execute Trigger).

**PIN<pin>**
Selects an output port connector pin. Pins 1 – 3 can be configured as external trigger sources. The [SOURCE:DIGital:PIN<n>:FUNCTION] command programs the function of each pin. The [SOURCE:DIGital:PIN<n>:POLARITY] command programs the polarity of each pin.

**TRANsient<chan>**
Selects the transient system of one of the output channels as the external trigger source. The following commands are used to generate triggers from the transient system: [SOURCE:STEP:OUTPUT], [SOURCE:LIST:OUTPUT:BOStep], and [SOURCE:LIST:OUTPUT:EOSTep].

The *RST value = BUS.

**TRIGGER:TRANsient[:IMMediate] (@<chanlist>)**

This command generates an immediate transient trigger regardless of the selected trigger source. Output triggers affect the following functions: voltage, current, and current limit. You must initiate the output trigger system before you can send any triggers.

When sent, the output trigger will:

- Initiate an output change as specified by the Current Triggered or Voltage Triggered commands.
- Clears the WTG-tran bit in the Status Operation Condition register after the transient trigger sequence has completed.

**TRIGGER:TRANsient:SOURce {BUS|PIN<pin>|TRANsient<chan>}, (@<chanlist>)**

This command selects the trigger source for the output trigger system. The following trigger sources can be selected:

**BUS**
GPIB device trigger, *TRG, or <GET> (Group Execute Trigger).

**PIN<pin>**
Selects an output port connector pin. Pins 1 – 3 can be configured as external trigger sources. The [SOURCE:DIGital:PIN<n>:FUNCTION] command programs the function of each pin. The [SOURCE:DIGital:PIN<n>:POLARITY] command programs the polarity of each pin.

**TRANsient<chan>**
Selects the transient system of one of the output channels as the trigger source. The following commands are used to generate triggers from the transient system: [SOURCE:STEP:OUTPUT], [SOURCE:LIST:OUTPUT:BOStep], and [SOURCE:LIST:OUTPUT:EOSTep].

The *RST value = BUS.

***TRG**

This command generates a trigger when the trigger subsystem has BUS selected as its source. The command has the same affect as the Group Execute Trigger (<GET>) command.
This chapter contains several example programs to help you develop programs for your own application. The example programs are for illustration only, and are provided with the assumption that you are familiar with the programming language being demonstrated and the tools used to create and debug procedures. See Chapter 6, “Language Dictionary” for the SCPI command syntax.

You have a royalty-free right to use, modify, reproduce and distribute the example programs (and/or any modified version) in any way you find useful, provided you agree that Agilent Technologies has no warranty, obligations, or liability for any example programs.

The example programs are in Microsoft Visual BASIC 6.0 using the VISA COM IO library. You must first load the VISA COM library to use these examples. The VISA COM IO library is available with version M or later of the Agilent IO libraries for Windows.

Before using the example code in Visual BASIC, you must reference two VISA COM objects. In Visual BASIC, go to Projects>References and select Agilent VISA COM Resource Manager 1.0 (filename = AgtRM.dll) and VISA COM 1.0 Type Library (filename = VisaCom.tlb). To use this sample code in Visual Basic .NET, see the VISA COM documentation to reference VISA COM in a Visual BASIC project. Copy the code provided in this chapter and call the subroutine for each example.

Microsoft, Visual BASIC, and Windows are U.S. registered trademarks of Microsoft Corporation.
Output Programming Example

This is a simple program that sets a voltage, current, over-voltage, and the status of over-current protection. When done, the program checks for instrument error and gives a message if there is an error.

```vba
Sub main_List()
    Dim IDN As String
    Dim GPIBaddress As String
    Dim ErrString As String

    ' This variable controls the channel number to be programmed
    Dim channel As String

    ' This variable controls the voltage
    Dim VoltSetting As Double

    ' This variable measures the voltage
    Dim MeasureVoltString As String

    ' This variable controls the current
    Dim CurrSetting As Double

    ' This variable controls the over voltage protection setting
    Dim overVoltSetting As Double

    ' These variables are necessary to initialize the VISA COM.
    Dim ioMgr As AgilentRMLib.SRMCls
    Dim Instrument As VisaComLib.FormattedIO488

    ' The following command line provides the program with the VISA name of the
    ' interface that it will communicate with. It is currently set to use GPIB.
    GPIBaddress = "GPIB0::5::INSTR"

    ' Use the following line instead for LAN communication
    ' TCPIPaddress="TCPIP0::141.25.36.214"

    ' Use the following line instead for USB communication
    ' USBaddress = "USB0::2391::1799::USB000000002"

    ' Initialize the VISA COM communication
    Set ioMgr = New AgilentRMLib.SRMCls
    Set Instrument = New VisaComLib.FormattedIO488
    Set Instrument.IO = ioMgr.Open(GPIBaddress)

    ' The next three command lines set the voltage, current, and over voltage
    VoltSetting = 3
    CurrSetting = 1.5 ' amps
    overVoltSetting = 10

    ' This variable can be changed to program any channel in the mainframe
    channel = "(@1)" ' channel 1

    With Instrument
        ' Send a power reset to the instrument
        .WriteString "*RST"

        ' Query the instrument for the IDN string
        .WriteString "*IDN?"
        IDN = .ReadString
    End With
End Sub
```
' Set the voltage
    .WriteString "VOLT" & Str$(VoltSetting) & "," & channel

' Set the over voltage level
    .WriteString "VOLT:PROT:LEV " & Str$(overVoltSetting) & "," & channel

' Set current level
    .WriteString "CURR " & Str$(CurrSetting) & "," & channel

' Turn on over current protection
    .WriteString "CURR:PROT:STAT ON," & channel

' Turn the output on
    .WriteString "OUTP ON," & channel

' Measure the voltage
    .WriteString "MEAS:VOLT? " & channel
    MeasureVoltString = .ReadString
    MsgBox "Measured Voltage is " & MeasureVoltString & " At channel" & channel

' Check instrument for any errors
    .WriteString "Syst:err?"
    ErrString = .ReadString

' give message if there is an error
    If Val(ErrString) Then
        MsgBox "Error in instrument!" & vbCrLf & ErrString
    End If

End With
End Sub

List Programming Example

This program executes a 10 point current and voltage list. It also specifies 10 different dwell times. When done, the program checks for instrument error and gives a message if there is an error.

Sub main_List()
    Dim IDN As String
    Dim GPIBaddress As String
    Dim ErrString As String
    Dim channel As String

    'These variable are necessary to initialize the VISA COM.
    Dim ioMgr As AgilentRMLib.SRMCls
    Dim Instrument As VisaComLib.FormattedIO488

    ' The following command line provides the program with the VISA name of the
    ' interface that it will communicate with. It is currently set to use GPIB.
    GPIBaddress = "GPIB1::5::INSTR"

    ' Use the following line instead for LAN communication
    ' TCPIPaddress="TCP/IP0::141.25.36.214"

    ' Use the following line instead for USB communication
    ' USBaddress = "USB0::2391::1799::US00000002"
' Initialize the VISA COM communication
Set ioMgr = New AgilentRMLib.SRMCls
Set Instrument = New VisaComLib.FormattedIO488
Set Instrument.IO = ioMgr.Open(GPIBaddress)

' These next three strings are the points in the list.
' All three strings are the same length.
' The first one controls voltage, the second current, and the third dwell time
Const voltPoints = "1,2,3,4,5,6,7,8,9,10"
Const currPoints = "0.5,1,1.5,2,2.5,3,3.5,4,4.5,5"
Const dwellPoints = "1,2,0.5,1,0.25,1.5,0.1,1,0.75,1.2"

' This variable can be changed to program any channel in the mainframe
channel = "(@1)"                                        ' channel 1

With Instrument
    ' Send a power reset to the instrument
    .WriteString "*RST"
    
    ' Query the instrument for the IDN string
    .WriteString "*IDN?"
    IDN = .ReadString
    
    ' Set the voltage mode to list
    .WriteString "VOLT:MODE LIST," & channel
    
    ' Set the current mode to list
    .WriteString "CURR:MODE LIST," & channel
    
    ' Send the voltage list points
    .WriteString "LIST:VOLT " & voltPoints & "," & channel
    
    ' Send the current list points
    .WriteString "LIST:CURR " & currPoints & "," & channel
    
    ' Send the dwell points
    .WriteString "LIST:DWEL " & dwellPoints & "," & channel
    
    ' Turn the output on
    .WriteString "OUTP ON," & channel
    
    ' Set the trigger source to bus
    .WriteString "TRIG:TRAN:SOUR BUS," & channel
    
    ' Initiate the transient system
    .WriteString "INIT:TRAN " & channel
    
    ' Trigger the unit
    .WriteString "**TRG"
    
    ' Check instrument for any errors
    .WriteString "Syst:err?"
    ErrString = .ReadString
    
    ' give message if there is an error
    If Val(ErrString) Then
        MsgBox "Error in instrument!" & vbCrLf & ErrString
    End If
End With

End Sub
Digitizer Programming Example

This program uses the voltage in step mode and also demonstrates how to set up and use the digitizer. When done, the program checks for instrument error and gives a message if there is an error.

Sub main_List()
Dim IDN As String
Dim GPIBaddress As String
Dim ErrString As String
Dim channel As String
Dim measPoints As Long
Dim measOffset As Long
Dim VoltSetting As Double
Dim finalVoltage As Double
Dim VoltPoints() As Variant
Dim i As Long

' These variables are necessary to initialize the VISA COM.
Dim ioMgr As AgilentRMLib.SRMCls
Dim Instrument As VisaComLib.FormattedIO488

' The following command line provides the program with the VISA name of the interface that it will communicate with. It is currently set to use GPIB.
GPIBaddress = "GPIB0::5::INSTR"

' Use the following line instead for LAN communication
' TCPIPaddress="TCPIP0::141.25.36.214"

' Use the following line instead for USB communication
' USBaddress = "USB0::2391::1799::US000000002"

' Initialize the VISA COM communication
Set ioMgr = New AgilentRMLib.SRMCls
Set Instrument = New VisaComLib.FormattedIO488
Set Instrument.IO = ioMgr.Open(GPIBaddress)

' This controls the number of points the measurement system measures
measPoints = 100

' This controls the number of points to offset the measurement (positive for forward, negative for reverse)
measOffset = 0

' This sets the time between points
timeInterval = 0.0025

' This controls the voltage
VoltSetting = 5

' This is the final voltage that will be triggered
finalVoltage = 10

' This variable can be changed to program any channel in the mainframe
channel = "(@1)" ' channel 1

With Instrument
  ' Send a power reset to the instrument
  .WriteString "*RST"
End With

End Sub
' Query the instrument for the IDN string
.WriteString "*IDN?"
IDN = .ReadString

' Put the Voltage into step mode which causes it to transition from one
' voltage to another upon receiving a trigger
.WriteString "VOLT:MODE STEP," & channel

' Set the voltage
.WriteString "VOLT" & Str$(VoltSetting) & "," & channel

' Go to final value
.WriteString "VOLT:TRIG" & Str$(finalVoltage) & "," & channel

' Turn the output on
.WriteString "OUTP ON," & channel

' Set the bus as the transient trigger source
.WriteString "TRIG:TRAN:SOUR BUS," & channel

' Set the number of points for the measurement system to use as an offset
.WriteString "SENS:SWE:OFFS:POIN" & Str$(measOffset) & ",," & channel

' Set the number of points that the measurement system uses
.WriteString "SENS:SWE:POIN" & Str$(measPoints) & ",," & channel

' Set the time interval between points
.WriteString "SENS:SWE:TINT" & Str$(timeInterval) & ",," & channel

' Set the measurement trigger source
.WriteString "TRIG:ACQ:SOUR BUS," & channel

' Initiate the measurement trigger system
.WriteString "INIT:ACQ " & channel

' Initiate the transient trigger system
.WriteString "INIT:TRAN " & channel

' Trigger the unit
.WriteString "*TRG"

' Read back the voltage points
.WriteString "FETC:ARR:VOLT? " & channel
VoltPoints = .ReadList

' Print the first 10 voltage points
For i = 0 To 9
    Debug.Print i, VoltPoints(i)
Next i

' Check instrument for any errors
.WriteString "Syst:err?"
ErrString = .ReadString

' give message if there is an error
If Val(ErrString) Then
    MsgBox "Error in instrument!" & vbCrLf & ErrString
End If